BREAKING SAMSUNG'S ARM TRUSTZONE

Quarkslab

Black Hat USA 2019
OUR TEAM

Maxime Peterlin - R&D Engineer at Quarkslab
@pandasec_

Alexandre Adamski - R&D Engineer at Quarkslab
@NeatMonster_

Joffrey Guilbon - R&D Engineer at Quarkslab
@patateQBool
PRESENTATION OUTLINE

- Current state of embedded security
- Introduction to the ARM TrustZone technology
- Samsung's TrustZone Overview
- Trusted Components
- Vulnerability Research Tools
- Vulnerability Analysis
- Exploitation
- Post-Exploitation Demonstrations
CURRENT STATE OF EMBEDDED SECURITY
A LONG TIME AGO...

- Kernel unbreakable...?
HOW DO WE PROTECT OURSELVES...

- ... if the kernel is corrupted during the boot process?
- ... if the kernel is corrupted when the system is already running?
PROTECTION DURING THE BOOT PROCESS

Secure Boot

- Prevent the execution of untrusted or unauthorized code on end users devices
RUNTIME PROTECTION USING AN HYPERVISOR

- Hypervisor based guest kernel protection
- **Problem**: VM escapes and hypervisor compromissions
TRUSTED EXECUTION ENVIRONMENTS

Virtual Processor (e.g. ARM TrustZone)

On-SoC Processor (e.g. Apple SEP)

External Coprocessor (e.g. Google Titan M)

Taken from Le TEE, nouvelle ligne de défense dans les mobiles, SSTIC 2013
ARM TRUSTZONE TECHNOLOGY
OVERVIEW

ARM TrustZone is a system-wide hardware isolation mechanism

- **Hardware architecture**
  - Partitioning of all the SoC's hardware and software resources
  - TZPC, TZASC, TZMA, etc.

- **Software architecture**
  - Software implementation used in secure state
  - Communications between secure and non-secure components
SECURE AND NON-SECURE WORLDS

- **Secure World**
  - Runs trusted code
  - Performs sensitive operations
- **Normal World**
  - Considered as compromised by design
  - Performs non-sensitive operations
SECURE CONFIGURATION REGISTER

The **Secure** (or **Non-Secure**) state of the CPU is determined by the least significant bit of the **Secure Configuration Register** (SCR).

- **NS BIT [0]:**
  - 0 SECURE STATE
  - 1 NON-SECURE STATE

**SOFTWARE-CONTROLLED MU ориенти**

**BIOS-CONTROLLED SMI ориенти**

**HARDWARE-CONTROLLED SMI ориенти**
COMMUNICATING BETWEEN WORLDS

- Switches between worlds are performed by the **Secure Monitor**
  - Runs at the highest privilege level (EL3 in ARMv8/**Monitor Mode** in ARMv7)

- Data exchanged through
  - Exceptions
  - Interruptions
  - Writing to the PSTATE/CPSR registers (privileged operation)
# PRIVILEGES SEPARATION

## ARMv7 Privilege Levels

<table>
<thead>
<tr>
<th>Mode</th>
<th>Normal World</th>
<th>Secure World</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Mode (PL0)</td>
<td>APP</td>
<td>APP</td>
</tr>
<tr>
<td>Supervisor Mode (PL1)</td>
<td>GUEST OS</td>
<td>SECURE OS</td>
</tr>
<tr>
<td>Hypervisor Mode (PL2)</td>
<td>HYPERVERSOR</td>
<td></td>
</tr>
<tr>
<td>Monitor Mode</td>
<td></td>
<td>SECURE MONITOR</td>
</tr>
</tbody>
</table>

## ARMv8 Exception Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Normal World</th>
<th>Secure World</th>
</tr>
</thead>
<tbody>
<tr>
<td>EL0</td>
<td>APP</td>
<td>APP</td>
</tr>
<tr>
<td>EL1</td>
<td>GUEST OS</td>
<td>SECURE OS</td>
</tr>
<tr>
<td>EL2</td>
<td>HYPERVERSOR</td>
<td></td>
</tr>
<tr>
<td>EL3</td>
<td></td>
<td>SECURE MONITOR</td>
</tr>
</tbody>
</table>
THE TRUSTED COMPONENTS
FRAGMENTATION ISSUE

