Fault Injection Attacks on Secure Boot

Niek Timmers
timers@risecure.com
(@tieknimmers)

Albert Spruyt
spruyt@risecure.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

- 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
2. Voltage
3. Electromagnetic
4. 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

- Instruction corruption
  
  mov r0, r1  11100000110100000000000000000001
  mov r0, r3  1110000011010000000000000000001

- Instruction skipping
  
  mov r0, r1  11100000110100000000000000000001
  mov r0, r0  11100000110100000000000000000000

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

- Instruction corruption

\[
\begin{align*}
\text{mov r0, r1} & : 11100001101000000000000000000001 \\
\text{mov r0, r3} & : 11100001101000000000000000000011 \\
\text{mov r0, r0} & : 11100001101000000000000000000000
\end{align*}
\]

- Instruction skipping

\[
\begin{align*}
\text{mov r0, r1} & : 11100001101000000000000000000001 \\
\text{mov r0, r0} & : 11100001101000000000000000000000
\end{align*}
\]

Is this useful?

\[\text{2 Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells – Roscian et. al., 2015}\]
\[\text{3 High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures – Riviere et al., 2015}\]
\[\text{4 Formal verification of a software countermeasure against instruction skip attacks – Moro et. al., 2014}\]
Type of faults

Faults that affect hardware

- Registers
- Buses

Faults that affect hardware that does software

- 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

- Instruction corruption

```
mov r0, r1  11100001110100000000000000000001
mov r0, r3  11100001110100000000000000000011
```

- Instruction skipping

```
mov r0, r1  11100001110100000000000000000001
mov r0, r0  11100001110100000000000000000000
```

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

- Instruction corruption

\[
\begin{align*}
\text{mov } r0, r1 & \quad 111000011010000000000000000000001 \\
\text{mov } r0, r3 & \quad 11100001101000000000000000000011
\end{align*}
\]

- Instruction skipping

\[
\begin{align*}
\text{mov } r0, r1 & \quad 111000011010000000000000000000001 \\
\text{mov } r0, r0 & \quad 111000011010000000000000000000000
\end{align*}
\]

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

- Instruction corruption
  
  | mov r0, r1 | 11100001110100000000000000000001 |
  | mov r0, r3 | 11100001110101000000000000000001 |

- Instruction skipping
  
  | mov r0, r1 | 11100001110100000000000000000001 |
  | mov r0, r0 | 11100001110100000000000000000000 |

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

Remarks

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

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

- Integrity and confidentiality of flash contents are not assured!
- A mechanism is required for this assurance: **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\textsuperscript{5} found in browsers
- One root of trust composed of immutable code and key

\textsuperscript{5}Public Key Infrastructure
Secure boot

- Assures integrity (and confidentiality) of flash contents
- The chain of trust is similar to PKI\textsuperscript{5} found in browsers
- One root of trust composed of immutable code and key

\textsuperscript{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...

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 ...

**Normal World**

- To Hypervisor / Linux Kernel
- BL33
  - Non-Trust Firmware: Load the Non-Secure OS.
  - (e.g.: U-Boot, EDK2)
- BL31
  - EL3 Runtime Firmware
- SMCCC
- World switch

**Secure World**

- BL32
  - Secure EL1 payload
  - Trusted OS kernel
- BL32
  - Trusted Boot Firmware
  - Trusted boot board
- BL1
  - AP Boot ROM
  - Trusted boot board
- 1st level Boot Loader loads 2nd level image
- 2nd level Boot Loader loads all 3rd level images

**Key**

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

**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 vulnerabilities likely exist

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

---

\(^6\) [https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow](https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow)

\(^7\) [http://www.fredericb.info/2016/10/amlogic-s905-soc-bypassing-not-so.html](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 vulnerabilities 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)\textsuperscript{6}
- Amlogic S905\textsuperscript{7}

However

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

\textit{Other attack(s) must be used when not logically flawed!}

\textsuperscript{6} https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow
\textsuperscript{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 vulnerabilities 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\textsuperscript{8})

