Secure Portability

Damien Miller (djm@mindrot.org)

October 2005

Abstract

This paper introduces the issues of portability for C applications between Unix variants, including semantic differences in libraries and system calls, API support and reasonable minimum platform requirements. It also describes the approach used by Portable OpenSSH to the problems of secure portability and points to some areas where more work is needed by platform vendors.

1 Introduction

This paper introduces issues of portability for C applications on Unix, GNU/Linux and Windows/cygwin platforms, including semantic differences in libraries and system calls, API support and reasonable minimum platform requirements. It also describes the approach used by Portable OpenSSH [16] to the problems of secure portability. Finally it points to some areas where more work is needed by platform vendors.

Software running on modern Unix-like systems must deal with innumerable differences in libraries and in system behaviour. Significant differences are evident even between the various GNU/Linux distributions. This variance ranges from the trivial, such as differing filesystems layouts, though to the complex, such as platform-specific authentication methods or differences in system call semantics.

Coping with these differences adds complexity to applications, making them more difficult and much less enjoyable to develop and verify. Some of these differences have serious security implications and the additional complexity required to cope with them also increases the likelihood of security problems. Another confounding factor is that the best APIs (from a security perspective) do not have wide platform support, indeed some platform maintainers have actively militated against their adoption.

2 Approaching Portability

Some projects include portability as an original, explicit requirement or goal, but the majority of software packages are not written to be portable. Rather, they have portability incrementally added after the software has been developed on an original "golden" platform (usually either an explicit target platform, or whatever the developer likes or has available). This is not necessarily a bad thing - it provides a reference against which the correct operation of other platforms can be measured.

In the case of OpenSSH, the original platform was OpenBSD, though some portability code from the legacy ssh-1.2.x code-base was retained. OpenSSH differs from many other software projects in its separation into "core" and "portable" versions. The OpenBSD developers want a clean code-base, free of portability clutter as the canonical home for OpenSSH is their CVS tree. The portable version of OpenSSH is maintained by a semi-separate team of developers in a separate CVS tree. This arrangement creates some extra work, as changes to the core version must be periodically merged, but automated tools render this process trivial and the existence of the OpenBSD version has proved useful many times in determining whether bugs in the core product, the portability code or the new target platform. Table 1 lists the platforms that are supported by portable OpenSSH.

AIX	DragonflyBSD	QNX
Apple MacOS X	HP/UX 10.x, 11.x	SCO OpenServer 5
BSDi	Irix 5.x, 6.x	SCO Unixware
Cray Unicos	Linux	SNI ReliantUNIX
Cygwin	LynxOS	Solaris 2.6, 7, 8, 9, 10
DEC OSF/1	NCR SVR4 MP-RAS	Sony NewsOS BSD
DGUX	NeXTSTEP	SunOS 4
Darwin	NetBSD	Ultrix
FreeBSD	OpenBSD	

Table 1: Platforms supported by portable OpenSSH

Most software projects, however, maintain a single version that supports multiple platforms, usually with some leaning towards a favourite (e.g. most recent free software projects implicitly prefer GNU/Linux). In either case, portability becomes an issue when developers encounter a difference between platforms.

At this point, it is worthwhile to consider some goals of portability. The prime objective is to have the software carry out its desired function(s) on the desired platforms. However, there are several less obvious goals:

1. To retain readability of the code
2. To ensure that the software behaves similarly on different platforms (including avoiding the introduction of platform-specific bugs)
3. To facilitate the addition of support for new platforms
4. To minimise maintenance costs for developers of the software

5. To minimise support costs from users of the software

Achieving one of these goals should not involve trading off against the others. Indeed, focusing on the prime objective generally makes the others much easier to achieve, though it may require a little more up-front effort. In addition, porting software can expose hidden assumptions and bugs that may only occur rarely on the base platform, finding and fixing these issues improves software quality as a whole.

