漏洞分析利器之Linux内核调试器LKMD

徐荣维(elemeta)

椒图科技【天择实验室】

目录

听这堂课你将得到的知识点

- Linux上常用的调试器
- 调试器具备的能力和功能
- 调试器工作的原理
- 开发内核调试器需要的知识
- **LKMD**，又一个调试器

什么是调试器（Debugger）

- 调试器是一种特殊的软件，用来分析其他软件的运行过程
  1. 调试器相对于被调试的目标，处于上帝模式
- 调试器是排查BUG的利器
  1. Debugger = de-bug-ger
  2. 软件中存在bug不可避免
  3. 我们需要找到bug并修复
- 开发人员：解决bug，调试程序流程
- 安全人员：破解软件内部原理，漏洞挖掘

调试器与逆向工程

- 一般会结合静态分析和动态分析的方法
- 静态分析
  - IDA
- 动态分析
  - 调试器
  - 在调试器中运行目标程序
  - 动态跟踪程序的执行流(程序的运行状态，内存使用状况，寄存器状态)
  - 理解程序的内部实现方式
  - 手工脱壳、动态加密的程序……

调试器与漏洞挖掘

- 漏洞动态分析技术
  - 附加上调试器，运行目标程序，执行程序fuzz过程
  - 利用调试器构造特殊数据，分析执行结果
  - 发现bug，分析是否可以成为漏洞
- 开发shellcode、Exploit
  - 直接加载shellcode到内存并跳转到shellcode执行，调试很方便
  - 在存在漏洞的系统上调试我们的利用代码

Linux常用的调试器

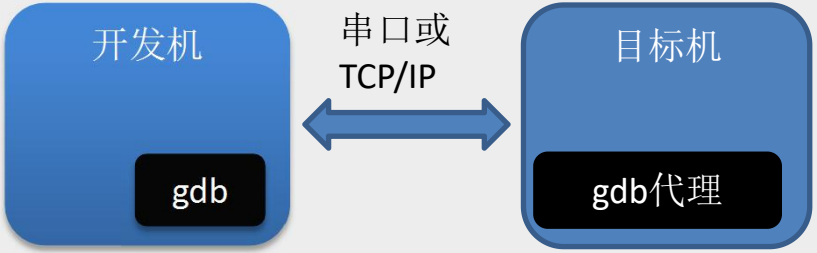
- 调试应用程序
  - gdb
- 调试内核
  - kgdb（需要双机调试）
  - kdb（单机，只支持指令级调试）

```
early console in decompress_kernel

Decompressing Linux... Parsing ELF... done.
Booting the kernel.

Entering kdb (current=0xffff88007b670000, pid 1) on processor 0 due to Keyboard
Entry
[0]kdb> bp sys_open
Instruction(i) BP #0 at 0xffffffff811a1514 (SyS_open)
    is enabled addr at ffffffff811a1514, hardtype=0 installed=0

[0]kdb> go_
```

本课接下去要讲的是LINUX内核调试器

内核调试器 - kgdb

- 1.需要双机调试
  - 目标机：运行被调试内核
  - 开发机：运行gdb
- 2.内核参数：CONFIG_KGDB
- 3.kgdboc=ttyS1,115200 kgdbwait




```
→ ~ gdb linux/linux-3.13.11-dream/vmlinux
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
Reading symbols from linux/linux-3.13.11-dream/vmlinux...done.
(gdb) set serial baud 115200
(gdb) target remote /dev/ttyS1
Remote debugging using /dev/ttyS1
kgdb_breakpoint () at kernel/debug/debug_core.
1042            wmb(); /* Sync point after
(gdb)
```

目标机
启动参数

```
linux   /boot/vmlinuz-3.13.11-dream root=UUID=966cb0[illegible]9c1-4d88-a50
8-be131dbd266a ro find_preseed=/preseed.cfg auto noprompt priority=critical
locale=en_US quiet crashkernel=384M-:128M crashkernel=384M-:128M kgdboc=kbd,
ttyS1,115200 kgdbwait
```

内核调试器 - kdb

- 1.在本地机器上运行
- 2.内核参数：CONFIG_KGDB_KDB
- 3.kgdboc=kbd kgdbwait

只要一台机器
我喜欢这种方式

```
early console in decompress_kernel

Decompressing Linux... Parsing ELF... done.
Booting the kernel.

Entering kdb (current=0xffff88007b670000, pid 1) on processor 0 due to Keyboard
Entry
[0]kdb> bp sys_open
Instruction(i) BP #0 at 0xffffffff811a1514 (SyS_open)
    is enabledoaddr at ffffffff811a1514, hardtype=0 installed=0

[0]kdb> go_
```

