Embedded Linux Security

i.MX93 FRDM variant

Practical Labs

bootlin

https://bootlin.com

June 17, 2026

About this document

Updates to this document can be found on https://bootlin.com/training/security.

This document was generated from LaTeX sources found on https://github.com/bootlin/training-materials.

More details about our training sessions can be found on https://bootlin.com/training.

Copying this document

Corrections, suggestions, contributions and translations are welcome!

Training setup

Download files and directories used in practical labs

Install lab data

For the different labs in this course, your instructor has prepared a set of data (kernel images, kernel configurations, root filesystems and more). Download and extract its tarball from a terminal:

```
$ cd
$ wget https://bootlin.com/doc/training/security/security-imx93-frdm-labs.tar.xz
$ tar xvf security-imx93-frdm-labs.tar.xz
```

Lab data are now available in an `security-imx93-frdm-labs` directory in your home directory. This directory contains directories and files used in the various practical labs. It will also be used as working space, in particular to keep generated files separate when needed.

Update your distribution

To avoid any issue installing packages during the practical labs, you should apply the latest updates to the packages in your distro:

```
$ sudo apt update
$ sudo apt dist-upgrade
```

You are now ready to start the real practical labs!

Install extra packages

Feel free to install other packages you may need for your development environment. In particular, we recommend to install your favorite text editor and configure it to your taste. The favorite text editors of embedded Linux developers are of course *Vim* and *Emacs*, but there are also plenty of other possibilities, such as Visual Studio Code[1], *GEdit*, *Qt Creator*, *CodeBlocks*, *Geany*, etc.

It is worth mentioning that by default, Ubuntu comes with a very limited version of the `vi` editor. So if you would like to use `vi`, we recommend to use the more featureful version by installing the `vim` package.

More guidelines

Can be useful throughout any of the labs

- Read instructions and tips carefully. Lots of people make mistakes or waste time because they missed an explanation or a guideline.
- Always read error messages carefully, in particular the first one which is issued. Some people stumble on very simple errors just because they specified a wrong file path and didn't pay enough attention to the corresponding error message.
- Never stay stuck with a strange problem more than 5 minutes. Show your problem to your colleagues or to the instructor.
- You should only use the `root` user for operations that require super-user privileges, such as: mounting a file system, loading a kernel module, changing file ownership, configuring the network. Most regular tasks (such as downloading, extracting sources, compiling...) can be done as a regular user.

[1] This tool from Microsoft is Open Source! To try it on Ubuntu: `sudo snap install code --classic`

- If you ran commands from a root shell by mistake, your regular user may no longer be able to handle the corresponding generated files. In this case, use the `chown -R` command to give the new files back to your regular user.
  Example: `$ sudo chown -R myuser.myuser linux/`

Board setup

Objective: setup your board to be used during all following labs

During this lab, you will:

- Build a Yocto image and the associated SDK
- Deploy it on the board
- Setup SSH connection with the board

Build Yocto

Go to the `$HOME/security-imx93-frdm-labs/board_setup` directory.

Install the kas command, either from you distribution repositories or from https://pypi.org/project/kas/

```
$ python3 -m venv .venv
$ . ./.venv/bin/activate
$ pip install kas
```

Then run kas to build the Yocto image. This might take a few hours.

```
$ kas build security-training-imx93-frdm.yaml
```

You might have to explicitly allow `bitbake` to use user namespaces:

First, add the following content in `/etc/apparmor.d/bitbake`:

```
abi <abi/4.0>,
include <tunables/global>

profile bitbake /path-to-your/bitbake flags=(unconfined) {
  userns,
}
```

```
$ cat /etc/apparmor.d/bitbake | sudo apparmor_parser -a
```

Set up the SD card

We will now use an SD card to store the bootloader, kernel and root filesystem files. The SD card image has been generated in the build folder and is named `security-training-image-freiheit93.rootfs.wic`.

Now flash the image with the following command:

```
sudo dd if=build/tmp-glibc/deploy/images/freiheit93/\
    security-training-image-freiheit93.rootfs.wic of=/dev/mmcblkX conv=fdatasync bs=1M \
    status=progress
```

Setting up serial communication with the board

To set up serial communication with the board, simply connect the USB-C cable to the port labeled DEBUG.

Once the cable is plugged in, a new serial port should appear: `/dev/ttyACM0`. You can also see this device appear by looking at the output of `dmesg`.

To communicate with the board through the serial port, install a serial communication program, such as `picocom`:

```
sudo apt install picocom
```

If you run `ls -l /dev/ttyACM0`, you can also see that only `root` and users belonging to the `dialout` group have read and write access to this file. Therefore, you need to add your user to the `dialout` group:

```
sudo adduser $USER dialout
```

Important: for the group change to be effective, in Ubuntu 18.04, you have to *completely reboot* the system [2]. A workaround is to run `newgrp dialout`, but it is not global. You have to run it in each terminal.

Now, you can run `picocom -b 115200 /dev/ttyACM0`, to start serial communication on `/dev/ttyACM0`, with a baudrate of `115200`. If you wish to exit `picocom`, press `[Ctrl][a]` followed by `[Ctrl][x]`.

There should be nothing on the serial line so far, as the board is not powered up yet.

Boot

Insert the SD card in the dedicated slot on the imx93-frdm. Make sure that switches 1 and 2 in the center of the board are set to ON, and switches 3 and 4 are set to OFF.

Plug in the power USB-C. You should see boot messages on the console.

During the very first boot, a full re-labelling of SELinux file context will occur, followed by a reboot. The whole process should take about 2 minutes.

Wait until the login prompt, then enter `root` as user. Congratulations! The board has booted and you now have a shell.

SSH access

Plug a network cable on any of the two Ethernet ports of the board. If the network is providing DHCP, the board should automatically get an IP address; otherwise you will need to set it up by manually.

You can run the following command on the board to obtain the board IP address

```
# ip addr show eth0 | grep "inet "
```

On your computer, you should be able to connect to the board using SSH:

```
$ BOARD=192.168.10.89
$ ssh root@$BOARD
root@freiheit93:~#
```

Installing SDK

Yocto has also been generating an SDK, providing a toolchain that can be used to cross-compile software for your target. As we will need to compile code during various labs, you should install this SDK now:

```
$ ./build/tmp-glibc/deploy/sdk/oecore-security-training-image-x86_\
    64-cortexa55-freiheit93-toolchain-1.0.sh -d $HOME/security-imx93-frdm-labs/sdk/
```

You can then use the SDK by sourcing the environment file:

```
$ . $HOME/security-imx93-frdm-labs/sdk/environment-setup-cortexa55-oe-linux
```

[2] As explained on https://askubuntu.com/questions/1045993/after-adding-a-group-logoutlogin-is-not-enough-in-18-04/.

Experiment basic crytpographic tools

Objective: Learn how to generate and use basic cryptographic elements: keys, certificates and revocation lists

To demonstrate certificate manipulation, we will create our own Certificate Authority (CA) and issue two certificates: one for the target device and one for the host computer. We will demonstrate how these certificates can be used to authenticate their owners and enable encrypted communication. We will also look at what happens when a certificate is revocated.

Finally, we will measure asymmetric cryptography performances and compare that with symmetric cryptography.

Setting up OpenSSL configuration

The very first step is to create an OpenSSL configuration file. We could start from an example template configuration available in OpenSSL source code (<https://github.com/openssl/openssl/blob/master/apps/openssl.cnf>), but as this is a bit tedious, an example one is provided in the lab folder: `$HOME/security-imx93-frdm-labs/fundamental-concepts/ca`

We also need to initialize the certificates database (`index.txt`), the serial number of the next certificate (`serial`) and the serial number of the next CRL (`crlnumber`).

```
$ cd $HOME/security-imx93-frdm-labs/fundamental-concepts/ca
$ touch index.txt
$ echo 01 > serial
$ echo 1000 > crlnumber
```

Creating CA key and certificate

The next step is to create a key pair and certificate for our certificate authority. This will allow to issue signed certificates to our host computer and target.

Let's begin by creating an 4096 bits RSA key pair for our CA.

```
$ mkdir private
$ openssl genrsa -out private/cakey.pem 4096
```

All data about the created key can then be shown using:

```
$ openssl rsa -text -noout -in private/cakey.pem
Private-Key: (4096 bit, 2 primes)
modulus:
    00:b5:86:2a:ff:6c:6d:4a:33:d7:56:e5:76:5f:be:
    ...
```

We then will create a certificate corresponding to this key. This is our CA root certificate, so it is going to be self-signed. We did not create any configuration file for our PKI, so openssl will interactively ask for various details of our certificate.

```
$ openssl req -new -x509 -days 3650 -config openssl.cnf -key private/cakey.pem -out \
    cacert.pem
```



```
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [FR]:
State or Province Name (full name) []:
Locality Name (eg, city) [Oullins]:
Organization Name (eg, company) [Bootlin security training]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:Training CA
Email Address []:
```

The generated certificate can then be shown:

```
$ openssl x509 -noout -text -in cacert.pem
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            3f:cd:3a:b8:0f:3b:92:cc:89:ce:f0:ea:a4:ed:5c:90:d7:23:33:ee
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=FR, L=Oullins, O=Bootlin security training, CN=Training CA
        Validity
            Not Before: Jan 30 09:34:48 2026 GMT
            Not After : Jan 28 09:34:48 2036 GMT
        Subject: C=FR, L=Oullins, O=Bootlin security training, CN=Training CA
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (4096 bit)
                Modulus:
                    00:b5:86:2a:ff:6c:6d:4a:33:d7:56:e5:76:5f:be:
                    ...
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                5C:33:AA:15:C9:71:3E:FB:DF:0F:A2:9E:65:48:CF:C5:69:C6:14:ED
            X509v3 Authority Key Identifier:
                5C:33:AA:15:C9:71:3E:FB:DF:0F:A2:9E:65:48:CF:C5:69:C6:14:ED
            X509v3 Basic Constraints: critical
                CA:TRUE
    Signature Algorithm: sha256WithRSAEncryption
    Signature Value:
        ae:08:55:8f:83:d7:9e:a2:5d:0a:68:26:73:c2:66:8e:87:5a:
        ...
```

We now have a CA key and a CA root certificate. They can be used to generate and verify certificates for other identities in our PKI.

