



MASTER OF SCIENCE
IN ENGINEERING

Secure Embedded Systems

Buildroot - U-Boot -
Kernel Hardening

Samy Francelet, Landry Reynard

Lausanne, le 10.11.2023

Table of contents

2	Introduction
3	Nanopi from scratch
5	U-boot
13	Kernel configuration
16	Conclusion

This document contains the reports for the laboratories *NanoPi from Scratch*, *u-boot* and *Kernel configuration* from the *ses* course @ MSE.

Nanopi from scratch

Goal of this laboratory is to configure a fresh buildroot for the nanopi-neo-plus2.

Compile and solve the problems

The file `boot.cmd` is missing. We added the following content in

`board/friendlyarm/nanopi-neo-plus2/boot.cmd`

```
setenv bootargs console=ttyS0,115200 earlyprintk root=/dev/mmcblk0p2 rootwait
fatload mmc 0 $kernel_addr_r Image
fatload mmc 0 $fdt_addr_r sun50i-h5-nanopi-neo-plus2.dtb
booti $kernel_addr_r - $fdt_addr_r
```

The function header `psci_release_afflvl_locks` is not the same in the `psci_common.c` and `psci_private.h`. To correct this, we first corrected `psci_private.h`, compiled to make sure no more errors linked to this are left, then generated a patch using `git format-patch`:

```
psci_private.h | 4 +---
1 file changed, 2 insertions(+), 2 deletions(-)

diff --git a/psci_private.h b/psci_private.h
index 24a5604..788374d 100644
--- a/psci_private.h
+++ b/psci_private.h
@@ -100,8 +100,8 @@ void psci_acquire_afflvl_locks(int start_afflvl,
                                int end_afflvl,
                                aff_map_node_t *mpidr_nodes[]);
 void psci_release_afflvl_locks(int start_afflvl,
-                                int end_afflvl,
-                                mpidr_aff_map_nodes_t mpidr_nodes);
+                                int end_afflvl,
+                                aff_map_node_t *mpidr_nodes[]);
 void psci_print_affinity_map(void);
 void psci_set_max_phys_off_afflvl(uint32_t afflvl);
 uint32_t psci_find_max_phys_off_afflvl(uint32_t start_afflvl,
```

stored the patch at `board/friendlyarm/nanopi-neo-plus2/patches/arm-trusted-firmware`, deleted the downloaded `arm-trusted-firmware` and relaunched the build to apply the patch.

When the system starts, uboot must use the configuration file boot.scr. How do you solved this ?

In the `post_build` (`board/[MAN]/[BOARD]/post_build.sh`), instead of installing already precompiled binaries, we need to generate the config using `mkimage`.

```
#!/bin/sh
BOARD_DIR="$(dirname $0)"
BUILDR00T_DIR="/buildroot"

#install -m 0644 -D $BOARD_DIR/extlinux.conf $BINARIES_DIR/extlinux/extlinux.conf
mkimage -C none -A arm64 -T script -d $BOARD_DIR/boot.cmd $BUILDR00T_DIR/output/images/
boot.scr
```

Then, the `genimage.cfg` config must also be updated to add the `boot.scr` instead of the `extlinux` file.

```
image boot.vfat {
    vfat {
        files = {
            "Image",
            "sun50i-h5-nanopi-neo-plus2.dtb",
            "boot.scr"
        }
    }
    size = 64M
}
```

In order to be sure that Linux find the SD-card and the embedded eMMC card you must modify the flattened device tree.

As before, we must create a patch for that. The device tree (`sun50i-h5.dtsi`) is missing the aliases for the MMC devices:

```
aliases {
    mmc0 = &mmc0;
    mmc1 = &mmc1;
    mmc2 = &mmc2;
};
```

The patch will be stored in `/board/[MAN]/[BOARD]/patches/linux`

U-boot

Goal of this laboratory is to configure u-boot, create/modify the fragment file in order to save small modification, change the boot partition to ext4 partition and re-install the new u-boot

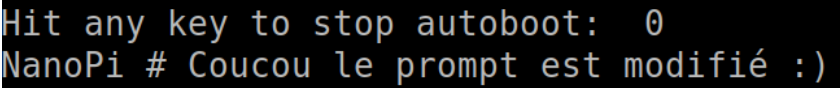
Question 1: U-boot configuration

1. Change the u-boot default prompt to “NanoPi #”

To change the default prompt, we need to go into the `/buildroot` folder and execute `make uboot-menuconfig`. Under *Command line interface*, modify the *Shell prompt* value.

