Fuzzing OP-TEE with AFL

Martijn Bogaard
Senior Security Analyst

martijn@riscure.com
@jmartijnb
About me

• Senior Security Analyst @ Riscure (2015 – now)
  • Code reviews & pentests of security critical sw components of embedded systems, hardware evals (chip design reviews, fault injection testing), sw security trainer
  • Evaluating security of (proprietary) TEEs since late 2015

• Actively researching fuzzing & simulation in last 6 years
  • Distributed fuzzing (Disfuzz AFL, 2014)
  • Fuzzing Windows drivers under Linux using kernel emulation (WKE, 2015)
  • OP-TEE simulator for Linux (2017)
  • Fault Injection attack simulation (FiSim, 2018)
  • Custom simulators for fuzzing & dynamic analysis of bootloaders, RTOS, Android services, ...
Today’s agenda

- Trusted Execution Environments?!?
- Why is fuzzing TEEs so difficult? (Is it...?)
- Fuzzing OP-TEE syscalls
Research motivation

- Trusted Execution Environments become widespread
  - Mandatory in Android 6+ (conditionally)
  - Starting to pick up in other markets e.g. automotive

- Highly privileged component in modern chips
  - Way more than Android/Linux
  - Strictly isolated from the rest

- Often controls access to device’s most secret crypto keys
  - KeyMaster, GateKeeper, DRM, Mobile Banking, ...
Concept

Normal world (REE) — Secure world (TEE)

App — OS (Linux, ...) — Secure OS — App — Keys
TEE Technology

- Arm TrustZone
- Intel SGX
- MIPS Virtualization
- RISC-V Keystone
TEE Technology

- Arm TrustZone → Linaro’s OP-TEE
- Intel SGX
- MIPS Virtualization
- RISC-V Keystone
Trusted Execution Environment

Normal world (REE)
- App
- App
- CA
- Shared memory
- Android / Linux

Secure world (TEE)
- Trusted Application (TA)
- Trusted Application (TA)
- Trusted Application (TA)
- OP-TEE

Monitor
Trusted Applications

- Often written by chip vendor or OEM in C
- Global Platform API for compatibility between TEEs
  - Specifies exported symbols + TA API

→ Not like any normal world application!
Example: AES in TA

```c
TEE_Result TA_InvokeCommandEntryPoint(void* sess_ctx, uint32_t cmd_id,
                                       uint32_t param_types, TEE_Param params[4])
{
    if (cmd_id == CMD_AES_ENCRYPT) {
        [...]
        TEE_AllocateOperation(&op_handle, TEE_ALG_AES_CBC_NOPAD, TEE_MODE_ENCRYPT, AES128_KEY_SIZE);
        TEE_AllocateTransientObject(TEE_TYPE_AES, AES128_KEY_SIZE, &key_handle);
        TEE_InitRefAttribute(&attr, TEE_ATTR_SECRET_VALUE, key, AES128_KEY_BYTE_SIZE);
        TEE_PopulateTransientObject(key_handle, &attr, 1);
        TEE_SetOperationKey(op_handle, key_handle);
        TEE_CipherInit(op_handle, iv, iv_sz);
        TEE_CipherUpdate(op_handle, buf_in, buf_in_len, buf_out, &buf_out_len);
    } else {
        return TEE_ERROR_BAD_PARAMETERS;
    }
}
```
Let’s go one layer deeper....
Example: AES in OP-TEE

uint32_t obj_handle1;
uint32_t obj_handle2;
uint32_t state_handle;

syscall_cryp_obj_alloc(0xa0000010, 0x80, &obj_handle1);
system_cryp_state_alloc(0x10000110, 0x0, obj_handle1, 0x0, &state_handle);
system_cryp_obj_alloc(0xa0000010, 0x80, &obj_handle2);
system_cryp_obj_populate(obj_handle2, {c0000000, buf_key, 0x10}, 0x1);
system_cryp_obj_reset(obj_handle1);
system_cryp_obj_copy(obj_handle1, obj_handle2);
system_cipher_init(state_handle, iv, 0x10);
system_cipher_update(state_handle, "Hello Nullcon!!!", 0x10, buf_out, &buf_out_len);
OP-TEE syscalls

