



Course Title: Kernel Programming & Device Drivers

Faculty: Rajesh Sola



L&T Technology Services



LTTS

GLOBAL
ENGINEERING
ACADEMY



Agenda

A background image showing a person's hands interacting with a tablet. The tablet screen displays various data visualizations, including a bar chart, a line graph, and a pie chart. The person is wearing a light blue shirt. In the background, there is a desk with a pair of glasses and some papers.

Introduction to Kernel

Building Custom Kernel

Kernel Modules

Static and Dynamic Modules

Kconfig entries

Adding System Calls

Summary

Version

| Version | Reviewed by | Approved by | Remarks |
|---------|-------------|-------------|---------|
| 1.0 | | | |



Learning Outcome of the Course

- Understanding Linux Kernel in depth
- Build Custom Kernel
- Writing Kernel Modules
- Adding new system calls

Pre-Requisites for the Course

- Linux Command Line Basics
- Programming in Linux Environment
- Knowledge of Makefiles
- Awareness on Kernel
- Awareness on System Call

Introduction to Kernel



History & Introduction

- // Initiated by Linus Torvalds as a hobby project in 1991
- // Inspired by Minix OS designed by Andrew S Tanenbaum
- // Driven by large and dynamic community
- // Licensed under GPLv2 terms



Linux Kernel - Highlights

- // Scalable for wide range of architectures and configurations
- // Standardized and interoperable programming interface (system calls, std compliance)
- // Secure, Stable and Reliable
- // Rich set of generic drivers
- // Rich set of networking drivers & protocol stacks
- // Upstream (mainline) vs Downstream kernel

Kernel Versions

// a.b.c-d format

- // a.b represents major version

- // c represents release version

- // d represents local version

// Older convention of kernel versions (major, minor, release versions)

// Patch set, optional fourth digit

// Key changes between 2.4.x – 2.6.x

// LTS versions – Long Term Support

// 5.9.x is latest stable version (As on Oct 2020)

Architecture

- // Dual Mode operations of modern CPUs
- // Components of OS
 - // Kernel
 - // Drivers
 - // Libraries
 - // Utilities
- // User mode and Kernel Mode
- // User space and Kernel space
- // Linux Kernel – Modular Architecture
- // Access to kernel space
 - // System Calls
 - // Pseudo file systems – device files, procfs, sysfs, debugfs etc.

System Calls

- // Interface between kernel space and userspace
- // Defined in kernel, part of kernel core (static part)
- // Invoked from user space and executed in kernel space
- // Mode switching , Trap instruction
- // Identified by unique number
- // Passing system call number
- // Passing parameters
- // Handling return values
- // Application Binary Interface (ABI)
- // System call wrappers in user space, workflow behind
- // System call workflow in kernel space
- // Additional Ref:- <https://courses.linuxchix.org/kernel-hacking-2002/09-understanding-system-calls.html>

Programming Paradigm in Kernel

- // Supported Languages – C and Assembly
- // No libraries, but similar APIs for most purposes
- // Avoid floating point operations
- // ANSI/C89 standard
- // Sensitive Memory Access Issues
- // System call interface is stable , e.g. POSIX standard
- // Volatile changes in kernel internal code & APIs

Kernel Source - Highlights

- // Large proportion of architecture independent code
- // Small amount of architecture dependent code
- // Drivers – major stake in kernel source
- // Top level directories in KSRC
 - // Refer your kernel source or online LXR
 - // Additional Ref:- <https://courses.linuxchix.org/kernel-hacking-2002/08-overview-kernel-source.html>

Qemu Based Emulation



Clean Workspace

- // Choose a clean directory for all the work,
- // You may choose directory like workspace/eworkspace/kworkspace/ebuildws/ews or with any sensible name under home directory
- // Don't use Desktop, Downloads, Documents, Music, Videos, Pictures etc, which are meant for other purpose.
- // Avoid spaces or special symbols in path names
- // Under this workspace keep different sub directories for downloaded packages, extracted source to build, configuration files, examples etc.

