System-level threats:

Dangerous assumptions in modern Product Security

Cristofaro Mune
(c.mune@pulse-sec.com)
@pulsoid
• **Cristofaro Mune (@pulsoid)**
  - *Product Security Consultant/Researcher*
  - *Keywords: TEE, IoT, Devices, HW, Fault Injection, Exploitation,...*

• *(Public) Research:*
  - [2009] - “Hijacking Mobile Data connections”
  - [2010] - AP Exploitation
  - [2015] - Breaking WB cryptography
  - [2017]:
    - TEEs secure initialization
    - IoT exploitation
    - Linux Privilege Escalation with Fault Injection
Devices and markets

Networking

Physical Security

Mobile devices

Smart homes

Mobile Payments

Automotive
Who “owns” a device?

It’s my wallet, I am CEO of BitFi
iPhone X components and suppliers (some)

Accelerometer
Bosch & IvenSense

Baseband Processor
Qualcomm

Batteries
Samsung & Shenzhen Desay Battery Technology

Chips
Cirrus Logic, Samsung, TSMC, MicroSemi, Broadcom & NXP

DRAM
TSMC & SK Hynics

eCompass
Alps Electric

*from capitalistlad.wordpress.org
Example: Apple suppliers 2018

Where Apple suppliers are headquartered

(number of companies)

- Singapore 5
- Hong Kong
- South Korea
- EU, Switzerland, UK
- China 27
- U.S. 39
- Japan 43
- Taiwan 51
- Saudi Arabia 1

Total 200

Source: Apple’s list of suppliers for 2018
Who owns a device?

“Nobody.
Really, nobody FULLY owns a device.”
Device == ecosystem effort → PRODUCT

- Each component in the chain has its own:
  - Threat models, use cases and assumptions
“Assumptions at boundaries may cause system-level vulnerabilities”
“A few recurring classes (out of my experience)...”
Completeness
“Component-level information is **sufficient** for characterizing security impacts on the final system”*

* Applies both to design and security assessments
Example: SW security assessment

- A `strcmp()` classic implementation
- Taken from open-source project uClibc-ng
- Aimed at embedded devices
  - Latest available version: v1.0.31
- Full source code available
Any vulnerability?

```c
int strcmp (const char *p1, const char *p2)
{
    register const unsigned char *s1 = (const unsigned char *) p1;
    register const unsigned char *s2 = (const unsigned char *) p2;
    unsigned reg_char c1, c2;

    do
    {
        c1 = (unsigned char) *s1++;
        c2 = (unsigned char) *s2++;
        if (c1 == '\0')
            return c1 - c2;
    }
    while (c1 == c2);

    return c1 - c2;
}
```

https://elixir.bootlin.com/uclibc-ng/v1.0.31/source/libc/string/generic/strcmp.c
Threat model (TM): only SW attacks

Not vulnerable
int strcmp (const char *p1, const char *p2)
{
    register const unsigned char *s1 = (const unsigned char *) p1;
    register const unsigned char *s2 = (const unsigned char *) p2;
    unsigned reg_char c1, c2;

    do
    {
        c1 = (unsigned char) *s1++;
        c2 = (unsigned char) *s2++;
        if (c1 == '\0')
            return c1 - c2;
    }
    while (c1 == c2);

    return c1 - c2;
}
What about now?

- If attacker is able:
  - to access a measurable channel
  - to sample with sufficient precision

- If compared quantity is a security asset:
  - e.g. A password, a MAC,...

Vulnerable

Depends on the system.
OK…which Threat Model applies?

**SW Component**

- **strcmp()**

**Threat Models**

- **SW attacks only**
  - **Not Vulnerable**

- **SW + Timing**
  - **Vulnerable**
Secure component $\rightarrow$ Vulnerable system

$TM(\text{Component}) = SW \text{ only}$

$TM(\text{System}) = SW + \text{Timing attacks}$
Remarks: Vulnerability

- Component level information insufficient
- Vulnerability cannot be assessed at component level
  - Depends on final system Threat Model
- Code review does not solve the problem
  - Unless a Threat Model is specified
- Only “next stage” integrators are able to assess vulnerability:
  - E.g: OEMs at the final product integration
Exploitability?

- String comparison:
  - “Impractical in the vast majority of cases” 2015 – Morgan & Morgan
  - Remote servers with fast CPUs

- But... *IoT systems are much slower!*

that differ in the first character vs. strings that differ only at the 10th character. This indicates that timing attacks on regular string comparison have to be assumed feasible for any embedded system.*

* 2014 – Mayer, Sandin – “Time Trial”
Demo

- Target: Arduino UNO
  - Clock speed: 16 Mhz
  - Media: Ethernet 100Mbit

- Numeric PIN: 8 digits

```
Candidate password: 31337890
Verifying: 31337890
Success!
Verification successful!
Password found!!!
[+] Attack Completed
Total requests: 68002
Bruteforce complexity: 100000000
Ratio: 0.07%
```
Remarks: Exploitability

- Exploitability cannot be assessed at component level
  - Even with full source code

- Impact *depends on final system*:
  - E.g. Clock speed, measurable channels,...

*Full impacts can be established only at final product integration*
Assumption of Completeness

“Component-level information is \textbf{sufficient} for characterizing component security impact on the final system,”* 

* Applies both to design and security assessments
Always ask for...

- Threat Model

- Existing development security processes:
  - SDLC, Design security reviews, Source code audits, Product Penetration Testing

- Existing in-field security processes:
  - Security fixes, Security maintenance (e.g. Firmware update for 5 yrs),...

- Security evaluation reports

“When buying components, you are also buying risks”
A good example: ARM TrustZone

—— Note ————

TrustZone technology is designed to provide a hardware-enforced logical separation between security components and the rest of the SoC infrastructure.

Lab attacks are outside of the scope of the protection provided TrustZone technology, although a SoC using TrustZone can be used in conjunction with an ARM SecurCore® smartcard if protection against physical attacks is needed for some assets.

Threat Model specified in documentation
Correctness
Assumption of Correctness

“The system will always behave as intended. Correctly executing as specified.”*

* Applies both to design and security assessments
Fault attacks

“Introducing faults in a target to alter its intended behavior.”

- A controlled environmental change leads to altered behavior in a target
- Leverage a vulnerability in a hardware subsystem
Assumption: Expensive

Chipwhisperer Lite

~$250

Microcontroller

< $30

FPGA

~$99

- Also “Cheapscate: Attacking IoT with less than $60” - Raphael Boix Carpi

VCC glitching cost ($): < 300
Other assumptions

• Physical access is required:
  - *Wrong.* SW-initiated FI attacks can be performed remotely:
    • Rowhammer and CLKSCREW

• No security decision point → Nothing to attack:
  - *Wrong.* New fault models allow for direct code execution.

• We have countermeasures in SW:
  - *Wrong.* New fault models can completely bypass SW FI countermeasures

• Our secure boot is encrypted. You need a key anyway:
  - *Wrong.*
  - Wait a minute. 😊
Secure Boot

```c
int load_exec_next_boot_stage(){
    uint32_t stage_addr=0xd0000000;
    uint32_t sig_addr=0xc0000000;

    // Copy stage from media to memory
    load_next_stage(stage_addr);

    // Copy signature to memory
    load_signature(sig_addr);

    // Verify signature
    if(!verify_signature(stage_addr,sig_addr)) {
        while(1);
    } else {
        // Execute next stage
        exec_stage(stage_addr);
    }
}
**Textbook attack**

```
int load_exec_next_boot_stage()
{
    uint32_t stage_addr=0xd0000000;
    uint32_t sig_addr=0xc0000000;
    // Copy stage from media to memory
    load_next_stage(stage_addr);
    // Copy signature to memory
    load_signature(sig_addr);
    // Verify signature
    if(!verify_signature(stage_addr,sig_addr)) {
        while(1);
    } else {
        //Execute next stage
        exec_stage(stage_addr);
    }
}
```

“Instruction skipping”
int load_exec_next_boot_stage()
{
    AES ctx;
    uint32_t stage_addr=0xd0000000;
    uint32_t sig_addr=0xc0000000;

    init_AES_engine(&ctx, key_id);

    // Copy stage from media to memory
    load_encrypted_next_stage(stage_addr);

    // Copy signature to memory
    load_signature(sig_addr);

    // Stage is encrypted. Decrypt first
    decrypt_stage(&ctx, stage_addr);

    // Verify signature over stage plaintext
    if(!verify_signature(stage_addr,sig_addr)) {
        while(1);
    } else {

        //Execute next stage
        exec_stage(stage_addr);
    }
```c
int load_exec_next_boot_stage()
{
    AES ctx;
    uint32_t stage_addr=0xd0000000;
    uint32_t sig_addr=0xc0000000;

    init_AES_engine(&ctx, key_id);

    // Copy stage from media to memory
    load_encrypted_next_stage(stage_addr);

    // Copy signature to memory
    load_signature(sig_addr);

    // Stage is encrypted. Decrypt first.
    decrypt_stage(&ctx, stage_addr);

    // Verify signature over stage plaintext
    if(!verify_signature(stage_addr,sig_addr)) {
        while(1);
    } else {
        //Execute next stage
        exec_stage(stage_addr);
    }
}
```
Creating execution primitives...out of thin air

- ARM32 has an interesting ISA
- Program Counter (PC) is directly accessible

**Valid ARM instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary 1</th>
<th>Binary 2</th>
<th>Binary 3</th>
<th>Binary 4</th>
<th>Binary 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV r7, r1</td>
<td>00000000</td>
<td>01110000</td>
<td>10100000</td>
<td>11100001</td>
<td></td>
</tr>
<tr>
<td>EOR r0, r1</td>
<td>00000000</td>
<td>00000000</td>
<td>00100000</td>
<td>11100000</td>
<td></td>
</tr>
<tr>
<td>LDR r0, [r1]</td>
<td>00000000</td>
<td>00000000</td>
<td>10010001</td>
<td>11100101</td>
<td></td>
</tr>
<tr>
<td>LDMIA r0, {r1}</td>
<td>00000010</td>
<td>00000000</td>
<td>10010000</td>
<td>11101000</td>
<td></td>
</tr>
</tbody>
</table>

**Corrupted ARM instructions may directly set PC**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary 1</th>
<th>Binary 2</th>
<th>Binary 3</th>
<th>Binary 4</th>
<th>Binary 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV pc, r1</td>
<td>00000000</td>
<td>11110000</td>
<td>10100000</td>
<td>11100001</td>
<td></td>
</tr>
<tr>
<td>EOR pc, r1</td>
<td>00000000</td>
<td>11111000</td>
<td>00101111</td>
<td>11100000</td>
<td></td>
</tr>
<tr>
<td>LDR pc, [r1]</td>
<td>00000000</td>
<td>11111000</td>
<td>10010001</td>
<td>11100101</td>
<td></td>
</tr>
<tr>
<td>LDMIA r0, {r1, pc}</td>
<td>00000010</td>
<td>10000000</td>
<td>10010000</td>
<td>11101000</td>
<td></td>
</tr>
</tbody>
</table>

**Attack variations (SP-control) also affect other architectures**

*also see [FDTC 2016]: Timmers, Spruyt, Witteman*
ROM code: secure boot + encryption

```c
int load_exec_next_boot_stage(){
    AES ctx;
    uint32_t stage_addr=0xd0000000;
    uint32_t sig_addr=0xc0000000;
    init_AES_engine(&ctx, key_id);
    // copy stage from media to memory
    load_encrypted_next_stage(stage_addr);

    // Copy signature to memory
    load_signature(sig_addr);

    // Stage is encrypted. Decrypt first.
    decrypt_stage(&ctx, stage_addr);

    // Verify signature over stage plaintext
    if(!verify_signature(stage_addr,sig_addr)) {
        while(1);
    } else {
        //Execute next stage
        exec_stage(stage_addr);
    }
}
```

Glitch while loading pointers $\Rightarrow$ PC set to pointers $\Rightarrow$ Code exec at stage_addr*

Never executed

*also see [FDTC 2016]: Timmers, Spruyl, Witteman
Remarks

- FI vulnerabilities located at physical level
- *Cannot be identified via HW or SW code review*
- Only testing the *real implementation* can provide indications of vulnerability
- Testing better performed right after silicon integration:
  - e.g. SoC Manufacturer
- Vulnerable HW may affect entire classes of devices
Assumption of Correctness

“The system will always behave as intended. Correctly executing as specified.”*

* Applies both to design and security assessments
FI in your threat model?

- Ask your SoC manufacturer for **Security evaluation reports**: 
- Do **NOT** rely only on HW/SW audits only
  - Only testing can uncover FI vulnerabilities

“Either you have countermeasures, or…
...you are painfully, **desperately vulnerable**”
Consistency
Assumption of Consistency

“The entire system has only one threat model and protects the same assets.”
Assumption

One single Threat Model

IP

Manufacturer(s) \{1, \ldots\}

SoC

Manufacturer(s) \{2, \ldots\}

SOC + Low-lvl SW

Manufacturer(s) \{3, \ldots\}

Final Product

OEM

Feasible?
A typical Security Model...

Kernel space

Kernel

User space

Users process

Users process

Users process

Users process
...with a TEE

ARM TrustZone SoC (TEE HW)
No intention to protect REE...
REE

Kernel memory overwrite!

Pass Kernel address for output

TEE has no way of distinguishing REE memory

Users process

Linux Kernel

Driver

TEE call (SMC)

BlackHat 2015
Remarks

- Different security models may be present at the same time
- May not be aware of each others
- May be leveraged AGAINST each others
Assumption of Consistency

“The entire system has only one threat model and protects the same assets.”
Recommendations

• Understand components’ Security Model
  - Does it fit with your Security Model?

• Evaluate within system Threat Model

“There may be no consistency, across components and subsystems.”

...expect none...
Isolation
Assumption of Isolation

“There is only sub-system. Mine”
A simplistic model
Reality: other IPs can access DDR
Example: Broadpwn

WiFi SoC

Main SoC

Kernel

DDR

User process

PCIe
Remarks

- SoCs → “execution units”
- Other SoCs may have access to Main SoC DDR
- May not be aware of each others’ Security Models:
  - Kernel vs userspace in SoC1 → Plain addressable memory for SoC2
What could have been done?

- SW review, testing
  (WiFi SoC SW)

- SMMU design
  (Main SoC HW)

- Main SoC

- WiFi SoC

- SMMU configuration
  (Main SoC: BL2/TEE)

- Kernel

- DDR

- User process
And we are missing...

- Other execution units:
  - Audio/Video Processors
  - GPUs
  - Power Modules
  - ...

- DMA-capable IPs:
  - USB
  - Firewire
  - PCMCIA
  - PCIe
  - ...

- Other Bus masters IPs

There can be hundreds...
Assumption of Isolation

“There is only sub-system. Mine”
Reality

My God, It's full of stars....
Conclusions
Reflections

- **System Security:**
  - Threat Models may differ between Components
  - \( \text{Security(System)} = \sum_i \text{Security(Component}_i) \)

- **Security Evaluation:**
  - Context and **system-level information required** for assessment
  - Code reviews cannot identify all vulnerabilities (e.g. FI)

- **Design:**
  - HW and SW must cooperate. **Across the whole system.**
    - Regardless of Manufacturer, Department, Teams boundaries
  - Protect **FROM** other sub-systems

“Unchecked assumptions at boundaries can be fatal”
Recommendations

• **Establish a system-level threat model:**
  - Apply and verify consistency everywhere

• For every HW/SW component:
  - *Gain understanding of use case and threat models*
  - *Test and review thoroughly*

• Security assessment and testing MUST consider:
  - *System Threat Model*

• For every 3rd party component ask:
  1. *Threat Model*
  2. *Security practices and processes*
  3. *A security evaluation report*
    - *You are already paying for it.*
Cristofaro Mune

Product Security Consultant

c.mune@pulse-sec.com