Total: 70 syscalls
However, does trusted also mean secure?
Fuzziing

- Random data
  - Prototype by colleagues in 2014
  - `cat /dev/urandom > /dev/tee_smc`
- Model-based
- Coverage guided evolutionary fuzzing
Coverage guided evolutionary fuzzing

Simple algorithm:

1. Generate new input from collection of corpora
   - By applying 1 or more mutations (e.g. bit flips)
2. Run target with input
3. Collect code coverage information
4. If coverage information shows a previously unseen code path is taken, add to corpus queue
Coverage tracking

Fundamental question: Which branches are taken?
Coverage tracking
Why is fuzzing operating systems difficult?

- Crashes
- Global state
- Coverage tracking
- Seeding
- Trace stability $\Rightarrow$ threading, SMP, interrupts

A lot of progress for Linux and other mainstream Oss
  e.g. AFL, Syzkaller, ...

Let’s make use of that!
Goals

- Reuse an existing fuzzer (AFL)
  → Focus on the TEE challenge, not building a fuzzer
- Good, not perfect results (limited time)
Why is fuzzing TEEs difficult?

All before
  +
  • Isolated environment
  • Separate operating system
  • Limited API
  • Seeding
Corresponds with $addr somewhere in binary
AFL as Trusted Application?

fork(...)?
execve(...)?
open(...)?

→ TEE_ERROR_NOT_IMPLEMENTED
→ TEE_ERROR_NOT_IMPLEMENTED
→ TEE_ERROR_NOT_IMPLEMENTED
AFL as Trusted Linux Application

- AFL
- CA Proxy
  - libteec
- Proxy TA (SVC Invoke TA)
  - syscall_?????(...)

- Linux
- crypto
- storage
- SE
- Monitor
  - bitmap
How to (randomly) invoke system calls using AFL?

AFL can only mutate a blob of (random) data by flipping bits or bytes...
Hello!

```
00000000: 01 00 00 00 4a 00 00 00 18 00 00 01 10 00 00 00 ....J...........
00000010: ff 00 00 00 00 00 00 00 0a 48 65 6c 6c 6f 20 43 ........Hello C
00000020: 6f 6e 6e 65 63 74 21 00 connect!.
```

t[0] = malloc(16);
memcpy(t[0], "\x0aHello Connect!\x00", 16);
r[0] = utee_log(t[0], 0x10);
free(t[0]);

Syscall wrapper function
System calls as data

• Syscalls consists of id + up to 8 arguments
  • Values
  • Pointers to data, structures, etc
  • Pointers to structures with pointers, etc.
• Syscall arguments often depend on prev. syscall
  • E.g. returned handles

⇒ Argument encoding is the hardest part!
System calls as data

- Simple binary format encoding 1 or more syscall invokes
  - Contains arguments inline except buffer content
  - Goal: every bit flip results in a slightly different invoke

- After invoke data follows section with raw data
  - Strings, buffer content, etc.
  - Can be flexible referenced from argument info
System calls as data

00000000: 01 00 00 00 4a 00 00 00 18 00 00 01 10 00 00 00 ....J...........
00000010: ff 00 00 00 00 00 00 00 0a 48 65 6c 6c 6f 20 43 ........Hello C
00000020: 6f 6e 6e 65 63 74 21 00 onnect!.
System calls as data

00000000: 01 00 00 00 4a 00 00 00 18 00 00 01 10 00 00 00 ....J...........
00000010: ff 00 00 00 00 00 00 0a 48 65 6c 6c 6f 20 43 ........Hello C
00000020: 6f 6e 6e 65 63 74 21 00 onnect!.

