

Discussion Questions

1. What is a kernel function? What is a system call?

A kernel function allows us to interact directly with the kernel. The kernel allows us to write functions that implement some of the kernel utilities on the system. For example, if we want our program to make a new directory, we can use the provided kernel functions on our operating system of choice. These functions will then execute the required system calls.

A system call requests something directly from the kernel. However, a kernel function might just be a wrapper around the system call that directly executes the responsible code.

2. what is KASLR? What is it for?

KASLR stands for **K**ernel **A**ddress **S**pace **L**ayout **R**andomizer. This utility will load the kernel in a random place in memory during boot time. If the kernel code were loaded in the same memory location every time, then we could exploit the location of the kernel functions for some nefarious use. We would just need to know the code structure and then find a way to insert malicious code into that address space. By loading the kernel at a random location in memory, an attack of this sort is made much harder.

We can turn this setting off during boot time by using the `nokaslr` option.

3. What are GDB's non-stop and all-stop modes?

In GDB, the all-stop and non-stop modes refer to how the program stops execution. In the former, *all* the currently executing threads stop. This allows us to view the entire state of the program at a certain point. The latter mode refers to only stopping certain threads while allowing other currently executing threads to continue.

One might be more useful for isolating the behavior of a single thread, while the other might be more useful for viewing the entire state of the program at a given point.

4. Explain what the command `echo g > /proc/sysrq-trigger` does.

The `/proc/sysrq-trigger` file allows us to issue instructions directly to the kernel. In the Linux file system, the `proc` directory contains information about the currently executing processes. The file `sysrq-trigger` triggers something to happen in the kernel. The `g` that we echo into the file is

specifically used by kgdb. This is why kgdb regains control after we echo it into this file. Besides this, we can also crash the system or immediately restart the system.

Questions From Do It Yourself 2

5. Perf also has the report command. Explain:

- What is it for?
- For fileCopyTest, show and interpret the results.

The `perf` program measures the performance of commands on Linux. We can get some other statistics on how our program is performing, and a complete trace of the system calls that are executing.

Your Screenshot Here

6. Perf has more commands. Select another command (besides report, trace and record), and explain what it is for and show how to use it.

If we look at the documentation for `perf`, we see that there are many commands available.

```
perf
usage: perf [--version] [--help] COMMAND [ARGS]

The most commonly used perf commands are:
annotate      Read perf.data (created by perf record) and display annotated code
archive       Create archive with object files with build-ids found in perf.data file
bench         General framework for benchmark suites
buildid-cache Manage <tt>build-id</tt> cache.
buildid-list  List the buildids in a perf.data file
diff          Read two perf.data files and display the differential profile
inject        Filter to augment the events stream with additional information
kmem          Tool to trace/measure kernel memory(slab) properties
kvm           Tool to trace/measure kvm guest os
list          List all symbolic event types
lock          Analyze lock events
probe         Define new dynamic tracepoints
record        Run a command and record its profile into perf.data
report        Read perf.data (created by perf record) and display the profile
sched         Tool to trace/measure scheduler properties (latencies)
script        Read perf.data (created by perf record) and display trace output
stat          Run a command and gather performance counter statistics
test          Runs sanity tests.
timechart     Tool to visualize total system behavior during a workload
top           System profiling tool.

See 'perf help COMMAND' for more information on a specific command.
```

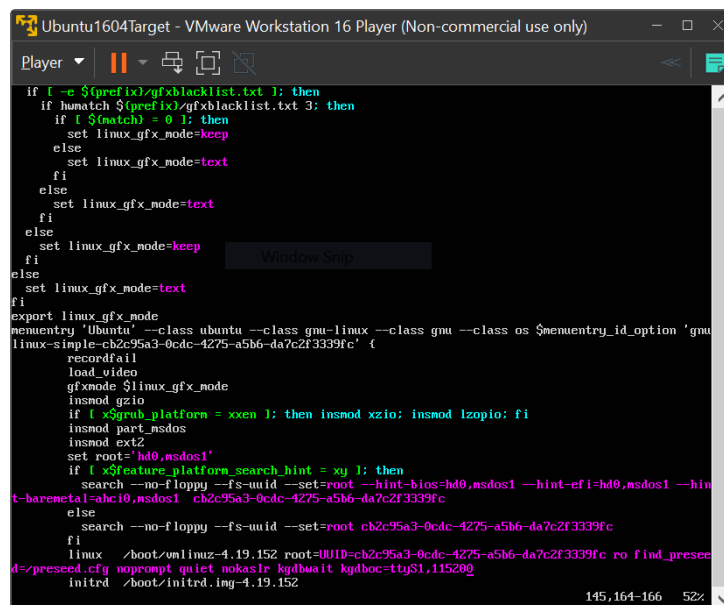
Figure 1: Some `perf` commands.