Setup Qemu

// Install Qemu, a full system emulator for ARM target architecture

```
sudo apt install qemu-system-arm  
qemu-system-arm -v  
qemu-system-aarch64 -v
```

// Alternative – Build from sources

```
Download latest stable version of Qemu source from  
https://www.qemu.org/download/#source  
tar -xvf qemu-5.x.y.tar.xz # replace 5.x.y by suitable version  
cd qemu-5.x.y  
./configure --targetlist=armsoftmmu, armlinuxuser  
                                     --enable-sdl  --prefix=/opt/qemu-5.x.y  
make && make install  
export PATH=/opt/qemu-4.x.y/bin:$PATH # you may add this to ~/.bashrc
```


Rootfs

// Download core-image-minimal-qemuarm.ext4 from

<http://downloads.yoctoproject.org/releases/yocto/yocto-2.5/machines/qemu/qemuarm/>

// Rename core-image-minimal-qemuarm.ext4 as rootfs.img

// Align the size of rootfs

```
e2fsck -f rootfs.img  
resize2fs rootfs.img 16M
```

// Alternative

```
# Download core-image-minimal-qemuarm.tar.bz2 from same link  
qemu-img create -f raw rootfs.img 64M  
mkfs.ext4 rootfs.img  
mount -o loop,rw,sync rootfs.img /mnt/image # mkdir for first time  
tar -jxvf core-image-minimal-qemuarm.tar.bz2 -C /mnt/image  
umount /mnt/image
```

Toolchain

// Install linaro toolchain from ubuntu package manager

```
sudo apt install gcc-arm-linux-gnueabi          # soft float
sudo apt install gcc-arm-linux-gnueabihf        # hard float
```

// We'll go for soft float for now, due rootfs compatibility

// Alternatively, download latest pre-built linaro toolchain from as per host architecture

// From <https://releases.linaro.org/components/toolchain/binaries/latest-7/arm-linux-gnueabi/>, say v7.5.0

```
tar -xvf gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabi.tar.xz -C /opt
export PATH=/opt/gcc-linaro-linux-gnueabi-7.5.0-2019.12_linux/bin:$PATH
```

// Similarly gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf.tar.xz for hard float

Your First Boot (Emulation)

- // Collect prebuilt zImage, vexpress-v2p-ca9.dtb from faculty
- // Ensure rootfs.img is also in same location
- // Emulate using Qemu – sdcard approach

```
qemu-system-arm -M vexpress-a9 -m 1024 -serial stdio \  
-kernel zImage -dtb vexpress-v2p-ca9.dtb \  
-sd rootfs.img -append "console=ttyAMA0 root=/dev/mmcblk0 rw"
```

- // Emulate using Qemu – initrd approach

```
qemu-system-arm -M vexpress-a9 -m 1024 -serial stdio \  
-kernel zImage -dtb vexpress-v2p-ca9.dtb \  
-initrd rootfs.img -append "console=ttyAMA0 root=/dev/ram0 rw"
```

First Steps on Target

```
uname -r  
uname -v  
uname -a  
cat /proc/cpuinfo  
free -m  
df -kh  
mount  
dmesg
```

Building Custom Kernel (Qemu)



Download Kernel Source

- // Download any recent LTS version of kernel source
- // Let's go with 4.14.x for now, for better compatibility with Qemu

```
wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.14.202.tar.xz  
tar -xvf linux-4.14.202.tar.xz
```

- // Or you can checkout kernel source from git.kernel.org, and switch to desired branch

```
git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git  
cd linux  
git checkout tags/v4.14 -b v4.14
```

- // Let's call extracted content (or) checked out content as KSRC

Obtain Configuration File

- “ Locate default config available in KSRC/arch/arm/configs, we'll refer vexpress_defconfig for Versatil Express target being used for Qemu emulation
- “ Or collect any well tested configuration file as base config

```
make ARCH=arm mrproper  
make ARCH=arm vexpress_defonfig  
(or)  
# copy custom config file as .config under KSRC
```

- “ Please note that mrproper will remove built files, including the configuration. So run this only for any new build.

