Debugging Using Container Technology

Pavel Šimerda

pavlix@pavlix.net
Everyday Debugging Tools
Tracing System Calls with `strace`

$ sudo strace -e socket ping -n prgcont.cz
socket(AF_INET, SOCK_DGRAM, IPPROTO_ICMP) = -1 EACCES (Permission denied)
socket(AF_INET, SOCK_RAW, IPPROTO_ICMP) = 3
socket(AF_INET6, SOCK_DGRAM, IPPROTO_ICMPV6) = -1 EACCES (Permission denied)
socket(AF_INET6, SOCK_RAW, IPPROTO_ICMPV6) = 4
socket(AF_UNIX, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, 0) = 5
socket(AF_UNIX, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, 0) = 5
socket(AF_INET, SOCK_DGRAM|SOCK_CLOEXEC|SOCK_NONBLOCK, IPPROTO_IP) = 5
socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE) = 5
socket(AF_INET6, SOCK_DGRAM|SOCK_CLOEXEC, IPPROTO_IP) = 5
socket(AF_INET, SOCK_DGRAM, IPPROTO_IP) = 5

What is a syscall by the way?
$ sudo ltrace -e getaddrinfo ping -n prgcont.cz
ping->getaddrinfo(
    "prgcont.cz", nil, 0x7ffec74ab4f0, 0x7ffec74ab4d8) = 0

What is the mechanism behind library calls and how can we intercept them?
Tracing with systemtap

stap resolver.stp -c "ping -n prgcont.cz"

resolver.stp:

probe process("/lib64/libc-2.27.so").function("getaddrinfo") {
    printf("%s: %s\n", probefunc(), user_string($name));
}

What’s the difference between systemtap mechanism and strace/ltrace mechanism?
GNU Debugger Sessions

$ sudo gdb --args ping -n prgcont.cz
(gdb) break getaddrinfo
(gdb) run

What are the most commonly used debugger features?

How to install debugging information on various distributions?
Automated Debugging Scripts

$ sudo gdb -x script.gdb --args ping -n prgcont.cz

script.gdb:

start
advance getaddrinfo
backtrace
print name
print service
print *hints
kill
quit
$ sudo gdb -x script.gdb --args ping -n prgcont.cz

script.gdb:

start
advance getaddrinfo
set name = "1.1.1.1"
cont
$ sudo \
    strace -e ptrace -o trace.out \ 
    strace -e open ls -l /proc/self
lrwxrwxrwx 1 root root 0 Oct 21 17:18 /proc/self -> 30895

$ grep 'ptrace(.*, 30895,' trace.out
ptrace(PTRACE_SEIZE, 30895, NULL, 
       PTRACE_O_TRACESYSGOOD|PTRACE_O_TRACEEXEC|PTRACE_O_TRACEFULL) = 0
ptrace(PTRACE_SYSCALL, 30895, NULL, SIG_0) = 0
ptrace(PTRACE_GETSIGINFO, 30895, NULL, 
       {si_signo=SIGCONT, si_code=SI_USER, si_pid=30893, si_uid=0}) = 0
ptrace(PTRACE_SYSCALL, 30895, NULL, SIGCONT) = 0
ptrace(PTRACE_GETREGSET, 30895, NT_PRSTATUS, 
       [{iov_base=0x562f887753c0, iov_len=216}]) = 0
ptrace(PTRACE_SYSCALL, 30895, NULL, SIG_0) = 0
...
Asking your Best Friend
Containerize It!
Objectives

- Freedom to automate any conceivable test scenario
- Scalable performance for a large number of tests
- Independence from the test runner operating system
- Virtualized network configuration for tests
- Ideally also guard the bare metal system from test runner bugs
Why don’t you just use...
Qemu or even LNST?

- Rather hard to drive tested processes across instances
- Too slow to setup and teardown full instances
- Inconsistent cleanup of precreated instances might spoil the tests
- LNST pollutes the network devices with its own communication channels
- Automation of Qemu and non-network communication channels is rather complex

Would you be willing to write all the boilerplate to handle the communication and remote debugging?
Docker, LXC or systemd-nspawn?

- Doesn’t solve most of the virtual machine concerns either
- Might still be suboptimal to setup and teardown
- Some boilerplate still needed for communication and remote debugging

Do we really need the features provided by those tools?
So what do we need?
Unshare the File System

- Use `unshare()` system call to create new mount namespace
- Use the `CLONE_NEWNS` flag to achieve that
- Override the (systemd) default `MS_SHARED` recursively on the whole file system tree
- Use `MS_PRIVATE` to detach from original namespace entirely
- Use `MS_SLAVE` to keep one way propagation from original namespace to the new one
Unshare the File System (C pseudocode)

```c
int status;

status = unshare(CLONE_NEWNS);
status = mount("none", "/", NULL, MS_REC | MS_SLAVE, NULL);

Check the status, Luke!
```
Intermezzo: Choosing the programming language
Stay with C?

