|  |
| --- |
| HỌC VIỆN KỸ THUẬT MẬT MÃ  **KHOA CÔNG NGHỆ THÔNG TIN**  ¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯  Logo HvKTMM |
| LẬP TRÌNH NHÂN LINUX  ĐỀ TÀI:  **Dịch, trình bày tóm tắt và thực hiện**  **các bài lab trong networking** |
| ***Giảng viên hướng dẫn:***  **ThS. Triệu Văn Vũ Quân**  ***Sinh viên thực hiện:***  **Nguyễn Văn Hùng CT040122**  **Nguyễn Văn Ninh CT040137**  **Phạm Ngọc Thanh CT040143**  **Nguyễn Đình Trung CT040149**  **Nguyễn Cường Việt CT040155**  ***Nhóm 20***    **Hà Nội, 2023** |
|  |

**MỤC TIÊU CỦA LAB**

* Hiểu kiến trúc mạng trong nhân Linux
* Học cách quản lý gói IP thực tiễn bằng cách sử dụng bộ lọc gói hoặc tường lửa
* Làm quen với cách sử dụng socket ở mức độ nhân Linux

**TỔNG QUAN**

Sự phát triển của Internet đã dẫn đến sự gia tăng mũi nhọn mạnh mẽ trong các ứng dụng mạng và, do đó, đẩy nhanh yêu cầu về tốc độ và năng suất của hệ thống phụ trợ mạng của hệ điều hành. Hệ thống phụ trợ mạng không phải là một thành phần cốt lõi của nhân hệ điều hành (nhân Linux có thể được biên dịch mà không cần hỗ trợ mạng). Tuy nhiên, rất khó có thể tìm được một hệ thống máy tính (hoặc thiết bị nhúng) mà không có hệ điều hành kết nối mạng do nhu cầu liên kết. Hệ điều hành hiện đại sử dụng bộ TCP/IP. Hạt nhân của họ thực hiện các giao thức lên tới tầng vận chuyển, trong khi các giao thức tầng ứng dụng thường được thực hiện trong không gian người dùng (HTTP, FTP, SSH, vv.).

**Kết nối mạng trong không gian người dùng**

* Trong không gian người dùng, trừu tượng hoá của giao tiếp mạng là socket. Socket trừu tượng hóa một kênh truyền thông và là giao diện tương tác của ngăn xếp TCP/IP dựa trên kernel. Một socket IP được liên kết với địa chỉ IP, giao thức tầng vận chuyển được sử dụng (TCP, UDP, vv) và một cổng. Các lệnh chức năng thông thường sử dụng socket bao gồm: tạo (socket), khởi tạo (bind), kết nối (connect), đợi kết nối (listen, accept), đóng một socket (close).
* Giao tiếp mạng được thực hiện thông qua các lệnh đọc / viết hoặc nhận / gửi cho các socket TCP và nhận / gửi đến cho các socket UDP. Các hoạt động truyền và nhận dữ liệu là minh bạch đối với ứng dụng, để phần đóng gói và truyền qua mạng được thực hiện tùy ý trong kernel. Tuy nhiên, có thể triển khai ngăn xếp TCP/IP trong không gian người dùng bằng cách sử dụng các socket raw (tùy chọn PF\_PACKET khi tạo socket), hoặc triển khai một giao thức tầng ứng dụng trong kernel (TUX web server).
* Để biết thêm chi tiết về lập trình không gian người dùng bằng cách sử dụng socket, xem  [Beej's Guide to Network Programming Using Internet Sockets](https://www.beej.us/guide/bgnet/).

**MẠNG LINUX**

Hạt nhân Linux cung cấp ba cấu trúc cơ bản để làm việc với các gói mạng: struct socket, struct sock và struct sk\_buff.

Hai cấu trúc đầu tiên là trừu tượng của một socket:

* struct socket là một trừu tượng rất gần với không gian người dùng, tức là BSD sockets được sử dụng để lập trình các ứng dụng mạng;
* struct sock hoặc INET socket trong thuật ngữ Linux là biểu diễn mạng của một socket.

Hai cấu trúc này liên quan đến nhau: struct socket chứa một trường INET socket, và struct sock có một BSD socket giữ nó.

Cấu trúc struct sk\_buff là biểu diễn của một gói mạng và trạng thái của nó. Cấu trúc được tạo ra khi một gói kernel được nhận, từ không gian người dùng hoặc từ giao diện mạng.

\* **Cấu trúc** ***struct socket***

Cấu trúc struct socket là biểu diễn kernel của một BSD socket, các hoạt động có thể được thực hiện trên nó tương tự như những gì được cung cấp bởi kernel (qua các cuộc gọi hệ thống). Các hoạt động thông thường với socket (tạo, khởi tạo / bind, đóng, vv) dẫn đến các cuộc gọi hệ thống cụ thể; chúng hoạt động với cấu trúc struct socket.

Các hoạt động socket bao gồm:

**Khởi tạo**

Tạo tương tự như việc gọi hàm socket () trong không gian người dùng, nhưng struct socket được tạo sẽ được lưu trữ trong tham số res:

+ int sock\_create(int family, int type, int protocol, struct socket \*\*res): tạo một socket sau cuộc gọi hệ thống socket();

+ int sock\_create\_kern(struct net \*net, int family, int type, int protocol, struct socket \*\*res): tạo một socket kernel;

+ int sock\_create\_lite(int family, int type, int protocol, struct socket \*\*res): tạo một socket kernel mà không có kiểm tra tham số hợp lệ.