For example, we have the `perf stat` command, which will gather performance counter statistics on our program. To use it we can run some shell command or execute some program, and it can relay information such as memory usage, cache misses, and other potentially useful information. To run it we just type

```
1 sudo perf stat [OPTIONS] command
```

where *command* is a shell command.

Screenshot Discussion



```
if [ -e $(prefix)/gfxblacklist.txt ]; then
if [ ! $(cat $(prefix)/gfxblacklist.txt 3) ]; then
if [ $(match) = 0 ]; then
set linux_gfx_mode=keep
else
set linux_gfx_mode=text
fi
else
set linux_gfx_mode=text
fi
else
set linux_gfx_mode=keep
fi
else
set linux_gfx_mode=text
fi
export linux_gfx_mode
menumenuentry 'Ubuntu' --class ubuntu --class gnu-linux --class gnu --class os $menuentry_id_option 'gnulinux-simple-cb2c95a3-0cdc-4275-a5b6-da7c2f3339fc' {
recordfail
load_video
gfxmode $linux_gfx_mode
insmod gzio
if [ x$grub_platform = xxen ]; then insmod xzio; insmod lzopio; fi
insmod part_msdos
insmod ext2
set root='hd0,msdos1'
if [ x$feature_platform_search_hint = xy ]; then
search --no-floppy --fs-uuid --set=root --hint-bios=hd0,msdos1 --hint-efi=hd0,msdos1 --hint-baremetal=ahci0,msdos1 cb2c95a3-0cdc-4275-a5b6-da7c2f3339fc
else
search --no-floppy --fs-uuid --set=root cb2c95a3-0cdc-4275-a5b6-da7c2f3339fc
fi
linux /boot/vmlinuz-4.19.152 root=UUID=cb2c95a3-0cdc-4275-a5b6-da7c2f3339fc ro find_preseed=/preseed.cfg noprompt quiet nokaslr kgdbwait kgdboc=ttyS1,115200
initrd /boot/initrd.img-4.19.152
```

Figure 2: Updating Grub

The first step in our homework assignment is to update the GRUB. We disable KASLR, which loads the kernel at random locations in memory to avoid potential attacks on the kernel's memory space. Since this is enabled by default, we have to adjust it on the kernel's command line parameters. After we do so, we have to run the `sudo update-grub` command, which generates `grub.cfg`. However, we have to add again our own kernel parameters which we added in 1.A. These parameters are the `kgdbwait` and the `kgdboc=ttyS1,115200`.

Ubuntu1604Target - VMware Workstation 16 Player (Non-commercial use only)

Player

```

# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# The abi is "common", "64" or "x32" for this file.
#
0    common read          __x64_sys_read
1    common write         __x64_sys_write
2    common open          __x64_sys_open
3    common close         __x64_sys_close
4    common stat          __x64_sys_newstat
5    common fstat         __x64_sys_newfstat
6    common lstat         __x64_sys_newlstat
7    common poll          __x64_sys_poll
8    common lseek         __x64_sys_lseek
9    common mmap          __x64_sys_mmap
10   common mprotect      __x64_sys_mprotect
11   common munmap        __x64_sys_munmap
12   common brk           __x64_sys_brk
13   64 rt_sigaction       __x64_sys_rt_sigaction
14   common rt_sigprocmask __x64_sys_rt_sigprocmask
15   64 rt_sigreturn       __x64_sys_rt_sigreturn/ptregs
16   64 ioctl             __x64_sys_ioctl
17   common pread64        __x64_sys_pread64
18   common pwrite64       __x64_sys_pwrite64
19   64 readv             __x64_sys_readv
20   64 writev            __x64_sys_writev
21   common access        __x64_sys_access
22   common pipe          __x64_sys_pipe
23   common select        __x64_sys_select
24   common sched_yield    __x64_sys_sched_yield
25   common mremap        __x64_sys_mremap
"syscall_64.tbl" [readonly] 388L, 15659C
1.1 Top

