Adding atomic FD_CLOEXEC support

Eric Blake

Last updated 2020-03-12

This white paper tracks the proposed resolution of Austin Group defect report 411 (http://austingroupbugs.net/view.php?id=411), aimed at extending support for atomic FD_CLOEXEC support on file descriptors in Issue 8 of POSIX. Issue 7 standardized O_CLOEXEC for *open( )* and added F_DUPFD_CLOEXEC for *fcntl( )*, but omitted support for atomically setting FD_CLOEXEC when allocating file descriptors through other interfaces.

Table of Contents

Issue

The resolution of http://austingroupbugs.net/view.php?id=149 documented that using *close( )* on arbitrary file descriptors is unsafe, and that applications should instead atomically create file descriptors with FD_CLOEXEC already set. This is possible with *open( )* and *openat( )* using O_CLOEXEC, and with *fcntl( )* using F_DUPFD_CLOEXEC, but there are numerous other interfaces in the standard which also create new file descriptors where atomic FD_CLOEXEC was not possible. Without support for atomic FD_CLOEXEC on all possible file descriptors, multi-threaded applications have a data race where one thread can call *fork( )* or *posix_spawn( )* in the window between when another thread has created a new file descriptor and had a chance to use *fcntl( )* to mark it FD_CLOEXEC, and thus leak unintended file descriptors into a child process.

This bug goes hand-in-hand with http://austingroupbugs.net/view.php?id=368 which guarantees that all other file descriptors not exposed to the application are atomically marked FD_CLOEXEC. As it adds new interfaces, it is necessarily targeted to Issue 8.

Most of these changes are modeled after existing practice in at least Linux and Cygwin. Also, the *pipe2( )* function is emulated on many other platforms in the gnulib library, although FD_CLOEXEC is not atomic without underlying support.

In the process of adding this, I also folded in the changes in C1X to add the *'x'* modifier to the *fopen( )* mode, although I have added some additional requirements with <CX> shading to make it more useful. We may decide to make further usability enhancements, such as allowing *"wx+"* to be a valid mode (as is the case in glibc), rather than limiting ourselves to just the C1X spelling of *"w+x"*.

I noticed that the standard is silent on the behavior of *accept( )* when used on a socket marked O_NONBLOCK, and since BSD and Linux behaviors differ (BSD inherits O_NONBLOCK while Linux always clears the flag in the new descriptor), I documented that, while mandating particular behavior for the new function. That is, it is implementation-defined whether `accept(sock, addr, len)` behaves like `accept4(sock, addr, len, 0)` or `accept4(sock, addr, len, SOCK_NONBLOCK)` when *sock* is O_NONBLOCK.

Also, I don't know of any platform where *posix_spawn_file_actions_adddup2( )* can clear FD_CLOEXEC, but that seems like a useful change to make while tightening the specification on how to properly use FD_CLOEXEC, since the only standardized alternative for explicitly handing a file descriptor to the child process while leaving it FD_CLOEXEC in the parent process involves more complexity and risks EMFILE failure.

Not added here, but worth thinking about:

BSD and glibc both provide *mkstemps( )* (create a temporary file with a specified suffix in the filename), and glibc added *mkostemps( )* as the counterpart for *mkstemps( )* to match its *mkostemp( )* call added here.

Adding *posix_spawn_file_actions_openat( )* might be useful.

The documentation for SCM_RIGHTS as a means of creating new file descriptors is very minimal (a one-line mention in <**sys/socket.h**>); perhaps the *sendmsg( )* and *recvmsg( )* pages should document it further, as well as adding pages to further document the various CMSG_ macros. It may also be worth adding requirements from RFC 3542 on the CMSG_ macros, including the addition of CMSG_SPACE.

The fact that *fcntl( )* has unspecified results on typed memory file descriptors seems a bit worrying; perhaps we should improve things to state that certain actions (F_GETFD, F_SETFD, F_DUPFD, F_DUPFD_CLOEXEC) are still well-defined, while leaving other actions unspecified, especially since *dup( )* is well-defined but is identical to *fcntl( )* with F_DUPFD.

Desired Action

<stdlib.h>

After page 356 line 12007 [XBD <**stdlib.h**>], add a line with CX shading:

```
int mkostemp(char *, int);
```

<sys/socket.h>

Change page 383 line 12865 [XBD <**sys/socket.h**>] from:

> The <**sys/socket.h**> header shall define the following symbolic constants with distinct values:

to

> The <**sys/socket.h**> header shall define the following socket types (see [xref to XSH 2.10.6]) as symbolic constants with distinct values:

At page 383 line 12869, add:

> Implementations may provide additional socket types.
>
> The header shall define the following socket creation flags, for use in *socket( )*, *socketpair( )*, and *accept4( )*. These flags shall be symbolic constants with values that are bitwise distinct from each other and from all SOCK_* constants representing socket types:
>
> SOCK_NONBLOCK Create a socket file descriptor with the O_NONBLOCK flag atomically set on the new open file description.
>
> SOCK_CLOEXEC Create a socket file descriptor with the FD_CLOEXEC flag atomically set on that file descriptor.