Các tham số của các cuộc gọi này như sau:

+ net, nếu có, được sử dụng như tham chiếu đến không gian mạng được sử dụng; chúng ta thường khởi tạo nó với init\_net;

+ family đại diện cho họ các giao thức được sử dụng trong việc truyền thông tin; thường bắt đầu với chuỗi PF\_ (Protocol Family); các hằng số đại diện cho họ của các gia đình giao thức được sử dụng

Để tạo TCP Socket trong kernel, sử dụng:

Text

Description automatically generated

Tạo UDP socket:

Text

Description automatically generated

**Đóng**

Đóng kết nối (đối với các socket sử dụng kết nối) và giải phóng tài nguyên liên quan:

void sock\_release(struct socket \*sock) gọi hàm release trong trường ops của cấu trúc socket:

Text, letter

Description automatically generated

**Gửi/nhận tin nhắn**

Các tin nhắn được gửi/nhận bằng các hàm sau:

+ int sock\_recvmsg(struct socket \*sock, struct msghdr \*msg, int flags);

+ int kernel\_recvmsg(struct socket \*sock, struct msghdr \*msg, struct kvec \*vec, size\_t num, size\_t size, int flags);

+ int sock\_sendmsg(struct socket \*sock, struct msghdr \*msg);

+ int kernel\_sendmsg(struct socket \*sock, struct msghdr \*msg, struct kvec \*vec, size\_t num, size\_t size);

Sau đó, các hàm gửi/nhận tin nhắn sẽ gọi hàm sendmsg/recvmsg trong trường ops của cấu trúc socket. Các hàm có tiền tố kernel\_ được sử dụng khi socket được sử dụng trong kernel.

Các tham số là:

+ msg, một cấu trúc struct msghdr, chứa tin nhắn cần gửi/nhận. Trong số các thành phần quan trọng của cấu trúc này là msg\_name và msg\_namelen, mà đối với các socket UDP, phải được điền vào địa chỉ mà tin nhắn được gửi đến (struct sockaddr\_in);

+ vec, một cấu trúc struct kvec, chứa con trỏ đến bộ đệm chứa dữ liệu của nó và kích thước; như có thể thấy, nó có cấu trúc tương tự với cấu trúc struct iovec (cấu trúc struct iovec tương ứng với dữ liệu không gian người dùng, và cấu trúc struct kvec tương ứng với dữ liệu không gian kernel).

Một ví dụ sử dụng có thể được thấy trong bộ xử lý hệ thống sys\_sendto():

Text

Description automatically generated with medium confidence

Các trường trong Struc socket

Table

Description automatically generated

\* **Cấu trúc** ***struct proto\_ops***

Cấu trúc struct proto\_ops chứa các thực hiện của các hoạt động cụ thể được thực hiện (TCP, UDP, vv); các hàm này sẽ được gọi từ các hàm chung thông qua cấu trúc struct socket (sock\_release (), sock\_sendmsg (), vv.)

Do đó, cấu trúc struct proto\_ops chứa một số con trỏ hàm cho các thực hiện giao thức cụ thể:

Table

Description automatically generated with medium confidence

Việc khởi tạo trường ops từ cấu trúc struct socket được thực hiện trong hàm \_\_sock\_create() bằng cách gọi hàm create() cụ thể cho mỗi giao thức; một cuộc gọi tương đương là việc triển khai của hàm \_\_sock\_create():

Graphical user interface, text

Description automatically generated

Điều đó sẽ khởi tạo các con trỏ hàm với các lời gọi cụ thể cho loại giao thức được liên kết với socket. Các lời gọi sock\_register() và sock\_unregister() được sử dụng để điền vào vectơ net\_families.

Đối với các hoạt động socket khác (ngoại trừ việc tạo, đóng và gửi/nhận một thông điệp như đã mô tả ở phần Operations on the socket structure), các hàm được gửi qua các con trỏ trong cấu trúc này sẽ được gọi. Ví dụ, đối với bind, nó liên kết một socket với một socket trên máy cục bộ, chúng ta sẽ có chuỗi mã sau:

Text, letter

Description automatically generated

Cấu trúc struct sock mô tả một socket INET. Cấu trúc như vậy được liên kết với một socket không gian người dùng và một cách ngầm định với một cấu trúc socket. Cấu trúc này được sử dụng để lưu trữ thông tin về trạng thái kết nối. Các trường và thao tác liên quan thường bắt đầu với chuỗi sk\_. Một số trường được liệt kê dưới đây:

Text

Description automatically generated with medium confidence

\* **Cấu trúc *struct sock***

Cấu trúc struct sock mô tả một socket INET. Cấu trúc như vậy được liên kết với một socket không gian người dùng và một cấu trúc struct socket một cách ngầm định. Cấu trúc được sử dụng để lưu trữ thông tin về trạng thái của một kết nối. Các trường của cấu trúc và các hoạt động liên quan thường bắt đầu bằng chuỗi sk\_. Một số trường được liệt kê bên dưới:

+ sk\_protocol là loại giao thức được sử dụng bởi socket;

+ sk\_type là loại socket (SOCK\_STREAM, SOCK\_DGRAM, v.v.);

+ sk\_socket là socket BSD chứa nó;

+ sk\_send\_head là danh sách các cấu trúc sk\_buff cho việc truyền;

các con trỏ hàm ở cuối là các gọi lại cho các tình huống khác nhau.

Khởi tạo struct sock và đính kèm nó vào một BSD socket được thực hiện bằng cách sử dụng gọi lại được tạo từ net\_families (gọi là \_\_sock\_create()). Dưới đây là cách khởi tạo cấu trúc struct sock cho giao thức IP, trong hàm inet\_create():

Graphical user interface, text

Description automatically generated

**Cấu trúc struct sk\_buff** (socket buffer) mô tả một gói tin mạng. Các trường trong cấu trúc chứa thông tin về cả tiêu đề và nội dung gói tin, các giao thức sử dụng, thiết bị mạng sử dụng và các con trỏ đến các struct sk\_buff khác. Một mô tả tóm tắt về nội dung của cấu trúc được trình bày dưới đây:

Text

Description automatically generated with low confidence

Table

Description automatically generated with medium confidence

Trong đó:

+ next và prev là các con trỏ tới phần tử tiếp theo và phần tử trước đó trong danh sách buffer;

+ dev là thiết bị gửi hoặc nhận buffer;

+ sk là socket được liên kết với buffer;

+ destructor là callback được sử dụng để giải phóng bộ nhớ của buffer;

+ transport\_header, network\_header và mac\_header là offset giữa đầu gói tin và đầu của các header khác nhau trong gói tin. Chúng được duy trì bên trong các lớp xử lý khác nhau mà gói tin đi qua. Để lấy con trỏ tới các header, sử dụng một trong các hàm sau: tcp\_hdr(), udp\_hdr(), ip\_hdr(), vv. Theo nguyên tắc, mỗi protocol cung cấp một hàm để lấy một tham chiếu tới header của protocol đó trong gói tin nhận được.

