FDTC 2017 Fault Diagnosis and Tolerance in Cryptography

Escalating Privileges in Linux using Fault Injection

Niek Timmers
timmers@riscure.com
(@tieknimmers)

Cristofaro Mune
c.mune@pulse-sec.com
(@pulsoid)

September 25, 2017

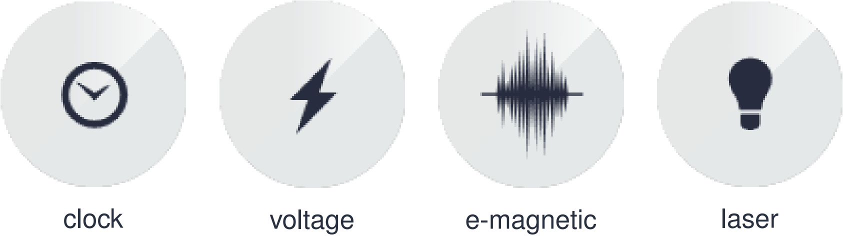
Fault Injection – A definition...

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

```
...
if( key_is_correct ) <-- Glitch here!
{
  open_door();
}
else
{
  keep_door_closed();
}
...
```

How can we introduce these faults?

Fault injection techniques



We used Voltage Fault Injection for all experiments!

Fault injection techniques

clock voltage e-magnetic laser

We used Voltage Fault Injection for all experiments!

Fault injection fault model

*Let's keep it simple: **instruction corruption***

Single-bit (MIPS)

```
addi $t1, $t1, 8    00100001001010010000000000001000
addi $t1, $t1, 0    00100001001010010000000000000000
```

Multi-bit (ARM)

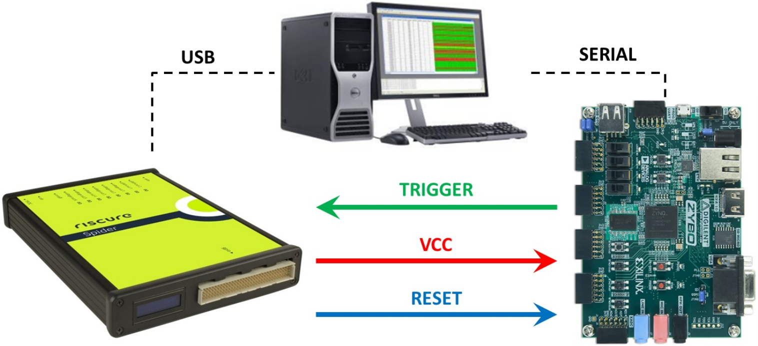
```
ldr w1, [sp, #0x8]  10111001010000000000101111100001
str w7, [sp, #0x20] 10111001000000000100011111100111
```

Remarks

- Limited control over which bit(s) will be corrupted
- May or may not be the true fault model
- Includes other fault models (e.g. instruction skipping)

Let’s inject faults!

Fault injection setup

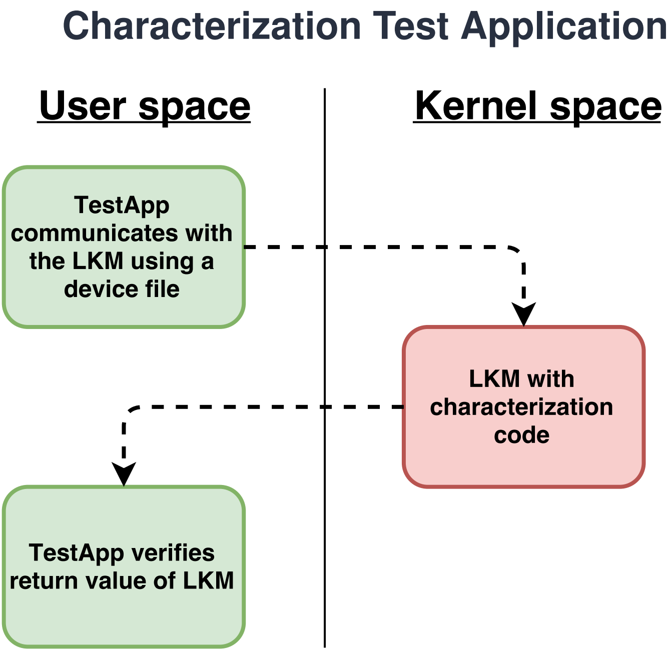


Target

- Fast and feature rich System-on-Chip (SoC)
- ARM Cortex-A9 (32-bit)
- Ubuntu 14.04 LTS (fully patched)