3 Platform differences

Differences between Unix and Unix-like platforms are far less painful today than they were a decade, or even five years ago. Beyond differing endianness and word sizes, hardware differences are largely transparent to the modern developer. Likewise C compilers are generally feature-compatible, with the major difference being the command line options required to compile and link a program. System libraries are largely standardised and tend to include popular functions, regardless of the lineage of the particular platform.

Spare a thought for the brave developers of twenty years ago who had to contend with differences at every level [9]: in the C compiler and tool-chain (still in a state of flux), in the network stack (changing as the TCP/IP protocols were refined), in an OS still in a stage of rapid evolution and in strange (by today's standards) features and limitations of the underlying hardware that were not abstracted away by the underlying OS.

However, portability issues remain; platforms are by no means homogenous. Deep differences, including subtle but critical differences in semantics exist between some systems. New APIs are being added frequently to both free and proprietary operating systems.

3.1 Trivial differences

Most platform related differences are trivial in nature. Table 2 mentions several of this nature and basic ways to deal with them. While these issues do not pose much of a problem to an aware developer, they can become more tricky to deal with through their composition.

3.1.1 Use of the C preprocessor

While the trivial differences are relatively easy to work around they can, because of the frequency of their occurrence, also be the ones that pose the greatest threat to readability of source code. Despite strong recommendations to the contrary [18], an all-too-common approach to fixing these differences is the liberal use of preprocessor directives to create an in-line replacement.

These inline replacements have a tendency to multiply, leading to a maze of platform-specific code wrapped up in deeply nested pre-processor logic. For example, listing 1 shows a section of

Problem	*Solution*
Differing system integer byte orders (endianness)	Use POSIX *ntohl*, *ntohs*, *htonl*, *htons* functions to convert
Different word sizes (e.g. of the *int* type)	Use width specified types, such as *int32_t* where word size matters
Missing type definitions (e.g. *u_int32_t*)	Include replacement definitions in header file
Missing functions (e.g. daemon, strlcpy)	Include portable replacements
Different underlying integer types for OS provided types, e.g. *uid_t*	Cast to wider type in printf, avoid direct use as array index or in pointer arithmetic (avoiding signed vs. unsigned bugs)

Table 2: Some common trivial platform differences

ntpd's [14] startup code where the simple action of becoming a daemon has been rendered nearly unreadable by a maze of compatibility fall-backs.

A far better approach for situations like this is to provide a replacement for the missing API (*daemon()* in this case) and to include it in a compatibility library. This clears the main code paths of inessential clutter, thereby making them far easier to read and follow. Removing the clutter is also conducive to better security by making it easier to audit for problems.

By locating the replacement function in a compatibility library, it will be available to every discrete program in the software distribution, obviating the need for each to implement its own replacement. This approach should be recursively applied to the compatibility functions too - merely refactoring the previous example into a *daemon()* function in a separate file doesn't produce the full benefit unless the whole tangle is unwound.

Of course, some preprocessor is almost always required - the goal is to avoid nesting it, which leads to an exponential growth in the number of paths through the code. As a rule, consider breaking code out whenever there is a need for more than one level of preprocessor nesting.

In many cases, an even more simple solution to the issue of replacing missing functions exists: import or adapt code from one of the BSD operating systems. The BSD code is well written, released under a liberal licence intended to facilitate exactly this type of reuse and is standard (for many functions it is the original and canonical implementation). Another major benefit is that this code is actively maintained, thus saving effort for application developers who integrate it. Portable OpenSSH makes extensive use of OpenBSD's standard library code to supplement missing or broken implementations of functions on other platforms.

This approach of *direct replacement* works well for platform functions, types and pre-processor defines that are missing, but it can be difficult to replace a platform supplied function that is broken. Prototypes for various functions differ between systems and, unless the application developer is willing to ship multiple prototypes for the replacement functions, it is impossible to make them work across all the necessary platforms. One such case is the RFC 3493 [6] address-family independent host and address lookup routines. Several platforms have shipped incomplete or buggy implementations of these routines that needed to be worked around, but direct replacements ran into the problems described above. The solution used by portable OpenSSH is to use the pre-processor to internally rename the functions to point to internal replacements (as shown in Listing 2). This approach is slightly ugly, in that it renames a system provided API. Another solution would be to introduce a wrapper API, but this would sacrifice some readability for the vast majority of systems where these functions work correctly.

