

利用 GDB、KGDB 调试 Linux 内核 驱动模块及应用程序

目录

目标	2
1 基础环境搭建	3
1.1 资源环境	4
1.2 安装 vmware 虚拟机及 ubuntu 操作系统	4
2 编译内核	5
2.1 配置内核参数	5
2.2 编译内核	7
3 配置双机通信	8
3.1 双机串口通信	8
3.2 验证串口通信配置	10
4 配置串口调试	11
4.1 客户机调试配置	11
4.2 目标机调试配置	12
5 GDB 双机调试环境	14
5.1 驱动代码及编译	15
5.2 应用程序及编译	17
5.3 验证驱动程序	18
6 调试应用程序及驱动程序调用接口	19
6.1 配置驱动调试	19
6.2 调试应用程序	22
6.3 调试驱动程序	24
备注	26

目标

这几天看了一下 linux 内核提权的一个漏洞，里面涉及到了驱动程序漏洞及驱动调试内容，由于各类 linux 操作系统版本的不同，如果不能在自己机器上亲自调试驱动程序，可以说即使给了漏洞利用的 POC 源码也根本无法成功利用。因为内核漏洞的利用涉及到指令集的 POC 构造，不同内核版本模块加载指令地址不同，导致即使有 POC 也根本无法利用，只有在自己系统中亲自调试，才能做出相应的修改，达到内核漏洞利用的效果。这样就要求我们对 linux 内核驱动的调试过程，调试方法有个深入了解。经过两天的各处查找，配置，调试，终于弄清楚了内核调试的基本方法，为之后内核调试，漏洞分析提供技术支持。

在驱动程序开发或是内核漏洞分析过程中经常需要对内核模块进行调试。在通常情下对于驱动程序的调试是利用最直接的方式即打印调试的方式，在驱动程序中通过 `printk`，加入调试信息。同时通过动态加载模块的方式，即可实现对驱动的动态调试，这也是最简单的调试方式。而对于内核漏洞的分析，由于 linux 系统是开源项目，所有不管对于应用程序的调试还是对内核驱动程序的调试都可以通过查看源码找到漏洞的触发点。

那如果想像调试用户态应用程序一样对内核驱动做动态的源码即调试或是更进一步的对驱动程序进行汇编级调试或是开发内核漏洞利用程序那又该怎么办呢？也许有人会说一般没有必要进行汇编级调试。但是在对内核漏洞利用过程中经常需要调试内核驱动程序，并且需要对内核驱动进行汇编指令级单步跟踪，这样才能确定程序的走向。或是我们需要构造特殊的指令块来完成某项功能。这样就对我们调试内核带来的新的挑战。

那内核里面又是怎么实现的呢，又如果能够去跟进内核内部去调试呢？

本文就是要解决这个问题：在动态汇编调试用户态应用程序的同时，能够跟进应用程序

的系统调用接口，直接源码级或是汇编级的调试（如果没有符号表）调试驱动程序。

本文演示的程序是通过一个应用程序 demo，调用自己写的一个驱动程序接口，通过在调试应用程序的时候能够跟进调试驱动程序。搭建这样的环境我们使用了 vmware 虚拟机，该虚拟机使用普遍，安装简单。为了能够调试程序，需要一个目标机和一个客户机。

目标机是用来安装驱动程序，同时运行应用程序，应用程序会调用驱动程序中的接口。同时目标机自己调试应用程序（用户态使用 GDB 调试）。

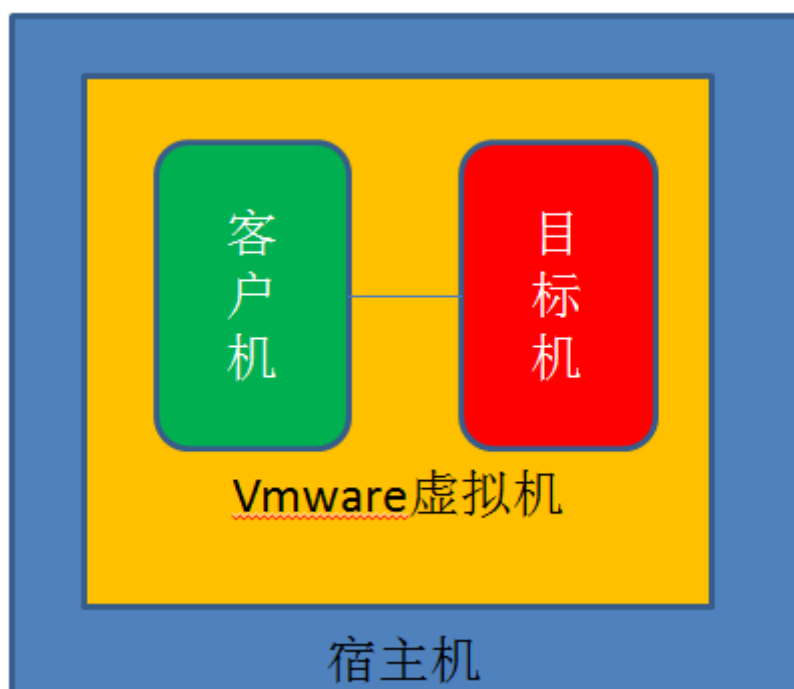
客户机是用来连接客户机，同时在客户机中调试目标机中的驱动程序（使用 GDB 调试）。

需要注意几点：

1. 为了能够能够调试目标机的驱动程序，要求目标机需要支持 KGDB 调试。
2. 为了能让客户机与目标机通信，我们在 vmware 中配置这两台机器通过串口通信调试。
3. 客户机如果要支持驱动的源码级调试需要将驱动程序的符号表加载的客户机的调试器中。
4. 本文用的 vmware 虚拟机需要将目标机和客户机同时安装在虚拟机中。
5. 由于在文章中会包含客户机和目标机的操作过程和调试过程，**本文中会使用绿框表示客户机相关操作，使用红框表示目标机上的相关操作。**