Characterization

- Determine if the target is vulnerable to fault injection
- Determine if the fault injection setup is effective
- Estimate required fault injection parameters for an attack
- An *open* target is required; but not required for an attack


Characterization Test Application
User space
Kernel space
TestApp communicates with the LKM using a device file
LKM with characterization code
TestApp verifies return value of LKM

Characterization – Alter a loop

```
. . .
set_trigger(1);

for(i = 0; i < 10000; i++) {  // glitch here
    j++;                      // glitch here
}                             // glitch here

set_trigger(0);
. . .
```

Remarks

- Implemented in a Linux Kernel Module (LKM)
- Successful glitches are **not that** time dependent

Characterization – Possible responses

Expected: 'glitch is too soft'

counter = 00010000

Mute/Reset: 'glitch is too hard'

counter =

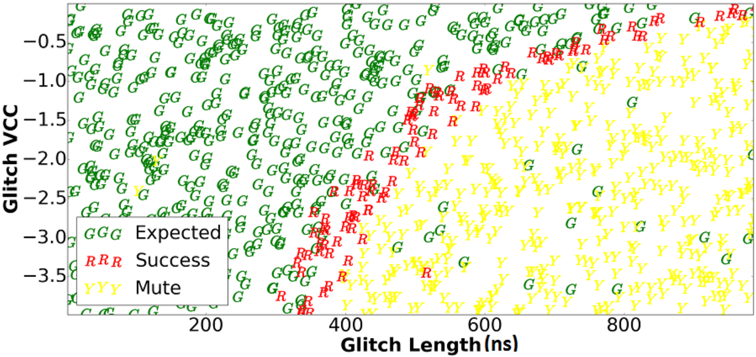
Success: 'glitch is exactly right'

counter = 00009999

counter = 00010015

counter = 00008687

Characterization – Alter a loop



Remarks

- We took 16428 experiments in 65 hours
- We randomize the **Glitch VCC Glitch Length Glitch Delay**
- We can fix either the **Glitch VCC** or the **Glitch Length**

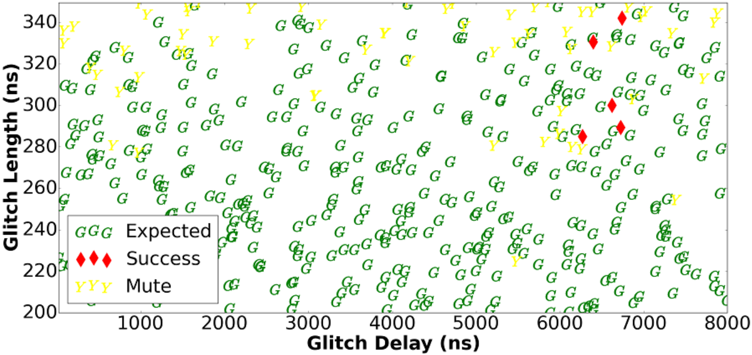
Characterization – Bypassing a check

```
. . .
set_trigger(1);

if(cmd.cmdid < 0 || cmd.cmdid > 10) {
  return -1;
}

if(cmd.length > 0x100) {     // glitch here
  return -1;                 // glitch here
}                            // glitch here

set_trigger(0);
. . .
```

Remarks

- Implemented in a Linux Kernel Module (LKM)
- Successful glitches **are** time dependent

Characterization – Bypassing a check



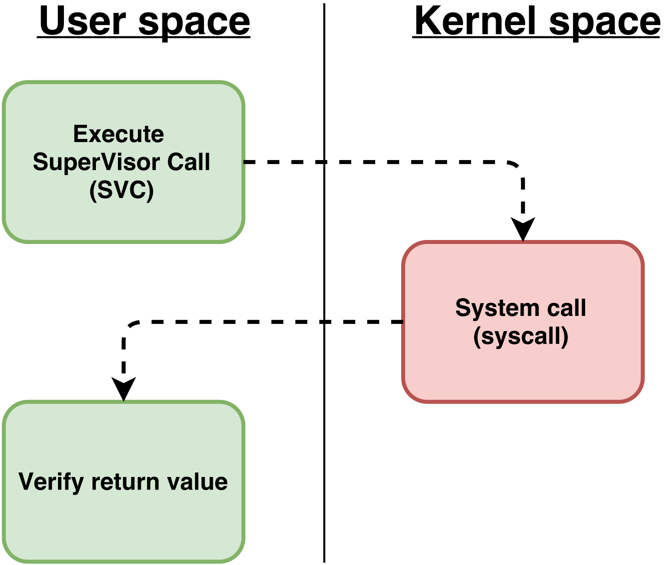
Remarks

- We took 16315 experiments in 19 hours
- The success rate between 6.2 µs and 6.8 µs is: 0.41%
- The check is bypassed every 15 minutes !

