riscure

Fault Injection Attacks on Secure Boot

Niek Timmers
timmers@riscure.com
(@tieknimmers)

Albert Spruyt
spruyt@riscure.com

April 13, 2017

Agenda

Practicalities

Fault injection

Bypasses

Mitigations

Secure boot

__Disclaimer:__ we are not talking about UEFI Secure Boot!

Agenda

Practicalities

Fault injection

Bypasses

Mitigations

Secure boot

Disclaimer: *we are not talking about UEFI Secure Boot!*

Who are we?

Albert & Niek

- Security Analysts
- Security testing of different products and technologies

Riscure

- Services (Security Test Lab)
  - Hardware / Software / Crypto
  - Embedded systems / Smart cards
- Tools
  - Side channel analysis (passive)
  - Fault injection (active)

This talk shows a bit of both...

Who are we?

Albert & Niek

- Security Analysts
- Security testing of different products and technologies

Riscure

- Services (Security Test Lab)
    - Hardware / Software / Crypto
    - Embedded systems / Smart cards
- Tools
    - Side channel analysis (passive)
    - Fault injection (active)

This talk shows a bit of both...

A fault injection 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?

A fault injection 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?

A fault injection 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?

A fault injection 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[1]

clock voltage e-magnetic laser

Source: http://www.limited-entropy.com/fault-injection-techniques/

[1] The Sorcerers Apprentice Guide to Fault Attacks. – Bar-El et al., 2004

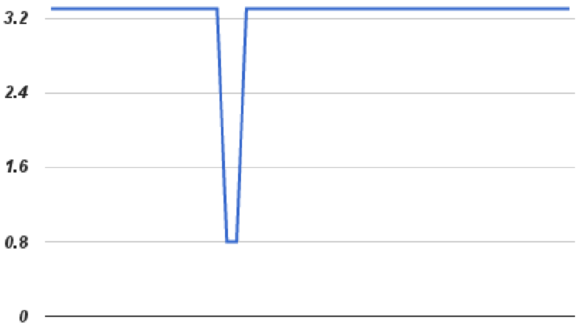
Fault injection techniques[1]

clock

voltage

e-magnetic

laser



Source: http://www.limited-entropy.com/fault-injection-techniques/

Type of faults

Faults that affect hardware

- Registers
- Buses

Faults that affect hardware that does software[2] [3] [4]

- Instruction corruption

```
mov r0, r1   11100001101000000000000000000001
mov r0, r3   11100001101000000000000000000011
```

- Instruction skipping

```
mov r0, r1   11100001101000000000000000000001
mov r0, r0   11100001101000000000000000000000
```

Is this useful?

[2] Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells – Roscian et. al., 2015

[3] High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures – Riviere et al., 2015

[4] Formal verification of a software countermeasure against instruction skip attacks – Moro et. al., 2014

Type of faults

Faults that affect hardware

- Registers
- Buses

Faults that affect hardware that does software[2] [3] [4]

- Instruction corruption

```
mov r0, r1   11100001101000000000000000000001
mov r0, r3   11100001101000000000000000000011
```

- Instruction skipping

```
mov r0, r1   11100001101000000000000000000001
mov r0, r0   11100001101000000000000000000000
```

Is this useful?

[2] Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells – Roscian et. al., 2015

[3] High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures – Riviere et al., 2015

[4] Formal verification of a software countermeasure against instruction skip attacks – Moro et. al., 2014

Type of faults

Faults that affect hardware

- Registers
- Buses

Faults that affect hardware that does software[2] [3] [4]

- Instruction corruption

```
mov r0, r1   11100001101000000000000000000001
mov r0, r3   11100001101000000000000000000011
```

- Instruction skipping

```
mov r0, r1   11100001101000000000000000000001
mov r0, r0   11100001101000000000000000000000
```

Is this useful?

[2] Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells – Roscian et. al., 2015

[3] High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures – Riviere et al., 2015

[4] Formal verification of a software countermeasure against instruction skip attacks – Moro et. al., 2014

Type of faults

Faults that affect hardware

- Registers
- Buses

Faults that affect hardware that does software[2] [3] [4]

- Instruction corruption

```
mov r0, r1   11100001101000000000000000000001
mov r0, r3   11100001101000000000000000000011
```

- Instruction skipping

```
mov r0, r1   11100001101000000000000000000001
mov r0, r0   11100001101000000000000000000000
```

Is this useful?

[2] Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells – Roscian et. al., 2015

[3] High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures – Riviere et al., 2015

[4] Formal verification of a software countermeasure against instruction skip attacks – Moro et. al., 2014

Type of faults

Faults that affect hardware

- Registers
- Buses

Faults that affect hardware that does software[2] [3] [4]

- Instruction corruption

```
mov r0, r1   11100001101000000000000000000001
mov r0, r3   11100001101000000000000000000011
```

- Instruction skipping

```
mov r0, r1   11100001101000000000000000000001
mov r0, r0   11100001101000000000000000000000
```

Is this useful?

[2] Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells – Roscian et. al., 2015

[3] High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures – Riviere et al., 2015

[4] Formal verification of a software countermeasure against instruction skip attacks – Moro et. al., 2014

Type of faults

Faults that affect hardware

- Registers
- Buses

Faults that affect hardware that does software[2] [3] [4]

- Instruction corruption

```
mov r0, r1   11100001101000000000000000000001
mov r0, r3   11100001101000000000000000000011
```

- Instruction skipping

```
mov r0, r1   11100001101000000000000000000001
mov r0, r0   11100001101000000000000000000000
```

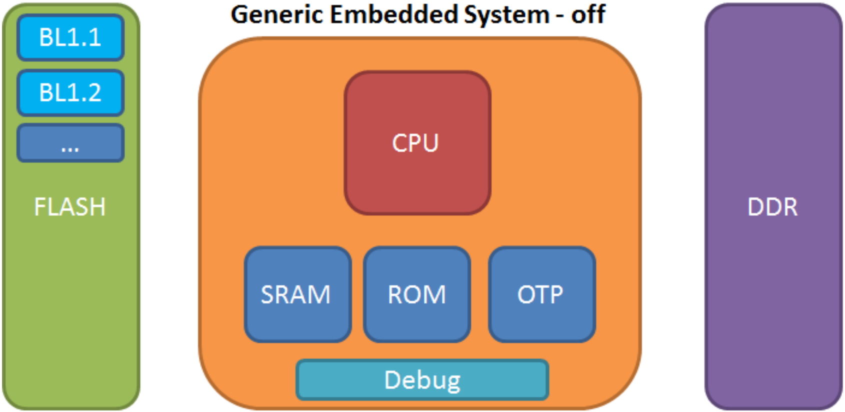
Is this useful?

[2] Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells – Roscian et. al., 2015

[3] High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures – Riviere et al., 2015

[4] Formal verification of a software countermeasure against instruction skip attacks – Moro et. al., 2014

Secure boot



Generic Embedded System - off

BL1.1

BL1.2

...

FLASH

CPU

SRAM

ROM

OTP

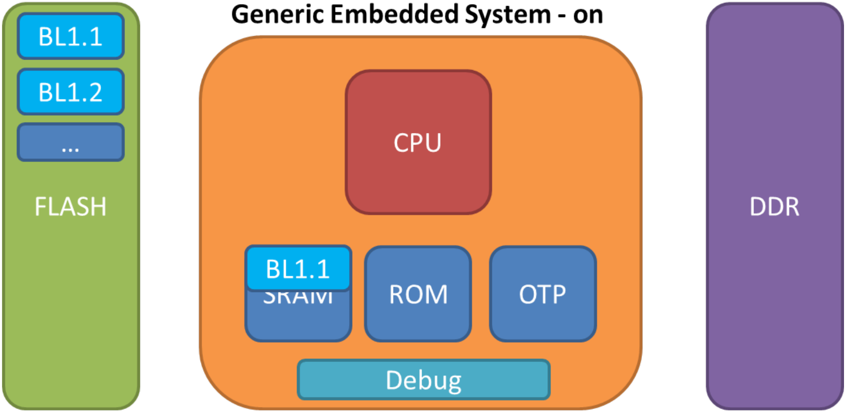
Debug

DDR

Remarks

- Integrity and confidentiality of flash contents are not assured!
- A mechanism is required for this assurance: **secure boot!**