1 基础环境搭建

下图是一个调试应用程序与驱动程序的一个框图，两个操作系统都安装在虚拟机中，一个客户机，一个目标机，客户机通过串口调试目标机中的驱动程序。



1.1 资源环境

Vmware 虚拟机
Ubuntu 操作系统
Ubuntu 源码


1.2 安装 vmware 虚拟机及 ubuntu 操作系统

这里使用的宿主机 win7 操作系统，所以直接在网上下载了 VMWARE WORKSTATION 10.0.4 版本的虚拟机。

安装虚拟机过程就省略。。。。

在安装完虚拟机后需要在虚拟机中安装好 ubuntu 操作系统，在虚拟机中安装 ubuntu 操作系统也省略。。。。

上面说到目标机和客户机两个操作系统，而这里我们在虚拟机里只安装了一个操作系统，先别急，后面就知道了。接下来从官网下载一个内核源码，如下图：

 https://www.kernel.org							
longterm:	3.18.48 [EOL]	2017-02-08	[tar.xz]	[pgp]	[patch]	[inc. p...	
longterm:	3.16.41	2017-02-26	[tar.xz]	[pgp]	[patch]	[inc. p...	
longterm:	3.12.70	2017-02-01	[tar.xz]	[pgp]	[patch]	[inc. p...	
longterm:	3.10.105	2017-02-10	[tar.xz]	[pgp]	[patch]	[inc. p...	
longterm:	3.4.113	2016-10-26	[tar.xz]	[pgp]	[patch]	[inc. p...	
longterm:	3.2.86	2017-02-26	[tar.xz]	[pgp]	[patch]	[inc. p...	
linux-next:	next-20170309	2017-03-09					

这里下载的是 X86-64 版本的 3.2.86 版本（上面忘了说了，之所以用 X86-64 版本是因为上面装的 ubuntu 操作系统也是这个架构）。

下载完成后将该源码拷贝到 ubuntu 虚拟机中，并解压，如下图所示：

```
root@ubuntu:/home/[redacted]/Downloads# ls
linux-3.2.86  linux-3.2.86.tar
root@ubuntu:/home/[redacted]/Downloads#
```

2 编译内核

为了能够调试驱动程序需要让目标机的操作系统支持调试模式，这样就需要从新编译内核，让目标机支持调试模式。

2.1 配置内核参数

首先进入目录 linux-3.2.86，之后执行命令 make menconfig，如下图所示：

```
root@ubuntu:/home/[redacted]/Downloads# cd linux-3.2.86
root@ubuntu:/home/[redacted]/Downloads/linux-3.2.86# make menuconfig
scripts/kconfig/mconf Kconfig
```

就会出现如下图形界面：

```
Linux/x86_64 3.2.86 Kernel Configuration

keys navigate the menu. <Enter> selects submenus --->. Hi
otkeys. Pressing <Y> includes, <N> excludes, <M> modularize
<Esc> to exit, <?> for Help, </> for Search. Legend: [*] bu
odule < > module capable

[*] DMA memory allocation support
  General setup --->
  [*] Enable loadable module support --->
  [*] Enable the block layer --->
  Processor type and features --->
  Power management and ACPI options --->
  Bus options (PCI etc.) --->
  Executable file formats / Emulations --->
[*] Networking support --->
```

这里每一项就是在编译内核之前需要选择的条目，可以根据需要来编译不同的条目。为了能够让操作系统支持内核级调试，需要打开 KGDB 调试开关参数，根据 ubuntu 版本不同该调试开关的位置有所不同。如下图所示：

```
[*] Verbose BUG() reporting (adds 70K)
[*] Compile the kernel with debug info
[ ] Reduce debugging information
[*] Debug VM
```

```
[ ] Debug credential management
[*] Compile the kernel with frame pointers
[*] Delay each boot printk message by N millis
```

```
[*] Sample kernel code --->
[*] KGDB: kernel debugger --->
<M> Test kstrt*() family of functions a
[*] Filter access to /dev/mem
```

为了能够支持 KGDB 调试上面这几项都需要选择上，在内核驱动调试过程中需要在驱动中下断点，这样就需要在内核地址上进行写操作，所以需要将下面这个选项去掉

```
[ ] Export kernel pagetable layout to userspace via debugfs
[ ] Write protect kernel read-only data structures
[ ] Set loadable kernel module data as NX and text as RO
< > Testcase for the NX non-executable stack feature
```

内核参数设置完成后，保持设置的 config 文件，默认保存文件名为.config 文件，保持在当前目录下。为了确保我们对内核写保护已经禁止，在开始编译内核之前，再检查一次 config 文件，打开.config 文件，如下图：

```
root@ubuntu:/home/.../Downloads/linux-3.2.86# vi .config
```

打开之后直接搜索“RO”找到下面两项：

```
# CONFIG_X86_PTDUMP is not set
# CONFIG_DEBUG_RODATA is not set
# CONFIG_DEBUG_SET_MODULE_RONX is not set
# CONFIG_DEBUG_NX_TEST is not set
CONFIG_IOMMU_DEBUG=y
# CONFIG_IOMMU_STRESS is not set
# CONFIG_IOMMU_LEAK is not set
```