Implementations may provide additional socket creation flags.

After page 384 line 12897, add a line:

MSG_CMSG_CLOEXEC Atomically set the FD_CLOEXEC flag on any file descriptors created via SCM_RIGHTS during *recvmsg( )*.

After page 385 line 12920, add a line:

```
int accept4(int, struct sockaddr *restrict, socklen_t
*restrict, int);
```

<unistd.h>

After page 443 line 15050 [XBD <**unistd.h**>], add a line:

```
int dup3(int, int, int);
```

After page 444 line 15094, add a line:

```
int pipe2(int [2], int);
```

XSH 2.4

At page 489 line 16722 [XSH 2.4.3 Signal Actions], insert the following functions in sorted order into the list of async-signal-safe functions:

accept4( )
dup3( )
pipe2( )

XSH 2.9

At page 512 line 17693 [XSH 2.9.5.2 Cancellation Points], insert "*accept4( )*" in sorted order into the list of cancellation point functions.

At page 516 line 17860 [XSH 2.9.7 Thread Interactions with Regular File Operations], insert "*dup3( )*" in sorted order into the list of atomic file operation functions.

XSH 2.10

At page 527 line 18326 [XSH 2.10.20.2 Compatibility with IPv4], change "in the *accept( )*," to "in the *accept( )*, *accept4( )*,".

accept()

At page 559 line 19366 [XSH *accept( )* NAME], change "accept - accept a new connection on a socket" to "accept, accept4 - accept a new connection on a socket"

After page 559 line 19370 [SYNOPSIS], add a line:

```
int accept4(int socket, struct sockaddr *restrict address,
socklen_t *restrict address_len, int flag);
```

After page 559 line 19395 [DESCRIPTION], add the following:

If O_NONBLOCK is set on the file description for *socket*, it is unspecified whether O_NONBLOCK will be set on the file description created by *accept( )*.

The *accept4( )* function shall be equivalent to the *accept( )* function, except that the O_NONBLOCK flag shall not be set on the new file description if the *flag* argument is 0. Additionally, the *flag* argument can be constructed from a bitwise-inclusive OR of flags from the following list:

SOCK_CLOEXEC Atomically set the FD_CLOEXEC flag on the new file descriptor.

SOCK_NONBLOCK Set the O_NONBLOCK file status flag on the new file description.

Implementations may define additional flags.

At page 559 line 19397 [RETURN VALUE], change "*accept( )*" to "*accept( )* and *accept4( )*"

At page 559 lines 19400 and 19417 [ERRORS], change "*accept( )* function" to "*accept( )* and *accept4( )* functions"

After page 560 line 19419 [ERRORS], add the following:

The *accept4( )* function may fail if:

[EINVAL] The value of the flag argument is invalid.

At page 560 line 19426 [RATIONALE], change "None." to:

The SOCK_CLOEXEC flag of *accept4( )* is necessary to avoid a data race in multi-threaded applications. Without it, a file descriptor is leaked into a child process created by one thread in the window between another thread creating a file descriptor with *accept( )* and then using *fcntl( )* to set the FD_CLOEXEC flag. The SOCK_NONBLOCK flag is for convenience in avoiding additional *fcntl( )* calls, as well as providing specific control over the O_NONBLOCK flag, since traditional implementations of *accept( )* differ on whether O_NONBLOCK is inherited from the socket argument.

creat()

After page 702 line 23749 [XSH *creat( )* RATIONALE], add the following:

In multi-threaded applications, the *creat( )* function can leak file descriptors into child processes. Applications should instead use *open( )* with the O_CLOEXEC flag to avoid the leak.

dup()

At page 741 line 24863 [XSH *dup( )* NAME], change "dup, dup2" to "dup, dup2, dup3"

After page 741 line 24867 [SYNOPSIS], add a line:

```
int dup3(int fildes, int fildes2, int flag);
```

After page 741 line 24881 [DESRIPTION], add the following:

> The *dup3( )* function shall be equivalent to the *dup2( )* function if the *flag* argument is 0, except that it shall be an error if *fildes* is equal to *fildes2*. Additionally, the *flag* parameter can be set to O_CLOEXEC (from **<fcntl.h>**) to cause FD_CLOEXEC flag to be set on the new file descriptor.

At page 741 line 24882 [DESCRIPTION], change "*dup2( )* function" to "*dup2( )* or *dup3( )* functions".

