University of Illinois Urbana-Champaign (UIUC)
Advisor
Professor Kirill Levchenko, PhD

Team Leads
Minh Duong, Jake Mayer, Emma Hartman, Hassam Uddin

Team Members
Juniper Peng, Timothy Fong, Krish Asher, Adarsh Krishnan, Liam Ramsey, Yash Gupta, Suchit Bapatla, Akhil Bharanidhar, Zhaofeng Cao, Ishaan Chamoli, Tianhao Chen, Kyle Chung, Vasunandan Dar, Jiming Ding, Sanay Doshi, Shivaditya Gohil, Seth Gore, Zexi Huang, George Huebner, Haruto Iguchi, Parithimaal Karmehan, Jasmeendar Kochhar, Arjun Kulkarni, Julia Li, Jingdi Liu, Richard Liu, Theodore Ng, Stefan Ninic, Henry Qiu, Neil Rayu, Ram Reddy, Sam Ruggerio, Naavya Shetty, Arpan Swaroop, Raghav Tirumale, Yaoyu Wu
Design Phase
Design Methodology

– No code until protocol was fully created
  – This gave us time to properly design our implementation to ensure that there were no fundamental vulnerabilities
  – After the protocol is created, writing code is simply following the protocol – it also allows team members to easily get into writing code

– Sub-teams for each area that we wanted to focus in:
  – Pre-boot (List, Replace, Attest)
  – Secure Communications (Boot, HIDE protocol)
  – Build (Post-Boot, secrets/generation, Rust library)
  – Attack (research HW attacks, build exploits for insecure example)
<table>
<thead>
<tr>
<th>Title</th>
<th>Team</th>
<th>Status</th>
<th>End date</th>
<th>Labels</th>
<th>Milestone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Boot/Attestation Subteam</td>
<td>Pre-Boot/Attest Subteam</td>
<td>Done</td>
<td>Mar 3, 2024</td>
<td>FR - List Components</td>
<td>Begin Testing</td>
</tr>
<tr>
<td>Implement List Components</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implement Attestation</td>
<td></td>
<td></td>
<td></td>
<td>FR - Attestiation</td>
<td></td>
</tr>
<tr>
<td>Implement Replacement</td>
<td></td>
<td></td>
<td></td>
<td>FR - Replace Components</td>
<td></td>
</tr>
<tr>
<td>Initial protocol for List Components</td>
<td></td>
<td></td>
<td>Feb 10, 2024</td>
<td>documentation</td>
<td></td>
</tr>
<tr>
<td>Initial protocol for Attestation</td>
<td></td>
<td></td>
<td></td>
<td>FR - Attest</td>
<td>Begin Implementation</td>
</tr>
<tr>
<td>Initial protocol for Replacement</td>
<td></td>
<td></td>
<td></td>
<td>FR - Replacement</td>
<td></td>
</tr>
<tr>
<td>Comms Subteam</td>
<td>Comms Subteam</td>
<td>Done</td>
<td>Mar 3, 2024</td>
<td>FR - Boot Verification</td>
<td>Begin Testing</td>
</tr>
<tr>
<td>Implement Boot Verification protocol using HIDE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implement HIDE protocol</td>
<td></td>
<td></td>
<td></td>
<td>FR - Secure Comms</td>
<td>Begin Implementation</td>
</tr>
<tr>
<td>Initial protocol for HIDE secure communications layer</td>
<td></td>
<td></td>
<td>Feb 10, 2024</td>
<td>documentation</td>
<td></td>
</tr>
<tr>
<td>Initial protocol for Boot Verification</td>
<td></td>
<td></td>
<td></td>
<td>FR - Boot</td>
<td>Begin Implementation</td>
</tr>
<tr>
<td>Build Subteam</td>
<td>Build Subteam</td>
<td>Done</td>
<td>Mar 5, 2024</td>
<td>Attack</td>
<td>Handoff</td>
</tr>
<tr>
<td>Implement fault-injection resistant patterns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add secure send/receive C interfaces for POST_BOOT code</td>
<td></td>
<td></td>
<td>Mar 4, 2024</td>
<td>FR - Build System</td>
<td>Begin Testing</td>
</tr>
<tr>
<td>Add mnemonic and literal support to POST_BOOT code</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Design Overview

- Rust (memory-safe)
- HIDE protocol with Ascon-128 cryptographic scheme
  - Transforms message into three-way challenge response handshake
  - Prevents forging/replay attacks
- Delays
  - Constant delays prevent brute-force attacks
  - Random delays deter hardware attacks (fault injection)
HIDE Protocol

- Sending of message initiates HIDE Protocol
- Sender of message sends message request to begin communication
- Receiver sends random, encrypted challenge nonce
- Sender must decrypt and solve challenge
- Challenge response is encrypted and sent with message
- Receiver validates response before executing message
- Protocol ensures messages are encrypted, authenticated, verified
HIDE Protocol

AP

Request to Send Message

Decrypt and Solve Challenge Nonce

Component

Encrypt Random Challenge Nonce

Validate Response and Execute Message
Improvements to Design

- Use key-derivation functions
  - Prevents key reuse and possible cryptography attacks
- Improve anti-glitching
  - Adding more random delays
- Reduce impact from exploits
  - Component does not need to store flags in plaintext since the AP is the one that presents all boot messages or Attestation Data
- Implement memory protection unit (MPU)
Attack Phase
Attack #0: Simple I²C Component

- Improper handling of I²C hardware conditions allows for a buffer overrun and arbitrary code execution
- This critical vulnerability affects the Component specifically and allows for **complete compromise** of the Component
- We developed an exploit for this vulnerability to extract Component flags and carry out attacks against the AP as well
- 85% of teams were vulnerable to this exploit since the bug originated from the reference implementation
Attack #1

Attacking boot process with a compromised supply chain
Here is a typical device configuration!
Component A
0x11111111

Component B
0x22222222

Component B becomes damaged!
An authorized technician orders a new Component...
... and runs the replacement routine on the AP.
The device should be able to boot!
Attacker’s Goal: Get the AP to boot despite an unauthentic Component being installed.
Simple Solution: Adding a validation step with a shared secret key prevents trivial attacks at booting.
Using the I²C Component exploit, we can extract secrets!
Using the I²C Component exploit, we can extract secrets!
Better Solution: Adding a validation step with unique secret keys and host signatures.
Better Solution: Even with the I²C exploit, the host signature is invalid because of the Component ID mismatch.

\[ \text{Comp}_A \rightarrow Sig(\text{Comp}_A + 0x11111111) \]

\[ \text{AP} \rightarrow 0x11111111, 0x33333333 \]

\[ \text{Evil Component} \rightarrow 0x33333333 \]
if validate_token():
    CompID_New <- input()
CompID_Old <- input()
for i in num_components:
    if CompID_Old == component_ids[i]:
        component_ids[i] <- CompID_New
    return Success
return Failure ("CompID_Old not found")
return Failure ("Incorrect Token")

This code does not check if CompID_New is already provisioned!
In other words: an AP can have two provisioned Components with same ID!
Attack #1: Exploiting Replace Code

– New problem: two same Component IDs means that they share the same I²C address, which will cause bus errors
  – Attacker’s fix: use the simple I²C exploit to disable Component A
  – This is done by changing Component A’s I²C address to 0x00
  – Our Evil Component will handle both validate and boot requests from the AP
Use the I²C Component exploit to extract the unique secret key and signature, then disable Component A!
Use the I²C Component exploit to extract the unique secret key and signature, then disable Component A!
The attacker has successfully tricked the AP into booting!

**Component A**

\[ K_{Comp \ A} \]

\[ Sig(K_{Comp \ A} + 0x11111111) \]

**AP**

\[ 0x11111111 \]

\[ 0x11111111 \]

**Evil Component**

\[ 0x11111111 \]

\[ K_{Comp \ A} \]

\[ Sig(K_{Comp \ A} + 0x11111111) \]
Attack #2

Hardware attacks against the MAX78000FTHR board
Attack #2: Hardware Attack

**Goal:** Skip an executing instruction with fault injection by a voltage glitch

**Method:**
- Connect ChipWhisperer to the voltage line MCU Arm core
- Pull the voltage to ground while the core is executing an instruction

**Challenges:**
- Pulling voltage to ground for too long will cause a power reset
- Requires precise timing to pinpoint instruction to skip
- Capacitors provide limited power even though we pull to ground
This year, we invested in a ChipWhisperer-Lite and an oscilloscope!

The oscilloscope demonstrates a voltage glitch attack, briefly bringing power to ground.
Reliable voltage glitching requires the removal of some capacitors.
Our test board setup for voltage glitch attacks!
Attack #2: Summary

- Implication: If you could skip any single instruction in the code, what instruction would you skip?
  - Most teams did not implement protections against this scenario
  - Voltage glitching allows bypassing security checks altogether

- Mitigations:
  - Adding truly random delays
    - If a delay is random, the attacker doesn’t know when to apply the glitch
  - Multiple if statements and condition guards
    - It’s difficult to skip multiple instructions in a row or time sequential skips
Other Attacks

- Attestation PIN brute force
  - Only 6 hexadecimal digits (000000 – ffffff)!
  - No delays means this can be cracked quickly
- Bad schemes + secrets sent over the wire to authenticate
  - Record these secrets with a logic analyzer, build new device with secrets
- For Damaged Boot, use the same working Component to respond to validation/boot requests for a broken Component
  - Requires a MITM device to translate the I²C addresses
Thank you! Any questions?