After that u-boot need to be rebuild with `make uboot-rebuild`, `make` and reflash the sdcard.

On the Figure 1 we see that the default prompt has been changed.

A terminal window with a black background and green text. The first line says "Hit any key to stop autoboot: 0". The second line shows the prompt "NanoPi # Coucou le prompt est modifié :)".

```
Hit any key to stop autoboot: 0
NanoPi # Coucou le prompt est modifié :)
```

Figure 1: Image showing that default prompt has been changed.

2. Modify uboot fragment for next installation

In order to do that we add `CONFIG_SYS_PROMPT="NanoPi # "` to the `uboot-extras.config` file.

Question 2: u-boot start a fit file

1) Create a file `kernel_fdt.its` which include the kernel (Image) and the flattened device tree (`sun50i-h5-nanopi-neo-plus2.dtb`). Hash these files with sha512.

We create the file `kernel_fdt.its` in `buildroot/board/friendlyarm/nanopi-neo-plus2/` that contains the following code:

```
/dts-v1/;
/ {
    description = "FIT file with kernel and flattened device tree";
    #address-cells = <1>;

    images {
        kernel {
            description = "Linux 6.3.6 kernel";
            data = /incbin("./Image");
            type = "kernel";
            arch = "arm64";
            os = "linux";
            compression = "none";
            load = <0x40080000>;
            entry = <0x40080000>;
            hash-1 {
                algo = "sha512";
            };
        };

        fdt {
            description = "Flattened Device Tree blob";
            data = /incbin("./sun50i-h5-nanopi-neo-plus2.dtb");
            type = "flat_dt";
            arch = "arm64";
            compression = "none";
            load = <0x4fa00000>;
            entry = <0x4fa00000>;
            hash-1 {
                algo = "sha512";
            };
        };
    };

    configurations {
        default = "default";
        default {
            description = "Boot Linux kernel with fdt blob";
            kernel = "kernel";
            fdt = "fdt";
        };
    };
};
```

2) Create the file Image.itb.

To generate *Image.itb*, we need to copy the .its file just created into `/buildroot/output/images/` and then, in this folder the .itb file is generated with `mkimage -f kernel_fdt.its -E Image.itb`

3) Modify boot.cmd and genimage.cfg in order to load Image.itb, solve and explain the problems

Modify the *boot.cmd* with the following code:

```
setenv bootargs console=ttyS0,115200 earlyprintk root=/dev/mmcblk0p2 rootwait
fatload mmc 0:1 0x40000000 Image.itb
bootm 0x40000000
```

In *genimage.cfg*, the files copied should be change in order to have to following:

```
image boot.vfat {
  vfat {
    files = {
      "Image.itb",
      "boot.scr"
    }
  }
}
```

In *uboot-menuconfig*, it's needed to enable *Support SHA512 in Boot images* and *Support SHA512 in SPL / TPL*. You can also add the following to `uboot-extras.config` to keep the changes after a `make clean` :

```
CONFIG_SPL_SHA512_SUPPORT=y
CONFIG_FIT_ENABLE_SHA512_SUPPORT=y
```

Now, the image is slightly larger than before and can't be load by *uboot*. For this we need to patch the *sunxi* configuration file in *uboot* (`/buildroot/output/build/uboot-2020.10-rc5/include/configs/sunxi-common.h`) by modifying the values `CONFIG_SYS_BOOTM_LEN` to double de size, and `CONFIG_SYS_LOAD_ADDR` to modify the load address accordingly to the modification of `CONFIG_SYS_BOOTM_LEN` (e.g. when doubling the size -> `CONFIG_SYS_LOAD_ADDR` goes from `0x42000000` to `0x44000000`)

4) Modify post-build.sh in order to create automatically these files for a new uboot installation

In *post-build*, we added the following commands to automatically create the flattened image tree blob.

```
IMAGES_DIR="/buildroot/output/images"

cp $BOARD_DIR/kernel_fdt.its $IMAGES_DIR
mkimage -f $IMAGES_DIR/kernel_fdt.its -E $IMAGES_DIR/Image.itb
```

Question 3: BOOT partition ext4

1) Modify *boot.cmd*

We are now loading an ext4 partition instead of a vFAT, so we need to update the load command to boot using the Image that will be stored on the ext4 partition.

```
setenv bootargs console=ttyS0,115200 earlyprintk root=/dev/mmcblk0p2 rootwait
ext4load mmc 0:1 0x40000000 Image.itb
bootm 0x40000000
```

2) In order to create a *sdcard.img* file, it is necessary to modify *genimage.cfg* and *post-build.sh*.