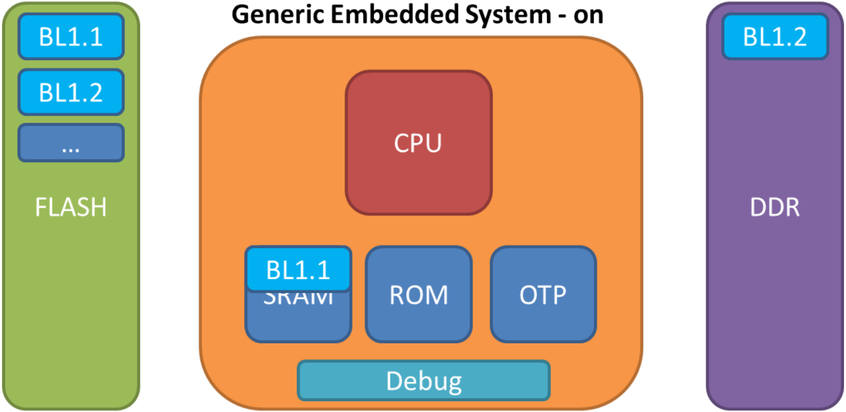
Secure boot



Remarks

- Integrity and confidentiality of flash contents are not assured!
- A mechanism is required for this assurance: **secure boot!**

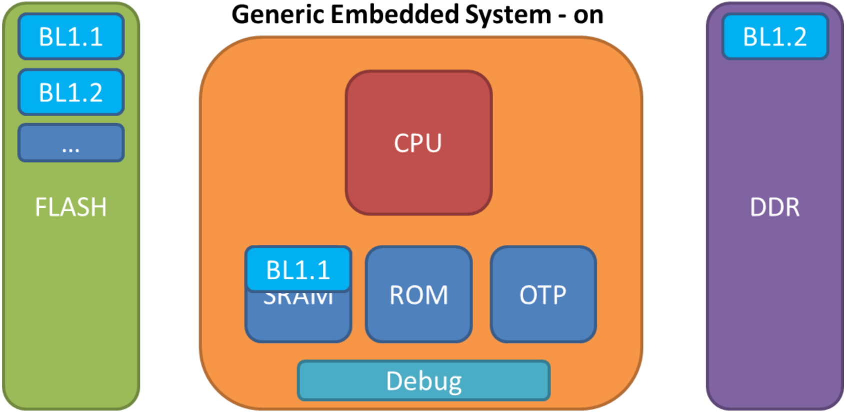
Secure boot



Remarks

- Integrity and confidentiality of flash contents are not assured!
- A mechanism is required for this assurance: **secure boot!**

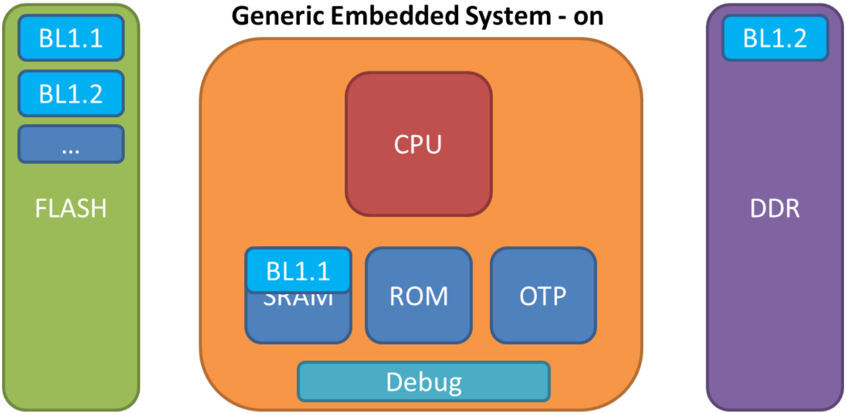
Secure boot



Remarks

- Integrity and confidentiality of flash contents are not assured!
- A mechanism is required for this assurance: **secure boot!**

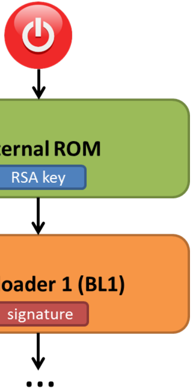
Secure boot



Remarks

- Integrity and confidentiality of flash contents are not assured!
- A mechanism is required for this assurance: **secure boot!**

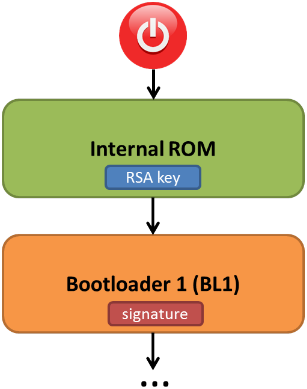
Secure boot



- Assures **integrity** (and **confidentiality**) of flash contents
- The **chain of trust** is similar to PKI[5] found in browsers
- One **root of trust** composed of immutable code and key

[5] Public Key Infrastructure

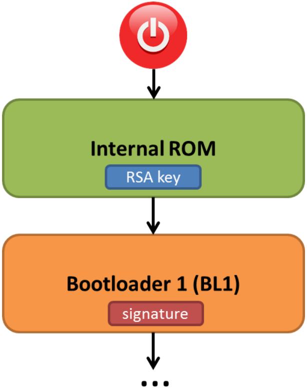
Secure boot



- Assures **integrity** (and **confidentiality**) of flash contents
- The **chain of trust** is similar to PKI[5] found in browsers
- One **root of trust** composed of immutable code and key

[5] Public Key Infrastructure

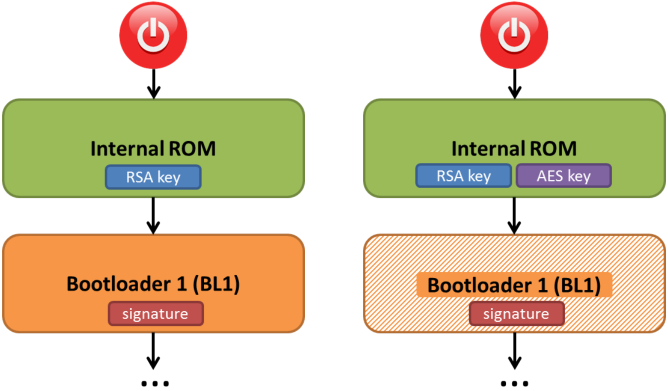
Secure boot



- Assures **integrity** (and **confidentiality**) of flash contents
- The **chain of trust** is similar to PKI[5] found in browsers
- One **root of trust** composed of immutable code and key

[5] Public Key Infrastructure

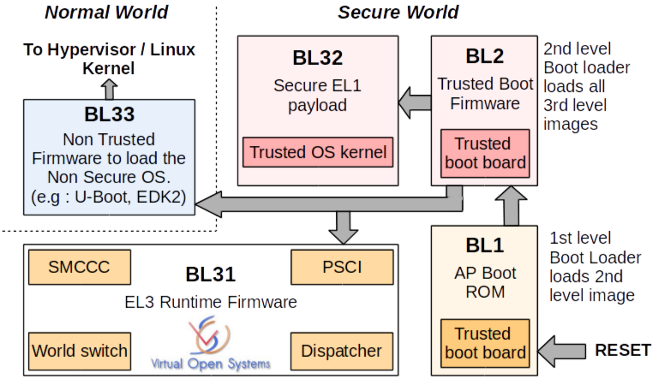
Secure boot



- Assures **integrity** (and **confidentiality**) of flash contents
- The **chain of trust** is similar to PKI[5] found in browsers
- One **root of trust** composed of immutable code and key

[5] Public Key Infrastructure

Secure boot in reality ...