**Hãy lưu ý rằng** trường network\_header không được thiết lập cho đến khi gói tin đến lớp mạng và trường transport\_header không được thiết lập cho đến khi gói tin đến lớp truyền tải.

Cấu trúc của một header IP (struct iphdr) có các trường sau:

Table

Description automatically generated

Trong đó:

+ protocol là giao thức lớp vận chuyển được sử dụng;

+ saddr là địa chỉ IP nguồn;

+ daddr là địa chỉ IP đích.

Cấu trúc của một header TCP (struct tcphdr) có các trường sau:

Timeline

Description automatically generated

Trong đó:

+ source là cổng nguồn;

+ dest là cổng đích;

+ syn, ack, fin là các cờ TCP được sử dụng; để xem thông tin chi tiết hơn, xem biểu đồ này.

Cấu trúc của header UDP (struct udphdr) bao gồm các trường sau:

Text

Description automatically generated

Trong đó:

+ source là cổng nguồn;

+ dest là cổng đích.

Một ví dụ về việc truy cập thông tin có trong các tiêu đề của gói mạng như sau:

Text, letter

Description automatically generated

**CHUYỂN ĐỔI**

Trong các hệ thống khác nhau, có nhiều cách sắp xếp byte trong một từ (Endianness), bao gồm: Big Endian (byte có trọng số lớn nhất đầu tiên) và Little Endian (byte có trọng số nhỏ nhất đầu tiên). Vì mạng lưới kết nối các hệ thống có nền tảng khác nhau, Internet đã áp đặt một chuỗi tiêu chuẩn cho việc lưu trữ dữ liệu số học, được gọi là network byte-order. Trái ngược với điều đó, chuỗi byte để biểu diễn dữ liệu số học trên máy tính host được gọi là host byte-order. Dữ liệu nhận/gửi từ/về mạng được định dạng theo network byte-order và cần được chuyển đổi giữa định dạng này và host byte-order.

Để chuyển đổi, chúng ta sử dụng các macro sau:

* u16 htons(u16 x) chuyển đổi một số nguyên 16 bit từ host byte-order sang network byte-order (host thành mạng ngắn);
* u32 htonl(u32 x) chuyển đổi một số nguyên 32 bit từ host byte-order sang network byte-order (host thành mạng dài);
* u16 ntohs(u16 x) chuyển đổi một số nguyên 16 bit từ network byte-order sang host byte-order (mạng thành host ngắn);
* u32 ntohl(u32 x) chuyển đổi một số nguyên 32 bit từ network byte-order sang host byte-order (mạng thành host dài).

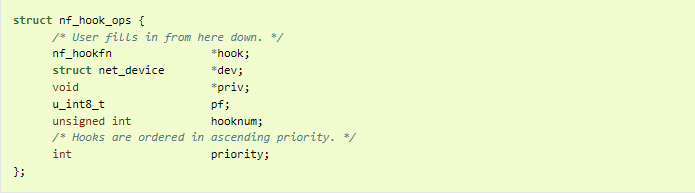
**Netfilter**

Netfilter là tên gọi của giao diện kernel để bắt các gói tin mạng để sửa đổi/ phân tích chúng (để lọc, NAT, vv.). Giao diện netfilter được sử dụng trong user space bởi iptables.

Trong kernel Linux, bắt gói tin bằng netfilter được thực hiện bằng cách gắn các hooks (kẹp). Các hooks có thể được chỉ định ở các vị trí khác nhau trong quá trình theo đuổi của gói tin mạng trong kernel, tùy theo yêu cầu. Bảng tổ chức với đường đi của gói và các khu vực có thể cho một hook có thể được tìm thấy tại đây.

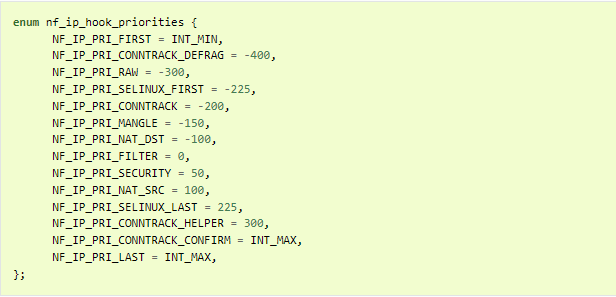
Header được bao gồm khi sử dụng netfilter là linux/netfilter.h.

Một hook được định nghĩa thông qua cấu trúc struct nf\_hook\_ops:

****

ở đó:

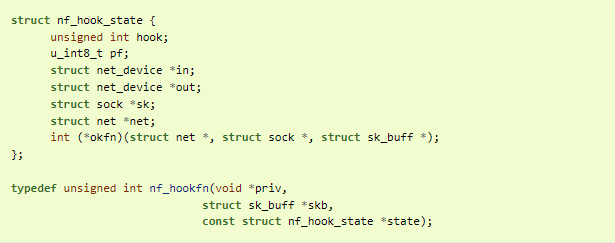
* pf là loại gói tin (PF\_INET, vv.);
* priority là độ ưu tiên; các độ ưu tiên được xác định trong uapi/linux/netfilter\_ipv4.h như sau:



* dev là thiết bị (giao diện mạng) mà bắt đầu việc bắt gói tin;
* hooknum là loại hook được sử dụng. Khi một gói tin được bắt, chế độ xử lý được xác định bởi các trường hooknum và hook. Đối với IP, các loại hook được xác định trong linux/netfilter.h:

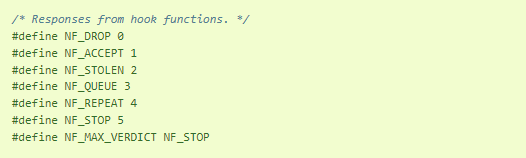


* hook là bộ xử lý được gọi khi bắt một gói tin mạng (gói tin được gửi dưới dạng cấu trúc struct sk\_buff). Trường private là thông tin riêng tư được chuyển đến bộ xử lý. Nguyên mẫu của bộ xử lý bắt gói tin được xác định bởi kiểu nf\_hookfn:



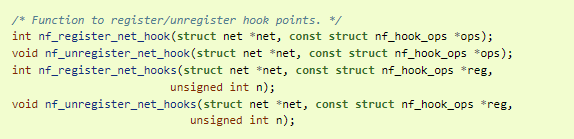
Đối với hàm bắt gói tin nf\_hookfn(), tham số priv là thông tin riêng tư được khởi tạo với struct nf\_hook\_ops. Biến skb là con trỏ đến gói tin mạng đã bắt được. Dựa trên thông tin của skb, quyết định lọc gói tin được đưa ra. Tham số trạng thái của hàm là thông tin trạng thái liên quan đến việc bắt gói tin, bao gồm giao diện đầu vào, giao diện đầu ra, độ ưu tiên, số hook. Độ ưu tiên và số hook hữu ích để cho phép cùng một hàm được gọi bởi nhiều hook.

Một trình xử lý bắt gói tin có thể trả về một trong các hằng số NF\_\*:



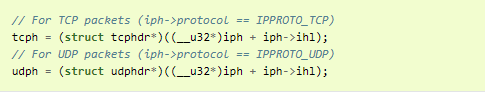
NF\_DROP được sử dụng để lọc (bỏ qua) một gói tin, và NF\_ACCEPT được sử dụng để chấp nhận một gói tin và chuyển tiếp nó.

Đăng ký / hủy đăng ký một hook được thực hiện bằng cách sử dụng các hàm được định nghĩa trong linux/netfilter.h:



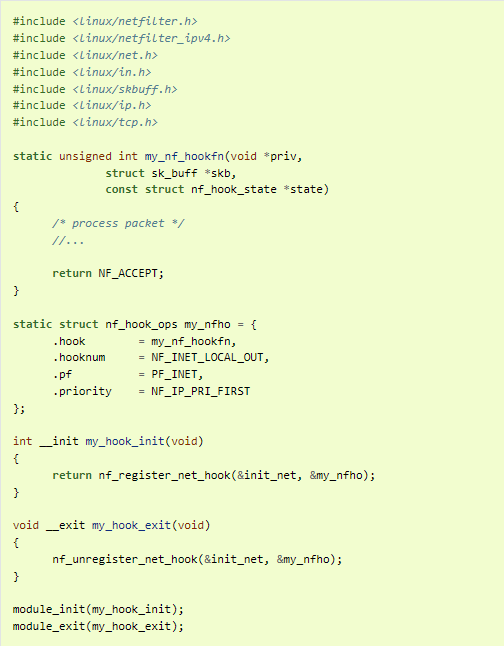
**Chú ý**

Có một số hạn chế trong việc sử dụng các hàm trích xuất header từ một cấu trúc struct sk\_buff được đặt làm tham số trong một netfilter hook . Trong khi header IP có thể được lấy ra mỗi khi sử dụng hàm ip\_hdr(), header TCP và UDP chỉ có thể được lấy ra bằng cách sử dụng tcp\_hdr() và udp\_hdr() đối với các gói tin được gửi từ bên trong hệ thống chứ không phải các gói tin được nhận từ bên ngoài hệ thống. Trong trường hợp sau, bạn phải tính toán thủ công offset của header trong gói tin:



This code works in all filtering situations, so it's recommended to use it instead of header access functions.

Một ví dụ sử dụng cho một netfilter hook được hiển thị bên dưới:



**Netcat**

Khi phát triển ứng dụng bao gồm mã lập trình mạng, một trong những công cụ được sử dụng nhiều nhất là netcat. Còn được gọi là "Swiss-army knife for TCP/IP". Nó cho phép:

* Khởi tạo kết nối TCP;
* Chờ đợi kết nối TCP;
* Gửi và nhận các gói UDP;
* Hiển thị traffic dưới dạng hexdump;
* Chạy một chương trình sau khi thiết lập kết nối (ví dụ, một shell);
* Đặt các tùy chọn đặc biệt trong các gói đã gửi.

Khởi tạo kết nối TCP:



Nghe một cổng TCP:



Gửi và nhận các gói UDP được thực hiện bằng cách thêm tùy chọn dòng lệnh -u.

**Chú ý:**

Lệnh sử dụng là nc; thường thì netcat là một cái tên định danh cho lệnh này. Có các phiên bản khác nhau của lệnh netcat, một số trong số đó có các tham số khác nhau so với phiên bản cổ điển. Chạy lệnh man nc hoặc nc -h để kiểm tra cách sử dụng lệnh này.

Để biết thêm thông tin về netcat, xem [tutorial](https://www.win.tue.nl/~aeb/linux/hh/netcat_tutorial.pdf) sau đây.

**ĐỌC THÊM**

1. Understanding Linux Network Internals
2. [Linux IP networking](http://www.cs.unh.edu/cnrg/gherrin/)
3. [The TUX Web Server](http://www.stllinux.org/meeting_notes/2001/0719/myTUX/)
4. [Beej's Guide to Network Programming Using Internet Sockets](https://www.beej.us/guide/bgnet/)
5. [Kernel Korner - Network Programming in the Kernel](http://www.linuxjournal.com/article/7660)
6. [Hacking the Linux Kernel Network Stack](http://phrack.org/issues/61/13.html)
7. [The netfilter.org project](http://www.netfilter.org/)
8. [A Deep Dive Into Iptables and Netfilter Architecture](https://www.digitalocean.com/community/tutorials/a-deep-dive-into-iptables-and-netfilter-architecture)
9. [Linux Foundation Networking Page](http://www.linuxfoundation.org/en/Net:Main_Page)

**BÀI TẬP:**

**Quan trọng**

Để giải quyết các bài tập, bạn cần thực hiện các bước sau:

1. Chuẩn bị khung từ các bản mẫu
2. Xây dựng các module
3. Sao chép các module vào VM
4. Khởi động VM và kiểm tra module trong VM
5. Tên lab hiện tại là networking. Xem các bài tập để biết tên nhiệm vụ.

