Restricting Network Access to System Daemons under SunOS

William LeFebvre

EECS Department
Northwestern University

Abstract

The implementation of most network daemons gives little consideration to the implications of worldwide access. In some cases, such access can permit the worldwide distribution of sensitive information, such as encrypted passwords. In other cases, local changes can be effected by processes running anywhere on the network. The shared library mechanism of SunOS can be used to provide a "wrapper" around certain daemons. This wrapper takes the form of an alternate `libc` shared library. Rather than linking against the standard `libc`, a daemon is directed to link against this alternate *secure* library. The secure library has an augmented form of certain network-related system calls which first perform the true system call then check the socket's peer against a configurable list of allowed hosts. If the peer is not found in the list, then the augmented call returns an indication of failure to the caller.

1 The Problem of Network Services

Local networking technology has provided a very powerful mechanism for the interaction of multiple machine. Services and information can be provided for machines on the local network via servers, or daemons, running on a select number of hosts. These services are vital to the operation of the local environment and their presence makes management of the machines significantly easier.

At the same time that local networking technology has exploded, so has wide area networking. As a result, most (if not all) of the daemons which intend to provide services for the local network inadvertently provide access to machines all around the world. These loopholes can be and have been exploited by unscrupulous individuals of malicious intent [5].

The most striking example of this problem is Sun's Network Information Service (NIS)[1]. This service provides a great deal of critical information for a UNIX system, including the data normally found in `/etc/passwd`, and especially including encrypted passwords. A list of known account names is an incredible benefit to an abuser, and common passwords can be easily discovered from their encrypted forms [4]. In all currently distributed forms, the program which provides this information, `ypserv`, will gladly give this information out to any machine that asks for it. Only the name of the NIS domain is needed.

Sites which now use Sun's "adjunct files" to protect encrypted passwords may have lured themselves into a false sense of security. All the data in the `passwd` map except the password

[1] In a previous life, NIS was known as Yellow Pages (YP).

is still available to anyone who can guess the domain name, including usernames and full names. Sites which, for the sake of convenience, provide NIS maps for the adjunct files via the map `passwd.adjunct` are putting themselves back in the same position they were in before using the adjunct files. Although it is true that `ypserv` will only answer queries for "protected" maps if the originator of the query is uid 0, no check is made against the host which originated the request. So uid 0 on any Internet host can still obtain the encrypted passwords.

Other examples exist as well. Even for a minimal level of security, some network services must be actively protected from access by non-local machines.

2 Possible Solutions

2.1 The Firewall

The most severe form of protection is a network barrier between the local organization and the rest of the world [1]. This barrier, usually called a *firewall*, is configured so that only packets for specific services are forwarded between local and global networks. Exactly which packets are forwarded is determined by the network administrator or his superiors. Typically it is limited to a very few protocols, including SMTP and NNTP but almost always excluding remote login and file transfer protocols (such as Telnet, FTP, and rexec).

A firewall insures the highest level of security short of removing the outside connection altogether [2]. But this security takes its toll in the form of added inconvenience for the legitimate users. Any remote logins which local users wish to make must be done it two hops: a login into the firewall, then a login from there to the remote host. Most organizations provide a mechanism to make this nearly transparent to the local users. Other activities, such as FTP, remain problematical.

Until global networks and network protocols reach a superior level of security, the firewall will remain the only choice for many organizations. Still, there are alternatives for those who are willing to sacrifice a small amount of security.

2.2 Secure RPC

RPC does not enforce any specific authentication scheme. Rather it uses open-ended authentication, allowing the applications to specify what type they require. Currently most RPC implementations provide only two forms of authentication: UNIX and DES. Those who are seriously concerned about the security of RPC communications may choose to use DES authentication, which actually encrypts the information in the RPC transaction [6, pp. 429–437].

2.3 Explicit Server Checking

The most obvious form of protection takes place in the server itself. When a server such as `ypserv` receives a request, it first checks the address of the originator to determine if it is a "trusted" host. It is the opinion of this author that all UNIX systems should provide this functionality in the form of library functions and that servers which provide sensitive information use such functions to protect themselves. Unfortunately, few vendors had the foresight to provide such functionality.

Very recently Sun began providing binaries which do this sort of checking [3] by releasing patch 100482–2. But this patch falls short in several ways:

- it is not yet part of any standard operating system distribution
- it only protects three binaries: `ypserv`, `ypxfrd`, `portmap`
- the configuration mechanism does not generalize well: it does not provide a mechanism to selectively protect services

A more generalized approach is needed, and it needs to be adopted, implemented, and used by *all* UNIX vendors.