At page 741 lines 24890 and 24894 [ERRORS], change "*dup2( )* function" to "*dup2( )* and *dup3( )* functions".

After page 741 line 24893 [ERRORS], add the following:

> The *dup3( )* function shall fail if:
>
> [EINVAL] The *fildes* and *fildes2* arguments are equal.

After page 741 line 24895 [ERRORS], add the following:

> The *dup3( )* function may fail if:
>
> [EINVAL] The value of the *flag* argument is invalid.

After page 742 line 24926 [RATIONALE], add the following:

> The *dup3( )* function with the O_CLOEXEC flag is necessary to avoid a data race in multi-threaded applications. Without it, a file descriptor is leaked into a child process created by one thread in the window between another thread creating a file descriptor with *dup2( )* and then using *fcntl( )* to set the FD_CLOEXEC flag. The safe counterpart for avoiding the same race in *dup( )* is the use of the F_DUP_CLOEXEC action of the *fcntl( )* function.

fdopen()

At page 820 line 27380 [XSH *fdopen( )* DESCRIPTION], change:

> The *mode* argument is a character string having one of the following values:

to:

> The *mode* argument points to a character string, which shall also be valid for *fopen( )*. The string prefix has the following effects:

After page 820 line 27386 [DESCRIPTION], change:

> The meaning of these flags is exactly as specified in *fopen( )*, except that modes beginning with *w* shall not cause truncation of the file.

to:

> The meaning of these flags is exactly as specified in *fopen( )*, except that modes beginning with *w* shall not cause truncation of the file, and the use of *x* shall have no effect. The FD_CLOEXEC flag of *fildes* shall be unchanged if *e* was not present, and shall be set if *e* is present.

At page 821 line 27424 [RATIONALE], add the following to the same paragraph:

Since *fdopen( )* does not create a file, the *x* mode modifier is silently ignored. The *e* mode modifier is not strictly necessary for *fdopen( )*, since FD_CLOEXEC must not be changed when it is absent; however, it is standardized here since that modifier it is necessary to avoid a data race in multi-threaded applications using *freopen( )*, and consistency dictates that all functions accepting *mode* strings should allow the same set of strings.

fdopendir()

At page 823 line 27471 [XSH *fdopendir( )* DESCRIPTION], change:

It is unspecified whether the FD_CLOEXEC flag will be set on the file descriptor by a successful call to *fdopendir( )*.

to:

It is unspecified whether the FD_CLOEXEC flag will be set on the file descriptor by a successful call to *fdopendir( )* if it was not previously set. However, the flag shall not be cleared if it was previously set.

fopen()

At page 877 line 29115 [XSH *fopen( )* DESCRIPTION], change:

The *mode* argument points to a string. If the string is one of the following, the file shall be opened in the indicated mode. Otherwise, the behavior is undefined.

to:

The *mode* argument points to a character string. If the string begins with one of these six prefixes, followed by a (possibly empty) suffix consisting of the additional characters documented below, then the file shall be opened in the mode indicated by the prefix. Otherwise, the behavior is undefined.

After page 877 line 29125 [DESCRIPTION], add the following:

Additionally, the following characters may appear anywhere in the suffix of the *mode* string, to further affect how the file is opened. Behavior is unspecified if a character occurs more than once.

<CX>*'e'* The underlying file descriptor shall have the FD_CLOEXEC flag atomically set, as if by the O_CLOEXEC flag to *open( )*.</CX>

'x' If specified with a prefix beginning with *'w'* <CX>or *'a'*</CX>, then the function shall fail if the file already exists, <CX>as if by the O_EXCL flag to *open( )*. If specified with a prefix beginning with *'r'*, this modifier shall have no effect.</CX>

After page 878 line 29168 [ERRORS], add a line:

<CX>[EEXIST] The *mode* argument begins with *'w'* or *'a'* and includes *'x'*, but the file already exists.</CX>

At page 879 line 29223 [RATIONALE], change "None." to:

The *e* mode suffix character is provided as a convenience to avoid a data race in multi-threaded applications. Without it, a file descriptor is leaked into a child process created by one thread in the window between another thread creating a file descriptor with *fopen( )* and then using *fileno( )* and *fcntl( )* to set the FD_CLOEXEC flag. It is also possible to avoid the race by using *open( )* with O_CLOEXEC followed by *fdopen( )*, however, there is no safe alternative for the *freopen( )* function, and consistency dictates that the *e* modifier should be standardized for all functions that accept *mode* strings.

The *x* mode suffix character was added by C1x only for files opened with a *mode* string beginning with *w*. However, this standard requires that it also work for mode strings beginning with *a*, as well as being silently ignored rather than being an error for mode strings beginning with *r*. Therefore, while *open( )* has undefined behavior if O_EXCL is specified without O_CREAT, the same is not true of *fopen( )*.