Mã khung được tạo ra từ các ví dụ nguồn đầy đủ nằm trong thư mục tools/labs/templates. Để giải quyết các nhiệm vụ, bắt đầu bằng cách tạo mã khung cho một lab hoàn chỉnh:

tools/labs $ make clean

tools/labs $ LABS=<lab name> make skels

Bạn cũng có thể tạo khung cho một tác vụ đơn lẻ, sử dụng:

tools/labs $ LABS=<lab name>/<task name> make skels

Khi trình điều khiển bộ xương được tạo, hãy tạo nguồn:

tools/labs $ make build

Sau đó, sao chép các mô-đun và khởi động VM:

tools/labs $ make copy

tools/labs $ make boot

Các mô-đun được đặt trong /home/root/skels/networking/<task\_name>.

Ngoài ra, chúng tôi có thể sao chép tệp qua scp để tránh khởi động lại VM. Để biết thêm chi tiết về cách kết nối với VM qua mạng, vui lòng kiểm tra Kết nối với Máy ảo.

Xem lại phần Bài tập để biết thêm thông tin chi tiết.

**Cảnh báo**

Trước khi bắt đầu làm các bài tập hoặc tạo các mã khung, hãy chạy lệnh git pull trong thư mục Linux repo để đảm bảo bạn có phiên bản mới nhất của các bài tập.

Nếu bạn có thay đổi địa phương, lệnh pull sẽ thất bại. Kiểm tra các thay đổi địa phương bằng cách sử dụng git status. Nếu bạn muốn giữ chúng, hãy chạy git stash trước khi pull và git stash pop sau đó. Để loại bỏ các thay đổi, hãy chạy git reset --hard master.

Nếu bạn đã tạo mã khung trước khi chạy git pull, bạn sẽ cần tạo lại mã khung đó.

**Quan trọng**

Bạn cần đảm bảo rằng hỗ trợ netfilter được kích hoạt trong kernel. Nó được kích hoạt thông qua CONFIG\_NETFILTER. Để kích hoạt nó, hãy chạy lệnh make menuconfig trong thư mục linux và kiểm tra tùy chọn Network packet filtering framework (Netfilter) trong Networking support -> Networking options. Nếu nó chưa được kích hoạt, hãy kích hoạt nó (dưới dạng builtin, không phải external module - nó phải được đánh dấu bằng \*).

1. **Hiển thị các gói tin trong không gian kernel**

Viết một module kernel hiển thị địa chỉ và cổng nguồn cho các gói tin TCP khởi tạo một kết nối ra ngoài. Bắt đầu từ mã trong 1-2-netfilter và điền vào các vùng được đánh dấu bằng TODO 1, lấy ý kiến ​​bên dưới vào cân nhắc.

Bạn sẽ cần đăng ký một netfilter hookcủa loại NF\_INET\_LOCAL\_OUT như được giải thích trong phần netfilter.

Cấu trúc struct sk\_buff cho phép bạn truy cập các tiêu đề gói tin bằng cách sử dụng các hàm cụ thể. Hàm ip\_hdr () trả về tiêu đề IP dưới dạng con trỏ đến cấu trúc struct iphdr. Hàm tcp\_hdr () trả về tiêu đề TCP dưới dạng con trỏ đến cấu trúc struct tcphdr.

Sơ đồ giải thích cách tạo kết nối TCP. Gói khởi tạo kết nối có cờ SYN được đặt trong tiêu đề TCP và cờ ACK bị xóa.

**Lưu ý**

Để hiển thị địa chỉ IP nguồn, sử dụng định dạng %pI4 của hàm printk. Chi tiết có thể được tìm thấy trong tài liệu kernel (phần địa chỉ IPv4). Đoạn mã ví dụ sau sử dụng %pI4:

printk("IP address is %pI4**\n**", &iph->saddr);

Khi sử dụng định dạng %pI4, đối số cho printk là một con trỏ. Do đó, xây dựng &iph->saddr (với toán tử & - ampersand) thay vì iph->saddr.

Cổng TCP nguồn, trong header của giao thức TCP, được biểu diễn dưới định dạng network byte-order. Để chuyển đổi giá trị này sang định dạng host byte-order, ta có thể sử dụng hàm ntohs().

Để thử nghiệm, hãy sử dụng tệp 1-2-netfilter/user/test-1.sh. Thử nghiệm tạo một kết nối đến máy chủ cục bộ, một kết nối sẽ bị chặn và hiển thị bởi mô-đun hạt nhân. Tập lệnh chỉ được sao chép trên máy ảo bằng lệnh make copy nếu nó được đánh dấu là có thể thực thi được. Tập lệnh sử dụng công cụ netcat được biên dịch tĩnh được lưu trữ trong skels/networking/netcat; chương trình này phải có quyền thực thi.

Sau khi chạy trình kiểm tra, đầu ra sẽ tương tự như dưới đây:

# ./test-1.sh

[ 229.783512] TCP connection initiated from 127.0.0.1:44716

Should show up in filter.

Check dmesg output.

1. **Lọc theo địa chỉ đích**

Mở rộng module từ bài tập 1 để bạn có thể chỉ định một địa chỉ đích thông qua một cuộc gọi ioctl MY\_IOCTL\_FILTER\_ADDRESS. Bạn chỉ sẽ hiển thị các gói chứa địa chỉ đích đã chỉ định. Để giải quyết nhiệm vụ này, hãy điền vào các vùng được đánh dấu TODO 2 và tuân theo các thông số kỹ thuật bên dưới.