- **Privilege escalation by design**
  - No hardware isolation between S-EL1 and EL3
  - Access to all the physical memory
  - Will be fixed in ARMv8.4 with *Secure Partitions*

- **Fragmentation**
  - System developed by different vendors
  - Cooperation and mutual trust required
TRUSTZONE SOFTWARE ARCHITECTURE

Several implementations of the software stack running in TrustZone are possible
TRUSTZONE'S USE CASES

- Accessing hardware-backed features:
  - Cryptographic engine
  - Credentials storage (Hardware-backed Keystore)
  - True random number generator
  - ...
- Digital Rights Management (by leveraging the cryptographic engine)
- Protecting and monitoring of the Normal World by the Secure World
  - **Example:** Samsung's Real-Time Kernel Protection (RKP) and Periodic Kernel Measurement (PKM)
SAMSUNG'S ARM TRUSTZONE
OVERVIEW

- **Samsung Devices**
  - Use both Samsung's Exynos and Qualcomm's Snapdragon SoCs
    - The same phone models can have different SoCs depending on the country

- **Samsung's TrustZone**
  - Found only on Exynos SoCs
  - First used in the Samsung Galaxy S3
  - Trusted OS used:
    - **Kinibi** developed by Trustonic (Galaxy S3 to Galaxy S9)
    - **TEEGRIS** developed by Samsung (Galaxy S10)
    - Both are used in other models too
    - This talk will focus on **Kinibi**
PREVIOUS WORKS

- **Reverse Engineering Samsung S6 SBOOT** (2-part article series) by **Fernand Lone Sang**
  - ARM Trusted Firmware usage on Samsung devices and extraction process from an OTA of the TEE-OS

- **Unbox Your Phone** (3-part article series) by **Daniel Komaromy**
  - Reverse-engineering of the Trusted OS and exploitation of vulnerabilities in trustlets

- **Trust Issues: Exploiting TrustZone TEEs** by **Gal Beniamini**
  - Security analysis of different Trusted Execution Environments
NORMAL WORLD COMPONENTS

- Drivers, daemons, libraries and interfaces used for communicating with the Secure World
- Communications pass through SMCs and shared memory buffers
• ARM Trusted Firmware
  - Open-source reference implementation of Secure World software provided by ARM
  - Contains a modular secure monitor implementation
  - Custom SMC handlers, called runtime services, can be added to fit the vendors requirements
  - Example: runtime services are used by Samsung to forward SMCs handled by Kinibi
SECURE WORLD COMPONENTS

- Secure world based on a micro-kernel architecture
MTK: KINIBI'S MICRO KERNEL

- Kinibi is a 32-bit OS developed by Trustonic
  - Used to be called Mobicore and t-base
- **MTK**: micro-kernel and only component running in S-EL1
- Provides syscalls (SVCs)
  - Memory mapping, process creation, SMCs, etc.
  - SVCs available depend on the privileges of the calling process
- Loads other components (embedded drivers, etc.) and especially **RTM**
RUN-TIME MANAGER

- Special Secure World trusted application equivalent to the init process on Linux
- **Main tasks**
  - starting and managing processes
  - notifying trustlets of incoming data from the NWd
- **Implements communication channels**
  - Inter-Process Communications
  - Mobicore Communication Interface (MCI)
    - A communication channel with the Normal World based on the Mobicore Control Protocol (MCP)
MCLIB: KINIBI'S STANDARD LIBRARY

- Provides standard functions to Trusted Applications, Secure Drivers and RTM
- Separated into two APIs:
  - **TlApi**: set of functions used by trusted applications
  - **DrApi**: set of functions used by secure drivers
- Useful during exploitation to find gadgets

**TlApi call example**

```plaintext
; _DWORD tlApiWaitNotification(_DWORD timeout)
MOV.W R1, #0x1000
LDR.W R2, [R1, #(tlApiLibEntry - 0x1000)]
MOV R1, R0
MOVS R0, #6
BX R2
```
TRUSTED APPLICATIONS

