Linux下的防火墙技术与搭建

袁昊男 2018091618008

【摘要】防火墙是整个数据包要进入主机前的第一道关卡。为了保护网络的安全，Linux系统中提供了可靠的防火墙机制Netfilter，并在此基础上提供了Iptables实现数据包的过滤。本文主要介绍Iptables内的filter和NAT功能，并实现使用Netfilter搭建防火墙。

【关键词】防火墙；Netfilter；Iptables；Linux

1 概述

网络安全除了关注网络上的软件漏洞和安全通报之外，还要求能根据环境的需要，设置符合自身要求的数据包防控机制。防火墙就是可以对流进流出网络的数据包进行分析和过滤一种安全机制，而在Linux系统中本身就提供了Netfilter防火墙功能，该功能分析进入网络的数据包，将数据包的头部数据提取出来，以决定该连接为放行或者抵挡。该方式可以直接分析数据包头部数据，包括硬件地址（MAC）、软件地址（IP）、TCP、UDP以及ICMP等数据包信息，因此用途非常广泛。

Iptables是Linux内部集成的信息包过滤系统，是Linux内核中的一个通用框架，该框架定义了数据包过滤子系统的系统功能，提供了Filter、NAT和Mangle这3个链表，默认使用的是Filter表。每个表中包括若干条内建的链（Chain），用户也可在表中创建自定义链。在每条链中，可定义一条或多条过滤规则（Rule），即链是规则的一个列表。每条规则应指定所要检查的包的特征以及如何处理与之匹配的包的关系，这种处理被称为目标（Target）。目标值可以是用户自定义的一个链名，可以根据该链的名字跳转到同一个表内的链里，以便对该链内的规则进行检查，目标还可以是ACCEPT、DROP、REJECT等。

2 Iptables中的表

**2.1 Filter表**

Filter表主要与进入Linux本机的数据包有关，是Iptables中默认存在的表之一。该表内建有3个链：INPUT链用于处理目标地址是本机的数据包；FORWARD链用于处理要通过或转发的数据包，即目标地址不是本机的数据包；OUTPUT链用于处理本地进程生成的要外发的数据包。当一个数据包从网卡进入防火墙时，内核首先根据路由表决定数据包的目标，根据数据包的目的地址来决定数据包将送往哪一条链进行规则匹配。Filter表中数据包具体的处理流程如下。

发现到达防火墙的数据包的目的地址是本机，则将数据包发送给INPUT链进行处理。当发现数据包的目的地址不是本机，则会检查内核设置是否允许数据包的转发，如果允许，则将数据包交给FORWARD链进行处理，如果不允许转发，数据包就会被丢弃。如果到达防火墙的数据包是由内部网络的进程产生的，则会交给OUTPUT链进行处理。

**2.2 NAT表**

NAT的全名是Network Address Translation，即网络地址转换。私有地址可以在不同的企业网内部重复使用，虽说现在使用IPv6技术，缓解了IP地址短缺的问题，但IPv4私有地址方便管理，企业内部仍然需要，但使用私有地址的主机不能访问互联网。利用NAT，可以实现私有地址与公有地址的互相转换。

NAT表内建有PREROUTING、POSTROUTING、OUTPUT共3个链。

**2.2.1 PREROUTING**

PREROUTING这个链在Filter表的最前面，当一个数据包来到Linux的网络接口时，首先通过mangle的PREROUTING，然后再通过NAT的PREROUTING，而这条链是数据包在经过路由之前就要过滤的。

**2.2.2 POSTROUTING**

该链是在路由判断之后，如果ACCEPT放行了这个包，那么它将进入POSTROUTING部分；如果是转发，应该要再次进行路由选择然后将其送出，此时Linux系统已经为该数据包选择好路由，并找到合适的接口送出该数据包。在该链里要进行一个非常重要的动作称为SNAT，即修改源IP地址。

**2.2.3 OUTPUT**

对本地进程产生并准备发出的数据包由OUTPUT链进行检查处理，可以在该链进行DNAT操作，即修改目的IP地址。

3 Linux下防火墙设置实例

理解一项技术的最佳方式是动手实现它。数据包过滤器只能在内核中实现，因此代码需要运行在内核中，也就意味着需要修改内核。Linux提供了两项重要技术，使得无需重新编译整个内核就能实现包过滤器。它们分别是Netfilter和可加载内核模块。