Creating host computer key and certificate

Now we will create a key for our host computer and create a certificate signing request for our CA.


```
$ mkdir ../host
$ cd ../host
$ openssl genrsa -out hostkey.pem 4096
$ openssl req -new -key hostkey.pem -out host.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:FR
State or Province Name (full name) [Some-State]:.
Locality Name (eg, city) []:Oullins
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Bootlin security training
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:host computer
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

The generated CSR needs to be sent to the CA, in order to obtain the corresponding certificate. Here, we have the CA key locally, so we can use it to generate the certificate.

```
$ cd ../ca
$ mkdir newcerts
$ openssl ca -config openssl.cnf -notext -in ../host/host.csr -out ../host/host.crt
Using configuration from openssl.cnf
Check that the request matches the signature
Signature ok
Certificate Details:
       Serial Number: 1 (0x1)
       Validity
           Not Before: Feb 5 12:32:01 2026 GMT
           Not After : Feb 5 12:32:01 2027 GMT
       Subject:
           countryName = FR
           organizationName = Bootlin security training
           commonName = host computer
Certificate is to be certified until Feb 5 12:32:01 2027 GMT (365 days)
Sign the certificate? [y/n]:y


1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Database updated
```

We can then show the certificate content:

```
$ openssl x509 -noout -text -in ../host/host.crt
Certificate:
```

```
    Data:
        Version: 3 (0x2)
        Serial Number: 1 (0x1)
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=FR, L=Oullins, O=Bootlin security training, CN=Training CA
        Validity
            Not Before: Feb 5 12:32:01 2026 GMT
            Not After : Feb 5 12:32:01 2027 GMT
        Subject: C=FR, O=Bootlin security training, CN=host computer
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (4096 bit)
                Modulus:
                    00:ae:59:b2:18:0e:47:cb:8a:6c:db:59:31:20:49:
                    ...
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                EA:AE:F9:73:BD:45:3F:6E:6F:39:00:84:AB:D5:CB:70:DC:0E:02:07
            X509v3 Authority Key Identifier:
                76:4D:2F:67:12:41:4B:4E:B6:DD:A5:77:71:E0:43:9A:DF:E9:F0:E1
    Signature Algorithm: sha256WithRSAEncryption
    Signature Value:
        2a:5a:f7:0c:23:5f:42:0c:d0:d1:a3:34:ee:62:c7:f8:2b:40:
        ...
```

Creating target key and certificate

Similarly, we will create a key pair for our target and issue the corresponding certificate. The best practice is to generate the key directly on the target device. This reduces the risk of key leakage and is often the only feasible method when keys are stored in hardware modules. The certificate will still have to be generated on the CA computer, so the CSR will have to be transferred between the target and the CA computer.

You can generate the key pair and CSR by running following commands on the target:

```
$ mkdir /keys
$ cd /keys
$ openssl genrsa -out target.key 4096
$ openssl req -new -key target.key -out target.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:FR
State or Province Name (full name) [Some-State]:.
Locality Name (eg, city) []:Oullins
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Bootlin security training
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:target computer
Email Address []:
```



```
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

On the host computer, transfer the CSR file from the target.

```
$ cd ..
$ scp root@${BOARD}:/keys/target.csr .
```

First, as the CA received a CSR request from the target, let's analyze CSR content to make sure the data are genuine.

```
$ openssl req -noout -text -in target.csr
Certificate Request:
    Data:
        Version: 1 (0x0)
        Subject: C=FR, L=Oullins, O=Bootlin security training, CN=target computer
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (4096 bit)
                Modulus:
                    00:99:95:07:1e:bd:e2:08:6d:02:0e:b7:c6:0c:05:
                    ...
                Exponent: 65537 (0x10001)
        Attributes:
            (none)
            Requested Extensions:
    Signature Algorithm: sha256WithRSAEncryption
    Signature Value:
        98:48:e2:3d:b4:d8:d7:e9:6d:e4:99:94:35:73:99:a5:62:e8:
        ...
```

As we are happy with the CSR content, we can generate the corresponding certificate.

```
$ cd ca
$ openssl ca -config openssl.cnf -notext -in ../target.csr -out ../target.crt
Using configuration from openssl.cnf
Check that the request matches the signature
Signature ok
Certificate Details:
        Serial Number: 2 (0x2)
        Validity
            Not Before: Feb 5 12:36:57 2026 GMT
            Not After : Feb 5 12:36:57 2027 GMT
        Subject:
            countryName = FR
            organizationName = Bootlin security training
            commonName = target computer
Certificate is to be certified until Feb 5 12:36:57 2027 GMT (365 days)
Sign the certificate? [y/n]:y


1 out of 1 certificate requests certified, commit? [y/n]y
```

```
Write out database with 1 new entries
Database updated
```

You can then transfer this certificate to your target. You should also transfer the CA certificate in the same move, as we will need it later.

```
$ scp ../target.crt cacert.pem root@${BOARD}:/keys/
```

Optional: Send a signed and encrypted message (RAW)

Let's write a message we want to send encrypted from the host computer to the target.

```
$ cd ..
$ echo "Hello target, this is a message from the host computer\!" > host_message
$ openssl pkeyutl -in host_message -out host_message.enc -certin -inkey target.crt -encrypt
```

Note: In real-world applications, RSA is rarely used to encrypt messages directly. Instead, it secures a session key for symmetric encryption. Also, you might see the encrypted message length is as the key: with a key size of 4096 bits, the encrypted message uses 512 bytes.

Now let's generate a signature file for this message. We will use the host private key to generate the signature, recipients will be able to verify this signature using the host public key contained in the certificate.

```
$ openssl dgst -sha256 -sign host/hostkey.pem -out host_message.sign host_message.enc
```

You can then transfer both the encrypted file and the signature file to the target along with the host machine certificate.

```
$ scp host_message.enc host_message.sign host/host.crt root@${BOARD}:
```

Optional: Verify a signature and decrypt file (RAW)

On the target, you can first verify the message is genuine, using the host certificate. Here we are using raw messages, so there is no direct command allowing to validate the message and the whole key chain in one step. So first, we will verify the host machine certificate and extract the host public key from it:

```
$ cd
$ openssl verify -verbose -CAfile /keys/cacert.pem host.crt
host.crt: OK
$ openssl x509 -pubkey -noout -in host.crt > host_pubkey.pem
```

Now we can verify the message itself:

```
$ openssl dgst -sha256 -verify host_pubkey.pem -signature host_message.sign host_message.enc
Verified OK
```

And finally, we can decrypt the message:

```
$ openssl pkeyutl -in host_message.enc -inkey /keys/target.key -decrypt
Hello target, this is a message from the host computer!
```

Send a signed and encrypted message (S/MIME)

Fine, let's write a reply to this message and send it back to the host computer. But now let's use the S/MIME container format.

```
$ echo "Hi host computer, this is my reply message" > target_message
$ openssl smime -sign -in target_message -signer /keys/target.crt -inkey /keys/target.key \
```



```
    -out target_message.sign
$ openssl smime -encrypt -in target_message.sign -out target_message.sign.enc host.crt
```

On the host, you can again transfer the encrypted file back from the target.

```
$ scp root@${BOARD}:target_message.sign.enc .
```

And you can now decrypt it and verify file signature.

```
$ openssl smime -decrypt -in target_message.sign.enc -recip host/host.crt -inkey host/\
    hostkey.pem -out target_message.dec -outform smime
$ openssl smime -verify -in target_message.dec -CAfile ca/cacert.pem
```

Revoke a certificate

Let's generate a Certificate Revocation List (CRL). We will start with an empty one.

```
$ cd ca
$ openssl ca -config openssl.cnf -gencrl -out ca.crl
```

You can then verify the CRL content:

```
$ openssl crl -in ca.crl -text -noout
Certificate Revocation List (CRL):
       Version 2 (0x1)
       Signature Algorithm: sha256WithRSAEncryption
       Issuer: C=FR, L=Oullins, O=Bootlin security training, CN=Training CA
       Last Update: Feb 5 12:53:56 2026 GMT
       Next Update: Mar 7 12:53:56 2026 GMT
       CRL extensions:
           X509v3 Authority Key Identifier:
               76:4D:2F:67:12:41:4B:4E:B6:DD:A5:77:71:E0:43:9A:DF:E9:F0:E1
           X509v3 CRL Number:
               4096
No Revoked Certificates.
    Signature Algorithm: sha256WithRSAEncryption
    Signature Value:
        71:b0:ca:fa:ae:a7:1b:96:67:98:0c:93:6d:0a:6a:e4:59:d7:
        ...
```