Normal World

Secure World

To Hypervisor / Linux Kernel

BL33

Non Trusted Firmware to load the Non Secure OS. (e.g : U-Boot, EDK2)

BL32

Secure EL1 payload

Trusted OS kernel

BL2

Trusted Boot Firmware

Trusted boot board

2nd level Boot loader loads all 3rd level images

BL31

EL3 Runtime Firmware

SMCCC

PSCI

World switch

Dispatcher

BL1

AP Boot ROM

Trusted boot board

1st level Boot Loader loads 2nd level image

RESET

Key

EL3 Execution

Secure EL1 Execution

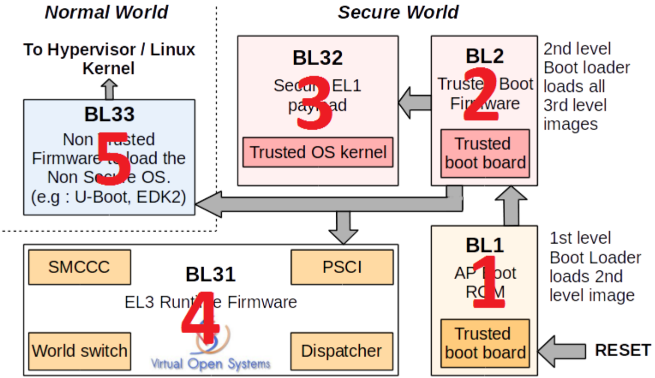
Normal EL2/EL1 Execution

Glossary

EDK2 – EFI Development Kit 2
EL – Exception Level
PSCI – Power State Control Interface
BL – Boot Loader
SMC – Secure Monitor Call

Source: http://community.arm.com/docs/DOC-9306

Secure boot in reality ...



Key

- EL3 Execution
- Secure EL1 Execution
- Normal EL2/EL1 Execution

Glossary

EDK2 – EFI Development Kit 2
EL – Exception Level
PSCI – Power State Control Interface
BL – Boot Loader
SMC – Secure Monitor Call

Source: http://community.arm.com/docs/DOC-9306

Why use a hardware attack?

”Logical issues exist in secure boot implementations!!?”

Bootloader vulnerabilities

- S5L8920 (iPhone)[6]
- Amlogic S905[7]

However

- A small code base results in a small logical attack surface
- Implementations without vulnerabililties likely exist

Other attack(s) must be used when not logically flawed!

[6] https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow

[7] http://www.fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html

Why use a hardware attack?

”Logical issues exist in secure boot implementations!!?”

Bootloader vulnerabilities

- S5L8920 (iPhone)[6]
- Amlogic S905[7]

However

- A small code base results in a small logical attack surface
- Implementations without vulnerabililties likely exist

Other attack(s) must be used when not logically flawed!

[6] https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow

[7] http://www.fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html

Why use a hardware attack?

”Logical issues exist in secure boot implementations!!?”

Bootloader vulnerabilities

- S5L8920 (iPhone)[6]
- Amlogic S905[7]

However

- A small code base results in a small logical attack surface
- Implementations without vulnerabililties likely exist

Other attack(s) must be used when not logically flawed!

[6] `https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow`

[7] `http://www.fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html`

Why use a hardware attack?

”Logical issues exist in secure boot implementations!!?”

Bootloader vulnerabilities

- S5L8920 (iPhone)[6]
- Amlogic S905[7]

However

- A small code base results in a small logical attack surface
- Implementations without vulnerabililties likely exist

Other attack(s) must be used when not logically flawed!

[6] https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow

[7] http://www.fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html

Why (not) fault injection on secure boot?

Cons

- Invasive
- Physical access
- Expensive

Pros

- No logical vulnerability required
- Typical targets not properly protected

*Especially **relevant** when **assets** are **not available** after boot!*

Why (not) fault injection on secure boot?

Cons

- Invasive
- Physical access
- Expensive

Pros

- No logical vulnerability required
- Typical targets not properly protected

*Especially **relevant** when **assets** are **not available** after boot!*

Why (not) fault injection on secure boot?

Cons

- Invasive
- Physical access
- Expensive

Pros

- No logical vulnerability required
- Typical targets not properly protected

*Especially **relevant** when **assets** are **not available** after boot!*

Why (not) fault injection on secure boot?

Cons

- Invasive
- Physical access
- Expensive

Pros

- No logical vulnerability required
- Typical targets not properly protected

*Especially **relevant** when **assets** are **not available** after boot!*

Typical assets

Secure code

- Boot code (ROM[8])

Secrets

- Keys (for boot code decryption)

Secure hardware

- Cryptographic engines

[8] Read Only Memory

Typical assets

Secure code

- Boot code (ROM[8])

Secrets

- Keys (for boot code decryption)

Secure hardware

- Cryptographic engines

[8] Read Only Memory

Typical assets

Secure code

- Boot code (ROM[8])

Secrets

- Keys (for boot code decryption)

Secure hardware

- Cryptographic engines

[8] Read Only Memory

Typical assets

Secure code

- Boot code (ROM[8])

Secrets

- Keys (for boot code decryption)

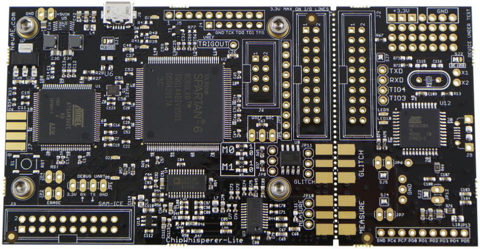
Secure hardware

- Cryptographic engines

[8] Read Only Memory

Open source tooling

ChipWhisperer



By NewAE Technology Inc. [9] [10]

[9] https://wiki.newae.com/CW1173_ChipWhisperer-Lite

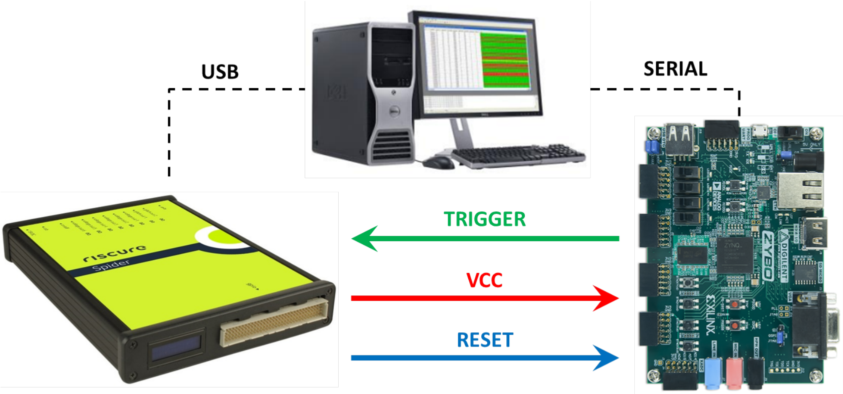
[10] https://www.youtube.com/watch?v=TeCQatNcF20

Commercial tooling

By Riscure[11]

[11] https://www.riscure.com/security-tools/hardware/spider

Fault injection setup



USB

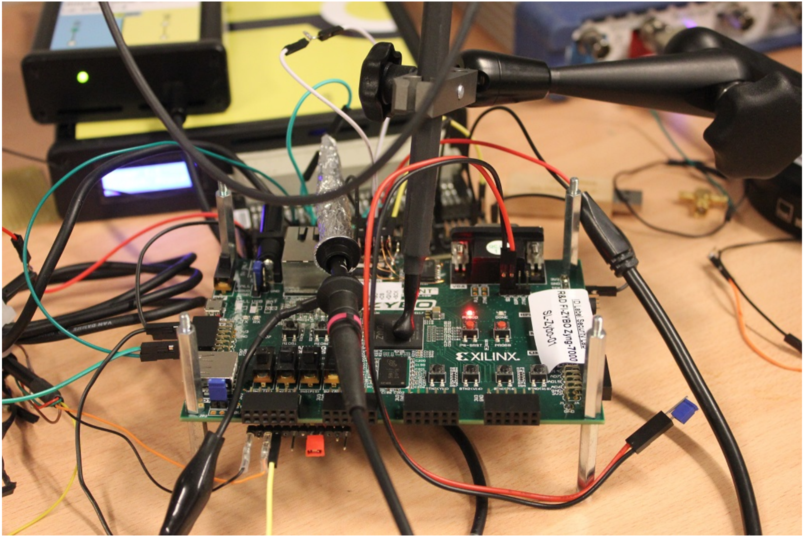
SERIAL

TRIGGER

VCC

RESET

In real life...


ID Labs Security Lab
R&D FI-ZYBO Zynq-7000
SL-Zybo-01
XILINX
ZYBO

That was the introduction ...

