No lightsaber is needed to break the Wookey
Who are we?

David BERARD
- Security researcher @Synacktiv
- Vulnerability research & exploitation

Jérémie BOUTOILLE
- Security researcher @Synacktiv
- Vulnerability research & exploitation

Synacktiv
- Offensive security company
- Based in France
- ~70 Ninjas
- We are hiring!!!
## Introduction

### CESTI Challenge

- Organized every two years to evaluate ITSEF/CESTI laboratories
- Until this year:
  - Two challenges were organized, one for hardware CESTIs, and one for software CESTIs
  - CESTIs have different products to evaluate depending on their agreement categories.
- This year a unique challenge has been organized on a unique product
  - The objective is to evaluate software laboratories to do hardware testing and vice versa
  - Common target: **Wookey**
What is Wookey

- Open-Source and Open-hardware
- Developed by ANSSI
- Secure USB storage device
  - Encrypted data on an SD card
  - Authentication through a touchscreen
  - Double authentication: PET & User PIN
- Multiple smartcards are used for cryptographic operations
  - User smartcard for authentication and data decryption
  - DFU smartcard to enter in update mode
  - Firmware signature
- Firmware is unique per device (contains encrypted secrets)
Wookey : Hardware

Hardware design

- Main MCU : STM32F4
  - JTAG only on debug boards
  - Production boards rely on Read Out Protection (RDP=2) to disable JTAG
  - MPU used for the multitask OS

- Used interfaces
  - SPI for the display
  - ISO7816 to communicate with the smartcard
  - Buttons for DFU mode and reset
  - USB HS/LS for USB Mass Storage
  - UART for logs (may be used as input on debug board)
### Developers
- Full software stack developed by ANSSI and available on Github

### Languages
- Bootloader : C
- Micro-Kernel : ADA
- Drivers and Task : C

### OS
- Cortex-m4 MPU is used to isolate tasks
- Syscalls are handled by the ADA Micro-Kernel
- Task and drivers have permissions that are verified by the kernel in syscalls
Challenge Scope

Methodology

1. CESTI asked to write a full test plan
2. ANSSI reviewed test plan and selected few tests (hardware and software)
3. CESTI do their analysis based on selected tests
   a. 3 boards were given to CESTIs: prod board, dev board, STM32F4 discovery
4. CESTI write their assessment report and send it to ANSSI
5. ANSSI will organize a debriefing session with all CESTIs

Synacktiv selected tests

- SW: ADA kernel syscalls analysis and fuzzing
- SW: Fuzzing of the ISO7816 library which handles smartcard messages
- HW: Review of the secure channel establishment
- HW: Analysis of the RDP2 protection (used to disable JTAG) regarding its resistance to power glitches
Software: Syscall fuzzing

Very basic Syscall fuzzer

- On a development board
- Syscall fuzzer is built inside a userland task
- Choose a random syscall number
- Choose argument values in a list that contains
  - random values
  - limit values
  - valid pointer pointing to random data
  - ...
- Collect result on the UART: kernel crash logs on it (even on the production boards)
Software : Syscall fuzzing

Results

- Quickly got multiple crashes on multiple syscalls
- One of them allows writing a zero (32bits) at an arbitrary address!
- Trivial exploit
  - MPU_CTRL register is memory mapped and allows to disable the MPU
  - MPU is the only feature used to isolate task memory
  - Without MPU, tasks can read and write the kernel

<table>
<thead>
<tr>
<th>EXPLOIT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPU_CTRL is @ 0xe000ed94</td>
<td></td>
</tr>
<tr>
<td>Writing 0...</td>
<td></td>
</tr>
<tr>
<td>MPU should be turned off!</td>
<td></td>
</tr>
<tr>
<td>Looking for tasks @ 0x10000000</td>
<td></td>
</tr>
<tr>
<td>struct task is @ 0x100006e0</td>
<td></td>
</tr>
<tr>
<td>name = EXPLOIT</td>
<td></td>
</tr>
<tr>
<td>entry_point = 0x8090001</td>
<td></td>
</tr>
<tr>
<td>ttype = TASK_TYPE_USER</td>
<td></td>
</tr>
<tr>
<td>control = 0x3</td>
<td></td>
</tr>
<tr>
<td>setting to ttype = TASK_TYPE_KERNEL</td>
<td></td>
</tr>
<tr>
<td>control = 0x2</td>
<td></td>
</tr>
<tr>
<td>Privileged mode!</td>
<td></td>
</tr>
</tbody>
</table>
Software : Syscall review

Code review
- ADA is not so easy to read for people not familiar with it (like us).
- Some low impact bugs found

