arm

Physical Attack Mitigation

Linaro Virtual Connect 2021

Raef Coles - Graduate Engineer Arm Ltd.
Tamas Ban - Staff Engineer Arm Ltd.

© 2021 Arm

2021
Agenda

• Physical attack overview
• Software countermeasures
• MCUboot overview
• SW countermeasures in MCUboot
• SW countermeasures in TF-M runtime
• QEMU based test tool
• Further verification
A high-level view on fault injection

- A fault is physical perturbation altering the correct / expected behaviour of a circuit.
- It can be a change in voltage or temperature, or a laser beam, or an EM pulse,... All have different effects.
- Effect can be permanent (damage) or transient
- Physical access is **not** always needed
  - rowhammer or clkscrew for example
- Strongly correlated with reliability:
  - Reliability is about “random” hazards
  - Fault injection is about an adversary actively introducing hazards

Figure from “Fault Attacks on Secure Embedded Software: Threats, Design and Evaluation”, Bilgiday Yuce, Patrick Schaumont, Marc Witteman

Slide from Arnaud De Grandmaison
A high-level view on fault injection (cont.)

• This is a complex domain!
  • Faults are not well understood
  • This is an active (but niche) research domain

• All models are wrong --- but each one address a specific aspect of some observed faults and is thus useful

• Ultimately it’s all about using different models to explore and reason about the unknown / complex All observable faults
Software countermeasures

- The objective is to **protect against data and control-flow tampering**.
- There are **dedicated hardware** components which can provide a level of protections, but there is an additional level of **defence provided by software** countermeasures—**defence-in-depth approach**.
- Although **there is no way to guarantee defence** from those attacks neither by hardware nor by software, the more countermeasures there are in place, the harder attacks are.
- There are practical techniques that can be applied to the coding and **significantly decrease the probability of successful attacks**.
Generic countermeasures

• SW mitigation techniques against physical attacks
  • **Complex (large hamming distance) constants**: More bits need to be flipped to change one valid value to another.
  • **Double checks, switch/case double checks**: Make it harder to attack the branch conditions by checking the same condition twice. Can actively detect tampering attempts if there is inconsistency.
  • **Loop integrity checks**: Make sure important loops are executed, check expected index value after the loop.
  • **Default failure**: Skipping instructions or attacking PC can bypass important code. Default value of checks and branches is the failure case.
  • **Flow monitor**: The state of the program is tracked, and its expected value checked to make sure that the program is not in an unexpected state.

• Good resources in the topic:
  • https://www.cl.cam.ac.uk/~rja14/Papers/whatyouc.pdf
Attack vs. Protection

How to perform an attack?

• While physical attacks appear to be a mystery, there are many, many resources available on how to perform them.
• There are commercial tools to break devices with fault injection: Chip Whispering.
• Software frameworks that are compatible with commercial devices lower barrier to entry.
• Offensive devices are low-cost

Hmmm...seems easier to attack than protect against...??

How to be protected?

• Generic solutions do not exist.
• Many research papers, but not much practical info.
• No tested open-source solution, no compiler support.
• Certified products contain usually proprietary and closed-source code.
• Compiled code depends on actual compiler, optimization level, architecture, etc.
• **Compiled code must be verified.** On C level seems safe, but the binary might not...
Why MCUboot is hardened primarily?

- TF-M consist of (roughly):
  - Secure boot code: MCUboot
  - Runtime secure firmware: Secure partition manager & Secure partitions

- With right timing the image authentication can be bypassed. If arbitrary image can be executed then all secrets could be disclosed from the device.

- Secure boot code has a time deterministic execution. With physical access it is easy to try thousands of attempts to determine the correct timing.

