Secure Portability - Portable OpenSSH's approach

Damien Miller <djm@mindrot.org>
AUUG 2005, October 2005

Introduction

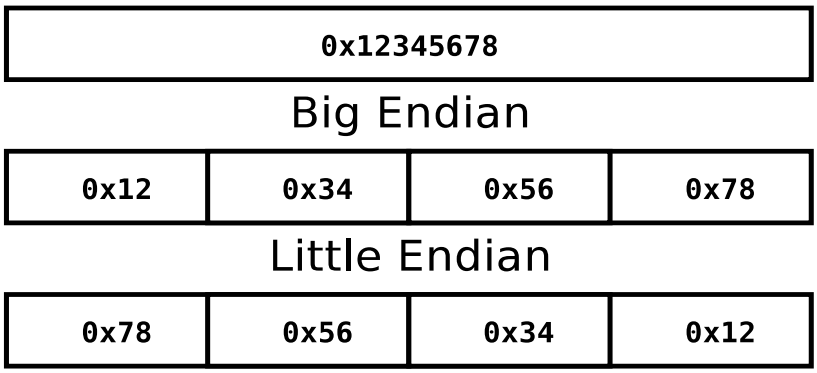
- Why port?
  - Greater audience
  - Porting exposes wrong assumptions and latent bugs
- What we will cover?
  - Platform differences: simple, complex and dangerous
  - Portable OpenSSH's approach to dealing with them
  - API choice
- Our focus:
  - C language with a reasonably modern Unix/POSIX API
  - Inclination towards free software
- OpenSSH
  - The most widely-used SSH implementation on the Internet
  - Primary version is maintained in OpenBSD's CVS tree
  - Portability team makes it run on 25+ platforms
  - This arrangement works surprisingly well

Simple differences - missing types and defines

- Types, such as int32_t, socklen_t, etc.
- Preprocessor symbols, e.g. IPTOS_THROUGHPUT, STDIN_FILENO
- Solution:
  - Provide your own replacements
- Be careful when defining things like PATH_MAX
  - You can introduce buffer overflows if you get it wrong
  - If you need to set it, then you have to provide functions that honour your limits

Simple differences - endianness

- Word order between processors



- Solution:
  - Use appropriate byte-swapping functions
    - htonl(), ntohl(), htons(), ntohs()
  - 4.2BSD byteorder(3) functions are more explicit:
    - htobe32(), betoh64(), htole16(), letoh64(), etc.
    - Not universally available

Simple differences - differing types

- Differing types in OS typedefs
  - E.g. width and signedness of gid_t
  - Beware of signed vs. unsigned issues here
- Solution:
  - Avoid use as array index or in pointer arithmetic
  - If you must, then sanitise first (ensure integers are >= 0)
  - Avoid annoying warnings in printf() conversions by casting up to a longer type (e.g. long)

Simple differences - Missing APIs

- Platforms often have missing APIs
- Perhaps the most common difference
- Some random examples from OpenSSH:

arc4random	getopt	prctl	setresuid
b64_ntop	getpeereid	pstat	setsid
b64_pton	_getpty	readpassphrase	setvbuf
bcopy	getrlimit	realpath	sigaction
bindresvport_sa	getttyent	recvmsg	snprintf
clock	glob	rresvport_af	socketpair
closefrom	inet_aton	sendmsg	strerror
dirfd	inet_ntoa	setdtablesize	strlcat
fchmod	inet_ntop	setegid	strlcpy
fchown	innetgr	setenv	strmode
freeaddrinfo	login_getcapbool	seteuid	strnvis
futimes	md5_crypt	setgroups	strtonum
getaddrinfo	memmove	setlogin	strtoll
getcwd	mkdtemp	setpcred	strtoul
getgrouplist	openlog_r	setproctitle	sysconf
getnameinfo	openpty	setrlimit	tcgetpgrp

Missing APIs (continued)

- Because these differences occur so frequently, they can cause the most confusion
- A common mistake is to try to replace the absent API in-line
- This requires in-line preprocessor, which has a nasty tendency to nest to the point of unreadability