Now let's pretend the target key was compromised and we want to revoke target certificate (serial number 02).

```
$ openssl ca -config openssl.cnf -revoke newcerts/02.pem
Using configuration from openssl.cnf
Revoking Certificate 02.
Database updated
```

So far we only revoked this certificate in the PKI internal database (index.txt):

```
$ cat index.txt
V 270205123201Z  01 unknown /C=FR/O=Bootlin security training/CN=host computer
R 270205123657Z 260205125936Z 02 unknown /C=FR/O=Bootlin security training/CN=target computer
```

The CRL can then be re-generated with the updated database. We can again show it's content to ensure it is correct.

```
$ openssl ca -config openssl.cnf -gencrl -out ca.crl
Using configuration from openssl.cnf
$ openssl crl -in ca.crl -text -noout
Certificate Revocation List (CRL):
        Version 2 (0x1)
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=FR, L=Oullins, O=Bootlin security training, CN=Training CA
        Last Update: Feb 5 13:02:41 2026 GMT
        Next Update: Mar 7 13:02:41 2026 GMT
        CRL extensions:
            X509v3 Authority Key Identifier:
                76:4D:2F:67:12:41:4B:4E:B6:DD:A5:77:71:E0:43:9A:DF:E9:F0:E1
            X509v3 CRL Number:
                4097
Revoked Certificates:
    Serial Number: 02
        Revocation Date: Feb 5 12:59:36 2026 GMT
    Signature Algorithm: sha256WithRSAEncryption
    Signature Value:
        1f:e2:49:bb:19:26:f2:c0:3b:d3:30:3c:0b:97:8e:ec:76:99:
        ...
```

Now let's try again to verify the signature of the message we received from the target:

```
$ cd ..
$ openssl smime -verify -in target_message.dec -CAfile ca/cacert.pem
Hi host computer, this my reply message
Verification successful
```

The verification succeeded because the certificate itself does not indicate revocation. To detect revoked certificates, we must explicitly check the Certificate Revocation List (CRL). One way to distribute the CRL is to concatenate it with the CA certificate itself. Additionally, we have to ask OpenSSL to use these CRL data.

```
$ cat ca/cacert.pem ca/ca.crl > ca/cacert_crl.pem
$ openssl smime -verify -in target_message.dec -CAfile ca/cacert_crl.pem -crl_check
Verification failure
40E764B4347F0000:error:10800075:PKCS7 routines:PKCS7_verify:certificate verify error:../\
    crypto/pkcs7/pk7_smime.c:301:Verify error: certificate revoked
```

Compare performances with symmetric cryptography

Let's demonstrate the difference of performances between symmetric and asymmetric cryptography algorithms. The idea will be to create a big files and encrypt it both using RSA and AES to compare the encryption and decryption speed.

A first issue will arise: as we have seen both RSA and AES operate on fixed size blocks of data. AES operates on 128-bit blocks, while with RSA the size will depend on the key length. The solution is then to split the data in several blocks and encrypt these blocks separately. For AES this is a common need, so most tools will allow to encrypt long data, with various block chaining modes. For RSA, encrypting huge amounts of data is not common: among several reasons, we will see that RSA performances clearly prevent from doing so. As a consequence, we will have to create our own software, allowing to encrypt huge amounts of data with RSA by splitting it in small blocks.

The `crypto-benchmark.py` script provided in the `perfs` folder does that: it allows to encrypt or decrypt files using either RSA with an ad-hoc block splitting mechanism or AES with ECB or GCM block chaining modes. Additionally, it allows to benchmark performances all of these operations.

Run this script and observe the results:

```
$ cd perfs
$ python3 -m venv .venv
$ . .venv/bin/activate
(.venv) $ pip install click cryptography
(.venv) $ ./crypto-benchmark.py benchmark ../host/hostkey.pem
```


Exploring the secure world

Objective: Observe Exception Levels and the secure world, build secure world software.

Observing Exception Levels

In this section, we are going to patch secure world software to display some information. The software we are going to patch is:

- ARM Trusted Firmware (TF-A)
- the Trusted Execution Environment: OP-TEE

Preparing to patch

```
$ git clone https://review.trustedfirmware.org/TF-A/trusted-firmware-a.git --revision \
    569e16caad976a0684147da1ecc6333fd9b7f813
$ git clone https://github.com/OP-TEE/optee_os.git --revision \
    18b424c23aa5a798dfe2e4d20b4bde3919dc4e99
```

Once you are done modifying the code in the next sections, do

```
$ git commit -s -m 'Bootlin Security Training'
$ git format-patch HEAD~1
```

This should generate `0001-Bootlin-Security-Training.patch`. Make sure the filename is correct!

Once you are happy with your patch, you can copy it to the appropriate destination to be applied.

For TF-A:

```
$ cp 0001-Bootlin-Security-Training.patch meta-security-training/recipes-bsp/\
    trusted-firmware-a/trusted-firmware-a/
```

For OP-TEE:

```
$ cp 0001-Bootlin-Security-Training.patch meta-security-training/recipes-security/optee/\
    optee-os/
```

And re-build the image:

```
$ cd board_setup
$ . openembedded-core/oe-init-build-env
$ bitbake security-training-image
```

Adding logs to TF-A

We are interested in observing two characteristics of the system:

- The current Exception Level
- The Security state

Once you've determined where that information is stored (e.g. by looking at the ARM documentation), you can either implement a routine for looking it up yourself, or by searching for relevant functionality in the TF-A codebase.

Adding logs to OP-TEE

After reviewing whether that information is accessible from OP-TEE

- think about the output you expect
- port the code you just used to OP-TEE

Writing a small Trusted Application (TA)

In order to understand how Trusted Applications work, we are going to write one. More specifically, we are going to write a User Mode TA which will be loaded from the REE filesystem.

The requirements for Trusted Application is documented in the build section of the OP-TEE documentation.

We are going to use OP-TEE's TA-devkit to build the TA, so we need to meet those requirements. In the labs data, under `board_setup/meta-security-training/recipes-security/test_ta/`, the basic infrastructure for building the TA is provided.

You will need to write or edit the following files:

- `board_setup/meta-security-training/recipes-security/test_ta/files/ta/Makefile`
- `board_setup/meta-security-training/recipes-security/test_ta/files/ta/include/test-ta.h`
- `board_setup/meta-security-training/recipes-security/test_ta/files/ta/test-ta.c`
- `board_setup/meta-security-training/recipes-security/test_ta/files/ta/user_ta_header_defines.h`

Once you are done, you can run

```
$ cd board_setup
$ . openembedded-core/oe-init-build-env
$ bitbake test-ta
```

Writing the userland client for the TA

User mode TAs, contrary to pseudo TAs, are loaded into OP-TEE when the REE attempts to call them. We now need a client on the REE side to call our TA. It will need to invoke the command we just implemented in the previous step.

You will need to write or edit the following files:

- `board_setup/meta-security-training/recipes-security/test_ta/files/host/test_ta_host.c`

Deploying the TA

OP-TEE's REE FS Trusted Applications should be installed in `/lib/optee_armtz`. Each one is stored as `<UUID>.ta`, we'll deploy ours in the same way:

```
$ cp <UUID>.ta /lib/optee_armtz/
```

Confirm that `tee-supplicant` is running, as that's the daemon that will load the TA:

```
$ ps | grep tee
```

You can now trigger your TA:


```
$ ./test_ta_host
```


Setup AHAB on i.MX93

Objective: Learn how to implement the first stage of a secure boot

Prerequisite

To build the AHAB container for the first stage of the i.MX93 secure boot, we are going to use NXP's Secure Provisioning SDK (SPSDK).

First, install it in your python virtual environment (the one you created to install kas in the `Build Yocto` section of the `Board setup` lab):

```
(.venv) $ pip install spsdk
```

AHAB also necessitates asymmetric keys to:

- sign the container on the host
- verify the signature on the target

It can include up to 4 of these keys, which must all be hashed together, and we therefore need to generate those 4 Super Root Keys (SRKs):

```
$ for i in `seq 0 3`; \
  do openssl genrsa -out keys/srk_$i.pem 4096; \
  openssl rsa -in keys/srk_$i.pem -pubout > keys/srk_$i.pub; \
done
```

Building a signed AHAB container

The first step in building the AHAB image container we will use for the first stage of secure boot, is to identify the system's components we want the ROM code to verify.

Remember that the container will actually be split into 2 container sets:

- The **primary** container set, which will contain the very early boot components
- The **secondary** container set, containing the next stage(s)

Both will be signed with one of the SRKs created at the previous step.