... let’s bypass secure boot!

That was the introduction ...

... let’s bypass secure boot!

Hash comparison

- Applicable to all secure boot implementations
- Bypass of authentication

```
if( memcmp( p, hash, hashlen ) != 0 )
    return( MBEDTLS_ERR_RSA_VERIFY_FAILED );

p += hashlen;

if( p != end )
    return( MBEDTLS_ERR_RSA_VERIFY_FAILED );

return( 0 );
```

Source: `https://tls.mbed.org/`

Hash comparison

- Applicable to all secure boot implementations
- Bypass of authentication

```
if( memcmp( p, hash, hashlen ) != 0 )
    return( MBEDTLS_ERR_RSA_VERIFY_FAILED );

p += hashlen;

if( p != end )
    return( MBEDTLS_ERR_RSA_VERIFY_FAILED );

return( 0 );
```

Source: `https://tls.mbed.org/`

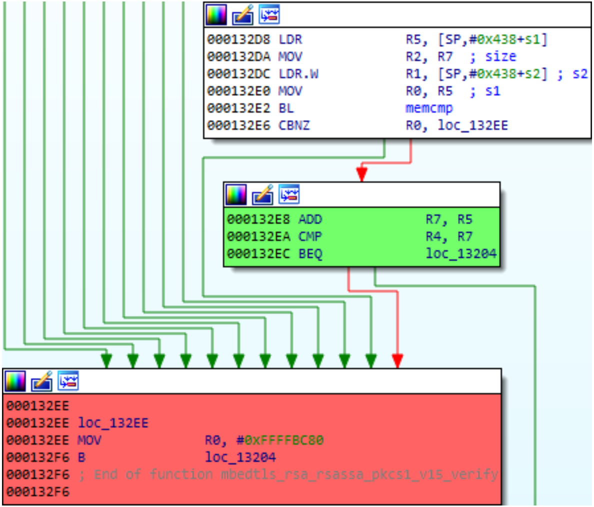
Hash comparison

- Applicable to all secure boot implementations
- Bypass of authentication

```
if( memcmp( p, hash, hashlen ) != 0 )
    return( MBEDTLS_ERR_RSA_VERIFY_FAILED );

p += hashlen;

if( p != end )
    return( MBEDTLS_ERR_RSA_VERIFY_FAILED );

return( 0 );
```

Source: `https://tls.mbed.org/`

Hash comparison



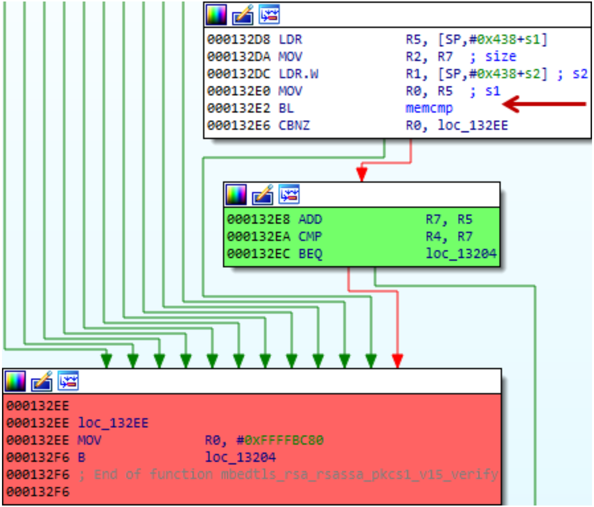
```
000132D8 LDR             R5, [SP,#0x438+s1]
000132DA MOV             R2, R7  ; size
000132DC LDR.W           R1, [SP,#0x438+s2] ; s2
000132E0 MOV             R0, R5  ; s1
000132E2 BL              memcmp
000132E6 CBNZ            R0, loc_132EE
```

```
000132E8 ADD             R7, R5
000132EA CMP             R4, R7
000132EC BEQ             loc_13204
```

```
000132EE
000132EE loc_132EE
000132EE MOV             R0, #0xFFFFBC80
000132F6 B               loc_13204
000132F6 ; End of function mbedtls_rsa_rsassa_pkcs1_v15_verify
000132F6
```

Multiple locations bypass the check with a single fault!

Hash comparison



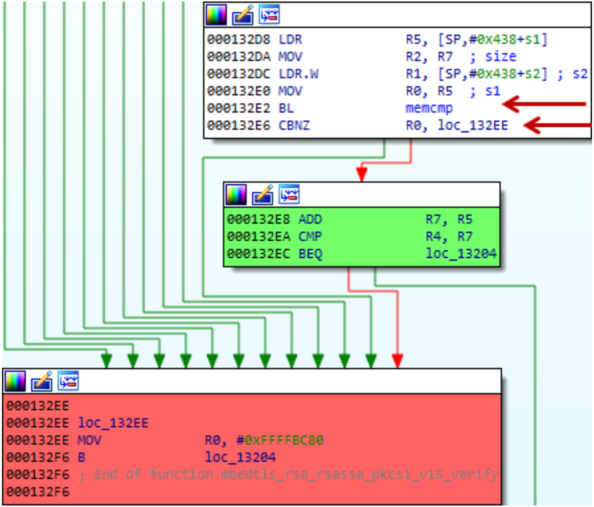
```
000132D8 LDR                    R5, [SP,#0x438+s1]
000132DA MOV                    R2, R7   ; size
000132DC LDR.W                  R1, [SP,#0x438+s2] ; s2
000132E0 MOV                    R0, R5   ; s1
000132E2 BL                     memcmp
000132E6 CBNZ                   R0, loc_132EE
```

```
000132E8 ADD                    R7, R5
000132EA CMP                    R4, R7
000132EC BEQ                    loc_13204
```

```
000132EE
000132EE loc_132EE
000132EE MOV                    R0, #0xFFFFBC80
000132F6 B                      loc_13204
000132F6 ; End of function mbedtls_rsa_rsassa_pkcs1_v15_verify
000132F6
```

Multiple locations bypass the check with a single fault!

Hash comparison



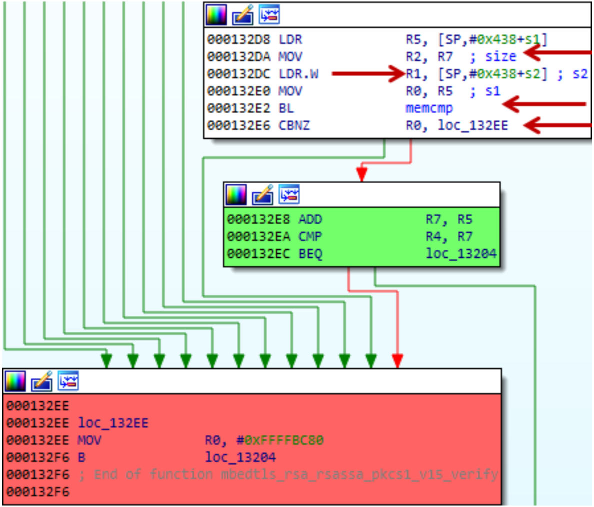
```
000132D8 LDR            R5, [SP,#0x438+s1]
000132DA MOV            R2, R7   ; size
000132DC LDR.W          R1, [SP,#0x438+s2] ; s2
000132E0 MOV            R0, R5   ; s1
000132E2 BL             memcmp
000132E6 CBNZ           R0, loc_132EE
```

```
000132E8 ADD            R7, R5
000132EA CMP            R4, R7
000132EC BEQ            loc_13204
```

```
000132EE
000132EE loc_132EE
000132EE MOV            R0, #0xFFFFBC80
000132F6 B              loc_13204
000132F6 ; End of function mbedtls_rsa_rsassa_pkcs1_v15_verify
000132F6
```

Multiple locations bypass the check with a single fault!