2.4 The Wrapper

Since UNIX vendors didn't compile the protection in to their daemons and since they also usually don't give out source, two options remain. One is to find the source for a reimplementation of the network daemon and alter it. Another is to find a way to wrap protection around the server.

If a service's executable is invoked once per connection (for example, those handled by `inet`), it is possible to start a generalized "wrapper" program which will check and possibly log the connection before invoking the real executable. The package "TCP Wrapper" by Wietse Venema does this [8]. Unfortunately, this approach will not work for true daemons such as `ypserv` and `portmap`. A true daemon is started once and continues to run in the background forking off children to handle requests.

3 Kernel Wrapper via Shared Libraries

Starting with version 4.0, SunOS began providing a library sharing mechanism. Nearly all executables distributed with the system are linked against a shared C library. SunOS versions 4.1 and higher also provides the files necessary to rebuild the shared C library. With this functionality, it is possible to build special-purpose copies of the shared C library and to invoke standard executables with alternate libraries. This is sufficient to hook in to the servers and force them to do appropriate source verification.

3.1 Implementation

To understand the implementation, one must first understand a very fundamental fact about the UNIX C run-time library. All kernel calls [7] are implemented by a front end in the C library. Different computers will have different machine instructions for generating the protected trap required of kernel calls, and the front-end routines hide this detail from C programmers. The C run-time library—the same library which contains `printf` and `malloc`—also contains a front-end function for every kernel call. For example, the kernel call `write` actually exists as a function in the C library. This function is trivial: after possibly moving or rearranging the arguments, it merely executes the appropriate machine language "trap" instruction.

Since the front-end functions exist in the C library, the SunOS shared library mechanism allows a sufficiently clever individual to replace such a function, effectively adding functionality to any kernel call. This is what the secure library package uses to implement its security checks: every kernel call pertaining to network access has its front-end function replaced with one that verifies the address of the connecting host.

```
int retval;

retval = syscall(...);
if(retval ≥ 0)
{
    if(_ok_address(socket, addr, *addrlen))
    {
        return(retval);
    }
    errno = ECONNREFUSED;
    return(−1);
}
return(retval);
```

Figure 1: Basic Network Wrapper Algorithm

It turns out that only three kernel calls need such protection:

`accept`	accept a connection on a socket
`recvfrom`	receive a message from a connectionless socket
`recvmsg`	receive a message using a `struct msghdr`

Other kernel calls read data from the network, but only if the data is read from a connected socket. Only `accept` can generate file descriptors for connected sockets. Therefore, having `accept` verify the remote host is sufficient.

Figure 1 gives the basic algorithm for the secured "wrapper" functions. Each of the front-end functions listed above is replaced with a wrapper function which is patterned after this algorithm. The actual C code is listed in appendix A.

Each replaced function calls *_ok_address* for verification. It is this function that verifies the remote host address, returning *true* (1) if the remote host is acceptable and *false* (0) if it is not. It takes three arguments:

1. a file descriptor for the socket
2. a pointer to the socket address (a `struct sockaddr *`)
3. the length of the socket address

Each of these values is readily available to each wrapper, since they are passed as arguments (either directly or indirectly) to the corresponding kernel call.

The function *_ok_address* uses the socket address and length arguments if they make sense. However, if the socket address pointer is `NULL` or the length is not sufficient, then *_ok_address* will attempt to get the remote host's address by calling *getpeername* with the file descriptor (the first argument). If the socket is connectionless, then the call to *getpeername* will fail and *_ok_address* takes the attitude "better safe than sorry" by returning failure.

It is important to realize that the file descriptor is only used if the socket address pointer and length do not provide sufficient information. In all three cases (`accept`, `recvfrom`,

```
# Configuration file for securelib.
# <name>        <address>       <mask>
all             127.0.0.0       0.255.255.255
all             129.105.5.0     0.0.0.255
ypserv          129.105.2.0     0.0.0.255
```

Figure 2: Example Configuration File

`recvmsg`), the socket address values are taken from the arguments supplied by the caller. Therefore, a well-written program should not encounter any problems.

3.2 Configuration

The first implementation of *_ok_address* used a static table to determine if an address was acceptable. When the first version of this package was released, one of its users kindly sent the author a better version of *_ok_address* which reads its information from a file. Availability of source means that *_ok_address* can be changed to suit any particular needs that a given site may have.

The location of the configuration file is determined at compile time. By default, it is named `/etc/securelib.conf`. Some may wish to provide an additional level of security by placing the configuration file in a directory readable only by `root`, such as `/etc/security`. The advantage is that a regular user cannot determine which hosts are allowed to connect to which local servers. The disadvantage is that only processes run as `root` can use the secured library. In most environments, this is not an issue since all network servers run as root anyway.