This standard follows the lead of C1x in requiring that *b* and + appear in the mode prefix and *x* in the mode suffix; however, implementations are encouraged to treat both *b* and + as part of the mode suffix (requiring only *w*, *a*, or *r* as the first character and letting all other characters appear in any order).

freeaddrinfo()

At page 917, line 30691 [XSH *freeaddrinfo( )* DESCRIPTION], change:

as defined in *socket( )*

to

as defined in [xref to 2.10.6]

At page 920 line 30800 [APPLICATION USAGE], add:

The *ai_socktype* field pointed to by *hints* is just the socket type; not the socket type and flags that can be specified when the socket is created. Passing in socket creation flags will cause a failure with EAI_SOCKTYPE.

freopen()

At page 923 line 30897 [XSH *freopen( )* DESCRIPTION], change "*freopen( )* Mode" to "*freopen( )* Mode Prefix".

After page 923 line 30903 [DESCRIPTION], add the following:

<CX>Additionally, the presence of *e* in the mode suffix shall behave as if the O_CLOEXEC flag to *open( )* were in use. The presence of *x* in the mode suffix where the prefix begins with *a* or *w* shall behave as if the O_EXCL flag to *open( )* were specified, and shall be ignored if the prefix begins with *r*.</CX>

After page 924 line 30914 [ERRORS], add a line:

<CX>[EEXIST] The *mode* argument begins with *'w'* or *'a'* and includes *'x'*, but the file already exists.</CX>

At page 925 line 30979 [RATIONALE], change "None." to:

The *e* mode suffix character is necessary to avoid a data race in multi-threaded applications. Without it, a file descriptor is leaked into a child process created by one thread in the window between another thread creating a file descriptor with *freopen( )* and then using *fileno( )* and *fcntl( )* to set the FD_CLOEXEC flag.

The *x* mode suffix character was added by C1x only for files opened with a *mode* string beginning with *w*. However, this standard requires that it also work for mode strings beginning with *a*, as well as being silently ignored rather than being an error for mode strings beginning with *r*. Therefore, while *open( )* has undefined behavior if O_EXCL is specified without O_CREAT, the same is not true of *freopen( )*.

This standard follows the lead of C1x in requiring that *b* and + appear in the mode prefix and *x* in the mode suffix; however, implementations are encouraged to treat both *b* and + as part of the mode suffix (requiring only *w*, *a*, or *r* as the first character and letting all other characters appear in any order).

mkdtemp()

At page 1292 line 42413 [XSH *mkdtemp( )* NAME], change "mkdtemp, mkstemp" to "mkdtemp, mkostemp, mkstemp".

After page 1292 line 42416 [SYNOPSIS], add a line:

```
int mkostemp (char *template, int flag);
```

After page 1292 line 42433 [DESCRIPTION], add the following:

The *mkostemp( )* function shall be equivalent to the *mkstemp( )* function, except that the *flag* argument may contain additional flags (from **<fcntl.h>**) to be used as if by *open( )*. Behavior is unspecified if the flag argument contains more than the following flags:

O_APPEND Set append mode.

O_CLOEXEC Set the FD_CLOEXEC file descriptor flag.

<SIO>O_DSYNC Write according to the synchronized I/O data integrity completion.</SIO>

<SIO>O_RSYNC Synchronized read I/O operations.</SIO>

<XSI|SIO>O_SYNC Write according to synchronized I/O file integrity completion.</XSI|SIO>

At page 1293 line 42463 [ERRORS], change:

The error conditions for the *mkstemp( )* function are defined in *open( )*.

to:

The error conditions for the *mkstemp( )* and *mkostemp( )* functions are defined in *open( )*. Additionally, the *mkostemp( )* function may fail if:

[EINVAL] The value of the *flag* argument is invalid.

At page 1293 line 42479 [RATIONALE], replace "None." with:

The function *mkostemp( )* with the O_CLOEXEC flag is necessary to avoid a data race in multi-threaded applications. Without it, a file descriptor is leaked into a child process created by one thread in the window between another thread creating a temporary file descriptor with *mkstemp( )* and then using *fcntl( )* to set the FD_CLOEXEC flag.

pclose()

At page 1396 lines 45748-45763 [XSH *pclose( )* RATIONALE], replace the *pclose( )* sample implementation (from “The code sample below” to the end of the section) with:

See the RATIONALE for *popen( )* for a sample implementation of *pclose( )*.

pipe()

At page 1400 line 45830 [XSH *pipe( )* NAME], change "pipe" to "pipe, pipe2".

