Advancing FI attacks: Fault Models opportunities

Cristofaro Mune
(c.mune@pulse-sec.com)
@pulsoid
Password check
A simple password check

Can we bypass this?

What do you think?
A better view
Traditional fault models

**Control flow corruption**
by skipping instructions

**Data corruption**
by flipping bits
Textbook attack

- If the check can be skipped...
- Execution falls through the right case.
  - Regardless of the password.
- SUCCESS!

“Instruction skipping”
Case study:
Secure Boot
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);
    }
}```
Code modified? → Hang

```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);
    }
}
```
Executing our code?

```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);
    }
}
```

How?
Textbook attack

```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);
    }
}
```

“Instruction skipping”
Secure Boot with countermeasures

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

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

    // Copy signature to memory
    load_signature(sig_addr);

    random_delay();
    // Verify signature (1)
    if (!verify_signature(stage_addr, sig_addr)) {
        while (1);
    }

    random_delay();
    // Verify signature (2)
    if (!verify_signature(stage_addr, sig_addr)) {
        while (1);
    }

    exec_stage(stage_addr);
}
```
But still doable!

```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);

    random_delay();
    // Verify signature (1)
    if(!verify_signature(stage_addr,sig_addr)) {
        while(1);
    }

    random_delay();
    // Verify signature (2)
    if(!verify_signature(stage_addr,sig_addr)) {
        while(1);
    }

    exec_stage(stage_addr);
}
```
Case study:
Encrypted Secure Boot
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);
    }
}
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);
    }
}
Security Certification

• Security Lab report:
  - “We have not been able to bypass Secure Boot”
  - “SCA attack needed for extracting encryption key"

• CC Attack rating goes stellar
  - "If attack possible at all"

• Product is SECURE
Reflections
• **Convenience fault model**
  - Widely used for characterizing effects on SW execution

• Has *limitations*:
  - Simplistic:
    • focused on conditionals, single-point decisions.
  - Not realistic:
    • Research shows instructions are most likely “corrupted”. Not skipped.

• **Insufficient for precise modeling of attacks aimed at SW execution.**
A different fault model

A generic one: "instruction corruption"*

**Single-bit (MIPS)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $t1, $t1, 8</td>
<td>001000010010100100000000000001000</td>
</tr>
<tr>
<td>addi $t1, $t1, 0</td>
<td>001000010010100100000000000000000</td>
</tr>
</tbody>
</table>

**Multi-bit (ARM)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldr w1, [sp, #0x8]</td>
<td>10111001010000000000001011111100001</td>
</tr>
<tr>
<td>str w7, [sp, #0x20]</td>
<td>1011100100000000001001111100111</td>
</tr>
</tbody>
</table>

Remarks

- Limited control over which bit(s) will be corrupted
- Also includes other fault models as sub-cases (e.g. instruction skipping)

*[FTDC 2016]: Spruyt, Timmers, Witteman*
Controlling PC (or SP)

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

**Valid ARM instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV r7,r1</td>
<td>00000001 01110000 10100000 11100001</td>
</tr>
<tr>
<td>EOR r0,r1</td>
<td>00000001 00000000 00100000 11100000</td>
</tr>
<tr>
<td>LDR r0,[r1]</td>
<td>00000000 00000000 10010001 11100101</td>
</tr>
<tr>
<td>LDMIA r0,{r1}</td>
<td>00000010 00000000 10010000 11101000</td>
</tr>
</tbody>
</table>

**Corrupted ARM instructions may *directly set* PC**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV pc,r1</td>
<td>00000001 11110000 10100000 11100001</td>
</tr>
<tr>
<td>EOR pc,r1</td>
<td>00000001 1111 0000 00101111 11100000</td>
</tr>
<tr>
<td>LDR pc,[r1]</td>
<td>00000000 11110000 10010001 11100101</td>
</tr>
<tr>
<td>LDMIA r0,{r1, pc}</td>
<td>00000010 10000000 10010000 11101000</td>
</tr>
</tbody>
</table>

*Attack variations (SP-control) also affect other architectures*
Execution primitives...out of thin air

- **ANY memory read** can be redirected to PC (or SP)
  - Hence, **ANY memcpy()**
- **PC** (or SP) **immediately assigned** with content from memory
  - Following SW checks may not be executed

- **A new target for FI:**
  - Security checks
  - Crypto algorithms
  - ...

**Code Execution**
Example: Secure Boot + countermeasures

