Fuzzing Trusted Applications via Shared Memory

Roland Nagy

CrySys Lab

whoami

Roland Nagy

PhD student @ CrySys Lab

HIT, BME

roland.nagy@crysys.hu

Agenda

- Trusted Apps
- Fuzzing basics
- TA Emulation
- Our custom fuzzer

Agenda

- Trusted Apps
- Fuzzing basics
- TA Emulation
- Our custom fuzzer

Agenda

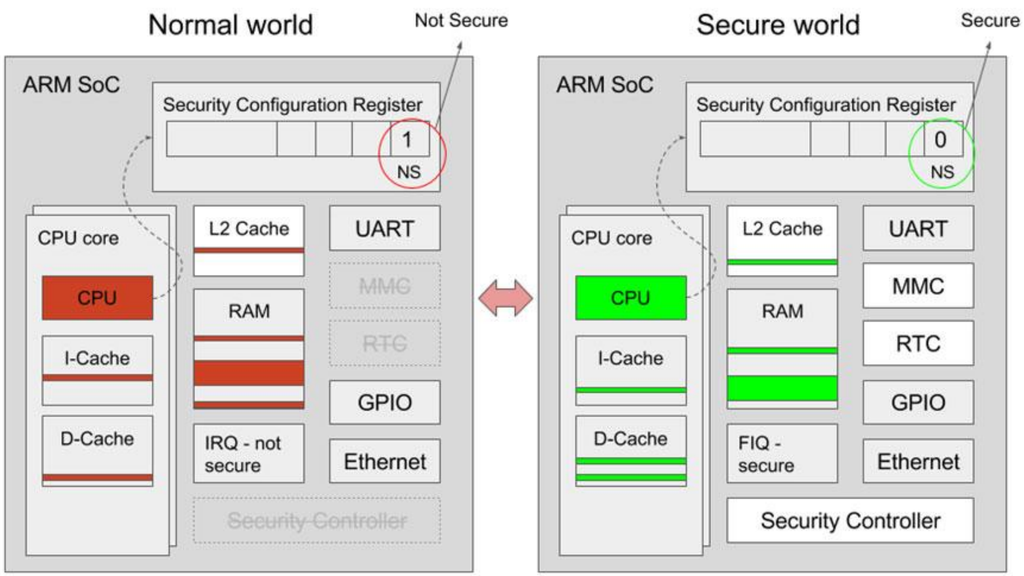
- **Trusted Apps**
- Fuzzing basics
- TA Emulation
- Our custom fuzzer

Trusted Apps

- Special software executed by a Trusted Execution Environment (TEE)
- Digitally signed, signature is verified before execution
- Isolated from “normal” application
  - Typically via some hardware feature
    - ARM TrustZone in our case
- Ideal for security-critical application
  - Cryptographic features
  - Financial applications
  - DRM

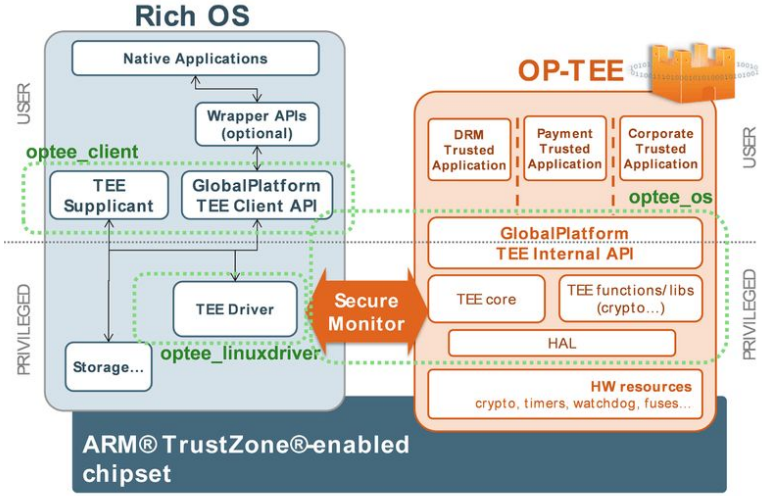
ARM TrustZone

- Non-Secure (NS) bit in the SCR
- Restrictions enforced by the system bus
- Normal & a Secure World
  - Linux/Android in the Normal World (REE)
  - OP-TEE OS in the Secure World (TEE)



