使用jprobe构建镜像协议栈的原理与感悟

原创

dog250 于 2014-06-07 13:00:45 发布 6407 收藏

突然回想起了往事，那是2007年的冬天的一个周五，我在看我的老湿调试Linux协议栈的IP层，只见他修改了路由查找的逻辑，然后直接make install了一下就即时生效了，当时我只知道的是，修改了这个逻辑需要重新编译内核，而他并没有重新编译，好像只是编译了一个文件...编译内核这个耗时又无聊的工作阻碍了我对Linux内核的探索进度，直到今天，我依然对编译内核有相当的恐惧，不怕出错，而是怕磁盘空间不够，initrd的组装拆解之类，太繁琐了。我之所以知道2007年的那天是周五，是因为第二天我要加班，没有谁逼我，我自愿的，因为我想知道师父是怎么做到不重新编译内核就能改变非模块的内核代码处理逻辑的，第二天的收获很多，不但知道了他使用了“镜像协议栈”，还额外赚了一天的加班费，我还记得周六加完班我和老婆去吃了一家叫做石工坊的羊排火锅，人家赠送了一只绿色的兔子玩偶。现在那个玩偶还在，我家小小特别喜欢，就是这么一堆看似无关却又巧合的事，让我在这个周末觉得必须写下一点什么。

好吧，从kprobe开始吧。如果我面试一个搞Linux内核的人，问他怎么调试内核，他回答先加入printk然后重新编译最后载入新内核运行，看dmesg，我会让他先等上几分钟，然后人事就会告诉他让他回去等通知。幸运的是，我没有碰到这样的人让我面试来展现我五十步笑百步的半瓶子晃荡作风，也从来没有碰到过如此不仁慈的面试者，我曾经在一次找工作的时候真的就是这么说的，人家也真的让我去等通知，然而我真的就等到了通知，通知入职的时间以及体检事宜...说这些的目的是想展示一个调试内核的利器，kprobe。它可以动态修改内核地址空间代码的二进制指令，然后执行任意你想让它执行的代码段，这也许应该可以称为二进制动态编程！多么黑的技术，完全无视源代码的逻辑，完全无视编译器的苦功，直接就这么把二进制机器码给改了。

kprobe的工作原理很简单，比如你有一个函数func，你可以在func被调用前和调用后各插入一段代码，我们假设func指令是

begin
go
end

kprobe要做的就是替换掉begin，将其变为：

jmp prefunc

当然在替换前还要保存原有的，以便执行完我们的钩子函数prefunc还能跳回原来的逻辑，至于复杂的jmp细节(长短跳，相对绝对跳之类的)以及Intel的INT 3调试模式单步模式本文不再赘述，赘这个字用得好，因为所有这些细节都是累赘，你换个非Intel平台的话，你就知道这些是多么累赘了，不过对一辈子不换平台的那些人来讲，理解这些细节就成了资本，因此想了解这些，还是去看雪吧，找级别高态度好的问，或者潜水也行，我觉得看雪的信息量已经够大了，基本上都能找到现成的。

虽然我不提倡在本文讲Intel的细节，但是有一个除外，那就是prefunc钩子函数的参数问题，比如我想钩住vfs_write函数，它的声明如下：

```
ssize_t vfs_write(struct file *file, const char __user *buf, size_t count, loff_t *pos);
```

如果这个prefunc钩子函数的参数和vfs_write的一样那多好啊，整个逻辑就成了：

```
ssize_t prefunc(struct file *file, const char __user *buf, size_t count, loff_t *pos)
{
    todo_something(.....);
    return vfs_write(file, buf, count, pos);
}
```

但是不幸的是，kprobe做不到。因为它是基于INT 3异常/中断来处理的，而Intel的异常/中断的处理有一套特定的规程，即保存所有的上下文环境，因此它的参数就只有struct pt_regs *regs一个，即所有的寄存器信息。要想还原vfs_write的参数，你必须针对这个regs参数做一个“深度解析”才行，而这又一次将你引入了平台相关的地狱，如果你在X86平台，你就不得不对它的寄存器使用规约做一番详细的了解才能还原被钩函数的参数，对于X86来讲，参数保存在栈中(也可以通过寄存器传参)，要想还原被钩函数的参数现场，你要分析的就是regs->sp，下面我就不说了。

