pidfds

Process file descriptors on Linux

Christian Brauner
christian.brauner@ubuntu.com
@brau_ner
https://people.kernel.org/brauner
https://brauner.io

pidfd: what's that?

file descriptor referring to a process
specifically, an fd referring to a thread-group leader

stable, private handle
fd guarantees to reference the same process

pidfds use pre-existing stable process handle
reference struct pid, not task_struct

```
struct pid
{
        refcount_t count;
        unsigned int level;
        /* lists of tasks that use this pid */
        struct hlist_head tasks[PIDTYPE_MAX];
        /* wait queue for pidfd notifications */
        wait_queue_head_t wait_pidfd;
        struct rcu_head rcu;
        struct upid numbers[1];
};
```

Why do this in the first place?

pid recycling

avoid pitfalls of pid recycling on high-pressure systems

CVE-2019-6133: https://bugs.chromium.org/p/project-zero/issues/detail?id=1692
CVE-2014-5033: https://www.cvedetails.com/cve/CVE-2014-5033/
pid-based mac exploits: https://objective-see.com/blog/blog_0x41.html
https://doc.qt.io/qt-5/qprocess.html#startDetached
https://marc.info/?l=openssl-dev&m=130289811108150&w=2
CVE-2017-13209: (Android - Hardware Service Manager Arbitrary Service Replacement due to getpidcon)
https://www.exploit-db.com/exploits/43513
Issue 851: Android: racy getpidcon usage permits binder service replacement
https://bugs.chromium.org/p/project-zero/issues/detail?id=851

Why do this in the first place?

shared libraries
allow to spawn invisible helper processes

process management delegation
hand of a handle to a non-parent process (e.g. for waiting, signaling)

ubiquity of fds
common patterns already exist everywhere in userspace

Does userspace really care about this feature?

dbus
https://gitlab.freedesktop.org/dbus/dbus/issues/274

qt
https://codereview.qt-project.org/c/qt/qtbase/+/108456

systemd
https://github.com/systemd/systemd/issues/13101

criu
https://github.com/checkpoint-restore/criu/issues/717

lmkd
https://android-review.googlesource.com/c/platform/system/core/+/1088157

bpftrace
https://github.com/iovisor/bpftrace/issues/880

mio
https://github.com/samuelbrian/mio-pidfd

Prior art

Illumos
pure userspace emulation of stable process handle
procopen(), procrun(), procclose(), procfree(), etc.

OpenBSD, NetBSD
no private, stable process handles

FreeBSD
procdesc: pdfork(), pdgetpid(), pdkill()

Linux
forkfd(), CLONE_FD

Building a new api

4 kernel releases
individual elements to create a complete api

5.1

sending signals
using pidfds to reliably send signals

```
SYSCALL_DEFINE4(pidfd_send_signal, int, pidfd, int, sig,
                siginfo_t __user *, info, unsigned int, flags)
{
        int ret;
        struct fd f;
        struct pid *pid;
        kernel_siginfo_t kinfo;

        /* Enforce flags be set to 0 until we add an extension. */
        if (flags)
                return -EINVAL;

        f = fdget(pidfd);
        if (!f.file)
                return -EBADF;

        /* Is this a pidfd? */
        pid = pidfd_to_pid(f.file);
        if (IS_ERR(pid)) {
                ret = PTR_ERR(pid);
                goto err;
        }
```

5.2

CLONE_PIDFD
create pidfds at process creation time

O_CLOEXEC
pidfds are close-on-exec by default

/proc/<pid>/fd/fdinfo
contains pid of process in procfs pidns

```
/*
 * This has to happen after we've potentially unshared the file
 * descriptor table (so that the pidfd doesn't leak into the child
 * if the fd table isn't shared).
 */
if (clone_flags & CLONE_PIDFD) {
	retval = get_unused_fd_flags(O_RDWR | O_CLOEXEC);
	if (retval < 0)
		goto bad_fork_free_pid;

	pidfd = retval;

	pidfile = anon_inode_getfile("[pidfd]", &pidfd_fops, pid,
				     O_RDWR | O_CLOEXEC);
	if (IS_ERR(pidfile)) {
		put_unused_fd(pidfd);
		retval = PTR_ERR(pidfile);
		goto bad_fork_free_pid;
	}
	get_pid(pid);/* held by pidfile now */

	retval = put_user(pidfd, args->pidfd);
	if (retval)
		goto bad_fork_put_pidfd;
}
```