- There are vulnerable function calls from physical attack point of view in the boot flow.

```c
rc = boot_go(&rsp);
if (rc != 0) {
    BOOT_LOG_ERR("Unable to find bootable image");
    while (1) ;
}
do_boot();
```

- Reset register
- Skip instructions
- Reset zero flag in status reg.
- Jump out from error loop
MCUboot overview

- Designed for 32bit MCUs
- Low memory footprint (~18KB of ROM)
- Compatible with several crypto library (mbedTLS, tinyCrypt)
- RSA, ECDSA support
- Encrypted image support
- Custom image manifest format (TLV)
- No X.509 support, No SUIT manifest support (yet)
- No fault injection and minimal side channel attack protection (Until now)
Boot flow

Original boot flow

Hardened boot flow

Secure image

Unhardened code

Crypto-lib
What has been produced

- Created a generic fault injection hardening library
- Integrated this library into MCUboot, and upstreamed the changes
- Created a QEMU-based fault injection tester, and upstreamed it to MCUboot
- Integrated the library into the TF-M runtime (WIP)
- Created an improved fault injection tester, to handle the increased complexity in testing the TF-M runtime (WIP)
SW countermeasures in MCUboot

- Primitives added to harden existing code
- Only added to critical code path
- Build time configurable, 4 profiles available (HIGH, MEDIUM, LOW, OFF)
- [https://github.com/mcu-tools/mcuboot/pull/776](https://github.com/mcu-tools/mcuboot/pull/776)

<table>
<thead>
<tr>
<th>Countermeasure</th>
<th>Status</th>
<th>Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control flow integrity</td>
<td>Implemented</td>
<td>LOW</td>
</tr>
<tr>
<td>Failure loop hardening</td>
<td>Implemented</td>
<td>LOW</td>
</tr>
<tr>
<td>Complex constants</td>
<td>Implemented</td>
<td>MEDIUM</td>
</tr>
<tr>
<td>Redundant variables and checks</td>
<td>Implemented</td>
<td>MEDIUM</td>
</tr>
<tr>
<td>Random delay</td>
<td>Implemented, but depends on device entropy capability.</td>
<td>HIGH</td>
</tr>
<tr>
<td>Loop integrity checks</td>
<td>Not implemented</td>
<td>-</td>
</tr>
</tbody>
</table>
Countermeasure details

- Generic implementation of most of the known countermeasures for fault injection.
- Written as a C library.
  - 2 headers + 1 source file, with extras for TRNG support.
  - Can't write in ASM as open-source projects may use multiple architectures / architecture versions.
- Code changes to enable library are minimized, all points of vulnerability (functions calls, variable tests) have a drop-in replacement.
- Code size increase with all countermeasures disabled is only 250 bytes, with status variables being returned to their standard types and checks to standard patterns.
- Assembly verified under GCC and ARMClang, although this may break with future versions.
- Not a total solution, but a good option for protection requiring minimal effort from integrators, and much better than nothing.
- Can be easily adapted into other projects because of generic implementation and open-source license.
Critical call path hardening

rc = boot_go(&rsp);
if (rc != 0) {
    BOOT_LOG_ERR("Unable to find bootable image");
    while (1)
        ;
}

FIH_CALL(boot_go, fih_rc, &rsp);
if (fih_not_eq(fih_rc, FIH_SUCCESS)) {
    BOOT_LOG_ERR("Unable to find bootable image");
    FIH_PANIC;
}

#define FIH_CALL(f, ret, ...) \
    do { \ 
        FIH_LABEL("START"); \ 
        FIH_CFI_PRECALL_BLOCK; \ 
        ret = FIH_FAILURE; \ 
        if (fih_delay()) { \ 
            ret = f(__VA_ARGS__); \ 
        } \ 
        FIH_CFI_POSTCALL_BLOCK; \ 
        FIH_LABEL("END"); \ 
    } while (0)

__attribute__((always_inline)) inline
int fih_not_eq(fih_int x, fih_int y)
{
    fih_int_validate(x);
    fih_int_validate(y);
    return (x.val != y.val) && fih_delay() && (x.msk != y.msk);
}

typedef volatile struct {
    volatile int val;
    volatile int msk;
} fih_int;
Countermeasure Performance

