Category:OP-TEE secure OS

Contents

1. Category:OP-TEE secure OS 3
2. How to configure OP-TEE 4
3. How to debug OP-TEE 17
4. OP-TEE overview 23

Stable: 17.06.2020 - 15:26 / Revision: 04.05.2020 - 13:50

Stable: 17.06.2020 - 10:20 | Revision: 04.06.2020 - 13:30

A quality version of this page, approved on *17 June 2020*, was based off this revision.

This category groups together all articles related to the **OP-TEE** secure OS.

It is recommended to first read the OP-TEE overview article.

Pages in category "OP-TEE secure OS"

The following 3 pages are in this category, out of 3 total.

- OP-TEE overview
- How to configure OP-TEE
- How to debug OP-TEE

Stable: 20.05.2021 - 15:08 / Revision: 20.05.2021 - 15:03

A quality version of this page, approved on *20 May 2021*, was based off this revision.

Contents

1 Purpose 5
2 Overview 6
3 OP-TEE core configuration 7
3.1 STM32MP15 7
4 Build with the Distribution Package 8
5 Build with the Developer Package or a Bare Environment 9
5.1 Initialize the cross compile environment 9
5.2 Build OP-TEE OS 9
5.2.1 Developer Package SDK 9
5.2.2 Bare Environment 10
5.2.3 Generated Files 10
5.2.4 Details on build directives 10
5.2.5 Troubleshoot 10
5.3 Build commands for other OP-TEE components 11
5.3.1 Build the secure components 11
5.3.2 Build the non-secure components 12
6 Update OP-TEE boot images 14
7 Update OP-TEE Linux files 15
7.1 Update on board 15
7.2 Update in a SD card 15
8 Update your boot device (including SD card on the target) 16
9 References 17

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.

Warning

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 $>
```

Warning

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
- **CFG_STM32_BSEC_WRITE=1**: Enable the program/write fuses capabilities (default disabled to avoid briking the chip)

For ecosystem release v3.0.0 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 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

Stable: 30.03.2021 - 07:28 / Revision: 29.03.2021 - 19:55

A quality version of this page, approved on *30 March 2021*, was based off this revision.

Contents

1 Purpose 18
2 Debugging OP-TEE core 19
2.1 OP-TEE Version number 19
2.2 Embedded assertions 19
2.3 Debug with traces 19
2.4 Stack unwind support 20
2.5 Debug with GDB 20
2.5.1 Debug boot sequence 20
2.5.2 Debugging during runtime execution 21
3 Debugging OP-TEE trusted applications 22
3.1 Embedded assertions 22
3.2 Debug with traces 22
3.3 Stack unwind support 22
4 References 23

1 Purpose

This article explains how to debug the OP-TEE secure world binaries.
This debug information is specifically linked to the CPU secure state (Arm® TrustZone®).

The OP-TEE secure world binaries include OP-TEE core (privilege firmware) and OP-TEE trusted applications and libraries (user space context):

- There are two main ways to debug OP-TEE core: using embedded traces, or using JTAG/SWD to access the secure world.

The focus here is on the solution integrated in OpenSTLinux: debug over GDB (ST-LINK or JTAG/SWD based).

- The OP-TEE trusted applications and libraries provide debug support relying on embedded traces only.

Debugging the secure userland binaries through JTAG/SWD resources is not recommended, and OP-TEE does not provide any means to embed a GDB server in the secure world.

Warning

This article focuses on OP-TEE debug.
Refer to STM32MP1 Platform trace and debug environment overview article for more generic information.

2 Debugging OP-TEE core

2.1 OP-TEE Version number

The starting point for debugging OP-TEE core is to identify the OP-TEE version embedded in the target. A version identifier is displayed on the console with the following format:

```
I/TC: OP-TEE version:<tag> #<buildcount> <date> <arch>
```

that is:

```
I/TC: OP-TEE version: openstlinux-19-01-11-10-g56ef3b0 #1 Wed Jan 30 09:12:56 UTC 2019 arm
```

2.2 Embedded assertions

OP-TEE core can embed debug assertions that panic the system when the tested condition is not met. Embedded assertions are implemented using the **assert()** function, that is, in the following code snippet, the system panics if **argument1** is negative, while the function returns the decremented value of the input argument:

```
static int increment_argument(int argument)
{
   assert(argument > 0);
   return argument - 1;
}
```

Assertions are embedded (or not) depending on the configuration directive **CFG_TEE_CORE_DEBUG={n|y}** when the OP-TEE core is built.

2.3 Debug with traces

OP-TEE provides trace message support and a configurable trace level build directive **CFG_TEE_CORE_LOG_LEVEL={0|1|2|3|4}**. This directive defines the trace levels that are embedded inside the firmware and output through the OP-TEE console. A low value reduces the memory footprint of the firmware and increases its runtime performance.

Value	Name	Description with related macros
0	-	All trace messages are disabled
1	Error trace level	Only non-tagged and error trace messages are embedded: MSG(), MSG_RAW(), EMSG(), EMSG_RAW()
2	Info trace level	+ info trace messages: IMSG(), IMSG_RAW()
3	Debug trace level	+ debug trace messages: DMSG(), DMSG_RAW()

Value	Name	Description with related macros
4	Flow trace level	+ flow trace messages: FMSG(), FMSG_RAW()

Warning

Concurrent enabling of all debug supports may not be possible due to the internal memory size constraint. If required, change some **DMSG()** into **MSG()** to force messages to output even at a low verbosity level.

Traces and errors are available on the console defined in the chosen node of the device tree by the stdout-path property:

```
chosen {
        stdout-path = "serial0:115200n8";
};
```

More information about OP-TEE build and update is available in the How to configure OP-TEE article.

2.4 Stack unwind support

The OP-TEE core can trace the execution backtrace when its panics. The execution backtrace allows analysis of the execution call sequence that led to the panic. The OP-TEE OS provides tools to analyse such backtraces based on the OP-TEE core ELF file generated at build time.

Backtrace unwind is embedded (or not) depending on the configuration directive **CFG_UNWIND={y|n}** Note that backtrace unwind increases the size of the OP-TEE core firmware. When the OP-TEE core executes from a small secure RAM, enabling stack unwind penalizes the OP-TEE core performance.

More information is available from the OP-TEE abort-dumps documentation[1].

2.5 Debug with GDB

The Debug OpenSTLinux BSP Components with GDB article describes how to set up the GDB / OpenOCD environment. OP-TEE can be debugged through JTAG/SWD using an ST-LINK or the JTAG/SWD output, depending on the target board.

Note that when the OP-TEE core executes in a small secure memory with the support of its pager (case build directive **CFG_WITH_PAGER=y**), use of hardware breakpoints rather than software breakpoints is highly recommended. Since most of the OP-TEE core instructions are dynamically loaded into the small secure memory, a loaded software breakpoint is likely to be discarded when the OP-TEE pager wipes memory content to load other OP-TEE core pages.

When OP-TEE executes in the large main memory (case build directive **CFG_WITH_PAGER** is disabled), all OP-TEE core resources are resident. In this case, hardware breakpoints as well as software breakpoints can be used without any issues.

2.5.1 Debug boot sequence

Load symbols to the target offset:

```
(gdb) add-symbol-file <path_to_build_folder>/tee.elf <load_address>
```

OP-TEE load address is available from the generated **tee-init_load_addr.txt** file.
It can also be found in the generated **tee.map** file:

```
...
Linker script and memory map

                0x000000002ffc0000                . = 0x2ffc0000
                0x0000000000000001                ASSERT (0x1, text start should align to
32bytes)
                0x000000002ffc0000                __text_start = .
                0x000000002ffc0000                __flatmap_unpg_rx_start =
((__text_start / 0x1000) * 0x1000)

.text           0x000000002ffc0000     0xc538
 *(SORT_BY_ALIGNMENT(.text._start))
 .text._start   0x000000002ffc0000       0x98 out/stm32mp157c-ev1/core/arch/arm/kernel
/entry_a32.o
                0x000000002ffc0000                _start  -> OP-TEE Load address
...
```

In this example, the OP-TEE load address is 0x2ffc0000.

All OP-TEE core symbols can be loaded:

```
(gdb) add-symbol-file <path_to_build_folder>/tee.elf 0x2ffc0000
```

Thanks to the Wrapper_for_FSBL_images, you will be able to debug the initial boot sequence. Once the board starts and waits into the FSBL debug wrapper, a hardware breakpoint can be set at the OP-TEE core entry point.

