SeS resume

Source: this is based on the PDF lectures from Luca Haab, Jean-Roland Schuler, Michael Mäder, and Florent Glück. Some sentences have been paraphrased or taken as is.

Buildroot

● General workflow: ● run `make ses_defconfig` to load `configs/ses_defconfig` into `.config`. ● download opensource softwares (packages, bootloader, kernel, …) via HTTPS, Git or any VCS + Local packages, into `dl`. ● One after the other, each package is **extracted** into `output/build/<packagename>`, **patched** with patches defined in `package/<packagename>` and the global patch folders (here `board/friendlyarm/nanopi-neo-plus2/patches`). ● Configured (e.g. `./configure`) ● Compilation of normal packages inside `output/build/<packagename>`. +build of the kernel, the bootloader, toolchain and rootfs images. ● Finally `genimage` generate a `sdcard.img` with 3 partition: bootloader, SPL, and rootfs. ● Rootfs <= rootfs skeleton + provided files from packages + rootfs_overlay folder board. ● The **Flattened Device Tree** (FDT) = hardware description. Device Tree Sources = `*.dts`.

```
. # Some of the buildroot tree at end of all labs
├── .config # the current config changed by `make menuconfig`
├── arch
├── board # board specific configs
├── boot # configurations for boot related packages
├── dl # download cache
├── fs # packages to generate rootfs
├── linux # config and patches to compile the kernel
├── Makefile # general Makefile
├── output # all generated files
├── package # all packages recipes
├── support
├── system # contains rootfs skeleton
├── toolchain
└── utils
configs/
├── ses_defconfig
...
package/
├── openssh
dl
├── arm-trusted-firmware
├── busybox
├── cryptsetup
├── uboot
├── uboot-tools
dl/openssh
└── openssh-9.1p1.tar.gz
...
output/build/openssh-9.1p1
├── ssh-keygen
├── ssh-keygen.c
...
# config file changed by `make busybox-menuconfig` and associated
Makefile. Same concepts for uboot, and linux.
output/build/busybox-1.35.0/.config
output/build/busybox-1.35.0/Makefile
...
output/host/bin # toolchain for the host
├── aarch64-linux-gcc -> toolchain-wrapper
├── autoconf
...
output/images # the final results
├── bl31.bin # the ARM Trusted Firmware, executed by SPL
├── boot.ext4 # boot partition with ext4 FS
├── boot.scr # boot.cmd compiled version
├── extlinux
├── Image # kernel image in ucompressed format -> booti
├── kernel_fdt.itb # image tree blob: Image + FDT (sun...dtb)
├── kernel_fdt.its # idem but as source description form
├── rootfs.ext2
├── rootfs.ext4 -> rootfs.ext2
├── rootfs.luks # the rootfs encrypted with LUKS
├── rootfs.tar # content of the rootfs
├── sdcard.img # the final file to flash on the SD card
├── sun50i-h5-nanopi-neo-plus2.dtb # device tree blob
├── sunxi-spl.bin # SPL
├── u-boot.bin # raw version (not ELF) of uboot
├── u-boot.itb
└── uInitrd # = mkimage <- Initrd.gz <- gzip <- Initrd <- cpio <-
InitRAMFS folder with init script
board/friendlyarm/nanopi-neo-plus2
├── boot.cmd # uboot script
├── busybox-extras.config # fragment file for busybox
├── genimage.cfg # config for genimage, creating final sdcard.img
├── kernel_fdt.its
├── linux-extras.config # linux config fragment
├── patches # patch folders for various packages
├── post-build.sh # post build script to finish assembling
├── rootfs_overlay/ # files to be merged w/ generated rootfs
├── setup-initramfs.sh # generates the initramfs, started by
post-build.sh
└── uboot-extras.config # uboot config fragment
```

