Using Fault Injection to Turn Data Transfers into Arbitrary Execution

Niek Timmers
niek@twentytwosecurity.com
@tieknimmers

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

Today's agenda

- Introduction
- Fault Injection basics
- Fault models and attacks
- Previous research:
  - From data transfer to arbitrary execution on ARM (AArch32)
- New research:
  - Generalizing the attack technique to other architectures
  - New techniques, attack simulations and demos
- Takeaways

Who are we…

- Cristofaro Mune
  - Product Security Consultant
  - Security trainer
  - Research:
    - Fault injection
    - TEEs
    - White-box Cryptography
    - Device exploitation

- Niek Timmers
  - Freelance Device Security Expert
  - Security trainer
  - Interests:
    - Secure boot
    - Fault injection
    - Low-level software
    - Fuzzing

FAULT INJECTION BASICS

Fault injection

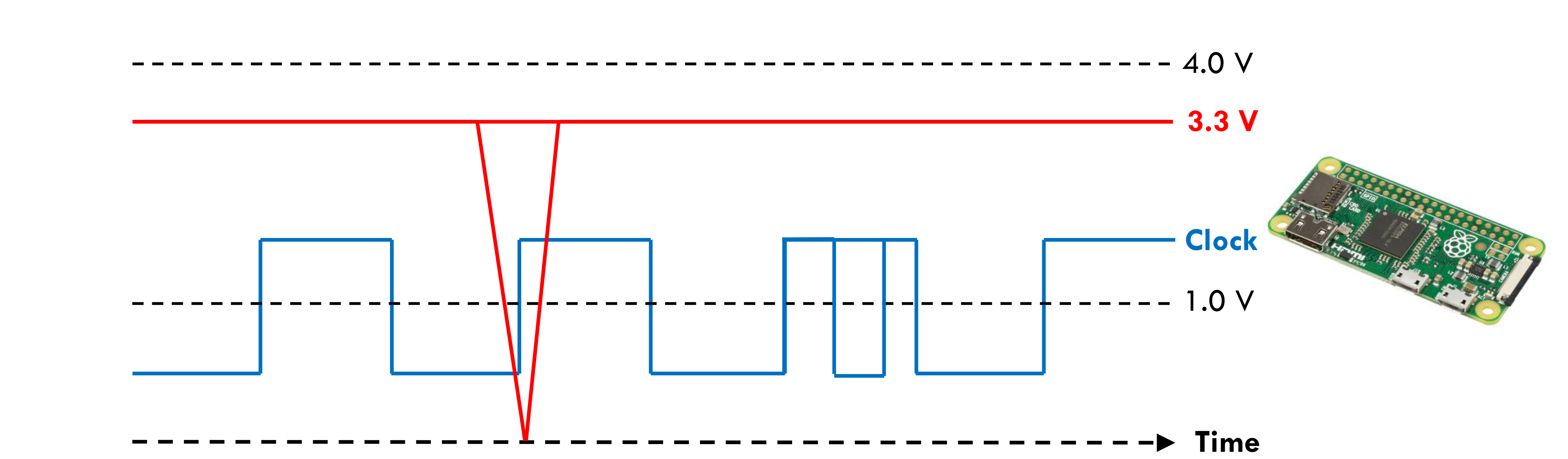
*“Introducing **faults** into a chip to alter its intended behavior.”*


```
// chec            boot is enabled
if(SECU            == 1) {
    auth           mage);
}

// execute the image
execute(&image);
```