Để thực hiện chức năng ioctl, bạn phải điền thông tin vào hàm my\_ioctl. Xem lại phần ioctl. Địa chỉ được gửi từ không gian người dùng ở dạng byte-order của mạng, do đó sẽ KHÔNG cần chuyển đổi.

**Lưu ý**

Địa chỉ IP được gửi qua ioctl được gửi theo địa chỉ, không phải theo giá trị. Địa chỉ phải được lưu trữ trong biến ioctl\_set\_addr. Để sao chép, sử dụng copy\_from\_user().

Để so sánh các địa chỉ, điền thông tin vào hàm test\_daddr. Các địa chỉ trong byte-order của mạng sẽ được sử dụng mà không cần chuyển đổi địa chỉ (nếu chúng bằng nhau từ trái sang phải thì chúng sẽ bằng nhau nếu đảo ngược lại).

Hàm test\_daddr phải được gọi từ netfilter hook để hiển thị các gói khởi tạo kết nối cho địa chỉ đích là địa chỉ được gửi thông qua chức năng ioctl. Gói khởi tạo kết nối có cờ SYN được đặt trong tiêu đề TCP và cờ ACK được xóa bỏ. Bạn phải kiểm tra hai điều sau đây:

1. Cờ TCP;
2. Địa chỉ đích của gói (sử dụng test\_addr).

Để kiểm tra, hãy sử dụng tập lệnh 1-2-netfilter/user/test-2.sh. Tập lệnh này cần biên dịch tệp test.c trong thư mục thực thi test từ tệp 1-2-netfilter/user/. Việc biên dịch được thực hiện tự động trên hệ thống vật lý khi chạy lệnh make build. Tập lệnh kiểm thử được sao chép đến máy ảo chỉ khi nó được đánh dấu là có quyền thực thi. Tập lệnh sử dụng công cụ netcat được biên dịch tĩnh trong skels/networking/netcat; tệp thực thi này phải có quyền thực thi.

Sau khi chạy trình kiểm tra, đầu ra sẽ tương tự như sau:

./test-2.sh[ 797.673535] TCP connection initiated from 127.0.0.1:44721Should show up in filter.Should NOT show up in filter.Check dmesg output.

Kiểm tra yêu cầu lọc gói trước tiên cho địa chỉ IP 127.0.0.1 và sau đó là địa chỉ IP 127.0.0.2. Gói khởi tạo kết nối đầu tiên (đến 127.0.0.1) bị chặn và hiển thị bởi bộ lọc, trong khi gói thứ hai (đến 127.0.0.2) không bị chặn.

**3. Lắng nghe trên một socket TCP**

Viết một module kernel tạo ra một socket TCP nghe kết nối trên cổng 60000 trên giao diện loopback (trong init\_module). Bắt đầu từ mã trong 3-4-tcp-sock điền vào các vùng được đánh dấu TODO 1, lưu ý những quan sát bên dưới.

Đọc các phần Operations on the socket structure và The struct proto\_ops structure.

Sock socket là một socket máy chủ và phải được đưa vào trạng thái lắng nghe. Tức là, các hoạt động bind và listen phải được áp dụng cho socket. Để tương đương với bind và listen, trong không gian kernel, bạn cần gọi sock->ops->...; các ví dụ về các hàm như vậy mà bạn có thể gọi là sock->ops->bind, sock->ops->listen vv.

**Chú ý**

Ví dụ, gọi sock->ops->bind hoặc sock->ops->listen, xem cách chúng được gọi trong các trình xử lý cuộc gọi hệ thống sys\_bind () và sys\_listen ().

Tìm các trình xử lý cuộc gọi hệ thống trong tệp net/socket.c trong cây mã nguồn kernel Linux.

**Chú ý**

Đối với đối số thứ hai của cuộc gọi listen (backlog), sử dụng LISTEN\_BACKLOG.

Hãy nhớ giải phóng socket trong hàm thoát của module và trong khu vực được đánh dấu với nhãn lỗi; sử dụng sock\_release ().

Để kiểm tra, chạy script 3-4-tcp\_sock/test-3.sh. Kịch bản được sao chép trên máy ảo bằng make copy chỉ khi nó được đánh dấu là có thể thực thi.

Sau khi chạy kiểm tra, một socket TCP sẽ được hiển thị bằng cách lắng nghe các kết nối trên cổng 60000.

**4.Chấp nhận các kết nối trong không gian kernel**

Mở rộng module từ bài tập trước để cho phép chấp nhận kết nối bên ngoài (không cần gửi bất kỳ thông điệp nào, chỉ chấp nhận các kết nối mới). Điền vào các vùng được đánh dấu TODO 2.

Đọc các phần Operations on the socket structure và The struct proto\_ops structure.

Đối với tương đương chấp nhận của kernel space, xem trình xử lý hệ thống cho sys\_accept4(). Theo cài đặt lnet\_sock\_accept, và cách gọi sock->ops->accept. Sử dụng 0 là giá trị cho đối số thứ hai từ cuối cùng (flags), và true cho đối số cuối cùng (kern).

**Lưu ý**

Tìm các trình xử lý hệ thống trong tệp net/socket.c trong cây mã nguồn kernel Linux.

**Lưu ý**

Số socket mới (new\_sock) phải được tạo ra với hàm sock\_create\_lite() sau đó hoạt động của nó phải được cấu hình bằng

newsock->ops = sock->ops;

In địa chỉ và cổng của socket đích. Để tìm tên ngang hàng của một socket (địa chỉ của nó), tham khảo trình xử lý hệ thống sys\_getpeername().

**Lưu ý**

Đối số đầu tiên cho hàm sock->ops->getname sẽ là socket kết nối, tức là new\_sock, socket được khởi tạo bằng cuộc gọi chấp nhận.

Đối số cuối cùng của hàm sock->ops->getname sẽ là 1, có nghĩa là chúng ta muốn biết về điểm cuối hoặc ngang hàng (đầu cuối xa hoặc ngang hàng).

Hiển thị địa chỉ ngang hàng (được chỉ định bởi biến raddr) bằng cách sử dụng macro print\_sock\_address được định nghĩa trong tệp.