Syscall id
System calls as data

Argument types

→ 1 nibble per argument
System calls as data

Argument types

0xa: argument 0 is a buffer with in-line data
System calls as data

Argument types

0xa: argument 0 is a buffer with in-line data
0x4: argument 1 is a 32-bit integer value
System calls as data

Argument 0: buffer
Argument 1: value
System calls as data

Argument 0:

Buffer offset (12-bit) → 0x18
Buffer length (20-bit) → 0x10
System calls as data

Argument 0:

Buffer offset (12-bit) → 0x18
Buffer length (20-bit) → 0x10
Data

00000000:  01 00 00 00 4a 00 00 00 18 00 00 01 10 00 00 00
00000010:  ff 00 00 00 00 00 00 00 0a 48 65 6c 6f 20 43
00000020:  6f 6e 6e 65 63 74 21 00

............
............Hello C
onnect!.
System calls as data

```c
00000000: 1b00 0000 4406 0000 1000 00a0 8000 0000
00000010: 0080 0000 0f00 0000 4447 0600 1001 0010
00000020: 0000 0000 0040 0000 0000 0000 0180 0000
00000030: 1e00 0000 c704 0000 0040 0000 0000 00c0
00000040: 8000 0001 0100 0000 1500 0000 a704 0000
00000050: 0140 0000 9000 0001 1000 0000 1600 0000
00000060: a764 0a00 0140 0000 a000 0001 1000 0000
00000070: 0300 0100 b000 8000 ff00 0000 0000 0000
00000080: 0001 0203 0405 0607 0809 0a0b 0c0d 0e0f
00000090: 0000 0000 0000 0000 0000 0000 0000 0000
000000a0: 4865 6c6c 6f20 436f 6e6e 6563 7421 2100
000000b0: 1000 0000 0000 0000

cryp_obj_alloc(0xa0000010, 0x80, &obj_handle);
cryp_state_alloc(0x10000110, 0, obj_handle, 0, &cryp_handle);
cryp_obj_populate (cryp_handle,
    {c0000000, "\x00\x01[...]\x0e\x0f"}, 1);
cipher_init(cryp_handle, "\x00\x00[...]\x00\x00", 0x10);
cipher_update(cryp_handle, "Hello Connect!!\x00", 0x10,
    buf_out, &buf_out_len);
```
System calls as data

```c
SYSCALL_INFO syscalls[] = {
    DEF_CALL(log, SCN_LOG, 2),
    DEF_CALL(panic, SCN_PANIC, 1),
    DEF_CALL(get_property, SCN_GET_PROPERTY, 7),
    DEF_CALL(get_time, SCN_GET_TIME, 2),
    DEF_CALL(set_ta_time, SCN_SET_TA_TIME, 1),
    DEF_CALL(cryp_state_alloc, SCN_CRYP_STATE_ALLOC, 5),
    [...]
};
```
How do we give AFL a good set of inputs to start from?

Creating them by hand is very tedious...
Seeding

• Difficult for the fuzzer to explore paths without good set of inputs (corpora)

• Ideally the start set covers the full interface
Can we use the test suite to seed AFL?
Test suite

• Contains thousands of (regression) tests

• Covers pretty much all syscalls!
Test case → corpus

- xtest libteec
- Linux
- Monitor
- Test TA 1
- Test TA 2
- Test TA 3
- crypto_state_alloc(...)
- Log
- AFL PTA
- OP-TEE
- crypto
Test case → corpus