*Awesome! **But…** how can these faults be introduced?*

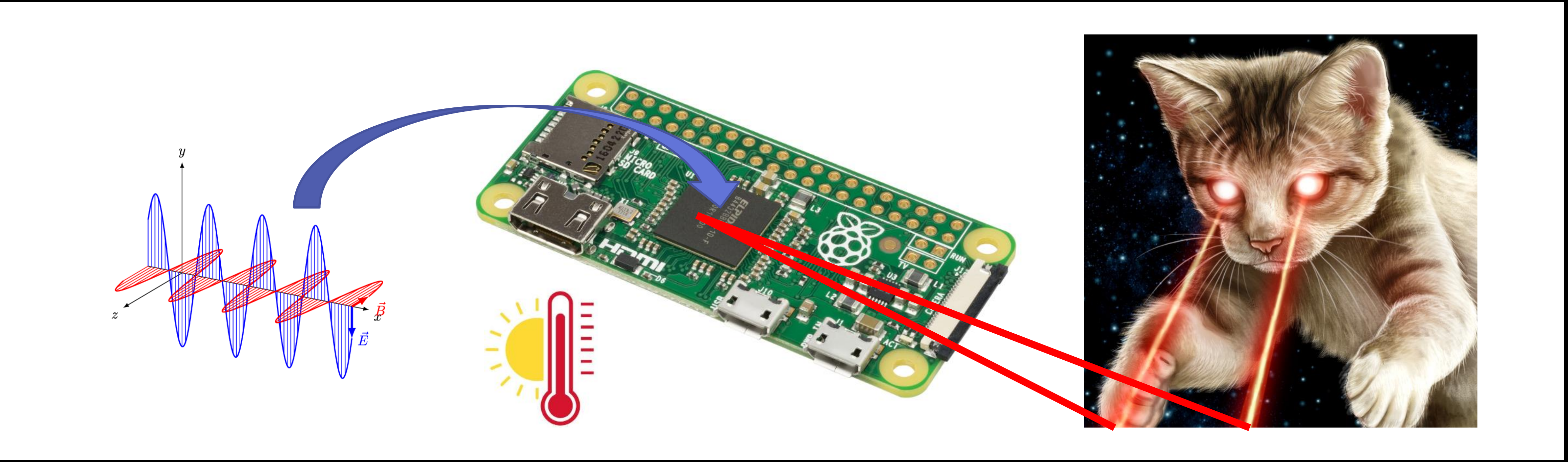
Fault injection techniques



Voltage

Clock

Fault injection techniques

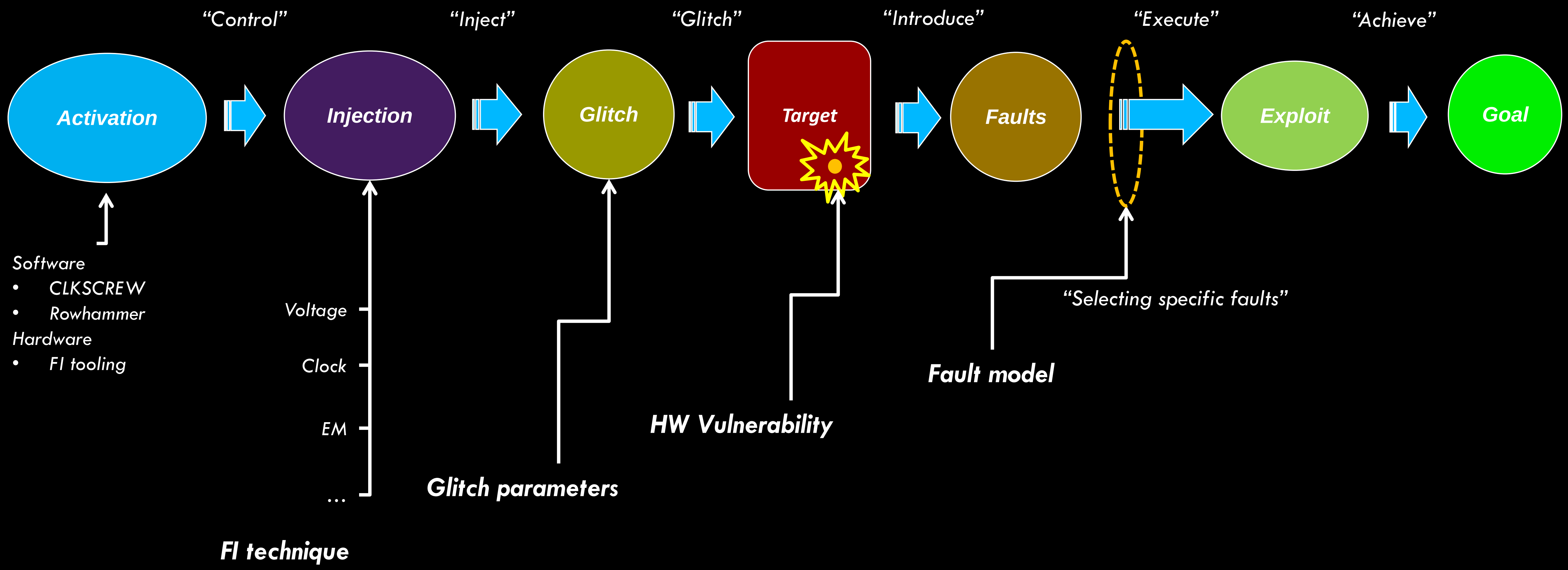


Voltage Clock Electromagnetic Laser

What goes wrong because of these glitches?

GOOD QUESTION; DIFFICULT ANSWER

Fault injection reference model



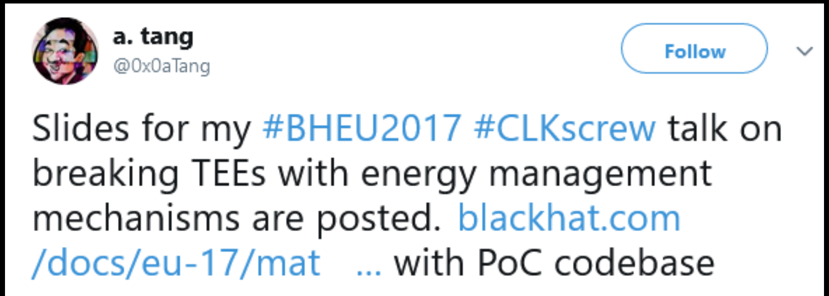
Fault models

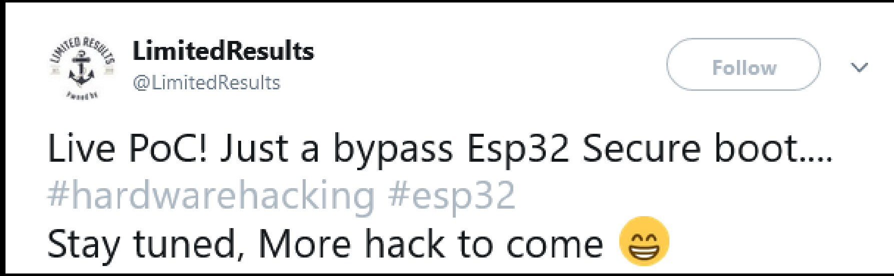
- Instruction skipping
  - E.g. bypassing a check by not executing a critical instruction
- Data corruption
  - E.g. recovering a key by corrupting cryptographic primitives
- Instruction corruption
  - Very powerful, we will cover this later on…

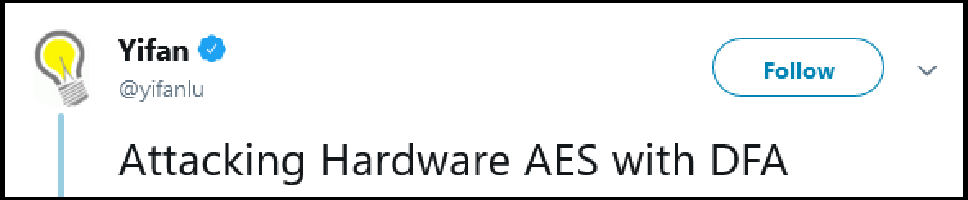
Just the typical ones…

Typical fault attacks

- Breaking crypto wallets
- Hacking smart phones
- Bypassing secure boot
- Recovering keys from crypto engines