The **primary** container set will contain:

- The ELE firmware, which must be downloaded from NXP. In our case, Yocto did that for us: `build/tmp-glibc/sysroots-components/cortexa55/firmware-ele-imx/usr/lib/firmware/imx/ele/mx93a1-ahab-container.img`
- The DRAM training binaries, which Yocto also downloaded:
  - `build/tmp-glibc/sysroots-components/cortexa55/imx-boot-firmware-files/firmware/lpddr4_imem_1d_v202201.bin`
  - `build/tmp-glibc/sysroots-components/cortexa55/imx-boot-firmware-files/firmware/lpddr4_imem_2d_v202201.bin`
  - `build/tmp-glibc/sysroots-components/cortexa55/imx-boot-firmware-files/firmware/lpddr4_dmem_1d_v202201.bin`

  - `build/tmp-glibc/sysroots-components/cortexa55/imx-boot-firmware-files/firmware/lpddr4_dmem_2d_v202201.bin`

- Finally, U-Boot's SPL built by Yocto, and located in `build/tmp-glibc/work/freiheit93-oe-linux/u-boot-kiss/git/build/imx93_frdm_defconfig/u-boot-spl-ddr.bin`

The **secondary** container set will contain:

- TF-A's BL31: `build/tmp-glibc/deploy/images/freiheit93/bl31.bin`
- OP-TEE: `build/tmp-glibc/deploy/images/freiheit93/optee/tee-raw.bin`
- Finally, U-Boot proper: `build/tmp-glibc/work/freiheit93-oe-linux/u-boot-kiss/git/build/imx93_frdm_defconfig/u-boot-dtb.bin`

The easiest way to write the YAML templates is to start from the ones included in spsdk:

```
(.venv) $ nxpimage bootable-image get-templates -f mimx9352 -t imx_boot_flash_singleboot -o \
    ahab_template
(.venv) $ ls ahab_template
bootable_image.yaml inputs spl.yaml uboot.yaml
```

Notice that these templates do not include the signing configuration that we saw in the examples of the lectures, so you'll have to add it. Once you are done filling the YAML templates, you can build the image:

```
(.venv) $ nxpimage -v bootable-image export -c ahab_template/bootable_image.yaml
```

This should create:

- The AHAB container image: `ahab_template/bootable_image.bin`
- One `nxpele` script for each subcontainer:
  - `output/ahab_oem0_srk0_hash_nxpele.bcf`
  - `output/ahab_oem1_srk0_hash_nxpele.bcf`

The SoC only has one set of SRK eFuses, so both container set should use the same keys, so these two scripts should be identical:

```
$ diff output/ahab_oem1_srk0_hash_nxpele.bcf output/ahab_oem0_srk0_hash_nxpele.bcf
```

You can confirm that the AHAB container you created is signed:

```
(.venv) $ nxpimage ahab verify -f mimx9352 -b ahab_container
```

Flashing the i.MX93 eFuses

> This step is irreversible: the eFuses can only be flashed once. Although the device will still boot unless the i.MX93 is transitioned into **OEM_CLOSED** state, this might still be undesirable.

The easiest way to interact with the SoC's eFuses is by using SPSDK's `nxpele` tool. In our case, we will use it to issue command to U-Boot over the command line. We therefore need to reset the board, and interrupt U-Boot's boot process.

Once this is done, we'll need to disconnect the serial client, as it would interfere with `nxpele`.

```
$ for i in `seq 128 135`; do nxpele -p /dev/ttyACM0 -d uboot_serial -f mimx9352 -v \
    read-common-fuse --index $i; done
```



```
INFO:spsdk.ele.ele_comm:ELE communicator is using 131072 B size buffer at 83800000 address in\
     mimx9352, Revision: latest target.
INFO:spsdk.ele.ele_comm:Sent message information:
Command: READ_COMMON_FUSE - (0x97)
Command words: 2
Command data: False
Response words: 3
Response data: False
Response status: Success

Read common fuse ends successfully.
Fuse ID_128: 0x00000000

[...]

Read common fuse ends successfully.
Fuse ID_135: 0x00000000
```

Before you flash the eFuses, take note of the output of U-Boot's `ahab_status`:

If the board is not booting from a **signed** AHAB container, the output should be:

```
$ Hit any key to stop autoboot: 0
u-boot=> ahab_status
Lifecycle: 0x00000008, OEM Open


       0x0287eed6
       IPC = MU APD (0x2)
       CMD = ELE_OEM_CNTN_AUTH_REQ (0x87)
       IND = ELE_NO_AUTHENTICATION_FAILURE_IND (0xEE)
       STA = ELE_SUCCESS_IND (0xD6)

       0x0287eed6
       IPC = MU APD (0x2)
       CMD = ELE_OEM_CNTN_AUTH_REQ (0x87)
       IND = ELE_NO_AUTHENTICATION_FAILURE_IND (0xEE)
       STA = ELE_SUCCESS_IND (0xD6)
```

If the board **is** booting from a **signed** AHAB container, this should be the output:

```
Hit any key to stop autoboot: 0
u-boot=> ahab_status
Lifecycle: 0x00000008, OEM Open


       0x0287fad6
       IPC = MU APD (0x2)
       CMD = ELE_OEM_CNTN_AUTH_REQ (0x87)
       IND = ELE_BAD_KEY_HASH_FAILURE_IND (0xFA)
       STA = ELE_SUCCESS_IND (0xD6)

       0x0287fad6
       IPC = MU APD (0x2)
       CMD = ELE_OEM_CNTN_AUTH_REQ (0x87)
```



```
       IND = ELE_BAD_KEY_HASH_FAILURE_IND (0xFA)
       STA = ELE_SUCCESS_IND (0xD6)
```

We can now flash the eFuses. Although this step is **irreversible**, it will not prevent your board from booting, as AHAB will not enforce the signature in **OEM_OPEN**.

```
$ nxpele -p /dev/ttyACM0 -d uboot_serial -f mimx9352 -v batch output/ahab_oem0_srk0_hash_\
    nxpele.bcf
INFO:spsdk.ele.ele_comm:ELE communicator is using 131072 B size buffer at 83800000 address in\
     mimx9352, Revision: latest target.
INFO:spsdk.ele.ele_comm:Sent message information:
Command: WRITE_FUSE - (0xd6)
Command words: 3
Command data: False
Response words: 3
Response data: False
Response status: Success

ELE write fuse ends successfully.

[...] Repeats 7 more times because the hash is 8*32 = 256 bits
```

We can now re-read the fuses, and they should contain our hash:

```
$ for i in `seq 128 135`; do nxpele -p /dev/ttyACM0 -d uboot_serial -f mimx9352 -v \
    read-common-fuse --index $i; done
INFO:spsdk.ele.ele_comm:ELE communicator is using 131072 B size buffer at 83800000 address in\
     mimx9352, Revision: latest target.
INFO:spsdk.ele.ele_comm:Sent message information:
Command: READ_COMMON_FUSE - (0x97)
Command words: 2
Command data: False
Response words: 3
Response data: False
Response status: Success

Read common fuse ends successfully.
Fuse ID_128: 0x09C79D69

[...]
Fuse ID_129: 0x340DBD87
[...]
Fuse ID_130: 0x6BA8EB5A
[...]
Fuse ID_131: 0xECD130AD
[...]
Fuse ID_132: 0xFD09593E
[...]
Fuse ID_133: 0x38FA50F1
[...]
Fuse ID_134: 0x18DC2FA2
[...]
Fuse ID_135: 0xC6DED0B6
```


Testing the AHAB stage

Now that the eFuses are flashed (if you finished the previous step), the ELE ROM should now be able to verify the signature on the AHAB subcontainers.

If you succesfully signed your AHAB containers, this is the output you should see when running `ahab_status`:

```
Hit any key to stop autoboot: 0
u-boot=> ahab_status
Lifecycle: 0x00000008, OEM Open


       No Events Found!
```

This indicates that the system would have booted, even in `OEM_CLOSED` or `OEM_LOCKED`.

Because we have not transitioned the system to either state, though, it will still boot an unsigned container, or a wrongly signed one.

For example, we can change the value of `used_srk_id` in `ahab_template/uboot.yaml` to `1`:

```
u-boot=> ahab_status
Lifecycle: 0x00000008, OEM Open


       0x0287f0d6
       IPC = MU APD (0x2)
       CMD = ELE_OEM_CNTN_AUTH_REQ (0x87)
       IND = ELE_BAD_SIGNATURE_FAILURE_IND (0xF0)
       STA = ELE_SUCCESS_IND (0xD6)
```

Notice that we only have one error this time, because we only changed one subcontainer. We also have a very slightly different error code.

To get back to good state, you can either change `used_srk_id` back to `0`, or adapt `signer` to use the SRK with ID `1`.

Secure boot: kernel FIT verification

Objective: Learn how to setup the second stage of secure boot: verification of the kernel signature by U-Boot.

Activating signature verification in U-Boot

Configuring signature verification

We now want U-Boot to verify the kernel FIT image before it loads it. There is one configuration option in U-Boot that needs to be enabled for this.

That option is actually already enabled in our build, you can see it by looking at the following patch:

```
$ cat board_setup/meta-security-training/recipes-bsp/u-boot/u-boot-kiss/freiheit93/\
    0009-Add-FIT-signature.patch
commit 88fd6a65083c4e8cc2cfa2a38719a128fa8f3770
Author: Olivier Benjamin <olivier.benjamin@bootlin.com>
Date: Sat May 9 14:45:09 2026 +0000

    Add FIT signature

    Signed-off-by: Olivier Benjamin <olivier.benjamin@bootlin.com>

diff --git a/configs/imx93_frdm_defconfig b/configs/imx93_frdm_defconfig
index 1deb7957cbc..8aef68c2f0b 100644
--- a/configs/imx93_frdm_defconfig
+++ b/configs/imx93_frdm_defconfig
@@ -132,6 +132,8 @@ CONFIG_BZIP2=y
 CONFIG_UTHREAD=y

 CONFIG_FIT=y
+CONFIG_FIT_SIGNATURE=y
+CONFIG_DEVICE_TREE_INCLUDES="secure-boot.dtsi"

 CONFIG_ENV_IS_NOWHERE=y
 CONFIG_ENV_IS_IN_MMC=y
```

Passing the signing key to U-Boot

Our build does not actually sign the FIT image, but U-Boot still boots it. This shows that activating FIT signature is not enough for U-Boot to refuse to boot unsigned or wrongly signed FIT images. It will only do so if it has an embedded key, and a list of images or configurations that **require** signature. U-Boot expects this configuration in the form of a DTSI file, that can be included in its Device Tree Blob (DTB). We can use the `key2dtsi.py` to generate the appropriate configuration based on a key.

As you can see, the patch above is already prepared to include this, in the form of `secure-boot.dtsi`. Looking at that file, which is located in `board_setup/meta-security-training/recipes-bsp/u-boot/u-boot-kiss/freiheit93/`.

You can now use `key2dtsi.py` to replace that file. You can use the version that Yocto deployed as part of the native build of the `u-boot tools`:

```
(.venv) $ pip install pycryptodomex
(.venv) $ export UBOOT_TOOLS_DIR=build/tmp-glibc/work/x86_64-linux/u-boot-tools-native/\
    2024.01
(.venv) $ ${UBOOT_TOOLS_DIR}/git/tools/key2dtsi.py --help
```

Testing

Once this is done, you can rebuild U-Boot and flash it to the micro SD card. Because Yocto does a lot of caching, the easiest way is probably to rebuild the entire image:

```
$ cd board_setup
$ . openembedded-core/oe-init-build-env
$ bitbake security-training-image
```

Once it has been rebuilt, you can flash it to your micro SD card:

```
$ bmaptool copy build/tmp-glibc/deploy/images/freiheit93/\
    security-training-image-freiheit93.rootfs.wic /dev/sdc
```

Once you boot, this is the result you should see:

```
Verifying Hash Integrity ... Failed to verify required signature 'key-fit_signing_key'
error!
Unable to verify required signature for '' hash node in 'kernel-1' image node
Bad Data Hash
ERROR -2: can't get kernel image!
u-boot=>
```

The system refuses to boot the FIT image, because it has not been signed. This is secure boot working as it should. In the next step, we'll build a signed FIT image that can be booted.

Signing the FIT image

We need to craft a new FIT image, properly signed for U-Boot to accept it. We could do this directly in Yocto, but we'll do it manually here to learn the process.

FIT images, just like for AHAB, are a type of "container", so our first step will be identifying what components need to go in. In this case, our FIT image will contain:

- our kernel: `build/tmp-glibc/work/freiheit93-oe-linux/linux-kiss/6.16/build/linux.bin`
- its DTB: `build/tmp-glibc/deploy/images/freiheit93/imx93-11x11-frdm.dtb`

Note that the file paths in the `.its` file are relative to the location of the `.its` file, not the directory where `mkimage` is run.

In the labs-data, we included `unsigned_fit.its`, which a FIT specification similar to the one that Yocto is currently using. Modify it to create your specification including the signature. You can now use `mkimage` to create the signed FIT:

```
$ sudo apt install -y device-tree-compiler
(.venv) $ ${UBOOT_TOOLS_DIR}/build/tools/mkimage --help
```

Once you're done, and you have flashed you new signed FIT image, your U-Boot boot log should look like this:

```
## Checking Image at 83000000 ...
   FIT image found
   FIT description: Kernel fitImage for security training Linux/6.16/freiheit93
    Image 0 (kernel-1)
     Description: Linux kernel
     Type: Kernel Image
     Compression: uncompressed
     Data Start: 0x830000ec
     Data Size: 20439552 Bytes = 19.5 MiB
     Architecture: AArch64
     OS: Linux
     Load Address: 0x80400000
     Entry Point: 0x83000000
     Hash algo: sha256
     Hash value: 654f25e2b7aa16277b871ec3c39adc2d7da6bd69ce59fbb59a5726d431ade708
     Sign algo: sha256,rsa4096:key-fit_signing_key
     Sign value: c57a3f0461f3ca2d64d0a4f14...cd7f1527375a55e55117b227ceb40e8b131268
    Image 1 (fdt-imx93-11x11-frdm.dtb)
     Description: Flattened Device Tree blob
     Type: Flat Device Tree
     Compression: uncompressed
     Data Start: 0x8437e698
     Data Size: 38662 Bytes = 37.8 KiB
     Architecture: AArch64
     Hash algo: sha256
     Hash value: c3e8f00b2b1c59b447a59a821e4b46c27270ab92bb14aa312236f2673b187b7a
     Sign algo: sha256,rsa4096:key-fit_signing_key
     Sign value: 5732c710e63f5dac330bd5f9abf...ce31baf45895f9bc951c4e96e9ec78c317f
    Default Configuration: 'conf-imx93-11x11-frdm.dtb'
    Configuration 0 (conf-imx93-11x11-frdm.dtb)
     Description: 1 Linux kernel, FDT blob
     Kernel: kernel-1
     FDT: fdt-imx93-11x11-frdm.dtb
     Hash algo: sha256
     Hash value: unavailable
     Sign algo: sha256,rsa4096:key-fit_signing_key
     Sign value: 2d2c1444d2094a9078ac7f35be446...bb136b3ca2a972a339afd0cbbc8e36cb14
## Checking hash(es) for FIT Image at 83000000 ...
   Hash(es) for Image 0 (kernel-1): sha256,rsa4096:key-fit_signing_key+ sha256+
   Hash(es) for Image 1 (fdt-imx93-11x11-frdm.dtb): sha256,rsa4096:key-fit_signing_key+ \
       sha256+
## Loading kernel (any) from FIT Image at 83000000 ...
   Using 'conf-imx93-11x11-frdm.dtb' configuration
   Verifying Hash Integrity ... OK
   Trying 'kernel-1' kernel subimage
     Description: Linux kernel
     Type: Kernel Image
     Compression: uncompressed
     Data Start: 0x830000ec
     Data Size: 20439552 Bytes = 19.5 MiB
     Architecture: AArch64
     OS: Linux
     Load Address: 0x80400000
```



```
    Entry Point: 0x83000000
    Hash algo: sha256
    Hash value: 654f25e2b7aa16277b871ec3c39adc2d7da6bd69ce59fbb59a5726d431ade708
    Sign algo: sha256,rsa4096:key-fit_signing_key
    Sign value: c57a3f0461f3ca2d64d0a4f14bd34...cd7f1527375a55e55117b227ceb40e8b131268
  Verifying Hash Integrity ... sha256,rsa4096:key-fit_signing_key+ sha256+ OK
```


Key management

Objective: Experiment with PKCS#11 and OP-TEE

Background

The OP-TEE codebase includes a PKCS#11 Trusted Application.

First, confirm that this TA is present on your system.

Setting up the TA

We could write a small client application as we did in the *secure world* lab, but we don't need to. Indeed, the API exposed by the TA can be compared to the internal API of an HSM, but the main advantage of PKCS#11 is the high-level API specification, which lets manufacturers provide compatibility modules in the form of shared objects.

OP-TEE is no exception, and the compatibility module is implemented by the OP-TEE client, in the form of libckteec, as documented here.

You can confirm that everything is working by showing some general info:

```
root@freiheit93:~# p11 -I
Cryptoki version 2.40
Manufacturer Linaro
Library OP-TEE PKCS11 Cryptoki library (ver 0.1)
Using slot 0 with a present token (0x0)
```

Notice the "Cryptoki" name showing up.

The next steps are the same as for a "normal" HSM, and described in the documenation linked above:

- Initialize the token, with a label and a Security Office (SO) PIN:

```
root@freiheit93:~# p11 --init-token --label optee
Using slot 0 with a present token (0x0)
Please enter the new SO PIN:
Please enter the new SO PIN (again):
Token successfully initialized

root@freiheit93:~# p11 -L
Available slots:
Slot 0 (0x0): OP-TEE PKCS11 TA - TEE UUID 94e9ab89-4c43-56ea-8b35-45dc07226830
  token label : optee
  token manufacturer : Linaro
  token model : OP-TEE TA
  token flags : login required, rng, token initialized
  hardware version : 0.0
  firmware version : 0.1
  serial num : 0000000000000000
  pin min/max : 4/128
Slot 1 (0x1): OP-TEE PKCS11 TA - TEE UUID 94e9ab89-4c43-56ea-8b35-45dc07226830
  token state: uninitialized
```



```
Slot 2 (0x2): OP-TEE PKCS11 TA - TEE UUID 94e9ab89-4c43-56ea-8b35-45dc07226830
  token state: uninitialized
```

- Log in as the Security Office and create a User PIN:

```
# p11 -l --login-type so --init-pin
Using slot 0 with a present token (0x0)
Logging in to "optee".
Please enter SO PIN:
Please enter the new PIN:
Please enter the new PIN again:
User PIN successfully initialized
```

Using the TA as a soft HSM

Now that the (virtual) token has been initialized, we can use it to store the target's private key that we generated earlier. This is done using the `--write-object` command:

```
# p11 -l --login-type user --write-object target.key --type privkey --label 'Target Key'
Using slot 0 with a present token (0x0)
Logging in to "optee".
Please enter User PIN:
Created private key:
Private Key Object; RSA
  label: Target Key
  Usage: none
  Access: sensitive
```

Unfortunately, this key risks to not be very useful. Experiment with the commands to fix that, and please make sure that key cannot be used to sign MD5 hashes.

For the purpose of the first lab, we generate the target key using `openssl`. This is typically the type of task that an HSM is well suited for.

Generate a new RSA 4096 key directly in OP-TEE.

Thinking about HW storage

Once your key is properly setup, reboot the device, and list the keys in the slot you were using:

```
$ p11 -l -O
```

You should still see the keys that you created in the previous step. But OP-TEE itself does not have a filesystem. Looking at the OP-TEE documentation and sources, try and figure out how OP-TEE stored those keys.

We now have an RSA key that is tied to the hardware, and cannot leave the secure world in cleartext. However, this key is not yet integrated into our PKI. We're going to enroll it now, following similar steps as we did earlier.

In order to be able to use this key with OpenSSL, however, we need specific configuration:

```
$ cat /root/openssl.conf
openssl_conf = openssl_init

[openssl_init]
engines = engine_section
```



```
[engine_section]
pkcs11 = pkcs11_section

[pkcs11_section]
engine_id = pkcs11
dynamic_path = /usr/lib/engines-3/pkcs11.so
MODULE_PATH = /usr/lib/libckteec.so.0.1.0
PIN = 5678
```

Now we can generate a CSR:

```
# OPENSSL_CONF=openssl.conf openssl req -new \
       -engine pkcs11 \
       -keyform engine \
       -key pkcs11:object=TargetOnlyKey \
       -out target.csr
```

After this, we can follow the steps from the first lab to sign the CSR.

Restricting userland applications

Objective: Learn how to restrict userspace attack surface using various tools: seccomp, SELinux, and systemd configuration

To demonstrate possibilities to filter userspace operations, we will show how various mechanisms can be used. First, we will use seccomp to restrict accessible syscalls in a simple application. Next, we will quickly manipulate SELinux contexts. Finally, we will show how userland security features can be used directly from Systemd.

Preventing a simple application from executing shellcode using SECCOMP

The `$HOME/security-imx93-frdm-labs/userland/seccomp` folder contains the code of a sample application. This application does two things: it prints the content of some existing file, then launches a shell. Of course, launching the shell is an expected behavior here, but we are actually trying to mimic a faulty application, where an attacker exploits a breach to launch a shell. To avoid that, we will demonstrate how *seccomp* can be used to restrict accessible syscalls.

First, you can have a look at the source code and compile the application. Then, launch it and observe the current behaviour.

```
$ cd $HOME/security-imx93-frdm-labs/userland/seccomp
$ . $HOME/security-imx93-frdm-labs/sdk/environment-setup-cortexa55-oe-linux
$ make
$ ssh root@${BOARD} mkdir -p /usr/local/bin
$ scp seccomp-test root@${BOARD}:/usr/local/bin
$ ssh -t root@${BOARD} /usr/local/bin/seccomp-test
Showing content of "/proc/cmdline":
console=ttyLP0,115200 root=/dev/mmcblk1p2 rootwait rw selinux=1 enforcing=0
Running "/bin/bash":
```

As you can see, after printing file content, a shell was started. We really want to avoid that.

One solution is to use basic seccomp mechanism. Remember, this will only allow processes to use very basic syscalls, such as `read()` or `write()`. Modify the `seccomp-test.c` source file, and uncomment the call to `enable_seccomp()`. Rebuild, and relaunch.

```
$ make
$ scp seccomp-test root@${BOARD}:/usr/local/bin
```

We do not get much output with SSH, so you should run it directly on the target:

```
$ ssh root@${BOARD} /usr/local/bin/seccomp-test
Killed
```

As expected, the process gets killed. However, what might not be expected is the content of `/proc/cmdline` is no longer shown. You can use `strace` to analyze what syscalls were used, and understand this new behaviour. On the board, you can run:

```
$ strace seccomp-test
```



```
...
seccomp(SECCOMP_SET_MODE_STRICT, 0, NULL) = 0
fstat(1, <unfinished ...>) = ?
+++ killed by SIGKILL +++
[1] 1202806 killed strace seccomp-test
```

As you can see, `printf()` did call `fstat()`. It tends to do that only on the very first call, so this could be avoided by calling `printf()` before enabling seccomp. But we are later calling `close()`, which is also denied by seccomp.

As an alternative, we will use libseccomp to add our own custom filters.

Modify again the source code, defining the `USE_SECCOMP_FILTERS` macro: the `enable_seccomp()` will now use libseccomp to define our custom filters. Some filters were already added by the example code, but you will have to complete this list. Again, you can use `strace` to identify the correct syscalls.

Remember, our goal is to have an application that can successfully show the file content, but cannot execute the shell. After having added all needed filters, you should have the following behaviour:

```
$ seccomp-test
Showing content of "/proc/cmdline":
console=ttyLP0,115200 root=/dev/mmcblk1p2 rootwait rw selinux=1 enforcing=0
Running "/bin/bash":
Bad system call (core dumped)
```

Manipulating SELinux contexts

To demonstrate SELinux use case, we will use a small TFTP server present on the board. This TFTP server has been misconfigured: instead of serving the content of `/srv/tftp/`, it does serve the content of the whole root filesystem (`/`). We can create two files on the board, to demonstrate this behaviour.

```
# echo "Public data" > /srv/tftp/public
# echo "Very secret data" > /root/secret
# chmod 777 /root
```

On the host computer, use a tftp client to download these files. As the tftp server will connect to a random UDP port on the host, you might need to disable some firewalling rules.

```
$ tftp $BOARD -vc get /srv/tftp/public
Connected to 192.168.0.168 (192.168.0.168), port 69
getting from 192.168.0.168:/srv/tftp/public to public [netascii]
Received 13 bytes in 0.0 seconds [3200 bit/s]
$ cat public
Public data
$ tftp $BOARD -vc get /root/secret
Connected to 192.168.0.168 (192.168.0.168), port 69
getting from 192.168.0.168:/root/secret to secret [netascii]
Received 18 bytes in 0.0 seconds [4270 bit/s]
$ cat secret
Very secred data
```

If downloads are failing, you might need to have a look at the tftp server logs:

```
# journalctl -u atftpd -f
```

Before we can use SELinux, we need to relable the filesystem. Run the command


```
$ fixfiles relabel
```

Then reboot the board.

Once we have this working setup, we can start experimenting with SELinux. SELinux enforcement is disabled by default, thanks to the kernel command line. This is a specific setup, made so we can run other labs without having SELinux blocking anything. The first step is then to enable SELinux until the next reboot:

```
# setenforce 1
```

SELinux status can be shown with the `sestatus` command:

```
# sestatus
SELinux status: enabled
SELinuxfs mount: /sys/fs/selinux
SELinux root directory: /etc/selinux
Loaded policy name: standard
Current mode: enforcing
Mode from config file: enforcing
Policy MLS status: disabled
Policy deny_unknown status: allowed
Memory protection checking: actual (secure)
Max kernel policy version: 34
```

Let's manipulate a bit SELinux: first we will look for SELinux users. You can both the list of users and the mapping between Linux and SELinux users.

```
# seinfo -u

Users: 7
   root
   staff_u
   sysadm_u
   system_u
   unconfined_u
   user_u
   xdm
# semanage login -l

Login Name SELinux User

__default__ user_u
root root
```

The first commands, shows existing SELinux users, the second one shows mapping from Linux users. We can see the *root* user is mapped to the *root* SELinux user, while all others are mapped to *user_u*.

Similarly, we can show the corresponding roles:

```
# seinfo -u -x

Users: 7
   user root roles { staff_r sysadm_r system_r };
   user staff_u roles { staff_r sysadm_r };
   user sysadm_u roles sysadm_r;
   user system_u roles system_r;
```



```
   user unconfined_u roles { system_r unconfined_r };
   user user_u roles user_r;
   user xdm roles xdm_r;
```

As our Linux root user is mapped to the SELinux root user, the corresponding line is the first one. Keep in mind this mapping might be different on production systems.