Further Customization

- // Run menuconfig for further customization
- // Resolve any host dependencies at this stage, e.g. libncurses5-dev, flex, bison etc.

```
make ARCH=arm menuconfig
```

- // Let's do these minimal changes for now
 - // General Setup -> Local Version -> "-custom"
 - // Device Drivers -> Block Devices ->
 - // Enable RAM Block device support
 - // Increase default RAM disk size to suitable limit, say 65536
 - // Enable the block layer
 - // Support for large (2TB+)

Build the kernel

// Run menuconfig for further customization

// Build kernel image

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- zImage -j <n>
```

// Build Device Tree Binaries

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- dtbs firmware
```

// Build dynamic modules (can skip for now)

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- modules
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- modules_install \
    INSTALL_MOD_PATH=<tempdir> # or mount point of target rootfs
```

Test the Built outcome

// Collect built outcome to a temporary location

```
# switch to KSRC
cp $KSRC/arch/arm/boot/zImage      <path-of-temp-boot-dir>
cp $KSRC/arch/arm/boot/dts/*.dtb  <path-of-temp-boot-dir>
```

// Ensure rootfs.img is also in same location

// Emulate using Qemu

```
qemu-system-arm -M vexpress-a9 -m 1024 -serial stdio \
    -kernel zImage -dtb vexpress-v2p-ca9.dtb \
    -sd rootfs.img -append "console=ttyAMA0 root=/dev/mmcblk0 rw"
```

Post Boot Checks

In Target

```
uname -r
```

```
uname -v
```

```
ls /boot                # observe new entry
```

```
ls /lib/modules          # observe new entry
```

In Host

```
ls -lh $KSRC/arch/arm/boot/zImage
```

```
ls -lh $KSRC/vmlinux
```

Cross Referencers – Browse Source code

// Cscope

```
make cscope
```

// Online LXR Tools

// elixir.bootlin.com

// lxr.linux.no

Building Custom Kernel (Native)



Download Kernel Source

- “ Let's consider the steps for Ubuntu 20.04, you can adapt suitable names, versions for other Ubuntu versions / Linux distributions
- “ Download custom source for Linux kernel from ubuntu package manager (compatible and similar one from which running kernel is built)

```
sudo apt install linux-source  
# switch to suitable dir under workspace  
tar -jxvf /usr/src/linux-source-5.4.0.tar.bz2  
# consider extracted source as KSRC
```

“ Alternatives:-

- “ Download kernel source from kernel.org or any other reputed sites
- “ Checkout kernel source from git.kernel.org or any other reputed repo

Configuration File

- Configuration file provided in /boot dir, i.e config-5.x.y-default
- Default configuration files under KSRC
- obtain configuration of running kernel

```
cp /proc/config.gz .  
gunzip config.gz  
mv config config_running
```

- Any other well tested configuration file, say config-custom

```
make mrproper  
make x86_64_defconfig # defconfig  
# (or) copy identified config as .config  
# make oldconfig     # can skip for now
```

Further Customization

// Run menuconfig

```
make menuconfig  
# change local version, i.e. General Setup --> Local version --> "xxxxx"
```


Build the Kernel Image & Modules

```
sudo apt-get builddep linux
sudo apt-get install libncurses-dev flex bison openssl libssl-dev \
    dkms libelf-dev libudev-dev libpci-dev libiberty-dev autoconf

make -j 4 bzImage      # or 8, depending on no.of CPUs
make modules
make firmware
```

Ref:- <https://wiki.ubuntu.com/Kernel/BuildYourOwnKernel>

Deployment

```
make modules_install    # deploys dynamic modules to /lib/modules
make install             # Deploys compresses kernel image to /boot
                        # and updates GRUB configuration with an
                        # additional entry to boot new kernel
```

Post Boot Checks

```
uname -r
uname -v
ls /boot          # observe new entry
ls /lib/modules   # observe new entry

ls -lh /boot/vmlinuz-5.x.y-custom
ls -lh $KSRC/arch/arm/boot/zImage
ls -lh $KSRC/vmlinux
```

Kernel Modules



Simple Hello Module

```
//hello.c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
int __init init_module(void)
{
    printk("Hello World..welcome\n");
    return 0;
}
void __exit cleanup_module(void)
{
    printk("Bye,Leaving the world\n");
}

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your name");
MODULE_DESCRIPTION("A Simple Module");
```