3.1.2 Activating replacements

Once a replacement function has been written or imported, the developer is now faced with another question: how is this replacement triggered? There are several popular approaches to

```
#   ifdef HAVE_DAEMON
        daemon(0, 0);
#   else /* not HAVE_DAEMON */
        if (fork())    /* HMS: What about a -1? */
            exit(0);
        {
#if !defined(F_CLOSEM)
            u_long s;
            int max_fd;
#endif /* not F_CLOSEM */
#if defined(F_CLOSEM)
            /*
             * From 'Writing Reliable AIX Daemons,' SG24-4946-00,
             * by Eric Agar (saves us from doing 32767 system
             * calls)
             */
            if (fcntl(0, F_CLOSEM, 0) == -1)
                msyslog(LOG_ERR, "ntpd: failed to close open files(): %m");
#else  /* not F_CLOSEM */
# if defined(HAVE_SYSCONF) && defined(_SC_OPEN_MAX)
            max_fd = sysconf(_SC_OPEN_MAX);
# else /* HAVE_SYSCONF && _SC_OPEN_MAX */
            max_fd = getdtablesize();
# endif /* HAVE_SYSCONF && _SC_OPEN_MAX */
            for (s = 0; s < max_fd; s++)
                (void) close((int)s);
#endif /* not F_CLOSEM */
            (void) open("/", 0);
            (void) dup2(0, 1);
            (void) dup2(0, 2);
#if defined(HAVE_SETPGID) || defined(HAVE_SETSID)
# ifdef HAVE_SETSID
            if (setsid() == (pid_t)-1)
                msyslog(LOG_ERR, "ntpd: setsid(): %m");
# else
            if (setpgid(0, 0) == -1)
                msyslog(LOG_ERR, "ntpd: setpgid(): %m");
# endif
#else /* HAVE_SETPGID || HAVE_SETSID */
```

Listing 1: excerpt from ntp-stable-4.2.0a-20050303 ntpdmain() function

```
#ifndef HAVE_GETNAMEINFO
#define getnameinfo(a,b,c,d,e,f,g) (ssh_getnameinfo(a,b,c,d,e,f,g))
int getnameinfo(const struct sockaddr *, size_t, char *, size_t,
    char *, size_t, int);
#endif /* !HAVE_GETNAMEINFO */
```

Listing 2: avoiding a buggy system-provided function

this, each involving some tradeoffs.

The most simple way to trigger platform-specific replacements is to use pre-processor definitions set by the user. These usually appear either in a *Makefile* or some configuration header. While this is very easy for the developer, it can be confusing for non-technical users and therefore likely to increase the number of support requests if the software is shipped as source code, though grouping together coherent sets of options by platform can reduce this burden. Another problem is that it can be difficult to manually maintain the list of definitions as the software grows more complex. If there are only a handful of defines, then this may be a useful solution for very small software packages.

This method can be trivially automated using the pre-processor definitions set by the compiler or system include files. For example, `#if defined(__OpenBSD__)`. This improves over the previous technique in that it needs no end user adjustment for the common cases. It is also easy for the developer: understanding which sets of options are set on a platform simplifies debugging. Unfortunately this method tends to become unwieldy when many platforms are added. It also fails to detect variants of a single OS, e.g. differences between Linux distributions.

A better approach is to provide pre-configured sets of consistent options in the build infrastructure (e.g. *Makefiles*, automatically selected by system architecture, OS and/or the user. A good example of this is the *imake* [5] system used by X11R6 and its set of per-platform definitions files. Again, this is simple for the user, so long as they fall into the set of provided platforms and offers determinism for the developer. Supporting the software on a new platform, or variant of an existing one does take some developer time.