After page 1400 line 45833 [SYNOPSIS], add a line:

```
int pipe2(int fildes[2], int flag);
```

After page 1400 line 45849 [DESCRIPTION], add the following:

The *pipe2( )* function shall be equivalent to the *pipe( )* function if the *flag* argument is 0. Additionally, the *flag* argument can be constructed from a bitwise-inclusive OR of flags from the following list (provided by <**fcntl.h**>):

O_CLOEXEC Atomically set the FD_CLOEXEC flag on both new file descriptors.

O_NONBLOCK Set the O_NONBLOCK file status flag on both new file descriptions.

At page 1400 line 45854 [ERRORS], change "*pipe( )* function" to "*pipe( )* and *pipe2( )* functions"

After page 1400 line 45858 [ERRORS], add the following:

The *pipe2( )* function may fail if:

[EINVAL] The value of the *flag* argument is invalid.

At page 1401 line 45897 [RATIONALE], add the following:

The O_CLOEXEC flag of *pipe2( )* is necessary to avoid a data race in multi-threaded applications. Without it, a file descriptor is leaked into a child process created by one thread in the window between another thread creating a file descriptor with *pipe( )* and then using *fcntl( )* to set the FD_CLOEXEC flag. The O_NONBLOCK flag is for convenience in avoiding additional *fcntl( )* calls.

popen()

At page 1407 line 46093 [XSH *popen( )* DESCRIPTION], change:

The *popen( )* function shall ensure that any streams from previous *popen( )* calls that remain open in the parent process are closed in the new child process.

to:

The *popen( )* function shall ensure that any streams from previous *popen( )* calls that remain open in the parent process are closed in the new child process, regardless of the FD_CLOEXEC status of the file descriptor underlying those streams.

At page 1407 lines 46095 and 46099 [DESCRIPTION], change "If *mode* is" to "If *mode* starts with".

After page 1407 line 46102 [DESCRIPTION], add the following:

3. If *mode* includes a second character of *e*, then the file descriptor underlying the stream returned to the calling process by *popen( )* shall have the FD_CLOEXEC flag atomically set. Additionally, if the implementation creates the file descriptor for use by the child process from within the parent process, then that file descriptor shall have the FD_CLOEXEC flag atomically set within the parent process. If *mode* does not have a second character, the FD_CLOEXEC flag of the underlying file descriptor returned by *popen( )* shall be clear.

At page 1407 line 46103 [DESCRIPTION], change "3." to "4.".

At page 1408 line 46148 [APPLICATION USAGE], change "*r* and *w*" to "*r*, *w*, *re*, and *we*".

After page 1409 line 46167 [RATIONALE], add the following:

The *e* mode modifier to *popen( )* is necessary to avoid a data race in multi-threaded applications. Without it, the parent's file descriptor is leaked into a second child process created by one thread in the window between another thread creating the pipe via *popen( )* then using *fileno( )* and *fcntl( )* on the result. Also, if the *popen( )* implementation temporarily has the child's file descriptor open within the parent, then that file descriptor could also be leaked if it is not atomically FD_CLOEXEC for the duration in which it is open in the parent.

The standard only requires that the implementation atomically set FD_CLOEXEC on file descriptors created in the parent process when the *e* mode modifier is in effect; implementations may also do so when the *e* modifier is not in use, provided that the FD_CLOEXEC bit is eventually cleared before *popen( )* completes, however, this is not required because any application worried about the potential file descriptor leak will already be using the *e* modifier.

Although the standard is clear that a conforming application should not call *popen( )* when file descriptor 0 or 1 is closed, implementations are encouraged to handle these cases correctly.

The following two examples demonstrate possible implementations of *popen( )* using other standard functions. These examples are designed to show FD_CLOEXEC handling rather than all aspects of thread safety, and implementations are encouraged to improve the locking mechanism around the state list to be more efficient, as well as to be more robust if file descriptor 0 or 1 is returned as either part of the pipe. Also, remember that other

implementations are possible, including one that uses an implementation-specific means of creating a pipe between parent and child where the parent process never has access to the child's end of the pipe. Both of these examples make use of the following helper functions, documented but not implemented here, to do the bookkeeping necessary to properly close all file descriptors created by other *popen( )* calls regardless of their FD_CLOEXEC status:

```
/* Obtain mutual exclusion lock, so that no concurrent popen( ) or
   pclose( ) calls are simultaneously modifying the list of tracked
   children. */
static void popen_lock(void);

/* Release mutual exclusion lock, without changing errno. */
static void popen_unlock(void);

/* Add the pid and stream pair to the list of tracked children, prior
   to any code that can clear FD_CLOEXEC on the file descriptor
   associated with stream. To be used while holding the lock. */
static void popen_add_pair(FILE *stream, pid_t pid);

/* Given a stream, return the associated pid, or -1 with errno set if
   the stream was not created by popen( ). To be used while holding
   the lock. */
static pid_t popen_get_pid(FILE *stream);

/* Remove stream and its corresponding pid from the list of tracked
   children. To be used while holding the lock. */
static void popen_remove(FILE *stream);

/* If stream is NULL, return the first tracked child; otherwise,
   return the next tracked child. Return NULL if all tracked children
   have been returned. To be used while holding the lock. */
static FILE *popen_next(FILE *stream);
```