- Secure World equivalent of regular applications in the Normal World (run at S-EL0)
- Allow trusted third-parties to extend the functionalities of the TEE-OS
  - Trusted UI, DRM, storage of secrets, etc.
- Signed binaries loaded directly from the Normal World (so are SDs)
TRUSTED APPLICATIONS LIFE-CYCLE

- Communications with the Normal World made through world-shared memory (named TCI buffer by Trustonic)
- The TCI buffer contains commands to be handled by the trustlet
- TCI buffer contains commands to be handled by the trustlet

- **Notifications**
  - tlApiWaitNotification
  - tlApiNotify
SECURE DRIVERS

- Special type of Trusted Applications
- Run at S-EL0 but have higher software-define privileges
- Have access to a richer set of API and syscalls
- Are used by trustlets as an interface to access physical memory and reach secure peripherals in a controlled manner
- Communications with TAs made through IPCs and shared memory
SECURE DRIVERS LIFE-CYCLE

- **Multi-threaded application**
  - **DCI**: Normal World communications
  - **IPC**: trustlet communications

- **Trustlet interactions**
  - Retrieves IPC data by mapping the entire trustlet
  - Notifications using `drApiIpcCallToIPCH`
VULNERABILITY RESEARCH TOOLS
ATTACK SURFACE
ATTACK SURFACE
ATTACK SURFACE
ATTACK SURFACE

- Must be reachable from the Normal World
- ATF is open-source, probably heavily reviewed
- Trusted Applications are low-hanging fruits
OUR JOURNEY IN 5 STEPS
STEP #1 - LOADING INTO IDA/GHIDRA
- Proprietary File Format - MobiCore Loadable Format (MCLF)
• mclf-ida-loader

• mclf-ghidra-loader
STEP #2 - IDENTIFYING FUNCTIONS
MCLIB - STANDARD LIBRARY
- Renames tlApi/drAPI functions
- Sets the functions prototypes
STEP #3 - MANUALLY FINDING VULNERABILITIES

LOADERS

EMULATOR

SCRIPTS
TRUSTLETS EMULATOR

• Based on Unicorn (external project)
• Split into simple tasks:
  ▪ Loading the MCLF binary
  ▪ Mapping the shared memory buffer
  ▪ Hooking the McLib functions
TRUSTLETS EMULATOR

```python
python emulator.py *41.tlb \cmd1.cmd --tci 0x40100 -v
[+] Binary is a trustlet
[+] Trustlet size = 0x1ba4c
[+] Mapping text section at 0x00001000 with a size of 0x4874
[+] Mapping data section at 0x00007000 with a size of 0x168
[+] Mapping BSS section at 0x00007168 with a size of 0x17070
[+] Mapping region at 0x07d00000 (0x1 bytes)
[+] Mapping TCI buffer at 0x00100000 with a size of 0x40100
[i] drApiLogvPrintf(u'ICCC:Trustlet ICCC::Starting\n')
[+] Loading input data
[i] drApiLogvPrintf(u'ICCC: we got a command: 1\n')
[i] drApiLogvPrintf(u'ICCC: Initialize failed - tamper fuse set\n')
[i] drApiLogvPrintf(u'ICCC: Measurements result ret = 65548, ret hex = 1000c\n')
[i] drApiLogvPrintf(u'ICCC: save data@\n')
[i] drApiLogvPrintf(u'Iccc_phys_read failed\n')
[i] drApiLogvPrintf(u'Iccc_check_magic failed\n')
[i] drApiLogvPrintf(u'End of ICCC_Init, ret=1000c\n')
[i] drApiLogvPrintf(u'Iccc: Error writing Trustboot flag\n')
[+] tlApiNotify: Quitting!
```
STEP #4 - FINDING VULNERABILITIES AUTOMATICALLY

LOADER

EMULATOR

SCRIPTS

FUZZER
TRUSTLETS FUZZER

- Based on AFL_Unicorn (internal project)
  - Interfaces the fuzzer AFL with Unicorn
  - Usability and performance improvements
  - 100% of the code is written in Python!
# TRUSTLETS FUZZER