Netfilter在数据包经过的路径上埋放了一些钩子（hook），它们位于内核中。自行编写的函数可以通过内核模块放进内核，并挂在那些钩子上。当数据包到达某个钩子上时，挂在钩子上面的函数就会被调用，可以在函数中对数据包进行审查和过滤，并告诉Netfilter如何处理。

**3.1 可加载内核模块实现**

**3.1.1 编写可加载内核模块**

Linux内核是模块化设计的，因此只有很小一部分被加载进内核。如果需要扩展内核的功能，可以将这些功能设计成内核模块，动态地载入内核。例如，为了支持一个新硬件，可以将它的设备驱动程序作为一个内核模块加载进内核。内核模块是可以根据实时需求载入或卸载的代码块。只有具有root权限或者CAP\_SYS\_MODULE能力的进程才能够载入或者卸载内核模块。每个内核模块都具有两个入口点，一个入口点用于载入模块，另一个用于卸载模块。宏module\_init()和module\_exit()是用来指定这两个入口点的函数。module\_init()指定的函数在模块载入是被调用，用来初始化。module\_exit()指定的函数在模块卸载时被调用，用来进行清理工作。下面的代码展示了如何编写一个可加载内核模块。

【代码3-1-1】一个简单的可加载内核模块（kMod.c）

|  |
| --- |
| 1. #include <linux/module.h> 2. #include <linux/kernel.h> 3. #include <linux/init.h> 5. **static** **int** kmodule\_init(**void**) 6. { 7. printk(KERN\_INFO"Initializing this module\n"); 8. **return** 0; 9. } 11. **static** **void** kmodule\_exit(**void**) 12. { 13. printk(KERN\_INFO"module cleanup\n"); 14. } 16. module\_init(kmodule\_init); //向内核注册模块提供新功能 (1) 17. module\_exit(kmodule\_exit); //注销由模块提供的所有功能 (2) 18. MODULE\_LICENSE("GPL"); //告诉内核该模块具有GNU公共许可证 |

代码中行（1）和行（2）处定义的宏module\_init()和宏module\_exit()分别指向插入、移除内核模块所需执行的函数（如图3-1-1所示）。这两个函数只是简单的打印了一些信息。在内核中打印信息不能用printf()函数把信息答应到屏幕上，而使用printk()函数打印内容至内核日志缓存。



图3-1-1 内核函数执行图示

**3.1.2 编译内核模块**

编译一个内核模块最简单最有效的方式莫过于使用makefile。下面是一个编译可加载内核模块的makefile：

【代码3-1-2】一个编译可加载内核模块的makefile

|  |
| --- |
| CONFIG\_MODULE\_SIG=n  obj-m := kMod.o  all:  make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules clean:  make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd)clean |

为了编译可加载内核模块，必须有一个和本机内核相对应的头文件及配置文件。每个Linux发行版本都提供访问这些文件的途径，这些文件多数保存在usr/src目录中。

在上面的makefile中，参数M表明一个外部模块将要被编译，以及模块生成后应该放在什么位置。选项C用于指定内核库文件的目录。编译结果如下：



图3-1-2 makefile编译结果

**3.1.3 安装内核模块**

内核模块生成后，可以使用insmod、rmmod和lsmod等命令管理内核模块。用户也可以选择使用更加复杂的modprobe命令如下所示。

【代码3-1-3】安装内核模块

|  |
| --- |
| //把内核模块载入内核  $ sudo insmod kMod.ko  //查看 kMod 模块是否已载入内核(也可直接 lsmod)  $ lsmod | grep kMod  kMod 12453 0  //把内核模块从内核中卸载  $ sudo rmmod kMod |

为证实模块已成功执行，可以检查模块的输出。在内核模块程序中，当模块被加载和卸载时会分别向内核日志缓冲区输出一些信息。Linux提供了dmesg命令用于检查内核日志缓冲区。运行该命令，就能找到这些模块输出的信息如下。

【代码3-1-4】证实模块已成功安装

|  |
| --- |
| $ dmesg 也可使用 dmesg > 输出文件名，例 dmesg > 1.txt  [65368.235725] Initializing this module  [65499.594389] Module cleanup |