Perhaps the most common approach today is to automatically detect platform characteristics by running compile-time tests, à la GNU autoconf [11], though this approach predates autoconf by many years [18]. This is simple and automatic for the user, and the same system can provide a standard and user-friendly way of making other compile-time customisations, such as selecting installation paths. This approach also offers a reasonable chance that the software will work unmodified on new platforms or on variants of existing platforms, thereby reducing support requirements. The big problem with this method is that it makes it difficult for the developer to ascertain the exact configuration parameters selected on a given system, if they don't have direct access to it. This makes debugging quite a bit more difficult in these cases. Also, the most popular tool (GNU autoconf) is somewhat fragile and therefore can be a source of complexity in itself, though this is not an inherent problem of the approach.

Most free software projects use either of the last two methods, or a combination of both. Portable OpenSSH uses a combination, by way of GNU autoconf: compile-time tests where possible, with some per-platform definitions. The per-platform definitions are required because some things are difficult to test. It is impractical to test for bugs in the networking functions in a general sense (what happens if the user is not connected to a network when running the tests?) and some other feature tests would require root to function. In portable OpenSSH, per-platform defines are usually used to mark certain platform features as “broken” and to enable platform authentication code.

3.2 Complex differences

Complex differences can be frustrating for the developer. One of these is a collection of essentially trivial differences: the wild variation between platforms in how login records are maintained. Most systems maintain some form of *utmp* (logged in users, indexed by TTY), *wtmp* (record of login and logout events), *btmp* (record of failed login attempts) and *lastlog* (per-UID record of most recent login activity) files but the fields present in the files, their contents and the way that login applications are expected to write to them frequently differ between platforms. Some platforms have adopted the POSIX *utmpx* format and associated functions, but this support is not universal even among actively maintained operating systems.

OpenSSH introduced the *logicrec* API (contributed by Andre Lucas) to deal with this morass. *logicrec* presents a high-level API that hides the gory details of updating the appropriate files from the main application behind simple functions to record a login or logout event. To achieve this, the *loginrec.c* code needs to present a superset of the fields present in supported platforms *utmp* files. Compare figures 3 and 4. GNU/Linux possesses a fairly complete utmp structure that closely matches the *loginrec* API, whereas other platforms often omit one or more fields, often the IPv6 address. This API has greatly simplified the task of supporting new systems in portable OpenSSH and has subsequently been adopted by at least one other free software project [10].

```
struct logininfo {
   char       progname[LINFO_PROGSIZE]; /* name of program (for PAM) */
   int        progname_null;
   short int  type;                     /* type of login (LTYPE_*) */
   int        pid;                      /* PID of login process */
   int        uid;                      /* UID of this user */
   char       line[LINFO_LINESIZE];     /* tty/pty name */
   char       username[LINFO_NAMESIZE]; /* login username */
   char       hostname[LINFO_HOSTSIZE]; /* remote hostname */
   /* 'exit_status' structure components */
   int        exit;                     /* process exit status */
   int        termination;              /* process termination status */
   unsigned int tv_sec;
   unsigned int tv_usec;
   union login_netinfo hostaddr;        /* caller's host address(es) */
}; /* struct logininfo */
```

Listing 3: main loginrec.c structure

A related approach is used in portable OpenSSH’s audit system (designed and implemented by Darren Tucker). If the platform supports login event auditing, a simple bridge can be written

```
struct utmpx
{
  short int ut_type;            /* Type of login.  */
  __pid_t ut_pid;               /* Process ID of login process.  */
  char ut_line[__UT_LINESIZE];  /* Devicename.  */
  char ut_id[4];                /* Inittab ID. */
  char ut_user[__UT_NAMESIZE];  /* Username.  */
  char ut_host[__UT_HOSTSIZE];  /* Hostname for remote login.  */
  struct __exit_status ut_exit; /* Exit status of a process marked
                                   as DEAD_PROCESS.  */
  long int ut_session;          /* Session ID, used for windowing.  */
  struct timeval ut_tv;         /* Time entry was made.  */
  __int32_t ut_addr_v6[4];      /* Internet address of remote host.  */
  char __unused[20];            /* Reserved for future use.  */
};
```

Listing 4: Linux utmpx structure (abridged)

between its native API and portable OpenSSH's abstract audit event API. At present, BSM auditing is supported as used by Sun and Mac OS X/OpenBSM, but other schemes would be trivial to add. Similarly, the password authentication / encryption code has per-platform hooks for OS vendors who have decided that they should make things complicated for application developers by using a function other than *crypt()* to perform password encryption.