```

Figure 3: Syscall Table

→ ↻ ↗ shell-storm.org/shellcode/files/syscalls.html

22	sys_oldumount	fs/super.c	char *	-	-	-	-
23	sys_setuid	kernel/sys.c	uid_t	-	-	-	-
24	sys_getuid	kernel/sched.c	-	-	-	-	-
25	sys_stime	kernel/time.c	int *	-	-	-	-
26	sys_ptrace	arch/386/kernel/ptrace.c	long	long	long	long	-
27	sys_alarm	kernel/sched.c	unsigned int	-	-	-	-
28	sys_fstat	fs/stat.c	unsigned int	struct __old_kernel_stat *	-	-	-
29	sys_pause	arch/386/kernel/sys_i386.c	-	-	-	-	-
30	sys_utime	fs/open.c	char *	struct utimbuf *	-	-	-
33	sys_access	fs/open.c	const char *	int	-	-	-
34	sys_nice	kernel/sched.c	int	-	-	-	-
36	sys_sync	fs/buffer.c	-	-	-	-	-
37	sys_kill	kernel/signal.c	int	int	-	-	-
38	sys_rename	fs/namei.c	const char *	const char *	-	-	-
39	sys_mkdir	fs/namei.c	const char *	int	-	-	-
40	sys_rmdir	fs/namei.c	const char *	-	-	-	-
41	sys_dup	fs/fcntl.c	unsigned int	-	-	-	-
42	sys_pipe	arch/386/kernel/sys_i386.c	unsigned long *	-	-	-	-

Figure 4: Online reference

→ ↻ ↗ https://faculty.nps.edu/cse/assembly/sys_call.html

25	sys_stime	kernel/time.c	int *	-	-	-	-
26	sys_ptrace	arch/386/kernel/ptrace.c	long	long	long	long	-
27	sys_alarm	kernel/sched.c	unsigned int	-	-	-	-
28	sys_fstat	fs/stat.c	unsigned int	struct __old_kernel_stat *	-	-	-
29	sys_pause	arch/386/kernel/sys_i386.c	-	-	-	-	-
30	sys_utime	fs/open.c	char *	struct utimbuf *	-	-	-
33	sys_access	fs/open.c	const char *	int	-	-	-
34	sys_nice	kernel/sched.c	int	-	-	-	-
36	sys_sync	fs/buffer.c	-	-	-	-	-
37	sys_kill	kernel/signal.c	int	int	-	-	-
38	sys_rename	fs/namei.c	const char *	const char *	-	-	-
39	sys_mkdir	fs/namei.c	const char *	int	-	-	-
40	sys_rmdir	fs/namei.c	const char *	-	-	-	-
41	sys_dup	fs/fcntl.c	unsigned int	-	-	-	-
42	sys_pipe	arch/386/kernel/sys_i386.c	unsigned long *	-	-	-	-
43	sys_times	kernel/sys.c	struct tms *	-	-	-	-
46	sys_setgid	kernel/sys.c	gid_t	-	-	-	-
47	sys_getgid	kernel/sched.c	-	-	-	-	-