```
(gdb) hb _start
```

Warning

OP-TEE symbols may override the FSBL one, so you cannot load both TF-A and OP-TEE symbols at the same time.

2.5.2 Debugging during runtime execution

Once U-Boot or the Linux kernel is running, secure memory or regions cannot be accessed, but it is possible to break by setting a hardware breakpoint on OP-TEE service handler. GDB breaks once it has switched into the secure world and reached the break instruction. Once halted, GDB can access secure resources as peripheral interfaces or memories. For example, to break into U-Boot use the following GDB instructions:

```
(gdb) hb stm32_sip_service
(gdb) continue
```

On the first service call occurrence GDB breaks into the **stm32_sip_service()** entry in the OP-TEE core.

3 Debugging OP-TEE trusted applications

3.1 Embedded assertions

OP-TEE trusted applications can embed assertions as well as the OP-TEE core, as described above.
Assertions in trusted applications are embedded on the configuration directive **CFG_TEE_CORE_DEBUG={y|n}** when the OP-TEE OS package is built.

3.2 Debug with traces

OP-TEE trusted applications can embed trace messages using the same macros as the OP-TEE core (**EMSG()** and similar functions described above).
The build directive that sets the trace level for a trusted application is **CFG_TEE_TA_LOG_LEVEL={0|1|2|3|4}**.
The level values match those described for **CFG_TEE_CORE_LOG_LEVEL** in the section above.

3.3 Stack unwind support

When an OP-TEE trusted application panics, it may output backtrace messages on the OP-TEE console through the OP-TEE core.
These backtrace messages are generated by the OP-TEE OS when built with **CFG_UNWIND=y**. OP-TEE OS package provides a tool to analyse backtrace messages.

More information is available from the OP-TEE abort-dumps documentation[1].

4 References

- [1.01.1] https://optee.readthedocs.io/en/latest/debug/abort_dumps.html

Stable: 13.05.2020 - 08:56 / Revision: 13.05.2020 - 08:54

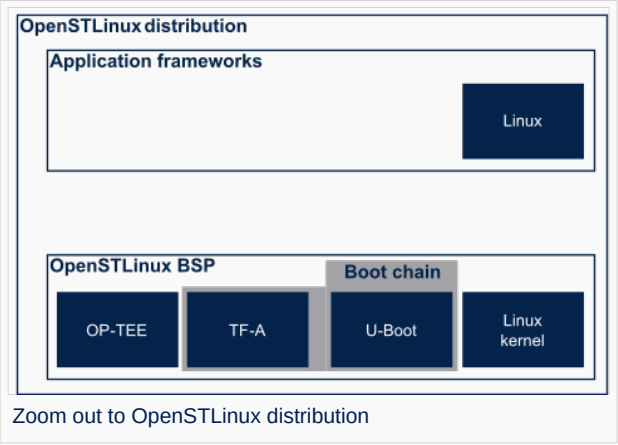
A quality version of this page, approved on *13 May 2020*, was based off this revision.

Contents

1 Overview of the OP-TEE open source project 24
2 Architecture 25
2.1 OP-TEE core 25
2.2 OP-TEE trusted libraries 25
2.3 TEE Linux driver 26
2.4 TEE Client API 26
2.5 TEE supplicant 26
2.6 Host tools 26
3 Booting with OP-TEE 27
4 Invoking the OP-TEE services from Linux based OS 28
5 Experiencing OP-TEE on a target 29
6 References 30

1 Overview of the OP-TEE open source project

OP-TEE allows the development and integration of secure services and applications under trusted execution environments, that is execution environments isolated from the Linux® -based OS.



Zoom out to OpenSTLinux distribution

Description extracted from the OP-TEE site[1]:

> *"OP-TEE is an open source project, which contains a full implementation to make up a complete Trusted Execution Environment using the ARM® TrustZone®. technology. OP-TEE meets the GlobalPlatform TEE System Architecture specification. It also provides the TEE Internal core API v1.1 as defined by the GlobalPlatform TEE Standard for the development of Trusted Applications. OP-TEE Trusted OS is accessible from the Linux based OS using the GlobalPlatform TEE Client API Specification v1.0, which also is used to trigger secure execution of applications within the TEE."*