**3.2 使用Netfilter搭建一个简单的防火墙**

**3.2.1 前置知识**

Linux内核通过Netfilter的钩子函数提供了强大的数据包处理及过滤框架。在Linux中，每个协议栈都在数据包经过的路径上定义了一系列钩子点。开发者使用内核模块将他们定义的函数“钩”到这些钩子点上，当每个数据包均到达钩子点时，协议栈会调用Netfilter框架查找是否有内核帧块挂在这个钩子点。如果有，这些模块注册的函数会被调用，这时它们得以分析或处理数据包。最后，它们将返回对数据包的处理结果。对数据包的处理结果有5种，说明如下。

（1）NF\_ACCEPT。允许数据包通过。

（2）NF\_DROP。丢弃数据包，这样数据包将不会在网络协议栈中继续传输。

（3）NF\_QUEUE。使用nf\_queue机制将数据包传递到用户空间处理。 (4)NF\_STOLEN。告知Netfilter框架忽略这个数据包。这个操作主要是把数据包进一步的处理工作从Netfilter转交给模块。数据包在内核内表中仍然是有效的。典型用途是把分片的数据包存储起来，以便在一个上下文中分析它们。

（5）NF\_REPEAT。请求Netfilter框架再次调用这个模块。Netfilter为IPv4定义了5种钩子函数。数据包在网络协议栈中的传输情况如图3-2-1所示。更详细的内容可参考相关资料。



图3-2-1 5种钩子函数

**3.2.2 实现数据包过滤器**

下面使用Netfilter框架实现一个简单的数据包过滤器。目标是阻止所有发往端口号23的TCP数据包，也就是阻止用户使用telnet连接到其他计算机。首先编写一个回调函数，在这个函数中实现过滤功能，然后将这个函数挂到一个Netfilter的钩子上。代码如下：

【代码3-2-1】数据包过滤器（telnetFilter.c）

|  |
| --- |
| 1. unsigned **int** telnetFilter(**void** \*priv,**struct** sk\_buff \*skb, **const** **struct** nf\_hook\_state \*state) 2. { 3. **struct** iphdr \*iph; 4. **struct** tcphdr \*tcph; 6. iph = ip\_hdr(skb); 7. tcph = (**void** \*)iph + iph->ihl\*4; 8. **if**(iph->protocol == IPPROTO\_TCP && tcph->dest == htons(23)) 9. { 10. printk(KERN\_INFO "Dropping telnet packet to %d %d %d %d\n" 11. ((unsigned **char** \*)&iph->daddr)[0], 12. ((unsigned **char** \*)&iph->daddr)[1], 13. ((unsigned **char** \*)&iph->daddr)[2], 14. ((unsigned **char** \*)&iph->daddr)[3]); 15. **return** NF\_DROP; 16. } 17. **else** 18. { 19. **return** NF\_ACCEPT; 20. } 21. } |

当回调函数telnetFilter()被调用时，它会得到整个数据包（参数skb指向的就是这个数据包）。该函数检查数据包的TCP头部，确定目的端口号是否为23，如果是，就丢弃该数据包，并输出一些信息到内核日志中，如果不是，数据包将被允许通行。

下面把上述回调函数挂到Netfilter的钩子上，可以使用NF\_INET\_LOCAL\_OUT或NF\_INET\_POST\_ROUTING，这两个钩子点都在数据包的流出路径上。在下面的示例代码中，使用NF\_INET\_POST\_ROUTING。

【代码3-2-2】用Netfilter的内核帧块屏蔽telnet（telnetFilter.c）

|  |
| --- |
| 1. #include <linux/kernel.h> 2. #include <linux/module.h> 3. #include <linux/netfilter.h> 4. #include <linux/netfilter\_ipv4.h> 5. #include <linux/ip.h> 6. #include <linux/tcp.h> 8. **static** **struct** nf\_hook\_ops telnetFilterHook; 10. /\* The implementation of the telnet,filter function is omitted here; filter was shown easlier in Listing 2. \*/ 11. **int** setUpFilter(**void**) 12. { 13. printk(KERN\_INFO "Registering a Telnet filter.\n"); 14. telnetFilterHook.hook = telnetFilter; 15. telnetFilterHook.hooknum = NF\_INET\_POST\_ROUTING; 16. telnetFilterHook.pf = PF\_INET; 17. telnetFilterHook.priority = NF\_IP\_PRI\_FIRST; 19. // Register the hook 20. nf\_register\_net\_hook(&init\_net,&telnetFilterHook); 21. **return** 0; 22. } 24. **void** removeFilter(**void**) 25. { 26. printk(KERN\_INFO "Telnetfilter is being removed.\n"); 27. nf\_unregister\_net\_hook(&init\_net,&telnetFilterHook); 28. } 30. module\_init(setUpFilter); 31. module\_exit(removeFilter); 32. MODULE\_LICENSE("GPL"); |

