Using Emilua to run desktop apps from LXC containers

A large amount of Emilua development time was spent on a sandboxing runtime for secure app development. As a result of this investment, Emilua became an interesting playground to experiment with Linux namespaces. In this post, I'll show how one can use Emilua to run desktop apps installed in LXC containers. At the end, I'll also show that the same solution can be used as an alternative for Firejail.

My threat model is not your threat model

There is no shortage of namesaces-oriented tooling assuming `unprivileged_userns_clone`. However this setting is not universally enabled in all Linux environments. Emilua can be used to do the same (that's how I wrote the initial prototype that eventually led to this blog post), but here we'll explore an environment with `unprivileged_userns_clone` disabled (i.e. you need to run a privileged daemon or SUID helper).

Process-based concurrency

Some OS concepts use properties that are attached at the process-level (e.g. credentials, umask, current working directory). Changing these properties is not safe if they are dependencies for other concurrent tasks in your process. It's unavoidable to interact with this type of property when you use Linux namespaces.

Emilua is a concurrency runtime and process-based concurrency is just one of the supported options. For shared-nothing concurrency, the actor model is implemented (each actor is a Lua VM). Usually you'll create new actors like this:

```
local my_actor = spawn_vm{
    module = module
}
```

However if you also set the parameter `subprocess` then the actor will be spawned in a new process:

```
local my_actor = spawn_vm{
    module = module,
    subprocess = {}
}
```

You can even define a Lua script to initialize several process attributes before the runtime for fibers and actors is initialized:

```
local my_actor = spawn_vm{
    module = module,
    subprocess = {
        init = {
            script = script,
            arg = some_fd --< e.g. an UNIX socket
        }
    }
}
```

You can use a subset of the POSIX API inside the initialization script to configure several process properties. Once the script finishes, all the memory allocated by the Lua VM is `explicit_bzero()`'ed and extra file descriptors are closed. By default, any function returning an error also aborts the process.

You can use actors and typical UNIX primitives (e.g. pipes, UNIX sockets, `SCM_RIGHTS`) to synchronize processes and make use of Linux namespaces.

Creating new Linux namespaces

To start a process in a new Linux namespace, you just set the appropriate parameter when you create a new actor:

```
local my_actor = spawn_vm{
    module = module,
    subprocess = {
        newns_user = true,
        newns_mount = true,
        newns_pid = true,
        newns_net = true,
        newns_uts = true,
        newns_ipc = true
    }
}
```

Except for the user namespace, the creation of namespaces require root privileges. On many patched Linux environments, even the creation of user namespaces is forbidden for unprivileged users. And even if you're authorized to create user namespaces you still need privileges to configure the network stack so one can build something interesting in a different network namespace.

As stated in the beginning of this post, it's possible to create tooling that work on environments where `unprivileged_userns_clone` is not disabled. However there are already too much of that. Here we'll create a tool that run a privileged daemon to start the containers.

An UNIX daemon

To start the daemon, we just set an UNIX socket that listens on some `/var/run` and loop on `accept()`:

```
local function handle_client(client)
    scope_cleanup_push(function() client:close() end)
    local buf = byte_span.new(1024)
    while true do
        local nread, fds = client:receive_with_fds(buf, 1)
        scope(function()
            scope_cleanup_push(function()
                for _, fd in ipairs(fds) do fd:close() end
            end)
            local req = json.decode(tostring(buf:slice(1, nread)))
            HANDLERS[req.request](conf, client, req.arguments, fds)
        end)
    end
end

while true do
    local client = acceptor:accept()
    spawn(function() handle_client(client) end):detach()
end
```

There are two ways to perform authorization to our socket. We can either use `filesystem.umask()` + `setres{u,g}id()` to create the socket managed by `acceptor` with proper permissions and let the traditional UNIX permission model take place, or we can control access programatically querying `client:get_option('remote_{credentials,security_label}')`. Once our handler is invoked, we're finally in business.

```
local HANDLERS = {
    spawn_linux_app = require('./spawn_linux_app').handler
}
```

Configuration file

Before we create our container, we need some user-provided settings. So let's read them from a file:

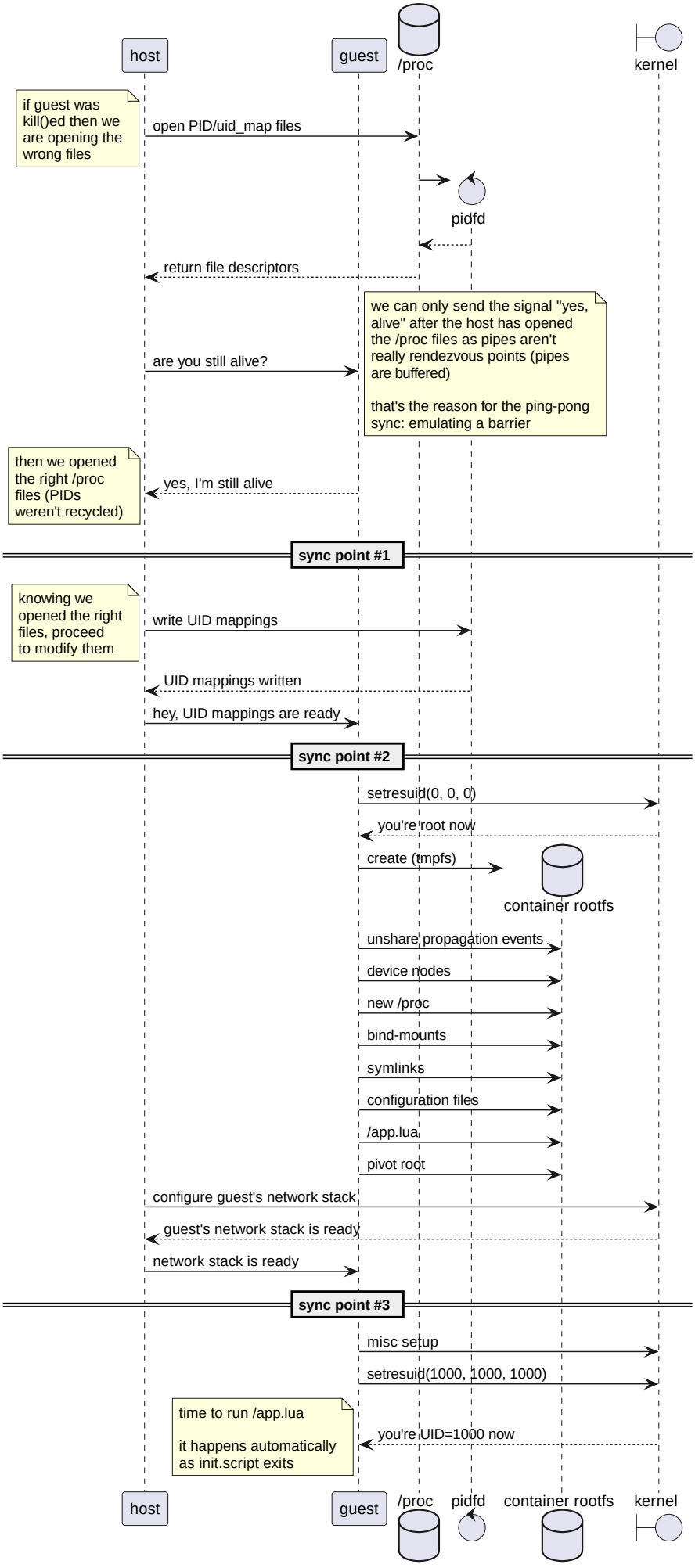
```
local conf

do
    conf = file.stream.new()
    conf:open(fs.path.new('/etc/packxd.json'), file.open_flag.read_only)
    local buf = byte_span.new(conf.size)
    stream.read_all(conf, buf)
    conf = json.decode(tostring(buf))
```

```
end
```

Here's the type of file we're expecting for our configuration:

```
{
    "permit_users": [1000],
    "guests": {
        "mycontainer01": {
            "rootfs": "/var/lib/lxc/mycontainer01/rootfs",
            "hostname": "myarchlinux",
            "resolv.conf": "nameserver 8.8.8.8",
            "veth": {
                "ifname": "veth-myarch01",
                "ipv4": {
                    "address": "192.168.12.104/24",
                    "gateway": "192.168.12.1"
                }
            },
            "environment": {
                "LANG": "C.UTF-8",
                "LC_CTYPE": "C.UTF-8",
                "PATH":
"/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/bin/site_perl:/usr/bin/vendor_perl:/usr/
bin/core_perl",
                "SHELL": "/bin/bash",
                "SHLVL": "0"
            },
            "root_symlinks": {
                "/bin": "usr/bin",
                "/lib": "usr/lib",
                "/lib64": "usr/lib",
                "/sbin": "usr/bin"
            }
        }
    }
}
```