确保红框中的两项是注释状态，如果不是注释状态可以直接在这里修改将他们的值改成 N。这样基本上就完成了内核参数的配置。

2.2 编译内核

保持好设置后，编译内核：

Make

Make bzImage

Make modules

Make modules_install

Make install

此时在当前目录下产生新的内核模块 vmlinuz

```

root@ubuntu:/home/[redacted]/Downloads/linux-3.2.86# ls -l vmlinux
-rwxr-xr-x 1 root root 138406982 Mar  8 00:25 vmlinux
root@ubuntu:/home/[redacted]/Downloads/linux-3.2.86#

```

同时在/boot/目录下产生新的内核系统，符号表等信息，如下图

```

12288 Mar  9 07:08 grub
112116729 Mar  8 00:38 initrd.img-3.2.86
139384 Mar  8 00:37 config-3.2.86
2854385 Mar  8 00:37 System.map-3.2.86
5016304 Mar  8 00:37 vmlinuz-3.2.86
14170860 Mar  4 02:44 initrd.img-3.2.0-23-generic
4965840 Apr 25 2012 vmlinuz-3.2.0-23-generic
2884358 Apr 10 2012 System.map-3.2.0-23-generic
791023 Apr 10 2012 abi-3.2.0-23-generic
140279 Apr 10 2012 config-3.2.0-23-generic
176764 Nov 27 2011 memtest86+.bin
178944 Nov 27 2011 memtest86+_multiboot.bin

```

红框是新文件，绿框是之前老的内核文件。

```

root@ubuntu:/boot# uname -a
Linux ubuntu 3.2.86 #6 SMP Wed Mar  8 00:24:52 PST 2017 x86_64 x86_64 x86_64

```

内核更新到了最新的版本，说明我们的内核已经编译成功。

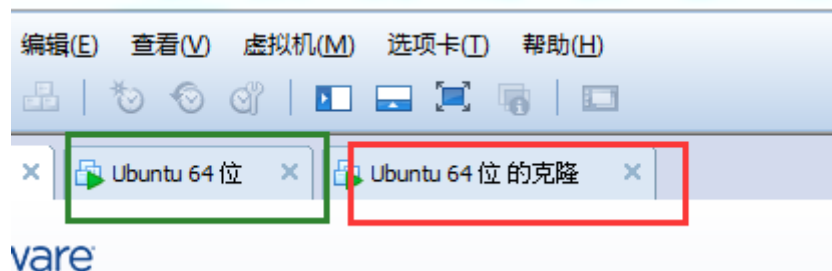
3 配置双机通信

驱动调试需要目标机与客户机两台虚拟机。在这里用了一个取巧的方式，不是安装两台虚拟机，而是直接将上面编译出来的 ubuntu 操作系统直接在 vmware 克隆一份，这样就有了两台 ubuntu 虚拟机，一台作为目标机，一台作为客户机，当然这两个系统都支持了 kgdb 调试模式，都使用了相同的内核。