Figure 5: Online reference 2

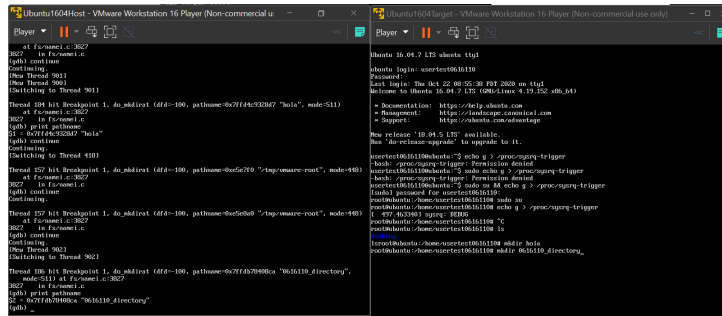


Figure 8: Pathname parameter of `mkdir`

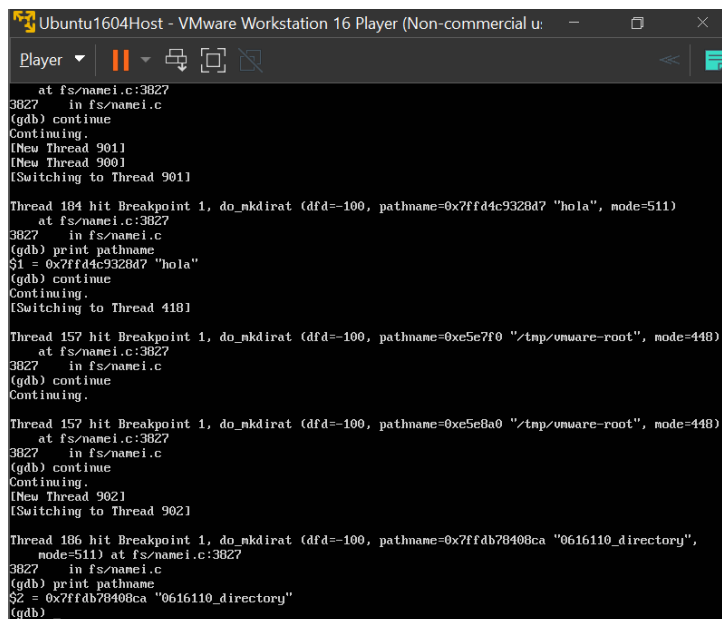


Figure 9: Host machine showing newly created directory

The next pair of screenshots show the process of debugging the kernel `mkdir` function. After looking up the system call definition in `fs/namei.c`, we created a breakpoint at the corresponding system function `do_mkdirat`. Then, we test it by creating the `0616110_directory`. The output is shown in the second screenshot.

Do It Yourself 1

For the first DIY section of the assignment, I chose the common `rmdir` function. The reason I chose this function is because the debugging process was quite similar. From the implementation, it was

also quite similar to the `mkdir` command. It was also being implemented in the `fs/namei.c` file, and the naming convention of the system call was also quite similar.

```

Player ▾  || ▾  ↵  ⌂  ✕
pathname=0x55b304c16b40 "/sys/fs/cgroup/systemd/system.slice/ondemand.service")
at fs/namei.c:3901
3901 fs/namei.c: No such file or directory.
(gdb) continue
Continuing.

Thread 2 hit Breakpoint 1, do_rmdir (dfd=-100,
    pathname=0x55b304c16b40 "/sys/fs/cgroup/devices/system.slice/ondemand.service")
    at fs/namei.c:3901
3901 in fs/namei.c
(gdb) continue
Continuing.
[New Thread 902]
[New Thread 901]

Thread 214 received signal SIGTRAP, Trace/breakpoint trap.
[Switching to Thread 902]
kgdb_breakpoint () at kernel/debug/debug_core.c:1086
1086 kernel/debug/debug_core.c: No such file or directory.
(gdb) continue
Continuing.
[New Thread 904]
[Switching to Thread 904]

Thread 216 hit Breakpoint 1, do_rmdir (dfd=-100, pathname=0x22ef0c0 "0616116_dir/")
    at fs/namei.c:3901
3901 fs/namei.c: No such file or directory.
(gdb) continue
Continuing.
[New Thread 907]
[New Thread 905]
[Switching to Thread 907]

Thread 217 hit Breakpoint 1, do_rmdir (dfd=-100, pathname=0x10a00c0 "0616110_directory/")
    at fs/namei.c:3901
3901 in fs/namei.c
(gdb) -

```

Figure 10: Image On And Responsive

In the first image, the gdb program already created the breakpoint at the `rmkdir` execution, which is called by the `do_rmkdir(...)` function call in the `fs/namei.c` file. We see the trigger caused by that function in the image, where I just typed `continue` to reclaim control of the target machine.