Starting the container


host
guest
/proc
kernel
if guest was kill()ed then we are opening the wrong files
open PID/uid_map files
pidfd
return file descriptors
we can only send the signal "yes, alive" after the host has opened the /proc files as pipes aren't really rendezvous points (pipes are buffered)
that's the reason for the ping-pong sync: emulating a barrier
are you still alive?
then we opened the right /proc files (PIDs weren't recycled)
yes, I'm still alive
sync point #1
knowing we opened the right files, proceed to modify them
write UID mappings
UID mappings written
hey, UID mappings are ready
sync point #2
setresuid(0, 0, 0)
you're root now
create (tmpfs)
container rootfs
unshare propagation events
device nodes
new /proc
bind-mounts
symlinks
configuration files
/app.lua
pivot root
configure guest's network stack
guest's network stack is ready
network stack is ready
sync point #3
misc setup
setresuid(1000, 1000, 1000)
time to run /app.lua
it happens automatically as init.script exits
you're UID=1000 now
host
guest
/proc
pidfd
container rootfs
kernel

To start the container, first we create a pair of connected UNIX sockets to perform synchronization between host and guest:

```
local shost, sguest = unix.seqpacket_socket.pair()
sguest = sguest:release()
```

Then we use an `init.script` like this:

```
-- sync point #1
C.read(arg, 1)
C.write(arg, '.')

-- sync point #2 as tmpfs will fail on mkdir()
-- with EOVERFLOW if no UID/GID mapping exists
-- https://bugzilla.kernel.org/show_bug.cgi?id=183461
C.read(arg, 1)

C.setresuid(0, 0, 0)
C.setresgid(0, 0, 0)

-- unshare propagation events
C.mount(nil, '/', nil, C.MS_PRIVATE)

-- we'll use /mnt as the new /
C.umask(0)
C.mount(nil, '/mnt', 'tmpfs', 0)
C.mkdir('/mnt/proc', mode(7, 5, 5))
C.mount(nil, '/mnt/proc', 'proc', 0)
C.mkdir('/mnt/tmp', bit.bor(mode(7, 7, 7), C.S_ISVTX))
C.mkdir('/mnt/run', mode(7, 5, 5))
C.mkdir('/mnt/run/user', mode(7, 5, 5))
C.mkdir('/mnt/run/user/1000', mode(7, 0, 0))
C.chown('/mnt/run/user/1000', 1000, 1000)
C.mkdir('/mnt/etc', mode(7, 5, 5))

C.mkdir('/mnt/dev', mode(7, 5, 5))
C.mkdir('/mnt/dev/pts', mode(7, 5, 5))
C.mkdir('/mnt/dev/shm', bit.bor(mode(7, 7, 7), C.S_ISVTX))
C.mkdir('/mnt/dev/mqueue', bit.bor(mode(7, 7, 7), C.S_ISVTX))
C.symlink('/proc/self/fd', '/mnt/dev/fd')
C.symlink('/dev/pts/ptmx', '/mnt/dev/ptmx')
C.symlink('/proc/self/fd/0', '/mnt/dev/stdin')
C.symlink('/proc/self/fd/1', '/mnt/dev/stdout')
C.symlink('/proc/self/fd/2', '/mnt/dev/stderr')
C.mknod('/mnt/dev/full', mode(6, 6, 6), 0)
C.mount('/dev/full', '/mnt/dev/full', nil, C.MS_BIND)
C.mknod('/mnt/dev/null', mode(6, 6, 6), 0)
C.mount('/dev/null', '/mnt/dev/null', nil, C.MS_BIND)
C.mknod('/mnt/dev/random', mode(6, 6, 6), 0)
```

```
C.mount('/dev/random', '/mnt/dev/random', nil, C.MS_BIND)
C.mknod('/mnt/dev/tty', mode(6, 6, 6), 0)
C.mount('/dev/tty', '/mnt/dev/tty', nil, C.MS_BIND)
C.mknod('/mnt/dev/urandom', mode(6, 6, 6), 0)
C.mount('/dev/urandom', '/mnt/dev/urandom', nil, C.MS_BIND)
C.mknod('/mnt/dev/zero', mode(6, 6, 6), 0)
C.mount('/dev/zero', '/mnt/dev/zero', nil, C.MS_BIND)
C.mount(nil, '/mnt/dev/pts', 'devpts', 0,
        'newinstance,gid=1000,mode=620,ptmxmode=0666')
C.mount(nil, '/mnt/dev/mqueue', 'mqueue', 0)

local passwd = C.open(
    '/mnt/etc/passwd', bit.bor(C.O_WRONLY, C.O_CREAT), mode(6, 4, 4))
write_all(passwd, 'user:x:1000:1000::/home/user:/bin/sh\n')

C.mkdir('/mnt/home', mode(7, 5, 5))
C.mkdir('/mnt/home/user', mode(7, 5, 5))
C.mount(
    '/var/lib/packxd/{user}/{app_name}/{profile}',
    '/mnt/home/user',
    nil,
    C.MS_BIND)

C.mkdir('/mnt/usr', mode(7, 5, 5))
C.mount('{rootfs}/usr', '/mnt/usr', nil, C.MS_BIND)

{root_symlinks}

-- pivot root
C.mkdir('/mnt/mnt', mode(7, 5, 5))
C.chdir('/mnt')
C.pivot_root('.', '/mnt/mnt')
C.chroot('.')
C.umount2('/mnt', C.MNT_DETACH)

local modulefd = C.open(
    '/app.lua',
    bit.bor(C.O_WRONLY, C.O_CREAT),
    mode(6, 4, 4))
send_with_fd(arg, '.', modulefd)

-- sync point #3 as we must await for
--
-- * loopback net device
-- * `/app.lua`
--
-- before we run the guest
C.read(arg, 1)

C.sethostname('{hostname}')
C.setdomainname('{hostname}')
```

```
C.umask(mode(0, 2, 2))

-- drop all root privileges
C.setgroups({{}})
C.setresgid(1000, 1000, 1000)
C.setresuid(1000, 1000, 1000)

C.setsid()
```

