Simulate Invalid Opcode

以龙芯二代 CPU 为核心的工作站(福珑电脑)在 2006 年底以试用的方式首发 1500 台至市场,国内 Linux geeker 也是福珑电脑的首批试用者在网上也贴出了一些专为福珑电脑的定制化 Linux 内核.看到了不觉眼馋,同时又想起差不多 5 年前龙芯一代 CPU 刚完成后,总设计师在网上发的一篇贴子,其中提到龙芯虽然是 MIPS 架构,但在指令集上并不是 100%兼容的.设计师提到有几条指令由于 MIPS 公司申请了专利,所以龙芯不能实现.设计师提到对这几条指令不能在 CPU 级支持,只能在 OS kernel 级(软件级)支持.

春节假期闲来无事,又对龙芯 CPU 很感兴趣,便想如何回答设计师的问题.由于下面两个原因,所以我只能用 x86 架构来做实验---当然原理是一样的。

1. 我现在没有福珑电脑,我更没有 SGI 的基于 MIPS CPU 的工作站
2. 对 MIPS CPU 处于学习阶段,还一知半解

其实这个问题的大致思路还是蛮简单的.

想象一下如下场景:

一个被编译为 MIP CPU 上运行的 application 恰恰含有龙芯 CPU 不支持的指令(比如 opcode_a),当我们把该 application 在福珑电脑上运行时,龙芯 CPU 解码到 opcode_a 时,会认为这是一条非法 opcode,CPU 会马上切换到非法指令异常(exception),这时候 OS 就有机会来弥补 CPU 不认识该指令的缺陷,执行原本该指令应有的动作,在完成工作后, OS 同时必须使 CPU 跳过该 opcode,然后让 application 从该 opcode 后继续运行.从 application 的角度而言,它甚至根本不会意识到发生的这一切.

下面是在 x86 CPU 上模拟上面的动作.

1. 先是要构造一条 invalid opcode(非法指令)

我选了如下的指令(实际上是我 8 年前做一个项目时的选择)

lock movl %ecx, %eax

这里我用的是 AT&T 格式的汇编语法,如果用 Intel 格式(Microsoft 用的就是 Intel 格式),那就是下面的

lock mov eax, ecx

二进制是 0xF089C8(3 bytes)至于为什么这是一条非法指令,具体解释去看 Intel CPU Manual 对 lock 的解释.这里只提一句,lock prefix 不能用在 register,这是与它的本意相违的.

该指令要实现乘方的功能，比如 eax = 3, ecx = 5，则执行该指令后
*eax = eax ** ecx = 3 * 3 * 3 * 3 * 3 = 243.*

2. 测试案例

按照 XP 的开发步骤,先设计测试案例.
由于该 simulation 必须能同时处理用户态和核心态两种情况,所以测试案例也有两个.

- **用户态的 application**

```
$cat invalid_op.cpp
#include <iostream>
int main()
{
        int    x;
        int    y;
        int    z;

        std::cout << "x ** y = ?" << std::endl;
        std::cout << "x : ";
        std::cin >> x;
        std::cout << "y : ";
        std::cin >> y;

        asm("movl %1, %%eax\n\t"
        "movl %2, %%ecx\n\t"
        "lock movl %%ecx, %%eax\n\t"                    //这就是非法指令
        "movl %%eax, %0\n\t"
        :"=r"(z)
        :"r"(x), "r"(y)
        :"%eax", "%ecx"
        );

        std::cout << x << " ** " << y << " = " << z << std::endl;

        return      0;
}

$g++ -o invalid_op invalid_op.cpp
$./invalid_op
x ** y = ?
```

x : 3
y : 5
Illegal instruction　　<=== OK, CPU 不认识非法指令,OS 打印出该提示后,杀死该进程

- 核心态的最简单驱动(LKM---Loadable Kernel Module)

```
$cat invalid_op.c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Walter Zhou");

static long     x = 1;
static long     y = 0;

MODULE_PARM(x, "l");
MODULE_PARM(y, "l");

MODULE_PARM_DESC(x, "This is x");
MODULE_PARM_DESC(y, "This is y.");

static int __init invalid_op_init(void)
{
    long result = 0;

    asm("movl %1, %%eax\n\t"
    "movl %2, %%ecx\n\t"
    "lock movl %%ecx, %%eax\n\t"            /*  这就是非法指令  */
    "movl %%eax, %0\n\t"
    :"=r"(result)
    :"r"(x), "r"(y)
    :"%eax", "%ecx"
    );

    printk(KERN_ALERT "%ld ** %ld = %ld", x, y, result);
    return 0;
}

static void __exit invalid_op_exit(void)
{
    printk(KERN_ALERT "Exit invalid_op module\n");
}
```

```
module_init(invalid_op_init);
module_exit(invalid_op_exit);
```

Makefile 如下

$cat Makefile

```
WARN        := -W -Wall -Wstrict-prototypes -Wmissing-prototypes
INCLUDE         := -isystem /usr/src/linux-`uname -r`/include
CFLAGS      := -O2 -DMODULE -D__KERNEL__ ${WARN} ${INCLUDE}
CC      := gcc
OBJS        := ${patsubst %.c, %.o, ${wildcard *.c}}

all: ${OBJS}

.PHONY: clean

clean:
    rm -rf *.o
```

```
$su root            //只有 root 才能载入 LKM
#/sbin/insmod invalid_op.o x=3 y=5
...(应该输出什么东西呢?)
```

你将看到的是系统 crash.因为非法指令发生在核心态,OS 也没办法,只能杀死系统本身.

3. Patch Linux OS kernel

当 CPU 发觉是非法指令后,它将陷入 invalid opcode exception,该 exception 的编号是 6.在 i386 架构下处理器为 arch/i386/kernel/trap.c 文件中的 do_trap() handler.这是多个 system exception 的入口,比如 invalid opcode exception, page fault exception 等
原来的代码如下

```
static void inline do_trap(int trapnr, int signr, char *str, int vm86,
               struct pt_regs * regs, long error_code, siginfo_t *info)
{
    if (regs->eflags & VM_MASK) {
        if (vm86)
            goto vm86_trap;
        else
            goto trap_signal;
    }
```

```
    if (!(regs->xcs & 3))
        goto kernel_trap;

    trap_signal: {
        struct task_struct *tsk = current;
        tsk->thread.error_code = error_code;
        tsk->thread.trap_no = trapnr;
        if (info)
            force_sig_info(signr, info, tsk);
        else
            force_sig(signr, tsk);
        return;
    }

    kernel_trap: {
        unsigned long fixup = search_exception_table(regs->eip);
        if (fixup)
            regs->eip = fixup;
        else
            die(str, regs, error_code);
        return;
    }

    vm86_trap: {
        int ret = handle_vm86_trap((struct kernel_vm86_regs *) regs,
    error_code, trapnr);
        if (ret) goto trap_signal;
        return;
    }
}
```

下面是添加了 invalid opcode 处理后的 do_trap() handler.

```
static void inline do_trap(int trapnr, int signr, char *str, int vm86,
              struct pt_regs * regs, long error_code, siginfo_t *info)
{
    long i;
    long tmp;
    if(trapnr == 6) {
        unsigned char *eip = (unsigned char *)(regs->eip);
        if(eip[0] == 0xF0 && eip[1] == 0x89 && eip[2] == 0xC8) {
            /* skip the invalid opcode (lock %ecx, %eax) */
            regs->eip += 3;
```

```
        /* I assume regs->ecx >= 0 */
        if(regs->ecx == 0) {
            regs->eax = 1;
        } else if(regs->ecx == 1) {
            /* do nothing */
        } else {
            tmp = regs->eax;

            for(i = 0; i < regs->ecx - 1; ++i) {
                regs->eax *= tmp;   // *I skip the overflow bug */
            }
        }
        /* ok, we already handle the invalid opcode, return */
        return;
    }
}

if (regs->eflags & VM_MASK) {
    if (vm86)
        goto vm86_trap;
    else
        goto trap_signal;
}

if (!(regs->xcs & 3))
    goto kernel_trap;

trap_signal: {
    struct task_struct *tsk = current;
    tsk->thread.error_code = error_code;
    tsk->thread.trap_no = trapnr;
    if (info)
        force_sig_info(signr, info, tsk);
    else
        force_sig(signr, tsk);
    return;
}

kernel_trap: {
    unsigned long fixup = search_exception_table(regs->eip);
    if (fixup)
        regs->eip = fixup;
    else
```

```
            die(str, regs, error_code);
        return;
    }

    vm86_trap: {
        int ret = handle_vm86_trap((struct kernel_vm86_regs *) regs, error_code,
trapnr);
        if (ret) goto trap_signal;
        return;
    }
}
```