The way we trigger this breakpoint is similar to the one in the given example. In our Target machine, we remove a directory using the `rmdir` command in the terminal and this syscall is then invoked. The `do_rmdir()` function gets called in the kernel which triggers our gdb instance to emit some information.

```

goto exit3;
error = vfs_rmdir(path.dentry->d_inode, dentry);
exit3:
dput(dentry);
exit2:
inode_unlock(path.dentry->d_inode);
mnt_drop_write(path.mnt);
exit1:
path_put(&path);
putname(name);
if (retry_estale(error, lookup_flags)) {
    lookup_flags |= LOOKUP_REVAL;
    goto retry;
}
return error;
}

SYSCALL_DEFINE1(rmdir, const char __user *, pathname)
{
    return do_rmdir(AT_FDCWD, pathname);
}

/**
 * vfs_unlink - unlink a filesystem object
 * @dir:      parent directory
 * @dentry:   victim
 * @delegated_inode: returns victim inode, if the inode is delegated.
 *
 * The caller must hold dir->i_mutex.
 *
 * If vfs_unlink discovers a delegation, it will return -EWOULDBLOCK and
 * return a reference to the inode in delegated_inode. The caller
 * should then break the delegation on that inode and retry. Because
 * breaking a delegation may take a long time, the caller should drop
 * dir->i_mutex before doing so.
 */

```

Figure 11: Creating the Breakpoint

For the breakpoint, we use the **break** `do_rmdir` command in GDB so that the Target machine stops executing. In the file where the syscall is defined in the kernel (`fs/namei.c`) in this case, we see the actual C function being called, the `do_rmdir(...)` function. We then tell GDB to make the breakpoint when this function executes.

```

(New Thread 904)
[Switching to Thread 904]

Thread 216 hit Breakpoint 1, do_rmdir (dfd=-100, pathname=0x22ef0c0 "0616116_dir/")
at fs/namei.c:3901
3901 fs/namei.c: No such file or directory.
(gdb) continue
Continuing.
(New Thread 907)
[New Thread 905]
[Switching to Thread 907]

Thread 217 hit Breakpoint 1, do_rmdir (dfd=-100, pathname=0x10a00c0 "0616110_directory/")
at fs/namei.c:3901
3901 in fs/namei.c
(gdb) continue
Continuing.
(New Thread 908)
[New Thread 909]
[New Thread 921]
[Switching to Thread 1]

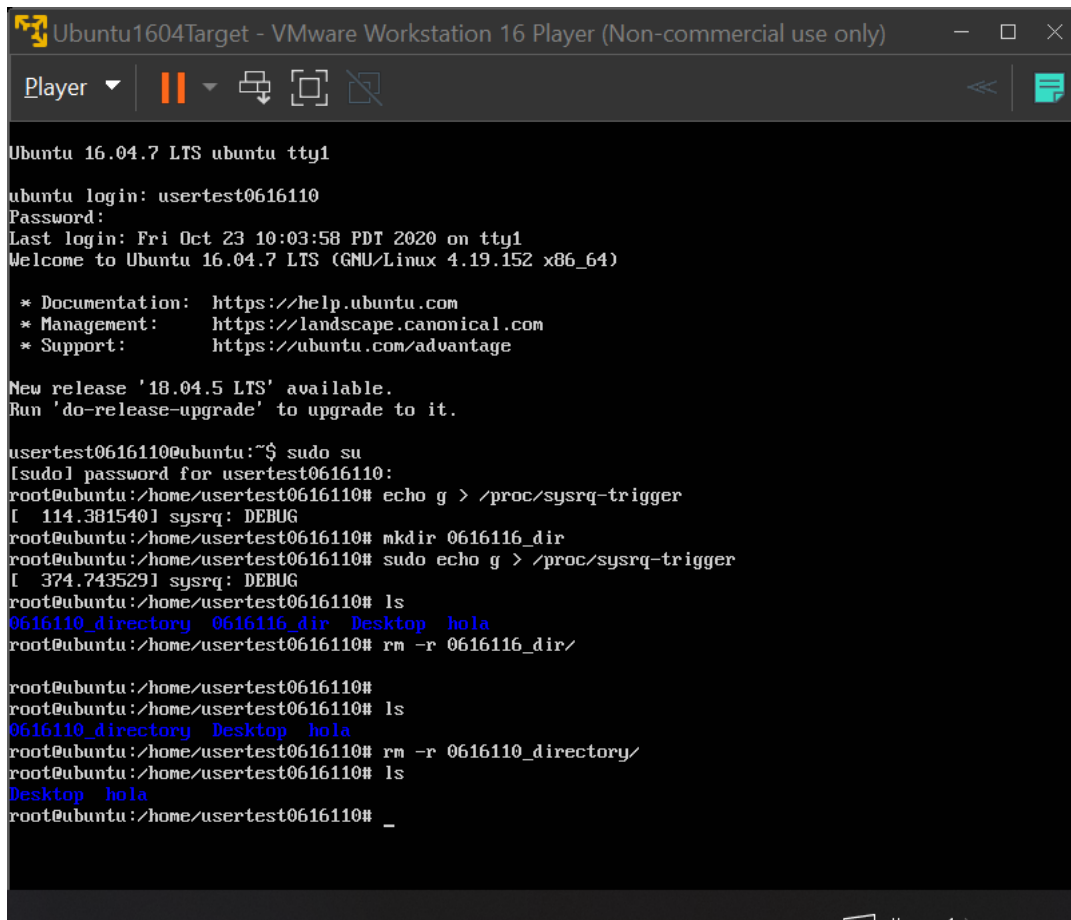
Thread 2 hit Breakpoint 1, do_rmdir (dfd=-100,
pathname=0x55b304c007d0 "/sys/fs/cgroup/systemd/system.slice/systemd-tmpfiles-clean.service")
at fs/namei.c:3901
3901 in fs/namei.c
(gdb) continue
Continuing.
(New Thread 922)

Thread 2 hit Breakpoint 1, do_rmdir (dfd=-100,
pathname=0x55b304c007d0 "/sys/fs/cgroup/devices/system.slice/systemd-tmpfiles-clean.service")
at fs/namei.c:3901
3901 in fs/namei.c
(gdb) continue
Continuing.