```
# Makefile:-
obj-m += hello.o
```

```
# compile the module
make -C ${KSRC} M=${PWD} modules

# use ARCH=arm,
# CROSS_COMPILE=arm-linux-gnueabi-
# if cross compiling for Qemu
```

Use `/lib/modules/`uname -r`/build` as KSRC
, if compiling for Host with available kernel headers.

Testing on Target (Qemu)

```
# copy the ko file to target rootfs
# if cross compiled for Qemu, skip this if
# compiled natively for ubuntu
```

```
sudo mount -o loop,rw,sync rootfs.img /mnt/rootfs
sudo cp hello.ko /mnt/rootfs/home/root
sudo umount /mnt/rootfs
```

```
# Testing module on target
```

```
dmesg -c
insmod hello.ko # sudo
lsmod
cat /proc/modules
dmesg
rmmod hello    # sudo
dmesg
```

Testing on Host (Ubuntu)

```
# copy the ko file to target rootfs
# if cross compiled for Qemu, skip this if
# compiled natively for ubuntu
```

```
sudo mount -o loop,rw,sync rootfs.img /mnt/rootfs
sudo cp hello.ko /mnt/rootfs/home/root
sudo umount /mnt/rootfs
```

```
# Testing module on target
```

```
dmesg -c
insmod hello.ko # sudo
lsmod
cat /proc/modules
dmesg
rmmod hello    # sudo
dmesg
```

Simple Hello Module

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
static int __init hello_init(void) {
    printk("Hello World..welcome\n");
    return 0;
}
static void __exit hello_exit(void) {
    printk("Bye, Leaving the world\n");
}
module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your name");
MODULE_DESCRIPTION("A Simple
Module");
```

```
//Makefile:-
obj-m += hello.o
KSRC = /lib/modules/$(shell uname -r)/build

all:
    make -C ${KSRC} M=${PWD} modules
clean:
    make -C ${KSRC} M=${PWD} clean

# use ARCH=arm,
# CROSS_COMPILE=arm-linux-gnueabi-
# if cross compiling for Qemu,
# and choose suitable KSRC depending on
# compiling for Qemu or Host
```

Significance of `__init`, `__exit` ??

Module Parameters

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
int ndevices=1
module_param(ndevice,int,S_IRUGO);
static int __init pdemo_init(void) {
    printk("Hello World..welcome\n");
    return 0;
}
static void __exit pdemo_exit(void) {
    printk("Bye,Leaving the world\n");
}
module_init(pdemo_init);
module_exit(pdemo_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your name");
MODULE_DESCRIPTION("Parameter demo Module");
```

```
insmod hello.ko ndevices=5
```

```
modinfo paramdemo.ko
```

Module Dependency - simple

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
int xvar=100;
void sayHello(void) { }
static int __init simple_init(void) {
    printk("Hello World..welcome\n");
    return 0;
}
static void __exit simple _exit(void) {
    printk("Bye,Leaving the world\n");
}
module_init(simple_init);
module_exit(simple_exit);
EXPORT_SYMBOL_GPL(xvar);
EXPORT_SYMBOL_GPL(sayHello);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your name");
MODULE_DESCRIPTION("A Hello, World Module");
```

```
//Makefile:-
obj-m += simple.o
```

Significance of MODULE_LICENSE,
EXPORT_SYMBOL_GPL!!

Module Dependency - sample

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
extern int xvar;
extern void sayHello(void);
static int __init sample_init(void) {
    printk("Hello World..xvar=%d\n",xvar);
    sayHello();
    return 0;
}
static void __exit sample _exit(void) {
    printk("Bye,Leaving the world\n");
}
module_init(sample_init);
module_exit(sample_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your name");
MODULE_DESCRIPTION("A Hello, World Module");
```

```
//Makefile:-
obj-m += hello.o
```

- ☐ Can you load sample module before simple?
- ☐ Can you unload simple module before sample module?
- ☐ Idle sequence of loading & unloading modules
- ☐ Check [modinfo](#) on these

Symbol Table

```
cat /proc/kallsyms | grep xxx
# System.map generated in KSRC
# System.map-($ uname -r) in /boot, in case of native
# difference between kallsyms and System.map ??
```