In *genimage.cfg*, we can remove the *boot.vfat* generation, and update the partition boot section:

```
partition boot {
    partition-type = 0x83
    bootable = "true"
    image = "boot.ext4"
}
```

info

`partition-type = 0x83` defines the partition type as `ext4`, instead of `0xC` for `vFAT`.

genimage.cfg unfortunately can't generate `ext4` images. This means that we need to generate it in the *post-build.sh* script. The generation simply creates an `ext4` partition full of zeroes, and copies the FIT blob and the *boot.scr* inside this partition.

```
# Generate boot.ext4 image
rm -rf $IMAGES_DIR/boot.ext4
dd if=/dev/zero of=$IMAGES_DIR/boot.ext4 bs=1024 count=65536
mkfs.ext4 -L boot $IMAGES_DIR/boot.ext4

# Insert Image.itb and boot.scr into boot.ext4
mount -o loop $IMAGES_DIR/boot.ext4 /mnt
cp $IMAGES_DIR/Image.itb /mnt
cp $IMAGES_DIR/boot.scr /mnt
umount /mnt
```


Question 4, -fstack-protector-strong gcc option.

2) Modify the u-boot's compilation option in order to improve the code security and add the *-fstack-protector-strong* option.

2.1) Adding -fstack-protector-strong compile option

In `uboot/Makefile`, replace:

```
KBUILD_CFLAGS += $(call cc-option,-fno-stack-protector)
```

by

```
ifeq ($(CONFIG_STACKPROTECTOR),y)
KBUILD_CFLAGS += $(call cc-option,-fstack-protector-strong)
else
KBUILD_CFLAGS += $(call cc-option,-fno-stack-protector)
endif
```

then, in `uboot/common/Kconfig`, create the `STACKPROTECTOR` configs under the *Security support* menu:

```
config STACKPROTECTOR
bool "Stack Protector buffer overflow detection"
default n
help
    Enable stack smash detection through the compiler built-in
    stack-protector canary logic

config SPL_STACKPROTECTOR
bool "Stack Protector buffer overflow detection for SPL"
depends on STACKPROTECTOR && SPL
default n

config TPL_STACKPROTECTOR
bool "Stack Protector buffer overflow detection for TPL"
depends on STACKPROTECTOR && TPL
default n
```

and in `uboot/scripts/Makefile.spl`, at line 66 add:

```
ifeq ($(CONFIG_$(SPL_TPL_)STACKPROTECTOR),y)
KBUILD_CFLAGS += -fstack-protector-strong
else
KBUILD_CFLAGS += -fno-stack-protector
endif
```

Now, we need a function to callback when a stack smashing has been detected. For that, we create `uboot/common/stackprot.c`

```
#include <common.h>

DECLARE_GLOBAL_DATA_PTR;

unsigned long __stack_chk_guard = (long)(0xf00ddeadbeef & ~0L);

void __stack_chk_fail(void)
{
    panic("Stack smashing detected !\nIn function: %p relocated from %p",
        __builtin_return_address(0),
        __builtin_return_address(0) - gd->reloc_off);
}
```

And we also need to add this file to build, so at the end of `uboot/common/Makefile` we add:

```
obj-$(CONFIG_$(SPL_TPL_)STACKPROTECTOR) += stackprot.o
```

2.2) u-boot test command

To enable testing of the stack protection feature, we might want to add a custom command to u-boot.

First, we create `uboot/cmd/stackprot_test.c`:

```
#include <command.h>

DECLARE_GLOBAL_DATA_PTR;

static int do_test_stackprot_fail(struct cmd_tbl *cmdtp, int flag,
    int argc, char *const agrv[])
{
    char a[128];

    memset(a, 0xa5, 512);
    return 0;
}

U_BOOT_CMD(stackprot_test, 1, 1, do_test_stackprot_fail,
    "test stack protector with buffer overflow", "");
```

add the config in `cmd/Kconfig`, in the *Security commands* or *Debug commands* menu:

```
config CMD_STACKPROTECTOR_TEST
    bool "Enable the 'stackprotector test' command"
    depends on STACKPROTECTOR
    default n
    help
        Enable stackprot_test command
        The stackprot_test command will force a stack overrun to test
        the stack smashing detection mechanisms.
```

and add the object build into `cmd/Makefile`:

```
obj-$(CONFIG_CMD_STACKPROTECTOR_TEST) += stackprot_test.o
```