Results
- ADA protect you from basic memory bugs, but for a OS kernel the same bug classes as C can be present
- Use of ada.unchecked_conversion have to be double-checked
- Fuzzer found bugs we didn’t find during code audit.
Software : lib7816 fuzzing

Coverage guided fuzzing

- C library
- Easy to make it standalone
- Parse smartcard messages on a X64 PC
- Libfuzzer + ASAN

Result

- Good coverage
- No bug found
- Studing this library was helpful for hardware tests.
Hardware: Secure channel

Decoding ISO7816 frames

- Logic analyzer to capture traffic from/to the smartcard
- Modification of the ISO7816 Saleae decoder to add a PCAP export
- Custom Wireshark dissector to parse Wookey specific frames
Hardware: Secure channel

from this
Hardware: Secure channel

to this
Hardware: RDP

STM32 Read Out Protection

- STM32 Debug functionalities can be limited/disabled with this protection
- RDP configuration is saved in options bytes
- 1 byte for 3 different states:
  - RDP0: \texttt{0xAA} No protection (default), JTAG is enabled
  - RDP2: \texttt{0xCC} All debug features are disabled, no JTAG
  - RDP1: \texttt{All other values} Flash memory is protected against reading
- No downgrade possible from RDP2

STM32 Read Out Protection: Fault attack

- Many public research on the subject on STM32F1, STM32F2 and STM32F3 (power glitches, EM, laser)
- Downgrade from RDP2 to RDP1 by injecting fault during the BootROM startup
- A single bit flip when BootROM reads RDP option byte allows the downgrade
  - RDP1 state is coded with many values
- A public research show how the RDP1 state can be bypassed
**Hardware : RDP**

<table>
<thead>
<tr>
<th>STM32 Read Out Protection : Wookey</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wookey uses RDP2 to disable all debug features</td>
</tr>
<tr>
<td>Wookey developers are aware of these vulnerabilities, the bootloader contains mitigations</td>
</tr>
<tr>
<td>- Double checks are implemented in critical places</td>
</tr>
<tr>
<td>- RDP value is read by the bootloader and checked with 0xCC (RDP2)</td>
</tr>
<tr>
<td>In case of anomaly detection <strong>tasks are erased from the flash</strong></td>
</tr>
<tr>
<td>Our objective : fault Wookey’s STM32F4 RDP with a single fault with cheap hardware</td>
</tr>
</tbody>
</table>
Hardware: Power glitches setup

Board selection
- The Wookey board should not be modified
- Wookey project can be built for STM32F4 discovery board
- Discovery boards are not expensive, we can risk to break some
- Full schematics are available online
External MCU power supply

- To inject power glitches power supply must be finely controlled
- On discovery board a jumper can be removed to place an ampere meter (in blue)
  - Can be used to isolate the board internal power supply
  - External power supply can be connected on these PINs
- Reset PIN is available on headers
Hardware: Power glitches setup

removing Decoupling capacitors

- To inject power glitches power supply must be finely controlled
- Decoupling capacitors are here to stabilize MCU power supply
- Fault will be injected with power pulses
- Decoupling capacitors absorb such rapid power changes
- Unsolder them! (in red)
Hardware: Power glitches setup

Test bench

- **External power**: DPS3005 ~30€
- **Multiplexer**: MAX4619 ~1€
- **FPGA**: Arty A7-100T ~200€
- **SWD probe**: Another STM32F4 discovery board

FPGA

- Drive multiplexer to switch from external power to ground
- Forward Wookey’s UART logs to the PC
- Drive Wookey RST to reboot board
Hardware: Power glitches setup
Hardware: Fault injection, pulse generation

PC sends width and delay parameters to the FPGA (counted in FPGA cycle: 1ns)
2 FPGA toggles RST
3 FPGA waits delay cycles
4 FPGA toggles multiplexer control PIN: MCU power is now connected to ground
5 FPGA waits width cycles
6 FPGA toggles multiplexer control PIN: MCU power is now reconnect to power supply
7 PC tries a JTAG connection
PC collects UART logs during all these operations
Hardware: Fault injection, parameters

Find correct fault parameters

- Try all combinations of **width** and **delay**
- Width: 1 to 15 FPGA cycles
  - MCU doesn’t survive if glitches are more than 15 cycles wide
- Delay: 0 to 52,000 cycles
  - Easy to spot the bootloader initialization by looking at the UART
Hardware: Fault injection, parameters
Hardware: Fault injection, parameters
Hardware: Fault injection, collect data