上面的代码构建了一个nf\_hook\_ops结构，其中包括回调函数（telnetFilter）、钩子号（NF\_INET\_POST\_ROUTING）、IPv4协议族（PF.INET）和优先级priority等关键信息。接下来使用nf\_register\_hook()函数把telnetFilter挂到指定的钩子上。其余代码已在前面讨论过。

使用insmod加载模块到内核后，可以尝试用telnet命令登录其他计算机。从下面的执行结果来看，telnet登录失败，使用dmesg命令能看到所有的telnet数据包均被丢弃。如果移除该内核模块，telnet登录能够成功。注意：务必在试验后移除这个内核模块，否则后续实验会受到影响。代码结果如下：

【代码3-2-3】尝试telnet登录

|  |
| --- |
| //把模块加载到内核  $ sudo insmod telnetFilter.ko  $ telnet 10.0.2.5  Tring 10.0.2.5...  telnet: Unable to connet to remote host: Connection timed out  //登录失败  $ dmesg  [1166456.149046] Registering a Telnet filter.  [1166535.962316] Dropping telnet packet to 10.0.2.5  [1166536.958065] Dropping telnet packet to 10.0.2.5  //把模块从内核中卸载  $ sudo rmmod telnetFilter  $ telnet 10.0.2.5  Trying 10.0.2.5...  Connected to 10.0.2.5.  Escape character is ‘^]’.  Ubuntu 12.04.2 LTS  ubuntu login: //登录成功 |