- Cool for understanding Linux internals
- Not so good for test automation
- Not so easy with complex data structures and reporting
Do it in shell!

- Cool for simple automation but probably not for complex cases
- Heavily based on external commands and processes
- We just need to call the right syscalls without forking
Looking for a dynamic language with easy text and data manipulation
But also easily integrated with the shared libraries and syscalls
I have some knowledge of Python
I have already used Python for exactly this class of tasks
The interactive Python interpreter (a REPL loop) makes it easy to test stuff
So does fast automatic compilation of modules
Switching all error handling to exceptions helps greatly
def mount_unshare():
    unshare(CLONE_NEWNS)
    return mount(
        "none",
        "/",
        None,
        MS_REC | MS_SLAVE,
        None)
The unshare and mount functions

```python
def unshare(flags):
    return _check(libc.unshare(flags))

def mount(source, target, fstype, flags, data):
    return _check(libc.mount(_encode(source),
                              _encode(target),
                              _encode(fstype),
                              flags,
                              _encode(data)))
```
The library functions

```python
ffi = cffi.FFI()
ffi.cdef(""
int unshare(int flags);
int mount(const char *source, const char *target,
    const char *filesystemtype, unsigned long mountflags,
    const void *data);
int umount2(const char *target, int flags);
int pivot_root(const char *new_root, const char *put_old);
"")
libc = ffi.dlopen("libc.so.6")
```

Note: You also need to look at the header files and define all the necessary symbolic constants as Python variables.
The ugly stuff

```python
def _check(status):
    if status == -1:
        raise OSError(ffi.errno, os.strerror(ffi.errno))
    return status

def _encode(name):
    if isinstance(name, str):
        return name.encode("utf-8")
    elif name is None:
        return ffi.NULL
    else:
        return name

How do we use the unshared mount namespace now?
Switching to an overlay over current root

```python
tmp = "/run/pycoz"
mount_tmpfs(tmp)

new_root = "/run/pycoz/root"
mount_overlay(
    target=new_root,
    lowerdir=base,
    upperdir="/run/pycoz/upper",
    workdir="/run/pycoz/work")

old_root = "/oldroot"
pivot_root(new_root, new_root + old_root)
```

Instead of the current root (a hacky way) you can use a full distribution root filesystem image or directory.
How tmpfs gets mounted

def mount_tmpfs(target, *, makedirs=True):
    if makedirs:
        _mkdir(target)
    return mount(
        "none",
        target,
        "tmpfs",
        0,
        """
How overlayfs gets mounted

```python
def mount_overlay(target, *,
    lowerdir="/",
    upperdir,
    workdir,
    flags=MS_DEFAULTS,
    makedirs=True):
    if makedirs:
        _mkdir(target)
        _mkdir(lowerdir)
        _mkdir(upperdir)
        _mkdir(workdir)
    return mount(
        "overlay",
        target,
        "overlay",
        flags,
        f"lowerdir={lowerdir},upperdir={upperdir},workdir={workdir}"
    )
```

Now we know how to create a working overlayfs root, so let’s look
Network namespace with virtual ethernet

```python
ip = pyroute2.IPDB()
ip.create(kind="veth", ifname="pycoz0", peer="pycoz1")
ip.commit()

ns = pyroute2.IPDB(nl=pyroute2.NetNS("pycoz"))

with ip.interfaces.pycoz0 as veth:
    veth.net_ns_fd = "pycoz"
with ns.interfaces.pycoz0 as veth:
    veth.add_ip("192.168.0.1/24")
    veth.up()
with ip.interfaces.pycoz1 as veth:
    veth.add_ip("192.168.0.2/24")
    veth.up()

Thanks Peter V. Savaliev for creating the pyroute2 package!
```
Back to debugging
containers.pivot_temporary_overlay()
containers.netns_with_veth()

command = ['ping', '-n', '192.168.0.2']

debugger = ptrace.debugger.PtraceDebugger()
process = debugger.addProcess(ptrace.debugger.child.createChild(command, False), True)

Thanks Victor Stinner for creating python-ptrace package!
while True:
    process.syscall()
    event = debugger.waitProcessEvent()
    if isinstance(event, ptrace.debugger.ProcessExit):
        break
    elif isinstance(event, ptrace.debugger.ProcessSignal):
        print(process.syscall_state.event(ptrace.func_call.FunctionCallOptions()))
        debugger.quit()

Note: The code is simplified.
https://github.com/crossdistro/container-debug-example

container.py, debug.py

pavlix@pavlix.net

prgcont.cz