3.1 双机串口通信

所以，将编译好的 ubuntu 关闭，然后利用 vmware 克隆功能，克隆一份（如果不关闭

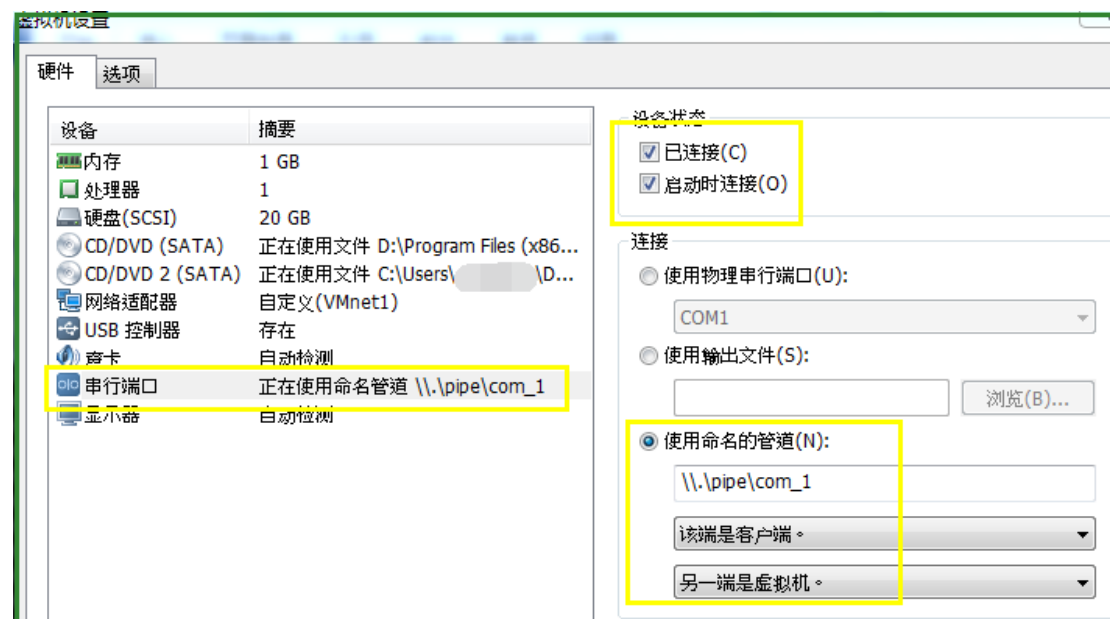
虚拟机没法克隆)。



将克隆的虚拟机作为目标机，将最开始编译的虚拟机作为客户机。

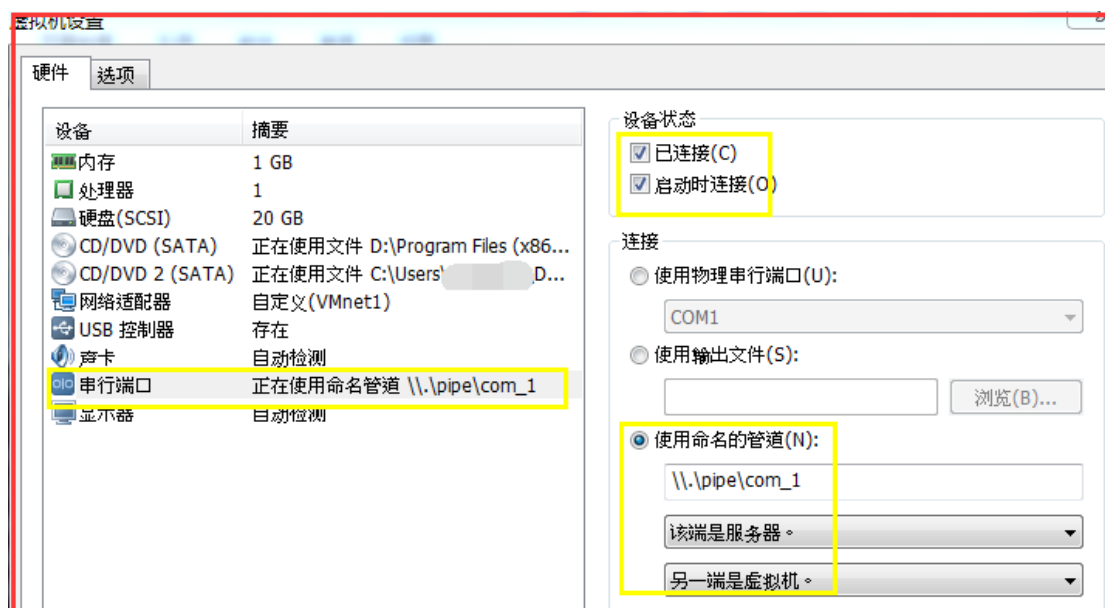
上面的截图都不分目标机，客户机因为都是在操作一个 ubuntu 操作系统，完成克隆后，有了目标机和客户机。所以下面的操作会用绿色框表示客户机，使用红色框表示目标机。

1. 配置客户机，两台机器采用串口通信配置客户机串口，如下图：



需要注意的是，在安装虚拟机的时候，会默认安装上并口，而没有串口，此时需要我们先将并口删除，然后再添加串口，并安装图中显示配置串口。

2，配置目标机，目标机作为服务端也要配置，如下图所示：



对于目标机端也同样需要注意，如果有并口，需要先删除并口，再添加一个串口，并进行相应配置。

3.2 验证串口通信配置

在配置完串口后可以验证一下配置的串口是否起作用。启动客户机和目标机，在一端向串口输入数据，在另一端接受数据，这里我们选择目标机输入数据，客户机接收数据。如下图所示：

```
root@ubuntu:/home# echo helloworld > /dev/ttyS0
root@ubuntu:/home#

root@ubuntu:/home# cat /dev/ttyS0
helloworld
```

当然此过程是先让客户机打开接收，再让目标机发送，这样客户机才能接收到数据。从上图可以看出，串口正常传输数据没有问题。

4 配置串口调试

上面的配置完成后,相当于在两台虚拟机之间连了一根串口线,如果想让两个系统之间通过串口线调试,还需要配置串口调试模式。

4.1 客户机调试配置

启动客户机系统,采用 root 模式登陆系统,修改 grub 启动配置文件。如下图

```
root@ubuntu:/home# cat /etc/default/grub |more
# If you change this file, run 'update-grub' afterwards to update
# /boot/grub/grub.cfg.
# For full documentation of the options in this file, see:
#   info -f grub -n 'Simple configuration'

GRUB_DEFAULT=0
#GRUB_HIDDEN_TIMEOUT=0
GRUB_HIDDEN_TIMEOUT_QUIET=false
GRUB_TIMEOUT=10
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash kgdboc=ttyS0,115200"
GRUB_CMDLINE_LINUX=""
```

黄框中的内容表示要串口连接,当然加在下面一项的“GRUB_COMLINE_LINUX”中也可以。

配置完后需要更新一下 grub,让配置生效:

Update-grub

```
root@ubuntu:/home# update-grub
Generating grub.cfg ...
Found linux image: /boot/vmlinuz-3.2.86
Found initrd image: /boot/initrd.img-3.2.86
Found linux image: /boot/vmlinuz-3.2.0-23-generic
Found initrd image: /boot/initrd.img-3.2.0-23-generic
Found memtest86+ image: /boot/memtest86+.bin
done
root@ubuntu:/home#
```

这样 grub 就完成了更新，重启设备后就会加载串口通信。Grub 更新配置后，会自动修改/boot/grub/grub.cfg 文件，如下图所示：

```
menuentry 'Ubuntu, with Linux 3.2.86' --class ubuntu --class
recordfail
gfxmode $linux_gfx_mode
insmod gzio
insmod part_msdos
insmod ext2
set root='(hd0,msdos1)'
search --no-floppy --fs-uuid --set=root 1abdc90d-611a-
linux /boot/vmlinuz-3.2.86 root=UUID=1abdc90d-611a-
splash kgdboc=ttyS0,115200 svt_handoff
initrd /boot/initrd.img-3.2.86
```

记住，直接修改 grub.cfg 是不行，而是如果/etc/default/grub 更新后，如果运行 update-grub 就又会更新一下 grub.cfg，导致直接在 grub.cfg 中的配置失效（当然只改变 /etc/default/grub，而没有运行 Update-grub 命令就不是使 grub.cfg 失效）。