Hash comparison



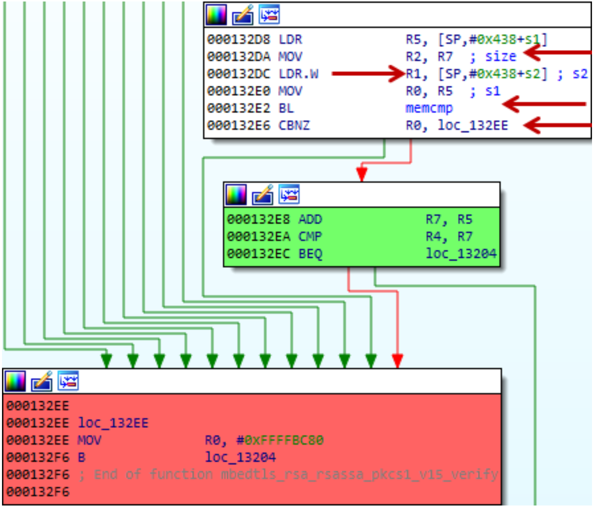
```
000132D8 LDR                 R5, [SP,#0x438+s1]
000132DA MOV                 R2, R7  ; size
000132DC LDR.W               R1, [SP,#0x438+s2] ; s2
000132E0 MOV                 R0, R5  ; s1
000132E2 BL                  memcmp
000132E6 CBNZ                R0, loc_132EE
```

```
000132E8 ADD                 R7, R5
000132EA CMP                 R4, R7
000132EC BEQ                 loc_13204
```

```
000132EE
000132EE loc_132EE
000132EE MOV                 R0, #0xFFFFBC80
000132F6 B                   loc_13204
000132F6 ; End of function mbedtls_rsa_rsassa_pkcs1_v15_verify
000132F6
```

Multiple locations bypass the check with a single fault!

Hash comparison



```
000132D8 LDR                  R5, [SP,#0x438+s1]
000132DA MOV                  R2, R7  ; size
000132DC LDR.W                R1, [SP,#0x438+s2] ; s2
000132E0 MOV                  R0, R5  ; s1
000132E2 BL                   memcmp
000132E6 CBNZ                 R0, loc_132EE
```

```
000132E8 ADD                  R7, R5
000132EA CMP                  R4, R7
000132EC BEQ                  loc_13204
```

```
000132EE
000132EE loc_132EE
000132EE MOV                  R0, #0xFFFFBC80
000132F6 B                    loc_13204
000132F6 ; End of function mbedtls_rsa_rsassa_pkcs1_v15_verify
000132F6
```

Multiple locations bypass the check with a single fault!

Signature check call

```
/* glitch here */
if(mbedtls_pk_verify(..., hash, signature, ...)) {
  /* do not boot up the image */
  no_boot();
} else {
  /* boot up the image */
  boot();
}
```

Remarks

- Bypasses can happen on all levels
- Inside functions, inside the calling functions, etc.

Signature check call

```
/* glitch here */
if(mbedtls_pk_verify(..., hash, signature, ...)) {
  /* do not boot up the image */
  no_boot();
} else {
  /* boot up the image */
  boot();
}
```

Remarks

- Bypasses can happen on all levels
- Inside functions, inside the calling functions, etc.

Signature check call

```
/* glitch here */
if(mbedtls_pk_verify(..., hash, signature, ...)) {
  /* do not boot up the image */
  no_boot();
} else {
  /* boot up the image */
  boot();
}
```

Remarks

- Bypasses can happen on all levels
- Inside functions, inside the calling functions, etc.

Infinite loop

- What to do when the signature verification fails?
- Enter an infinite loop!

```
/* glitch here */
if(mbedtls_pk_verify(..., hash, signature, ...)) {

  /* do not boot up the image */
  while(1);

} else {

  /* boot up the image */
  boot();
}
```

Infinite loop

- What to do when the signature verification fails?
- Enter an infinite loop!

```
/* glitch here */
if(mbedtls_pk_verify(..., hash, signature, ...)) {

  /* do not boot up the image */
  while(1);

} else {

  /* boot up the image */
  boot();
}
```

Infinite loop

- What to do when the signature verification fails?
- Enter an infinite loop!

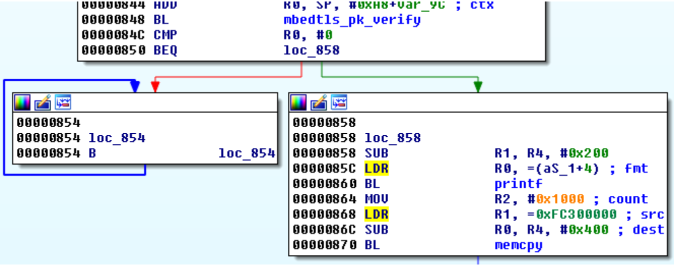
```
/* glitch here */
if(mbedtls_pk_verify(..., hash, signature, ...)) {

  /* do not boot up the image */
  while(1);

} else {

  /* boot up the image */
  boot();
}
```

Infinite loop

- What to do when the signature verification fails?
- Enter an infinite loop!

```
/* glitch here */
if(mbedtls_pk_verify(..., hash, signature, ...)) {

  /* do not boot up the image */
  while(1);

} else {

  /* boot up the image */
  boot();
}
```

Infinite loop



```
00000844 ADD                R0, SP, #0xA8+var_9C ; ctx
00000848 BL                 mbedtls_pk_verify
0000084C CMP                R0, #0
00000850 BEQ                loc_858
```

```
00000854
00000854 loc_854
00000854 B                  loc_854
```

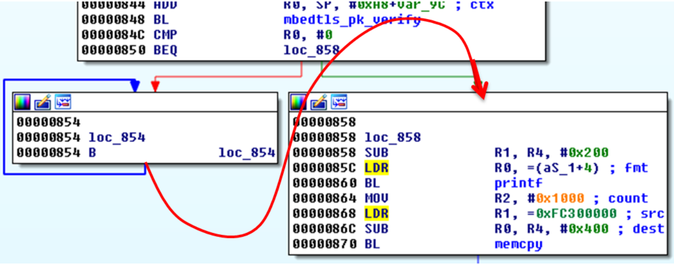
```
00000858
00000858 loc_858
00000858 SUB                R1, R4, #0x200
0000085C LDR                R0, =(aS_1+4) ; fmt
00000860 BL                 printf
00000864 MOV                R2, #0x1000 ; count
00000868 LDR                R1, =0xFC300000 ; src
0000086C SUB                R0, R4, #0x400 ; dest
00000870 BL                 memcpy
```

Remarks

- Timing is not an issue!
- Classic smart card attack [12]
- Better to reset or wipe keys

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

Infinite loop



```
00000844 ADD                     R0, SP, #0xA8+var_9C ; ctx
00000848 BL                      mbedtls_pk_verify
0000084C CMP                     R0, #0
00000850 BEQ                     loc_858
```

```
00000854
00000854 loc_854
00000854 B                 loc_854
```

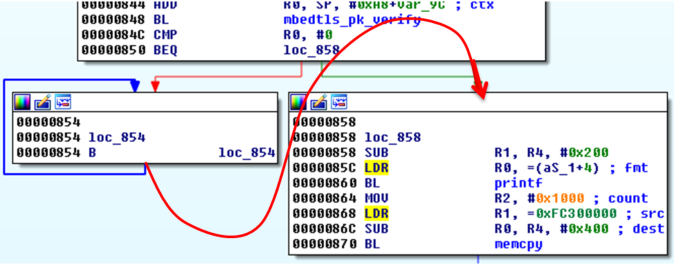
```
00000858
00000858 loc_858
00000858 SUB                     R1, R4, #0x200
0000085C LDR                     R0, =(aS_1+4) ; fmt
00000860 BL                      printf
00000864 MOV                     R2, #0x1000 ; count
00000868 LDR                     R1, =0xFC300000 ; src
0000086C SUB                     R0, R4, #0x400 ; dest
00000870 BL                      memcpy
```

Remarks

- Timing is not an issue!
- Classic smart card attack [12]
- Better to reset or wipe keys

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

Infinite loop



```
00000844 ADD                R0, SP, #0xA8+var_9C ; ctx
00000848 BL                 mbedtls_pk_verify
0000084C CMP                R0, #0
00000850 BEQ                loc_858
```

```
00000854
00000854 loc_854
00000854 B             loc_854
```