说了上述不幸，来点幸运的，那就是Linux内核提供了一种kprobe之上的机制，帮你实现了上面说的那些本应该由你自己完成的工作，这就是jprobe。总的来讲，jprobe的要点在于它实际上就是一个kprobe的prefunc，它的prefunc是这么实现的：

```
prefunc(kprobe, regs)
{
    保存regs寄存器现场
    保存栈的内容  //因为jprobe使用和被钩函数相同的栈，可能会改变栈的内容
    替换regs里面ip指针为jprobe钩子的指针
    返回
}
```

就这样一个kprobe的prefunc钩子函数就把INT 3返回正常流，但是请注意，在这个prefunc中，将regs的ip改变了，改成了jprobe的entry函数，而栈信息一点都没有变，因此返回正常流之后，栈上的参数信息没有变，只是执行的函数变了，变成了entry！等jprobe的entry执行完了之后，调用jprobe_return来还原，这个return实际上就是再次进入INT 3异常，然后调用kprobe的另一个钩子函数来还原现场，即将prefunc保存的regs现场以及栈现场还原。是不是很像setjmp和longjmp啊！是的，几乎是一样的！到此为止，程序进入了被钩函书，整个流程就是：

进入INT 3--进入prefunc保存现场以及ip替换为entry--返回被修改后的正常流在同一栈上执行entry--进入INT 3--还原原始的reg中的ip以及恢复原始栈的内容--返回原始的执行流执行被钩函数

jprobe的entry钩子函数的参数和原始的被钩函数的参数完全一样，这是因为它们的栈内容一模一样。以上就是jprobe的全部了，当然除了细节。

除了大致原理之外，值得注意的一个细节就是，jprobe的钩子函数中是可以发生进程切换的，因为它实际上是在一个正常流中执行，只不过这个正常流被修改了而已，而在kprobe的钩子函数中，是不能发生抢占的，因为本质上它还是在INT3的异常/中断处理函数中执行的。

那么，我们能用这个jprobe做些什么呢？如果你真的看懂了我的意图，那么我想说的也许正是你所想的，那就是使用jprobe可以实现一个镜像协议栈，我先将代码片断贴上：

```
static struct jprobe steal_jprobe = {
    .entry   = steal_ip_local_deliver,
    .kp = {
    .symbol_name    = "ip_local_deliver",
    }
};

int steal_ip_local_deliver(struct sk_buff *skb)
{
    if (skb && skb->mark == 1004) {
        ip_local_deliver_finish(skb);
    }
    jprobe_return();
    return 0;
}
```

这段代码也许表达了我的目的，即从ip_local_deliver开始，数据包将不再经原生的Linux协议栈处理，而是被偷到了我的steal_ip_local_deliver，在其内部，可以实现自己的协议栈处理逻辑，当然为了简单，我只是调用了 ip_local_deliver_finish将数据包直接绕过NF_HOOK往上传递。

可是，当你真的运行上面代码的时候，得到的将是无情的panic！因为在steal函数调用ip_local_deliver_finish之后，它一路走到了socket层，skb已经被free了，由于共享一个栈数据且skb传入的只是一个指向skb数据的指针，此时返回正常的ip_local_deliver之后，skb的字段取值将全不可用。我们需要做的是在steal函数内部阻止掉这个执行流，然而冯.诺依曼机器是串行处理机，且UNIX/Linux的执行流是靠fork分发的，也就是说你根本就不可能阻止掉任何一个执行流，除非调用exit，但是在softirq中是不能exit的，因为你根本不知道借用了哪个task_struct！为了不再panic，你只能：

```
int steal_ip_local_deliver(struct sk_buff *skb)
{
    if (skb && skb->mark == 1004) {
        ip_local_deliver_finish(skb_copy(skb, GFP_ATOMIC));
    }
    jprobe_return();
    return 0;
}
```