OP-TEE

- Open-source Portable TEE
  - Ported to ~50 IoT platforms
- OS for the Secure World + additional components
  - Regular application can request the execution of Trusted Apps
  - TAs are integrity checked & live in protected memory



Our Trusted App

- Rootkit detection checks on the Linux kernel
  - Several integrity & consistency checks
- ~4500 lines of C code (excluding autogenerated headers)
- Uses 4 services implemented in the trusted kernel
  - Testing these components is out of scope for now
- Checks can be split into 5 sections + init
  - We intend to fuzz these separately

Our Trusted App

- Checks can be split into 5 sections + init
  - Init
  - Virtual File System related checks
  - Task-related data structures
  - General kernel integrity checks
  - REE File System checks
  - Network stack checks

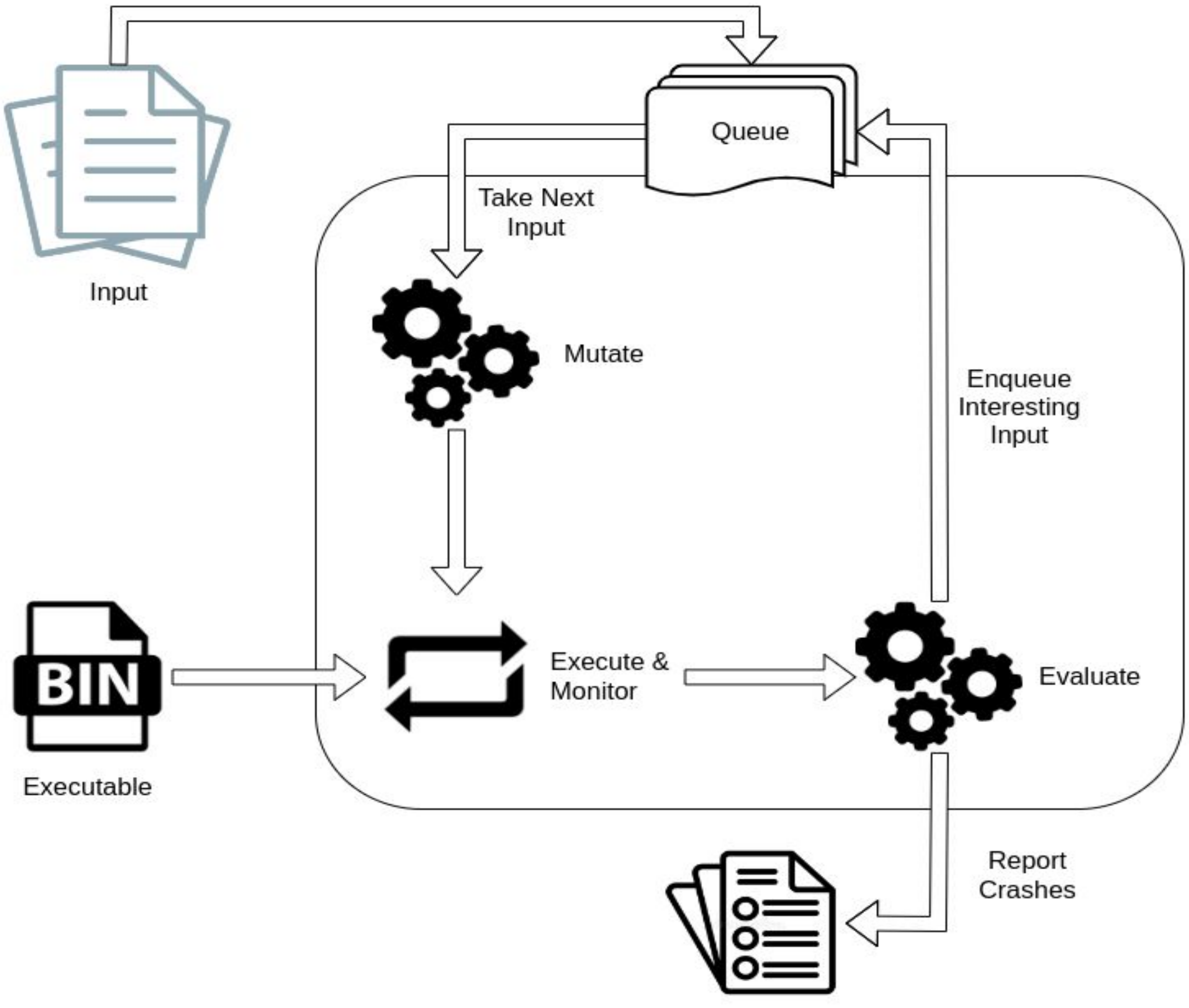
Agenda

- Trusted Apps
- **Fuzzing basics**
- TA Emulation
- Our custom fuzzer

Fuzzing

- Automated security testing
  - Random/semi-random test cases
- Highly effective
  - Superbly parallelizable
  - Instrumentation of the fuzzed app can improve efficiency
- Has some limitations
  - Reaching “deep” code might be hard
  - Monitoring the application can be complicated

Fuzzing - example



Naive approach for fuzzing

- Originally developed in Qemu
  - Let’s directly modify the memory content!
    - The kernel wouldn’t like this…
    - Too many irrelevant crashes
- Fake everything in unused kernel memory!
  - Too many changes needed in the TA

Less naive approach…

- Let's get rid of the kernel!
  - OPTEE wouldn't like this…
- TAs cannot be executed on regular systems