WE HAVE ATTACKS UP OUR SLEEVE AS WELL ☺

Arbitrary execution using fault injection

Controlling PC on ARM using Fault Injection

Niek Timmers
Riscure – Security Lab

Albert Spruyt
Riscure – Security Lab

Marc Witteman
Riscure – Security Lab

- One of the first examples where instruction corruption is described
- Introduces powerful attack: PC control on AArch32

Instruction corruption

- Glitches are used to corrupt instructions
  - Single bit corruptions

```
add   x0, x1, x3     = 10001011000000110000000000100000
add   x0, x1, x2     = 10001011000000100000000000100000
```

  - Multi bit corruptions

```
ldr   x0, [sp, #32] = 11111001010000000001001111100000
str   x0, [x0, #32] = 11111001000000000001000000000000
```

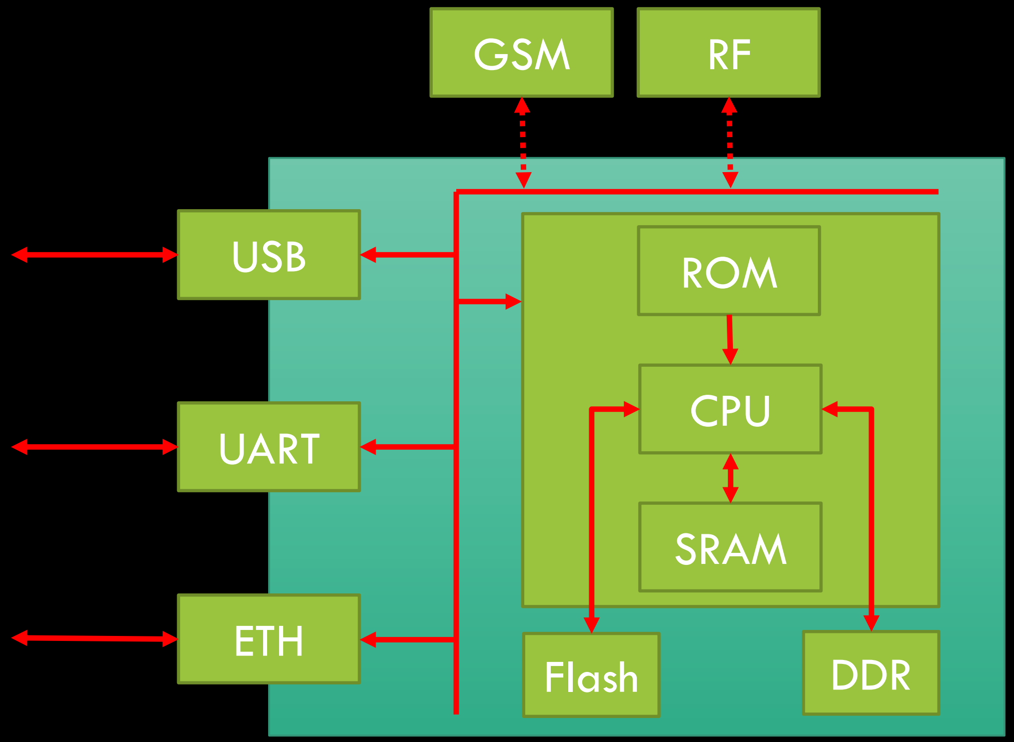
- Most chips are affected by this fault model
  - Which bits can be controlled, and how, depends on the target, …
- As software is modified; any software security model breaks

We decided to use this to devise powerful attacks…

TURNING DATA TRANSFERS INTO ARBITRARY EXECUTION

Data transfers are a great target

- All devices transfer data
- From memory to memory
- Using external interfaces



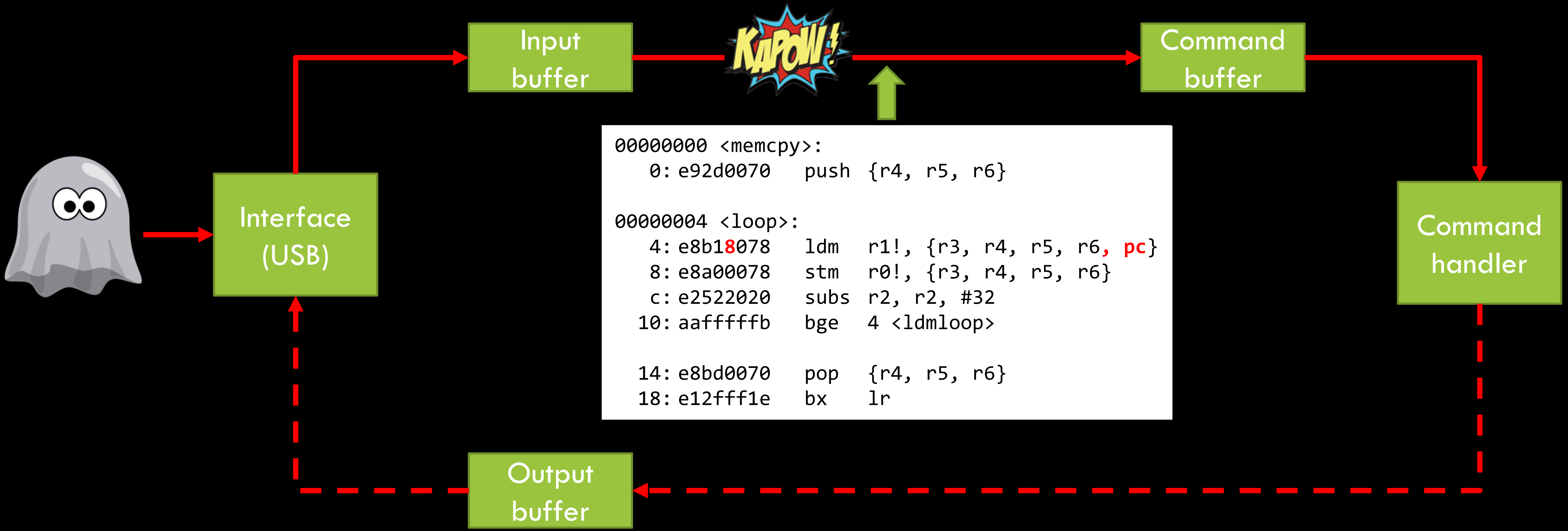
Transferring data is fundamental for devices!