We can list which types can be accessed by *staff_r*, *sysadm_r* or *system_r* roles.

```
# seinfo -rstaff_r -x

Roles: 1
   role staff_r types { auditadm_screen_t bluetooth_helper_t cdrecord_t chfn_t chkpwd_t \
       chromium_naclhelper_t chromium_renderer_t chromium_sandbox_t chromium_t consoletype_t \
       container_engine_t container_t crio_t cronjob_t crontab_t ddclient_t dhcpc_t dirmngr_t
   ...

# seinfo -rsysadm_r -x

Roles: 1
   role sysadm_r types { acngtool_t acpi_t admin_crontab_t aide_t amanda_recover_t apt_t \
       auditadm_screen_t auditctl_t backup_t bacula_admin_t bluetooth_helper_t bootloader_t \
       calamaris_t cdrecord_t certbot_t certwatch_t cgclear_t checkpc_t checkpolicy_t chfn_t
   ...

# seinfo -rsystem_r -x

Roles: 1
   role system_r types { NetworkManager_t abrt_dump_oops_t abrt_handle_event_t abrt_helper_t \
       abrt_retrace_coredump_t abrt_retrace_worker_t abrt_t abrt_upload_watch_t abrt_watch_\
       log_t accountsd_t acct_t acngtool_t acpi_t acpid_t afs_bosserver_t afs_fsserver_t
   ...
```

We can also observe the context of running processes, with `ps -Z`:

```
# ps -Z
  PID CONTEXT STAT COMMAND
    1 system_u:system_r:init_t S {systemd} /sbin/init
    2 system_u:system_r:kernel_t SW [kthreadd]
    3 system_u:system_r:kernel_t SW [pool_workqueue_]
    4 system_u:system_r:kernel_t IW< [kworker/R-rcu_g]
    5 system_u:system_r:kernel_t IW< [kworker/R-sync_]
    6 system_u:system_r:kernel_t IW< [kworker/R-kvfre]
    ...
  310 root:sysadm_r:sysadm_t S -sh
  338 system_u:system_r:kernel_t IW< [kworker/1:2H]
  371 system_u:system_r:kernel_t IW [kworker/u8:1-ev]
  386 system_u:system_r:kernel_t IW [kworker/u8:2-ev]
  389 root:sysadm_r:sysadm_t R {ps} /usr/bin/busybox.nosuid /usr/bin/ps -Z
```

We now want to compare the SELinux context of the two files we created earlier. This can be shown with `ls -Z`.

```
# ls -lZ /srv/tftp/public /root/secret
-rw-r--r--. 1 root root root:object_r:user_home_t 17 Mar 17 09:27 /root/secret
-rw-r--r--. 1 root root root:object_r:var_t 12 Mar 17 09:31 /srv/tftp/public
```


As you can see, the two files were automatically labelled differently, based on the folder they were created in. As SELinux is now enabled, we can again try to dowload them using tftp. You can run the same commands as previously on the host

```
$ rm -f public secret
$ tftp $BOARD -vc get /srv/tftp/public
Connected to 192.168.0.168 (192.168.0.168), port 69
getting from 192.168.0.168:/srv/tftp/public to public [netascii]
Received 13 bytes in 0.0 seconds [3200 bit/s]
$ cat public
Public data
$ tftp $BOARD -vc get /root/secret
Connected to 192.168.0.168 (192.168.0.168), port 69
getting from 192.168.0.168:/root/secret to secret [netascii]
Error code 1: File not found
```

Thanks to the SELinux policies, the tftp server no longer has access to the content of `/root`.

Let's pretend this access is legitimate. We could either modify SELinux rules or the file context. For demonstration purpose, we can try this later solution, changing the context of `/root/secret` to be the same as `/srv/tftp/public`.

```
# chcon root:object_r:var_t /root/secret
# ls -lZ /root/secret
-rw-r--r--. 1 root root root:object_r:var_t 17 Mar 17 09:27 /root/secret
```

And then try again to download the file:

```
$ tftp $BOARD -vc get /root/secret
Connected to 192.168.0.168 (192.168.0.168), port 69
getting from 192.168.0.168:/root/secret to secret [netascii]
Received 18 bytes in 0.0 seconds [4580 bit/s]
```

The `restorecon` command can then be used to revert to a sane context:

```
# restorecon /root/secret
root@freiheit93:~# ls -lZ /root/secret
-rw-r--r--. 1 root root root:object_r:user_home_t 17 Mar 17 09:27 /root/secret
```

Using systemd to restrict a daemon's access to resources

We now want to run an application with systemd, using systemd unit configuration to filter the operations the application can make.

This application is again an artificial one, trying to mimic one doing some legit operations, but also doing some unwanted ones, because of bugs or security breaches exploited by an attacker.

As legitimate operation, the application does:

- Adds some entropy to `/dev/urandom`: this does require the `CAP_SYS_ADMIN` capability. Note, this is mostly used for demonstration. The added entropy is based on current system time: this is not a particularly good entropy source and should not be used in real systems.
- Runs `ping -N`. This is again an option requiring the `CAP_NET_RAW` capability.

The application also simulates two bogus operations:

- It writes some data to a file in `/var/cache`.
- It modifies the machine hostname.

The $HOME/security-imx93-frdm-labs/userland/systemd contains both the application source code and a sample systemd unit. Cross-compile the application, and deploy it on the target, to the /usr/local/bin folder; then deploy the service file to /etc/systemd/system/.

```
$ cd $HOME/security-imx93-frdm-labs/userland/systemd
$ make
$ scp systemd-test root@${BOARD}:/usr/local/bin
$ scp training-systemd-test.service root@${BOARD}:/etc/systemd/system/
```

Now, start the service and monitor the log output.

```
# systemctl daemon-reload
# systemctl restart training-systemd-test --no-block
# journalctl -u training-systemd-test -f -n 50
Starting training-systemd-test.service - Security training systemd demo...
Adding entropy count
Writing data to "/var/cache/security_training_systemd"
Setting hostname to "changed-hostname"
hostname: changed-hostname
Executing ping command
PING google.com (172.217.22.206) 56(84) bytes of data.
64 bytes from muc11s01-in-f14.1e100.net (172.217.22.206): icmp_seq=1 ttl=111 time=14.1 ms
--- google.com ping statistics ---
1 packets transmitted, 1 received, 0 packet loss, time 0msrtt min/avg/max/mdev =
    14.112/14.112/14.112/0.000 mstraining-systemd-test.service: Deactivated
    successfully.Finished training-systemd-test.service - Security training systemd demo.
```

Looking at the log, we have some good and bad news: adding entropy and running ping did succeed. But so did changing the hostname and writing data to /var/cache/security_training_systemd. We can verify this:

```
# cat /var/cache/security_training_systemd
Hello!
# cat /proc/sys/kernel/hostname
changed-hostname
```

Let's revert these changes:

```
# rm /var/cache/security_training_systemd
# echo "myboard" > /proc/sys/kernel/hostname
```

To prevent these changes, we can limit the permissions of the process launched by systemd. First, instead of running as root, we should be using an unprivileged user account, thanks to the User= directive. We can ask systemd to create an ephemeral one, with DynamicUser=.

Modify the unit file with systemctl edit --full training-systemd-test to add following lines into the [Service] section. Then restart the service.

```
User=someuser
DynamicUser=true
```

```
# systemctl restart training-systemd-test --no-block
# journalctl -u training-systemd-test -f -n 50
Starting training-systemd-test.service - Security training systemd demo...
Writing data to "/var/cache/security_training_systemd"
Failed to open "/var/cache/security_training_systemd": Read-only file system
```



```
training-systemd-test.service: Main process exited, code=exited, status=1/FAILURE
training-systemd-test.service: Failed with result 'exit-code'.
Failed to start training-systemd-test.service - Security training systemd demo.
```

The application can no longer write to `/var/cache/`, that's great, but as a consequence it immediately stops. We can ask systemd to mount a tmpfs on /var/cache/, so the application can still write in there, without impacting the system. Modify the service file, and add the following line:

```
TemporaryFileSystem=/var/cache:mode=0777
```

Then restart the application, and look for logs.

```
# systemctl restart training-systemd-test --no-block
# journalctl -u training-systemd-test -f -n 50
Starting training-systemd-test.service - Security training systemd demo...
Writing data to "/var/cache/security_training_systemd"
Adding entropy count
Failed to add entropy: Operation not permitted
Setting hostname to "changed-hostname"
Failed to set hostname: Operation not permitted
hostname: myboard
Executing ping command
ping: socktype: SOCK_RAW
ping: socket: Operation not permitted
ping: => missing cap_net_raw+p capability or setuid?
training-systemd-test.service: Main process exited, code=exited, status=2/INVALIDARGUMENT
training-systemd-test.service: Failed with result 'exit-code'.
Failed to start training-systemd-test.service - Security training systemd demo.
```

So now, the application can happily write to `/var/cache/security_training_systemd`. You can verify that this file was not really created on your system, but on some tmpfs. We can also see the hostname was not changed. However, we now have two issues: adding entropy failed, and ping complains: `WARNING: failed to set mark: 42: Operation not permitted`. Both issues come from missing capabilities, as we are no longer running as root. We can modify the configuration to add the required ambient capabilities and restart the service.

```
AmbientCapabilities=CAP_NET_RAW CAP_SYS_ADMIN
```

```
# systemctl restart training-systemd-test --no-block
# journalctl -u training-systemd-test -f -n 50
Starting training-systemd-test.service - Security training systemd demo...
Writing data to "/var/cache/security_training_systemd"
Adding entropy count
Setting hostname to "changed-hostname"
hostname: changed-hostname
Executing ping command
PING google.com (172.217.22.78) 56(84) bytes of data.
64 bytes from tzpara-am-in-f14.1e100.net (172.217.22.78): icmp_seq=1 ttl=112 time=14.1 ms
--- google.com ping statistics ---
1 packets transmitted, 1 received, 0 packet loss, time 0msrtt min/avg/max/mdev =
    14.122/14.122/14.122/0.000 mstraining-systemd-test.service: Deactivated
    successfully.Finished training-systemd-test.service - Security training systemd demo.
```