  - Signature + ELF file, but not executable (shared lib)
  - ARM64 architecture
  - "Custom" syscalls

Less naive approach…

- Let's get rid of the kernel!
  - OPTEE wouldn't like this…
- TAs cannot be executed on regular systems
  - Signature + ELF file, but not executable (shared lib)
  - ARM64 architecture
  - "Custom" syscalls
- Let's emulate!
  - Hacktivity 2021, Marcel Seibert: Investigating the Exploitability of Trusted Execution Environments

Agenda

- Trusted Apps
- Fuzzing basics
- **TA Emulation**
- Our custom fuzzer

Emulation

- Emulation, pros & cons

  - Can execute only parts of the TA
  - Can provide coverage information
  - Easy to identify crashes

  - System calls must be mocked
  - Services must be mocked
  - Some library functions need special attention

Emulation - Syscalls

- Trivial via Qiling, just implement the syscall in python

utee_return	Exit
utee_log	Logging
utee_panic	Panic
utee_open_ta_session	Session management
utee_close_ta_session	
utee_invoke_ta_command	
utee_check_access_rights	
utee_get_time	Time
utee_cryp_obj_get_info	Secure Storage
utee_cryp_obj_close	
utee_storage_obj_open	
utee_storage_obj_read	

Emulation - Other services

- If we fuzz the app in parts, only memory & file reading must be implemented
- These are done via syscalls as well
  - utee_open_ta_session & utee_invoke_ta_command
- Commands we need to implement
  - hash_mem - SHA256 hash from memory range
  - hash_file - SHA256 hash of file
  - hash_dir - SHA256 hash of the content of a directory
    - hash checks must be patched out of the app
  - read_mem - read physical memory, we will use this to fuzz