Fault injection target: memcpy

- It's everywhere.
- Parameters are typically checked (dest, src and n)
- Data itself often not considered security critical

Let's use it as a Fault Injection target…

Attack description




```
00000000 <memcpy>:
   0: e92d0070   push  {r4, r5, r6}

00000004 <loop>:
   4: e8b18078   ldm   r1!, {r3, r4, r5, r6, pc}
   8: e8a00078   stm   r0!, {r3, r4, r5, r6}
   c: e2522020   subs  r2, r2, #32
  10: aafffffb   bge   4 <ldmloop>

  14: e8bd0070   pop   {r4, r5, r6}
  18: e12fff1e   bx    lr
```

Control flow is directly hijacked using an 1 bit fault!

Attack summary

- Corrupt instruction
- Modify load instruction operands (destination register)
- Directly addressable PC is set to attacker controlled value

We used this technique for different attacks

- Escalating privileges from user to kernel in Linux
  - R00ting the Unexploitable using Hardware Fault Injection @ BlueHat v17
- Bypassing encrypted secure boot
  - Hardening Secure Boot on Embedded Devices @ Blue Hat IL 2019
- Taking control of an AUTOSAR based ECU
  - Attacking AUTOSAR using Software and Hardware Attacks @ escar USA 2019

NICE!   BUT… AARCH32 SPECIFIC!

Our new research

- Generalize turning data transfers into code execution using FI
- Identify techniques that apply to all architectures (e.g. ARMv8)

We focus on…

- Software that transfers attacker controlled data
- Instructions that affect control flow:
  - Jumps (JMP, JX, etc.)
  - Calls (BL, CALL, etc.)
  - Returns (RET, etc.)
  - Exception returns (ERET, etc.)
  - …

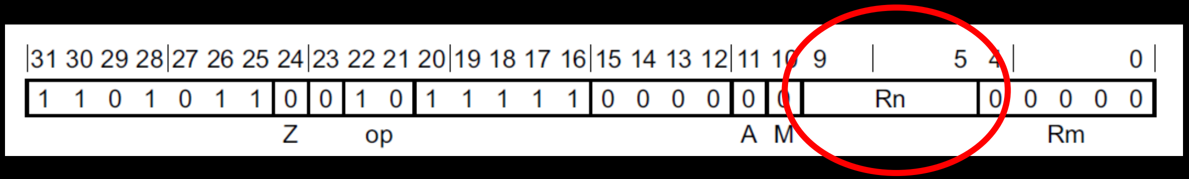
Present in all architectures and systems!

TECHNIQUE 42:
EPILOGUE: ‘RET’ CORRUPTION

The RET instruction

- We are talking here about RET on ARMv8

- It has the following encoding:



- Interestingly, the RET instruction can encode any register (x0 to x30)

How do we attack?

Real world example

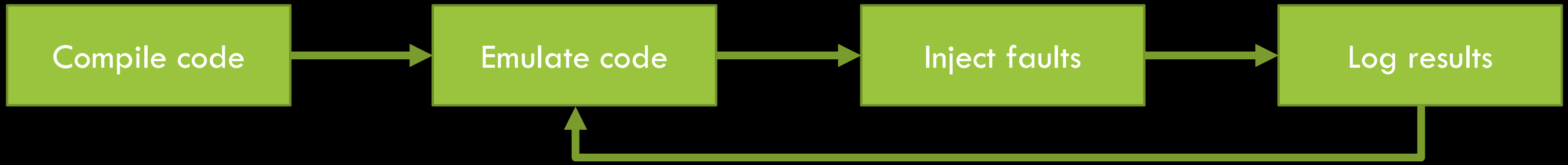
- Let's have a look at Google Bionic's (LIBC) memcpy
- Optimized memcpy uses different code based on length
- Copying 16 bytes executes the following code:
  - Source data resides in x6 and x7
  - Source data is not wiped before RET
- Glitch RET into RET x6 or RET x7

```
memcpy:
 0:8b020024 add  x4, x1, x2
 4:8b020005 add  x5, x0, x2
 8:f100405f cmp  x2, #0x10
 c:54000229 b.ls 50 <memcpy+0x50
...
50:f100205f cmp  x2, #0x8
54:540000e3 b.cc 70 <memcpy+0x70>
58:f9400026 ldr  x6, [x1]
5c:f85f8087 ldur x7, [x4, #-8]
60:f9000006 str  x6, [x0]
64:f81f80a7 stur x7, [x5, #-8]
68:d65f03c0 ret
```

What kind of fault do we need? Good question…

Fault injection simulator

- Manually applying fault models is tedious; we need automation



- We assume various faults at the instruction level
  - Bit flips: 1, 2, 3 and 4 (Flip{x})
  - Bits to zero: 1, 2, 3 and 4 (BiZe{x})
  - Bytes to zero: 1, 2, and 4 (ByZe{x})

Important: actual faults in a real target may be different!