Some inter-platform differences are more subtle, an example of this is the differing semantics of signal delivery: whether system calls are restarted after delivery of a signal or whether they return with an EINTR and whether or not signal handlers are reinstalled after they are used. OpenSSH uses a wrapper function *mysignal()* to provide BSD-like semantics, following the technique presented by Stevens[19].

```
mysig_t
mysignal(int sig, mysig_t act)
{
    struct sigaction sa, osa;

    if (sigaction(sig, NULL, &osa) == -1)
        return (mysig_t) -1;
    if (osa.sa_handler != act) {
        memset(&sa, 0, sizeof(sa));
        sigemptyset(&sa.sa_mask);
        sa.sa_flags = 0;
#ifdef SA_INTERRUPT
        if (sig == SIGALRM)
            sa.sa_flags |= SA_INTERRUPT;
#endif
        sa.sa_handler = act;
        if (sigaction(sig, &sa, NULL) == -1)
            return (mysig_t) -1;
    }
    return (osa.sa_handler);
}
```

Listing 5: Replacement for signal() using POSIX sigaction()

Astute readers may note that the OpenSSH *mysignal()* function is not exactly like Stevens' in that it does not activate the restart of system calls after receipt of a signal. This is because setting the POSIX SA_RESTART flag is not sufficient to ensure that a system call will fully complete upon receipt of a signal, for instance a *read()* may return fewer bytes than were requested when it is interrupted. OpenSSH needs to be careful here, as a SIGCHLD or other signal could arrive at any time during its execution, and it cannot afford to assume that a *read()* or *write()* will continue to completion. Instead, OpenSSH deals with interrupted or short reads and writes using a wrapper function *atomicio()*. This function will try to read or write the specified number of bytes, restarting if the system call returns a short transfer or an EINTR error. *atomicio* will run until either all the requested bytes are moved or an error or EOF condition has occurred.

3.3 Differences with security implications

Most of the time inter-platform differences will cause obvious failures when they are not properly dealt with. However, some differences are subtle and have effects that can seriously impact security.

One example of this is the PAM library [17]. PAM provides a standard API for programs to perform user authentication, authorisation and session setup. However and ambiguity in the specification leads to a nasty bug. PAM is a challenge/response API, providing the application a set of *pam_message* structures which can instruct it to display messages or prompt for user input with character echo enabled (for non-sensitive questions) or disabled (e.g. for passwords). Developers of some Sun-derived PAM implementations interpreted this set of messages as being passed as a *pointer to an array of struct pam_message*, whereas the Linux-PAM [15] developers took the view that it is passed as *An array of pointers to struct pam_message*. In the common case where only a single message is present, the two are equivalent. However when multiple messages are present, an application expecting the wrong behaviour could read to or write from an incorrect address, a behaviour that is potentially exploitable by an attacker to gain control of the process. Worse, because it is responsible for authentication, the application code that deals with PAM must run with super-user privileges, which a successful exploit would gleefully inherit.