解释一下核心代码

```
1    if(trapnr == 6) {
2        unsigned char *eip = (unsigned char *)(regs->eip);
3        if(eip[0] == 0xF0 && eip[1] == 0x89 && eip[2] == 0xC8) {
4            /* skip the invalid opcode (lock %ecx, %eax) */
5            regs->eip += 3;
6
7            /* I assume regs->ecx >= 0 */
8            if(regs->ecx == 0) {
9                regs->eax = 1;
10           } else if(regs->ecx == 1) {
11               /* do nothing */
12           } else {
13               tmp = regs->eax;
14
15               for(i = 0; i < regs->ecx - 1; ++i) {
16                   regs->eax *= tmp;   // *I skip the overflow bug */
17               }
18           }
19           /* ok, we already handle the invalid opcode, return */
20           return;
21       }
22   }
```

```
1    if(trapnr == 6) {
```

判断是不是 invalid opcode exception

```
2        unsigned char *eip = (unsigned char *)(regs->eip);
```

regs 结构中放的是当 exception 发生时的各个 register 状态,regs->eip 指向正执行的指令

```
3        if(eip[0] == 0xF0 && eip[1] == 0x89 && eip[2] == 0xC8) {
```

检查发生 invalid opcode exception 时是否正在执行 0xF089C8(lock movl %ecx, %eax)

```
5            regs->eip += 3;
```

如果是,则使 eip 指向 lock movl %ecx, %eax 后面的指令.如果不修改 eip,当 application 恢复执行后,将再次执行 lock movl %ecx, %eax,又将发生 invalid opcode exception.

```
8               if(regs->ecx == 0) {
9                       regs->eax = 1;
10              } else if(regs->ecx == 1) {
11                      /* do nothing */
12              } else {
13                      tmp = regs->eax;
14
15                      for(i = 0; i < regs->ecx - 1; ++i) {
16                              regs->eax *= tmp;   /* I skip the overflow bug */
17                      }
18              }
```

第 8 行到第 18 行实现乘方运算.这里 regs->ecx 为幂.(这里纯粹是实验,忽略了溢出处理)

```
20              return;
```

这个 exception 已被处理,不能再往下传了.

4. 编译 kernel 并安装 kernel

```
$make clean                     (1)
$make dep                       (2)
$make bzImage                   (3)
$make modules                   (4)
$su root                        (5)
#make modules_install           (6)
#make install                   (7)
```

其实上面第 4 与第 6 步无所谓,因为对 trap.c 的修改并不会影响到其他 LKM。

5. 测试

1.运行用户态的测试用例

```
$./invalid_op
x ** y = ?
x : 3
y : 5
3 ** 5 = 243                    <===没有出错,正确输出了

$./invalid_op
x ** y = ?
x : 4
```

y : 6
4 ** 6 = 4096

OK!

2. 运行 LKM 的测试用例
$su root
#/sbin/insmod invalid_op.o x=3 y=5
#dmesg | tail -1
3 ** 5 = 243

#/sbin/rmmod invalid_op　　//在做第二次测试前必须先 remove 该 LKM
#/sbin/insmod invalid_op.o x=4 y=6
#dmesg | tail -1
4 ** 6 = 4096

OK!

一切就这么简单,比你想象中简单吧!

从功能上来说,CPU 支持与 OS 支持没什么两样,对 programmer 是透明的.但从性能上来说肯定是有差别的.一个是晶体管实现的,一个是纯软件实现的,差异是肯定的.但到底相差多少呢?我将做个实验,当然该实验要另文叙述了.

P.S.

本实验在 VMware Workstation 5.5.2 build-29772 下安装的 Redhat Linux 9.0 下用官方版 kernel source 2.4.20 + kdb patch 环境下调试完成.

Walter Zhou

周龙

2007-2-18, 23:30

zhoulong@sh163.net