As can be seen, there's a lot of boilerplate that mostly deals with creating files for the container rootfs. A lot of commands just to create a typical FHS structure that applications depend on. It's not difficult in any way, but it's boring code for sure. On the positive side: your container contains a mostly empty `/` which is great for isolating applications!

We also drop privileges at the end. If applications aren't trusted to use user namespaces (`unprivileged_userns_clone`) then they aren't trusted to run as root inside the container.

There's some synchronization at the beginning while we write the UID mapping files that will make more sense once we see the host code:

```
local shost, sguest = unix.seqpacket_socket.pair()
sguest = sguest:release()

local root_symlinks = ''
for linkpath, target in pairs(conf.guests[guest_name].root_symlinks) do
    root_symlinks = root_symlinks ..
        format('C.symlink("{}", "/mnt{}")\n', target, linkpath)
end

local guest_channel = spawn_vm{
    module = '/app.lua',
    subprocess = {
        newns_user = true,
        newns_net = true,
        newns_mount = true,
        newns_pid = true,
        newns_uts = true,
        newns_ipc = true,
        init = {
            arg = sguest,
            script = format(
                INIT_SCRIPT,
                {'root_symlinks', root_symlinks},
                unpack(expansions))
        },
        stdout = 'share',
        stderr = 'share',
        environment = conf.guests[guest_name].environment
    }
```

```
}
scope_cleanup_push(function() guest_channel:close() end)
sguest:close()

local uidmap = file.stream.new()
uidmap:open(
    fs.path.new(format('/proc/{}/uid_map', guest_channel.child_pid)),
    file.open_flag.write_only)
scope_cleanup_push(function() uidmap:close() end)

local setgroups = file.stream.new()
setgroups:open(
    fs.path.new(format('/proc/{}/setgroups', guest_channel.child_pid)),
    file.open_flag.write_only)
scope_cleanup_push(function() setgroups:close() end)

local gidmap = file.stream.new()
gidmap:open(
    fs.path.new(format('/proc/{}/gid_map', guest_channel.child_pid)),
    file.open_flag.write_only)
scope_cleanup_push(function() gidmap:close() end)

local userns = file.stream.new()
userns:open(
    fs.path.new(format('/proc/{}/ns/user', guest_channel.child_pid)),
    file.open_flag.read_only)
userns = userns:release()
scope_cleanup_push(function() userns:close() end)

local netns = file.stream.new()
netns:open(
    fs.path.new(format('/proc/{}/ns/net', guest_channel.child_pid)),
    file.open_flag.read_only)
netns = netns:release()
scope_cleanup_push(function() netns:close() end)

-- Sync point #1
--
-- This sync point makes sure guest_channel.child_pid still is valid and we
-- haven't opened the wrong proc files. We only write to the files after
-- that.
shost:send(IGNORED_BUF)
shost:receive(IGNORED_BUF)

uidmap:write_some(byte_span.append(format('0 0 1\n1000 {} 1\n', uid)))
setgroups:write_some(byte_span.append('allow'))
gidmap:write_some(byte_span.append(format('0 0 1\n1000 {} 1\n', gid)))

-- sync point #2
shost:send(IGNORED_BUF)
```

```
system.spawn{
    program = 'ip',
    arguments = {'ip', 'link', 'set', 'dev', 'lo', 'up'},
    nsenter_user = userns,
    nsenter_net = netns
}:wait()

local module = select(2, shost:receive_with_fds(IGNORED_BUF, 1))[1]
module = file.stream.new(module)
stream.write_all(module, format(GUEST_CODE, unpack(expansions)))