Fault injection attack simulation

- The following faults change RET into RET x6 or RET x7:

```
PC is set to 4141414141414141 when at 00000908 c0035fd6 is set to c0005fd6 (Flip2) 'ret ' is set to 'ret x6'
PC is set to 4141414141414141 when at 00000908 c0035fd6 is set to c0005fd6 (ByZe1) 'ret ' is set to 'ret x6'
PC is set to 4141414141414141 when at 00000908 c0035fd6 is set to c0005fd6 (BiZe4) 'ret ' is set to 'ret x6'
PC is set to 4141414141414141 when at 00000908 c0035fd6 is set to c0005fd6 (BiZe2) 'ret ' is set to 'ret x6'
PC is set to 4141414141414141 when at 00000908 c0035fd6 is set to c0005fd6 (BiZe3) 'ret ' is set to 'ret x6'

        attack result                  offset   BinIns              BinIns    FM     Ins                 Ins
```

- Interestingly, the simulator also finds faults we did not think of:

```
PC is set to 4141414141414141 when at 000008f8 260040f9 is set to 3e0040f9 (Flip2) 'ldr x6, [x1]' is set to 'ldr x30, [x1]'
```

The simulator seems to be effective!

Observations

- Attack simulation helps identifying interesting faults
- Assuming corrupting a single bit is easier; likelihood depends upon:
  - instruction encoding
  - operands/registers that are used (by compiler and/or developer)
- Different type of faults can be used for successful attacks

Let's explore some other avenues…

TECHNIQUE 66:
‘CALL/JMP INSTRUCTION’ CORRUPTION

CALL/JMP instructions

- All architectures have variants of CALL/JMP instructions
- We are interested in the variant that uses a <u>register</u> as an <u>operand</u>
  - ARMv7: `bx r0`
  - ARMv8: `blr x0`
  - MIPS: `jr $a0`
  - Xtensa: `callx0 a4`
  - …

Used e.g. to call functions and function pointers!

Attack example

- Simple command handler implemented using a function pointer
- The arguments are under control of the attackers

```
typedef struct Command {
    unsigned int cmdid;
    int(*function)(size_t arg1, size_t arg2, size_t arg3);
} Command;

typedef struct CommandIn {
    unsigned int cmdid; size_t arg1; size_t arg2; size_t arg3;
} CommandIn;

const struct Command commands[] = { ... };

int command_loop() {
      ...
      while(commands[i].cmdid != 0) {
         if(commands[i].cmdid == cmd->cmdid) {
             ret = (*commands[i].function)(cmd->arg1, cmd->arg2, cmd->arg3);
         }
         i++;
      }
      ...
```

```
0000000000000010 <command_loop>:
  10: a9be7bfd    stp   x29, x30, [sp, #-32]!
  14: 52800400    mov   w0, #0x20
  18: 910003fd    mov   x29, sp
  1c: a90153f3    stp   x19, x20, [sp, #16]
  20: 94000000    bl    0 <malloc_s>
  24: 90000013    adrp  x19, 0 <f1>
  28: 91000273    add   x19, x19, #0x0
  2c: aa0003f4    mov   x20, x0
<cut>
  54: b9400282    ldr   w2, [x20]
  58: 6b01005f    cmp   w2, w1
  5c: 540000c1    b.ne  74 <command_loop+0x64>
  60: f9400263    ldr   x3, [x19]
  64: a9408680    ldp   x0, x1, [x20, #8]
  68: f9400e82    ldr   x2, [x20, #24]
  6c: d63f0060    blr   x3
  70: 93407c00    sxtw  x0, w0
  74: 91004273    add   x19, x19, #0x10
  78: 17fffff2    b     40 <command_loop+0x30>
```

Attack example

- The function call is performed using a blr instruction
- The attacker's goal is to change x3 into x0, x1 or x2

```c
typedef struct Command {
    unsigned int cmdid;
    int(*function)(size_t arg1, size_t arg2, size_t arg3);
} Command;

typedef struct CommandIn {
    unsigned int cmdid; size_t arg1; size_t arg2; size_t arg3;
} CommandIn;

const struct Command commands[] = { ... };

int command_loop() {
      ...
      while(commands[i].cmdid != 0) {
         if(commands[i].cmdid == cmd->cmdid) {
             ret = (*commands[i].function)(cmd->arg1, cmd->arg2, cmd->arg3);
         }
         i++;
      }
      ...
```

```
0000000000000010 <command_loop>:
  10: a9be7bfd   stp   x29, x30, [sp, #-32]!
  14: 52800400   mov   w0, #0x20
  18: 910003fd   mov   x29, sp
  1c: a90153f3   stp   x19, x20, [sp, #16]
  20: 94000000   bl    0 <malloc_s>
  24: 90000013   adrp  x19, 0 <f1>
  28: 91000273   add   x19, x19, #0x0
  2c: aa0003f4   mov   x20, x0
<cut>
  54: b9400282   ldr   w2, [x20]
  58: 6b01005f   cmp   w2, w1
  5c: 540000c1   b.ne  74 <command_loop+0x64>
  60: f9400263   ldr   x3, [x19]
  64: a9408680   ldp   x0, x1, [x20, #8]
  68: f9400e82   ldr   x2, [x20, #24]
  6c: d63f0060   blr   x0 // or x1 or x2
  70: 93407c00   sxtw  x0, w0
  74: 91004273   add   x19, x19, #0x10
  78: 17fffff2   b     40 <command_loop+0x30>
```