1. Log raw function call
2. Extract argument semantics
3. Encode behavior, not concrete values
Test case → corpus

SYSCALL_INFO syscalls[] = {
    DEF_CALL(log, SCN_LOG, 2, { ARG_BUF_IN_ADDR | ARG_BUF_LEN_ARG(1), ARG_VALUE })
    DEF_CALL(panic, SCN_PANIC, 1, { ARG_VALUE })
    DEF_CALL(get_property, SCN_GET_PROPERTY, 7, { ARG_VALUE, ARG_VALUE, ARG_BUF_IN_ADDR | ARG_BUF_LEN_ARG(3), ARG_VALUE_INOUT_PTR, ARG_VALUE_INOUT_PTR, ARG_VALUE, ARG_VALUE_OUT_PTR })
    DEF_CALL(get_time, SCN_GET_TIME, 2, { ARG_VALUE, ARG_BUF_OUT_ADDR | ARG_BUF_SIZE(sizeof(TEE_Time)) })
    DEF_CALL(set_ta_time, SCN_SET_TA_TIME, 1, { ARG_BUF_IN_ADDR | ARG_BUF_SIZE(sizeof(TEE_Time)) })
    DEF_CALL(cryp_state_alloc, SCN_CRYP_STATE_ALLOC, 5, { ARG_VALUE, ARG_VALUE, ARG_HANDLE, ARG_HANDLE, ARG_HANDLE_OUT_PTR })
    ...
};

Use expected argument count to log raw calls

cryp_obj_alloc[27](a0000010, 80, 40000dfc)
    [*0x40000dfc => 1e4660]

cryp_state_alloc[15](10000110, 0, 1e4660, 0, 40020a88)
    [*0x40020a88 => 1e44e0]

cryp_obj_alloc[27](a0000010, 80, 40000e6c)
    [*0x40000e6c => 1e3fa0]

cryp_obj_populate[30](1e3fa0, *40000df0:18, 1)
    attr 0 { id: c0000000, a: 40023290, b: 10 }

cryp_obj_reset[29](1e4660)

cryp_obj_copy[31](1e4660, 1e3fa0)

cipher_init[21](1e44e0, *40024270:10, 10)

cipher_update[22](1e44e0, *400222b0:10, 10, *40000e38=10)
SYSCALL_INFO syscalls[] = {
    DEF_CALL(log, SCN_LOG, 2, { ARG_BUF_IN_ADDR | ARG_BUF_LEN_ARG(1), ARG_VALUE }),
    DEF_CALL(panic, SCN_PANIC, 1, { ARG_VALUE }),
    DEF_CALL(get_property, SCN_GET_PROPERTY, 7, { ARG_VALUE, ARG_VALUE, ARG_BUF_IN_ADDR | ARG_BUF_LEN_ARG(3), ARG_VALUE_INOUT_PTR, ARG_VALUE_INOUT_PTR, ARG_VALUE, ARG_VALUE_OUT_PTR }),
    DEF_CALL(get_time, SCN_GET_TIME, 2, { ARG_VALUE, ARG_BUF_OUT_ADDR | ARG_BUF_SIZE(sizeof(TEE_Time)) }),
    DEF_CALL(set_ta_time, SCN_SET_TA_TIME, 1, { ARG_BUF_IN_ADDR | ARG_BUF_SIZE(sizeof(TEE_Time)) }),
    DEF_CALL(cryp_state_alloc, SCN_CRYP_STATE_ALLOC, 5, { ARG_VALUE, ARG_VALUE, ARG_HANDLE, ARG_HANDLE, ARG_HANDLE_OUT_PTR }),
};

cryp_obj_alloc[27](a0000010, 80, 40000dfc)
[*0x40000dfc => 1e4660]
cryp_state_alloc[15](10000110, 0, 1e4660, 0, 40020a88)
[*0x40020a88 => 1e44e0]
cryp_obj_alloc[27](a0000010, 80, 40000e6c)
[*0x40000e6c => 1e3fa0]
cryp_obj_populate[30](1e3fa0, *40000df0:18, 1)
attr 0 { id: c0000000, a: 40023290, b: 10 }
cryp_obj_reset[29](1e4660)
cryp_obj_copy[31](1e4660, 1e3fa0)
cipher_init[21](1e44e0, *40024270:10, 10)
cipher_update[22](1e44e0, *400222b0:10, 10, *400222b0:10, 40000e38=10)
Test case → corpus