Uboot

● Boot sequence: BootROM in CPU, SPL `sunxi-spl.bin` loads `u-boot.itb`, Uboot loads a Linux kernel image and a Device Tree Blob (DTB) and start it, kernel starts, mounts various fs (rootfs, tmpfs, …) and run the first user process `init`. ● Uboot shells has command like `ext4ls mmc 0:1` (list all files into ext4 partition in the mmc storage 0 at partition 1 (first one)). Uboot variables `bootargs`, `bootcmd`. Setting kernel args `setenv bootargs console=ttyS0,115200 earlyprintk root=/dev/mmcblk0p2 rootwait`. Load the kernel image to DRAM `fatload mmc 0:1 $kernel_addr_r Image`. Same for DTB `fatload mmc 0:1 $fdt_addr_r sun50i-h5-nanopi-neo-plus2.dtb`. Boot kernel with kernel and DTB addr: `booti $kernel_addr_r - $fdt_addr_r`. ● Uboot scripts: `boot.cmd` -> mkimage -> `boot.scr` ● **Flattened Image Tree** (FIT): allows to insert different files into a single file: .its: **Image Tree Source** file (text def of images to include) -> `mkimage` -> .itb: an **Image Tree Blob** file (binary). `u-boot.itb` contains `u-boot-notdtb.bin`, `bl32.bin`, `sun50i-h5-nanopi-neo-plus2.dtb`. ● SPL can only read FIT files and raw binaries -> no ELF support. raw binary generation via `objcopy -O binary` : `u-boot.bin` and `u-boot-nodtb.bin` ● `sdcard.img` contains finally: Linux partition `rootfs` with Ext4 FS, FAT32 partition `boot` with a vfat FS, raw data part for `u-boot.itb` and `sunxi-spl.bin`, and the partition table at very start. Boot partition contains `Image`, `sun50i-h5-nanopi-neo-plus2.dtb` and `boot.scr`.

Linux Kernel hardening

Kernel space: ● buffer overflow -> make the stack not executable -> or Position Independent Executable (PIE) (can be positionned anywhere in the virtual address space) + ASLR. ● kernel config accessible via `/proc/config.gz` -> restrict ● `/dev/mem` (arbitrary memory access): disable or restrict access ● **Canary checks** added in kernel code ● **Use hardware RNG** if kernel software RNG `/dev/urandom` not enough ● Restrict access to **kernel logs** via `dmesg` ● **Heap memory zeroing**: on alloc or free. ● Check memory copies between kernel and userspace: done by drivers. `HARDENED_USERCOPY` to prevent heap overflow exploits. ● Check string and memory functions for buffer overflows: `FORTIFY_SOURCE`, only in the libc `memcpy, mempcpy, memmove, memset, strcpy, strncat, sprintf, vsprintf, snprintf, vs`
Level 1 = normal checks, level 2 = more checking that could break the app. Mix of compile and runtime checks.

User space: ● /proc security risks : expose process info like cmd args, env variables, open fds, … write access means security protections disabling is possible. ● /sys security risks: virtual filesystem that exposes to userspace live view and control interface to devices, drivers, firmwares, and buses ● -> risk of sensitive hardware, kernels or memory info leaks, even container escape via cgroups. can change the hardware state if writable. ● return to libc attacks, return oriented programming (ROP)

Startup scripts in `/etc/init.d/`: scripts to start services, started in alphab. order.

Protection against DOS attacks: **TCP SYN cookie protection** (does not allocate memory before the client `ACK` to avoid `SYN` flooding).

Network stack hardening: IPv4 protection settings, disable IPv6, block ICMP redirect messages, enable source route verification in order to prevent IP spoofing, increase the size of the SYN queue.

Set `umask 0027` (`rwxr-xr-x`) instead of `0022` (`rwxr-x---`)

Application hardening

● `CFLAGS="-fPIE -fstack-protector-all -D _FORTIFY_SOURCE=2 -ftrapv"` ● `LDFLAGS="-Wl,-z,now,-z,relro -z,noexecstack, -pie"` ● `-z noexecstack` -> enable the NX bit (No Executable) to mark memory zone as not executable ● "RELRO (RELocation Read-Only) is a security feature that makes some binary sections (GOT zone) read-only before starting the program". Dynamic symbols resolved at the start of the program to avoid attack on the GOT (Global Offset Table), which contains addresses to each used function. ● From `man ld`: *`now`: When generating an executable or shared library, mark it to tell the dynamic linker to resolve all symbols when the program is started, or when the shared library is loaded by dlopen, instead of deferring function call resolution to the point when the function is first called.* ● Integer (only `int`) overflow detection with `-ftrapv`, runs SIGABRT to sigal the issue, which crash if not handled. ● Replace `str*` by `strn*` variants with a given size.