2.3) Enabling and testing the stack protection

Finally, to enable the stack-protection and the testing command, we must add to `/buildroot/board/[MAN]/[BOARD]/uboot-extras.config`:

```
CONFIG_STACKPROTECTOR=y
CONFIG_CMD_STACKPROTECTOR_TEST=y
```

Build and test the command, if a stack smashing is successfully detected, a patch with all the changes from points 2.1) and 2.2) can be created.

info

If the command doesn't trigger a stack smashing detection, a nice starting point is to search for `stack_chk_fail` in the compiled code with the command:

```
aarch64-linux-objdump -d output/build/uboot-2020.10-rc5/u-boot | grep stack_chk_fail
```

3) Configure buildroot in order to apply the patch for a uboot new download and compilation

Generate a patch for *u-boot* with the modifications from previous point and save it under `/buildroot/board/[MAN]/[BOARD]/patches/uboot/`

Question 5: delete and re-install uboot

In order to check your modifications, delete the uboot actual version and re-install uboot, check if all modifications are made.

To do that we delete the generated files with the command `rm -rf /buildroot/output/build/uboot-2020.10-rc5/` and we recompile them all with `make`. When the build is done, we can quickly check with `aarch64-linux-objdump -d output/build/uboot-2020.10-rc5/u-boot | grep stack_chk_fail` if canaries have been inserted, and test again the `stackprot_test` *u-boot* command we created.

tip

If you want to export your config to your colleague, you can simply copy every board specific configuration (e.g. `post-build.sh`, `genimage.cfg`, `boot.cmd`, ...) and patches to a `my_config/board` folder in your workspace, and your defconfig to `my_config/configs`.

Then like the given script for this laboratory, you can create a `get_my_buildroot.sh` script to create a buildroot folder for your specific configuration :

```
#!/usr/bin/env bash

set -o errexit
set -o pipefail
set -o nounset
# set -o xtrace

git clone git://git.buildroot.net/buildroot /buildroot

cd /buildroot
git checkout -b ses 2022.08.3

rsync -a /workspace/my_config/board/ /buildroot/board/
rsync -a /workspace/my_config/configs/ /buildroot/configs/

chmod +x /buildroot/board/friendlyarm/nanopi-neo-plus2/post-build.sh

make my_defconfig
```

and if you'd like to synchronise those config files using git, you can also add an `update_my_buildroot.sh` script to update the files contained in your buildroot :

```
rsync -a /workspace/my_config/board/ /buildroot/board/
rsync -a /workspace/my_config/configs/ /buildroot/configs/

chmod +x /buildroot/board/friendlyarm/nanopi-neo-plus2/post-build.sh
```

Kernel configuration

1 Configure a secure kernel

Requirements

- Configure a secure kernel (use the Compile Kernel course).
- Configure the kernel so that the kernel size is approximately 20MB
- Activate on buildroot the HAVEGED service [Hardware Volatile Entropy Gathering and Expansion]
- For a next laboratory: Activate: General setup -> Initial RAM File system and RAM Disk (Initramfs/initrd support)
- Compile, install and test the modifications
- Configure buildroot in order to save all modifications for a next linux download and compilation

tip

If you want to easily save and share your custom linux config, first save the current linux kernel config with the save option in `linux-menuconfig` under `path/to/your/hardened/kernel/defconfig`

Then you can add to your buildroot `menuconfig` and `defconfig` (in this lab: under `ses_defconfig`) the following:

```
BR2_LINUX_KERNEL_USE_CUSTOM_CONFIG=y
BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE="path/to/your/hardened/kernel/defconfig"
```

also remove the following lines:

```
BR2_LINUX_KERNEL_USE_ARCH_DEFAULT_CONFIG=y
BR2_LINUX_KERNEL_CONFIG_FRAGMENT_FILES="board/friendlyarm/nanopi-neo-plus2/linux-extras.config"
```

as the old `linux-extras.config` fragment configuration is now embedded in our new `defconfig`.

and use `make linux-update-defconfig` to transfer the linux kernel configuration to the specified file.

Because during this lab we remove unused hardware platforms, this config is very hardware specific. Thus it makes sense to save it under the board folder.

Secure kernel (following the Compile Kernel course)

This config was made following the *SES compile kernel course*.