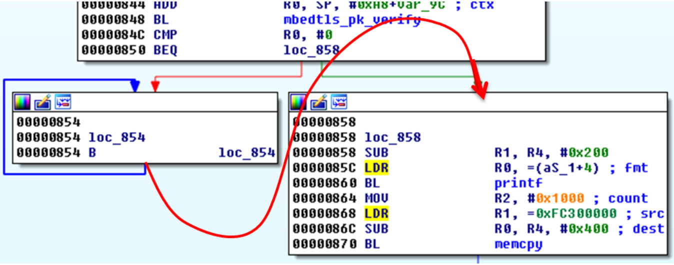
```
00000858
00000858 loc_858
00000858 SUB                R1, R4, #0x200
0000085C LDR                R0, =(aS_1+4) ; fmt
00000860 BL                 printf
00000864 MOV                R2, #0x1000 ; count
00000868 LDR                R1, =0xFC300000 ; src
0000086C SUB                R0, R4, #0x400 ; dest
00000870 BL                 memcpy
```

Remarks

- Timing is not an issue!
- Classic smart card attack [12]
- Better to reset or wipe keys

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

Infinite loop



```
00000844 ADD                R0, SP, #0xA8+var_9C ; ctx
00000848 BL                 mbedtls_pk_verify
0000084C CMP                R0, #0
00000850 BEQ                loc_858
```

```
00000854
00000854 loc_854
00000854 B                  loc_854
```

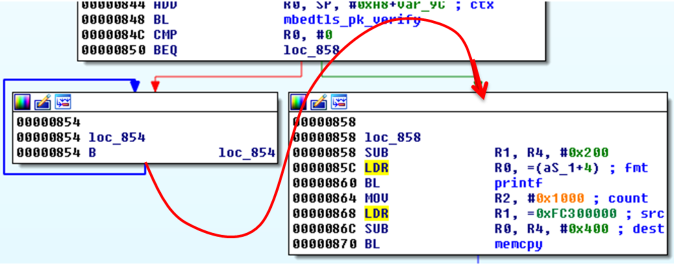
```
00000858
00000858 loc_858
00000858 SUB                R1, R4, #0x200
0000085C LDR                R0, =(aS_1+4) ; fmt
00000860 BL                 printf
00000864 MOV                R2, #0x1000 ; count
00000868 LDR                R1, =0xFC300000 ; src
0000086C SUB                R0, R4, #0x400 ; dest
00000870 BL                 memcpy
```

Remarks

- Timing is not an issue!
- Classic smart card attack [12]
- Better to reset or wipe keys

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

Infinite loop



```
00000844 ADD               R0, SP, #0xA8+var_9C ; ctx
00000848 BL                mbedtls_pk_verify
0000084C CMP               R0, #0
00000850 BEQ               loc_858
```

```
00000854
00000854 loc_854
00000854 B                 loc_854
```

```
00000858
00000858 loc_858
00000858 SUB               R1, R4, #0x200
0000085C LDR               R0, =(aS_1+4) ; fmt
00000860 BL                printf
00000864 MOV               R2, #0x1000 ; count
00000868 LDR               R1, =0xFC300000 ; src
0000086C SUB               R0, R4, #0x400 ; dest
00000870 BL                memcpy
```

Remarks

- Timing is not an issue!
- Classic smart card attack [12]
- Better to reset or wipe keys

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

Mitigations

Hardware countermeasures [13] [14]

- Detect the glitch or fault

Software countermeasures [15]

- Lower the probability of a successful fault
- Do not address the root cause

*You can **lower the probability** but not rule it out!*

[13] The Sorcerers Apprentice Guide to Fault Attacks – Bar-El et al., 2004

[14] The Fault Attack Jungle - A Classification Model to Guide You – Verbauwhede et al., 2011

[15] Secure Application Programming in the Presence of Side Channel Attacks – Witteman

Mitigations

Hardware countermeasures [13] [14]

- Detect the glitch or fault

Software countermeasures [15]

- Lower the probability of a successful fault
- Do not address the root cause

*You can **lower the probability** but not rule it out!*

[13] The Sorcerers Apprentice Guide to Fault Attacks – Bar-El et al., 2004

[14] The Fault Attack Jungle - A Classification Model to Guide You – Verbauwhede et al., 2011

[15] Secure Application Programming in the Presence of Side Channel Attacks – Witteman

Mitigations

Hardware countermeasures [13] [14]

- Detect the glitch or fault

Software countermeasures [15]

- Lower the probability of a successful fault
- Do not address the root cause

*You can **lower the probability** but not rule it out!*

[13] The Sorcerers Apprentice Guide to Fault Attacks – Bar-El et al., 2004

[14] The Fault Attack Jungle - A Classification Model to Guide You – Verbauwhede et al., 2011

[15] Secure Application Programming in the Presence of Side Channel Attacks – Witteman

Mitigations

Hardware countermeasures [13] [14]

- Detect the glitch or fault

Software countermeasures [15]

- Lower the probability of a successful fault
- Do not address the root cause

*You can **lower the probability** but not rule it out!*

[13] The Sorcerers Apprentice Guide to Fault Attacks – Bar-El et al., 2004

[14] The Fault Attack Jungle - A Classification Model to Guide You – Verbauwhede et al., 2011

[15] Secure Application Programming in the Presence of Side Channel Attacks – Witteman

Mitigations

Hardware countermeasures [13] [14]

- Detect the glitch or fault

Software countermeasures [15]

- Lower the probability of a successful fault
- Do not address the root cause

*You can **lower the probability** but not rule it out!*

[13] The Sorcerers Apprentice Guide to Fault Attacks – Bar-El et al., 2004

[14] The Fault Attack Jungle - A Classification Model to Guide You – Verbauwhede et al., 2011

[15] Secure Application Programming in the Presence of Side Channel Attacks – Witteman

Mitigations

Hardware countermeasures [13] [14]

- Detect the glitch or fault

Software countermeasures [15]

- Lower the probability of a successful fault
- Do not address the root cause

You can ***lower the probability*** *but not rule it out!*

[13] The Sorcerers Apprentice Guide to Fault Attacks – Bar-El et al., 2004

[14] The Fault Attack Jungle - A Classification Model to Guide You – Verbauwhede et al., 2011

[15] Secure Application Programming in the Presence of Side Channel Attacks – Witteman

Combined Attacks

Those were the classics and their mitigations ..

... the attack surface is larger![16]

[16] All attacks have been performed successfully on multiple targets!

Combined Attacks

Those were the classics and their mitigations ..

... the attack surface is larger![16]

[16] All attacks have been performed successfully on multiple targets!

Combined attack: Copy

- Introducing logical vulnerabilities using fault injection
  - Build your own buffer overflow!
- Easy approach: change *memcpy* the size argument

Before corruption

```
memcpy(dst, src, 0x1000);
```

After corruption

```
memcpy(dst, src, 0xCEE5);
```

Remark

- Works when dedicated hardware is used (e.g. DMA[17] engines)

[17] Direct Memory Access

Combined attack: Copy

- Introducing logical vulnerabilities using fault injection
  - Build your own buffer overflow!
- Easy approach: change *memcpy* the size argument

Before corruption

```
memcpy(dst, src, 0x1000);
```

After corruption

```
memcpy(dst, src, 0xCEE5);
```

Remark

- Works when dedicated hardware is used (e.g. DMA[17] engines)

[17] Direct Memory Access

Combined attack: Copy

- Introducing logical vulnerabilities using fault injection
  - Build your own buffer overflow!
- Easy approach: change *memcpy* the size argument

Before corruption

```
memcpy(dst, src, 0x1000);
```

After corruption

```
memcpy(dst, src, 0xCEE5);
```

Remark

- Works when dedicated hardware is used (e.g. DMA[17] engines)

[17] Direct Memory Access

Combined attack: Copy

- Introducing logical vulnerabilities using fault injection
  - Build your own buffer overflow!
- Easy approach: change *memcpy* the size argument

Before corruption

```
memcpy(dst, src, 0x1000);
```

After corruption

```
memcpy(dst, src, 0xCEE5);
```

Remark

- Works when dedicated hardware is used (e.g. DMA[17] engines)

[17] Direct Memory Access

Combined attack: Copy

- Introducing logical vulnerabilities using fault injection
  - Build your own buffer overflow!
- Easy approach: change *memcpy* the size argument

