Understanding the Security of ARM Debugging Features

Zhenyu Ning and Fengwei Zhang

COMPASS Lab
Wayne State University

May 21, 2019
Outline

- Introduction
- Obstacles for Attacking the Traditional Debugging
- Nailgun Attack
- Mitigations
- Conclusion
Outline

- Introduction
- Obstacles for Attacking the Traditional Debugging
- Nailgun Attack
- Mitigations
- Conclusion
Introduction

Modern processors are equipped with hardware-based debugging features to facilitate on-chip debugging process.

- E.g., hardware breakpoints and hardware-based trace.

- It normally requires cable connection (e.g., JTAG [1]) to make use of these features.
Traditional Debugging

Debug Target (TARGET)

JTAG Interface

Debug Host (HOST)

Understanding the Security of ARM Debugging Features, S&P 19
Traditional Debugging

Debug Target (TARGET)

JTAG Interface

Debug Host (HOST)

Understanding the Security of ARM Debugging Features, S&P 19
Traditional Debugging

Debug Authentication

Debug Target (TARGET)

JTAG Interface

Debug Host (HOST)
Traditional Debugging

Debug Authentication

JTAG Interface

Debug Target (TARGET)

Debug Host (HOST)

Security?
Security? We have obstacles for attackers!

- Obstacle 1: Physical access.
- Obstacle 2: Debug authentication mechanism.
Introduction

Security? We have obstacles for attackers!

- **Obstacle 1**: Physical access.
- **Obstacle 2**: Debug authentication mechanism.

Do these obstacles work?
Outline

» Introduction

» **Obstacles for Attacking the Traditional Debugging**

» Nailgun Attack

» Mitigations

» Conclusion
Obstacles for Attacking the Traditional Debugging

Obstacles for attackers:

- **Obstacle 1**: Physical access.
- **Obstacle 2**: Debug authentication mechanism.

Does it really require physical access?
Inter-Processor Debugging

We can use one processor on the chip to debug another one on the same chip, and we refer it as inter-processor debugging.

- Memory-mapped debugging registers.
  - Introduced since ARMv7.

- No JTAG, No physical access.
Obstacles for Attacking the Traditional Debugging

Obstacles for attackers:

- **Obstacle 1**: Physical access.
- **Obstacle 2**: Debug authentication mechanism.

Does debug authentication work as expected?
Processor in Normal State

TARGET
(Normal State)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register 1</th>
<th>Register 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>x3, #3</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>x4, #4</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>x0, x3</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>x1, x4</td>
<td></td>
</tr>
<tr>
<td>LDR</td>
<td>pc, [pc, #-0x10]</td>
<td></td>
</tr>
</tbody>
</table>

TARGET is executing instructions pointed by pc
### Processor in Non-invasive Debugging

**TARGET**  
(Normal State)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register 1</th>
<th>Register 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>x3</td>
<td>#3</td>
</tr>
<tr>
<td>MOV</td>
<td>x4</td>
<td>#4</td>
</tr>
<tr>
<td>MOV</td>
<td>x0,</td>
<td>x3</td>
</tr>
<tr>
<td>MOV</td>
<td>x1,</td>
<td>x4</td>
</tr>
<tr>
<td>LDR</td>
<td>pc,</td>
<td>[pc, #0x10]</td>
</tr>
</tbody>
</table>

**Non-invasive Debugging**: Monitoring without control
Processor in Invasive Debugging

**TARGET**
(Debug State)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>x3, #3</td>
</tr>
<tr>
<td>MOV</td>
<td>x4, #4</td>
</tr>
<tr>
<td>MOV</td>
<td>x0, x3</td>
</tr>
<tr>
<td>MOV</td>
<td>x1, x4</td>
</tr>
<tr>
<td>LDR</td>
<td>pc, [pc, #-0x10]</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

**Invasive Debugging:** Control and change status
**ARM Debug Authentication Mechanism**

**TARGET**  
(Normal State)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>x3, #3</td>
</tr>
<tr>
<td>MOV</td>
<td>x4, #4</td>
</tr>
<tr>
<td>MOV</td>
<td>x0, x3</td>
</tr>
<tr>
<td>MOV</td>
<td>x1, x4</td>
</tr>
<tr>
<td>LDR</td>
<td>pc, [pc, #-0x10]</td>
</tr>
</tbody>
</table>