**3.2.3 实现结果**

（1）先将源程序telnetFilter.c通过makefile编译再使用insmod命令将模块加载入内核，用lsmod命令查看结果。

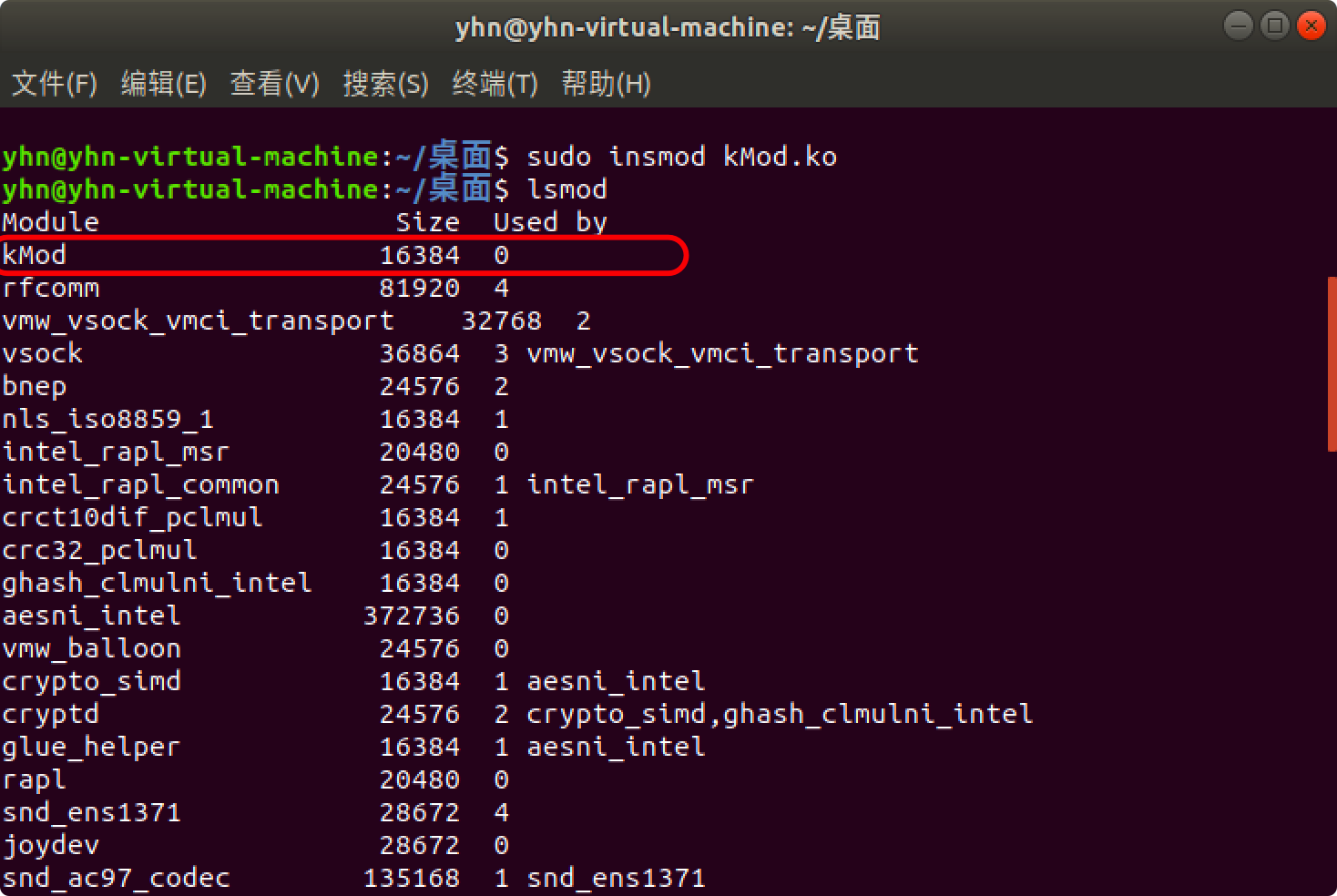


图3-2-2 载入内核结果