The syntax of the configuration file is typical for UNIX. A hash mark (`#`) starts a comment which ends at the end of the line. Each line has three fields separated by white space:

1. the service name
2. the permissible address
3. the comparison mask

An example configuration file is given in figure 2. The function *_ok_address* maintains an internal copy of each applicable line from the file. It only considers a line "applicable" if the service name is "`all`" or if it matches the name of this process's service (the method used to determine that name is discussed in section 3.3). To verify a connection, *_ok_address* checks every applicable line as follows:

- the socket's Internet address is masked via a "bitwise and" of the one's complement of the specified mask (in retrospect, the configuration file should have specified a true subnet mask)
- the result is compared against the address specified in the configuration file
- success is indicated if and only if the result is true

```
LD_LIBRARY_PATH=/usr/lib/secure
export LD_LIBRARY_PATH
exec $@
```

Figure 3: Shell script to start secured programs

```
SECURE=""
if [ -x /usr/lib/secure/start ]; then
   echo 'Using network secure library where appropriate.'
   SECURE="/usr/lib/secure/start"
fi
```

Figure 4: Possible addition to `rc.local`

3.3 Use

After proper configuration, the `Makefile` distributed with the package (in conjunction with a few shell scripts) will perform all steps required to build a new shared C library. The library should then be installed in a location separate from `/usr/lib`. This library is *not* designed to replace the standard libc. Rather, it is intended to be used only in certain cases. The author chose to create a special directory for the task: `/usr/lib/secure`. Any process started with the environment variable `LD_LIBRARY_PATH` set to this directory will be dynamically linked against the secure C library instead of the standard one. Figure 3 gives an Bourne shell script which can be used to start "secured" daemons.

Normally, the name of a network service is determined *a priori* or by looking at the process's zeroth argument (`argv[0]`). The secure library cannot use either method for determining the service name. It must resort to either heuristics or sneaky tricks. The author of the configuration file code opted for the latter. Any process using the secure library is already being started with an altered environment, so requiring one more change to the environment was deemed acceptable. The function *_ok_address* uses the value of the environment variable `SL_NAME` to determine the name of the service. Only those lines in the configuration file which start with the same name or the name `all` will have significance.

The shell file presented in figure 3 is easily modified to accommodate this method by adding one line to the beginning:

```
SL_NAME=`basename $1`
```

The only other change required is the obvious one to the `export` command. This modified script is provided in the secure library package and is called `start`. The installation step places a copy in the same directory as the secure library itself.

Actual invocation of the `start` script will almost certainly be limited to `/etc/rc.local`. Those who wish to keep `rc.local` as adaptable as possible should make modifications as follows. Near the beginning of `rc.local` a check is made for the existence of `/usr/lib/secure` and an environment variable is set accordingly. The script fragment in figure 4 accomplishes this.

The lines in `rc.local` which invoke the daemons in need of protection are modified so that they start with `$SECURE`. If the library exists on this machine, the `start` script makes sure that each daemon is started with the appropriate environment. Otherwise, `$SECURE` expands to nothing and the daemon is started normally.

3.4 Limitations

This technique is not intended to solve all network security problems. It insures that servers have some control over the network location of clients who are requesting information. Network administrators must use every tool at their disposal to secure their systems. This is just another tool for the toolbox.

The most serious shortcoming is its reliance on peer information. The wrappers have no choice but to trust the information about the remote host which the kernel gives it. But this information is based solely on the data in the IP packet header—information that can be forged. The more common Internet problem of falsifying IP address to host name translations will not affect the secure library, since its checking is based solely on IP addresses.

Another limitation is time. It takes time to check even one packet. For most protocols, this extra overhead has little impact. But for heavily used stateless and connectionless protocols, such as NFS, the impact is very noticeable. This technique is not well suited to such applications. This is a very disappointing realization. It implies that an NFS daemon which does explicit checking for every request would be too slow for any practical purposes.

4 In the Absence of Shared Libraries

This technique was developed under SunOS specifically for a network of SunOS machines. It can easily be adapted to any operating system which supports and uses shared libraries, provided that there is a mechanism for rebuilding a shared C run-time library. Although implementation would certainly be difficult, the idea may be applicable to operating systems which do not support shared libraries.

An unstripped executable still contains the symbol table, which includes enough information to find the entry point for any external function in the program. This would include the front ends for kernel calls. One can conceive of a program that would alter the first instruction in a function with a jump to a new function added to the executable. Adding additional code is the difficult part: even an unstripped executable typically does not contain the relocation information, making it impossible to move any existing symbols. Ironically, application of virus writing technology would make it possible to add the necessary code to the executable.

