Part 1: Establish a root of trust

link to OneDrive assignment1-kwsgMvJM8R: https://bham-my.sharepoint.com/personal/jxx443_student_bham_ac_uk/Documents/dmss/Submission.zip?csf=1&web=1&e=2fchNR

In the process of booting the entire platform, we first load U-Boot as the boot program through QEMU command, and then U-Boot loads the FIT image and verifies the signature with the public key. After validation, the kernel in the FIT image is copied into RAM. After the kernel is started, it initializes and loads the file system, configuring security policies. Then, the Arch Linux operating system starts to run, performing security measures such as file system encryption and permission management. As for QEMU, we use the `-machine virt,virtualization=on` parameter to enable virtualization. This virtualization environment provides a high level of isolation, ensuring that the virtual environment does not directly interact with the host. For U-Boot, it employs device tree and FIT image signature verification, using a public key to validate the integrity of them. This ensures that the loaded kernel, device tree, and other components have not been tampered with. Additionally, the *u-boot.bin* file has already been compiled, making it difficult to directly modify its configuration. Finally Arch Linux is launched, which includes some security measures such as file system encryption, permission management, etc.

To enable U-Boot Verified Boot, we first generate a binary device tree file (*output.dtb*) with the command `dtc -I dts -O dtb board.dts -o output.dtb`. Next, we add the following parameters for the device tree `-device loader,file=dmss.dtb,addr=0x10c308` and FIT image file `-device loader,file=archlinux_fit.itb,addr=0x40200000` to the QEMU startup command. When we need to mount the encrypted file system later, we add the following command: `-drive file=encrypted_filesystem.img,if=virtio,format=raw`. And the full command is saved in *run_qemu.sh* (submitted to Onedrive)

During U-Boot startup, we interrupt the boot process to enter the U-Boot command line, then use the `printenv`, `printenv bootcmd`, and `iminfo` commands to check the environment parameters and image information. We also used the commands `fdt addr $fdtcontroladdr` and `fdt print /signature` to view the device tree signature. We found that during the boot process, there is a signature verification process for the device tree and image files, which mainly uses the SHA-256 hashing algorithm and RSA-2048 encryption algorithm. This verification process ensures that the device tree and image files have not been tampered with, indicating that **the root of trust for the platform has been established**. However, we noticed potential security issues in this U-Boot, such as the lack of environment variable encryption and the absence of a U-Boot boot password. These issues may require modifying the relevant configurations before compiling the *u-boot.bin* file, such as enabling `CONFIG_ENV_ENCRYPTION` and `CONFIG_CMD_PASSWORD`.

To disable U-Boot's verified boot, we first removed `required = "conf";` from *dmss_board.dts*, and the result is *dmss_board_fake.dts*. Then we ran `dtc -I dts -O dtb dmss_board_fake.dts -o dmss_board_fake.dtb` to create a forged device tree (All of *.dts* and *.dib* were submitted to Onedrive). Finally, we used `-device loader,file=dmss_board_fake.dtb,addr=0x10c308` instead of `-device loader,file=output.dtb,addr=0x10c308` in the QEMU command. Another approach is passing in parameters via the U-boot commandline, which is removing the `required` attribute from *output.dtb* directly in the U-Boot console by using the commands `fdt addr $fdtcontroladdr` and `fdt rm /signature/key-verification required`. In these ways, U-boot doesn't verify signature in device tree.

Part 2: Configure full-disk encryption

We started by using the command `cryptsetup luksDump encrypted_filesystem.img` to analyze the LUKS header and key information. This allowed us to identify that the disk encryption settings were using only a single key in keyslot 3, which was *default.key*. The encryption algorithm was AES-XTS-plain64, with a 512-bit key size, which is a standard choice for disk encryption, providing strong protection against brute-force attacks. However, using only one key and storing it in a single keyslot posed a risk, as it made the system vulnerable to key compromise if the key was leaked or accessed by an unauthorized party.

Next, we examined the U-Boot environment variables using the `printenv` command, finding that *u-boot.bin* bootloader had a configuration in `bootargs`: `root=/dev/mapper/rootfs rw cryptdevice=/dev/vda:rootfs cryptkey=rootfs:/mnt/fs.key`, which indicates that the system would mount */dev/vda* as *rootfs* (in /dev/*mapper*) at booting time and use this keyfile `/mnt/fs.key` to unlock the disk image during boot time. However, since this keyfile was specified in the `bootargs` and no passphrase was required, the system could be easily compromised if an attacker had physical access to the system or the boot environment.

From these analysis, we concluded that **the default file system protection was not fully secure**. Thus, we identified a need to enhance the file system's security by adding a passphrase and an external keyfile, and removing original key file as below, which would provide greater resistance against unauthorized access.

In *Submission/change_key.sh* (submitted to Onedrive), we initially added a passphrase "000" using keyfile *default.key*. Then we used the passphrase "000" to add a new keyfile *new.key* and remove *default.key*. This configuration enables the disk image to be unlocked at startup entering the passphrase "000".

To pass *new.key* to QEMU, we created a directory with `mkdir key && cp new.key key/new.key`, then appended `-hdb fat:rw:/work/key` to the QEMU command to create a virtual FAT image, allowing *new.key* to be accessible within the virtual machine. After launching QEMU, we stopped auto-boot, entered the U-Boot console, and set `cryptkey` to *new.key* in *vdb1* using `setenv bootargs "earlycon console=ttyAMA0 mem=5G root=/dev/mapper/rootfs rw cryptdevice=/dev/vda:rootfs cryptkey=/dev/vdb1:vfat:new.key"`. We then executed `boot`.

Alternatively, to enable automatic boot without modifying `bootargs` manually, we edited *u-boot.bin* in a hex editor (010 Editor) . We replaced `bootargs=earlycon console=ttyAMA0 mem=5G root=/dev/mapper/rootfs rw cryptdevice=/dev/vda:rootfs cryptkey=rootfs:/mnt/fs.key` with `bootargs=  console=ttyAMA0 mem=5G root=/dev/mapper/rootfs rw cryptdevice=/dev/vda:rootfs cryptkey=/dev/vdb1:vfat:new.key`. Replacing `earlycon` with spaces here is intentional; this prevents an increase in the *u-boot.bin* size by 4 bytes, which would cause boot failure (even though we adjusted *output.dtb*'s load address from `0x10C308` to `0x10C30C`). Using `-bios u-boot_modified.bin` instead of `-bios u-boot.bin` allows us to skip using `setenv` manually.

Additionally, to pass *new.key* via the command line and configure the disk image to auto-unlock with it, we made numerous attempts to edit *mkinitcpio.conf*, *crypttab*, *fstab*, and other files, followed by running `mkinitcpio`, but made no progress. Feeling powerless, we decided to abandon *initramfs* and instead adopt the `setenv` approach and modify *u-boot.bin*.