**Debug Authentication Signal**: Whether debugging is allowed
ARM Debug Authentication Mechanism

TARGET
(Normal State)

<table>
<thead>
<tr>
<th>pc</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>x3, #3</td>
</tr>
<tr>
<td>MOV</td>
<td>x4, #4</td>
</tr>
<tr>
<td>MOV</td>
<td>x0, x3</td>
</tr>
<tr>
<td>MOV</td>
<td>x1, x4</td>
</tr>
<tr>
<td>LDR</td>
<td>pc, [pc, #-0x10]</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Four signals for: Secure/Non-secure, Invasive/Non-invasive
ARM Ecosystem

- ARM licenses technology to the SoC Vendors.
  - E.g., ARM architectures and Cortex processors

- Defines the debug authentication signals.
ARM Ecosystem

- The SoC Vendors develop chips for the OEMs.
  - E.g., Qualcomm Snapdragon SoCs

- **Implement** the debug authentication signals.
The OEMs produce devices for the users.
- E.g., Samsung Galaxy Series and Huawei Mate Series

Configure the debug authentication signals.
Finally, the User can enjoy the released devices.
- Tablets, smartphones, and other devices

Learn the status of debug authentication signals.
Obstacles for Attackers:

- **Obstacle 1**: Physical access.

- **Obstacle 2**: Debug authentication mechanism.

Does debug authentication work as expected?
Debug Authentication Signals

- What is the status of the signals in real-world device?

- How to manage the signals in real-world device?
# Debug Authentication Signals

<table>
<thead>
<tr>
<th>Category</th>
<th>Platform / Device</th>
<th>Debug Authentication Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DBGEN</td>
</tr>
<tr>
<td>Development Boards</td>
<td>ARM Juno r1 Board</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td>NXP i.MX53 QSB</td>
<td>✗</td>
</tr>
<tr>
<td>IoT Devices</td>
<td>Raspberry PI 3 B+</td>
<td>✔️</td>
</tr>
<tr>
<td>Cloud Platforms</td>
<td>64-bit ARM miniNode</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td>Packet Type 2A Server</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td>Scaleway ARM C1 Server</td>
<td>✔️</td>
</tr>
<tr>
<td>Mobile Devices</td>
<td>Google Nexus 6</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>Samsung Galaxy Note 2</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td>Huawei Mate 7</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td>Motorola E4 Plus</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td>Xiaomi Redmi 6</td>
<td>✔️</td>
</tr>
</tbody>
</table>

Understanding the Security of ARM Debugging Features, S&P 19
## Debug Authentication Signals

**Table:** Debug Authentication Signals on Real Devices.

<table>
<thead>
<tr>
<th>Category</th>
<th>Platform / Device</th>
<th>Debug Authentication Signals</th>
<th>DBGEN</th>
<th>NIDEN</th>
<th>SPIDEN</th>
<th>SPNIDEN</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Development Boards</strong></td>
<td>ARM Juno r1 Board</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>NXP i.MX53 QSB</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><strong>IoT Devices</strong></td>
<td>Raspberry PI 3 B+</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Cloud Platforms</strong></td>
<td>64-bit ARM miniNode</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Packet Type 2A Server</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Scaleway ARM C1 Server</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Mobile Devices</strong></td>
<td>Google Nexus 6</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>Samsung Galaxy Note 2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>Huawei Mate 7</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Motorola E4 Plus</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Xiaomi Redmi 6</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
## Table: Debug Authentication Signals on Real Devices.

<table>
<thead>
<tr>
<th>Category</th>
<th>Platform / Device</th>
<th>Debug Authentication Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DBGEN</td>
</tr>
<tr>
<td>Development Boards</td>
<td>ARM Juno r1 Board</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>NXP i.MX53 QSB</td>
<td>✗</td>
</tr>
<tr>
<td>IoT Devices</td>
<td>Raspberry PI 3 B+</td>
<td>✓</td>
</tr>
<tr>
<td>Cloud Platforms</td>
<td>64-bit ARM miniNode</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Packet Type 2A Server</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Scaleway ARM C1 Server</td>
<td>✓</td>
</tr>
<tr>
<td>Mobile Devices</td>
<td>Google Nexus 6</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>Samsung Galaxy Note 2</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Huawei Mate 7</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Motorola E4 Plus</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Xiaomi Redmi 6</td>
<td>✓</td>
</tr>
</tbody>
</table>
Debug Authentication Signals

How to manage the signals in real-world device?

▶ For both development boards with manual, we cannot fully control the debug authentication signals.

- Signals in i.MX53 QSB can be enabled by JTAG.

- The DBGEN and NIDEN in ARM Juno board cannot be disabled.

▶ In some mobile phones, we find that the signals are controlled by One-Time Programmable (OTP) fuse.

For all the other devices, nothing is publicly available.
Obstacles for attackers:

- **Obstacle 1:** Physical access.
  We don’t need physical access to debug a processor.

- **Obstacle 2:** Debug authentication mechanism.
  The debug authentication mechanism allows us to debug the processor.
Outline

▶ Introduction

▶ Obstacles for Attacking the Traditional Debugging

▶ **Nailgun Attack**

▶ Mitigations

▶ Conclusion
Inter-processor Debugging

Debug Target (TARGET)

Memory-mapped Interface

Debug Host (HOST)
Inter-processor Debugging

Memory-mapped Interface

Debug Target (TARGET)

Debug Host (HOST)
Nailgun Attack

A Multi-processor SoC System

An example SoC system:

- Two processors as HOST and TARGET, respectively.
- Low-privilege and High-privilege resource.
Nailgun Attack

A Multi-processor SoC System

High-privilege Resource (Secure RAM/Register/Peripheral)

Low-privilege Resource (Non-Secure RAM/Register/Peripheral)

- Low-privilege refers to non-secure kernel-level privilege
- High-privilege refers to any other higher privilege
Nailgun Attack

A Multi-processor SoC System

Both processors are only access low-privilege resource.

- Normal state
- Low-privilege mode
HOST sends a **Debug Request** to TARGET,

- TARGET checks its authentication signal.
- Privilege of HOST is ignored.
Nailgun Attack

A Multi-processor SoC System

TARGET (Debug State) (Low Privilege)
HOST (Normal State) (Low Privilege)

High-privilege Resource (Secure RAM/Register/Peripheral)
Low-privilege Resource (Non-Secure RAM/Register/Peripheral)

TARGET turns to **Debug State** according to the request.

- Low-privilege mode
- No access to high-privilege resource
HOST sends a **Privilege Escalation Request** to TARGET,

- E.g., executing DCPS series instructions.
- The instructions can be executed at any privilege level.
TARGET turns to **High-privilege Mode** according to the request.

- Debug state, high-privilege mode
- Gained access to high-privilege resource
HOST sends a Resource Access Request to TARGET,

- E.g., accessing secure RAM/register/peripheral.
- Privilege of HOST is ignored.
TARGET return the result to HOST,
▶ i.e., content of the high-privilege resource.
▶ Privilege of HOST is ignored.
HOST gains access to the high-privilege resource while running in,

- Normal state
- Low-privilege mode

Understanding the Security of ARM Debugging Features, S&P 19
Nailgun Attack

Nailgun: Break the privilege isolation of ARM platform.

- Achieve access to high-privilege resource via misusing the ARM debugging features.

- Can be used to craft different attacks.
Attack Scenarios

- Implemented Attack Scenarios:
  - Inferring AES keys from TrustZone.
  - Read Secure Configuration Register (SCR).
  - Arbitrary payload execution in TrustZone.

- Covered Architectures:
  - ARMv7, 32-bit ARMv8, and 64-bit ARMv8 architecture.

- Vulnerable Devices:
  - Development boards, IoT devices, cloud platforms, mobile devices.
Nailgun Attack

Fingerprint extraction in commercial mobile phone.

- Device: Huawei Mate 7 (MT-L09)
- Firmware: MT7-L09V100R001C00B121SP05
- Fingerprint sensor: FPC1020
Nailgun Attack

- Step 1: Learn the location of fingerprint data in secure RAM.
  - Achieved by reverse engineering.

- Step 2: Extract the data.
  - With the inter-processor debugging in Nailgun.

- Step 3: Restore fingerprint image from the extracted data.
  - Read the publicly available sensor manual.
Nailgun Attack

- The right part of the image is blurred for privacy concerns.
- Source code: https://compass.cs.wayne.edu/nailgun/
Disclosure

- March 2018: Preliminary findings are reported to ARM
- August 2018: Report to ARM and related OEMs with enriched result
- October 2018: Issue is reported to MITRE
- February 2019: PoCs and demos are released
- April 2019: CVE-2018-18068 is released
Outline

- Introduction
- Obstacles for Attacking the Traditional Debugging
- Nailgun Attack
- Mitigations
- Conclusion
Mitigations

Simply *disable* the signals?
Mitigations

Simply disable the authentication signals?

- Existing tools rely on the debug authentication signals.
  - E.g., [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

- Unavailable management mechanisms.

- OTP feature, cost, and maintenance.
Mitigations

We suggest a comprehensive defense across different roles in the ARM ecosystem.

▶ For ARM, additional restriction in inter-processor debugging model.

▶ For SoC vendors, refined signal management and hardware-assisted access control to debug components.

▶ For OEMs and cloud providers, software-based access control.
Outline

▶ Introduction
▶ Obstacles for Attacking the Traditional Debugging
▶ Nailgun Attack
▶ Mitigations
▶ Conclusion
Conclusion

- We present a study on the security of hardware debugging features on ARM platform.
- “Safe” components in legacy systems may be vulnerable in advanced systems.
- We suggest a comprehensive rethink on the security of legacy mechanisms.
References


Thank you!

Questions?

zhenyu.ning@wayne.edu

http://compass.cs.wayne.edu
Nailgun in different ARM architecture

- **64-bit ARMv8 architecture: ARM Juno r1 board.**
  - Embedded Cross Trigger (ECT) for debug request.
  - Binary instruction to Instruction Transfer Register (ITR).

- **32-bit ARMv8 architecture: Raspberry PI Model 3 B+.**
  - Embedded Cross Trigger (ECT) for debug request.
  - First and last half of binary instruction should be reversed in ITR.

- **ARMv7 architecture: Huawei Mate 7.**
  - Use Debug Run Control Register for debug request.
  - Binary instruction to Instruction Transfer Register (ITR).
Instruction Execution in Debug State

**TARGET**
(Normal State)

<table>
<thead>
<tr>
<th>pc</th>
<th>MOV</th>
<th>x3, #3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MOV</td>
<td>x4, #4</td>
</tr>
<tr>
<td></td>
<td>MOV</td>
<td>x0, x3</td>
</tr>
<tr>
<td></td>
<td>MOV</td>
<td>x1, x4</td>
</tr>
<tr>
<td></td>
<td>LDR</td>
<td>pc, [pc, #-0x10]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

In normal state, TARGET is executing instructions pointed by `pc`
In debug state, TARGET stops executing the instruction at pc
### Instruction Execution in Debug State

**TARGET**
*Debug State*

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>x3,</td>
<td>#3</td>
</tr>
<tr>
<td>MOV</td>
<td>x4,</td>
<td>#4</td>
</tr>
<tr>
<td>MOV</td>
<td>x0,</td>
<td>x3</td>
</tr>
<tr>
<td>MOV</td>
<td>x1,</td>
<td>x4</td>
</tr>
<tr>
<td>LDR</td>
<td>pc,</td>
<td>[pc, #0x10]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ITR**

Binary Instruction

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>x4,</td>
<td>#0</td>
</tr>
</tbody>
</table>

In debug state, write binary instruction to ITR for execution.
Instruction Execution in Debug State

**TARGET**
(Debug State)

<table>
<thead>
<tr>
<th>pc</th>
<th>MOV</th>
<th>x3, #3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MOV</td>
<td>x4, #4</td>
</tr>
<tr>
<td></td>
<td>MOV</td>
<td>x0, x3</td>
</tr>
<tr>
<td></td>
<td>MOV</td>
<td>x1, x4</td>
</tr>
<tr>
<td></td>
<td>LDR</td>
<td>pc, [pc, #-0x10]</td>
</tr>
</tbody>
</table>

**ITR**

Binary Instruction

MOV x4, #0

0xB20003E4

In debug state, write binary instruction to ITR for execution.
Instruction Execution in Debug State

**TARGET**
(Debug State)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register 1</th>
<th>Register 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>x3, #3</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>x4, #4</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>x0, x3</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>x1, x4</td>
<td></td>
</tr>
<tr>
<td>LDR</td>
<td>pc, [pc, #-0x10]</td>
<td></td>
</tr>
</tbody>
</table>

...  

**ITR**

- 0xB20003E4
  - MOV x4, #0
  - 0xB20003E4

In debug state, write binary instruction to ITR for execution.