-- sync point #3
shost:close()
```

That's how one sets UID/GID mappings for user namespaces: `/proc` files. Mapping files have the following format:

ID-inside-ns	ID-outside-ns	length

Each line defines a range of IDs to be used. In our case, we only use 1-sized ranges (root-to-root and user/1000):

ID-inside-ns	**ID-outside-ns**	**length**
0	0	1
1000	uid/gid	1

Last we spawn a program from the host to configure the container's network. The code also demonstrates how we can use `system.spawn()` to spawn individual processes in different namespaces. If you don't enter the PID namespace of the container, the container won't even see your process. And if you don't enter the mount namespace of the container, there's no risk on dealing with a compromised rootfs image that lies outside of the TCB.

lwn.net has a good series explaining Linux namespaces in more detail. As you've seen here, the Lua API is easy to deal with.

Once the script finishes, it runs our module (`/app.lua` here) as usual. So that's what we'd need next. However: what are we going to run in our container?

xpra

Xpra is an application that runs X clients on a dedicated X server and direct their display to the local X server without exposing the local X server connection to the X client. Xpra can actually do much more than that as it's a general remote desktop solution, but we're only interested in filtering calls to the X server (in a fashion similar to `ssh -X`). Firejail can also optionally use xpra to filter X calls.

Xpra setup is quite convoluted and took me a lot of trial-and-error to finally have a functional

solution. You'll need extra bind-mounts to `/etc` (`xpra`), a shared `tmpfs` with properly configured permissions (possibly `setfacl`'ed) to create UNIX sockets, symlinks to translate file names that depend on the machine's hostname (they differ between guest and host), and a lot of CLI options. Here's the contents for our `/app.lua`:

```
local ipc_channel = unix.seqpacket_socket.new()
ipc_channel:assign(inbox:receive())
inbox:close()

local xpra_env = {{
    HOME = '/home/user',
    LOGNAME = 'user',
    USER = 'user',
    RUNTIME_DIRECTORY = '/run/user/1000',
    XDG_RUNTIME_DIR = '/run/user/1000'
}}
for k, v in pairs(system.environment) do
    xpra_env[k] = v
end

local p = system.spawn{{
    program = 'xpra',
    arguments = {{
        'xpra',
        'start',
        '--daemon=no',
        '--mdns=no',
        '--dbus-proxy=no',
        '--dbus-launch=no',
        '--notifications=no',
        '--bind=/home/user/.xpra/xpra',
        '--speaker=no',
        '--use-display=no',
        '--xsettings=no',
        '--forward-xdg-open=no',
        '--terminate-children=yes',
        '--exit-with-children=yes',
        '--env=QT_QUICK_BACKEND=software',
        '--start-child={app_bin}',
        '--mmap=/home/user/.xpra/mmap/xpra'
    }},
    environment = xpra_env,
    stdout = 'share',
    stderr = 'share',
    working_directory = fs.path.new('/home/user')
}}

local waiter = spawn(function()
    pcall(function()
        local buf = byte_span.new(1)
```

```
        ipc_channel:receive(buf)
    end)
    if not p.pid then
        return
    end
    p:kill(system.signal.SIGTERM)
    print('SIGTERM sent')
    time.sleep(10)
    if not p.pid then
        return
    end
    print('SIGTERM timeout reached... sending SIGKILL')
    p:kill(system.signal.SIGKILL)
end)

p:wait()
waiter:interrupt()
waiter:detach()
```