Emulation - Library functions

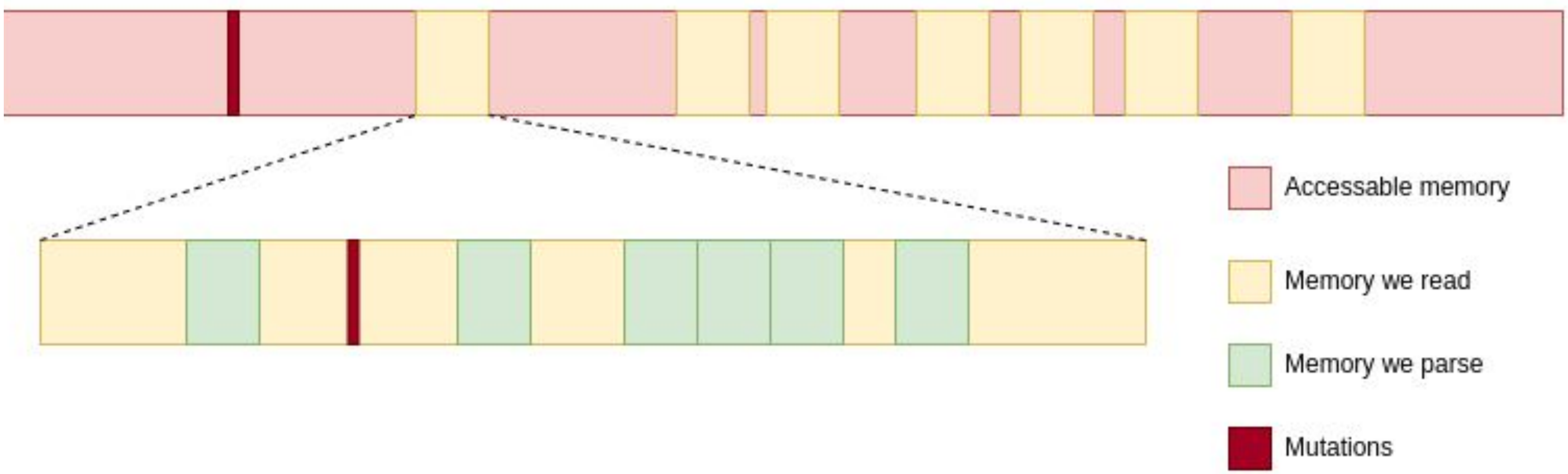
- Library functions are compiled into the TAs
  - Some initialization needed
- TAs have their own heap implementation
  - Heap in the BSS segment, like a global variable
  - It must be initialize, otherwise allocations return OOM
    - » malloc_add_pool(heap_start, heap_size)
  - Via Qiling, snapshot after initialization