The first example is based on *fork( )*:

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <unistd.h>
FILE *popen(const char *command, const char *mode)
{
  int fds[2];
  pid_t pid;
  FILE *stream;
  int target = mode[0] == 'w'; /* index of fds used by parent */

  /* Validate mode */
  if ((mode[0] != 'w' && mode[0] != 'r') ||
      mode[1 + (mode[1] == 'e')]) {
    errno = EINVAL;
    return NULL;
  }

  /* Create pipe and stream with FD_CLOEXEC set */
  if (pipe2(fds, O_CLOEXEC) < 0)
```

```
  return NULL;
stream = fdopen(fds[target], mode);
if (!stream) {
  int saved = errno;
  close(fds[0]);
  close(fds[1]);
  errno = saved;
  return NULL;
}

/* Create child process */
popen_lock();
pid = fork();
if (pid < 0) {
  int saved = errno;
  close(fds[!target]);
  fclose(stream);
  popen_unlock();
  errno = saved;
  return NULL;
}

/* Child process. */
if (!pid) {
  FILE *tracked = popen_next(NULL);
  while (tracked) {
    int fd = fileno(tracked);
    if (fd < 0 || close(fd))
      _exit(127);
    tracked = popen_next(tracked);
  }
  target = mode[0] == 'r'; /* Opposite fd in the child */
  /* Use dup2 or fcntl to clear FD_CLOEXEC on child's descriptor,
     FD_CLOEXEC will take care of the rest of fds[]. */
  if (fds[target] != target) {
    if (dup2(fds[target], target) != target)
      _exit(127);
  } else {
    int flags = fcntl(fds[target], F_GETFD);
    if (flags < 0 ||
        fcntl(fds[target], F_SETFD, flags & ~FD_CLOEXEC) < 0)
      _exit(127);
  }
  execl("/bin/sh", "sh", "-c", command, NULL);
  _exit(127);
}

/* Parent process. From here on out, the close and fcntl system
   calls are assumed to pass, since all inputs are valid and do not
   require allocating any fds or memory. Besides, excluding
   failures due to undefined behavior (such as another thread
   closing an fd it knows nothing about), cleanup from any defined
   failures would require stopping and reaping the child process,
   which may have worse consequences. */
close(fds[!target]);
popen_add_pair(stream, pid);
popen_unlock();
if (mode[1] != 'e') {
```

```
    int flags = fcntl(fds[target], F_GETFD);
    if (flags >= 0)
      fcntl(fds[target], F_SETFD, flags & ~FD_CLOEXEC);
  }
  return stream;
}
```

The second example is based on *posix_spawn( )*:

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <unistd.h>
#include <spawn.h>
extern char **environ;
FILE *popen(const char *command, const char *mode)
{
  int fds[2];
  pid_t pid;
  FILE *stream;
  int target = mode[0] == 'w'; /* index of fds used by parent */
  const char *argv[] = { "sh", "-c", command, NULL };
  posix_spawn_file_actions_t actions;
  int saved;
  FILE *tracked;

  /* Validate mode */
  if ((mode[0] != 'w' && mode[0] != 'r') ||
      mode[1 + (mode[1] == 'e')]) {
    errno = EINVAL;
    return NULL;
  }

  /* Create pipe and stream with FD_CLOEXEC set */
  if (pipe2(fds, O_CLOEXEC) < 0)
    return NULL;
  stream = fdopen(fds[target], mode);
  if (!stream) {
    saved = errno;
    close(fds[0]);
    close(fds[1]);
    errno = saved;
    return NULL;
  }

  /* Create child process */
  if (posix_spawn_file_actions_init(&actions)) {
    saved = errno;
    goto spawnerr1;
  }
  popen_lock();
  tracked = popen_next(NULL);
  while (tracked) {
    int fd = fileno(tracked);
    if (fd < 0 || posix_spawn_file_actions_addclose(&actions, fd))
      goto spawnerr2;
```

```
    tracked = popen_next(tracked);
  }
  if (posix_spawn_file_actions_adddup2(&actions, fds[!target], !target))
    goto spawnerr2;
  if (posix_spawn(&pid, "/bin/sh", &actions, NULL, (char **)argv,
environ)) {
spawnerr2:
    saved = errno;
    posix_spawn_file_actions_destroy(&actions);
    popen_unlock();
spawnerr1:
    close(fds[!target]);
    fclose(stream);
    errno = saved;
    return NULL;
  }

  /* From here on out, system calls are assumed to pass, since all
     inputs are valid and do not require allocating any fds or memory.
     Besides, excluding failures due to undefined behavior (such as
     another thread closing an fd it knows nothing about), cleanup
     from any defined failures would require stopping and reaping the
     child process, which may have worse consequences. */
  posix_spawn_file_actions_destroy(&actions);
  close(fds[!target]);
  popen_add_pair(stream, pid);
  popen_unlock();
  if (mode[1] != 'e') {
    int flags = fcntl(fds[target], F_GETFD);
    if (flags >= 0)
      fcntl(fds[target], F_SETFD, flags & ~FD_CLOEXEC);
  }
  return stream;
}
```