```
arm-linux-gnueabi-nm vmlinux | less
arm-linux-gnueabi-objdump -t vmlinux | less
arm-linux-gnueabi-objdump -d vmlinux | less
```

In-Tree Module : Dynamic

```
#Create a sub dir in KSRC
mkdir drivers/char/dtest
# place hello.c in dtest
```

```
# create a Makefile in dtest
obj-m += hello.o
```

```
# add following entry to drivers/char/Makefile
# be cautious about this step, as editing
# existing file
obj-m += dtest/
```

```
# Re-build the kernel & redeploy
```

```
# Reboot with updated kernel image
# updated rootfs in case of Qemu
```

```
find /lib/modules -name hello.ko
```

```
dmesg -c
modprobe hello
```

TODO:- Try dynamic modules which are dependent each other

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- zImage modules
sudo mount -o loop,rw,sync rootfs.img /mnt/rootfs
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- modules_install INSTALL_MOD_PATH=/mnt/rootfs
sudo umount /mnt/rootfs
```

Static Module (In-Tree)

```
#Create a sub dir in KSRC
mkdir drivers/char/stest
# place sdemo.c in dtest

# create a Makefile in dtest
obj-y += sdemo.o

# add following entry to drivers/char/Makefile
obj-y += stest/
# be cautious , as editing existing file

# Re-build the kernel & redeploy
# reboot with new kernel
# No need to rebuild modules (or) modules_install
```

```
arm-linux-gnueabi-nm vmlinux | grep sdemo_init
arm-linux-gnueabi-objdump -t vmlinux | grep sdemo_init
# normal nm, objdump in case of native
```

```
#check /proc/kallsyms
cat /proc/kallsyms | grep sdemo_init
cat /proc/kallsyms | grep svar
cat /proc/kallsyms | grep sayHello
dmesg | grep sdemo

# check sdemo_init under generated
# System.map also
```

- ☐ Do [sdemo_exit](#) visible under kernel symbol table?
- ☐ Check symbol table with or without `__init`
- ☐ Do sdemo is listed under `lsmod` or available under `/lib/modules`?

Static Module (In-Tree)

```
//sdemo.c - ref code for static module

int svar=100;
void sayHello(void);

static int __init sdemo_init(void) {
    int i;
    for(i=1;i<=4;i++)
        printk("sdemo, i=%d\n",xvar);
    return 0;
}

static void __exit sdemi _exit(void) {
    printk("Bye,Leaving the world\n");
}

EXPORT_SYMBOL_GPL(svar);
EXPORT_SYMBOL_GPL(sayHello);
```

Kconfig entries

```
# Create a dir driver/char/mtest under KSRC
# Place simple.c, sample.c in mtest
```

```
# driver/char/mtest/Kconfig

config SIMPLE
    tristate "Simple module"      #bool
    default n
    help  A simple module

config SAMPLE
    tristate "Sample module"      #bool
    depends on SIMPLE
    default n
    help  A sample module
```

```
# drivers/char/mtest/Makefile
obj-$(CONFIG_SIMPLE) += simple.o
obj-$(CONFIG_SAMPLE) += sample.o
```

```
# drivers/char/Makefile
obj-y += mtest/
```

```
# drivers/char/Kconfig
source "drivers/char/mtest/Kconfig"
```


Kconfig entries

```
# driver/char/mtest/Kconfig - v2

menu "My Custom Modules"

config SIMPLE
    tristate "Simple module"      #bool
    default n
    help A simple module

config SAMPLE
    tristate "Sample module"      #bool
    depends on SIMPLE
    default n
    help A sample module

endmenu
```

No changes required for

- drivers/char/Makefile
- drivers/char/Kconfig
- drivers/char/mtest/Makefile

- ☐ Observe changes in menuconfig
- ☐ Observe generated .config, and entries CONFIG_SIMPLE, CONFIG_SAMPLE
- ☐ Check the dependency between config entries