这样客户端口的串口通讯模块就配置好了。

4.2 目标机调试配置

配置目标机和配置客户机基本一致，不过也有一些差别。

启动客户机系统，采用 root 模式登陆系统，修改 grub 启动配置文件。如下图

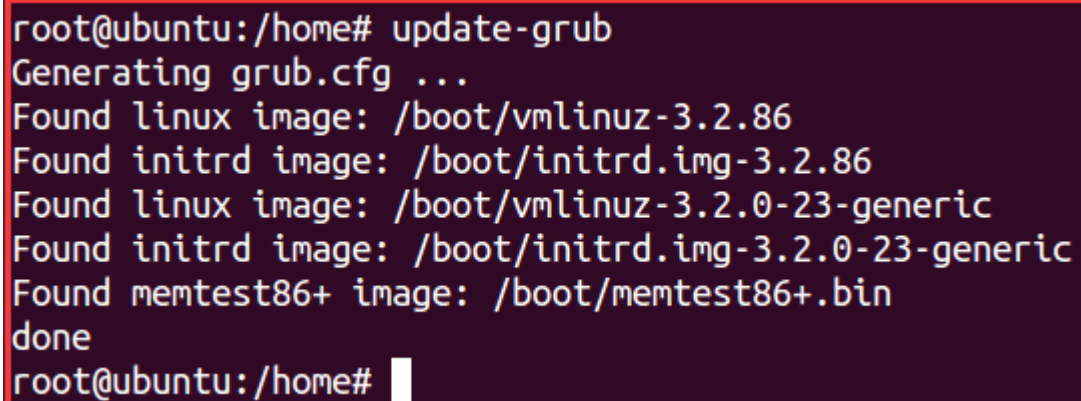
```
root@ubuntu:/home# cat /etc/default/grub
# If you change this file, run 'update-grub' afterwards to update
# /boot/grub/grub.cfg.
# For full documentation of the options in this file, see:
#   info -f grub -n 'Simple configuration'

GRUB_DEFAULT=0
#GRUB_HIDDEN_TIMEOUT=0
GRUB_HIDDEN_TIMEOUT_QUIET=false
GRUB_TIMEOUT=10
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash text kgdboc=ttyS0,115200"
GRUB_CMDLINE_LINUX=""
```

在这里与客户机配置比多加了一个参数 `text` ,这个参数的意思是系统启动后以 `text` 界面而不是图形界面显示 (当然这个不是必须的 , 但是作为目标机我们进入系统直接用 `text` 界面界面就可以了)。

配置完后需要更新一下 `grub` , 让配置生效 :

`Update-grub`

A terminal window with a dark background and red border. The text shows the command 'update-grub' being executed. The output indicates that GRUB configuration files are being generated and existing kernel and initrd images are being found. The process completes with the word 'done'.

```
root@ubuntu:/home# update-grub
Generating grub.cfg ...
Found linux image: /boot/vmlinuz-3.2.86
Found initrd image: /boot/initrd.img-3.2.86
Found linux image: /boot/vmlinuz-3.2.0-23-generic
Found initrd image: /boot/initrd.img-3.2.0-23-generic
Found memtest86+ image: /boot/memtest86+.bin
done
root@ubuntu:/home#
```

更新完成后 , 自动修改 `/boot/grub/grub.cfg` 文件 , 如下图所示

```

menuentry 'Ubuntu, with Linux 3.2.86' --class ubuntu --class gnu-
class os {
    recordfail
    gfxmode $linux_gfx_mode
    insmod gzio
    insmod part_msdos
    insmod ext2
    set root='(hd0,msdos1)'
    search --no-floppy --fs-uuid --set=root 1abdc90d-611a-4b5d
    linux /boot/vmlinuz-3.2.86 root=UUID=1abdc90d-611a-4b5d
    quiet splash text kgdboc=ttyS0,115200 $vt_handoff
    initrd /boot/initrd.img-3.2.86
}

menuentry 'Ubuntu, with Linux 3.2.86---wait' --class ubuntu --clas
gnu --class os {
    recordfail
    gfxmode $linux_gfx_mode
    insmod gzio
    insmod part_msdos
    insmod ext2
    set root='(hd0,msdos1)'
    search --no-floppy --fs-uuid --set=root 1abdc90d-611a-4b5d
    linux /boot/vmlinuz-3.2.86 root=UUID=1abdc90d-611a-4b5d
    quiet splash text kgdbwait kgdboc=ttyS0,115200 $vt_handoff
}

```

图中已经更改了配置。从上图看又多出来了一个“Ubuntu,with Linux 3.2.86---wait”选项，这个选项是从上面那个选项复制下来，同时在里面又添加了新的“kgdbwait”标签，添加一个新的选项就是在 grub 启动时多了一个启动项，添加“kgdbwait 参数就为了在系统刚启动时就可以进入调试模式。而对于上面一个启动选项没有添加该参数，所以在系统启动后才能进行调试。这样目标机就支持了两种调试，一种是系统刚开始启动时调试内核，一种是系统启动后调试内核或驱动。

到目前为止我们的调试模式已经建立好了。

5 GDB 双机调试环境

从新启动目标机进入 grub，会看到多出了的启动选项如下图所示。

```
Ubuntu, with Linux 3.2.86
Ubuntu, with Linux 3.2.86---wait
Ubuntu, with Linux 3.2.86 (recovery mode)
Previous Linux versions
Memory test (memtest86+)
Memory test (memtest86+, serial console 115200)
```

如果我们选择“Ubuntu,with Linux 3.2.86---wait”启动模式，系统就会进入如下图模式，等待远程调试器的连接，也就是客户机上的 GDB 调试器的连接。

```
[ 2.025861] kgdb: Waiting for connection from remote gdb...
Entering kdb (current=0xffff88003d6a8000, pid 1) on processor 0 due to Keyboard Entry
[0]kdb> _
```