（2）选择另一台主句查看IP地址（这里的示例选择的是IP地址为10.0.2.5的kali虚拟机），然后使用telnet命令登录，发现登录失败。

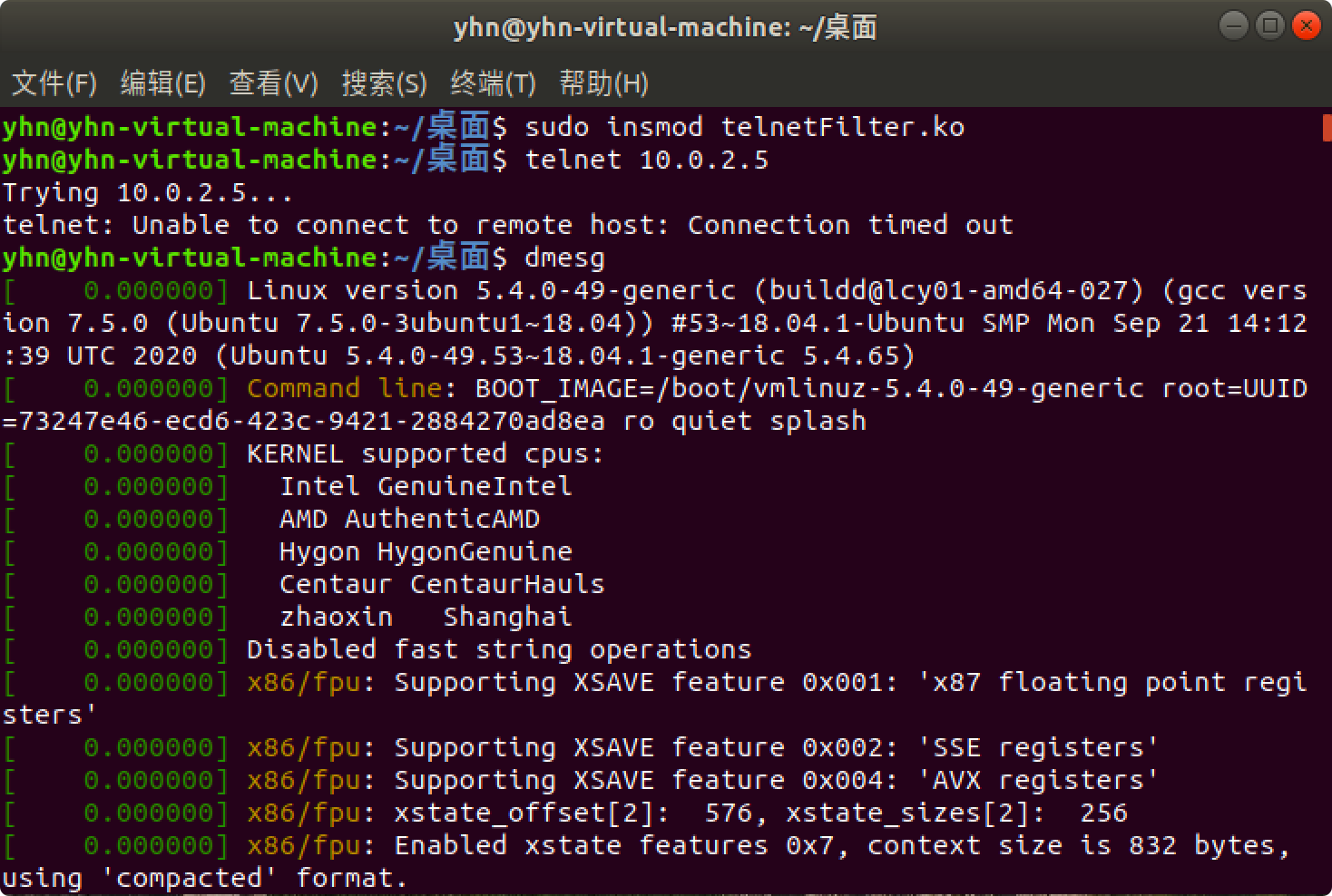


图3-2-3 尝试telnet远程登录