We make use of Emilua's fiber primitives to synchronize different tasks that might signal us to terminate the container (`xpra` exited on its own or the client `ipc_channel` disconnected). After `init.script` finishes, the Emilua runtime automatically detects whether it's running as PID1 (i.e. a new PID namespace) and takes care of many boilerplate tasks for us (e.g. orphaned processes, and signal forwarding). Once the actor `/app.lua` finishes, PID1 also exits (shutting down the whole container in the process).

Our xpra instance assumes a lot of directories that are created before we start the container:

```
system.setresgid(-1, gid, -1)
system.setresuid(-1, uid, -1)

local runtime_path = fs.path.new(format(
    '/run/user/{user}/packxd/{app_name}/{profile}', unpack(expansions)))
fs.create_directories(runtime_path / 'xpra/mmap')
fs.chmod(runtime_path / 'xpra', fs.mode(7, 0, 0))
fs.chmod(runtime_path / 'xpra/mmap', fs.mode(7, 0, 0))
```

Given that we need to change global process properties (`setresuid()`) and we're running a concurrent server that might at anytime handle different clients, we run this code in a separate process to not meddle with shared global properties from the server process:

```
spawn_vm{
    module = tostring(_FILE.parent_path / 'worker_create_userrundirs.lua'),
    subprocess = {
        stdout = 'share',
        stderr = 'share'
    }
```

```
}
```

And there's also a lot of extra bind-mounts to `init.script` that I'll omit here. Now to the client:

```
local arguments = system.arguments
table.remove(arguments, 2)
table.remove(arguments, 1)

if #arguments ~= 4 then
    print('Syntax:')
    print('<guest_name> <app_name> <profile> <app_bin>')
    system.exit(1)
end

local SPAWN_LINUX_APP_ARGS = {
    guest_name = arguments[1],
    app_name = arguments[2],
    profile = arguments[3],
    app_bin = arguments[4]
}

local SOCK_PATH = fs.path.new('/var/run/packxd.socket')

local runtime_path = fs.path.new(format(
    '/run/user/{user}/packxd/{app_name}/{profile}',
    {'user', select(2, system.getresuid()), nil},
    {'app_name', SPAWN_LINUX_APP_ARGS.app_name},
    {'profile', SPAWN_LINUX_APP_ARGS.profile}))

local sock = unix.seqpacket_socket.new()
sock:connect(SOCK_PATH)
local buf = byte_span.append(json.encode {
    request = 'spawn_linux_app',
    arguments = SPAWN_LINUX_APP_ARGS
})
sock:send(buf)

local buf = byte_span.new(1024)
local nread, fds = sock:receive_with_fds(buf, 1)
local reply = json.decode(tostring(buf:slice(1, nread)))
if reply.result ~= 'ok' then
    print('Failed: ' .. reply.error)
    system.exit(1)
end

local ipc_control = unix.seqpacket_socket.new()
ipc_control:assign(fds[1])

spawn(function()
    pcall(function()
```

```
        local buf = byte_span.new(1)
        ipc_control:receive(buf)
    end)
    system.exit(0)
end):detach()

local sigset = system.signal.set.new(
    system.signal.SIGTERM, system.signal.SIGINT)
spawn(function()
    sigset:wait()
    system.exit(0)
end):detach()

while true do
    local p = system.spawn{
        program = 'xpra',
        arguments = {
            'xpra',
            'attach',
            tostring(runtime_path / 'xpra/xpra')
            '--opengl=no',
            '--session-name=' .. SPAWN_LINUX_APP_ARGS.app_name .. '/'
                .. SPAWN_LINUX_APP_ARGS.profile,
            '--mmap=' .. tostring(runtime_path / 'xpra/mmap/xpra')
        },
        stdout = 'share',
        stderr = 'share',
        environment = system.environment,
        pdeathsig = system.signal.SIGTERM
    }
    p:wait()
    if p.exit_code == 0 then
        system.exit(0)
    end
    if p.exit_signal then
        print('xpra-attach killed by signal', p.exit_signal)
        system.exit(0)
    end
    time.sleep(1)
end
```