Let's attack Linux!

Relevant when vulnerabilities are not known!

Attacking Linux



Opening /dev/mem – Description

(1) Open */dev/mem* using open syscall

(2) Bypass check performed by Linux kernel using a glitch

(3) Map arbitrary address in physical address space

Opening /dev/mem – Code

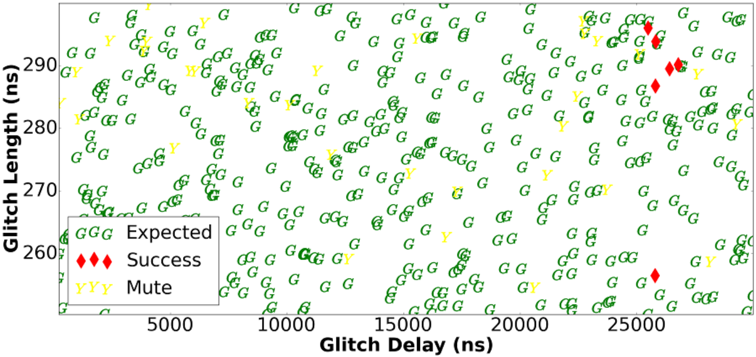
Algorithm 1 Open /dev/mem

$r1 \leftarrow 2$
$r0 \leftarrow "/dev/mem"$
$r7 \leftarrow 0x5$
$svc\ \#0$
if $r0 == 3$ **then**
    $address \leftarrow mmap(...)$
    $printf(*address)$
end if

Remarks

- Implemented using ARM assembly
- Linux syscall: sys_open (0x5)

Opening /dev/mem – Results



Remarks

- We took 22118 experiments in 17 hours
- The success rate between 25.5 µs and 26.8 µs is: 0.53%
- The Linux kernel is compromised every 10 minutes !

Privilege escalation #1

Spawning a root shell – Description

(1) Set all registers to 0 to increase the probability[1]

(2) Perform setresuid syscall to set process IDs to root

(3) Bypass check performed by Linux kernel using a glitch

(4) Execute root shell using system function

[1] Linux uses 0 for valid return values

Spawning a root shell – Code

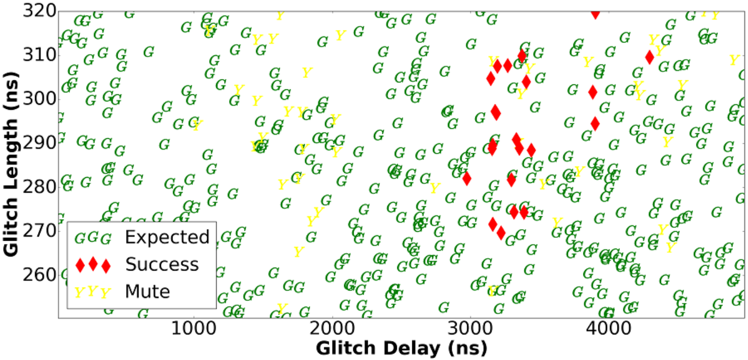
Algorithm 2 Executing a root shell

$r0 \leftarrow r1 \leftarrow r2 \leftarrow 0$
$r3 \leftarrow r4 \leftarrow r5 \leftarrow 0$
$r6 \leftarrow r7 \leftarrow r8 \leftarrow 0$
$r9 \leftarrow r10 \leftarrow r11 \leftarrow 0$
$r7 \leftarrow 0xd0$
$svc\ \#0$
if $r0 == 0$ **then**
    $system("/bin/sh")$
end if

Remarks

- Implemented using ARM assembly
- Linux syscall: sys_setresuid (0xd0)

Spawning a root shell – Results



Remarks

- We took 18968 experiments in 21 hours
- The success rate between 3.14 µs and 3.44 µs is: 1.3%
- We spawn a root shell every 5 minutes !

Privilege escalation #2

Reflection

- Linux checks can be (easily) bypassed using fault injection
- Attacks are identified and reproduced within a day
- Full fault injection attack surface not explored

Can we mitigate these type of attacks?

Reflection

- Linux checks can be (easily) bypassed using fault injection
- Attacks are identified and reproduced within a day
- Full fault injection attack surface not explored

Can we mitigate these type of attacks?