- Small (< 2.6k), and configurable impact on code size
- Good performance in defending against skip faults

<table>
<thead>
<tr>
<th>Profile</th>
<th>Image size</th>
<th>Executed tests</th>
<th>Boots with corrupted image</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCUBOOT_FIH_PROFILE_OFF</td>
<td>Flash: 25.1 kB, RAM: 25.4 kB</td>
<td>520</td>
<td>26 (5%)</td>
</tr>
<tr>
<td>MCUBOOT_FIH_PROFILE_LOW</td>
<td>Flash: 25.5 kB, RAM: 25.4 kB</td>
<td>820</td>
<td>15 (1.8%)</td>
</tr>
<tr>
<td>MCUBOOT_FIH_PROFILE_MEDIUM</td>
<td>Flash: 27.7 kB, RAM: 25.4 kB</td>
<td>1205</td>
<td>7 (0.5%)</td>
</tr>
</tbody>
</table>
Testing tool (V1)

- [https://github.com/mcu-tools/mcuboot/pull/789](https://github.com/mcu-tools/mcuboot/pull/789)
- Executes in QEMU, using the mps2-an521 machine
- Attaches GDB as a debugger
- Simulates faults using the debugger
  - Skipping instructions by modifying the program counter
- Evaluates the success of the faults based on whether the bootloader succeeds in booting an image with an invalid signature
SW countermeasures in TF-M runtime

- Same FIH library (fault injection hardening) is used.
- Goal is to protect part of the code which is critical to properly configure the hardware components that are responsible for memory isolation.
- Design proposal:
  - Physical attack mitigation in Trusted Firmware-M
- Prototype implementation:
  - https://review.trustedfirmware.org/c/TF-M/trusted-firmware-m/+/8544/
- Enhanced QEMU based test environment
  - Faster execution via QEMU state save/load
  - Support for detecting changes in important memory regions
  - Extended output reports
Testing tool (V2)

- [https://review.trustedfirmware.org/c/TF-M/tf-m-tools/+/9072](https://review.trustedfirmware.org/c/TF-M/tf-m-tools/+/9072)
- Controls the testing using python scripts
- Executes the program without faults, to determine the correct state
- Uses QEMU state-saving to speed up execution of tests
- Simulates a wider range of faults, with support for easily adding more
  - Setting registers to random values
  - Setting registers to 0
- Evaluates the success of the faults based on the difference between the fault state and the correct state, and whether any failure handlers have been triggered
- Uses labels in the code to determine places at which critical memory might impact security, and uses those to determine when program state should be evaluated
- Outputs JSON, containing information about the fault and the state changes it caused.
Further verification

• A 6-months Arm internship just started to test and evaluate the added SW countermeasures in MCUboot:
  • Collaboration with Sorbonne University, other experts will be involved as well.
  • Results to be available mid Q3

• Testing is planned to be done:
  • First in simulator environments
  • Depending on the progress, might move to testing on real hardware

• Expectation is to get an external feedback on the implementation quality
  • General assessment by experts in physical attacks
  • Findings can be used in certification process, which requires resistance against physical attacks
  • Could be used for enhancements
Resources

• White papers
  • https://www.cl.cam.ac.uk/~rja14/Papers/whatyouc.pdf

• Implementation in MCUboot
  • https://github.com/mcu-tools/mcuboot/pull/776

• Implementation in TF-M runtime
  • Physical attack mitigation in Trusted Firmware-M
  • https://review.trustedfirmware.org/c/TF-M/trusted-firmware-m/+/8544/

• Developed test tools
  • https://github.com/mcu-tools/mcuboot/pull/789
  • https://review.trustedfirmware.org/c/TF-M/tf-m-tools/+/9072
Thank You
Danke
Gracias
谢谢
ありがとうございます
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה