How to configure OP-TEE

Contents

1 Purpose

This article describes the configuration and process used for building several OP-TEE components from sources and deploying them the target.

The build example is based on the OpenSTLinux Developer Package or Distribution Package, and also presents build instructions for a bare environment.

2 Overview

OP-TEE is a trusted execution environment for Arm®v7-A and Arm®v8-A platforms. OP-TEE is made of several components described in OP-TEE architecture overview.

OP-TEE components generate boot images and files stored in the filesystem embedded in the target.

- OP-TEE OS generates 3 boot image files to be loaded in the platform boot media, in the predefined partitions. The generated boot images include a STM32 binary header enabling the use of the authenticated boot and flash programming facilities.
- OP-TEE client (package optee_client) can be built to generate non-secure services for the OP-TEE OS. The files generated from optee_client build are stored in the embedded filesystem.
- OP-TEE project releases other packages intended for test and demonstration. These can be built and embedded in the target filesystem. Building optee_examples and optee_test generates client and trusted applications together with libraries which are all stored in the target filesystem. Note the OP-TEE Linux driver is built into the Linux kernel image and is part of the OP-TEE ecosystem.

OP-TEE can be embedded as BL32 in the STM32 MPU platforms for the ST trusted configuration.

⚠ **OP-TEE boot images must be embedded in the FIP binary that is loaded by BL2 and can be automatically authentified**

3 OP-TEE core configuration

3.1 STM32MP15

OP-TEE OS requires more than 256Ko RAM. SYSRAM is only 256Ko, the OP-TEE core must use the pager mode to extends memory using DDR.

OP-TEE OS is loaded at the beginning of the SYSRAM by the FSBL. The OP-TEE could extend the memory to the full SYSRAM. As pager is used, a second part of the code is loaded in DDR (pageable part) in a restricted secure accessible area.

OP-TEE OS manages low power mode by saving its context in DDR (encrypted area) that is restored by a protected execution code saved in secured backup SRAM.

OP-TEE OS implements the following secure services:

- PSCI services
  - System reset
  - CPU hotplug
  - Low power
- SCMI services
  - Clock management
  - Reset management
- OTP access services
- PWR services
  - PWR regulator access for non secure IPs
  - Wakeup source management
- RCC services Limited access
  - OPP request management
  - Calibration triggering

4 Build with the Distribution Package

The Distribution Package provides means to build the following OP-TEE components from their related bitbake target:

```
PC $> bitbake optee-os-stm32mp                    # OP-TEE core firmware
PC $> bitbake optee-os-sdk-stm32mp                # OP-TEE development kit for Trusted
Applications
PC $> bitbake optee-client                        # OP-TEE client
PC $> bitbake optee-test                          # OP-TEE test suite (optional)
PC $> bitbake optee-examples                      # TA and CA examples
```

Distribution Package build process includes fetching the source files, compiling them and installing them to the target images.

The Yocto recipes for the OP-TEE packages can be found in:

```
meta-st/meta-st-stm32mp/recipes-security/optee/optee-os-stm32mp*
meta-st/meta-st-openstlinux/recipes-security/optee/optee-client*
meta-st/meta-st-openstlinux/recipes-security/optee/optee-examples*
meta-st/meta-st-openstlinux/recipes-security/optee/optee-test*
```

5 Build with the Developer Package or a Bare Environment

Both Developer Package and bare build environments expect you to fetch/download the OP-TEE package source file trees in order to build the embedded binary images.

The instruction set below assumes all OP-TEE package source trees are available in the base directory referred as <sources>/. The source files are available from the github repositories:

```
PC $> cd <sources>/
PC $> git clone https://github.com/STMicroelectronics/optee_os.git
PC $> git clone https://github.com/OP-TEE/optee_client.git
PC $> git clone https://github.com/OP-TEE/optee_test.git
PC $> git clone https://github.com/linaro-swg/optee_examples.git
PC $> ls -1 <sources>/
optee_client
optee_examples
optee_os
optee_test
PC $>
```

The STM32 MPU platform may not be fully merged in the official OP-TEE repository [1] hence the URL provided above refers to the ST distribution [2]

5.1 Initialize the cross compile environment

The compilation toolchain provided by the Developer Package can be used, refer to Setup Cross Compile Environment.

Alternatively other bare toolchains can be used to build the OP-TEE **secure** parts. In such case, the instructions below expect the toolchain to be part of the **PATH** and its prefix is defined by **CROSS_COMPILE**. One can use something like:

```
PC $> export PATH=<path-to-toolchain>:$PATH
PC $> export CROSS_COMPILE=<toolchain-prefix>-
```

5.2 Build OP-TEE OS

5.2.1 Developer Package SDK

The OP-TEE OS can be built from the Developer Package **Makefile.sdk** script that is present in the tarball. It automatically sets the proper configuration for the OP-TEE OS build. To build from shell command:

```
PC $> make -f Makefile.sdk CFG_EMBED_DTB_SOURCE_FILE=<board_dts_file_name>.dts
```

5.2.2 Bare Environment

Alternatively one can also build OP-TEE OS based a bare cross compilation toolchains, for example for the stm32mp157c-ev1 board:

```
PC $> cd <optee-os>
PC $> make PLATFORM=stm32mp1 \
           CFG_EMBED_DTB_SOURCE_FILE=stm32mp157c-ev1.dts \
           CFG_TEE_CORE_LOG_LEVEL=2 O=out all
```

5.2.3 Generated Files

The 3 OP-TEE boot images are generated at following paths:

```
<optee-os>/out/core/tee-header_v2.bin
<optee-os>/out/core/tee-pageable_v2.bin
<optee-os>/out/core/tee-pager_v2.bin
```

One can get the configuration directives used for the build are available in this file:

```
<optee-os>/out/conf.mk
```

The build also generates a development kit used to build Trusted Application binaries:

```
<optee-os>/out/export-ta_arm32/
```

5.2.4 Details on build directives

Mandatory directives to build OP-TEE OS:

- **PLATFORM=<platform>**
  - Ex: **PLATFORM=stm32mp1**
- **CFG_EMBED_DTB_SOURCE_FILE=<device-tree-source-file>**: in-tree (core/arch/arm/dts/) device tree filename with its **.dts** extension.

Common optional directives:

- **CFG_TEE_CORE_DEBUG={n|y}**: disable/enable debug support
- **CFG_TEE_CORE_LOG_LEVEL={0|1|2|3|4}**: define the trace level (0: no trace, 4: overflow of traces)
- **CFG_UNWIND={n|y}**: disable/enable stack unwind support

For ecosystem release v3.0.0 ℹ compatibility

It is still possible to generate the the STM32 binary files with an option flag:

- CFG_STM32MP15x_STM32IMAGE=1: Generate the STM32 files for ecosystem release v3.0.0 ℹ compatibility.

Note: internal memory size constrains the debug support level that can be provided.

5.2.5 Troubleshoot

The Developer Package toolchain may report dependency error in the traces such as:

```
PC $> make PLATFORM=stm32mp1 ...
arm-ostl-linux-gnueabi-ld.bfd: unrecognized option '-Wl,-O1'
arm-ostl-linux-gnueabi-ld.bfd: use the --help option for usage information
core/arch/arm/kernel/link.mk:165: recipe for target 'out/arm-plat-stm32mp1/core/tee.elf' failed
make: *** [out/arm-plat-stm32mp1/core/tee.elf] Error 1
```

This is linked to default CFLAGS and LDFLAGS exported by SDK. Just remove them from the environment and rebuild

```
PC $> unset -v CFLAGS LDFLAGS
```

Other reported issues:

```
PC $> make PLATFORM=stm32mp1 ...
arm-openstlinux_weston-linux-gnueabi-ld.bfd: cannot find libgcc.a: No such file or
directory
```

To overcome the issue, add the directive **LIBGCC_LOCATE_CFLAGS=--sysroot=$SDKTARGETSYSROOT**. I.e:

```
PC $> cd <optee-os>
PC $> make PLATFORM=stm32mp1 \
           CFG_EMBED_DTB_SOURCE_FILE=stm32mp157c-ev1.dts \
           CFG_TEE_CORE_LOG_LEVEL=2 \
           LIBGCC_LOCATE_CFLAGS=--sysroot=${SDKTARGETSYSROOT} \
           O=out all
```

5.3 Build commands for other OP-TEE components

This section describes how the several OP-TEE components (excluding OP-TEE OS described in above section) can be built. All those components generate files targeting the embedded Linux OS based filesystem (i.e the rootfs). These files are the secure Trusted Applications (TAs) binaries as well as non-secure Client Applications (CAs), libraries and test files.

There are several ways to build the OP-TEE components. The examples given below refer to OP-TEE client, test and examples source file tree paths as <optee-client>, <optee-test> and <optee-examples>.

Building these components expect, at least for the trusted applications, that the OP-TEE OS was built and the generated TA development kit is available at <optee-os>/out/export-ta_arm32/.

It is recommended to use CMake for building the Linux userland part whereas secure world binaries (TAs) must be build from their GNU makefiles as the OP-TEE project has not yet ported the secure world binaries build process over CMake.

5.3.1 Build the secure components

Build the TAs: This step expects OP-TEE OS is built to generate the 32bit TA development kit. Assuming OP-TEE OS was built at path <optee-os>/out, the TA development kit is available from path <optee-os>/out/export-ta_arm32/.

Instructions below build and copy the Trusted Application binaries to a local **./target/** directory that can be used to populate the target filesystem.

```
PC $> export TA_DEV_KIT_DIR=$PWD/optee_os/out/export-ta_arm32
PC $> mkdir -p ./target/lib/optee_armtz
PC $> for f in optee_test/ta/*/Makefile; do \
            make -C `dirname $f` O=out; \
            cp -f `dirname $f`/out/*.ta ./target/lib/optee_armtz; \
      done
```

Content in local directory **./target/** are the TA binary files:

```
PC $> tree target/
target
└── lib
    └── optee_armtz
      ├── 614789f2-39c0-4ebf-b235-92b32ac107ed.ta
      ├── 731e279e-aafb-4575-a771-38caa6f0cca6.ta
      └── (...)
```

These files need to be copied to the the target filesystem.

5.3.2 Build the non-secure components

Download the OP-TEE source files in a base directory and create a **CMakeLists.txt** file in the base directory that lists all package to be built through CMake. For example:

```
PC $> ls
optee_client
optee_examples
optee_os
optee_test
CMakeLists.txt
PC $> cat CMakeLists.txt
add_subdirectory (optee_client)
add_subdirectory (optee_test)
add_subdirectory (optee_examples)
PC $>
```

From base directory, run **cmake** then **make**. The example below also creates the tree file system **./target/** that is populated with files generated that need to be installed in the target file system.
Note this examples also sets the toolchain environment:

```
PC $> cmake -DOPTEE_TEST_SDK=$PWD/optee_os/out/export-ta_arm32 \
            -DCMAKE_INSTALL_PREFIX=  -DCMAKE_BUILD_TYPE=Release -DBUILD_SHARED_LIBS=y
PC $> make
PC $> make DESTDIR=target install
```

Note the empty **CMAKE_INSTALL_PREFIX** value to get thing installed from root **/**, not from **/usr/**. **DESTDIR=target** makes the embedded files be populated in the local **./target/** directory.

Note also that stm32mp1 expects tool **tee-supplicant** to be located in directory **/usr/bin** whereas CMake installs it in directory **/usr/sbin**. To overcome this issue, one can build a link to the effective location, i.e:

```
PC $> ln -s ../bin/tee-supplicant target/sbin/tee-supplicant
```

Once done, local directory **./target/** contains the files to be copied in the target filesystem.

```
PC $> tree target/
target/
├── bin
│   ├── benchmark
│   ├── optee_example_acipher
│   ├── optee_example_aes
│   ├── optee_example_hello_world
│   ├── optee_example_hotp
│   ├── optee_example_random
│   ├── optee_example_secure_storage
│   ├── tee-supplicant
│   └── xtest
├── include
│   ├── tee_bench.h
│   ├── tee_client_api_extensions.h
│   ├── tee_client_api.h
│   └── teec_trace.h
├── lib
│   ├── libteec.so -> libteec.so.1
│   ├── libteec.so.1 -> libteec.so.1.0.0
│   ├── libteec.so.1.0.0
│   └── optee_armtz
│       └── (...)                 # This directory was previously filled with TAs
└── sbin
    └── tee-supplicant -> ../bin/tee-supplicant
```

6 Update OP-TEE boot images

OP-TEE boot images are part of the FIP binary.
The next step to deploy the OP-TEE OS is to update the FIP binary following the FIP update process.

7 Update OP-TEE Linux files

7.1 Update on board

The other OP-TEE images are stored in the target filesystem.

For example, if using an SD card as target boot media, the card can be plugged in its PC card reader and the images copied. The files can be simply copied into the mounted rootfs.

7.2 Update in a SD card

The OP-TEE files that need to be copied to the target filesystem were installed in a local directory **./target/**.

They can now be copied to the target SD card rootfs partition once the SD card is plugged to the host computer and its filesystems are mounted in the host, i.e

```
PC $> cp -ar target/* /media/$USERNAME/rootfs/
```

8 Update your boot device (including SD card on the target)

Refer to the STM32CubeProgrammer documentation to update your target.

9 References

- https://github.com/OP-TEE/optee_os
- https://github.com/STMicroelectronics/optee_os

Open Portable Trusted Execution Environment

Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Operating System

Linux® is a registered trademark of Linus Torvalds.

Boot Loader stage 3-2

Microprocessor Unit

Boot Loader stage 2

Random Access Memory (Early computer memories generally hadserial access. Memories where any given address can be accessed when desired were then called "random access" to distinguish them from the memories where contents can only be accessed in a fixed order. The term is used today for volatile random-acces ssemiconductor memories.)

Doubledata rate (memory domain)

First Stage Boot Loader

Power State Coordination Interface

Central processing unit

One Time Programmed

Operating Performance Point (link to voltage and frequency scaling)

Trusted Application

Software development kit (A programming package that enables a programmer to develop applications for a specific platform.)

Device Tree Binary (or Blob)

Trusted Execution Environment

Firmware Image Package is a packaging format used by TF-A

SD memory card (https://www.sdcard.org)