OP-TEE is delivered under a BSD style license and can run secure (trusted) applications without restriction on their licensing model.

The OP-TEE project is maintained by the Linaro Security Working Group.

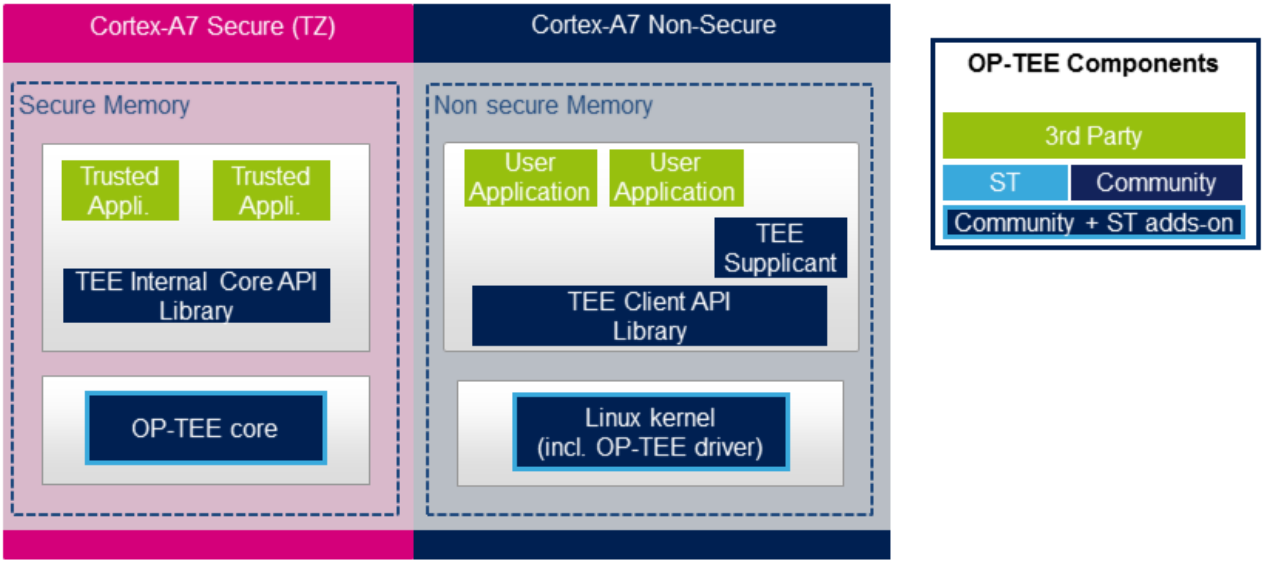
- OP-TEE official site[1]
- OP-TEE source repositories [2][3][4]
- OP-TEE documentation[5]

GlobalPlatform Device TEE specifications (TEE Client API, TEE Internal Core API and few more) is available from the GlobalPlatform site[6].

2 Architecture

The OP-TEE project includes several secure and non-secure embedded components, as well as some tools for development and debugging purposes.

The figure below shows the main OP-TEE embedded components, namely the OP-TEE core and trusted application standard libraries on the secure side, and the Client API library, the OP-TEE supplicant daemon and the OP-TEE Linux kernel driver on the non-secure side.



2.1 OP-TEE core

The main OP-TEE component is the OP-TEE core. The OP-TEE core execution is done in Arm® Cortex®-A secure state while the non-secure world (likely a Linux based OS) is done in the non-secure state of the processor. The OP-TEE core executes in secure privileged (kernel) mode, while trusted applications are executed in secure user mode.

OP-TEE can load signed trusted applications stored in the Linux OS file system or embedded in the OP-TEE core boot image.

On devices with secure external memory, the OP-TEE core runs as a monolithic image in the secure memory. On devices with a small secure memory, the OP-TEE core can run in paging-on-demand configuration: a small resident agent is loaded in the small secure memory and can securely page-in/page-out data from/to the non-secure (or less secure) external memory.

OP-TEE core source files can be found from optee_os repository [2].

2.2 OP-TEE trusted libraries

OP-TEE embeds utility libraries for trusted application development including the GlobaPlatform Device TEE Internal Core API Library, which provides the standard services a trusted application can expect from the TEE. OP-TEE supports the loading of static and dynamic libraries in the TEE.

The OP-TEE standard trusted application libraries source files can be found in the optee_os repository[2].

2.3 TEE Linux driver

The OP-TEE Linux driver is part of the Linux kernel since release 4.12.

The OP-TEE Linux driver is enabled via the CONFIG_OPTEE configuration directive through the usual Linux kernel configuration means. The driver can be probed thanks to a device tree node.

2.4 TEE Client API

The OP-TEE project embeds an implementation of the GlobaPlatform Device TEE Client API specification for Linux based OS. This TEE Client API specification is partly implemented as a userland library and partly as a Linux kernel OP-TEE driver. The API allows userland clients to invoke trusted applications and the OP-TEE core services exported to non-secure world with a standard API.

The OP-TEE Client API library source files can be found in the optee_client repository[3].

2.5 TEE supplicant

The OP-TEE core can rely on non-secure remote services. OP-TEE embeds an implementation of a non-secure userland supplicant, that can be invoked by the OP-TEE core through the OP-TEE Linux kernel driver. An example of such service is the access to a non-volatile media device that is controlled in the non-secure world.

