BKK19-419: Debugging with OP-TEE

Sumit Garg
Agenda

● Brief OP-TEE background
● Basics for debugging OP-TEE
  ○ GlobalPlatform return code origins
  ○ Print log levels
  ○ Abort dumps / call stack
  ○ GDB using QEMU
● Profiling in OP-TEE
  ○ Benchmark framework
  ○ Profiling using gprof
● ftrace for Linux TEE driver
● Function tracing in OP-TEE?
  ○ ftrace mechanism
Brief OP-TEE background

- **Normal World**
  - Client Applications
  - TEE Supplicant
  - TEE Client API
  - Linux OS
  - TEE Driver

- **Secure World**
  - Trusted Applications
  - TA Interface
  - TEE Internal API
  - OP-TEE OS
  - Secure Monitor

- SMC
Basics for debugging OP-TEE
## Basics: GlobalPlatform return code origins

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEEC_ORIGIN_API ¹</td>
<td>0x00000001</td>
<td>The return code is an error that originated within the TEE Client API implementation.</td>
</tr>
<tr>
<td>TEEC_ORIGIN_COMMS ¹</td>
<td>0x00000002</td>
<td>The return code is an error that originated within the underlying communications stack linking the rich OS with the TEE.</td>
</tr>
<tr>
<td>TEEC_ORIGIN_TEE ¹</td>
<td>0x00000003</td>
<td>The return code is an error that originated within the common TEE code.</td>
</tr>
<tr>
<td>TEEC_ORIGIN_TRUSTED_APP</td>
<td>0x00000004</td>
<td>The return code originated within the Trusted Application code. This includes the case where the return code is a success.</td>
</tr>
</tbody>
</table>

Source: [https://globalplatform.org/specs-library/?filter-committee=tee](https://globalplatform.org/specs-library/?filter-committee=tee)

¹ Indicates a return code that originates from the TEE.
Basics: Print log levels

OP-TEE uses a secure UART to dump all the debug prints. So OP-TEE provides following log levels which can be enabled during compilation:

- CFG_TEE_CORE_LOG_LEVEL
- CFG_TEE_TA_LOG_LEVEL

Levels:
- 0: none
- 1: error
- 2: error + info
- 3: error + info + debug
- 4: error + info + debug + flow

Default value for log levels is “1: error”.
Abort dumps / call stack

In case of any abort, information is dumped onto secure UART.

Abort dump types:
- Data or prefetch abort exception
- Call to TEE_Panic()
- TEE core fatal error
Symbolize call stack addresses

OP-TEE provides a helper script called symbolize.py (<optee_os>/scripts/symbolize.py) to symbolize call stack addresses.

Example usage:

```bash
$ cat dump.txt | ./optee_os/scripts/symbolize.py -d ./optee_examples/*/ta
<snip>
```

```plaintext
E/TC:0 Call stack:
E/TC:0 0x001044a8 utee_panic at optee_os/lib/libutee/arch/arm/utee_syscalls_a32.S:74
E/TC:0 0x0010ba59 TEE_Panic at optee_os/lib/libutee/tee_api_panic.c:35
E/TC:0 0x00101093 hmac_sha1 at optee_examples/hotp/ta/hotp_ta.c:63
E/TC:0 0x001013ed get_hotp at optee_examples/hotp/ta/hotp_ta.c:171
E/TC:0 0x00101545 TA_InvokeCommandEntryPoint at optee_examples/hotp/ta/hotp_ta.c:225
E/TC:0 0x0010441b entry_invoke_command at optee_os/lib/libutee/arch/arm/user_ta_entry.c:207
E/TC:0 0x00104477 __untee_entry at optee_os/lib/libutee/arch/arm/user_ta_entry.c:235
<snip>
```

Usage details: https://optee.readthedocs.io/debug/abort_dumps.html
GDB using QEMU

GDB can use the debug stubs built into QEMU to perform full symbolic debug of OP-TEE. Steps to debug OP-TEE on QEMU using GDB:

# Launch QEMU for OP-TEE (using build repo)
$ cd <qemu-project>/build
$ make run

# Launch GDB in another console
$ cd <qemu-project>/toolchains/aarch64/bin
$ ./aarch64-linux-gnu-gdb -q

# Connect to QEMU GDB server, load symbols from tee.elf, ta.elf and set breakpoints to debug
(gdb) target remote localhost:1234
(gdb) symbol-file <qemu-project>/optee_os/out/arm/core/tee.elf
(gdb) add-symbol-file <path-to-ta-elf>/<uuid>.elf <load-address>
(gdb) b tee_entry_std
(gdb) b TA_InvokeCommandEntryPoint
(gdb) c
Profiling in OP-TEE
Profiling: Benchmark framework

OP-TEE provides a framework to measure latency values for various architectural layers involved during runtime operation as below:

- Round-trip time from a Client Application to a Trusted Application.
- Latency to go through each layer:
  - libTEEC -> Linux TEE driver
  - Linux TEE driver -> OP-TEE OS core
  - OP-TEE OS core -> TA entry point (not supported yet)
  - The same way back

Usage details: https://optee.readthedocs.io/building/gits/optee_benchmark.html#optee-benchmark
Profiling: Benchmark framework design
Profiling using gprof

OP-TEE supports profiling of user Trusted Applications with gprof. Details of profiling data collected are as follows:

- **Call graph information**
  - Records a pair of program counters (caller and callee). Also, records count for particular arc of the call graph being invoked.