```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);

    random_delay()
    // Verify signature (1)
    if(!verify_signature(stage_addr,sig_addr)) { 
        while(1);
    }

    random_delay()
    // Verify signature (2)
    if(!verify_signature(stage_addr,sig_addr)) { 
        while(1);
    }

    exec_stage(stage_addr);
}
```

*also see [FDTC 2016]: Timmers, Spruyt, Witteman*
Analysis

• Assumptions:
  - PC directly addressable (e.g. ARM32)
  - Attackers knows code location
  - Destination memory writable and executable:

• All FI SW countermeasures ineffective
  - Fault → Code execution transition happens in HW
    • More nuanced for code execution achieved via SP-control
A SW attacker’s dream...

• Like a SW exploit. *Without a SW vulnerability.*
• *ALL SW exploitation techniques fully applicable*
  - e.g. ROP, JOP, COP,… for SP-control

• *SW exploitation mitigations can be effective*
Defeating

Encrypted Secure Boot
Attack preparation

- We can apply the same fault model:
  - Extends applicability of *Timmers, Spruyt, Witteman* technique (see FTDC 2016 presentation)
  - Also see *Timmers, Spruyt* @BlackHat 2016

- Strategy:
  - Redirect control flow via PC hijacking → Code execution
    - *In the running context!*
  - Signature check not executed → Secure Boot bypass
  - Decryption not executed → Plaintext code execution

- Flash boot stage preparation:
  - Execution payload
  - “Sled” of PC target address
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);
    }
}
Analysis

• **Signature verification not performed**
  - Secure boot defeated

• **Decryption not performed**
  - Plaintext code execution

• Code execution achieved in **verifying stage context**

• **ROM-level code execution**
Countermeasures
• Hardware FI countermeasures *fully applicable*
  - Detect glitch injection or fault generation

• *FI SW countermeasures likely not executed*
  - A successful attack hijacks control flow immediately

• *Localized software FI countermeasures are insufficient*
  - Any instruction is a potential target for corruption
SW Exploit mitigations

- Fully applicable.
- **Relevant:** Limiting usage of a hijacked control flow
  - DEP/NX
  - ASLR
  - CFI
  - ...
- **Irrelevant:** Preventing control flow hijacking:
  - Stack cookies
  - SEHOP
  - ...

SW Exploit mitigations
Recommendations

• Use **FI HW countermeasures** for *prevention*
  - Applicable regardless of fault model

• **FI SW countermeasures** *can only mitigate “classical attacks”*

• **Adopt modern SW security paradigms:**
  - *SW exploit mitigations*
  - Defense in depth
  - Secure SDLC
  - ... 

• **Learn from:**
  - *SW attackers*
  - *SW and Mobile security industry*
Final thoughts
A new attack

- Technique for bypassing encrypted secure boot:
  - One single fault
  - ROM-level code execution
  - Arbitrary plaintext payload
  - No encryption key needed
Advancing FI

• Different fault models can yield \textit{dramatically different results}.

• Simplistic fault modeling can be \textit{dangerous}:
  - High impact attacks may go undetected for decades.

• \textit{SW execution integrity is critical nowadays}:
  - Must be fully in scope for FI.

• Must also fall scope for modern HW security industry, in general.
HW & SW security

- **HW and SW attacks more intertwined:**
  - Also see *micro-architectural attacks*
  - Unexpected implications

- **HW security industry needs SW security experts**
  - And vice versa

- We need more “in between” expertise:
  - Single domain expertise not sufficient anymore

- Attackers’ and engineers’ perspectives need blending

- **Holystic, system-level view needed**
Finally...

- If your HW and SW security engineering teams...
- do not talk to each other...

You are likely doing it wrong.
Contacts

Cristofaro Mune
Product Security Consultant

c.mune@pulse-sec.com