Giải phóng socket mới được tạo ra (sau khi chấp nhận kết nối) trong hàm exit của module và sau nhãn lỗi. Sau khi thêm mã chấp nhận vào hàm khởi tạo module, thao tác insmod sẽ bị khóa cho đến khi một kết nối được thiết lập. Bạn có thể mở khóa bằng cách sử dụng netcat trên cổng đó. Do đó, tập lệnh kiểm tra từ bài tập trước sẽ không hoạt động.

Để kiểm tra, chạy tập lệnh 3-4-tcp\_sock/test-4.sh. Tập lệnh được sao chép vào máy ảo bởi make copy chỉ khi nó được

Không có gì đặc biệt sẽ được hiển thị (trong bộ đệm kernel). Sự thành công của thử nghiệm sẽ được xác định bởi thiết lập kết nối. Sau đó, sử dụng Ctrl+c để dừng tập lệnh kiểm tra, sau đó bạn có thể xóa mô-đun kernel

**5. Gửi socket UDP**

Viết một kernel module tạo một socket UDP và gửi tin nhắn từ macro MY\_TEST\_MESSAGE trên socket đến địa chỉ loopback trên cổng 60001.

Bắt đầu từ mã trong 5-udp-sock.

Đọc phần Operations on the socket structure và The struct proto\_ops structure.

Để xem cách gửi tin nhắn trong kernel space, hãy xem sys\_send() system call handler hoặc Sending/receiving messages.

**Mẹo**

Trường msg\_name của struct msghdr structure phải được khởi tạo với địa chỉ đích (con trỏ tới struct sockaddr) và trường msg\_namelen là kích thước địa chỉ.

Khởi tạo trường msg\_flags của struct msghdr structure thành 0.

Khởi tạo trường msg\_control và msg\_controllen của struct msghdr structure thành NULL và 0 tương ứng.

Để gửi tin nhắn, sử dụng kernel\_sendmsg().

Tham số truyền tin được lấy từ kernel space. Ép kiểu con trỏ struct iovec structure thành con trỏ struct kvec trong cuộc gọi kernel\_sendmsg().

**Mẹo**

Hai tham số cuối của kernel\_sendmsg() là 1 (số I/O vectors) và len (kích thước tin nhắn).

Để kiểm tra, sử dụng tệp test-5.sh. Tệp này được sao chép vào máy ảo bằng lệnh make copy chỉ khi nó được đánh dấu là có thể thực thi. Tập lệnh này sử dụng công cụ netcat được biên dịch tĩnh được lưu trữ trong skels/networking/netcat; tệp này phải có quyền thực thi.

Để hiện thực đúng, chạy tập lệnh test-5.sh sẽ khiến tin nhắn kernelsocket được hiển thị như đầu ra dưới đây:

/root # ./test-5.sh

pid=1059

sleep 1

nc -l -u -p 60001

insmod udp\_sock.ko

kernelsocket

rmmod udp\_sock

kill 1059.

\* \* SO2 - Networking Lab (#10) \* \* Exercise #1, #2: simple netfilter module \* \* Code skeleton. \*/ #include <linux/kernel.h> #include <linux/sched.h> #include <linux/init.h> #include <linux/module.h> #include <asm/uaccess.h> #include <linux/fs.h> #include <linux/cdev.h> #include <asm/atomic.h> #include <linux/netfilter.h> #include <linux/netfilter\_ipv4.h> #include <linux/net.h> #include <linux/in.h> #include <linux/skbuff.h> #include <linux/ip.h> #include <linux/tcp.h> #include "filter.h" MODULE\_DESCRIPTION("Simple netfilter module"); MODULE\_AUTHOR("SO2"); MODULE\_LICENSE("GPL"); #define LOG\_LEVEL KERN\_ALERT #define MY\_DEVICE "filter" static struct cdev my\_cdev; static atomic\_t ioctl\_set; static unsigned int ioctl\_set\_addr; /\* Test ioctl\_set\_addr if it has been set. \*/ static int test\_daddr(unsigned int dst\_addr) { int ret = 0; /\* TODO 2/4: return non-zero if address has been set \* \*and\* matches dst\_addr \*/ if (atomic\_read(&ioctl\_set) == 1) ret = (ioctl\_set\_addr == dst\_addr); else ret = 1; return ret; } /\* TODO 1/20: netfilter hook function \*/ static unsigned int my\_nf\_hookfn(void \*priv, struct sk\_buff \*skb, const struct nf\_hook\_state \*state) { /\* get IP header \*/ struct iphdr \*iph = ip\_hdr(skb); if (iph->protocol == IPPROTO\_TCP && test\_daddr(iph->daddr)) { /\* get TCP header \*/ struct tcphdr \*tcph = tcp\_hdr(skb); /\* test for connection initiating packet \*/ if (tcph->syn && !tcph->ack) printk(LOG\_LEVEL "TCP connection initiated from " "%pI4:%u\n", &iph->saddr, ntohs(tcph->source)); } /\* let the package pass \*/ return NF\_ACCEPT; } static int my\_open(struct inode \*inode, struct file \*file) { return 0; } static int my\_close(struct inode \*inode, struct file \*file) { return 0; } static long my\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg) { switch (cmd) { case MY\_IOCTL\_FILTER\_ADDRESS: /\* TODO 2/4: set filter address from arg \*/ if (copy\_from\_user(&ioctl\_set\_addr, (void \*) arg, sizeof(ioctl\_set\_addr))) return -EFAULT; atomic\_set(&ioctl\_set, 1); break; default: return -ENOTTY; } return 0; } static const struct file\_operations my\_fops = { .owner = THIS\_MODULE, .open = my\_open, .release = my\_close, .unlocked\_ioctl = my\_ioctl }; /\* TODO 1/6: define netfilter hook operations structure \*/ static struct nf\_hook\_ops my\_nfho = { .hook = my\_nf\_hookfn, .hooknum = NF\_INET\_LOCAL\_OUT, .pf = PF\_INET, .priority = NF\_IP\_PRI\_FIRST }; int \_\_init my\_hook\_init(void) { int err; /\* register filter device \*/ err = register\_chrdev\_region(MKDEV(MY\_MAJOR, 0), 1, MY\_DEVICE); if (err != 0) return err; atomic\_set(&ioctl\_set, 0); ioctl\_set\_addr = 0; /\* init & add device \*/ cdev\_init(&my\_cdev, &my\_fops); cdev\_add(&my\_cdev, MKDEV(MY\_MAJOR, 0), 1); /\* TODO 1/3: register netfilter hook \*/ err = nf\_register\_net\_hook(&init\_net, &my\_nfho); if (err) goto out; return 0; out: /\* cleanup \*/ cdev\_del(&my\_cdev); unregister\_chrdev\_region(MKDEV(MY\_MAJOR, 0), 1); return err; } void \_\_exit my\_hook\_exit(void) { /\* TODO 1/1: unregister hook \*/ nf\_unregister\_net\_hook(&init\_net, &my\_nfho); /\* cleanup device \*/ cdev\_del(&my\_cdev); unregister\_chrdev\_region(MKDEV(MY\_MAJOR, 0), 1); } module\_init(my\_hook\_init); module\_exit(my\_hook\_exit); hãy giải thích code trên