Bad preprocessor: ntpd 4.2.0a-20050303

ntpd/main.c

```
451| #   ifdef HAVE_DAEMON
452|     daemon(0, 0);
453| #   else /* not HAVE_DAEMON */
454|     if (fork())    /* HMS: What about a -1? */
455|       exit(0);
456|
457|     {
458| #if !defined(F_CLOSEM)
459|       u_long s;
460|       int max_fd;
461| #endif /* not F_CLOSEM */
462|
463| #if defined(F_CLOSEM)
464|       /*
465|        * From 'Writing Reliable AIX Daemons,' SG24-4946-00,
466|        * by Eric Agar (saves us from doing 32767 system
467|        * calls)
468|        */
469|       if (fcntl(0, F_CLOSEM, 0) == -1)
470|         msyslog(LOG_ERR, "ntpd: failed to close open files(): %m");
471| #else  /* not F_CLOSEM */
472|
473| # if defined(HAVE_SYSCONF) && defined(_SC_OPEN_MAX)
474|       max_fd = sysconf(_SC_OPEN_MAX);
475| # else /* HAVE_SYSCONF && _SC_OPEN_MAX */
476|       max_fd = getdtablesize();
477| # endif /* HAVE_SYSCONF && _SC_OPEN_MAX */
478|       for (s = 0; s < max_fd; s++)
479|         (void) close((int)s);
480| #endif /* not F_CLOSEM */
481|       (void) open("/", 0);
482|       (void) dup2(0, 1);
483|       (void) dup2(0, 2);
484| #ifdef SYS_DOMAINOS
485|       {
486|         uid_$t puid;
487|         status_$t st;
489|         proc2_$who_am_i(&puid);
490|         proc2_$make_server(&puid, &st);
491|       }
492| #endif /* SYS_DOMAINOS */
493| #if defined(HAVE_SETPGID) || defined(HAVE_SETSID)
494| # ifdef HAVE_SETSID
495|       if (setsid() == (pid_t)-1)
496|         msyslog(LOG_ERR, "ntpd: setsid(): %m");
497| # else
498|       if (setpgid(0, 0) == -1)
499|         msyslog(LOG_ERR, "ntpd: setpgid(): %m");
500| # endif
501| #else /* HAVE_SETPGID || HAVE_SETSID */
502|       {
503| # if defined(TIOCNOTTY)
504|         int fid;
505|
506|         fid = open("/dev/tty", 2);
507|         if (fid >= 0)
508|         {
509|           (void) ioctl(fid, (u_long) TIOCNOTTY, (char *) 0);
510|           (void) close(fid);
511|         }
512| # endif /* defined(TIOCNOTTY) */
513| # ifdef HAVE_SETPGRP_0
514|         (void) setpgrp();
515| # else /* HAVE_SETPGRP_0 */
516|         (void) setpgrp(0, getpid());
517| # endif /* HAVE_SETPGRP_0 */
518|       }
519| #endif /* HAVE_SETPGID || HAVE_SETSID */
520| #ifdef _AIX
521|       /* Don't get killed by low-on-memory signal. */
522|       sa.sa_handler = catch_danger;
523|       sigemptyset(&sa.sa_mask);
524|       sa.sa_flags = SA_RESTART;
```

Missing APIs (continued)

- Each #ifdef is a separate code path
  - Exponential complexity for each level of nesting
  - Difficult to get test coverage of all paths (even compilation coverage can be hard)
  - Complex code == more bugs == more security bugs
- Better to replace the function in a separate file
  - Keeps main code clean of replacement junk
    - (you can see the point of the code, not portability goop)
  - Placing code in a library makes it available throughout a package (it will probably be needed more than once)
  - Can be independantly unit tested

Missing APIs (continued)

- In the previous example, the whole mess could be replaced with a single **daemon()** replacement
- Not enough just to move the mess to a different file
  - Improves readability, but does yield the other benefits
  - Instead, replace recursively
- Rather than writing your own replacement, borrow one from a free OS (e.g. OpenBSD)
  - Tested, audited and actively maintained
  - Not writing code saves you effort

Replacing broken APIs

- Direct replacement of missing APIs works well, but broken APIs are more difficult
- Differences in function prototypes between platform-supplied function and replacement can cause compilation failures
- Replacing broken platform-supplied functions is critical for secure code
  - Example: snprintf's return value
- OpenSSH example: RFC3493 replacements
  - getnameinfo(), getaddrinfo()
- To avoid clashes, we use an ugly (but useful) trick:
  - #define the name out of the way
  - Make sure you #define *after* inclusion of system headers