To tackle this, portable OpenSSH implemented the accessor macro shown in Listing 6 to hide the PAM implementation's *pam_message* passing convention.

```
#ifdef PAM_SUN_CODEBASE
# define PAM_MSG_MEMBER(msg, n, member) ((*(msg))[(n)].member)
#else
# define PAM_MSG_MEMBER(msg, n, member) ((msg)[(n)]->member)
#endif
```

Listing 6: Working around different PAM semantics

Another platform difference of concern is in the semantics of the *setuid* family of calls: *setuid*, *setreuid*, *setresuid*, and their group ID manipulation counterparts. Chen and Wagner [3] have found that several naive usage patterns of these system calls can lead to an incomplete revocation of privilege. Portable OpenSSH adopts their recommendations, and implements a somewhat paranoid approach when permanently discarding privilege:

1. Drop group privileges: *setgroups*, *setegid* and *setgid*
2. Drop user privileges: *seteuid* and *setuid*
3. Try to restore group privileges and raise a fatal error if successful
4. Try to restore user privileges and raise a fatal error if successful

Where possible, the *setresuid* and *setresgid* API is used in favour of separate calls to set the real and effective IDs. These functions offers the most unambiguous semantics and ensure that saved IDs are set correctly [3]. As a result, OpenBSD is replacing all uses of these older functions to permanently drop privileges with calls to *setresuid* and *setresgid* throughout its code-base.

4 Choosing the right API

A popular, but naive view of portability is that it consists of "avoiding unportable APIs". This may ease some of the more trivial portability problems encountered by developers, but it has the negative effect of dumbing software down to the lowest common denominator. Nearly all of the best APIs from a security perspective are incompletely portable. However the benefit for using these APIs in favour of more portable, but less secure ones greatly outweighs the cost of having to include portable replacements, especially when one considers that the better APIs are only going to become more common with time.

One concrete example of this is the *closefrom* system call (atomically close all open file descriptors numbered above a certain bound) - it was first introduced in Sun Solaris, but subsequently added to OpenBSD and then NetBSD. An application that used the lowest common denominator approach of manually closing file descriptors would never benefit from the improved API, even if it was introduced to the application's native platform.

A system interface that went the other way is the */dev/random* cryptographic random number device [20]; first implemented on Linux and the BSDs, but eventually added to Solaris. This is a kernel facility that provides a central pool of cryptographically unguessable random numbers, made available to user applications via the */dev/random* device node. Unguessable random numbers are critically important for cryptographic applications such as key generation and agreement. The provision of a central and strong API removed the temptation for application developers to "roll their own" random pooling and seeding code, often with insecure results [7]. The use of such kernel facilities where available, or a good user-level replacement (such as PRNGd [8]) is strongly recommended.

Sadly, sometimes better APIs aren't always universally adopted. For instance, the *strlcpy* and *strlcat* functions [13]. These are designed to replace *strcpy*, *strcat*, *strncpy* and *strncat*. These latter functions are standard POSIX, but suffer serious deficiencies: *strcpy* and *strcat* do not check the boundaries of the target buffer and therefore can easily overrun it if used without the highest degree of care (giving rise to the famous stack-smashing attack [2]). The bounds-checked *strncpy* and *strncat* are not much better; they fail to nul-terminate the target string if the source string is equal or longer in length than the target buffer (thereby opening a related class of security bugs) and they do not return enough information to allow the application developer to detect cases where a string truncation has occurred.

The *strlcpy* and *strlcat* API properly check the target buffer's bounds, nul-terminate in all cases and return the length of the source string, allowing detection of truncation. This API has been adopted by most modern operating systems and many standalone software packages, including OpenBSD (where it originated), Sun Solaris, FreeBSD, NetBSD, the Linux kernel, rsync and the GNOME project. The notable exception is the GNU standard C library, glibc [12], whose maintainer steadfastly refuses to include these improved APIs, labelling them "horribly inefficient BSD crap" [4], despite prior evidence that they are faster is most cases than the APIs they replace [13]. As a result, over 100 of the software packages present in the OpenBSD ports tree maintain their own *strlcpy* and/or *strlcat* replacements or equivalent APIs - not an ideal state of affairs.

Finding replacements for good APIs isn't hard - it is highly probable that a free software project has already solved the problem and has made a licence-compatible replacement available. Nor should the task result in a dramatic increase in a project's size, most of these APIs are trivial to replace. Table 3 shows the sizes of the largest replacement functions used in portable OpenSSH. In almost all of these cases, the code was obtained or adapted from OpenBSD's standard C library. A good long-term approach in this age of open source operating systems is to also contribute this support code to their libraries or kernels.