```

Figure 12: How to trigger

After we create the break point in GDB, we actually go ahead and execute the trigger. The result in the above screenshot is the Host machine, which tells us in which file the command executes, and in the topmost section of the screenshot, we can see the value of the parameters with which this function was called.



```
Ubuntu1604Target - VMware Workstation 16 Player (Non-commercial use only)
Player
Ubuntu 16.04.7 LTS ubuntu tty1
ubuntu login: usertest0616110
Password:
Last login: Fri Oct 23 10:03:58 PDT 2020 on tty1
Welcome to Ubuntu 16.04.7 LTS (GNU/Linux 4.19.152 x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

New release '18.04.5 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

usertest0616110@ubuntu:~$ sudo su
[sudo] password for usertest0616110:
root@ubuntu:/home/usertest0616110# echo g > /proc/sysrq-trigger
[ 114.381540] sysrq: DEBUG
root@ubuntu:/home/usertest0616110# mkdir 0616116_dir
root@ubuntu:/home/usertest0616110# sudo echo g > /proc/sysrq-trigger
[ 374.743529] sysrq: DEBUG
root@ubuntu:/home/usertest0616110# ls
0616110_directory 0616116_dir Desktop hola
root@ubuntu:/home/usertest0616110# rm -r 0616116_dir/
root@ubuntu:/home/usertest0616110#
root@ubuntu:/home/usertest0616110# ls
0616110_directory Desktop hola
root@ubuntu:/home/usertest0616110# rm -r 0616110_directory/
root@ubuntu:/home/usertest0616110# ls
Desktop hola
root@ubuntu:/home/usertest0616110# _
```

Figure 13: Done

In the final screenshot, we can just see the action is executed in our Target machine. The directory we wanted removed was deleted with the command and we were able to analyze the output in the GDB console on the Host machine.