Fixing a broken API: OpenSSH 4.2p1

libopenbsd-compat/fakerfc2553.h

```
142| #ifndef HAVE_GETADDRINFO
143| #ifdef getaddrinfo
144| # undef getaddrinfo
145| #endif
146| #define getaddrinfo(a,b,c,d) (ssh_getaddrinfo(a,b,c,d))
147| int getaddrinfo(const char *, const char *,
148|     const struct addrinfo *, struct addrinfo **);
149| #endif /* !HAVE_GETADDRINFO */
150|
151| #if !defined(HAVE_GAI_STRERROR) && !defined(HAVE_CONST_GAI_STRERROR_PROTO)
152| #define gai_strerror(a)  (ssh_gai_strerror(a))
153| char *gai_strerror(int);
154| #endif /* !HAVE_GAI_STRERROR */
155|
156|  #ifndef HAVE_FREEADDRINFO
157|  #define freeaddrinfo(a)  (ssh_freeaddrinfo(a))
158|  void freeaddrinfo(struct addrinfo *);
159|  #endif /* !HAVE_FREEADDRINFO */
160|
161|  #ifndef HAVE_GETNAMEINFO
162|  #define getnameinfo(a,b,c,d,e,f,g) (ssh_getnameinfo(a,b,c,d,e,f,g))
163|  int getnameinfo(const struct sockaddr *, size_t, char *, size_t,
164|      char *, size_t, int);
165|  #endif /* !HAVE_GETNAMEINFO */
```

Activating replacements

- To actually use a replacement or work-around, the code needs to "know" when it is necessary
- Several approaches:
  - Manual definition of pre-processor defines (e.g. in Makefile)
  - Basing decisions on platform-supplied preprocessor symbols, such as __OpenBSD__
  - Maintaining per-platform sets of options in the build systems
  - Testing for platform features at build-time

Manual preprocessor

- The most simple approach
  - Very easy to apply and maintain
- Doesn’t "scale" well to lots of configuration options
- Confusing for non-technical users
- Has its uses:
  - Small projects with only one or two options
  - Technical or infrequently used options (e.g. debugging code)

Platform-supplied preprocessor

- Most C preprocessors symbols describing the OS
  - E.g. "sun", "linux", "__osf__", "__FreeBSD__", "__OpenBSD__"
- Becomes messy when lots of platforms involved
- Doesn't detect variants within a particular platform (OS distributions) or local changes
  - No hope for something like Gentoo Linux
- Little chance that the code will work on a never-before-seen platform
- Again, good when there are only few options and broad differences between platforms

Example: Jef Poskanzer's mini_httpd 1.19

port.h

```
 3| #if defined(__FreeBSD__)
 4| # define OS_FreeBSD
 5| # define ARCH "FreeBSD"
 6| #elif defined(__OpenBSD__)
 7| # define OS_OpenBSD
 8| # define ARCH "OpenBSD"
 9| #elif defined(__NetBSD__)
10| # define OS_NetBSD
11| # define ARCH "NetBSD"
12| #elif defined(linux)
13| # define OS_Linux
14| # define ARCH "Linux"
15| #elif defined(sun)
16| # define OS_Solaris
17| # define ARCH "Solaris"
18| #elif defined(__osf__)
19| # define OS_DigitalUnix
20| # define ARCH "DigitalUnix"
21| #elif defined(__svr4__)
22| # define OS_SysV
23| # define ARCH "SysV"
24| #else
25| # define OS_UNKNOWN
26| # define ARCH "UNKNOWN"
27| #endif
29| #ifdef OS_FreeBSD
30| # include <osreldate.h>
31| # define HAVE_DAEMON
32| # define HAVE_SETSID
33| # define HAVE_SETLOGIN
34| # define HAVE_WAITPID
35| # define HAVE_HSTRERROR
36| # define HAVE_TM_GMTOFF
37| # define HAVE_SENDFILE
38| # define HAVE_SCANDIR
39| # define HAVE_INT64T
40| # ifdef SO_ACCEPTFILTER
41| #  define HAVE_ACCEPT_FILTERS
42| #  if ( __FreeBSD_version >= 411000 )
43| # define ACCEPT_FILTER_NAME "httpready"
44| #  else
45| # define ACCEPT_FILTER_NAME "dataready"
46| #  endif
47| # endif /* SO_ACCEPTFILTER */
48| #endif /* OS_FreeBSD */
49|
50| #ifdef OS_OpenBSD
51| # define HAVE_DAEMON
52| # define HAVE_SETSID
```