Fault injection attack simulation

The following faults corrupt blr x3 to blr x0, blr x1 or blr x2:

```
PC is set to 4141414141414141 when at 00000620 60003fd6 is set to 40003fd6 (Flip1) 'blr x3' is set to 'blr x2'
PC is set to 4141414141414141 when at 00000620 60003fd6 is set to 20003fd6 (Flip1) 'blr x3' is set to 'blr x1'
PC is set to 4141414141414141 when at 00000620 60003fd6 is set to 00003fd6 (Flip2) 'blr x3' is set to 'blr x0'
PC is set to 4141414141414141 when at 00000620 60003fd6 is set to 40003fd6 (BiZe1) 'blr x3' is set to 'blr x2'
PC is set to 4141414141414141 when at 00000620 60003fd6 is set to 20003fd6 (BiZe1) 'blr x3' is set to 'blr x1'
PC is set to 4141414141414141 when at 00000620 60003fd6 is set to 40003fd6 (BiZe2) 'blr x3' is set to 'blr x2'
PC is set to 4141414141414141 when at 00000620 60003fd6 is set to 20003fd6 (BiZe2) 'blr x3' is set to 'blr x1'
PC is set to 4141414141414141 when at 00000620 60003fd6 is set to 00003fd6 (BiZe2) 'blr x3' is set to 'blr x0’
... <cut BiZe3 out>
PC is set to 4141414141414141 when at 00000620 60003fd6 is set to 40003fd6 (BiZe4) 'blr x3' is set to 'blr x2'
PC is set to 4141414141414141 when at 00000620 60003fd6 is set to 20003fd6 (BiZe4) 'blr x3' is set to 'blr x1'
PC is set to 4141414141414141 when at 00000620 60003fd6 is set to 00003fd6 (BiZe4) 'blr x3' is set to 'blr x0'
PC is set to 4141414141414141 when at 00000620 60003fd6 is set to 00003fd6 (ByZe1) 'blr x3' is set to 'blr x0'
PC is set to 4141414141414141 when at 00000620 60003fd6 is set to 00003fd6 (ByZe2) 'blr x3' is set to 'blr x0'
PC is set to 4141414141414141 when at 00000620 60003fd6 is set to 20003fd6 (Flip1) 'blr x3' is set to 'blr x1'
PC is set to 4141414141414141 when at 00000620 60003fd6 is set to 40003fd6 (Flip1) 'blr x3' is set to 'blr x2'
PC is set to 4141414141414141 when at 00000620 60003fd6 is set to 00003fd6 (Flip2) 'blr x3' is set to 'blr x0'
```

Different faults yield full PC control!

FAULT INJECTION INTERMEZZO

Demo of 'CALL/JMP' corruption on ESP32 (Xtensa)

- Our goal is to jump to an arbitrary location
- Xtensa LX6
  - Program counter is not directly addressable
  - Function pointers implemented using callx8
- Test program:
  - Load address of normal() in register a7
  - Load address of pwn() in all other registers
  - Likelihood increased by using a callx8 sled
- Corrupt callx8 to use a different register

```
asm volatile (

"movi a2, 0x400d1dc0;"    // pwn() address
“movi a3, 0x400d1dc0;"    // pwn() address
<cut init of registers>
"movi a14, 0x400d1dc0;"   // pwn() address
"movi a15, 0x400d1dc0;"   // pwn() address
"movi a7, 0x400d1da8;"    // normal() address
"callx8 a7;"
“callx8 a7;"
“callx8 a7;"
<cut 10,000 callx8 instructions>
"callx8 a7;"
"callx8 a7;"
"callx8 a7;“
...
);
```

Test is successful when pwn() gets executed!

Demo summary

Using a glitch, we corrupted callx8's operand to use a different register:

```
Total: 630 illegal_ins: 114 [18.10%] expected: 299 [47.46%] core_panic: 29 [4.60%] PC_control: 34 [5.40%] PC_dirty_control: 4 [0.63%] mute: 1 [0.16%]
Total: 631 illegal_ins: 114 [18.07%] expected: 300 [47.54%] core_panic: 29 [4.60%] PC_control: 34 [5.39%] PC_dirty_control: 4 [0.63%] mute: 1 [0.16%]
Total: 632 illegal_ins: 115 [18.20%] expected: 300 [47.47%] core_panic: 29 [4.59%] PC_control: 34 [5.38%] PC_dirty_control: 4 [0.63%] mute: 1 [0.16%]
Total: 634 illegal_ins: 115 [18.14%] expected: 301 [47.48%] core_panic: 29 [4.57%] PC_control: 34 [5.36%] PC_dirty_control: 4 [0.63%] mute: 1 [0.16%]
Total: 635 illegal_ins: 115 [18.11%] expected: 302 [47.56%] core_panic: 29 [4.57%] PC_control: 34 [5.35%] PC_dirty_control: 4 [0.63%] mute: 1 [0.16%]
Total: 636 illegal_ins: 116 [18.24%] expected: 302 [47.48%] core_panic: 29 [4.56%] PC_control: 34 [5.35%] PC_dirty_control: 4 [0.63%] mute: 1 [0.16%]
```

- Success rates:
  - Success per experiment: ~5%
  - Time per success: a few minutes

This demonstrates PC control is possible!

Observations

- Control flow redirected to arbitrary locations
- Code construct not realistic but sufficient for PoC (☺)
- An attacker would need to corrupt a specific instruction:
  - Feasibility depends on lots of variable like the target, timing, technique, …

TECHNIQUE 43:
EPILOGUE : ‘REGISTER RESTORE’ CORRUPTION

Leftover data in registers

- Typical software does not wipe registers after being used
- Attacker controlled data may persist between callers and callees
- Leftover data can be very useful for attacks…

Attack example

- Extracted from a modern mobile phone's bootloader
- Attacker controlled data is copied to internal memory

```
...
BL    memcpy_s
CMP   X0, #8
STR   XZR, [SP, #0x30,+var_28]
LDR   X8, [SP, #0x30,+var_28]
CBZ   X8, loc_622C97C
MOV   W1, #1
MOV   X0, X20
MOV   X2, X19
SUB   SP, X29, #0x20
LDP   X29, X30, [SP, #0x20+var_s0]
LDP   X20, X19, [SP, #0x20+var_10]
LDP   X22, X21, [SP, #0x20+var_20]
RET
; End of function calling memcpy_s
```

```
...
BL    memcpy
MOV   X0, X19
LDP   X29, X30, [SP, #0x10]
LDP   X20, X19, [SP], #0x20
RET
; End of function memcpy_s
```

```
...
LDR   W10, [x8], #-4
STR   W10, [x9], #-4
SUBS  X2, X2, #4
B.NE  loc_6219B38
RET
; End of function memcpy
```

Where do we attack?