（3）使用dmesg命令检查内核日志缓冲区，查看输出。

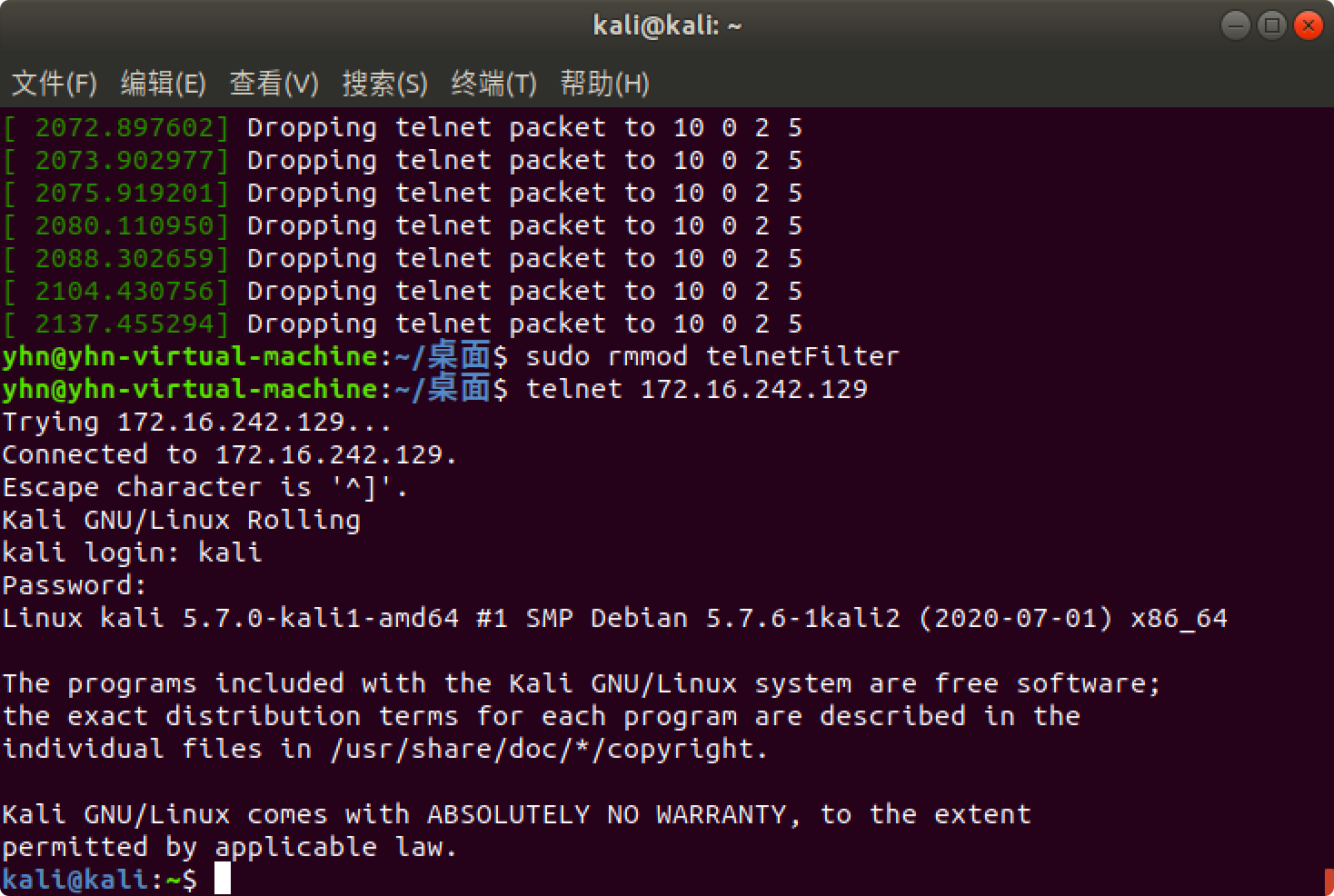


图3-2-4 内核日志缓冲区输出

（4）使用rmmod命令将内核模块从内核中卸载，再用lsmod命令查看原模块已被删除。