<table>
<thead>
<tr>
<th>process timing</th>
<th>overall results</th>
</tr>
</thead>
<tbody>
<tr>
<td>run time</td>
<td>cycles done: 0</td>
</tr>
<tr>
<td>last new path</td>
<td>total paths: 65</td>
</tr>
<tr>
<td>last uniq crash</td>
<td>uniq crashes: 21</td>
</tr>
<tr>
<td>last uniq hang</td>
<td>uniq hangs: 0</td>
</tr>
<tr>
<td>cycle progress</td>
<td>map coverage</td>
</tr>
<tr>
<td>now processing</td>
<td>map density: 0.02% / 1.79%</td>
</tr>
<tr>
<td>paths timed out</td>
<td>count coverage: 1.17 bits/tuple</td>
</tr>
<tr>
<td>stage progress</td>
<td>findings in depth</td>
</tr>
<tr>
<td>now trying: havoc</td>
<td>favored paths: 1 (1.54%)</td>
</tr>
<tr>
<td>stage execs: 4253/6528 (65.15%)</td>
<td>new edges on: 65 (100.00%)</td>
</tr>
<tr>
<td>total execs: 5407</td>
<td>total crashes: 88 (21 unique)</td>
</tr>
<tr>
<td>exec speed: 21.94/sec (slow!)</td>
<td>total tmouts: 0 (0 unique)</td>
</tr>
<tr>
<td>fuzzing strategy yields</td>
<td>path geometry</td>
</tr>
<tr>
<td>bit flips: 2/32, 1/31, 2/29</td>
<td>levels: 2</td>
</tr>
<tr>
<td>byte flips: 1/4, 0/3, 0/1</td>
<td>pending: 65</td>
</tr>
<tr>
<td>arithmetics: 10/224, 0/204, 0/68</td>
<td>pendl fav: 1</td>
</tr>
<tr>
<td>known ints: 1/8, 0/18, 0/10</td>
<td>own finds: 64</td>
</tr>
<tr>
<td>dictionary: 0/0, 0/0, 0/0</td>
<td>imported: n/a</td>
</tr>
<tr>
<td>havoc: 0/0, 0/0, 0/0</td>
<td>stability: 100.00%</td>
</tr>
<tr>
<td>trim: 50.00%/1, 0.00%</td>
<td></td>
</tr>
</tbody>
</table>
TRUSTLETS SYMBOLIC EXECUTOR

- Based on Manticore by Trail of Bits
- Uses very simple strategies:
  - Mark the shared memory buffer symbolic
  - Explore all the paths of the trustlet
  - Check reads or writes to memory
  - Ask the solver for an invalid address
CRASH EXAMPLE

Command line:
`'../tainter.py -s 1036 fffffffff00000000000000000000000000000005.tlbinfo -t -c coverage.txt'`

Status:
Invalid symbolic memory access (mode:r)

Memory:
```
000000000001000-0000000000000000 r x 00000094 ffffffff00000000000000000000000000000005.tlbinfo
0000000000009000-000000000024000 rw 00006dc8 ffffffff00000000000000000000000000000005.tlbinfo
000000000010000-000000000101000 rw 00000000
0000000007d00000-000000007d01000 rx 00000000 CPU:
```

INSTRUCTION: 0x000000000000059ec:      pid  [r1, #0x80]  
APSR: 0x000000000000000  
R0 : 0x0000000000000000a0ac  
R1 : <BitVecExtract at 7f2571dbdeb8-T>  
R10: 0x0000000000000000  
R11: 0x0000000000000000  
R12: 0x0000000000000000  
R13: 0x000000000000023a20
STEP #5 - EXPLOITING THE VULNERABILITIES

LOADER  EMULATOR  BINDINGS

SCRIPTS  FUZZER
SOFTWARE STACK

NORMAL WORLD

- APP
- SERVICE PROVIDER PROVISIONING AGENT
- ROOT PROVISIONING AGENT
- KINIBI CLIENT API
- KINIBI DAEMON
- KINIBI DRIVERS
- EMBEDDED KERNEL OS
- SMC

TRUSTED APPLICATION CONNECTOR

- TRUSTED APPLICATION COMMUNICATION INTERFACE (TCI)
- SHARED NON-SECURE MEMORY

SECURE WORLD

- SP/OEM SYSTEM CONTAINERS
- SECURE DRIVERS
- SYSTEM TA
- DRIVER TA
- TLAPI
- MCLIB
- DRAPI
- DRCRYPT
- RTM
- SYSCALL HANDLERS
- KINIBI MICRO-KERNEL
- MONITOR (ATF BL31)
- TEE-OS DISPATCHER