Filesystems security

● `/etc/passwd` = users list including lines with format `<username>:x:<uid>:<primary gid>:<user full name>:<$HOME>:<$SHELL>` ● `/etc/group` = groups list including lines with format `<groupname>:x:<gid>:<comma separated list of uid of users in this group>`
● Each user/group has a **real ID and effective ID**. `setgid`/`setegid` functions for groups, `setuid`/`seteuid` for users. "For the test file permissions, Linux uses only the effective user ID." ● `/etc/shadow` contains passwords for users in format `<username>:<hash>:<last passwords change in days since Unix Epoch>:0:<number of days between pwd change>:7:::` ● Files: in `rwx` `r` is read permission, `w` is write persisted and `x` is execute permission ● Folders: `r` = read folder content (listing filenames), `w` is change the folder content (create/rename/delete entries) and `x` = can enter directory and access inodes. ● **SUID, SGID bit** (e.g. the `s` in owner or group section of `rwsr-xr-x`): this allow to change UID or GID on execution with the ID of owner/group permission of the executable. ● **Sticky bit** (e.g. the `t` in `rwtr-Sr-`): this avoids users deleting files they don't own, inside a folder with this bit set ● If `t` or `s`, the execute bit is also defined. In uppercase `T` and `S` means no execute persmission. ● Sticky bit + SUID/SGID is a nonsense. ● These 3 bits do work in the same way as `rwx` in terms of octal numerotation used by `chmod`: 7 = 4 (SUID) + 2 (SGID) + 1 (sticky). `chmod 4000` where the `4` is SUID only. ● **ACL: Access Control List**, a way to manage complex list of permissions different for each user. A `setfacl` can give several rights at once on given files or directories. Inheritance based on folders with ACL is possible with `-d` flag. ● **$PATH with `.`** at first position is a major security risk. If `root` runs a command `ls` in a directory where a fake `ls` file exist, it will be picked up instead of the real `/bin/ls`. -> can run anything as root. ● As `/tmp` has open permissions, it could be hardened to avoid some bots, malware and co. -> move `/tmp` into its own partition and mount with `nosuid, noexec, and nodev`. A loopback device can be used to replace a partition. ● "loopback is a pseudo-device that makes a file accessible as a block device"

Filesystems overview

● Partition (a section of a disk) != Filesystem (the way data is organized in a given space). ● Examples of filesystems used in embedded: ext4, btrfs, squashfs on SD-Card, MMC, eMMC. tmpfs, initramfs in RAM (volatile). ● Journalized filesystems exist to restore from a corrupted state. Changes are saved in journal first, on mount, journal is checked and temporary modifications are dropped if doesn't match journal. ● BTRFS (Binary Tree) has CoW (Copy on Write). A change makes a copy, write change to the copy and delete original element only when modifications are done. ● Log filesystem work as a circular buffer with new blocks written to the end. A form of CoW. ● Squash filesystem is compressed + read-only. ● TMPFS: temporary = no persistence, only in RAM, everything deleted on unmount. ● Devtmpfs: automatically creates virtual device files like in `/dev`

LUKS - Linux Unified Key Setup

● LUKS provide a way to create an encrypted partition, a kernel module `dmcrypt` let us create a device mapper under `/dev/mapper/` which can be mounted. ● Configured via `cryptsetup`. Steps to setup: 1. create a LUKS partition, create a mapper `/dev/mapper/luks` with `cryptsetup open`, format the virtual block device `mkfs.ext4 /dev/mapper/luks` and mount it. ● The encryption key is derived from the passphrase (Argon2 or PBKDF2). A number of iterations can be configured to make bruteforce harder. 2 levels of key to allow easy passphrase change + multiple passaphrase. Master key is encrypted separately with each passphrase derived key (temporary key). Only secondary key metadata is stored (iterations, salt).

Initramfs

● Root filesystem mounted early during the kernel setup. ● Difference in the boot process: uboot copies initramfs into RAM, then start the kernel with second arg of `booti` = address of initramfs in RAM. The kernel mount the initramfs and execute the `/init` script. This script can open a LUKS partition and activate a new rootfs.