Executables which have had the symbol table stripped pose an additional challenge. The only way to patch it would be to do some sort of disassembly. Prior knowledge of the program's structure would aid the disassembly process, and such knowledge can be gleaned from the freely available BSD network program sources. The vendor's executable may not be identical to the BSD programs, but similarities should still exist. Each network daemon has essentially the same structure: initialization followed by the main loop. Near the beginning of the main loop one would find a call to one of the three networking system calls: `accept`, `recvfrom` or `recvmsg`. Once this call is found, the location of the appropriate kernel front end function would be known and the technique used in the previous paragraph could be applied. It would be difficult—perhaps impossible—to automate this analysis.

5 Availability

The secured C library package is freely redistributable. It is available via anonymous FTP from `eecs.nwu.edu` in the directory `/pub/securelib`. At the time this paper was published, the Internet address for `eecs.nwu.edu` was 129.105.5.103.

6 Conclusions

Security is a very difficult problem. This package takes one step in the right direction by providing an extra level of checking. It prevents access to critical system services by clients outside a specified realm. It provides added functionality which should have been there all along, but it does so in a way that does not require source from the original operating system. The secure library can be installed and used on any stock Sun system provided these simple requirements are met: SunOS version 4.1, 4.1.1, or 4.1.2 and installation of the option `shlib_custom` (available on all distribution tapes, but not preinstalled by Sun).

7 Acknowledgements

The author would like to thank all the brave people who tried the first version of his package and to Northwestern University for giving him a sandbox to play in. He would especially like to thank Sam Horrocks of UCI for providing the code which reads the configuration file.

References

[1] William R. Cheswick. The design of a secure internet gateway. In *Proceedings of the Summer 1990 USENIX Conference*. USENIX Association, 1990.

[2] William R. Cheswick. An evening with berferd in which a cracker is lured, endured, and studied. In *Proceedings of the Winter 1992 USENIX Conference*, pages 163–174. USENIX Association, 1992.

[3] Computer Emergency Response Team. SunOS NIS vulnerability. CERT Advisory 92:13, June 4 1992.

[4] Daniel V. Klein. Foiling the cracker: A survey of, and improvements to, password security. In *UNIX Security Workshop II*, pages 5–14. USENIX Association, 1990.

[5] Eugene H. Spafford. The internet worm incident. Technical Report CSD-TR-933, Department of Computer Science, Purdue University, September 1991.

[6] Sun Microsystems. *Network and Communications Administration*, March 27 1990.

[7] *Unix Programmers Reference Manual*. Section 2.

[8] Wietse Venema. TCP wrapper, a tool for network monitoring, access control, and for setting up booby traps. In *Third UNIX Security Symposium*, 1992. To be published.

A Kernel Call Wrappers

This is the C function used in place of the kernel call `accept`.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/syscall.h>
#include <errno.h>

accept(s, addr, addrlen)

int s;
struct sockaddr *addr;
int *addrlen;

{
    register int retval;
    struct sockaddr sa;
    int salen;

    salen = sizeof(sa);
    if ((retval = syscall(SYS_accept, s, &sa, &salen)) >= 0)
    {
        if (_ok_address(retval, &sa, salen))
        {
            _addrcpy(addr, addrlen, &sa, salen);
            return (retval);
        }
        close(retval);
        errno = ECONNREFUSED;
        return (-1);
    }
    return (retval);
}
```

This is the C function used in place of the kernel call `recvfrom`.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/syscall.h>
#include <errno.h>

recvfrom(s, buf, len, flags, from, fromlen)

int s;
char *buf;
int len, flags;
struct sockaddr *from;
int *fromlen;

{
    register int retval;

    if ((retval = syscall(SYS_recvfrom, s, buf, len, flags,
                          from, fromlen)) >= 0)
    {
        if (_ok_address(s, from, *fromlen))
        {
            return (retval);
        }
        errno = ECONNREFUSED;
        return (-1);
    }
    return (retval);
}
```

This is the C function used in place of the kernel call `recvmsg`.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/syscall.h>
#include <errno.h>

recvmsg(s, msg, flags)

int s;
struct msghdr *msg;
int flags;

{
    register int retval;

    if ((retval = syscall(SYS_recvmsg, s, msg, flags)) >= 0)
    {
        if (_ok_address(s, (struct sockaddr *)(msg->msg_name),
                        msg->msg_namelen))
        {
            return (retval);
        }
        errno = ECONNREFUSED;
        return (-1);
    }
    return (retval);
}
```