Makefile-supplied preprocessor

- Define consistent sets of options per-platform in a Makefile
- Removes the compile-time customisation from the code to somewhere easier to maintain
- Good for developers: determinisim in which options are active
- Easy to set up for cross-compilation
- Again, it doesn't cope too well with...
  - OS variants and customisation
  - never-before-seen platforms

Example: Postfix 2.2.5

makedefs

```
 87| case "$SYSTEM.$RELEASE" in
 88|   SCO_SV.3.2) SYSTYPE=SCO5
 89|             # Use the native compiler by default
 90|             : ${CC="/usr/bin/cc -b elf"}
 91|             CCARGS="$CCARGS -DPIPES_CANT_FIONREAD $CCARGS"
 92|             SYSLIBS="-lsocket -ldbm"
 93|             RANLIB=echo
 94|             ;;
 95|  UnixWare.5*) SYSTYPE=UW7
 96|             # Use the native compiler by default
 97|             : ${CC=/usr/bin/cc}
 98|             RANLIB=echo
 99|             SYSLIBS="-lresolv -lsocket -lnsl"
100|              ;;
101|  UNIX_SV.4.2*)       case "`uname -v`" in
102|             2.1*) SYSTYPE=UW21
103|                 # Use the native compiler by default
104|                 : ${CC=/usr/bin/cc}
105|                 RANLIB=echo
106|                 SYSLIBS="-lresolv -lsocket -lnsl -lc -L/usr/ucblib -lucb"
107|                 ;;
108|             *) error "Seems to be UnixWare`uname -v`. Untested.";;
109|              esac
110|              ;;
```

Automatic compile-time detection

- Detect platform characteristics at the time of compilation
  - As implemented in GNU autoconf, though the technique is older
- Good for users:
  - Fairly easy to use interface
  - Good chance at working on a never-before-seen platform
  - Copes very well with customised platforms
  - Handy place for compile-time customisation (e.g. paths)
- Not so good for developers:
  - The most popular tool (autoconf) is a little fragile
  - Difficult to reconstruct which options were selected
- Probably the most popular method

Example: Portable OpenSSH cvs-20051008

configure.ac

```
2831| AC_CACHE_CHECK([for struct addrinfo], ac_cv_have_struct_addrinfo, [
2832|         AC_TRY_COMPILE(
2833|                 [
2834| #include <sys/types.h>
2835| #include <sys/socket.h>
2836| #include <netdb.h>
2837|                 ],
2838|                 [ struct addrinfo s; s.ai_flags = AI_PASSIVE; ],
2839|                 [ ac_cv_have_struct_addrinfo="yes" ],
2840|                 [ ac_cv_have_struct_addrinfo="no" ]
2841|         )
2842| ])
2843| if test "x$ac_cv_have_struct_addrinfo" = "xyes" ; then
2844|         AC_DEFINE(HAVE_STRUCT_ADDRINFO, 1,
2845|                 [define if you have struct addrinfo data type])
2846| fi
2847|
2848| AC_CACHE_CHECK([for struct timeval], ac_cv_have_struct_timeval, [
2849|         AC_TRY_COMPILE(
2850|                 [ #include <sys/time.h> ],
2851|                 [ struct timeval tv; tv.tv_sec = 1;],
2852|                 [ ac_cv_have_struct_timeval="yes" ],
2853|                 [ ac_cv_have_struct_timeval="no" ]
2854|         )
2855| ])
```

Example: Portable OpenSSH (continued)

- Actually uses a combination of platform-specified and compile-time detection
- Can't detect everything at compile-time
  - Broken network APIs
  - Subtle platform bugs
  - Tests that require root access
- So, quite a few things are set by the platform. Examples:
  - Use of pipes vs socketpair for stdio
  - Broken set user/group primitives
  - setproctitle() replacement method
  - Character sequence in /etc/passwd for locked account
- "make survey" (optional)
  - Collects the options that detected by autoconf
  - Send them back to the portability team