SYSCALL_INFO syscalls[] = {
  DEF_CALL(log, SCN_LOG, 2, { ARG_BUF_IN_ADDR | ARG_BUF_LEN_ARG(1), ARG_VALUE })
  DEF_CALL(panic, SCN_PANIC, 1, { ARG_VALUE })
  DEF_CALL(get_property, SCN_GET_PROPERTY, 7, { ARG_VALUE, ARG_BUF_IN_ADDR | ARG_BUF_LEN_ARG(3), ARG_VALUE_INOUT_PTR, ARG_VALUE_INOUT_PTR, ARG_VALUE, ARG_VALUE_OUT_PTR })
  DEF_CALL(get_time, SCN_GET_TIME, 2, { ARG_VALUE, ARG_BUF_OUT_ADDR | ARG_BUF_SIZE(sizeof(TEE_Time)) })
  DEF_CALL(set_ta_time, SCN_SET_TA_TIME, 1, { ARG_BUF_IN_ADDR | ARG_BUF_SIZE(sizeof(TEE_Time)) })
  DEF_CALL(cryp_state_alloc, SCN_CRYP_STATE_ALLOC, 5, { ARG_VALUE, ARG_VALUE, ARG_HANDLE, ARG_HANDLE, ARG_HANDLE_OUT_PTR })
}[
...]

cryp_obj_alloc[27](a0000010, 80, 40000dfc)
  [*0x40000dfc => 1e4660]
cryp_state_alloc[15](10000110, 0, 1e4660, 0, 40020a88)
  [*0x40020a88 => 1e44e0]
cryp_obj_alloc[27](a0000010, 80, 40000e6c)
  [*0x40000e6c => 1e3fa0]
cryp_obj_populate[30](1e3fa0, *40000df0:18, 1)
  attr 0 { id: c0000000, a: 40023290, b: 10 }
cryp_obj_reset[29](1e4660)
cryp_obj_copy[31](1e4660, 1e3fa0)
cipher_init[21](1e44e0, *40024270:10, 10)
cipher_update[22](1e44e0, *400222b0:10, 10, *400222b0:10, 40000e38=10)

Generate behavioral description of test case
Test case ➔ corpus

```c
b[0] = malloc(8);
cryp_obj_alloc(0xa0000010, 0x80, b[0]);
b[1] = malloc(8);
cryp_state_alloc(0x10000110, 0x0, *((uint32_t*)b[0]),
                 b[1], b[1]);
b[2] = malloc(8);
cryp_obj_alloc(0xa0000010, 0x80, b[2]);
cryp_obj_populate(*((uint32_t*)b[2]),
                  {c0000000, \0x00\01\02[...]\0d\0e\0f},
                  0x1);
cryp_obj_reset(*((uint32_t*)b[0]));
cryp_obj_copy(*((uint32_t*)b[0]), *((uint32_t*)b[2]));
t[1] = malloc(16);
memcpy(t[1], \0x00[...]\0x00, 16);
cipher_init(*((uint32_t*)b[1]), t[1], 0x10);
free(t[1]);
t[1] = malloc(16);
memcpy(t[1], "Hello Connect!!\00", 16);
b[3] = malloc(16);
t[4] = malloc(8);
memcpy(t[4], \0x10\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00", 8);
cipher_update(*((uint32_t*)b[1]), t[1], 0x10, b[3], t[4]);
```
Test case ➔ corpus

* regression_6001 Test TEE_CreatePersistentObject
  o regression_6001.1 Storage id: 00000001
  Write trace to /tmp/trac/fileWd213
    regression_6001.1 OK
  o regression_6001.2 Storage id: 00000000
  Write trace to /tmp/trac/file0VP1s4
    regression_6001.2 OK
    regression_6001 OK