Both examples can share a common *pclose( )* implementation.

```
int pclose(FILE *stream)
{
  int status;
  popen_lock();
  pid_t pid = popen_get_pid(stream);
  if (pid < 0) {
    popen_unlock();
    return -1;
  }
  popen_remove(stream);
  popen_unlock();
  fclose(stream); /* Ignore failure */
  while (waitpid(pid, &status, 0) == -1) {
    if (errno != EINTR) {
      status = -1;
      break;
    }
  }
  return status;
}
```

Note that, while a particular implementation of *popen( )* (such as the two above) can assume a particular path for the shell, such a path is not necessarily valid on another system. The above examples are not portable, and are not intended to be.

posix_openpt()

After page 1420 line 46469 [XSH *posix_openpt( )* DESCRIPTION], add a line:

O_CLOEXEC Atomically set the FD_CLOEXEC flag on the file descriptor.

After page 1421 line 46515 [RATIONALE], add the following:

The function *posix_openpt( )* with the O_CLOEXEC flag is necessary to avoid a data race in multi-threaded applications. Without it, a file descriptor is leaked into a child process created by one thread in the window between another thread creating a file descriptor with *posix_openpt( )* and then using *fcntl( )* to set the FD_CLOEXEC flag.

posix_spawn_file_actions_adddup2()

At page 1433 line 46976 [XSH *posix_spawn_file_actions_adddup2( )* DESCRIPTION], add a sentence:

If *fildes* and *newfildes* are equal, then the action shall ensure that *newfildes* is inherited by the new process with FD_CLOEXEC clear, even if the FD_CLOEXEC flag of *fildes* is set at the time the new process is spawned, and even though *dup2( )* would not make such a change.

After page 1433 line 46999 [RATIONALE], add the following:

Although *dup2( )* is required to do nothing when *fildes* and *newfildes* are equal and *fildes* is an open descriptor, the use of *posix_spawn_file_actions_adddup2( )* is required to cause *fildes* to be accessible in the child with FD_CLOEXEC clear. This is because there is no counterpart *posix_spawn_file_actions_fcntl( )* that could be used for clearing the flag as an independent file action. It would also be possible to achieve this effect by using two calls to *posix_spawn_file_actions_adddup2( )* and a temporary *fildes* value known to not conflict with any other file descriptors, coupled with a *posix_spawn_file_actions_close( )* to avoid leaking the temporary, but this approach is complex, and risks EMFILE or ENFILE failure that can be avoided with the in-place removal of FD_CLOEXEC.

There is no need for *posix_spawn_file_actions_adddup3( )*, since it makes no sense to create a file descriptor with FD_CLOEXEC set before spawning the child process, where that file descriptor would immediately be closed again.

posix_typed_mem_open()

At page 1516 line 48927 [XSH *posix_typed_mem_open( )* DESCRIPTION], change:

The FD_CLOEXEC file descriptor flag associated with the new file descriptor shall be cleared.

to:

The FD_CLOEXEC file descriptor flag associated with the new file descriptor shall be cleared unless *oflag* includes O_CLOEXEC.

At page 1516 line 48930 [DESCRIPTION], change "*dup( ), dup2( ),*" to "*dup( ), dup2( ), dup3( ),*".

After page 1516 line 48937 [DESCRIPTION], add the following:

Additionally, the value of *oflag* may include the following flag:

O_CLOEXEC Set the FD_CLOEXEC file descriptor flag.

At page 1517 line 48967 [RATIONALE], replace "None." with:

The use of the O_CLOEXEC flag to posix_typed_mem_open( ) is necessary to avoid leaking typed memory file descriptors to child processes, since *fcntl( )* has unspecified results on typed memory objects and therefore cannot be used to set FD_CLOEXEC after the fact. The *exec* family of functions already unmaps all memory associated with a typed memory object, but does not close the file descriptor unless FD_CLOEXEC is also set.

pselect()

At page 1524 line 49184 [XSH *pselect( )* DESCRIPTION], change "*accept( )* function" to "*accept( )* or *accept4( )* function".

recvmsg()

After page 1764 line 56369 [XSH *recvmsg( )* DESCRIPTION], add the following:

MSG_CMSG_CLOEXEC On sockets that permit a *cmsg_type* of SCM_RIGHTS in the *msg_control* ancillary data as a means of copying file descriptors into the process, the file descriptors shall be created with the FD_CLOEXEC flag atomically set.