Beyond simple API replacement

- Some platform differences resist simple solutions like direct API replacement
- A common case of this is when platform export different APIs to do the same thing
- An example in OpenSSH is login/logout recording
- Most platforms support one or more of utmp(x), wtmp(x), lastlog and btmp
  - But they tend to pick different subsets of the possible fields
    - E.g. skipping IPv6 support
- There is a standard API (pututxline), but
  - It isn't widely supported
  - It doesn't do everything we need

Beyond simple API replacement (continued)

- To cope, we step back a little and offer an *abstract API* that provides a superset of platform's requirements
- The *loginrec* API in portable OpenSSH hides the detail of updating all these files behind four simple operations:
  - Record a login
  - Record a logout
  - Record a failed login attempt
  - Find the time of the user's last login
- This keeps the (already hairy) session code free of fiddly platform-specific details

Example: glibc-2.3.5

/usr/include/bits/utmpx.h (abridged)

```
43| struct utmpx
44| {
45|   short int ut_type;           /* Type of login.  */
46|   __pid_t ut_pid;              /* Process ID of login process.  */
47|   char ut_line[__UT_LINESIZE];  /* Devicename.  */
48|   char ut_id[4];               /* Inittab ID. */
49|   char ut_user[__UT_NAMESIZE];  /* Username.  */
50|   char ut_host[__UT_HOSTSIZE];  /* Hostname for remote login.  */
51|   struct __exit_status ut_exit; /* Exit status of a process marked
...|
64|   long int ut_session;         /* Session ID, used for windowing.  */
65|   struct timeval ut_tv;        /* Time entry was made.  */
...|
67|   __int32_t ut_addr_v6[4];     /* Internet address of remote host.  */
68|   char __unused[20];           /* Reserved for future use.  */
69| };
```

Example: Portable OpenSSH cvs-20051008

loginrec.h

```
68| struct logininfo {
69|         char       progname[LINFO_PROGSIZE];    /* name of program (for PAM) */
70|         int        progname_null;
71|         short int  type;                        /* type of login (LTYPE_*) */
72|         int        pid;                         /* PID of login process */
73|         int        uid;                         /* UID of this user */
74|         char       line[LINFO_LINESIZE];        /* tty/pty name */
75|         char       username[LINFO_NAMESIZE];    /* login username */
76|         char       hostname[LINFO_HOSTSIZE];    /* remote hostname */
77|         /* 'exit_status' structure components */
78|         int        exit;                        /* process exit status */
79|         int        termination;                 /* process termination status */
80|         /* struct timeval (sys/time.h) isn't always available, if it isn't we'll
81|          * use time_t's value as tv_sec and set tv_usec to 0
82|          */
83|         unsigned int tv_sec;
84|         unsigned int tv_usec;
85|         union login_netinfo hostaddr;       /* caller's host address(es) */
86|  }; /* struct logininfo */
```

Another tricky difference: signals

- Signal delivery semantics differ between Unix-like systems
  - Mainly based on SysV vs. BSD heritage
- BSD-ish systems generally restart active syscalls when a signal is received
- SysV-ish systems let the syscall return EINTR
- There are also ambiguities over whether signal handlers are reinstated after they are run
- OpenSSH uses a technique from W. Richard Stevens to cope with this
  - Write our own signal() replacement with consistent semantics

Example: Portable OpenSSH cvs-20051008

openbsd-compat/bsd-misc.c

```
189| mysig_t
190| mysignal(int sig, mysig_t act)
191| {
192| #ifdef HAVE_SIGACTION
193|         struct sigaction sa, osa;
194|
195|         if (sigaction(sig, NULL, &osa) == -1)
196|                 return (mysig_t) -1;
197|         if (osa.sa_handler != act) {
198|                 memset(&sa, 0, sizeof(sa));
199|                 sigemptyset(&sa.sa_mask);
200|                 sa.sa_flags = 0;
201| #ifdef SA_INTERRUPT
202|                 if (sig == SIGALRM)
203|                         sa.sa_flags |= SA_INTERRUPT;
204| #endif
205|                 sa.sa_handler = act;
206|                 if (sigaction(sig, &sa, NULL) == -1)
207|                         return (mysig_t) -1;
208|         }
209|         return (osa.sa_handler);
210| #else
211|         #undef signal
212|         return (signal(sig, act));
213| #endif
214| }
```