到这里说明对目标机的配置没有问题，可以启动客户机去连接调试。

由于是调试自己写的应用程序和驱动程序，而驱动程序又是通过动态加载。所以没必要在系统刚启动时就对系统调试。只要在系统启动后，动态加载模块后，再调试模块也不晚。所以重启目标机，进入“Ubuntu,with Linux 3.2.86”启动模式，直接启动系统。

5.1 驱动代码及编译

在调试代码之前，先看一下我们应用程序和驱动所实现的功能。在目标机上实现了程序代码。

首先看看驱动程序代码，由于本文只是讲解驱动程序调试，所以对驱动程序代码只是实现了一个小的功能，驱动程序中的文件打开功能，如下图：

```

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/device.h>

#define DEVICE_NAME "drv1"

static int device_open(struct inode *, struct file *);
static struct class *class;
static int major_no;

static struct file_operations fops = {
    .open = device_open
};

static int device_open(struct inode *i, struct file *f) {
    printk(KERN_INFO "device opened!\n");
    printk(KERN_INFO "device opened return!\n");
    return 0;
}

static int m_init(void) {
    printk(KERN_INFO "drv1 init.....\n");
    major_no = register_chrdev(0, DEVICE_NAME, &fops);
    class = class_create(THIS_MODULE, DEVICE_NAME);
    device_create(class, NULL, MKDEV(major_no, 0), NULL, DEVICE_NAME);
    return 0;
}

static void m_exit(void) {
    device_destroy(class, MKDEV(major_no, 0));
    class_unregister(class);
    class_destroy(class);
    unregister_chrdev(major_no, DEVICE_NAME);
    printk(KERN_INFO "Driver unloaded\n");
}

module_init(m_init);
module_exit(m_exit);
MODULE_LICENSE("GPL");

```

代码很简单，就是生成一个驱动程序 drv1,该驱动程序实现一个接口，就是打开功能，在应用程序中调用该驱动的系统调用时就会打印两条信息“device opened”“device opened return!”。如果驱动加载成功会显示“drv1 init.....”，如果驱动加载失败会显示“Driver unloaded”。

编写 Makefile 文件编译该驱动程序


```

root@ubuntu:/home/test/drv1# cat Makefile
obj-m += drv1.o

CC=gcc
ccflags-y += "-g"
ccflags-y += "-O2"

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

```

直接编译生产 drv.ko 文件：

```

root@ubuntu:/home/test/drv1# make
make -C /lib/modules/3.2.86/build M=/home/test/drv1 modules
make[1]: Entering directory `/home/jackson/Downloads/linux-3.2.86'
CC [M] /home/test/drv1/drv1.o
Building modules, stage 2.
MODPOST 1 modules
CC /home/test/drv1/drv1.mod.o
LD [M] /home/test/drv1/drv1.ko
make[1]: Leaving directory `/home/jackson/Downloads/linux-3.2.86'
root@ubuntu:/home/test/drv1#

```

5.2 应用程序及编译

最后看一下应用程序如下图：

```

root@ubuntu:/home/test/drv1# cat drv1_app.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

#define DEVICE_PATH "/dev/drv1"
int main(int argc, char *argv[])
{
    int fd;
    printf("open function\n");
    fd = open(DEVICE_PATH, O_RDONLY);
    if (fd == -1) {
        perror("open");
    }
    printf("open function finish\n");
    return 0;
}
root@ubuntu:/home/test/drv1# _

```

应用程序只是调用了驱动程序的打开功能，我们关心的是如何搭建调试环境，所以

demo 没有提供更多的代码。驱动程序默认加载路径为/dev/下面，所以应用程序打开的文件为/dev/drv1，在系统调用前后打印信息，能够直观感受打印是否成功。

编译该应用程序

```
Gcc drv1_app.c -o drv1_app
```

5.3 验证驱动程序

在调试驱动程序之前，先要验证驱动程序，保证应用程序能够正确调用驱动程序的接口。

在应用程序系统调用之前，要保证驱动模块已经被加载到了内核。

从上面的驱动打印可以看出在驱动加载完成后会先打印“drv1 init....”，那接下来加载驱动程序并验证该驱动程序是否加载成功：

```
root@ubuntu:/home/test/drv1#  
root@ubuntu:/home/test/drv1# dmesg | grep -E "(init\\.|\\.|\\.|\\.|return|unloaded|finish)"  
root@ubuntu:/home/test/drv1# insmod drv1.ko  
root@ubuntu:/home/test/drv1# dmesg | grep -E "(init\\.|\\.|\\.|\\.|return|unloaded|finish)"  
[ 362.094147] drv1 init.....  
root@ubuntu:/home/test/drv1#
```

如上图，先查看 dmesg 信息，在驱动为加载前没有任何信息输出，在加载模块后，打印出来了“/drv1 init....”，表明驱动已经加载成功，之后我们运行应用程序。

通过应用程序代码我们已经知道了，如果能在系统调用前后打印出信息，则系统调用接口 open 已经被执行，从下图可知信息被正确打印。

```
root@ubuntu:/home/test/drv1# ./drv1_app  
open function  
open function finish  
root@ubuntu:/home/test/drv1# _
```

在驱动内部，对 open 函数的调用也会打印相关信息，包括卸载模块时的打印信息，如下图所示

```
root@ubuntu:/home/test/drv1# dmesg | grep -E "(init\\.\\.\\.\\.|return|unloaded|finish)"
[ 362.094147] drv1 init.....
[ 520.974083] device opened return!
root@ubuntu:/home/test/drv1# rmmod drv1
root@ubuntu:/home/test/drv1# dmesg | grep -E "(init\\.\\.\\.\\.|return|unloaded|finish)"
[ 362.094147] drv1 init.....
[ 520.974083] device opened return!
[ 950.531185] Driver unloaded
root@ubuntu:/home/test/drv1#
```