The OP-TEE supplicant source files can be found in the optee_client repository[3].

2.6 Host tools

The OP-TEE optee_os component, once built, generates a so-called Trusted Application Development Kit to ease the development and integration of trusted applications on a target system. The Trusted Application Development Kit includes the libraries, with their header files and makefile scripts, that allow the generation of signed trusted applications from their respective source files.

Optee_os package also provides a tool to analyse call stack backtraces in case of trusted application and/or OP-TEE core crash. Refer to script **symbolize.py** in optee_os source tree[2].

3 Booting with OP-TEE

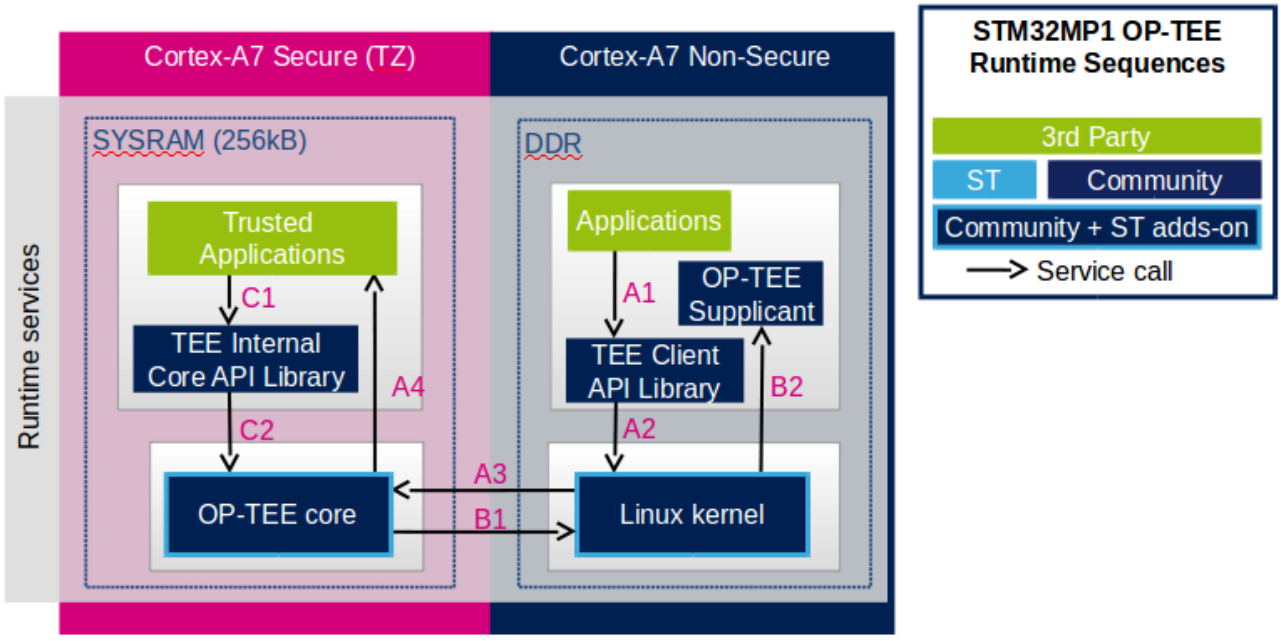
The OP-TEE core is a secure firmware. It must be booted prior to the non-secure world on Arm Cortex-A core(s). The secure bootloader must therefore load the OP-TEE core images in memory and run its initialization prior to executing the first booted non-secure image.

Refer to the target system boot sequences for more details.

4 Invoking the OP-TEE services from Linux based OS

Once the Linux kernel is booted, the OP-TEE core is already initialized and ready to serve.

The figure below shows the main run time sequences in which the OP-TEE can be involved.



Sequence A: an non-secure application invokes a service from a trusted application.

The non-secure application calls the TEE Client API library (**A1**), which in turns invokes (**A2**) the Linux kernel OP-TEE driver. The OP-TEE driver invokes the secure world (**A3**) and reaches the OP-TEE core. The last OP-TEE core transfers the request (**A4**) to the target trusted application. Once the trusted application has completed the request, the system branches back to the calling application with the request status.

If an invoked trusted application is not yet loaded into the TEE, the OP-TEE core loads it by calling remote services through the non-secure TEE supplicant as described in **sequence B** below.

In addition, any invocation of the TEE from the non-secure world must go through the Linux kernel OP-TEE driver.

Sequence B: the OP-TEE core must invoke a non-secure remote service.

The OP-TEE core invokes (**B1**) the Linux kernel OP-TEE driver which in turns notifies the TEE supplicant daemon (**B2**) for a request. Once the supplicant has completed the request, the system branches back to the OP-TEE core with the request status.

Sequence C: a trusted application invokes an OP-TEE core service.

Most of the services defined by the GlobaPlatform Device TEE Internal Core API must be executed in OP-TEE core privileged mode. The trusted application calls the corresponding service from the TEE Internal Core API library (**C1**), which issues a system call (**C2**) to the OP-TEE core. Once the core has completed the request, the system branches back to the calling trusted application with the request status.

5 Experiencing OP-TEE on a target

First make sure your setup includes OP-TEE in the boot sequence. If the OP-TEE core console traces are enabled, you should see the OP-TEE banner after secure bootloader traces and before non-secure bootloader traces. The OP-TEE core banner looks like this:

```
I/TC:  OP-TEE version: <some-reference-version-info> #1 Mon Jun 25 08:59:21 UTC 2018 arm
I/TC:  Initialized
```

The Linux kernel boot traces also show the successful probing of the OP-TEE Linux kernel driver:

```
optee: probing for conduit method from DT.
optee: initialized driver
```

The OP-TEE non-secure components are stored in the file system:

- By default the TEE supplicant is installed at **/usr/bin/tee-supplicant**.
- By default, the TEE Client API library is installed at **/usr/lib/teec.so**.
- By default the TEE regression test tool is installed at **/usr/bin/xtest**.

In the default OP-TEE configuration, trusted applications are stored in the non-secure filesystem at **/lib/optee_armtz/*.ta**.

OP-TEE provides means to protect the trusted application binary images from corruption as image signature or installation in the OP-TEE secure storage. In any case, it is likely that Othe P-TEE core needs to invoke a non-secure service to retrieve the trusted application(s) from some non-secure filesystem data in order to load trusted application(s) in the TEE. This service requires the availability of the OP-TEE supplicant.

Therefore, once the non-secure OS has booted, it must launch the OP-TEE supplicant as a background daemon. Use the following shell command to start the OP-TEE supplicant from a booted Linux system, :

```
sh> tee-supplicant &
```

The OP-TEE package comes with some examples and regression tests. Use the following embedded shell command to run the regression tests:

```
sh> xtest
```

or to run only selective tests:

```
sh> xtest 1002     # Invokes some OP-TEE internal core services
sh> xtest 1004     # Invokes a trusted application loaded from the non-secure filesystem
```

6 References

- [1.01.1] https://op-tee.org
- [2.02.12.22.3] https://github.com/OP-TEE/optee_os
- [3.03.13.2] https://github.com/OP-TEE/optee_client
- https://github.com/OP-TEE/optee_test
- https://optee.readthedocs.io/
- https://globalplatform.org/