未解决的问题

- 无论是kgdb还是kdb，都需要**重新编译内核**

1. 云锁的Linux操作系统加固功能需要内核驱动支持
2. 在上百个不同版本的Linux操作系统上测试开发的驱动
3. 测试环境模拟用户的真实环境

上百个内核版本，
都要重新编译，
会疯掉的！！

我想要一个内核调试器

1. 一台机器就搞定
2. 不用重新编译内核
3. 不需要内核有KGDB或者KDB支持
4. 独立的调试器
5. 以驱动的形式存在，即插即用
6. 就像SoftICE那样

LKMD (Linux Kernel Module Debugger)

LKMD概述

- 全称 Linux Kernel Module Debugger
- 基于kdb开发，兼容kdb的命令，独立于kdb存在
- 不依赖CONFIG_KGDB或者CONFIG_KGDB_KDB选项
- 本身是一个驱动模块，不需要重新编译内核
- 即插即用
- 指令级调式

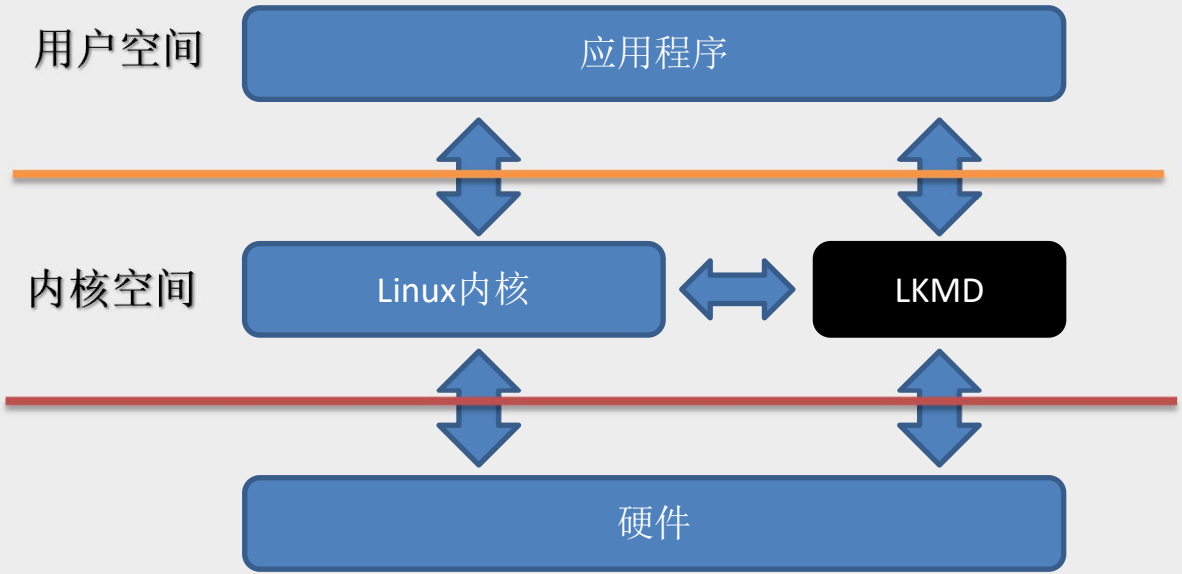
LKMD能做什么

- 设置断点
- 单步调试
- 操作内存
- 操作寄存器
- 反汇编
- 更多功能开发中.....

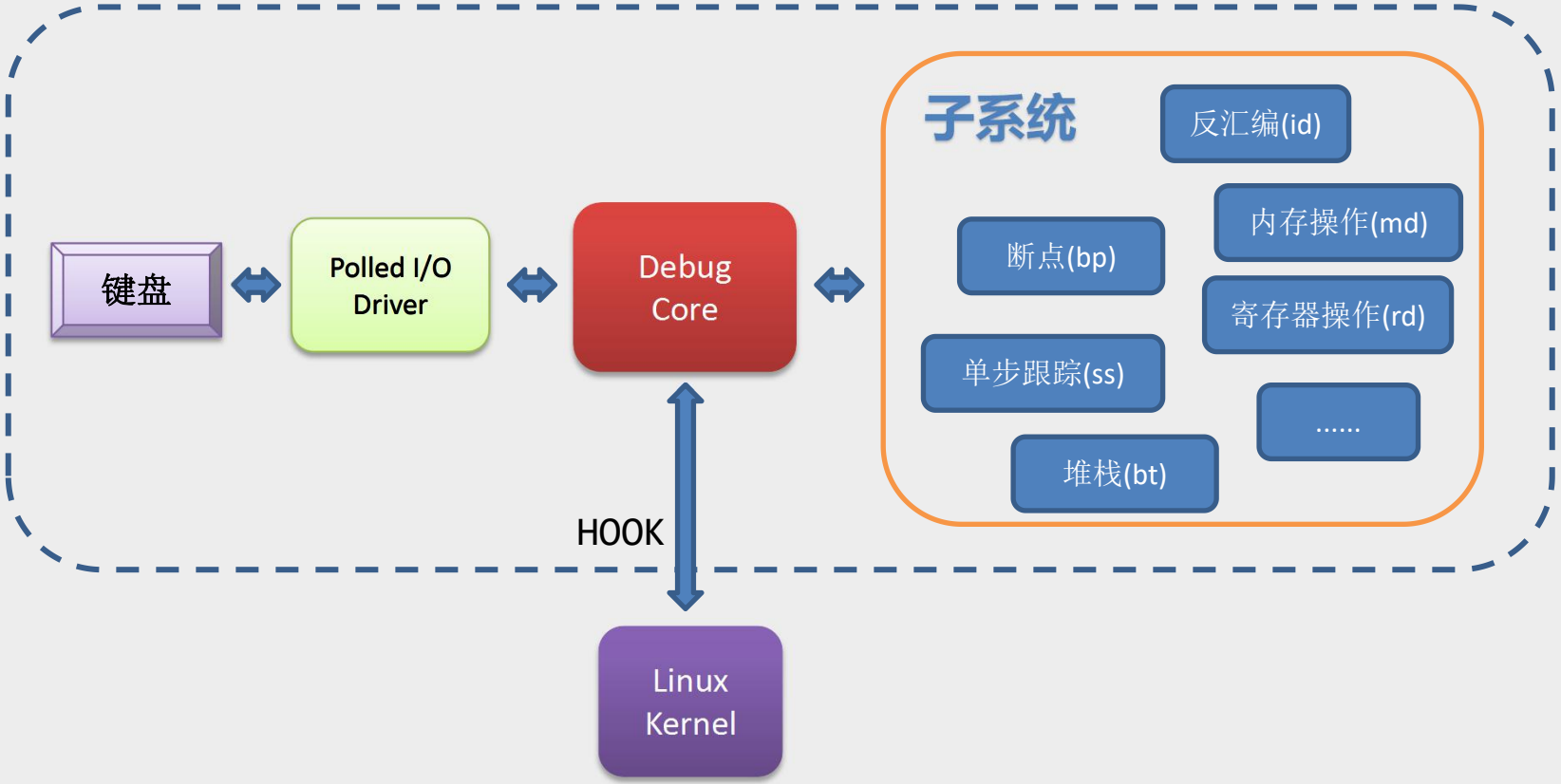
```
[0]LKMD> help
Command         Usage                   Description
-------------------------------------------------------
md              <vaddr>                 Display Memory Contents, also mdWcN, e.g. m
d8c1
mdr             <vaddr> <bytes>         Display Raw Memory
mdp             <paddr> <bytes>         Display Physical Memory
mds             <vaddr>                 Display Memory Symbolically
mm              <vaddr> <contents>      Modify Memory Contents
id              <vaddr>                 Display Instructions
go              [<vaddr>]               Continue Execution
rd                                      Display Registers
rm              <reg> <contents>        Modify Registers
ef              <vaddr>                 Display exception frame
ll              <first-element> <lin    Execute cmd for each element in linked list
env                                     Show environment variables
set                                     Set environment variables
help                                    Display Help Message
?                                       Display Help Message
cpu             <cpunum>                Switch to new cpu
ps              [<flags>|A]             Display active task list
pid             <pidnum>                Switch to another task
reboot                                  Reboot the machine immediately
lsmod                                   List loaded kernel modules
per_cpu                                 Display per_cpu variables
bp              [<vaddr>]               Set/Display breakpoints
bl              [<vaddr>]               Display breakpoints
bpa             [<vaddr>]               Set/Display global breakpoints
bph             [<vaddr>]               Set hardware breakpoint
bpha            [<vaddr>]               Set global hardware breakpoint
bc              <bpnum>                 Clear Breakpoint
be              <bpnum>                 Enable Breakpoint
bd              <bpnum>                 Disable Breakpoint
ss                                      Single Step
ssb                                     Single step to branch/call
```

LKMD在系统中的位置

- LKMD寄生在Linux kernel里
- 控制应用程序的执行流
- 控制Linux Kernel的执行流
- 独立访问硬件资源

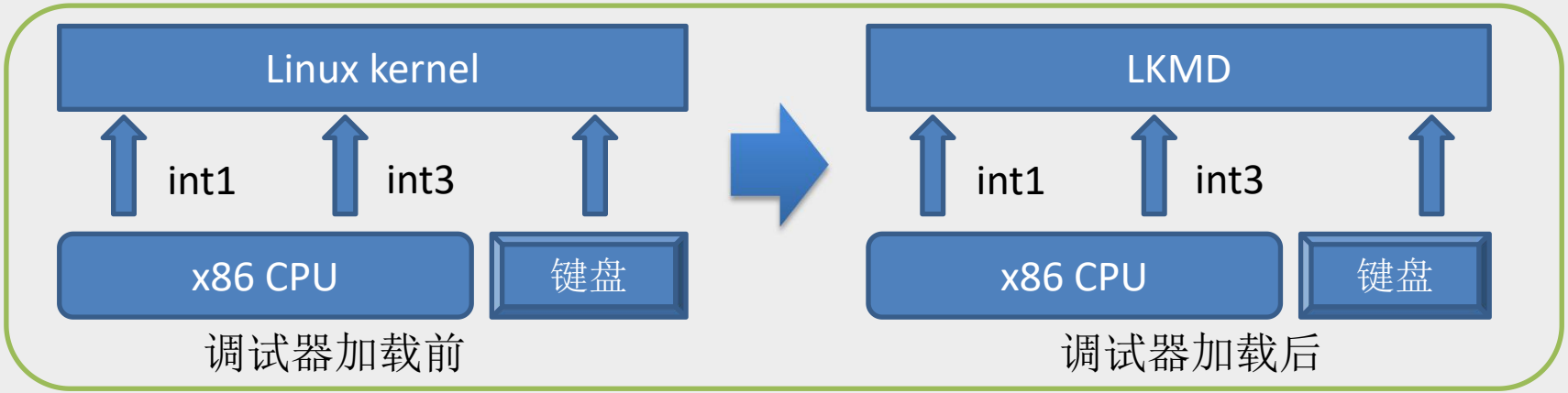


LKMD架构


键盘
Polled I/O Driver
Debug Core
子系统
反汇编(id)
内存操作(md)
断点(bp)
寄存器操作(rd)
单步跟踪(ss)
.....
堆栈(bt)
HOOK
Linux Kernel

接管内核态调试入口

- int1中断：硬件断点、单步调试
- int3中断：软件断点、进入调试器
- I/O设备驱动：键盘、串口、TCP/IP



Debug Core - 进入调试器

- 调试器插入到Linux kernel后
  - 内核继续工作，LKMD等待被召唤
- 触发以下情况来召唤LKMD
  - 代码中插入int3指令，被执行

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    asm("int3");
    return 0;
}
```

```
Entering LKMD (current=0xffff88007a618b80, pid 2037) on processor 0 due to KDB_E
NTER() @ 0x4004fc
[0]LKMD> _
```

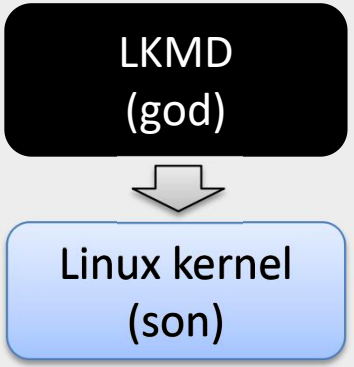
  - CPU执行到断点处

```
Instruction(i) breakpoint #0 at 0xffffffff811de050 (adjusted)
0xffffffff811de050 SyS_open:           int3

Entering LKMD (current=0xffff88007c7a8000, pid 1) on processor 0 due to Breakpoi
nt @ 0xffffffff811de050
[0]LKMD> _
```

  - 出现异常后

Debug Core - 接管CPU和中断处理

- 召唤LKMD后，处于上帝模式
  - Linux kernel被暂停
  - 屏蔽所有中断
  - Local APIC，让非当前CPU处于忙等状态
  - 接收键盘输入的命令并执行



Debug Core - 总结

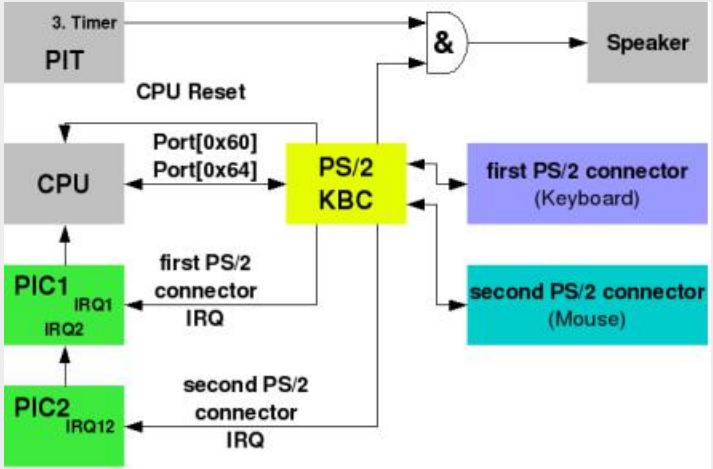
Polled I/O Driver - 如何接收键盘按键

- Linux Kernel已经暂停，需要自己实现键盘驱动
- i8042芯片, CPU通过IO端口直接和它通信
- 使用轮询的方式获取键盘输入，非IRQ1
  - 不断轮询键盘的状态(kbd_status)
  - 键盘按下时，接收键盘扫描码(scancode)
  - 扫描码再翻译成ASCII字符

```
"inb    $0x60, %al\n"
"movb   %al, scancode\n"
"inb    $0x64, %al\n"
"movb   %al, kbd_status\n"
```

Polled I/O Driver - i8042键盘控制器

- IO端口
  - 0x60 (数据)
  - 0x64 (状态/命令)
- 寄存器
  - 状态寄存器(8 bit)
  - 命令寄存器(8 bit)
  - 输出缓冲器(8 bit)
  - 输入缓冲器(8 bit)



- 参考资料

http://wiki.osdev.org/%228042%22_PS/2_Controller

Polled I/O Driver - i8042 - 状态寄存器

- 在键盘上，大小为8bit，任何时候都可以被CPU读取

Bit7: PARITY-EVEN(P_E): 从键盘获得的数据奇偶校验错误

Bit6: RCV-TMOUT(R_T): 接收超时，置1

Bit5: TRANS_TMOUT(T_T): 发送超时，置1

Bit4: KYBD_INH(K_I): 为1，键盘没有被禁止。为0，键盘被禁止。

Bit3: CMD_DATA(C_D): 为1，输入缓冲器中的内容为命令，为0，输入缓冲器中的内容为数据。

Bit2: SYS_FLAG(S_F): 系统标志，加电启动置0，自检通过后置1

Bit1: INPUT_BUF_FULL(I_B_F): 输入缓冲器满置1，i8042 取走后置0

BitO: OUT_BUF_FULL(O_B_F): 输出缓冲器满置1，CPU读取后置0

Polled I/O Driver - i8042 - 命令寄存器

- Controller Command Byte（控制器命令字节), 大小为8bit

Bit7: 保留，应该为0

Bit6: 将第二套扫描码翻译为第一套

Bit5: 置1，禁止鼠标

Bit4: 置1，禁止键盘

Bit3: 置1，忽略状态寄存器中的 Bit4

Bit2: 设置状态寄存器中的 Bit2

Bit1: 置1，enable 鼠标中断

Bit0: 置1，enable 键盘中断

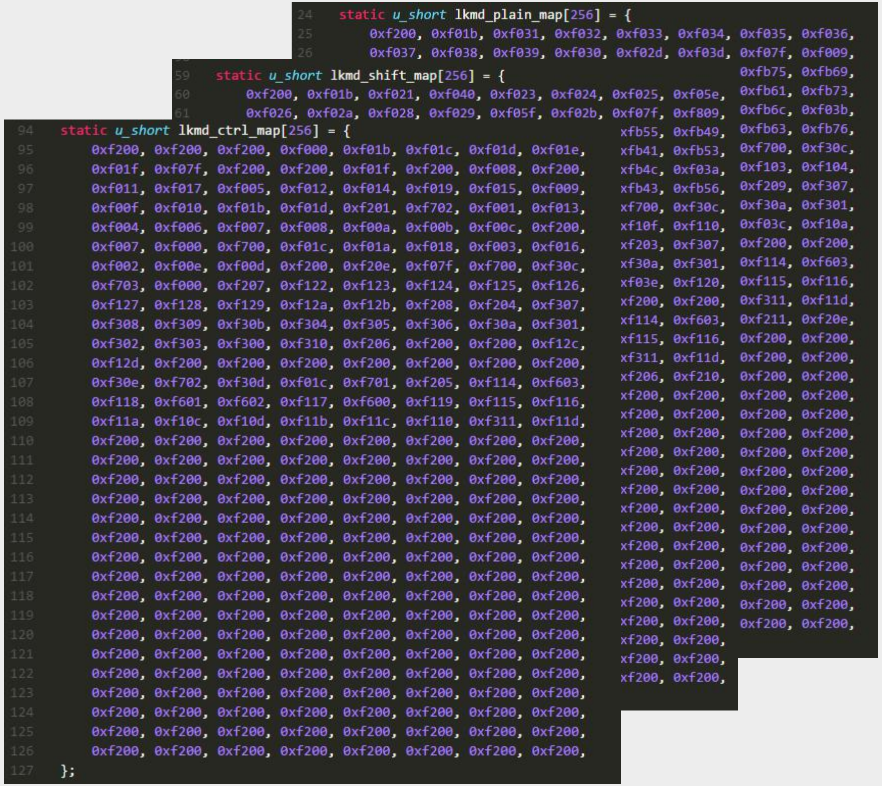
Polled I/O Driver - 与CPU通信

- 数据端口
  - 读取扫描码：`inb $0x60 %al`
  - 设置LED灯：`outb %al $0x60`
- 状态寄存器
  - 读取状态：`inb $0x64 %al`
- 命令寄存器
  - 下达命令：`outb %al $0x64`

IO端口	访问权限	功能
0x60	读/写	数据
0x64	读	状态寄存器
0x64	写	命令寄存器

Polled I/O Driver - 键盘驱动 - 扫描码翻译

- 获取键盘状态kbd_status
    - 忽略鼠标事件
    - 键盘事件包括**按下**和**释放**两种状态
    - 忽略按键释放的事件(Shift键除外)
- 处理控制键
    - CapsLock (设置led指示灯)
    - Ctrl
    - Shift
- 翻译按键
    - 映射表(IBM-PC键盘)
- 下一轮回

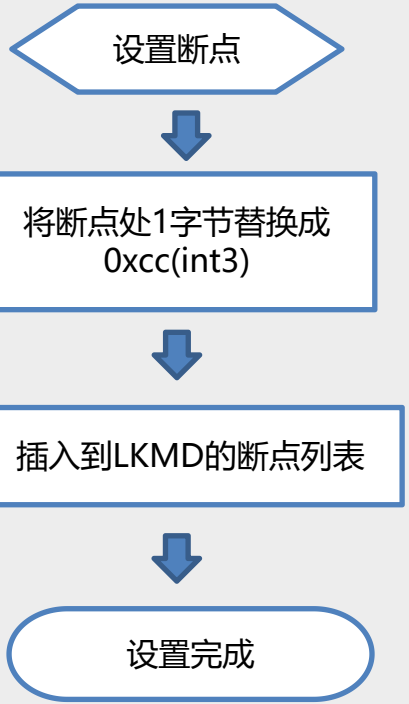


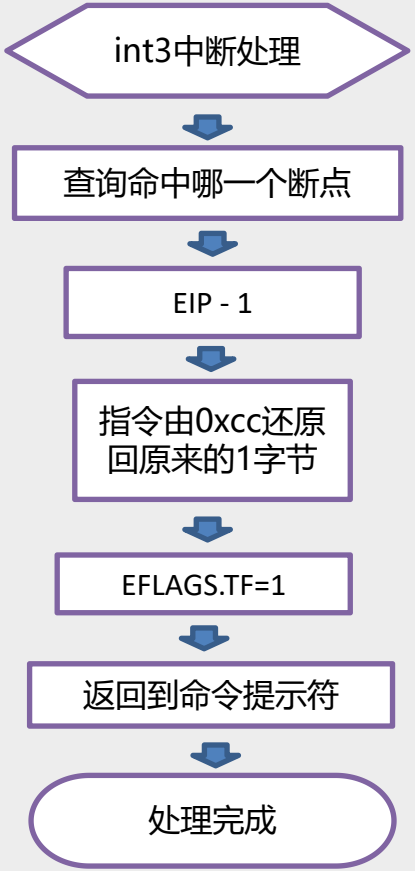
子系统 - 断点 - 基础知识

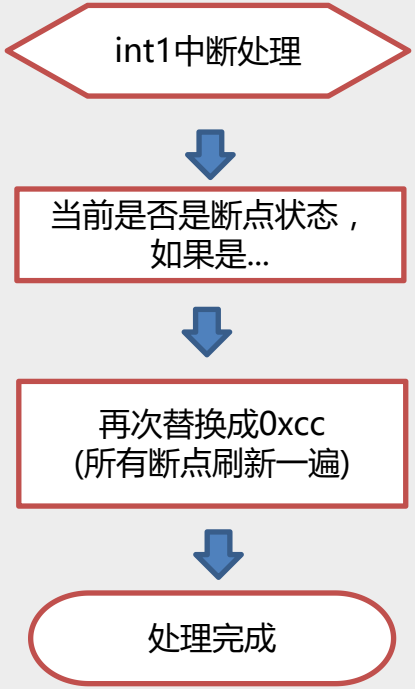
- INT3(#BP, Breakpoint)
  - 软件断点
  - 由软件调用(用户态和内核态的程序均可调用)
- INT1(#DB, Debug Exception)
  - 提供硬件断点、单步调试的支持
  - 命中硬件断点的时候产生一个int1异常
  - 如果EFLAGS.TF=1, 执行完当前指令, 产生一个int1异常，并设置EFLAGFS.TF=0
- 标志寄存器EFLAGS的TF(Trap Flag)位
  - TF=1时，执行完当前指令后，产生一个int1异常，并将TF改回0

值得重复
强调一遍

子系统 - 断点(bp) - 软件断点







子系统 - 断点(bp) - 例子

```
.text:08000C10                push    ebp
.text:08000C11                mov     ebp, esp
.text:08000C13                sub     esp, 4
.text:08000C16                mov     eax, large gs:14h
.text:08000C1C                mov     [ebp+var_4], eax
.text:08000C1F                xor     eax, eax
.text:08000C21                mov     edx, [ebp+var_4]
.text:08000C24                xor     edx, large gs:14h
.text:08000C2B                jnz     short loc_8000C2F
.text:08000C2D                leave
.text:08000C2E                retn
```

```
.text:08000C10                int     3               ; Trap to Debugger
.text:08000C11                mov     ebp, esp
.text:08000C13                sub     esp, 4
.text:08000C16                mov     eax, large gs:14h
.text:08000C1C                mov     [ebp-4], eax
.text:08000C1F                xor     eax, eax
.text:08000C21                mov     edx, [ebp-4]
.text:08000C24                xor     edx, large gs:14h
.text:08000C2B                jnz     short loc_8000C2F
.text:08000C2D                leave
.text:08000C2E                retn
```

- int3触发后
  - EIP = 800C11
  - EIP-1(即改回800C10)
  - 800C10处再替换成 push ebp;
  - 设置EFLAGS.TF=1
  - 从800C10出继续执行
  - 执行完当前指令后产生int1异常
- int1触发后
  - EIP=800C11
  - 判断是软件中断状态
  - 将上一条指令开头替换成int3
  - （指令长度不知道，所有断点刷新一遍即可）
  - 完成

子系统 - 断点(bp) - 硬件断点

- 硬件断点指使用CPU的调试寄存器(Debug Register)
- 只能设置4个(dr0、dr1、dr2、dr3)
- dr6 调试状态寄存器
  - 处理int1异常时检查dr6，判断是否命中硬件断点
- dr7 调试控制寄存器
  - 地址长度、指令执行断点、数据读写断点、I/O读写断点
- 当产生int1中断时处理硬件断点

子系统 - 单步跟踪(ss)

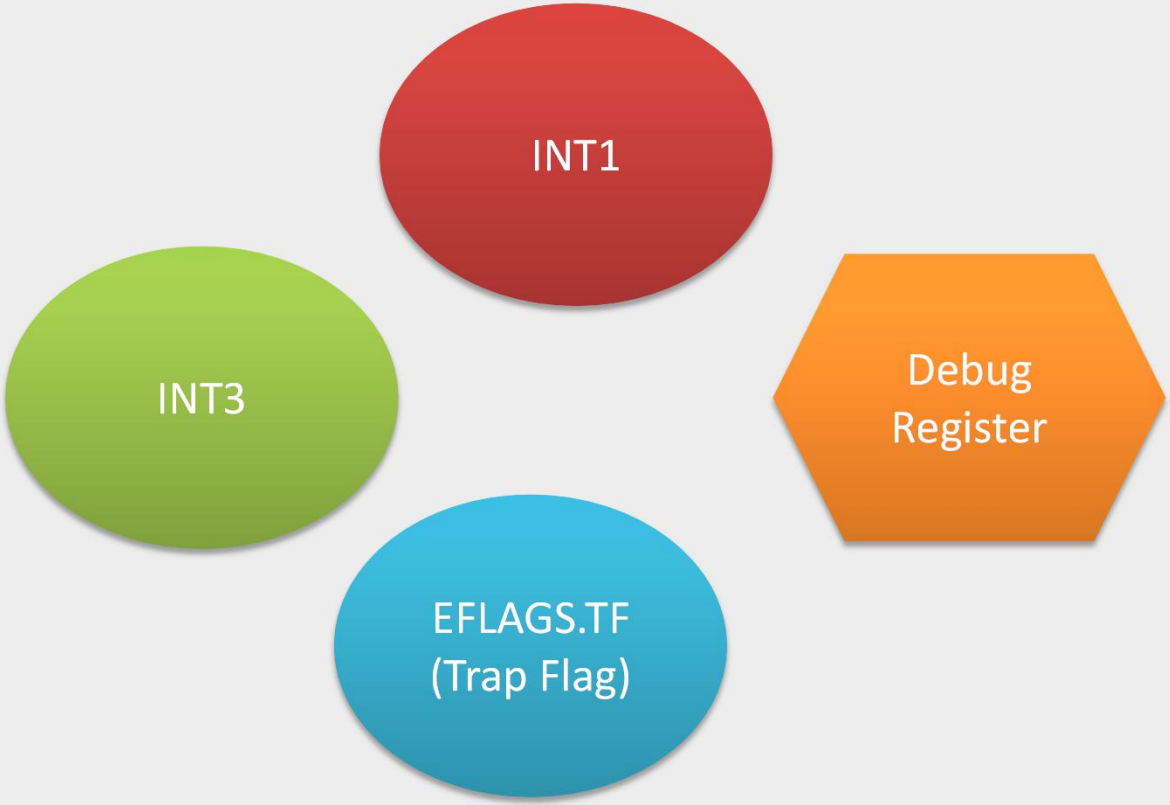
- 指令级单步跟踪
  - 设置EFLAGS.TF （ Trap Flag ）= 1
  - CPU会在当前指令执行完产生int1中断
  - 当 int1终端发生时检查是否是单步调试状态

```
.text:08000C10                push    ebp
.text:08000C11                mov     ebp, esp
.text:08000C13                sub     esp, 4
.text:08000C16                mov     eax, large gs:14h
.text:08000C1C                mov     [ebp+var_4], eax
.text:08000C1F                xor     eax, eax
.text:08000C21                mov     edx, [ebp+var_4]
.text:08000C24                xor     edx, large gs:14h
.text:08000C2B                jnz     short loc_8000C2F
.text:08000C2D                leave
.text:08000C2E                retn
```

- 源码级单步跟踪（ LKMD暂不支持 ）
  - 在指令级的基础上进一步处理，忽略同一行的其他指令
  - 需要调试信息支持，用来查找源代码行和指令间的对应关系

```
261  static void lkmd_console_write(const char *s, unsigned len)
262  {
263      struct console *c = console_drivers;
264
265      /* Write to all consoles. */
266      while (c) {
267          if ((c->flags & CON_ENABLED) && c->write) {
268              c->write(c, s, len);
269          }
270          touch_nmi_watchdog();
271          c = c->next;
272      }
273
274      printk(s);
275  }
```

子系统 - 断点 - 总结



子系统 - 内存操作

调试器：我在内核态运行，可以直接访问内核虚拟地址，其他形式的地址要映射成内核虚拟地址，才能访问!!!

其他子系统

- 反汇编（id)
  - 使用gdb中的反汇编引擎
- 寄存器操作(rd)
  - 进入调试器前已经push到栈上了
- 进程列表(ps)
  - task_struct
- ……

```
struct pt_regs {
    unsigned long bx;
    unsigned long cx;
    unsigned long dx;
    unsigned long si;
    unsigned long di;
    unsigned long bp;
    unsigned long ax;
    unsigned long ds;
    unsigned long es;
    unsigned long fs;
    unsigned long gs;
    unsigned long orig_ax;
    unsigned long ip;
    unsigned long cs;
    unsigned long flags;
    unsigned long sp;
    unsigned long ss;
};
```

使用方法

- git clone http://github.com/elemeta/lkmd.git ~/lkmd
- cd ~/lkmd
- make
- insmod lkmd.ko

当前的状态和未来的工作

- 当前的开发状态
  - 基于kdb
  - 支持x86（64位和32位）
  - 指令级调式
- 未来的工作
  - 查看调用堆栈
  - 支持TCP/IP远程调试
  - 支持Arm，Arm64
  - 添加更多的功能

总结

- LKMD是另一个Linux内核调试器
  - **不用编译内核，即插即用**
  - 基于kdb开发，兼容kdb的命令
  - 指令级调式
  - 支持x86架构(x64,x32)
  - 开源，使用GPL协议
  - 持续开发中……

反调试（Anti-debug）

- 判断两个指令之间的执行时间
- 检查/proc/kallsyms中是否包含LKMD的符号
- 检查idt中的中断处理程序地址是否在内核text段地址范围内
- 检查函数开头是否是int3
- 检查父进程是否是gdb
- 检查硬件断点
- 代码混淆
- 动态解密代码
- 代码段CRC校验
- ……

谢谢观看！

https://github.com/elemeta/lkmd

elemeta 椒图【天择实验室】
Email：xurw@jowto.com