TPM - Trusted Platform Module

A passive device (usually physical) that provides cryptographic features (hashing, RNG, symmetric and asymetric encryption) and has a small (64KB) volatile (VRAM) and non volatile storage (NVRAM). Multiple types exist: discrete TPM (dtpm): dedicated chip. Integrated TPM: integrated into another chip. Firmware TPM: implement in software, run on the CPU in a TEE (Trusted Execution Environment). Software TPM: a TPM emulator for testing and dev. vTPM: in hypervisors.

Use cases: storing encryption keys or passwords, secure auth, platform state integrity (OS, drivers, firmware), storing keys for disk encryption, … It contains several keys hierarchy, based on a random seed at the top. The primary key inside (the first in the tree) is generated with a KDF from the seed. Private keys are encrypted by their parent key when they are sent outside the TPM. Keys below the primary are child keys. Command `tpm2_createprimary` and `tpm2_create` are used respectively.

● Endorsement hierarchy: `eseed` created by the TPM manufacturer and cannot change ● Platform hierarchy: created by the OEM (Original Equipment Manufacturer). ● Owner hierarchy: the end user hierarchy. ● Null hierarchy: temporary hierarchy for ephemeral keys, the seed is not persisted and regenerated at reboot.

Command example for RSA: `tpm2_createprimary -C o -G rsa2048 -c o_primary.ctx`: create a **primary** key in RSA in the owner hierarchy (`-C`) and save a context file `o_primary.ctx` Commands `tmp2_*`: `createprimary`, `create` child key, `load` a key, `rsaencrypt`, `rsadecrypt`, `sign` (hash and sign the hash of) a file, `verifysignature`, `loadexternal` public keys, `verifysignature`.

AES example for RSA: `tpm2_create -C primary -G aes128cbc -u aeskey.pub -r aeskey.priv`: create a child key (not a primary key) in AES 128 CBC Commands `tmp2_*`: `encryptdecrypt` AES encrypt or decrypt, `create` a child key, `load` a key

TPM PCR and Seals

PCRs are **Platform Configuration Registers** used to store arbitrary hashes. There are 24 registers in my TPM. They can be read or extended (modified by = hash (current hash value + another hash)). SHA256, SHA1 and other algos are available. Some of them can be reset.

Command example: `tpm2_pcrextend 0:sha1=8c8393ac8939430753d7cb568e2f2237bc62d683`: extend the PCR 0 in SHA1 with a new hash. Commands `tmp2_*`: `pcrread`, `pcrextend`, `pcrreset`.

Seals are a way to store (seal) and load (unseal) small amount of data in the TPM. Command example: `tpm2_create -C primary.ctx -i secret -u sec.pub -r sec.priv`: seal the content of file `secret` (this is not a keypair generation because of the `-i`). Commands `tmp2_*`: `create` to seal a file, `readpublic` read public part of object, `evictcontrol` to save the sealed object to NVRAM.

Sessions and policies allow to restrict access to operations on the TPM. For example, a sealed object can only be unsealed if the associated policy is respected. If the policy says that the PCR 8 must be in the same state as when the policy was created, the unseal operation will only work if this is the case. As some PCR could be defined via `tpm2_pcrextend` on boot via multiple processes (uboot, kernel, SPL, …), the set of values can be chosed to make sure a LUKS passphrase can only be unsealed if the kernel binary file has not changed.

IEC 62443

● **The goal of standards** is to help companies be compliant with regulations, contracts and national laws, to minimize business risks. IT and OT (operationnal technologies = technologies in factories) have very different constraints. In OT, the differences: the lifecycle is much longer. Patching strategy is slower and require planification. Offline and manual testing. Physical security way higher. Endpoint security more complex because of limited resources. ● **Goal of IEC 62443**: 3 dimensions (PPT -> People, Processes, Technologies). A family of standards to maintain security of "Industrial and Automation Control Systems". The security concerns several domains: Data, Application, Equipment, Network, Perimter (Firewall, DMZ, VPN), Physical and IACS specific policy. There are security levels: from 0 to 5. There are 5 maturity levels -> good way to define the lowest level that is required.