We need to loop `xpra-attach` until the server socket is ready. Then we observe a few conditions that should signal us to stop the loop and exit the process (e.g. `ipc_control` will close when the container dies).

The final version actually has more logic to create and share PulseAudio UNIX sockets (possibly through PipeWire proxies). However this article is already getting too big and I'll omit `pdeathsig` tricks used in the PulseAudio proxies. You can still see these tricks in the source code repository.

Configuring the network

The solution is somewhat interesting already, but pretty limited in usefulness without an usable network namespace. To configure a simple NAT traversal network, you'll need to run these commands in the host:

```
iptables -t nat -A POSTROUTING -o <external interface such as eth0 or wlan0> -j
MASQUERADE
sysctl net.ipv4.ip_forward=1
```

Nftables users

If you use nftables instead of iptables, make sure the table nat exists:

```
nft add table ip nat
nft add chain ip nat postrouting "{ type nat hook postrouting priority
srcnat; }"
```

Then add the NAT rule:

```
nft add rule ip nat postrouting oifname <external interface such as
eth0 or wlan0> masquerade
```

In our program, we need to create a veth pair, move one end to the network namespace of the container, and configure the veth end that stays in the host network namespace. Thankfully we have root privileges and can do all of these steps. However we need to avoid IP address clashes for network interfaces so we create semi-persistent network namespaces using `ip-netns`. `ip-netns` will create namespaces bind-mounts in `/var/run/netns`. Here's the code for the whole setup:

```
local netns_name = 'packxd-' .. guest_name .. '-0'
local p = system.spawn{
    program = 'ip',
    arguments = {'ip', 'netns', 'add', netns_name},
}
p:wait()
if p.exit_code == 0 then
    local guest_address = conf.guests[guest_name].veth.ipv4.address
    local gateway_address = conf.guests[guest_name].veth.ipv4.gateway
    local netmask = regex.match(NETMASKREGEX, guest_address)

    system.spawn{
        program = 'ip',
        arguments = {
            'ip', 'netns', 'exec', netns_name,
            'ip', 'link', 'set', 'dev', 'lo', 'up'}
    }:wait()
```

```
    local veth_name0 = conf.guests[guest_name].veth.ifname

    scope(function()
        veth_mtx:lock()
        scope_cleanup_push(function() veth_mtx:unlock() end)

        system.spawn{
            program = 'ip',
            arguments = {
                'ip', 'link', 'add',
                veth_name0, 'type', 'veth', 'peer', 'name', '_packxd'}
        }:wait()

        system.spawn{
            program = 'ip',
            arguments = {
                'ip', 'link', 'set', '_packxd', 'netns', netns_name}
        }:wait()
    end)

    system.spawn{
        program = 'ip',
        arguments = {
            'ip', 'netns', 'exec', netns_name,
            'ip', 'link', 'set', 'dev', '_packxd', 'name', 'eth0'}
    }:wait()

    system.spawn{
        program = 'ip',
        arguments = {
            'ip', 'netns', 'exec', netns_name,
            'ip', 'address', 'add', guest_address, 'dev', 'eth0'}
    }:wait()

    system.spawn{
        program = 'ip',
        arguments = {
            'ip', 'netns', 'exec', netns_name,
            'ip', 'link', 'set', 'dev', 'eth0', 'up'}
    }:wait()

    system.spawn{
        program = 'ip',
        arguments = {
            'ip', 'netns', 'exec', netns_name,
            'ip', 'route', 'add', 'default', 'via', gateway_address}
    }:wait()

    system.spawn{
        program = 'ip',
```

```
        arguments = {
            'ip', 'address', 'add',
            gateway_address .. netmask, 'dev', veth_name0}
    }:wait()

    system.spawn{
        program = 'ip',
        arguments = {'ip', 'link', 'set', 'dev', veth_name0, 'up'}
    }:wait()
end
```

We use the interface name `_packxd` for a temporary interface in the host system. To avoid clashes with other fibers trying to create the same interface, we use a simple mutex.