Đây là một module netfilter đơn giản trong Linux kernel. Module này sử dụng hook để theo dõi các gói tin mạng đi qua hệ thống, nếu địa chỉ đích của gói tin được đặt bởi ứng dụng sử dụng ioctl thì nó sẽ ghi log lại thông tin về kết nối TCP được khởi tạo từ địa chỉ đó.

Một số đoạn mã quan trọng trong module này:

1. Hook function

rustCopy code

static unsigned int my\_nf\_hookfn(void \*priv, struct sk\_buff \*skb, const struct nf\_hook\_state \*state) { /\* get IP header \*/ struct iphdr \*iph = ip\_hdr(skb); if (iph->protocol == IPPROTO\_TCP && test\_daddr(iph->daddr)) { /\* get TCP header \*/ struct tcphdr \*tcph = tcp\_hdr(skb); /\* test for connection initiating packet \*/ if (tcph->syn && !tcph->ack) printk(LOG\_LEVEL "TCP connection initiated from %pI4:%u\n", &iph->saddr, ntohs(tcph->source)); } /\* let the package pass \*/ return NF\_ACCEPT; }

Đây là hàm hook chính, được gọi mỗi khi một gói tin mạng đi qua hệ thống. Hàm này kiểm tra nếu địa chỉ đích của gói tin tương ứng với địa chỉ được đặt bởi ứng dụng sử dụng ioctl thì nó sẽ ghi log lại thông tin về kết nối TCP được khởi tạo từ địa chỉ đó.

1. Hàm ioctl

cCopy code

static long my\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg) { switch (cmd) { case MY\_IOCTL\_FILTER\_ADDRESS: /\* TODO 2/4: set filter address from arg \*/ if (copy\_from\_user(&ioctl\_set\_addr, (void \*) arg, sizeof(ioctl\_set\_addr))) return -EFAULT; atomic\_set(&ioctl\_set, 1); break; default: return -ENOTTY; } return 0; }

Hàm ioctl được sử dụng để đặt địa chỉ đích mà hook sẽ theo dõi. Nó lấy địa chỉ được truyền vào từ ứng dụng sử dụng ioctl và lưu trữ nó trong biến ioctl\_set\_addr. Sau đó, nó đặt giá trị của biến ioctl\_set là 1 để cho phép hàm test\_daddr có thể kiểm tra xem địa chỉ được đặt bởi ứng dụng sử dụng ioctl có tương ứng với địa chỉ đích của gói tin không.

1. Đăng ký hook

csharpCopy code

static struct nf\_hook\_ops my\_nfho = { .hook = my\_nf\_hookfn, .hooknum = NF\_INET\_LOCAL\_OUT, .pf = PF\_INET, .priority = NF\_IP\_PRI\_FIRST }; int \_\_init my\_hook\_init(void) { int err; /\* TODO 1/3: register netfilter hook \*/ err = nf\_register\_net\_hook(&init

tiếp tục

Tiếp tục phân tích code:

* Hàm **test\_daddr** được sử dụng để kiểm tra xem địa chỉ đích của gói tin có phù hợp với địa chỉ đã được set bởi ioctl hay không. Hàm này trả về 1 nếu địa chỉ chưa được set hoặc địa chỉ đã được set và trùng với địa chỉ đích của gói tin.
* Hàm **my\_nf\_hookfn** được đăng ký như một hook function để được gọi mỗi khi có gói tin được gửi đi. Hàm này kiểm tra xem gói tin có phải là gói tin TCP kết nối mới hay không bằng cách kiểm tra trường **syn** và **ack** của header TCP. Nếu đúng, nó sẽ in ra thông tin về địa chỉ nguồn và cổng nguồn của gói tin.
* Hàm **my\_open** và **my\_close** được sử dụng để đăng ký thiết bị trong hệ thống.
* Hàm **my\_ioctl** được sử dụng để xử lý các ioctl được gọi trên thiết bị. Nếu ioctl là **MY\_IOCTL\_FILTER\_ADDRESS**, địa chỉ sẽ được set từ giá trị được truyền vào thông qua đối số **arg**.
* Cấu trúc **my\_fops** chứa các định nghĩa cho các hàm mở, đóng và ioctl.
* Cấu trúc **my\_nfho** chứa thông tin về hook function, số hook, protocol và priority.
* Hàm **my\_hook\_init** được gọi khi module được tải vào kernel và thực hiện việc đăng ký thiết bị, đăng ký hook function và cấu hình atomic variable **ioctl\_set** và **ioctl\_set\_addr**.
* Hàm **my\_hook\_exit** được gọi khi module bị gỡ bỏ khỏi kernel và thực hiện việc hủy đăng ký hook function và xóa thiết bị đã được đăng ký trước đó.