FORWARD TO CUSTOM TEE-OS HANDLER

SECURE DRIVERS
TRUSTED APPLICATION CONNECTOR
ROOT PROVISIONING AGENT
EMBEDDED KERNEL OS
CLIENT API
PYTHON BINDINGS

- Writing C is tedious, writing Python is a lot easier
- Bindings of the mcClient API called pymcclient
- Provides various utilities: hexdump, (dis)assemble, etc.
- Provides a command interpreter which is based on IPython
with Device(DEVICE_ID) as dev:
    with dev.buffer(TCI_BUFFER_SIZE) as tci:
        with open(TRUSTLET_FILE, "rb") as fd:
            buf = fd.read()

        with Trustlet(dev, tci, buf) as app:
            tci.seek(0)
            tci.write_dword(1)
            app.notify()
            app.wait_notification()
            tci.seek(0)
            print(tci.read_dword())
VULNERABILITY ANALYSIS & EXPLOITATION
OVERVIEW

• **Target:**
  - Samsung Galaxy S7 running Android 7.0

• **Main goal:**
  - Obtaining code execution in EL3

• **Prerequisites:**
  - Being part of the radio group
  - Being able to write files somewhere on the device
ATTACK PLAN

NORMAL WORLD

- APP
- SERVICE PROVIDER PROVISIONING AGENT
- ROOT PROVISIONING AGENT
- KINIBI CLIENT API
- KINIBI DAEMON
- KINIBI DRIVERS
- EMBEDDED KERNEL

SECURE WORLD

- TRUSTED APPLICATION COMMUNICATION INTERFACE (TCI)
- SYSTEM TA
- DRIVER TA
- TLAPI
- DRCRYPT
- SMC
- KINIBI MICRO-KERNEL
- SYSCALL HANDLERS
- MCLIB
- DRAPI

SHARED NON-SECURE MEMORY

FOREST CHASE

MONITOR (ATF BL 31)

FORWARD TO CUSTOM TEE-OS HANDLER

TEE-OS DISPATCHER

HERE!
# SOFTWARE MITIGATIONS

<table>
<thead>
<tr>
<th>Model</th>
<th>XN bit</th>
<th>Canary</th>
<th>ASLR</th>
<th>PIE</th>
</tr>
</thead>
<tbody>
<tr>
<td>S6</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>S7</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>S8</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>S9</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
ATTACKING A TRUSTED APPLICATION

Overview
ATTACKING A TRUSTED APPLICATION

SEM Trustlet Vulnerability

Stack-based buffer overflow in the handler of the command ID #5

Call to memcpy at the beginning of the 5th command handler

Before this call, the registers are set as follow:

- R0 = SP+0x4F8-0xF0, the destination buffer
- R1 = tci_buffer + 0x8, the source buffer
- R2 = *(tci_buffer + 0x16808), the length of the buffer

```
.text:00020FD2
.text:00020FD6
.text:00020FBE
.text:00020FC8
.text:00020FC0
.text:00020FC4

ADD W R1, R0, #0x16800
MOV R4, R1
LDR W R2, [R1,#0x888]
ADD W R1, R0, #8
ADD W R0, SP, #0x4F8+var_F0
BLX memcpy_aligned
```
ATTACKING A TRUSTED APPLICATION

Exploitation Results

- Code execution in S-EL0
- **It is now possible to:**
  - Communicate with Secure Drivers
  - Make some syscalls (e.g. print characters, get system information, etc.)
- **Next target:** Secure Driver
ATTACKING A SECURE DRIVER
VALIDATOR Secure Driver Vulnerability

A vulnerability was found in the **VALIDATOR secure driver**

Stack-based buffer overflow in the handler of the command ID #15

Equivalent to the one found in the trustlet (i.e. `memcpy` in the stack and a user-controlled size)

```
.text:00001362
.text:00001364
.text:00001366
.text:00001368
MOVS
MOV
ADDS
BLX
R2, #0x37 ; '7'
R1, R4
R0, R6, #1
memcpy
```
ATTACKING A SECURE DRIVER