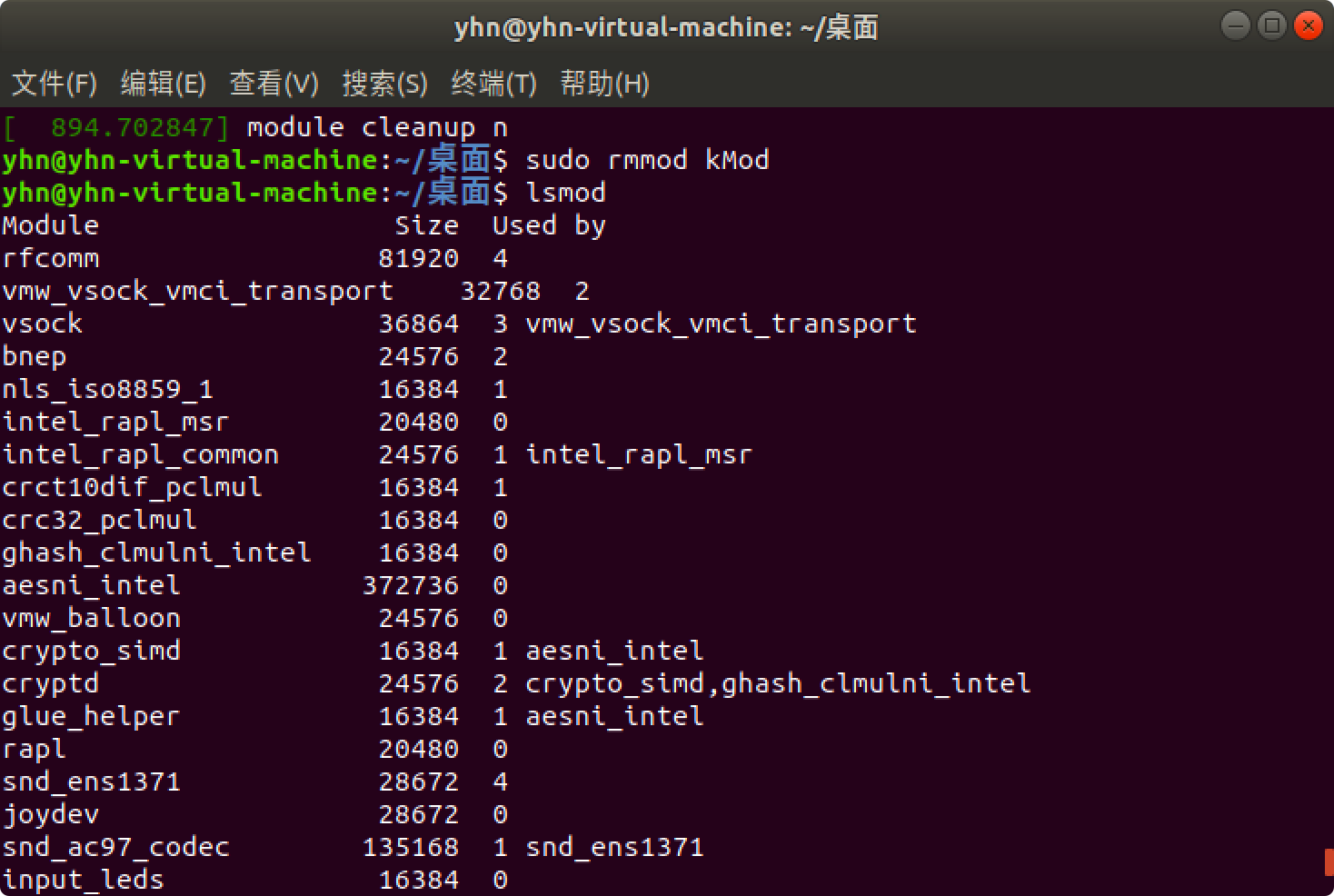


图3-2-5 卸载内核模块结果

（5）模块卸载后，再使用telnet命令登录kali。登录成功。

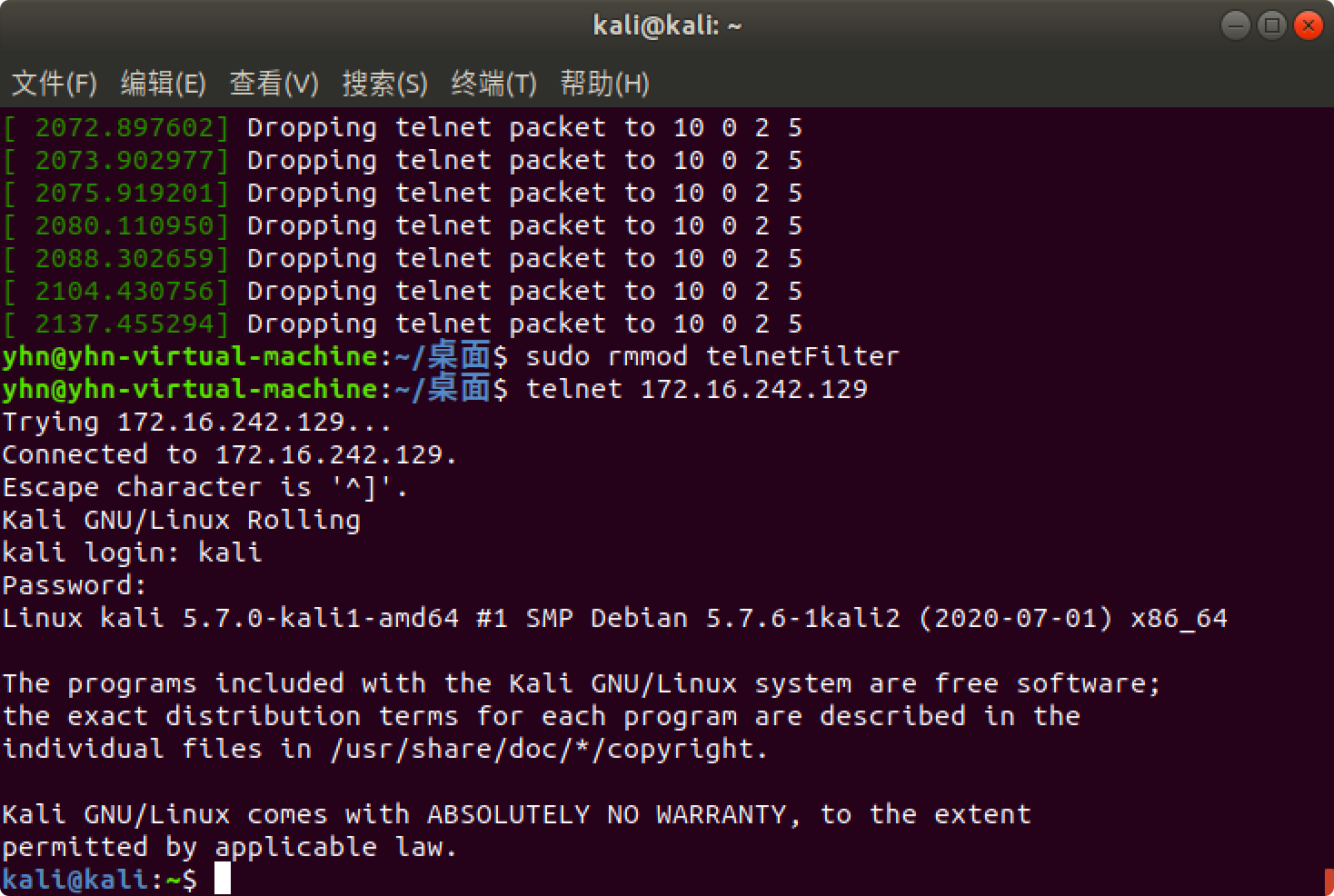


图3-2-6 telnet远程登录成功