However now we need to put our container into this namespace. The container's user namespace has no capabilities under this network namespace, so we instead spawn an actor that shares our root privileges just to enter this namespace briefly before creating our actual container:

```
local netns = file.stream.new()
netns:open(
    fs.path.new('/var/run/netns/' .. netns_name),
    file.open_flag.read_only)
netns = netns:release()
scope_cleanup_push(function() netns:close() end)

local guest_channel = spawn_vm{
    module = tostring(_FILE.parent_path / 'worker_nsenternet.lua'),
    subprocess = {
        init = {
            arg = netns,
            script = 'C.setns(arg, C.CLONE_NEWNET)'
        },
        stdout = 'share',
        stderr = 'share'
    }
}
```

Up-to-date repo

You may find the code for this project at https://gitlab.com/emilua/packxd. To use the project, start the daemon as root (remember to write the configuration file at `/etc/packxd.json` first):

```
emilua packxd/init.lua
```

Then use the client to run desktop applications from LXC containers:

```
emilua packxd-client/init.lua -- mycontainer01 firefox test firefox
```

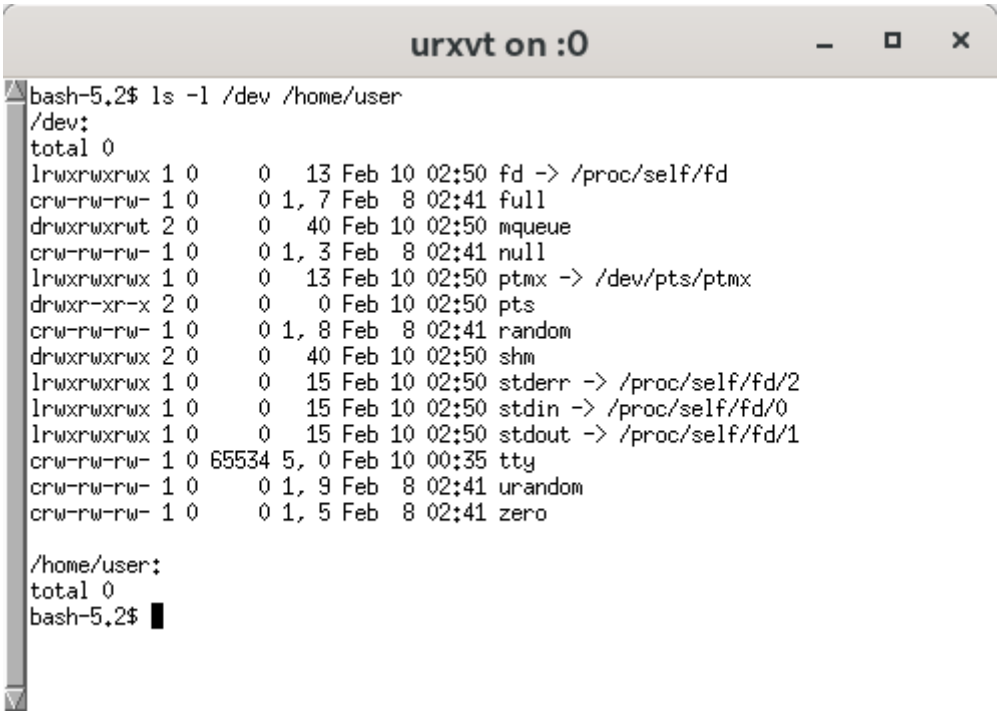


Figure 1. rxvt running inside our container

If you use / as the rootfs for the container, you can run host programs inside containers. This option presents itself as an alternative to Firejail in some circumstances.

The repository already contains several features not discussed in this blog post (e.g. systemD daemon readiness notification, xpra-server pipe redirections).