With the added capabilities, both the ping and entropy adding operation succeed, but we also granted the possibility to change the hostname. One solution is to use the `ProtectHostname=` option to forbid this.

```
ProtectHostname=on
```

```
# systemctl restart training-systemd-test --no-block
# journalctl -u training-systemd-test -f -n 50
Starting training-systemd-test.service - Security training systemd demo...
Writing data to "/var/cache/security_training_systemd"
Adding entropy count
Setting hostname to "changed-hostname"
Failed to set hostname: Operation not permitted
hostname: myboard
Executing ping command
PING google.com (172.217.22.78) 56(84) bytes of data.
64 bytes from tzpara-am-in-f14.1e100.net (172.217.22.78): icmp_seq=1 ttl=112 time=13.8 ms
--- google.com ping statistics ---
1 packets transmitted, 1 received, 0 packet loss, time 0msrtt min/avg/max/mdev =
    13.830/13.830/13.830/0.000 mstraining-systemd-test.service: Deactivated
    successfully.Finished training-systemd-test.service - Security training systemd demo.
```

Alternatively, one could use seccomp to prevent call to `sethostname()`. Systemd `SystemCallFilter=` configuration allows to do so easily.

```
SystemCallFilter=~sethostname
```

An interesting difference is the process is killed, except if we also use `SystemCallErrorNumber=`.

```
# systemctl restart training-systemd-test --no-block
# journalctl -u training-systemd-test -f -n 50
Starting training-systemd-test.service - Security training systemd demo...
Writing data to "/var/cache/security_training_systemd"
Adding entropy count
Setting hostname to "changed-hostname"
training-systemd-test.service: Main process exited, code=killed, status=31/SYS
training-systemd-test.service: Failed with result 'signal'.
Failed to start training-systemd-test.service - Security training systemd demo.
```


Software Bill of Materials (SBoM)

Objective: Generate and analyze the image's SBoM.

Prerequisites

We are going to use `sbom-cve-check` to analyze our SBoM.

```
(.venv) $ pip install sbom-cve-check[extra]
```

The default SPDX2.2 SBoM

Yocto is able to generate SPDX SBoMs, but the version we are currently using (Scarthgap) is using the 2.2 standard by default.

This SBoMs are generated during the build of each recipe. You can find them under `build/tmp-glibc/deploy/spdx/2.2`

```
$ tree -d build/tmp-glibc/deploy/spdx/2.2/freiheit93/
|-- packages
|-- recipes
`-- runtime
```

One downside of SPDX2.2 is, that the SBoM is fragmented into a large number of files. This makes it less obvious to quickly get an overview:

```
$ find build/tmp-glibc/deploy/spdx/2.2/freiheit93/ -type f | wc -l
1952
```

Generating an SPDX3 SBoM

To be able to generate an SPDX3 SBoM, we need to add the following lines in `board_setup/meta-security-training/conf/distro/security-training.conf`:

```
INHERIT:remove = "create-spdx"
INHERIT:append = " create-spdx-3.0"
```

(notice the space on the second line)

In order to generate the full SBoM, Yocto needs to rebuild the image (but it will use the cache).

```
$ cd board_setup
$ . openembedded-core/oe-init-build-env
$ bitbake security-training-image
```

You should now see a `build/tmp-glibc/deploy/spdx/3.0.1` directory. The "main" SBoM, the one for the image, is under `build/tmp-glibc/deploy/spdx/3.0.1/freiheit93/image/security-training-image-freiheit93-image.spdx.json`

This SBoM is rather short, so it can be reviewed manually, but it is high-level. However, the SPDX3 directory, `build/tmp-glibc/deploy/spdx/3.0.1` includes an SBoM for the rootFS. That one is not so short:

```
$ jq . build/tmp-glibc/deploy/spdx/3.0.1/freiheit93/rootfs/\
```



```
    security-training-image-freiheit93-rootfs.spdx.json | wc -l
23938
```

Analyzing this SBoM manually is not a good option, even though the system we are using is a rather bare-bones one.

Analyzing the SBoM

In this section, we are going to use sbom-cve-check to analyze the SBoM generated and determine which CVEs affect our target's rootFS.

```
$ sbom-cve-check --sbom-type spdx3 -S build/tmp-glibc/deploy/images/freiheit93/\
    security-training-image-freiheit93.rootfs.spdx.json -f summary -o cve_summary_novex
```

We can now generate the VEX, by again modifying `board_setup/meta-security-training/conf/distro/security-training.conf`:

```
INHERIT:remove = "create-spdx"
INHERIT:append = " create-spdx-3.0"
INHERIT:append = " vex"
```

```
$ cd board_setup
$ . openembedded-core/oe-init-build-env
$ bitbake security-training-image
```

This should have created the VEX manifest in `build/tmp-glibc/deploy/images/freiheit93/security-training-image-freiheit93.rootfs.json`, which we can now use to add context to `sbom-cve-check`:

```
$ sbom-cve-check --sbom-type spdx3 \
       -S build/tmp-glibc/deploy/images/freiheit93/\
           security-training-image-freiheit93.rootfs.spdx.json \
       --yocto-vex-manifest build/tmp-glibc/deploy/images/freiheit93/\
           security-training-image-freiheit93.rootfs.json \
       -f summary -o cve_summary_vex
```

Compare both outputs to find out the impact of the annotations built into Yocto.

Take a look at CVE-2024-9287. Does it actually apply here? If not, try to correct `sbom-cve-check`'s assessment.

The `summary` format does not indicate **why** `sbom-cve-check` concluded that this CVE was applicable. You might want to generate a `yocto-cve-check-manifest` output.

Patching a CVE reported by sbom-cve-check

In this section, we are going to fix one of the CVEs. Specifically, CVE-2026-33317.

Take a look at the CVE, patch it, and make sure it no longer appears in `sbom-cve-check`'s report.

A/B updates

Objective: Learn to configure RAUC and deploy A/B updates to our board

Prerequisites

For this stage, we are going to run RAUC also on the development host, to generate the update bundle(s). Run the following command:

```
$ sudo apt install -y rauc squashfs-tools
```

A/B update partition setup

To avoid rebuilds and reflashes, the system already has an A/B setup. You can confirm this and look at the details in `board_setup/meta-security-training/wic/security-training-image.freiheit93.wks.in`

Configuring RAUC on the platform

Systemd should have tried to start the RAUC daemon:

```
# systemctl status rauc
```

This should show that the service has not been started. You can look at the logs to figure out why:

```
# journalctl -xeu rauc.service
```

RAUC needs a few things it currently does not have to function:

- its main configuration file
- the keyring which will be used to verify the bundle's signature
- functional `fw_printenv` and `fw_setenv` commands

Setting up fw_printenv/fw_setenv

Running `fw_printenv` should currently fail:

```
$ fw_printenv
Cannot initialize environment
```

The reason for this is that the configuration is missing. Considering the A/B setup as defined in `board_setup/meta-security-training/wic/security-training-image.freiheit93.wks.in`, create the configuration file in `/etc/fw_env.config`.

The keyring

RAUC does not support **unsigned** updates, by design. Therefore, we need a key pair for our RAUC updates. We'll also want a certificate, that we'll deploy to the target for RAUC to use to verify the signature. We could use:

- A self-signed certificate
- the root CA certificate

The first will mean less flexibility in handling certificates, while the second risks exposing your root key, which is unnecessary. Fortunately, we already have the key pair (and certificate) we need: the bundle will be built on the host, we'll just use its key pair and certificate.

We could use `host.crt` directly as the keyring, but using `cacert.pem` is better, as we expect that certificate to change very infrequently, and it is the root of trust of our PKI.

In a real case, we would possibly have at least one child CA, and therefore one intermediate certificate, but we'll keep things simple here.

Building and installing the bundle

RAUC bundles use a configuration file called a manifest. This is what you'll be writing in this section.

Once you're done, you can build the update bundle:

```
$ mkdir bundle
# edit bundle/manifest.raucm
$ cd bundle
$ mkdir root
$ echo "The rootfs has been updated" > root/UPDATED.txt
$ cp ../../board_setup/build/tmp-glibc/deploy/images/freiheit93/\
    security-training-image-freiheit93.rootfs.tar.gz rootfs.tar.gz
$ gzip -d rootfs.tar.gz
$ tar -uvf rootfs.tar ./root
$ rm -rf root
$ gzip rootfs.tar
$ rauc bundle . update.bundle --signing-keyring=../ca/cacert.pem --key=../host/hostkey.pem \
    --cert=../host/host.crt
Creating 'verity' format bundle
rauc-Message: Keyring given, doing signature verification
rauc-Message: Verified inline signature by 'C = FR, ST = Some-State, O = Bootlin security \
    training, CN = host computer'
$ scp update.bundle root@${BOARD}:
```

Disable SELinux

And then install it and reboot:

```
$ rauc install update.bundle
```

You might get SELinux errors during bundle installation: this can be fixed by disabling SELinux in the kernel command line with `selinux=0`.

RAUC might fail because the system's date is too long in the past for the certificate we generated to already be valid.

```
$ date -s 'YYYY-MM-DD'
```

Alternatively, you can also instruct RAUC to use the time at which the bundle was signed instead of the system time, by adding

```
use-bundle-signing-time=true
```

to the `[keyring]` section of `/etc/rauc/system.conf`.

Even once the bundle is correctly installed, the board should not show UPDATED.txt. Running `mount` should help figure out why.

You can also look at the U-Boot environment to see what RAUC attempted to do:


```
$ fw_printenv
BOOT_ORDER=rootB
BOOT_rootB_LEFT=3
```

U-Boot configuration

We now need to change the way U-Boot boots up the system. You can take some inspiration from this example.