Ulrich Drepper

[**website**|My Website]
[**userinfo**|livejournal userinfo]
[**archive**|journal archive]

Secure File Descriptor Handling

[Aug. 1st, 2008|**04:24 pm**]

[**Tags**|programming security linux]

During the 2.6.27 merge window a number of my patches were merge and now we are at the point where we can securely create file descriptors without the danger of possibly leaking information. Before I go into the details let's get some background information.

A file descriptor in the Unix/POSIX world has lots of state associated with it. One bit of information determines whether the file descriptor is automatically closed when the process executes an exec call to start executing another program. This is useful, for instance, to establish pipelines. Traditionally, when a file descriptor is created (e.g., with the default open() mode) this close-on-exec flag is not set and a programmer has to explicitly set it using

```
fcntl(fd, F_SETFD, FD_CLOEXEC);
```

Closing the descriptor is a good idea for two main reasons:

- the new program's file descriptor table might fill up. For every open file descriptor resources are consumed.
- more importantly, information might be leaked to the second program. That program might get access to information it normally wouldn't have access to.

It is easy to see why the latter point is such a problem. Assume this common scenario:

> A web browser has two windows or tabs open, both loading a new page (maybe triggered through Javascript). One connection is to your bank, the other some random Internet site. The latter contains some random object which must be handled by a plug-in. The plug-in could be an external program processing some scripting language. The external program will be started through a fork() and exec sequence, inheriting all the file descriptors open and not marked with close-on-exec from the web browser process.

The result is that the plug-in can have access to the file descriptor used for the bank connection. This is especially bad if the plug-in is used for a scripting language such a Flash because this could make the descriptor easily available to the script. In case the author of the script has malicious intentions you might end up losing money.

Until not too long ago the best programs could to is to set the close-on-exec flag for file descriptors as quickly as possible after the file descriptor has been created. Programs would break if the default for new file descriptors would be changed to set the bit automatically.

This does not solve the problem, though. There is a (possibly brief) period of time between the return of the open() call or other function creating a file descriptor and the fcntl() call to set the flag. This is problematic because the fork() function is signal-safe (i.e., it can be called from a signal handler). In multi-threaded code a second thread might call fork() concurrently. It is theoretically possible to avoid these races by blocking all signals and by ensuring through locks that fork() cannot be called concurrently. This very quickly get far too complicated to even contemplate:

- To block all signals, each thread in the process has to be interrupted (through another signal) and in the signal handler block all the other signals. This is complicated, slow, possibly unreliable, and might introduce deadlocks.

- Using a lock also means there has to be a lock around fork() itself. But fork() is signal safe. This means this step also needs to block all signals. This by itself requires additional work since child processes inherit signal masks.
- Making all this work in projects which come from different sources (and which non-trivial program doesn't use system or third-party libraries?) is virtually impossible.

It is therefore necessary to find a different solution. The first set of patches to achieve the goal went into the Linux kernel in 2.6.23, the last, as already mentioned, will be in the 2.6.27 release. The patches are all rather simple. They just extend the interface of various system calls so that already existing functionality can be taken advantage of.

The simplest case is the open() system call. To create a file descriptor with the close-on-exec flag atomically set all one has to do is to add the O_CLOEXEC flag to the call. There is already a parameter which takes such flags.

The next more complicated is the solution chosen to extend the socket() and socketcall() system calls. No flag parameter is available but the second parameter to these interfaces (the type) has a very limited range requirement. It was felt that overloading the parameter is an acceptable solution. It definitely makes using the new interfaces simpler.

The last group are interfaces where the original interface simply doesn't provide a way to pass additional parameters. In all these cases a generic flags parameter was added. This is preferable to using specialized new interfaces (like, for instance, dup2_cloexec) because we do and will need other flags. O_NONBLOCK is one case. Hopefully we'll have non-sequential file descriptors at some point and we can then request them using the flags, too.

The (hopefully complete) list of interface changes which were introduced is listed below. Note: these are the **userlevel** change. Inside the kernel things look different.

Userlevel Interface	What changed?
open	O_CLOEXEC flag added
fcntl	F_DUPFD_CLOEXEC command added
recvmsg	MSG_CMSG_CLOEXEC flag for transmission of file descriptor over Unix domain socket which has close-on-exec set atomically
dup3	New interface taking an addition flag parameter (O_CLOEXEC, O_NONBLOCK)
pipe2	New interface taking an addition flag parameter (O_CLOEXEC, O_NONBLOCK)
socket	SOCK_CLOEXEC and SOCK_NONBLOCK flag added to type parameter
socketpair	SOCK_CLOEXEC and SOCK_NONBLOCK flag added to type parameter
paccept	New interface taking an addition flag parameter (SOCK_CLOEXEC, SOCK_NONBLOCK) and a temporary signal mask
fopen	New mode 'e' to open file with close-on-exec set
popen	New mode 'e' to open pipes with close-on-exec set
eventfd	Take new flags EFD_CLOEXEC and EFD_NONBLOCK
signalfd	Take new flags SFD_CLOEXEC and SFD_NONBLOCK
timerfd	Take new flags TFD_CLOEXEC and TFD_NONBLOCK
epoll_create1	New interface taking a flag parameter. Support EPOLL_CLOEXEC and EPOLL_NONBLOCK
inotify_init1	New interface taking a flag parameter (IN_CLOEXEC, IN_NONBLOCK)

When should these interfaces be used? The answer is simple: whenever the author is not sure that no asynchronous fork()+exec can happen or a concurrently running threads executes fork()+exec (or posix_spawn(), BTW).

Application writers might have control over this. But I'd say that in all library code one has to play it safe. In glibc we do now in almost all interfaces open the file descriptor with the close-on-exec flag set. This means a lot of work but it has to be done. Applications also have to change (see this autofs bug, for instance).

link Reply

Comments:

From: *(Anonymous)*
2008-08-03 09:48 pm (UTC) **(Link)**

Example?

Hi, I'm a bit confused about how the vulnerability here is supposed to work. Could you provide a small example schedule of concurrent syscalls that would illustrate the problem? Thanks in advance!

(Reply) (Thread)

From: **udrepper**
2008-08-04 03:55 pm (UTC) **(Link)**

Re: Example?

> "I'm a bit confused about how the vulnerability here is supposed to work. Could you provide a small example schedule of concurrent syscalls that would illustrate the problem?"

I thought I explained it sufficiently. Apparently not. There are two scenarios, both of which can and do happen widely. Most obviously is the multi-thread code:

Thread 1	Thread 2	New process
fd = open(...)		
	fork()	
fcntl(fd, F_SETFD, FD_CLOEXEC)		
		execve(...)

The new process inherits all the open file descriptors without close-o-exec set. Just look at, for example, your web browser. This scenario probably happens dozens of times a day in there.

But one doesn't need threads. It can happen in a single-threaded application as well.

Normal Code	In Signal handler	New Process
fd = socket(...)		
	fork()	
fcntl(fd, F_SETFD, FD_CLOEXEC)		execve(...)

fork() is a signal-safe function and can be called from a signal handler. Therefore the above can at any point in time happen. And just because your program doesn't use it doesn't mean it cannot happen: a library a program links with might do it. Of course, library implementers always have to assume the worst.

(Reply) (Parent) (Thread)