这样做之后，在steal中传入 ip_local_deliver_finish的只是skb的一个副本，待返回正常的ip_local_deliver后，原始的skb还是可用的。可是这就将一个数据流fork成了两个，对于TCP协议而言，TCP逻辑会自动丢掉重复的，但是对于像UDP或者ICMP之类的数据流而言，将会收到双份的数据，一个来自正常的协议栈，另一个来自steal的协议栈。现在的问题在于，如何阻止掉

正常的协议栈处理。
想当然的办法就是让正常的ip_local_deliver直接返回0。这实际上也是一种正确的做法。现在我们回到最开始，膜拜一下那个阴招，那就是二进制动态编程！我能不能将被钩的函数也改掉呢？思路很清晰，接下来就是找解决问题的方法了，我定义了一个stub函数：

```
int stub(struct sk_buff *skb)
{
    return 0;
}
```

我要做的就是将返回原始正常流后原本要调用ip_local_deliver的指令改为调用stub，要实现这个就要进行动态的二进制指令修改。深入到kprobe细节的都应该知道kprobe结构体包含一个字段：

```
    /* copy of the original instruction */
    struct arch_specific_insn ainsn;
```

我连注释也一并贴上了，因为这省了我解释了，注意命名，ainsn中的a就是arch的意思，这个多加的层为上层屏蔽了平台相关的细节，对于X86而言，它就是：

```
u8 *insn;
```

是的，一连串的二进制指令，很显然，这里保存的指令肯定是jmp ip_local_deliver之类的，因为这段指令的目的就是跳转回原始的执行流。我只需要将其改为jmp stub就可以了。就是说，在jprobe的entry钩子中，将kprobe的ainsn.insn改为jmp stub，然后为了不影响不相关的后续的执行流返回ip_local_deliver，在stub中再将kprobe的ainsn.insn改回去。
接下拉的任务就是找指令了，前面说了，与其看大部头全英文的Intel手册，不如直接看雪！我并不反对看Intel手册，但是为了这么一个简单的问题一头扎进去也有点太彪了。看雪上的内容很多很全！我尝试了两种方式：
方式1：短跳转，指令码为0xFF 0x04 $小端倒序的stub函数地址
失败！不恋战，因为我的目的不是搞清楚Intel的指令集。不过还是稍微有一点想不通，早就开始平坦内存模式了，怎么现在还有人用长跳啊！不管怎么样，换一种方式。
方式2：借助寄存器。即mov rax $小端倒序的stub函数地址; jmp rax;指令码为0x48 0xB8 $小端倒序的stub函数地址 0xFF 0xE0。
这次成功了。不欢呼，不庆祝，因为这只是一个环节而已。
完整的代码如下：

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/hardirq.h>

#include <linux/skbuff.h>

// 从/proc/kallsyms中找出的ip_local_deliver_finish地址
// 我只是想在jprobe函数中直接调用finish，企图跳过NF_HOOK
#define func 0xffffffff812b70f3

int (*f)(struct sk_buff *);
// 保存全局变量，因为无法从steal钩子函数中取到kprobe
struct kprobe *k = NULL;

#define JMP_CODE_SIZE   12
#define ADDR_SIZE       sizeof(void *)

u8 saved[MAX_INSN_SIZE] = {0};

// 注意,不要太在意下面的二进制指令码的具体细节!主要含义理解即可:将地址送入寄存器,jmp到该处
u8 jmpcode[JMP_CODE_SIZE] = {0x48, 0xb8, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xe0};

int stub(struct sk_buff *skb)
{
    memcpy(k->ainsn.insn, saved, MAX_INSN_SIZE);
    return 0;
```