Attack example

- Always focus on the attacker controlled data
- Attacker controlled data (x10) is not clobbered by callee or callers

```
...
BL    memcpy_s
CMP   X0, #8
STR   XZR, [SP, #0x30,+var_28]
LDR   X8, [SP, #0x30,+var_28]
CBZ   X8, loc_622C97C
MOV   W1, #1
MOV   X0, X20
MOV   X2, X19
SUB   SP, X29, #0x20
LDP   X29, X30, [SP, #0x20+var_s0]
LDP   X20, X19, [SP, #0x20+var_10]
LDP   X22, X21, [SP, #0x20+var_20]
RET
; End of function calling memcpy_s
```

```
...
BL    memcpy
MOV   X0, X19
LDP   X29, X30, [SP, #0x10]
LDP   X20, X19, [SP], #0x20
RET
; End of function memcpy_s
```

```
...
LDR   W10, [x8], #-4
STR   W10, [x9], #-4
SUBS  X2, X2, #4
B.NE  loc_6219B38
RET
; End of function memcpy
```

Attack example

- Corrupt LDP to set X29 and X30 with attacker controlled data
- Set X30 to any function epilogue where X29 is used to restore SP

```
...
BL    memcpy_s
CMP   X0, #8
STR   XZR, [SP, #0x30,+var_28]
LDR   X8, [SP, #0x30,+var_28]
CBZ   X8, loc_622C97C
MOV   W1, #1
MOV   X0, X20
MOV   X2, X19
SUB   SP, X29, #0x20
LDP   X29, X30, [SP, #0x20+var_s0]
LDP   X20, X19, [SP, #0x20+var_10]
LDP   X22, X21, [SP, #0x20+var_20]
RET
; End of function calling memcpy_s
```

```
...
BL    memcpy
MOV   X0, X19
LDP   X29, X30, [SP, #0x10]
LDP   X20, X19, [SP], #0x20
RET
; End of function memcpy_s
```

```
...
LDR   W10, [x8], #-4
STR   W10, [x9], #-4
SUBS  X2, X2, #4
B.NE  loc_6219B38
RET
; End of function memcpy
```

Now we control PC and SP…

Attack simulation

- The following faults corrupt ldp x29, x30, [sp, #0x10] to use x10:
- Our assumed fault models… do not make it possible ☹:
  - No pattern for changing SP (X31) into X10
- Attacks may still be possible in real targets, when:
  - Leftover data in different registers
  - Different fault models apply

Let's simulate it…

What if leftover data is in other registers than x10?

```
x29 and x30 set when at 00000690 fd7b41a9 is fd7841a9 (BiZe2) 'ldp x29, x30, [sp, #0x10]' >> 'ldp x29, x30, [x7, #0x10]'
x29 and x30 set when at 00000690 fd7b41a9 is fd7941a9 (BiZe1) 'ldp x29, x30, [sp, #0x10]' >> 'ldp x29, x30, [x15, #0x10]'
x29 and x30 set when at 00000690 fd7b41a9 is fd7a41a9 (BiZe1) 'ldp x29, x30, [sp, #0x10]' >> 'ldp x29, x30, [x23, #0x10]'
x29 and x30 set when at 00000690 fd7b41a9 is 3d7b41a9 (BiZe2) 'ldp x29, x30, [sp, #0x10]' >> 'ldp x29, x30, [x25, #0x10]'
x29 and x30 set when at 00000690 fd7b41a9 is 7d7b41a9 (BiZe1) 'ldp x29, x30, [sp, #0x10]' >> 'ldp x29, x30, [x27, #0x10]'
x29 and x30 set when at 00000690 fd7b41a9 is 9d7b41a9 (BiZe2) 'ldp x29, x30, [sp, #0x10]' >> 'ldp x29, x30, [x28, #0x10]'
x29 and x30 set when at 00000690 fd7b41a9 is bd7b41a9 (BiZe1) 'ldp x29, x30, [sp, #0x10]' >> 'ldp x29, x30, [x29, #0x10]'
```

Success may depend on which registers are used!

Attack summary

- Using a single bit fault we control x29, x30, SP and PC
  - But only if leftover data is in the 'right' registers…
- Perfect start for doing things like Return-Oriented-Programming (ROP)
- Very powerful when successful…

TECHNIQUE 67:
‘POINTER REGISTER’ CORRUPTION

Attack example

- Remember our simple command handler?
- With the arguments under control of an attacker…

```c
typedef struct Command {
    unsigned int cmdid;
    int(*function)(size_t arg1, size_t arg2, size_t arg3);
} Command;

typedef struct CommandIn {
    unsigned int cmdid; size_t arg1; size_t arg2; size_t arg3;
} CommandIn;

const struct Command commands[] = { ... };

int command_loop() {
      ...
      while(commands[i].cmdid != 0) {
         if(commands[i].cmdid == cmd->cmdid) {
             ret = (*commands[i].function)(cmd->arg1, cmd->arg2, cmd->arg3);
         }
         i++;
      }
      ...
```

```
0000000000000010 <command_loop>:
  10: a9be7bfd   stp   x29, x30, [sp, #-32]!
  14: 52800400   mov   w0, #0x20
  18: 910003fd   mov   x29, sp
  1c: a90153f3   stp   x19, x20, [sp, #16]
  20: 94000000   bl    0 <malloc_s>
  24: 90000013   adrp  x19, 0 <f1>
  28: 91000273   add   x19, x19, #0x0
  2c: aa0003f4   mov   x20, x0
<cut>
  54: b9400282   ldr   w2, [x20]
  58: 6b01005f   cmp   w2, w1
  5c: 540000c1   b.ne  74 <command_loop+0x64>
  60: f9400263   ldr   x3, [x19]
  64: a9408680   ldp   x0, x1, [x20, #8]
  68: f9400e82   ldr   x2, [x20, #24]
  6c: d63f0060   blr   x3
  70: 93407c00   sxtw  x0, w0
  74: 91004273   add   x19, x19, #0x10
  78: 17fffff2   b     40 <command_loop+0x30>
```