On each try

- Try JTAG connection
- Collect bootloader logs for future analysis

```
====== Wookey Loader ======
Built date : Dec 19 2019 at 08:52:29
Board      : STM32F407
RDP_value  : 0xcc

=================================
Hard fault
  scb.hfsr 40000000  scb.cfsr 100
-- registers (frame at 20001f74, EXC_RETURN
 r0 5000000  r1 80  r2 7b
 r4 0  r5 8000188  r6 0  r7 ca0c
 r8 0  r9 0  r10 0  r11 0
 r12 0  pc 2035c30  lr 8003025
-- stack trace
  20001f70: 8003973 0 8000188 0
  20001f80: ca0c 0 0 0
  20001f90: 0 ffffffff 500000c 80
  20001fa0: 7b 500000c 0 8003025
  20001fb0: 2035c30 0 20001fc0 800123d
  20001fc0: 0 3000003 0 c
  20001fd0: 3 fc0ca3f3 20001fe0 80012e3
  20001fe0: 1 3000003 20001ff0 80012ff
Oops! Kernel panic!
```
## Hardware: Fault injection, results

### RDP downgrade: Results
- Wookey protections are resistant
- Bootloader detects RDP inconsistency
- Erase sensitive data and reboot the board

### Bootloader glitches
- Many glitches detected in UART log
  - PANIC
  - Values modification
  - State machine state changes
- Replaying parameters (**glitch** + **delay**) give a good reproduction rate
- Only the bootloader has protections
- **Other software components can also be targeted**
Hardware: Fault injection, enlarge your scope

libiso7816

- Already analyzed / fuzzed, no vulnerabilities found
- Handle smartcard messages before user authentication
- Rapid source code review to find a place where a glitch can create a software vulnerability
Hardware: Fault injection, smartcard library

```c
atr->h_num = atr->t0 & 0x0f;
for(i = 0; i < atr->h_num; i++){
    if(SC_getc_timeout(& atr->h[i]), WT_wait_time)){
        goto err;
    }
    checksum ^= atr->h[i];
}
```

**Answer To Reset message**
- ATR is the first message from the smartcard after reset
- Parsed by libiso7816

**ATR parsing**
- `atr->h` is a 16 bytes long stack buffer
- `atr->t0` value comes from the smartcard
- If a glitch affects `h_num` value a stack-overflow can occur
Hardware: Fault injection, smartcard library

Stack-Overflow
- `h_num` is computed from masked `atr -> t0` with a single instruction.
- Glitching this instruction will cause the usage of a non-masked value, and leads to overflow.

OK! but Wookey has stack cookies!
- Are you sure?
Hardware: Fault injection, smartcard library

Stack cookie code present, but not used
Hardware: Fault injection, smartcard library

config STACK_PROT_FLAG
  bool "Activate -fstack-protection-strong"
  default y

...  
config STACKPROTFLAGS
  string
  default "-fstack-protector-strong"
  depends on STACK_PROT_FLAGS

Typo in the build chain
Hardware : Fault injection, PoC

```c
void glitch_me() {
    char buffer[16] = {0};
    int size = 0;

    size = src_buffer[0] & 0x0F;
    memcpy(buffer, src_buffer, size);
}

int _main(uint32_t my_id) {
    // [...] 
    printf("init done.\n");
    glitch_me();
    printf("test ends\n");
}
```

- Patch the **BLINKY** demo task to add similar code
- Produce same assembly code for masking length
- UART logs *init done.* and *test ends* help to identify the temporal range to target
Hardware: Fault injection, PoC

- **VDD**
- **UART**
- **NRST**

**GLITCH**

Message «test ends»
Hardware: Fault injection, PoC

- Try all **delay** values in the targeted temporal range
- Expect **test ends** message on the UART
- Collect UART logs
- Got some **PC = 0x41414140** :)}
Hardware: Fault injection, PoC

Low reproduction rate

- Targeted code is after bootloader, OS initialization, many hardware interactions, etc.
- Execution of the targeted instruction is not stable
- Can be improved: 7 FPGA cycle look to be the optimal **width** value

Successful glitches parameters
Hardware: Fault injection, PoC

On the real device

- This research has only been done on the discovery board
- Attack on real devices require to implement smartcard protocol in the glitch setup
- Fault injection can be synchronized with ISO7816 frames to improve the reproduction rate
Conclusion: Impacts

<table>
<thead>
<tr>
<th>On the device</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glitch on the smartcard library allows gaining code execution</td>
</tr>
<tr>
<td>Can be chained with the EoP (syscall bug) to gain privileged code execution</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clone, by injecting dumped encrypted secret in a new Wookey</td>
</tr>
<tr>
<td>Modify firmware, privileged code can alter flash data</td>
</tr>
</tbody>
</table>
Conclusion: Impacts

**Encrypted data**
- Wookey design relies on smartcard for cryptographic operations
- Gaining code execution before user authentication does not allow decrypting data
- Complex attack scenarios (clone, steal and modify) can be used by an attacker to gain access to decrypted data
QUESTIONS?