Before corruption

```
memcpy(dst, src, 0x1000);
```

After corruption

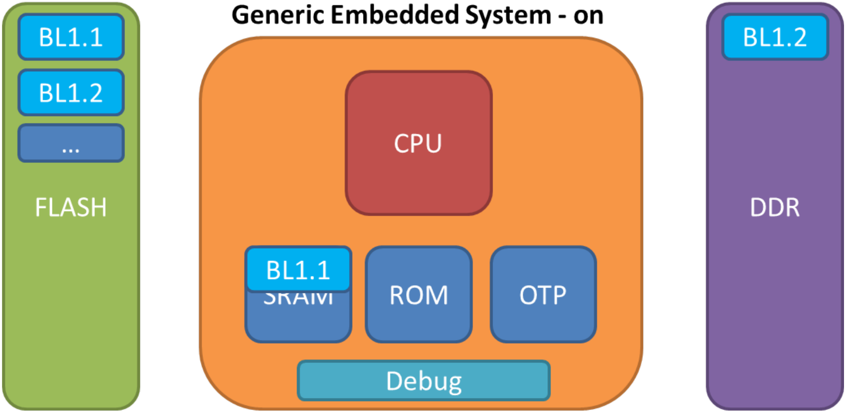
```
memcpy(dst, src, 0xCEE5);
```

Remark

- Works when dedicated hardware is used (e.g. DMA[17] engines)

[17] Direct Memory Access

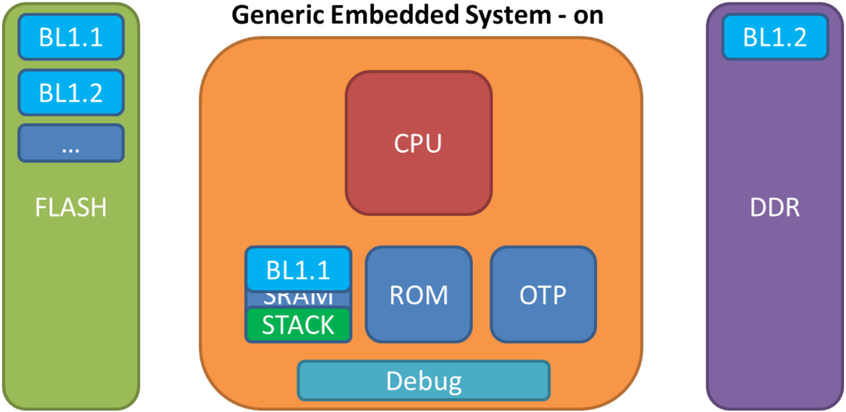
Combined attack: Copy



Remark

- Targeting the copy function arguments

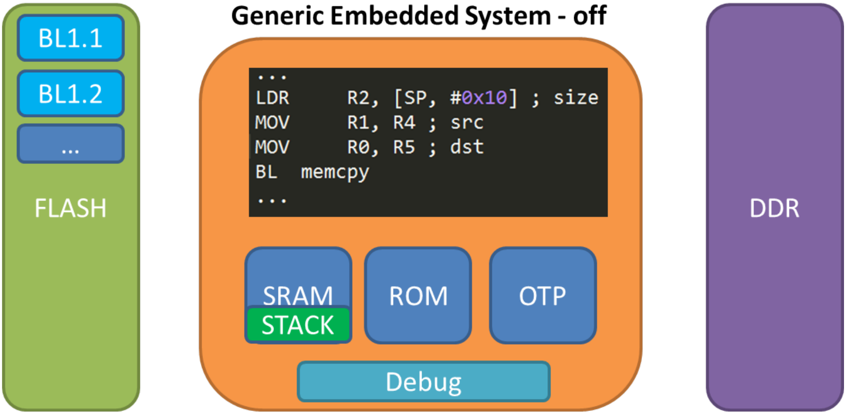
Combined attack: Copy



Remark

- Targeting the copy function arguments

Combined attack: Copy

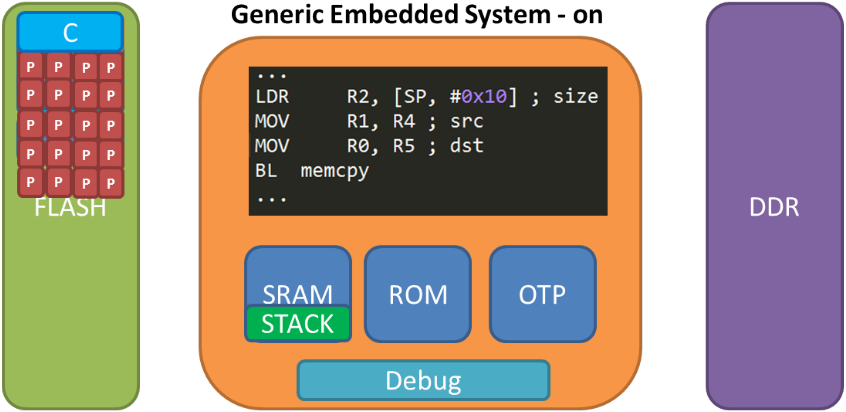


```
...
LDR     R2, [SP, #0x10] ; size
MOV     R1, R4 ; src
MOV     R0, R5 ; dst
BL   memcpy
...
```

Remark

- Targeting the copy function arguments

Combined attack: Copy

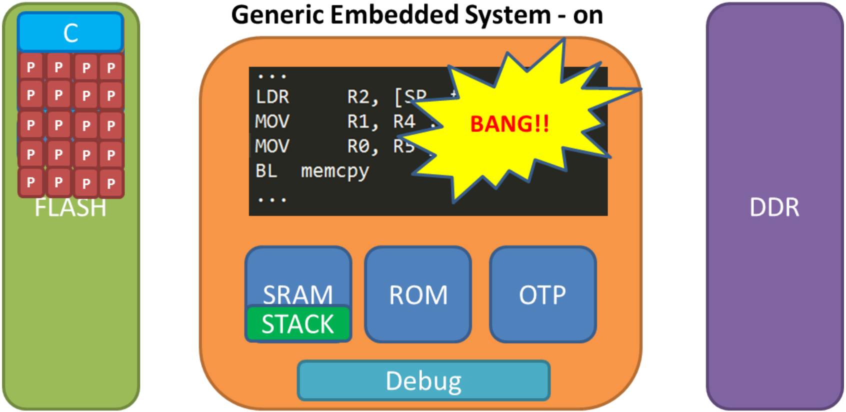


```
...
LDR     R2, [SP, #0x10] ; size
MOV     R1, R4 ; src
MOV     R0, R5 ; dst
BL  memcpy
...
```

Remark

- Targeting the copy function arguments

Combined attack: Copy

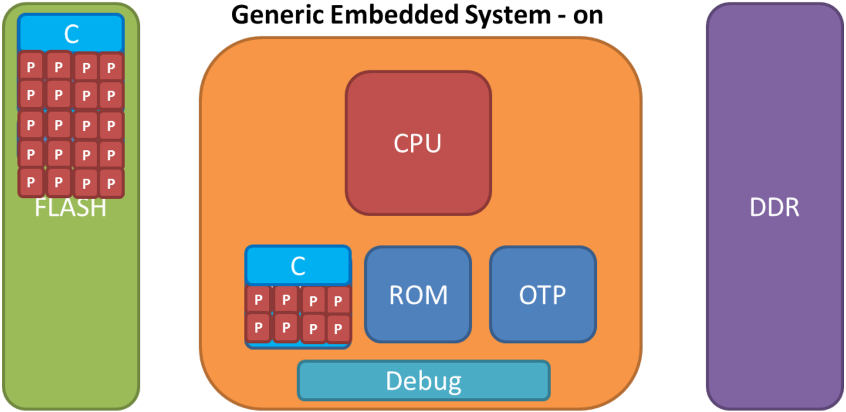


```
...
LDR      R2, [SP
MOV      R1, R4
MOV      R0, R5
BL   memcpy
...
```

Remark

- Targeting the copy function arguments

Combined attack: Copy



Remark

- Targeting the copy function arguments

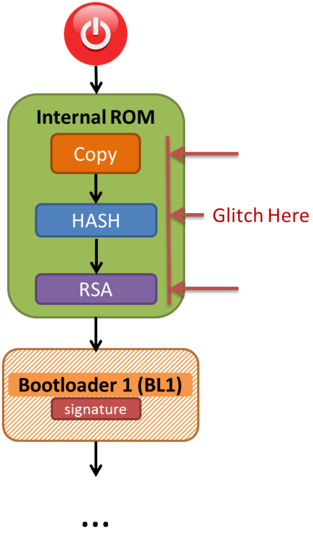
Combined attack: Wild jungle jump[18]