Signals continued

- This gives us consistency, but it doesn't entirely solve the syscall restart problem
  - Reads and writes may still return early
  - Other syscalls aren't guaranteed to complete either
- It is still necessary to retry important calls on failure
- For read() and write() we use the *atomicio()* wrapper
  - Automatically restarts reads or writes, updating buffer offset and transfer count
  - Also returns size_t (unsigned), to make our code easier to rid of signed vs. unsigned bugs
- Downside: parts of the application block
  - Makes turning ssh into a library more difficult

Example: OpenSSH cvs-20051008

atomicio.c

```
35| size_t
36| atomicio(f, fd, _s, n)
37|         ssize_t (*f) (int, void *, size_t);
38|         int fd;
39|         void *_s;
40|         size_t n;
41| {
42|         char *s = _s;
43|         size_t pos = 0;
44|         ssize_t res;
45|
46|         while (n > pos) {
47|                 res = (f) (fd, s + pos, n - pos);
48|                 switch (res) {
49|                 case -1:
50|                         if (errno == EINTR || errno == EAGAIN)
51|                                 continue;
52|                         return 0;
53|                   case 0:
54|                           errno = EPIPE;
55|                           return pos;
56|                 default:
57|                         pos += (u_int)res;
58|                 }
59|         }
60|         return (pos);
61| }
```

Differences with security implications: PAM

- Pluggable Authentication Modules (PAM)
  - Ambiguities in specification
  - Linux-PAM API passes prompt messages as *array of pointers to struct pam_message*
  - Sun-derived API and OpenPAM do *pointer to array of struct pam_message*
  - Result: possible overflow in privileged code
- We use an accessor macro to avoid this:

```
#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
```

Differences with security implications: setuid

- Differences in setuid/setgid semantics
- Great paper: Setuid Demystified, H. Chen, D. Wagner and D. Dean, USENIX 2002
  - Various systems implement setuid differently
  - Some usage patterns can lead to incomplete privilege revocation
- OpenSSH implements their recommendations:
  - Prefer *setresuid()* when available
    - Unambiguous semantics
  - Do a more complicated dance when it isn't:
    - seteuid()
    - setuid()
    - Try to restore uid and fatal() if it succeeds
    - Fetch effective and real uid and fatal() if privilege hasn't been dropped

Differences with security implications (more!)

- Dangerous platform defaults
  - AIX linker allowed loading for shared libraries from current directories
  - Portable OpenSSH did the right thing when using IBM's compiler
  - Got it wrong when using gcc instead
- Sharing binaries between adjusted systems
  - Differences in limits, e.g. PATH_MAX, NGROUPS_MAX
  - Bang - buffer overflow
- Some systems don't disable ptrace() after set[ug]id

Choosing the right API

- The best APIs (from a security perspective) are not universally supported
  - More recent platforms generally have better support
- Some good examples:
  - setresuid() and setresgid() (just mentioned)
  - /dev/random
    - First implemented on Linux and the BSDs, now adopted by Solaris
    - Absolutely essential for security software
  - closefrom()
    - Started on Solaris, now on the BSDs
- An unfortunate example: strlcpy() and strlcat()
  - First appeared in OpenBSD
  - Now in all the BSDs, Solaris, Linux (kernel)
  - Basically everywhere *except* glibc
    - maintainer is philosophically opposed
    - Result: over 100 packages maintain their own replacement

Choosing the right API (continued)

- Secure portability is **not** picking a lowest common denominator
- It is usually easy to replace missing APIs when they aren't there
  - More difficult to make software use them when they are
- Modern systems typically include good APIs anyway
  - If they aren't there already, then they will probably be in the next release
  - Why write your applications for yesterday's systems?

Conclusion - Simple rules

1. Don’t clutter main code paths

2. Don’t nest preprocessor

3. Pick the best possible API, even if it isn’t universally supported

4. Replace missing or broken functions in a separate library

5. Wherever possible, use a maintained replacement (e.g. libc)

6. When a simple replacement isn’t sufficient to code with variance between systems, abstract back to a superset of their functionality

7. Be alert for subtle platform differences, especially in privileged areas

Questions?