由此可知，我们的驱动程序和应用程序都没有问题。

6 调试应用程序及驱动程序调用接口

接下来就到了最关键的一步，就是如何利用现有的环境去调试应用程序的同时也能够调试驱动程序。

6.1 配置驱动调试

重新启动目标机，启动完成后加载驱动程序（具体过程已经演示要一遍了）。

启动客户机，由于我们客户机和目标机是同一版本系统。所以内核模块也一致。进入包含内核模块的路径，gdb 调试内核模块，如下图所示：

同时设置串口信息（这一步不是必须的，因为我们的客户机启动时已经设置了串口调试，如果客户机启动时没有设置串口调试可以在这里设置，但是目标机必须在系统启动时设置串口调试模式，当然也不是必须的，够绕吧，除非你会在目标机启动进入系统后设置串口，否则你就要在目标机启动时设置串口调试，目标机启动进入系统后怎么设置串口调试，自己网上查去我也没查，我也不知道），设置串口调试，如下图：

```

root@ubuntu:/home/ /Downloads/linux-3.2.86# gdb vmlinux
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show co
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/jacksun/Downloads/linux-3.2.86/vmlinux...
(gdb) set remotebaud 115200
(gdb)

```

设置完成后，就需要先打开目标机调试模式（要不然客户机调试谁呀），打开目标机调试模式，如下图所示：

```

root@ubuntu:/home/test/drv1#
root@ubuntu:/home/test/drv1# echo g > /proc/sysrq-trigger _

```

输入该命令后回车，目标机就会进入假死状态，目标机没有任何反应，这是正常现象，这是在等待调试器的链接。我们这个时候就要返回到客户机中启动串口调试如下图：

```

Reading symbols from /home/jacksun/Downloads/linux-3.2.
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
kgdb_breakpoint () at kernel/debug/debug_core.c:955
955          wmb(); /* Sync point after breakpoint
(gdb)

```

客户机启动调试后就会出现如上信息。此时客户机与目标机的调试环境就建立成功了。这样就可以进行驱动调试了，为了能够查看到驱动中的符号信息，需要在客户机中加载符号文件。客户机加载符号文件如下：

首先因为要调试的是目标机的驱动程序，需要知道在目标机中模块的加载地址。

由于目前目标机还处在调试模式假死状态，所以目标机还不能做任何动作。为了能够让目标机继续运行，需要在客户机中让目标机可以继续运行，这样在客户机的 GDB 调试中，就可以像调试普通应用程序一样让目标级继续运行，如下图：

```

Reading symbols from /home/jacksun/Downloads/linux-3.2.
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
kgdb_breakpoint () at kernel/debug/debug_core.c:955
955          wmb(); /* Sync point after breakpoint */
(gdb) c
Continuing.

```

此时目标机就回复到了运行模式，如下图所示

```

root@ubuntu:/home/test/drv1# echo g > /proc/sysrq-trigger
[ 65.576193] SysRq : DEBUG

Entering kdb (current=0xffff880019da96f0, pid 2380) on processor
[0]kdb> root@ubuntu:/home/test/drv1# _

```

此时就可以查看驱动的加载基地址。如下图两种查看方式

```

root@ubuntu:/home/test/drv1# cat /proc/modules |grep drv1
drv1 1652 0 - Live 0xffffffffa008f000 (0)
root@ubuntu:/home/test/drv1# cat /sys/module/drv1/sections/.text
0xffffffffa008f000
root@ubuntu:/home/test/drv1#

```

两种方式均可查到加载基地址。

此时就可以在客户机中加载符号表，由于目前目标机运行，所以需要先将目标机断下来

才能让客户机处于调试模式，所以目标机再一次下断下来

```

root@ubuntu:/home/test/drv1# cat /sys/module/drv1/sections/.text
0xffffffffa008f000
root@ubuntu:/home/test/drv1# echo g > /proc/sysrq-trigger _

```

这样目标机就又处于假死状态，等待客户机的调试，同时客户机又进入调试模式，如下图：

```

955          wmb(); /* Sync point
(gdb) c
Continuing.

Program received signal SIGTRAP, Trap
kgdb_breakpoint () at kernel/debug/c
955          wmb(); /* Sync point
(gdb)

```

加载符号文件同时为驱动的 Open 函数设置断点，如下图所示

```

(gdb) add-symbol-file /home/drv1/drv1.ko 0xfffffffffa008f000
Add symbol table from file "/home/drv1/drv1.ko" at
      .text_addr = 0xfffffffffa008f000
y or n) y
Reading symbols from /home/drv1/drv1.ko...done.
(gdb) break device_oper
Breakpoint 1 at 0xfffffffffa008f000: file /home/jacksun/test/drv1/drv1.c, line 33.
(gdb)

```

需要注意的是不管是图中的驱动文件还是驱动程序源文件都应该是位于客户机中(所以, 前面忘了介绍, 要把目标机编译好的驱动文件及源文件, 拷贝一份给客户机), 需要注意的是客户机中拷贝过来的驱动文件可以放在任何位置, 而源文件的位置应该和目标机中源文件位置保持一致, 这样才能进行源码级调试(上面黄框中源代码的位置表示是在目标机中的位置, 所以客户机驱动的源码也应该放在这样的位置, 这样驱动调试器才能找到符号位置。如果对驱动程序不进行源码调试, 只进行汇编调试就不需要源文件), 而加载地址是目标机的驱动加载基地址, 这一点千万别搞错了。

在给 device_open 函数下断点后断点又指向了目标机的驱动文件源文件 drv1.c , 入上图。到此为止驱动加载完成, 符号文件加载完成。就可以在目标机启动应用程序并调试, 同时在客户机上调试驱动程序。

客户机让调试机继续运行, 如下图:

```

(gdb) break device_open
Breakpoint 1 at 0xfffffffffa008f000: file /home/jacksun/test/drv1/drv1.c, line 33.
(gdb) c
Continuing.