Agenda

- Trusted Apps
- Fuzzing basics
- TA Emulation
- **Our custom fuzzer**

Fuzzing - challenges

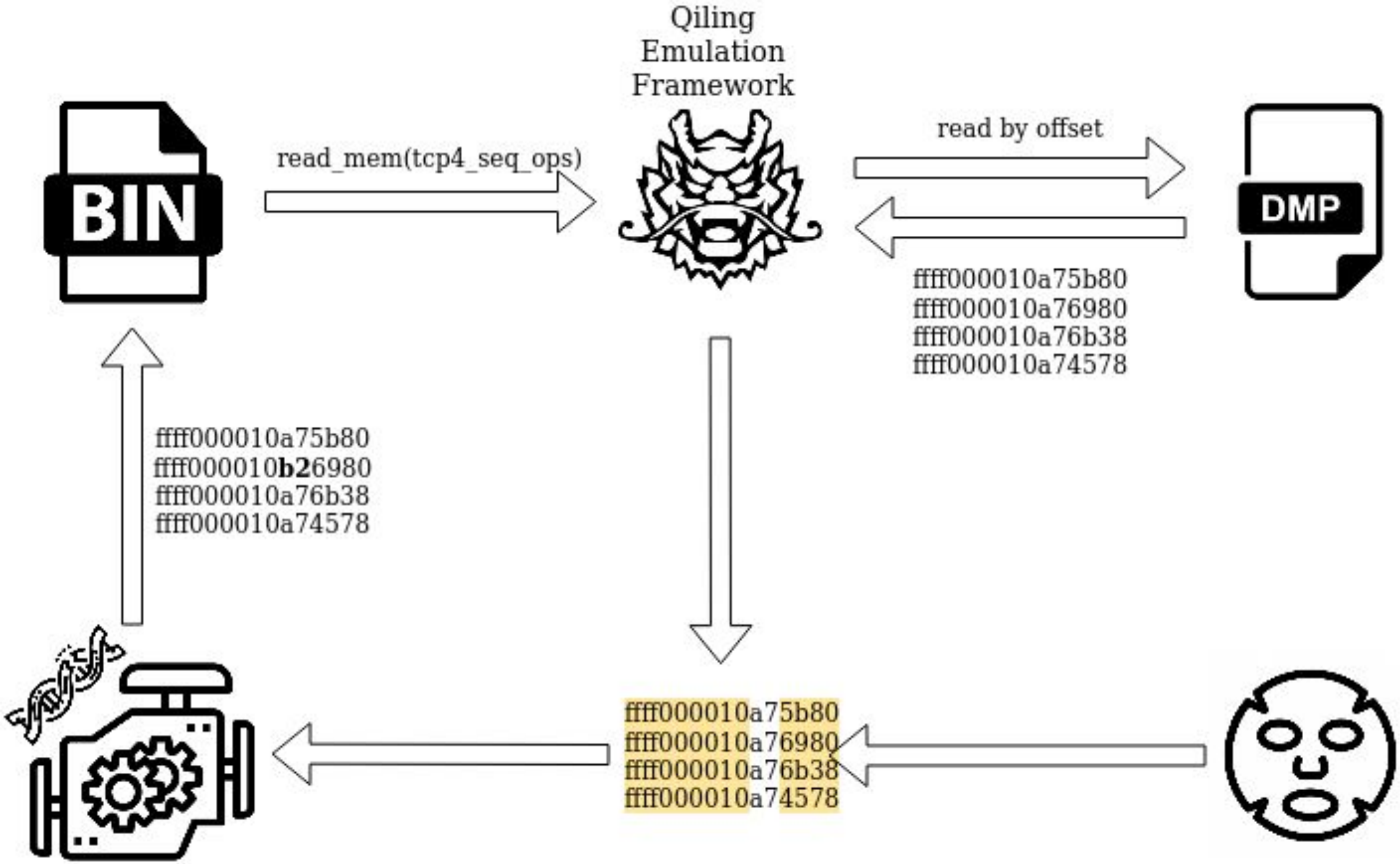
- Fuzzers need valid example inputs
- A memory image can be dumped from Qemu
  - Too big & too “sparse”, most of it is irrelevant
    - » Most of the image is not even read
    - » Even the read parts are not fully processed



Fuzzing - implementation

- Mutating a whole memory dump is ineffective
  - We take a memory image
  - Implement mutations in the read_mem command
    - » No mutations in irrelevant parts of the image
    - » “Masking” to avoid mutating uninteresting parts of read data
  - Mutations must be stored & saved on crashes

Fuzzing - architecture



Fuzzing - advanced topics

- We implement coverage guided fuzzing
  - The emulator provides coverage information

$interesting \leftarrow initial\_test\_seeds$
$seen \leftarrow \emptyset$
while true do
    Choose an input i from interesting
    $input \leftarrow MUTATE(i)$
    $path \leftarrow EXECUTE(input)$
    if $\{path\} \not\subset seen$ then
        $interesting \leftarrow interesting \cup \{input\}$
        $seen \leftarrow seen \cup \{path\}$

  - Most fuzzers can do this, we need to re-implement it

Status report

- No bugs found yet
- 3 sections of the TA run fine, 2 more & init are WIP
- A framework to fuzz OP-TEE TAs is under construction
  - An incomplete list of syscalls implemented (fully or partially)
  - Heap initialization implemented
- Coverage guided fuzzing is (almost completely) implemented
- A solution to fuzz via sparse shared memory is implemented

Acknowledgement

This work was supported by the SETIT project (no. 2018-1.2.1-NKP-2018-00004), which has been implemented with the support provided by the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme.

NEMZETI KUTATÁSI, FEJLESZTÉSI ÉS INNOVÁCIÓS HIVATAL

AZ INNOVÁCIÓ LENDÜLETE

AZ NKFI ALAPBÓL MEGVALÓSULÓ PROJEKT