1. Remove unused hardware platforms from build (keep only *Allwinner sunxi 64-bit Family* and *Broadcom BCM2835 Family* support in this case)
2. Disable Kernel .config file to be saved in the `/prog/config.gz` file
3. Add canaries with the `-fstack-protector-all` compile flag
4. Randomize Heap allocation
5. Randomize SLAB Allocator (allocation of kernel objects.o)
6. Randomize base load address of the kernel Image
7. Make the kernel text section and module section read-only
8. Optimize for performance with the `-O2` compile flag
9. Enable the random number generator (in the `linux-menuconfig`)
10. Enable HAVEGED (in the buildroot `menuconfig`)
11. Restrict access to `/dev/mem` device
12. Strip assembler symbols during link and remove debug info
13. Restrict access to kernel syslog (`dmesg`)
14. Remove automatic stack and heap initialization
15. Enable heap memory zeroing on allocation and on free by default
16. Harden memory copies between kernel and userspace
17. Harden common str/mem functions against buffer overflows
18. Enable Filesystems extended attributes (POSIX Access Control Lists & Security Labels)

Reduce kernel size

With the current setup, the kernel Image is about 23MB, to reduce it further down to 20MB, we can simply change the compile flag from `-O2` to `-Os`.

Activate initramfs initrd

This step is needed to prepare kernel config for future labs:

General setup -> Initial RAM File system and RAM Disk (Initramfs/initrd) support

Testing the added protections

Now for the tests, we can check the following:

- Linux boots up properly
- Test if `/dev/mem` is accessible -> `cat /dev/mem | wc` returns `0 0 0`
- Check if heap randomization is enabled -> `cat /proc/sys/kernel/randomize_va_space` returns `2`
- Test the kernel base address randomization by reading first lines of `/proc/kallsyms` and see where the `_text` section is loaded at two different boot -> we got twice the same address, so base load address of kernel isn't randomized and we weren't able to fix this
- Check available entropy -> `cat /proc/sys/kernel/random/entropy-avail`
- Check that kernel config isn't stored on the image `cat /proc/config.gz` is not found

2 Improve kernel security during the startup

In `board/[MAN]/[BOARD]/rootfs_overlay/etc/` create `sysctl.conf` with the following config:

```
# Randomize virtual address space
kernel.randomize_va_space = 2

# Network stack hardening
## IPv4
net.ipv4.ip_forward = 0
net.ipv4.conf.default.rp_filter = 1
net.ipv4.conf.all.rp_filter = 1
net.ipv4.conf.all.accept_source_route = 0
net.ipv4.conf.all.forwarding = 0
net.ipv4.conf.all.mc_forwarding = 0
net.ipv4.conf.all.accept_redirects = 0
net.ipv4.conf.all.secure_redirects = 0
net.ipv4.conf.all.send_redirects = 0
net.ipv4.icmp_echo_ignore_broadcasts = 1
net.ipv4.icmp_ignore_bogus_error_responses = 1
net.ipv4.conf.all.log_martians = 1
net.ipv4.tcp_max_syn_backlog = 4096
net.ipv4.tcp_syncookies = 1

## IPv6
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.all.forwarding = 0
net.ipv6.conf.all.mc_forwarding = 0
net.ipv6.conf.all.accept_redirects = 0
```

and also create `init.d/S00KernelParameter` with simply : `sysctl -p`

3 Find the difference between normal str/mem function and fortified ones

By checking with `objdump`, we can quickly see that the typical `printf`, `gets`, etc.. get replaced by `__printf_chk` and `__gets_chk`, which all gets an extra parameter used to detect if a buffer overflow will occur with this function call (e.g. `destlen` in `strcpy`).

4 Check your Linux kernel configuration

Using the `kernel-hardening-checker`, we can check if our kernel config can be even more hardened. With the check results from `kernel-hardening-checker` we patched most of the problems, expect when the config was not found.

To save the config made during this lab, please refer to the **tip** at the beginning of this chapter.

Conclusion

Those 3 laboratories gave us a great introduction to Linux image generation with *buildroot*, from the basics (i.e. applying patches, using `menuconfig`, etc..) to more complex parts (how the build systems works, how the image generation works).

They also made a great introduction to using *u-boot* and customizing it (even writing custom commands), and a gave us great insights to the world of secure embedded systems by securing the systems against simple buffer overflows and configuring the kernel to reduce attack surface and add self protection.

Even with little to no experience in the Linux world, we were able to learn a lot without too much hassle. We are now able to use buildroot to generate a basic, hardened Linux image for an embedded system.