5.3

clone3 syscall
dedicated pidfd argument

```c
/*
 * Arguments for the clone3 syscall
 */
struct clone_args {
        __aligned_u64 flags;
        __aligned_u64 pidfd;
        __aligned_u64 child_tid;
        __aligned_u64 parent_tid;
        __aligned_u64 exit_signal;
        __aligned_u64 stack;
        __aligned_u64 stack_size;
        __aligned_u64 tls;
  };

SYSCALL_DEFINE2(clone3, struct clone_args __user *, uargs, size_t, size)
{
        int err;

        struct kernel_clone_args kargs;

        err = copy_clone_args_from_user(&kargs, uargs, size);
        if (err)
                return err;

        if (!clone3_args_valid(&kargs))
                return -EINVAL;

        return _do_fork(&kargs);
  }
```

5.3

polling support

exit notification for non-parents

```c
static void do_notify_pidfd(struct task_struct *task)
{
        struct pid *pid;

        WARN_ON(task->exit_state == 0);
        pid = task_pid(task);
        wake_up_all(&pid->wait_pidfd);
}

/*
 * Poll support for process exit notification.
 */
static unsigned int pidfd_poll(struct file *file, struct poll_table_struct *pts)
{
        struct task_struct *task;
        struct pid *pid = file->private_data;
        int poll_flags = 0;

        poll_wait(file, &pid->wait_pidfd, pts);

        rcu_read_lock();
        task = pid_task(pid, PIDTYPE_PID);
        /*
         * Inform pollers only when the whole thread group exits.
         * If the thread group leader exits before all other threads in the
         * group, then poll(2) should block, similar to the wait(2) family.
         */
        if (!task || (task->exit_state && thread_group_empty(task)))
                poll_flags = POLLIN | POLLRDNORM;
        rcu_read_unlock();

        return poll_flags;
  }
```

5.3

pidfds without CLONE_PIDFD

pidfd_open() to create pidfd

```c
SYSCALL_DEFINE2(pidfd_open, pid_t, pid, unsigned int, flags)
{
	int fd, ret;
	struct pid *p;

	if (flags)
		return -EINVAL;

	if (pid <= 0)
		return -EINVAL;

	p = find_get_pid(pid);
	if (!p)
		return -ESRCH;

	ret = 0;
	rcu_read_lock();
	if (!pid_task(p, PIDTYPE_TGID))
		ret = -EINVAL;
	rcu_read_unlock();

	fd = ret ?: pidfd_create(p);
	put_pid(p);
	return fd;
}
```

5.4

waiting through pidfds

P_PIDFD for waitid()

```
case P_PIDFD:
	type = PIDTYPE_PID;
	if (upid < 0)
		return -EINVAL;

	pid = pidfd_get_pid(upid);
	if (IS_ERR(pid))
		return PTR_ERR(pid);
	break;
default:
	return -EINVAL;
}

wo.wo_type    = type;
wo.wo_pid     = pid;
wo.wo_flags   = options;
wo.wo_info    = infop;
wo.wo_rusage  = ru;
ret = do_wait(&wo);
```

Kill-on-close

SIGKILL on last close
kill process when last fd referencing it is closed

exclusive waiting

CLONE_WAIT_PID
hide process from generic wait requests (e.g. waitid(P_ALL)

pidfds & namespaces

using pidfds for some namespace management tasks

Lessons learned

speed matters
choose a sustainable speed for developing features

be open about being "dumb"
it's ok to say "I don't know" or "I can't review that"

be resilient
reviews are a form of critique