Attack example

- Another attack targets the ldr instructions setting the arguments
- Attack goal is to load an argument in x3 instead of x0, x1 or x2

```
typedef struct Command {
    unsigned int cmdid;
    int(*function)(size_t arg1, size_t arg2, size_t arg3);
} Command;

typedef struct CommandIn {
    unsigned int cmdid; size_t arg1; size_t arg2; size_t arg3;
} CommandIn;

const struct Command commands[] = { ... };

int command_loop() {
      ...
      while(commands[i].cmdid != 0) {
         if(commands[i].cmdid == cmd->cmdid) {
             ret = (*commands[i].function)(cmd->arg1, cmd->arg2, cmd->arg3);
         }
         i++;
      }
      ...
```

```
0000000000000010 <command_loop>:
  10: a9be7bfd    stp   x29, x30, [sp, #-32]!
  14: 52800400    mov   w0, #0x20
  18: 910003fd    mov   x29, sp
  1c: a90153f3    stp   x19, x20, [sp, #16]
  20: 94000000    bl    0 <malloc_s>
  24: 90000013    adrp  x19, 0 <f1>
  28: 91000273    add   x19, x19, #0x0
  2c: aa0003f4    mov   x20, x0
<cut>
  54: b9400282    ldr   w2, [x20]
  58: 6b01005f    cmp   w2, w1
  5c: 540000c1    b.ne  74 <command_loop+0x64>
  60: f9400263    ldr   x3, [x19]
  64: a9408680    ldp   x3, x3, [x20, #8]
  68: f9400e82    ldr   x3, [x20, #24]
  6c: d63f0060    blr   x3
  70: 93407c00    sxtw  x0, w0
  74: 91004273    add   x19, x19, #0x10
  78: 17fffff2    b     40 <command_loop+0x30>
```

Fault injection attack simulation

The following faults corrupt ldp or ldr to use x3 as the destination:

```
PC is 4141414141414141 when at 00000618 808640a9 is 838640a9 (Flip2) 'ldp x0, x1, [x20, #8]' is set to 'ldp x3, x1, [x20, #8]'
PC is 4141414141414141 when at 00000618 808640a9 is 808e40a9 (Flip3) 'ldp x0, x1, [x20, #8]' is set to 'ldp x0, x3, [x20, #8]'
PC is 4141414141414141 when at 0000061c 820e40f9 is 830e40f9 (Flip1) 'ldr x2, [x20, #0x18]'  is set to 'ldr x3, [x20, #0x18]'
```

However…

An interesting observation...

- Compiled instructions depend on argument type
- Compiler uses 32-bit (w0) register addressing instead of 64-bit (x0)

```
typedef struct Command {
    unsigned int cmdid;
    int(*function)(uint32_t arg1, uint32_t arg2, uint32_t arg3);
} Command;

typedef struct CommandIn {
    unsigned int cmdid; uint32_t arg1; uint32_t arg2; uint32_t arg3;
} CommandIn;

const struct Command commands[] = { ... };

int command_loop() {
      ...
      while(commands[i].cmdid != 0) {
         if(commands[i].cmdid == cmd->cmdid) {
             ret = (*commands[i].function)(cmd->arg1, cmd->arg2, cmd->arg3);
         }
         i++;
      }
      ...
```

```
0000000000000010 <command_loop>:
  10: a9be7bfd    stp   x29, x30, [sp, #-32]!
  14: 52800400    mov   w0, #0x20
  18: 910003fd    mov   x29, sp
  1c: a90153f3    stp   x19, x20, [sp, #16]
  20: 94000000    bl    0 <malloc_s>
  24: 90000013    adrp  x19, 0 <f1>
  28: 91000273    add   x19, x19, #0x0
  2c: aa0003f4    mov   x20, x0
<cut>
  54: b9400282    ldr   w2, [x20]
  58: 6b01005f    cmp   w2, w1
  5c: 540000c1    b.ne  74 <command_loop+0x64>
  60: f9400263    ldr   x3, [x19]
  64: a9408680    ldp   w0, w1, [x20, #4]
  68: f9400e82    ldr   w2, [x20, #12]
  6c: d63f0060    blr   x3
  70: 93407c00    sxtw  x0, w0
  74: 91004273    add   x19, x19, #0x10
  78: 17fffff2    b     40 <command_loop+0x30>
```

Fault injection attack simulation

The following faults corrupt ldp or ldr to use w3 as the destination:

```
PC is 0000000041414141 when at 00000618 80864029 is 83864029 (flip2) 'ldp w0, w1, [x20, #4]’ is 'ldp w3, w1, [x20, #4]'
PC is 0000000041414141 when at 0000061c 820e40b9 is 830e40b9 (flip1) 'ldr w2, [x20, #0xc]’   is 'ldr w3, [x20, #0xc]'
PC is 0000000041414141 when at 00000618 80864029 is 808e4029 (flip1) 'ldp w0, w1, [x20, #4]' is 'ldp w0, w3, [x20, #4]'
```

Unfortunately, PC control is only partial…

But...

How many bits difference between:

ldr w2, [x20, #12] and ldp x2, x3, [x20] ?

```
PC is 4141414141414141 when at 0000061c 820e40b9 is 820e40a9 (BiZe1) 'ldr w2, [x20, #0xc]' is 'ldp x2, x3, [x20]'
```

Get full PC control by corrupting only 1 bit to zero!

LET'S WRAP UP…

Takeaways

- Data transfers are great FI targets
  - Used in all systems and devices
  - They may operate on attacker controlled data
- Transferred data itself is relevant for FI attacks
  - Typically not considered security critical for software attacks
- Data transfers may yield code execution using FI on all architectures

Thank you!

Niek Timmers
niek@twentytwosecurity.com
@tieknimmers

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

Feel free to contact us!