Secrets
- Keys (for boot code decryption)

Secure hardware
- Cryptographic engines
Typical assets

Secure code
- Boot code (ROM\textsuperscript{8})

Secrets
- Keys (for boot code decryption)

Secure hardware
- Cryptographic engines

\textsuperscript{8} Read Only Memory
Typical assets

Secure code
  • Boot code (ROM\textsuperscript{8})

Secrets
  • Keys (for boot code decryption)

Secure hardware
  • Cryptographic engines

\textsuperscript{8} Read Only Memory
Typical assets

Secure code
- Boot code (ROM\textsuperscript{8})

Secrets
- Keys (for boot code decryption)

Secure hardware
- Cryptographic engines

\textsuperscript{8} Read Only Memory
Open source tooling

ChipWhisperer

By NewAE Technology Inc.  

9 10

https://wiki.newae.com/CW1173_ChipWhisperer-Lite
https://www.youtube.com/watch?v=TeCQatNcF20
Fault injection setup
In real life...
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

```c
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

```c
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/](https://tls.mbed.org/)
Hash comparison

Multiple locations bypass the check with a single fault!
Hash comparison

Multiple locations bypass the check with a single fault!
Hash comparison

Multiple locations bypass the check with a single fault!
Multiple locations bypass the check with a single fault!
Hash comparison

Multiple locations bypass the check with a single fault!
Signature check call

```c
/* 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

```c
/* 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

```c
/* 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!

```c
/* 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!

```c
/* 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!

```c
/* 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!

```c
/* glitch here */
if (mbedtls_pk_verify(..., hash, signature, ...)) {
    /* do not boot up the image */
    while (1);
}
else {
    /* boot up the image */
    boot();
}
```
Infinite loop

Remarks

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

Infinite loop

Remarks

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

Infinite loop

Remarks

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

Infinite loop

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

Remarks

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

Mitigations

Hardware countermeasures

- Detect the glitch or fault

Software countermeasures

- 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

- Detect the glitch or fault

Software countermeasures

- 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

• Detect the glitch or fault

Software countermeasures

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

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

Hardware countermeasures

- Detect the glitch or fault

Software countermeasures

- 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
15. Secure Application Programming in the Presence of Side Channel Attacks – Witteman
Mitigations

Hardware countermeasures

- Detect the glitch or fault

Software countermeasures

- 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

- Detect the glitch or fault

Software countermeasures

- 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!\textsuperscript{16}

\textsuperscript{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**

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

**After corruption**

```c
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

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

After corruption

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

Remark

- Works when dedicated hardware is used (e.g. DMA\(^\text{17}\) engines)

\(^\text{17}\) 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

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

After corruption

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

Remark

- Works when dedicated hardware is used (e.g. DMA\textsuperscript{17} engines)

\textsuperscript{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**

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

**After corruption**

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

**Remark**

- Works when dedicated hardware is used (e.g. DMA\(^\text{17}\) engines)

---

\(^\text{17}\) 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

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

After corruption

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

Remark

- Works when dedicated hardware is used (e.g. DMA\textsuperscript{17} engines)

---

\textsuperscript{17} Direct Memory Access
Combined attack: Copy

Remark
- Targeting the copy function arguments
Combined attack: Copy

Generic Embedded System - on

Remark
- Targeting the copy function arguments
Combined attack: Copy

Generic Embedded System - off

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

Remark
- Targeting the copy function arguments
Combined attack: Copy

Generic Embedded System - on

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

Remark
• Targeting the copy function arguments
Combined attack: Copy

Remark
- Targeting the copy function arguments
Remark

- Targeting the copy function arguments
Combined attack: Wild jungle jump

- 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

Proving the wild jungle jump – Gratchoff, 2015
Combined attack: Wild jungle jump

- 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

- 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

- 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

- 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

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 possibilities 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 possibilities 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 possibilities 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 possibilities 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 possibilities 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!*
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