* regression_6002 Test TEE_OpenPersistentObject
  o regression_6002.1 Storage id: 00000001
  Write trace to /tmp/trac/filej3Ssa1
    regression_6002.1 OK
  o regression_6002.2 Storage id: 00000000
  Write trace to /tmp/trac/fileS52NYT
    regression_6002.2 OK
    regression_6002 OK

* regression_6003 Test TEE_ReadObjectData
  o regression_6003.1 Storage id: 00000001
  Write trace to /tmp/trac/fileTOTSIE
    regression_6003.1 OK
  o regression_6003.2 Storage id: 00000000
  Write trace to /tmp/trac/fileABY1H
    regression_6003.2 OK
    regression_6003 OK

* regression_6004 Test TEE_WriteObjectData
  o regression_6004.1 Storage id: 00000001
  Write trace to /tmp/trac/fileS1LeV
    regression_6004.1 OK
  o regression_6004.2 Storage id: 00000000
Progress

• Working fuzzer, but...
  • Trace (in)stability
  • Code coverage
  • Performance
Trace (in)stability

• Same input should always result in same bitmap output
• However:
  • Threading
  • Interrupts
  • RPC calls
  • Global state

→ AFL thinks input results in new code path while it doesn’t!
Trace (in)stability

```c
struct tee_ta_session {
    TAILQ_ENTRY(tee_ta_session) link;
    TAILQ_ENTRY(tee_ta_session) link_tsd;
    struct tee_ta_ctx *ctx;
    TEE_Identity clnt_id;
    bool cancel;
    bool cancel_mask;
    TEE_Time cancel_time;
    void *user_ctx;
    uint32_t ref_count;
    struct condvar refc_cv;
    struct condvar lock_cv;
    int lock_thread;
    bool unlink;
#if defined(CFG_AFL_ENABLE)
    struct afl_ctx* afl_ctx;
    struct afl_svc_trace_ctx* svc_trace_ctx;
#endif
};
```

```c
typedef struct afl_ctx {
    bool enabled;
    char bitmap[MAP_SIZE];
    uint64_t prev_loc;
};
```
But which parts did we fuzz?

We know already which parts are covered by each input. Can we aggregate and visualize this information?
Coverage tracking

DEMO

Lighthouse plugin for IDA Pro - https://github.com/gaasedelen/lighthouse
Performance

• QEMU ~30 execs/sec
• HiKey ~250-300 execs/sec

• Ideally? 1000 - 10000 execs/sec!

• Could we do something radically different?
About me

• Senior Security Analyst @ Riscure (2015 – now)
  • Code reviews & pentests of security critical sw components of embedded systems, hardware evals (chip design reviews, fault injection testing), sw security trainer
  • Evaluating security of (proprietary) TEEs since late 2015

• Actively researching fuzzing & simulation in last 6 years
  • Distributed fuzzing (Disfuzz AFL, 2014)
  • Fuzzing Windows drivers under Linux using kernel emulation (WKE, 2015)
  • **OP-TEE emulator for Linux (2017)**
  • Fault Injection attack simulation (FiSim, 2018)
  • Custom simulators for fuzzing & dynamic analysis of bootloaders, RTOS, Android services, …
OP-TEE emulator for Linux

CA process \[\rightarrow\] IPC \[\rightarrow\] "Kernel" process

Reuses OP-TEE core code i.e. crypto SVCs

Fork + ptrace

"SVC"

TA Process(es)

Linux
OP-TEE emulator for Linux

Linux

CA process

IPC

“Kernel” process

Fork + ptrace

“SVC”

TA Process(es)
Idea

- Seed
- Corpus
- OP-TEE Simulator (fast)
- SVCs Queue
- Full system QEMU / Hardware (slow)
Idea

• Run OP-TEE as user-mode Linux process

• Find inputs that trigger new code paths

• Use found inputs to seed real fuzzer
Approach

• Port (almost) entire OP-TEE kernel as Linux process

• Accepts on stdin sequence of SVCs to execute

• Use AFL to find unique SVC sequences
  • Ignore crashes as they are mostly meaningless