Kconfig entries

```
# driver/char/mtest/Kconfig - v3
menuconfig CUSTOM
tristate "My Custom Modules"
select SIMPLE
help "My Custom modules"
if CUSTOM
config SIMPLE
    tristate "Simple module"          #bool
    default n
    help A simple module

config SAMPLE
    tristate "Sample module"          #bool
    depends on SIMPLE
    default n
    help A sample module
endif
```

```
# drivers/char/Makefile
obj-$(CONFIG_CUSTOM) += mtest/
```

No changes required for

- drivers/char/Kconfig
- drivers/char/mtest/Makefile

TODO:-

Add Kconfig entries to sdemo (static module)

Adding System Calls



Adding system call in kernel space (ARM)

```
//include/linux/syscalls.h  
asmlinkage long sys_mytestcall(void);
```

```
//arch/arm/tools/syscall.tbl  
398    common  mytestcall msys_testcall
```

```
//kernel/sys.c (or)  
//add in kernel/msys.c and  
//adjust kernel/Makefile with  
//      obj-y +=msys.o  
  
SYSCALL_DEFINE0(testcall)  
{  
    printk("This is my test call\n");  
    return 0;  
}
```

TODO:-

- ☐ Receiving simple arguments
- ☐ Receiving strings
- ☐ Receiving structure variables
- ☐ Returning results via structures

Checking system call presence

On host:-

[arm-linux-gnueabi-nm vmlinux | grep mytestcall](#)

On target:-

[cat /proc/kallsyms | grep mytestcall](#)

Invoking System Call from Userspace

```
//Method-1 : generic wrapper syscall
#include<unistd.h>
#include<stdio.h>

#define __NR_testcall 398

int main() {
    int ret;
    ret=syscall(__NR_testcall);
    if(ret<0)
        perror("testcall");
    return 0;
}
```

TODO:-

Invoking System calls

- ☐ Passing simple arguments
- ☐ Passing strings
- ☐ Passing structure variables
- ☐ Retrieving results via structures

Invoking System Call from Userspace

```
//Method-2A : Simple Assembly  
Code  
mov r7,#398  
SVC 0  
mov r7,#1  
mov r0,#0  
SVC 0
```

```
//Method-2B : Inline Assembly  
asm("MOV r7, #398;"  
    "SVC #0;"  
    "MOV %[result], r0" : [result] "=r" (ret)  
);
```

```
arm-linux-gnueabi-as test.s -o test.o  
arm-linux-gnueabi-ld test.o -o test.out
```

```
arm-linux-gnueabi-gcc test.c -o t.out
```

```
arm-linux-gnueabi-objdump -d test.o
```

Coding Time – Own System Calls

- // Write your own system call which takes simple parameters
- // Write your own system call which returns length of passed string
- // Write your own system call which returns reversed string (or) echo back
- // Write your own system call which logs pid, ppid of calling process
- // Write your own system call which returns pid of calling process
- // Write your own system call which logs various attributes of calling process
- // Write your own system call which returns various attributes of calling process (by passing an empty structure variable), Hints:- sched.h, struct task_struct, current macro
 - // pid, ppid, command name
 - // uid, gid of process owner
 - // state, priority
- // Write your own system call which traverses entire process list and logs pid, ppid, command name of each process , Hint:- next_task, init_task

Userspace test code to invoke each system call
you may use syscall macro for simplicity

Appendix – Signing Modules



Secure Boot : Signing Modules (Ubuntu Native)

```
mokutil --sb-state # check if secure boot is enabled or not

# Choose a secure dir, say /root/mysignatures
openssl req -new -x509 -newkey rsa:2048 -keyout MOK.priv -outform DER -out MOK.der \
    -nodes -days 36500 -subj "/CN=YOUR_NAME/"

mokutil --import /root/mysignatures/MOK.der # register password when prompted
# reboot the system, feed specific characters of registered password
/usr/src/linux-headers-5.4.0-52-generic/scripts/sign-file sha256 \
    /root/mysignatures/MOK.priv /root/mysignatures/MOK.der hello.ko
```

```
# Create script sign-my-module in /usr/local/sbin
/usr/src/linux-headers-5.4.0-52-generic/scripts/sign-file sha256 \
    /root/mysignatures/MOK.priv /root/mysignatures/MOK.der $1

# To sign any module
sign-my-module
```



Thank You !



L&T Technology Services