Mitigations

Software fault injection countermeasures

- Double checks
- Random delays
- Flow counters

Can these be implemented easily for larger code bases?

Hardware fault injection countermeasures

- Redundancy
- Integrity
- Sensors and detectors

Are these implemented for standard embedded technology?

Mitigations

Software fault injection countermeasures

- Double checks
- Random delays
- Flow counters

Can these be implemented easily for larger code bases?

Hardware fault injection countermeasures

- Redundancy
- Integrity
- Sensors and detectors

Are these implemented for standard embedded technology?

Mitigations

Software fault injection countermeasures

- Double checks
- Random delays
- Flow counters

Can these be implemented easily for larger code bases?

Hardware fault injection countermeasures

- Redundancy
- Integrity
- Sensors and detectors

Are these implemented for standard embedded technology?

Mitigations

Software fault injection countermeasures

- Double checks
- Random delays
- Flow counters

Can these be implemented easily for larger code bases?

Hardware fault injection countermeasures

- Redundancy
- Integrity
- Sensors and detectors

Are these implemented for standard embedded technology?

Is this all?

There are more attack vectors!

Controlling PC directly[2]

- ARM (AArch32) has an interesting ISA characteristic
- The program counter (PC) register is directly accessible

Several valid ARM instructions

```
MOV r7,r1          00000001 01110000 10100000 11100001
EOR r0,r1          00000001 00000000 00100000 11100000
LDR r0,[r1]        00000000 00000000 10010001 11100101
LDMIA r0,{r1}      00000010 00000000 10010000 11101000
```

Several corrupted ARM instructions setting PC directly

```
MOV pc,r1          00000001 11110000 10100000 11100001
EOR pc,r1          00000001 11110000 00101111 11100000
LDR pc,[r1]        00000000 11110000 10010001 11100101
LDMIA r0,{r1, pc}  00000010 10000000 10010000 11101000
```

Variations of this attack affect other architectures!

[2] Controlling PC on ARM using Fault Injection – Timmers et al. (FDTC2016)

Controlling PC directly[2]

- ARM (AArch32) has an interesting ISA characteristic
- The program counter (PC) register is directly accessible

Several valid ARM instructions

```
MOV r7,r1          00000001 01110000 10100000 11100001
EOR r0,r1          00000001 00000000 00100000 11100000
LDR r0,[r1]        00000000 00000000 10010001 11100101
LDMIA r0,{r1}      00000010 00000000 10010000 11101000
```

Several corrupted ARM instructions setting PC directly

```
MOV pc,r1          00000001 11110000 10100000 11100001
EOR pc,r1          00000001 11110000 00101111 11100000
LDR pc,[r1]        00000000 11110000 10010001 11100101
LDMIA r0,{r1, pc}  00000010 10000000 10010000 11101000
```

Variations of this attack affect other architectures!

[2] Controlling PC on ARM using Fault Injection – Timmers et al. (FDTC2016)

Controlling PC directly[2]

- ARM (AArch32) has an interesting ISA characteristic
- The program counter (PC) register is directly accessible

Several valid ARM instructions

```
MOV r7,r1          00000001 01110000 10100000 11100001
EOR r0,r1          00000001 00000000 00100000 11100000
LDR r0,[r1]        00000000 00000000 10010001 11100101
LDMIA r0,{r1}      00000010 00000000 10010000 11101000
```

Several corrupted ARM instructions setting PC directly

```
MOV pc,r1          00000001 11110000 10100000 11100001
EOR pc,r1          00000001 11110000 00101111 11100000
LDR pc,[r1]        00000000 11110000 10010001 11100101
LDMIA r0,{r1, pc}  00000010 10000000 10010000 11101000
```

Variations of this attack affect other architectures!

[2] Controlling PC on ARM using Fault Injection – Timmers et al. (FDTC2016)

Controlling PC directly[2]

- ARM (AArch32) has an interesting ISA characteristic
- The program counter (PC) register is directly accessible

Several valid ARM instructions

```
MOV r7,r1          00000001 01110000 10100000 11100001
EOR r0,r1          00000001 00000000 00100000 11100000
LDR r0,[r1]        00000000 00000000 10010001 11100101
LDMIA r0,{r1}      00000010 00000000 10010000 11101000
```

Several corrupted ARM instructions setting PC directly

```
MOV pc,r1          00000001 11110000 10100000 11100001
EOR pc,r1          00000001 11110000 00101111 11100000
LDR pc,[r1]        00000000 11110000 10010001 11100101
LDMIA r0,{r1, pc}  00000010 10000000 10010000 11101000
```

Variations of this attack affect other architectures!

[2] Controlling PC on ARM using Fault Injection – Timmers et al. (FDTC2016)

Controlling PC directly – Description

(1) Set all registers to an arbitrary value (e.g. 0x41414141)

(2) Execute random Linux system calls

(3) Load the arbitrary value into the PC register using a glitch

Controlling PC directly – Code

Algorithm 3 Linux user space code

$r0 \leftarrow r1 \leftarrow r2 \leftarrow 0x41414141$
$r3 \leftarrow r4 \leftarrow r5 \leftarrow 0x41414141$
$r6 \leftarrow r7 \leftarrow r8 \leftarrow 0x41414141$
$r9 \leftarrow r10 \leftarrow r11 \leftarrow 0x41414141$
$r7 \leftarrow getRandom()$
$svc\ \#0$

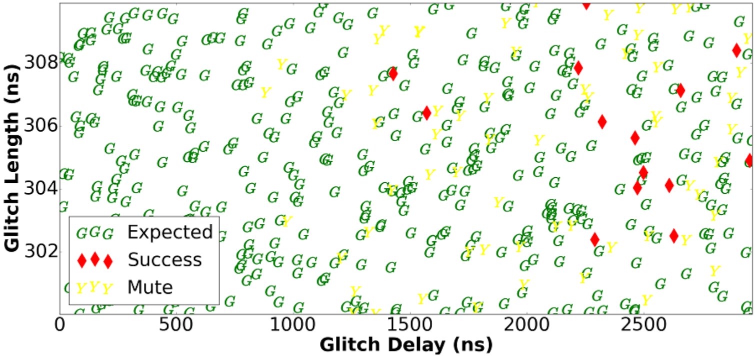
Remarks

- Implemented using ARM assembly
- Linux syscall: initially random
- Found to be vulnerable: sys_getgroups and sys_prctl

Controlling PC directly – Successful

```
Unable to handle kernel paging request at virtual addr 41414140
pgd = 5db7c000..[41414140] *pgd=0141141e(bad)
Internal error: Oops - BUG: 8000000d [#1] PREEMPT SMP ARM
Modules linked in:
CPU: 0 PID: 1280 Comm: control-pc Not tainted <redacted> #1
task: 5d9089c0 ti: 5daa0000 task.ti: 5daa0000
PC is at 0x41414140
LR is at SyS_prctl+0x38/0x404
pc : 41414140  lr : 4002ef14  psr: 60000033
sp : 5daa1fe0  ip : 18c5387d  fp : 41414141
r10: 41414141  r9 : 41414141  r8 : 41414141
r7 : 000000ac  r6 : 41414141  r5 : 41414141  r4 : 41414141
r3 : 41414141  r2 : 5d9089c0  r1 : 5daa1fa0  r0 : ffffffea
Flags: nZCv IRQs on FIQs on  Mode SVC_32 ISA Thumb Segment user
Control: 18c5387d  Table: 1db7c04a  DAC: 00000015
Process control-pc (pid: 1280, stack limit = 0x5daa0238)
Stack: (0x5daa1fe0 to 0x5daa2000)
```

Controlling PC directly – Results



Remarks

- We took 12705 experiments in 14 hours
- The success rate between 2.2 µs and 2.65 µs is: 0.63%
- We load a controlled value in PC every 10 minutes !

Privilege escalation #3

What is so special about this attack?

- Load an arbitrary value in any register
- We do not need to have access to source code
- The control flow is fully hijacked
- Software under full control of the attacker

Software fault injection countermeasures are ineffective!

What is so special about this attack?

- Load an arbitrary value in any register
- We do not need to have access to source code
- The control flow is fully hijacked
- Software under full control of the attacker

Software fault injection countermeasures are ineffective!

What can be done about it?

- Fault injection resistant hardware
- Software exploitation mitigations
- Make assets inaccessible from software

Exploitation must be made hard!

What can be done about it?

- Fault injection resistant hardware
- Software exploitation mitigations
- Make assets inaccessible from software

Exploitation must be made hard!

Conclusion

- Fault injection is an effective method to compromise Linux
- All attacks are identified and reproduced within a day
- Full code execution can be reliably achieved
- A new fault injection attack vector discussed
- Exploit mitigations becoming fundamental for fault injection
- Fault injection may be cheaper than software exploitation

Any questions?

Niek Timmers
timmers@riscure.com
(@tieknimmers)

Cristofaro Mune
c.mune@pulse-sec.com
(@pulsoid)

www.riscure.com/careers
inforequest@riscure.com