```

6.2 调试应用程序

此时在目标机已经解除假死状态, 可以在目标机中启动 gdb 调试应用程序。如下图。

```

root@ubuntu:/home/test/drv1# gdb drv1_app
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/test/drv1/drv1_app...(no debugging symbols found)
(gdb)

```

应用程序的调试应该不用多说。我们在系统调用处下一个断点，运行到系统调用为止

```

root@ubuntu:/home/test/drv1# gdb drv1_app
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/test/drv1/drv1_app...(no debugging symbols found)
(gdb) b main
Breakpoint 1 at 0x400588
(gdb) x/15i main
0x400584 <main>: push    %rbp
0x400585 <main+1>: mov     %rsp,%rbp
0x400588 <main+4>: sub     $0x20,%rsp
0x40058c <main+8>: mov     %edi,-0x14(%rbp)
0x40058f <main+11>: mov     %rsi,-0x20(%rbp)
0x400593 <main+15>: mov     $0x4006cc,%edi
0x400598 <main+20>: callq   0x400460 <puts@plt>
0x40059d <main+25>: mov     $0x0,%esi
0x4005a2 <main+30>: mov     $0x4006da,%edi
0x4005a7 <main+35>: mov     $0x0,%eax
0x4005ac <main+40>: callq   0x400480 <open@plt>
0x4005b1 <main+45>: mov     %eax,-0x4(%rbp)
0x4005b4 <main+48>: cmpl    $0xffffffff,-0x4(%rbp)
0x4005b8 <main+52>: jne     0x4005c4 <main+64>
0x4005ba <main+54>: mov     $0x4006e4,%edi
(gdb) _

```

单步运行到 open 函数调用处 0x4005ac :

```

(gdb) r
Starting program: /home/test/drv1/drv1_app

Breakpoint 1, 0x000000000400588 in main ()
(gdb) ni
0x00000000040058c in main ()
(gdb)
0x00000000040058f in main ()
(gdb)
0x000000000400593 in main ()
(gdb)
0x000000000400598 in main ()
(gdb)
open function
0x00000000040059d in main ()
(gdb)
0x0000000004005a2 in main ()
(gdb)
0x0000000004005a7 in main ()
(gdb)
0x0000000004005ac in main ()
(gdb)

```

运行到 0x4005ac 处有就进入了假死状态，程序不能再运行，这是因为已经在客户机上将目标机的驱动程序中的 device_open 函数设置了断点。

6.3 调试驱动程序

返回到客户机，客户机已经处于调试模式如下图：

```

Breakpoint 1, device_open (i=0xffff88003b21ed58, f=0xffff88003b8e1600)
at /home/jacksun/test/drv1/drv1.c:33
33 static int device_open(struct inode *i, struct file *f) {
(gdb)

```

这就进入了驱动程序的调试模式。和调试普通应用程序一样对驱动就可以进行调试，由于前面已经加载了符号文件，可以查看驱动源码，并单步调试，如下图：


```
(gdb) l
28     static int major_no;
29
30     static struct file_operations fops = {
31         .open = device_open
32     };
33     static int device_open(struct inode *i, struct file *f) {
34         printk(KERN_INFO "device opened!\n");
35         printk(KERN_INFO "device opened re
36         return 0;
37     }
(gdb) n
34         printk(KERN_INFO "device opened!\n");
(gdb)
35         printk(KERN_INFO "device opened re
(gdb)
37     }
(gdb) ni
0xfffffffffa008f027    37    }
(gdb) ni
0xfffffffffa008f028    37    }
(gdb)
```

查看源码

源码调试

单步运行后内核打印信息

还可以汇编级调试

从图中可以现在驱动程序不但可以查看源码还可以源码级调试，汇编级调试。最后在用 continue 让驱动运行起来。这样在目标机上应用程序又可以运行，知道应用程序成功退出，

如图：

```
0x0000000000004005ac in main ()
(gdb)

0x0000000000004005b1 in main ()
(gdb)
0x0000000000004005b4 in main ()
(gdb) c
Continuing.
open function finish
[Inferior 1 (process 2647) exited normally]
(gdb) c
The program is not being run.
(gdb) qui
root@ubuntu:/home/test/drv1#
```

那对于驱动程序调试器与应用程序调试模式正式结束形式如下：

1. 让目标机处于被调试模式即运行 echo g >/proc/sysrq-trigger (如果 GDB 刚好也正在调试目标机中的应用程序则为<gdb>shell echo g >/proc/sysrq-trigger)

2. 此时客户机处于调试模式，在调试模式中直接退出<gdb> quit.此时客户机就推出了调试模式。
3. 目标机也就推出了内核调试模式显示内容为：

```
root@ubuntu:/home/test/drv1# echo g > /proc/sysrq-trigger  
[ 9683.916562] SysRq : DEBUG  
root@ubuntu:/home/test/drv1#
```

4. 驱动程序调试结束。

至此应用程序和驱动程序的调试环境和调试方式就介绍完毕。

备注

1.如果在用 gdb 调试应用程序过程中，突然想回到驱动调试模式，可以在应用程序中直接输入 shell echo g > /proc/sysrq-trigger，这样应用程序调试器就处于假死状态，内核调试器的 GDB 就可以调试了。如果要回复到应用程序调试器 gdb 中，那就直接在内核调试器的 GDB 使用 continue 命令即可，回到应用程序调试器。

2.在调试包含多个函数或是循环函数的驱动程序或是应用程序时，还可以使用 finish 命令来直接运行到函数结束，方便调试。