- Start glitching while/after loading the image but before decryption
- Lots of ’magic’ pointers around, which point close to the code
- Get them from: stack, register, memory
- The more magic pointers, the higher the probability

[18] Proving the wild jungle jump – Gratchoff, 2015

Combined attack: Wild jungle jump[18]

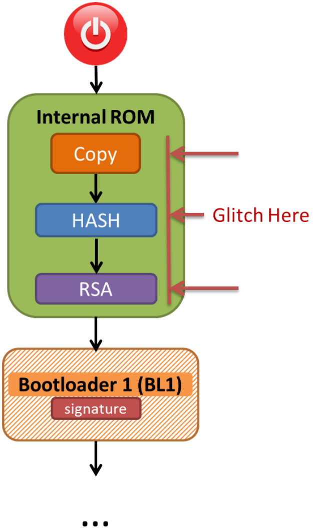
- Start glitching while/after loading the image but before decryption
- Lots of ’magic’ pointers around, which point close to the code
- Get them from: stack, register, memory
- The more magic pointers, the higher the probability



[18] Proving the wild jungle jump – Gratchoff, 2015

Combined attack: Wild jungle jump[18]

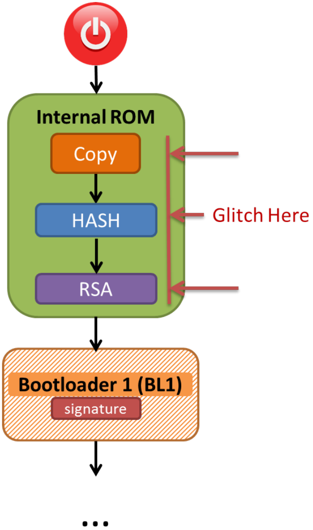
- Start glitching while/after loading the image but before decryption
- Lots of ’magic’ pointers around, which point close to the code
- Get them from: stack, register, memory
- The more magic pointers, the higher the probability



[18] Proving the wild jungle jump – Gratchoff, 2015

Combined attack: Wild jungle jump[18]

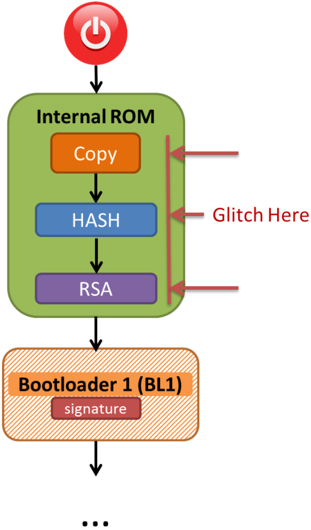
- Start glitching while/after loading the image but before decryption
- Lots of ’magic’ pointers around, which point close to the code
- Get them from: stack, register, memory
- The more magic pointers, the higher the probability



[18] Proving the wild jungle jump – Gratchoff, 2015

Combined attack: Wild jungle jump[18]

- Start glitching while/after loading the image but before decryption
- Lots of ’magic’ pointers around, which point close to the code
- Get them from: stack, register, memory
- The more magic pointers, the higher the probability



[18] Proving the wild jungle jump – Gratchoff, 2015

Combined attacks – Summary

- Bypass of both authentication and decryption
- Typically little software exploitation mitigation during boot
- Fault injection mitigations in software may not be effective

The possibilites are endless...

Combined attacks – Summary

- Bypass of both authentication and decryption
- Typically little software exploitation mitigation during boot
- Fault injection mitigations in software may not be effective

The possibilites are endless...

Combined attacks – Summary

- Bypass of both authentication and decryption
- Typically little software exploitation mitigation during boot
- Fault injection mitigations in software may not be effective

The possibilites are endless...

Combined attacks – Summary

- Bypass of both authentication and decryption
- Typically little software exploitation mitigation during boot
- Fault injection mitigations in software may not be effective

The possibilites are endless...

Combined attacks – Summary

- Bypass of both authentication and decryption
- Typically little software exploitation mitigation during boot
- Fault injection mitigations in software may not be effective

The possibilites are endless...

Attacker Practicalities

- Prepare the target
- Timing of the glitch
- Finding the right glitch shape
- Preparing the image

Attacker Practicalities

- Prepare the target
- Timing of the glitch
- Finding the right glitch shape
- Preparing the image

Attacker Practicalities

- Prepare the target
- Timing of the glitch
- Finding the right glitch shape
- Preparing the image

Attacker Practicalities

- Prepare the target
- Timing of the glitch
- Finding the right glitch shape
- Preparing the image

Attacker Practicalities

- Prepare the target
- Timing of the glitch
- Finding the right glitch shape
- Preparing the image

Conclusion

Today's standard technology not resistant to fault attacks

Minimize attack surface

- Authenticate all code and data
- Minimize code size of boot stages
- Drop privileges at an early stage

Lower the probability

- Implement fault injection countermeasures
- Implement software exploitation mitigations

*Robustness can only be determined using **testing**!*

Conclusion

Today's standard technology not resistant to fault attacks

Minimize attack surface

- Authenticate all code and data
- Minimize code size of boot stages
- Drop privileges at an early stage

Lower the probability

- Implement fault injection countermeasures
- Implement software exploitation mitigations

Robustness can only be determined using ***testing****!*

Conclusion

Today's standard technology not resistant to fault attacks

Minimize attack surface

- Authenticate all code and data
- Minimize code size of boot stages
- Drop privileges at an early stage

Lower the probability

- Implement fault injection countermeasures
- Implement software exploitation mitigations

*Robustness can only be determined using **testing**!*

Conclusion

Today's standard technology not resistant to fault attacks

Minimize attack surface

- Authenticate all code and data
- Minimize code size of boot stages
- Drop privileges at an early stage

Lower the probability

- Implement fault injection countermeasures
- Implement software exploitation mitigations

*Robustness can only be determined using **testing**!*

Conclusion

Today’s standard technology not resistant to fault attacks

Minimize attack surface

- Authenticate all code and data
- Minimize code size of boot stages
- Drop privileges at an early stage

Lower the probability

- Implement fault injection countermeasures
- Implement software exploitation mitigations

*Robustness can only be determined using **testing**!*

Conclusion

Today's standard technology not resistant to fault attacks

Minimize attack surface

- Authenticate all code and data
- Minimize code size of boot stages
- Drop privileges at an early stage

Lower the probability

- Implement fault injection countermeasures
- Implement software exploitation mitigations

*Robustness can only be determined using **testing**!*

Conclusion

Today’s standard technology not resistant to fault attacks

Minimize attack surface

- Authenticate all code and data
- Minimize code size of boot stages
- Drop privileges at an early stage

Lower the probability

- Implement fault injection countermeasures
- Implement software exploitation mitigations

*Robustness can only be determined using **testing**!*

Conclusion

Today's standard technology not resistant to fault attacks

Minimize attack surface

- Authenticate all code and data
- Minimize code size of boot stages
- Drop privileges at an early stage

Lower the probability

- Implement fault injection countermeasures
- Implement software exploitation mitigations

*Robustness can only be determined using **testing**!*

Conclusion

Today’s standard technology not resistant to fault attacks

Minimize attack surface

- Authenticate all code and data
- Minimize code size of boot stages
- Drop privileges at an early stage

Lower the probability

- Implement fault injection countermeasures
- Implement software exploitation mitigations

*Robustness can only be determined using **testing**!*

Conclusion

Today's standard technology not resistant to fault attacks

Minimize attack surface

- Authenticate all code and data
- Minimize code size of boot stages
- Drop privileges at an early stage

Lower the probability

- Implement fault injection countermeasures
- Implement software exploitation mitigations

*Robustness can only be determined using **testing**!*

riscure

Challenge your security

Niek Timmers

Senior Security Analyst

timmers@riscure.com (@tieknimmers)

Albert Spruyt

Senior Security Analyst

spruyt@riscure.com

www.riscure.com/careers

inforequest@riscure.com