At page 1766 line 56432 [RATIONALE], replace "None." with:

The use of the MSG_CMSG_CLOEXEC flag to *recvmsg( )* when using SCM_RIGHTS to receive file descriptors via ancillary data is necessary to avoid a data race in multi-threaded applications. Without it, a file descriptor is leaked into a child process created by one thread in the window between another thread calling *recvmsg( )* and using *fcntl( )* to set the FD_CLOEXEC flag.

shm_open()

At page 1901 line 60458 [XSH *shm_open( )* RATIONALE], add a sentence:

The O_CLOEXEC open flag is not listed, because all shared memory objects are created with the FD_CLOEXEC flag already set; an application may later use *fcntl( )* to clear that flag to allow the shared memory file descriptor to be preserved across the *exec* family of functions.

socket()

After page 1968 line 62515 [XSH *socket( )* DESCRIPTION], add the following:

Additionally, the *type* argument may contain the bitwise-inclusive OR of flags from the following list:

SOCK_CLOEXEC Atomically set the FD_CLOEXEC flag on the new file descriptor.

SOCK_NONBLOCK Set the O_NONBLOCK file status flag on the new file description.

Implementations may define additional flags.

At page 1969 line 62547 [RATIONALE], replace "None." with:

The use of the SOCK_CLOEXEC flag in the *type* argument of *socket( )* is necessary to avoid a data race in multi-threaded applications. Without it, a file descriptor is leaked into a child process created by one thread in the window between another thread calling *socket( )* and using *fcntl( )* to set the FD_CLOEXEC flag. The SOCK_NONBLOCK flag is for convenience in avoiding additional *fcntl( )* calls.

socketpair()

After page 1970 line 62586 [XSH *socketpair( )* DESCRIPTION], add the following:

Additionally, the *type* argument may contain the bitwise-inclusive OR of flags from the following list:

SOCK_CLOEXEC Atomically set the FD_CLOEXEC flag on the new file descriptors.

SOCK_NONBLOCK Set the O_NONBLOCK file status flag on the new file descriptions.

Implementations may define additional flags.

At page 1971 line 62620 [RATIONALE], replace "None." with:

The use of the SOCK_CLOEXEC flag in the *type* argument of *socketpair( )* is necessary to avoid a data race in multi-threaded applications. Without it, a file descriptor is leaked into a child process created by one thread in the window between another thread using *socketpair( )* and using *fcntl( )* to set the FD_CLOEXEC flag. The SOCK_NONBLOCK flag is for convenience in avoiding additional *fcntl( )* calls.

tmpfile()

After page 2122 line 67169 [XSH *tmpfile( )* APPLICATION USAGE], add the following:

In multi-threaded applications, the *tmpfile( )* function can leak file descriptors into child processes. Applications should instead use *mkostemp( )* with the O_CLOEXEC flag, followed by *fdopen( )*, to avoid the leak.

tmpnam()

At page 2123 line 67239 [XSH *tmpnam( )* APPLICATION USAGE], change "*mkstemp( )*" to "*mkostemp( ), mkstemp( )*".

XRAT B.2

At page 3533 line 119194 [XRAT B.2.8.3 Memory Management], change "*dup2( ),*" to "*dup2( ), dup3( ),*".

XRAT B.3

At page 3628 line 123297 [XRAT B.3.3 Examples for Spawn], change:

```
    if (dup2(fd, newfd) == −1)
```

to:

```
    if (fd == newfd)
    {
        int flags = fcntl(fd, F_GETFD);
        if (flags == -1)
            exit(127);
        flags &= ~FD_CLOEXEC;
        if (fcntl(fd, F_SETFD, flags) == -1)
            exit(127);
    }
    else if (dup2(fd, newfd) == -1)
```

XRAT D.2

At page 3690 line 125602 [XRAT D.2.3 Access to Data], change "*fopen( ),* and *pipe( )*" to "*fopen( ), freopen( ), pipe( ),* and *pipe2( )*".

At page 3690 line 125605, change "*dup2( ),*" to "*dup2( ), dup3( ),*".

XRAT E.1

At page 3713 line 126332 [XRAT E.1 Subprofiling Option Groups], add "*dup3( )*" in order to POSIX_FD_MGMT.

At page 3714 line 126381, add "*accept4( )*" in order to POSIX_NETWORKING.

At page 3714 line 126391, add "*pipe2( )*" in order to POSIX_PIPE.