Exploitation Results

- Code execution in S-EL0 but with higher privileges
- It is now possible to:
  - Communicate with the RunTime Manager (or RTM, an init-like process)
  - Access more syscalls (e.g. map physical memory, create threads, make SMCs, etc.)
- Next target: Trusted OS & Monitor
ATTACKING KINIBI AND THE MONITOR

*Vulnerability Analysis*

- **mmap**: secure and non-secure physical memory mapping syscall

- **Vulnerability**
  - Monitor mapped at 0x2022000
  - Can be mapped using `mmap` to modify an SMC
  - Calling the hijacked SMC allows code execution in EL3

- **Patch**
  - Fixed in the newest versions by using a blacklist
ATTACKING KINIBI AND THE MONITOR

Exploitation Results

- Code execution in **EL3**
- Now possible to do anything we want!
POST-EXPLOITATION
TRUSTPWN FRAMEWORK

- Based on the previous vulnerabilities
- **Internals**
  - Uses the EL3 vulnerability to have arbitrary access to Kinibi
  - Adds a SVC and a drApi function to execute code in S-EL1 "natively"
    - SVCs and drApi functions are referenced in pointer arrays
- **Usage**
  - Read or write memory arbitrarily
  - Execute code in S-EL1 and EL3
DEMÖ

Finding the Master Key in the Monitor
FINDING THE MASTER KEY IN THE MONITOR

DrApi Reversing

- Reversing the crypto-driver `drcrypto` (found embedded in Kinibi)

- **DrApi 0x1030**
  - Takes four possible command IDs (0xAA, 0xAB, 0xAC, 0xAD)
    - The interesting one is 0xAB
  - Wrapper around SMC 0xB200005

- **SMC 0xB200005**
  - SMC arguments:
    - **R0**: SMC ID
    - **R0**: command number (four possible values [0-3])
    - **R1**: number of bytes to read
  - Reads 0x10 bytes of the master key at 0x101E4000
FINDING THE MASTER KEY IN THE MONITOR

DrApi Function
FINDING THE MASTER KEY IN THE MONITOR

SMC Function
Bypassing Signature Checks
BYPASSING SIGNATURE CHECKS

Methodology

- Reversing RTM
- Finding the SHA-256 of the public key corresponding to the private key used to sign TAs and SDs
- Signature is verified using t1ApiSignatureVerify
- Patch the checks and load your own TA or SD
**FINDING THE MASTER KEY IN THE MONITOR**

*RTM Verifications*

**First check**

<table>
<thead>
<tr>
<th>ROM: 00006E62</th>
<th>BL</th>
<th>tIApiSignatureVerify</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM: 00006E66</td>
<td>LDR</td>
<td>R4, =0x40000009</td>
</tr>
<tr>
<td>ROM: 00006E68</td>
<td>ADDS</td>
<td>R4, R4, #6</td>
</tr>
<tr>
<td>ROM: 00006E6A</td>
<td>CBNZ</td>
<td>R0, loc_6E7A</td>
</tr>
<tr>
<td>ROM: 00006E6C</td>
<td>LDRB.W</td>
<td>R0, [SP, #0x40000009+var_44]</td>
</tr>
</tbody>
</table>

**Second check**

<table>
<thead>
<tr>
<th>ROM: 000073E0</th>
<th>BL</th>
<th>tIApiSignatureVerify</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM: 000073E4</td>
<td>CBNZ</td>
<td>R0, loc_73FA</td>
</tr>
<tr>
<td>ROM: 000073E6</td>
<td>LDRB.W</td>
<td>R0, [SP, #0x10000000+var_48]</td>
</tr>
</tbody>
</table>
DEMO

Trusted-OS Instrumentation
TRUSTED-OS INSTRUMENTATION

Methodology

- Handles ARMv7 and Thumb
- Based on the **Undefined Instruction** exception
- **Undefined Instruction** handler is replaced by our own code
- Patch an instruction with the ARM undefined instruction UDF 0xNNNN
- When a breakpoint triggers the current context of the CPU is saved
  - Current context is saved
  - Overwritten instruction is executed
THANK YOU!