• Use AFL queue to seed real fuzzer
Design

AFL

Queue

SVCs

Proxy TA
(SVC Invoke TA)

crypto
storage
SE

SVCs

~100 symbols
functions & globals

SVC parser

SVC dispatcher

Stubs

AFL instrumentation

bitmap

SVCs

OP-TEE sim

crypto
storage
SE

SVCs

~100 symbols
functions & globals

SVC parser

SVC dispatcher

Stubs

AFL instrumentation

Proxy TA
(SVC Invoke TA)

SVCs

OP-TEE sim

crypto
storage
SE

SVCs

~100 symbols
functions & globals

SVC parser

SVC dispatcher

Stubs

AFL instrumentation

Proxy TA
(SVC Invoke TA)
Improving results

• Rebuild essential kernel data structures

• Custom memory allocator “user-mode” memory
  • Register allocation in TA session ctx
  • Guard page after allocation (optional)
  • Unmap on free (optional)

• Optimize initialization
  • Fork loop only receives SVCs and executes them
  • Reuse process 1000x
  • Manually resetting global state (handles, objects, etc.) → deterministic exec

• Stubbing slow code that doesn’t impact behavior i.e. PRNG

• Coverage information using lcov
OP-TEE Sim
# LCOV - code coverage report

Current view: top level - /research/optee-svc/optee_os/core/tee

Test: coverage.info  
Date: 2019-09-13 13:21:30

<table>
<thead>
<tr>
<th>Filename</th>
<th>Line Coverage</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>fs dirfile.c</td>
<td>8.7%</td>
<td>12.5%</td>
</tr>
<tr>
<td>fs htree.c</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>tee cryp concat kdf.c</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>tee cryp hkdf.c</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>tee cryp pbkdf2.c</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>tee cryp util.c</td>
<td>69.9%</td>
<td>66.7%</td>
</tr>
<tr>
<td>tee fs key manager.c</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>tee fs rpc.c</td>
<td>7.3%</td>
<td>21.1%</td>
</tr>
<tr>
<td>tee fs rpc cache.c</td>
<td>37.1%</td>
<td>100.0%</td>
</tr>
<tr>
<td>tee obj1.c</td>
<td>57.6%</td>
<td>72.4%</td>
</tr>
<tr>
<td>tee obj2.c</td>
<td>39.8%</td>
<td>50.0%</td>
</tr>
<tr>
<td>tee tee fs.c</td>
<td>13.0%</td>
<td>25.8%</td>
</tr>
<tr>
<td>tee svc.c</td>
<td>38.8%</td>
<td>59.3%</td>
</tr>
<tr>
<td>tee svc cryp.c</td>
<td>63.8%</td>
<td>65.3%</td>
</tr>
<tr>
<td>tee svc storage.c</td>
<td>48.6%</td>
<td>70.8%</td>
</tr>
<tr>
<td>tee time generic.c</td>
<td>11.4%</td>
<td>20.0%</td>
</tr>
</tbody>
</table>

Generated by: LCOV version 1.14
Limitations

• RPC mechanism not implemented
  • Cripples secure storage

• No TA support
  • No fuzzing of TA loader, cross TA calls

• Several mechanisms stubbed
  • Mutexes, spinlocks, threads, …
Results

• 10x more unique inputs

• Up to 10x faster
  • 15 exec/s (doing real crypto) – 2500+ exec/s per CPU core
  • Scales on SMP systems!

• Found several new bugs!
  • Some issues are much easier to debug / find with the simulator!
Current status

- Most code open-sourced
  - Repository: [https://github.com/Riscure/optee_fuzzer](https://github.com/Riscure/optee_fuzzer)

- Upstreaming difficult / infeasible
  - Requires invasive core modifications
  - Licensing

- Needs to be rebased against latest version

- More things to triage than time allows 😞
Thank you! Any questions?

Martijn Bogaard
Senior Security Analyst
martijn@riscure.com / @jmartijnb