Function	Lines of code (inc. comments)
glob()	914
snprintf()	652
getrrsetbyname()	585
base64_ntop() and base64_pton()	324
getcwd()	242
vis()	239
inet_ntop()	229
getaddrinfo(), getnameinfo() and support functions	224
openpty()	201
realpath()	196

Table 3: largest replacement functions in Portable OpenSSH

4.1 Conclusion

The main recommendations of this paper may be briefly summarised into six simple rules:

1. Avoid cluttering main code paths with portability code, especially that wrapped up in nested preprocessor
2. Pick the best possible API available, even if it isn't available on every platform - it can be replaced or imported if it doesn't exist somewhere
3. Replace missing or broken functions in a separate library rather than performing the surgery inline
4. Where possible, obtain an existing, known-good replacement instead of developing new code (e.g. from OpenBSD's libc)
5. When dealing with areas of great platform variability, abstract the API back to a superset of the platforms' features
6. Be alert for subtle differences or bugs between platforms, and doubly so in areas of an application that wield privilege

This paper has detailed some common portability problems, ranging from the simple to the complex, and approaches to solve them. These approaches may not be optimum for every application, but they have served portable OpenSSH well in allowing it to function on over twenty platforms while retaining maintainability of the code-base.

References

[1] *IEEE Std 1003.1, 2004: IEEE standard portable operating system Interface for computer environments*, Institute of Electrical and Electronics Engineers, 2004

[2] Aleph One, *Smashing the stack for fun and profit*, Phrack Magazine, Vol. 7, Issue 49

[3] H. Chen, D. Wagner and D. Dean, *Setuid Demystified*, Proceedings of the 11^{th} USENIX Security Symposium

[4] U. Drepper, , post to libc-alphasources.redhat.com mailing list, http://sources.redhat.com/ml/libc-alpha/2000-08/msg00053.html, August 2000

[5] J. Fulton, *Configuration management in the X Window system*, Technical report, MIT Laboratory for Computer Science, 1989

[6] R. Gilligan,*Basic Socket Interface Extensions for IPv6*, RFC 3493, February 2003

[7] I. Goldberg, D. Wagner, *Randomness and the Netscape Browser*, Dr. Dobb's Journal, 1996

[8] L. Jänicke, *PRNGD - Pseudo Random Number Generator Daemon*, http://www.aet.tu-cottbus.de/personen/jaenicke/postfix_tls/prngd.html

[9] S. C. Johnson, D. M. Ritchie, *Portability of C Programs and the UNIX System*, The Bell System Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978

[10] M. Johnston, *Dropbear SSH server and client*, http://matt.ucc.asn.au/dropbear/dropbear.html

[11] D. MacKenzie, et. al., *GNU Autoconf*, http://www.gnu.org/software/autoconf/

[12] R. McGrath, et. al., *GNU C Library*, http://www.gnu.org/software/libc/libc.html

[13] T. Miller, T. de Raadt, *strlcpy and strlcat – Consistent, Safe, String Copy and Concatenation*, Proceedings of the 1999 USENIX Security Symposium

[14] D. Mills, et. al., *ntp-stable-4.2.0a-20050303 software distribution*, http://www.ntp.org/

[15] A. G. Morgan, *Linux-PAM*, http://www.kernel.org/pub/linux/libs/pam/

[16] OpenSSH project, *OpenSSH*, http://www.openssh.org/

[17] V. Samar, R. Schemers, *Unified login with pluggable authentication modules (PAM)*, Open Software Foundations, Request for comments 86.0, October 1995

[18] H. Spencer, *#ifdef Considered Harmful, or Portability Experience with C News*, Summer USENIX 1992

[19] W. Richard Stevens, *Advanced Programming in the UNIX Environment*, Addison Wesley Publishing Company, 1992, ISBN 0-201-56317-7

[20] T. Ts'o, *random.c – A strong random number generator*, Linux kernel, 1994