- **PC distribution over time**
  - TEE core samples TA program counter (PC) during interruption of TA execution like normal world interrupts etc.

Usage details: [https://optee.readthedocs.io/debug/gprof.html](https://optee.readthedocs.io/debug/gprof.html)
Function Tracing (ftrace)
ftrace - Function tracing

GCC compilation with -pg option inserts a special function `_mcount()` for every function entry.

In case of ftrace, this API could be used for various purposes like:

- To record caller and callee functions.
- To record entry and exit point for every function.
- To record function execution time.

Code compiled with -pg

```
<release_thread>:
stp   x29, x30, [sp,#-16]!
mov   x29, sp
mov   x0, x30
bl    ffff000008092ea0 <_mcount>
ldp   x29, x30, [sp],#16
ret
```
ftrace for Linux TEE driver

Linux provides ability for dynamic function tracing (CONFIG_DYNAMIC_FTRACE=y)

Linux can dynamically enable/disable tracing

```
<release_thread>:
stp    x29, x30, [sp,#-16]!
mov    x29, sp
mov    x0, x30
nop
ldp    x29, x30, [sp],#16
ret
```

```
<release_thread>:
stp    x29, x30, [sp,#-16]!
mov    x29, sp
mov    x0, x30
bl     ftrace_caller
ldp    x29, x30, [sp],#16
ret
```

ftrace for Linux TEE driver: usage

Example usage of function graph for TEE driver apis:

```
# Mount debugfs to access ftrace
$ mount -t debugfs nodev /sys/kernel/debug
$ cd /sys/kernel/debug/tracing/

# Set filter for TEE driver apis
$ echo "optee_do_call_with_arg" > set_ftrace_filter
$ echo "optee_open_session" >> set_ftrace_filter
$ echo "optee_close_session" >> set_ftrace_filter
$ echo "optee_invoke_func" >> set_ftrace_filter

# Enable the function_graph tracer
$ echo "function_graph" > current_tracer
```

Output after running xtest

<p>|</p>
<table>
<thead>
<tr>
<th>CPU</th>
<th>DURATION</th>
<th>FUNCTION CALLS</th>
</tr>
</thead>
</table>
| 22) | 65.910 us | optee_open_session() {
| 22) | 75.340 us | optee_do_call_with_arg();
| 22) | 1922.170 us | optee_invoke_func() {
| 22) | 1935.320 us | optee_do_call_with_arg();
| 22) | 71.560 us | optee_close_session() {
| 22) | 81.230 us | optee_do_call_with_arg();
| ... |          |                 |
Function tracing in OP-TEE?

Ftrace in OP-TEE seems to be a promising feature providing enhanced debugging capabilities. Especially function graph trace which could be used:

- To get in-depth view of runtime TA execution flow.
- To get useful information during abort dumps.

```c
// Example hello world TA function graph
Call graph:
<snip>
TA_CreateEntryPoint();
TEE_Malloc() {
    malloc() {
        bget();
    }
}
ta_header_get_session();
__utee_to_param();
memcpy();
TA_OpenSessionEntryPoint();
__utee_from_param();
memset();
__utee_entry() {
    ta_header_get_session();
    __utee_to_param();
    memcpy();
    TA_InvokeCommandEntryPoint() {
        TEE_Panic() {
```

OP-TEE: ftrace mechanism

<release_thread>:

```
stp    x29, x30, [sp,#-16]!
mov    x29, sp
mov    x0, x30
bl     __utee_mcount
ldp    x29, x30, [sp],#16
ret
```

```
ftrace_enter()  → Dump entry point in ftrace buffer

Current fp/sp in _mcount()

X29
X30
...  
Instrumented function fp
Instrumented function pc

Instrumented parent function fp
Instrumented parent function pc

X29
X30

X30’ = ftrace_return

ftrace_return()  → Dump exit point in ftrace buffer

Note: here original return addresses are stored in ftrace_ret_stack[] used to return from ftrace_return()```
OP-TEE: ftrace implementation

- **Secure World**
  - Trusted Applications
  - OP-TEE OS
    - TA Interface
    - TEE Internal API
    - TA init: register ftrace buffer
    - TA abort or close session
    - Secure Monitor
    - RPC call
  - ftrace buffer

- **Normal World**
  - User
    - Client Applications
    - TEE Supplicant
    - TEE Client API
  - Kernel
    - Linux OS
    - TEE Driver
    - RPC call to dump ftrace buffer to normal world
OP-TEE ftrace: next steps...

- Create patch-set for upstream review from this ftrace PoC work.
- Extend TA ftrace buffer dump to include syscall function graph from OP-TEE OS.
- Implement ftrace for OP-TEE OS core.
Thank you

Join Linaro to accelerate deployment of your Arm-based solutions through collaboration

contactus@linaro.org