```
}

int steal_ip_local_deliver(struct sk_buff *skb)
{

    if (skb && skb->mark == 1234) {
        // 先保存原始的替换指令码。
        memcpy(saved, k->ainsn.insn, MAX_INSN_SIZE);
        // 替换为jmp到steal函数的指令码。
        memcpy(k->ainsn.insn, jmpcode, JMP_CODE_SIZE);
        // 调用自己的函数,为了简单,我只是调用了ip_local_deliver_finish。
        (*f)(skb);
        // 从这里返回后，由于指令码已被替换为steal函数stub，因此就不会
        // 再返回正常的ip_local_deliver了。
    }
    jprobe_return();
    return 0;
}

static struct jprobe steal_jprobe = {
    .entry   = steal_ip_local_deliver,
    .kp = {
    .symbol_name    = "ip_local_deliver",
    }
};

static int __init jprobe_init(void)
{
    int ret;
    int i = 0, j = 9;
    unsigned long addr = (unsigned long)&stub;
    ret = register_jprobe(&steal_jprobe);
    if (ret < 0) {
        printk("register_jprobe failed:%d\n", ret);
        return -1;
    }
    k = &steal_jprobe.kp;
    f = func;
    // 根据stub函数的地址填充jmpcode指令码数组
    for (i = 0; i < ADDR_SIZE; i++, j--) {
        jmpcode[j] = (addr&0xff00000000000000)>>56;
        addr <<= 8;
    }
    return 0;
}

static void __exit jprobe_exit(void)
{
    unregister_jprobe(&steal_jprobe);
}

module_init(jprobe_init)
module_exit(jprobe_exit)
MODULE_LICENSE("GPL");
```

这就是几年前我看到的一个镜像协议栈的原理。虽然Linux很难直接通过make config将整个网络协议栈编译成一个模块，但是我们自己可以手工构建一个网络协议栈模块，无非就是把net/ipv4目录编译成一个模块，然后使用jprobe钩住netif_receive_skb这个底层函数，将控制权导入到我们自己的协议栈模块中。说白了在冯.诺依曼这种串行处理的机器中，争夺的就是控制权，

只要你占有了CPU，那控制权就属于你，一旦你有了控制权，你不光可以增删改查内存中的数据，还可以增删改查内存中的代码，因为数据和代码都在内存...

关于kprobe的文档，最好的还是Linux内核自带的Documentation/kprobes.txt。

本文解读了一个镜像协议栈的实现原理，但是同时也展示了一个Linux内核调试的方法，那就是使用kprobe上面的jprobe进行调试，实际上基于kprobe的调试工具很多，比如SystemTap之类的，但是个人觉得，在你亲自动手step by step编写一个原生的jprobe模块之前，还是不用那些工具为好，因为光是仅仅熟悉工具本身的用法就要花费不少时间和精力，而且如果底层原理还不理解的话，即使学会了工具的用法也会很快忘记。也许是我太老土了，但是我一直都记得教计算机编程的老师说过的，在亲自用命令行编译一个完整的程序之前，不要用IDE，是这个道理。等亲自动手玩转了kprobe和jprobe，再去学习基于它们封装的工具，那就简单多了，一旦学会便更加难以忘记。

后记：关于panic

编程和生活相比，其快感在于panic后的reset！不管你犯了多大的错误（段错误？栈溢出？被渗透？被抹屎？），不管你有多大的遗憾（/etc/sysctl.conf文件添加了kernel.panic = 1以后忘了sysctl -p...关键是我是远程连的公司的机器...可恨没有ipmi的支持！！），reset后一切成云烟！如果有什么过不去的坎，panic吧，然后reset！

时间过得太快了，从2007年至今，也就弹指一挥间，当时的我多么希望能成为像我的老湿那样的人，事实上，我也正是凭着这种简单的崇拜与对网络技术的好奇而一步步走到了现在的，水平谈不上什么登峰造极，但起码也是从菜鸟一步步来的，现在充其量是一只过度肥胖的老鸟。平白无故，突然就追忆起了往事，岁月不饶人啊！