Linux netfilter Hacking HOWTO

Rusty Russell

netfilter (at) lists.samba.org

도운이

金子研史

kenji (at) rtlinux.pe.kr

본 문서에서는, 먼저 리눅스에 대한 넷필터 아키텍처와 이를 파헤치는 방법, 그리고 넷필터의 최상위에 존재하는 일부 중요한 시스템, 즉 패킷 필터링, 커넥션 트랙킹 그리고 NAT(Network Address Translation)등에 대하여 기술하고자 한다.

고친 과정

고침 0.22001-02-1 고친이 kenji
문장을 보다 매끄럽게 수정하였습니다.
고침 0.12001-01-20고친이 kenji
최초 번역

1. 서문

본 문서는 여행과도 같으며, 일부분은 아주 쉽게 여행할 수 있고 또 다른 부분에서는 독자 여러분 스스로 길을 찾아야 할 것이다. 필자가 독자에게 할 수 있는 최상의 충고는 아주 큰 머그잔에 커피나 핫쵸콜릿을 가득 담아 편안한 의자에 앉아 위험스런 길을 가기 전에 본문의 내용을 넷트웍 해킹이라는 아주 위험스러운 세상에 부합시켜 깊이 생각해 보라는 것 밖에 없다.

넷필터 프레임웍의 최상위에 있는 내부구조의 사용법을 보다 잘 이해하기 위해서는, Packet Filtering HOWTO와 NAT HOWTO를 읽어보는 것이 좋을 것이다. 커널 프로그래밍에 대한 정보를 얻고자 한다면 Rusty's Unreliable Guide to Kernel Hacking과 Rusty's Unreliable Guide to Kernel Locking을 참고하기 바란다.

(C) 2000 Paul 'Rusty' Russell. Licensed under the GNU GPL.

1.1. 넷필터(netfilter)란 무엇인가?

넷필터는 표준 Berkeley socket interface의 외부에 존재하는 packet mangling(패킷을 토막내는 일)에 대한 프레임웍으로, 크게 네 부분으로 구성되어 있다. 먼저 각각의 프로토콜은 "hooks"라는 것을 정의하며, 이는 패킷 프로토콜 스택의 packet's traversal에 있는 잘 정의된 포인터를 의미한다. 이러한 포인터에서, 각각의 프로토콜은 패킷과 훅넘버(hook number)를 이용하여 넷필터 프레임웍을 호출하게 된다.

두 번째로, 커널의 일부분은 각 프로토콜에 대하여 다른 hook을 감시하도록 등록할 수 있다. 따라서 패킷이 넷필터 프레임웍을 통과할 때, 누가 그 프로토콜과 훅을 등록했는지 확인하게 된다. 이러한 것이 등록되어 있다면, 등록된 순서대로 패킷을 검사하고, 패킷을 무시하거나(NF_DROP), 통과시키고(NF_ACCEPT), 또는 패킷에 대한 것을 잊어버리도록 넷필터에게 지시하거나(NF_STOLEN), 사용자 공간에 패킷을 대기시키도록(queuing) 넷필터에게 요청한다(NF_QUEUE).

세 번째 부분은 대기된 패킷을 사용자 공간으로 보내기 위해 제어하는 것으로 이러한 패킷은 비동기방식으로 처리된다.

마지막 부분은 코드와 문서에 기록된 주석문으로 구성되어 있으며, 이는 어떠한 실험적 프로젝트에 대해서도 도움이 되는 부분이다. 넷 필터의 모토는 다음과 같다.

"그래서... KDE보다 얼만큼 좋다는 거죠?"

이러한 저수준 프레임웍과 더불어, 다양한 모듈이 작성되었으며, 이는 이전 버전의 커널에 대하여 유사한 기능, 확장 가능한 NAT시스템 그리고 확장 가능한 패킷 필터링 시스템을 제공한다.

1.1.1. 커널 2.0과 2.2에서의 문제점?

1. 사용자 공간을 통과하는 패킷에 대하여 어떠한 하부구조도 만들어져 있지 않으며 그 이유는 다음과 같다.
   - 커널수준에서 코딩하는 것이 어렵다
   - 반드시 C/C++로 커널 수준의 코딩이 되어야 한다.
   - 동적 필터링 정책이 커널에 포함되어 있지 않다.
   - 커널 2.2에서는 넷링크를 거쳐 사용자 공간으로 패킷을 복사하는 방법을 제시하였으나, 재전송 패킷이 느리고 'sanity' check에 지배된다. 예를 들면, 재전송 패킷이 기존의 인터페이스로 들어오도록 요청하는 것이 불가능하다.
2. 투명한 프락시 구현이 어렵다.
   - 그 주소와 연관된 소켓이 존재하는 지 알아보기 위해 모든 패킷을 살펴본다.
   - Root는 외부 주소와 연결되는 것을 허가해야 한다.
   - 로컬에서 생성된 패킷을 리다이렉트할 수 없다.

- REDIRECT는 UDP에 대한 응답을 처리하지 못 한다. 예를 들면, UDP 패킷을 1153으로의 리다이렉트가 동작을 하지 않는 경우도 있으며, 이는 일부 클라이언트가 53번 포트가 아닌 다른 곳으로부터 들어오는 응답을 싫어하기 때문이다.
- REDIRECT는 tcp/upd 포트 재배치에 대하여 관여하지 않는다. 즉, 사용자가 REDIRECT 룰에 의해 결정된 포트를 얻어와야 한다.
- 커널 2.1 시리즈 개발기간 중 최소 두 번 정도 중단되었다.
- 코드가 상당히 지저분하다. 커널 2.2.1에서 #ifdef CONFIG_IP_TRANSPARENT_PROXY의 사용 통계를 보면, 11개의 파일에 34번 나타난다. CONFIG_IP_FIREWALL과 비교해 보면, 5개의 파일에 10번만 나타난다.

3. 인터페이스 어드레스와 별개로 패킷필터 룰을 만드는 것이 불가능하다.
   - 인터페이스를 거쳐오는 패킷이 로컬에서 생성된 것인지 로컬을 들어오는 것인지 구분하기 위해서 반드시 로컬 인터페이스의 주소를 알아야 한다.
   - 리다이렉션이나 매스커레이딩의 경우는 정보가 충분하지 않다..
   - 포워딩 체인만이 외부로 향하는 인터페이스에 대한 정보를 가지고 있다. 네트웍 구성에 대한 지식을 이용하여 어느 인터페이스로부터 패킷이 들어오는지 알고 있어야만 한다.

4. 매스커레이딩이 필터링에 포함되어 있다:

   필터링과 매스커레이딩간의 상호작용이 방화벽 구축을 복잡하게 만든다:
   - 입력 필터링인 경우, 응답 패킷이 박스 자체를 향하는 것으로 나타난다.
   - 디매스커레이드(demasqueraded)되는 패킷이 포워드 필터링에서는 전혀 나타나지 않는다.
   - 출력 필터링에서는 패킷이 로컬 박스로부터 들어오는 것처럼 보인다.

5. 포트 포워딩, 라우팅과 QoS에 영향을 줄 수 있는 TOS 처리, 리다이렉트, ICMP 도달불가(unreachable)과 마크(mark) 등이 패킷 필터링 코드에 포함되어 있다.
6. ipchains 코드는 모듈화되어 있지도 않고 확장할 수도 없다. (예: MAC 어드레스 필터링, 옵션 필터링 등)
7. 하부구조가 불충분하기 때문에 다른 기술을 낭비하게 만들었다.
   - per-protocol 모듈을 더한 매스커레이딩
   - 라우팅 코드에 의한 빠른 정적 NAT(per-protocol 처리를 포함하지 않았다).
   - 포트 포워딩, 리다이렉트, 자동 포워딩
   - The Linux NAT and Virtual Server Projects.

8. CONFIG_NET_FASTROUTE와 패킷 필터링간의 호환성 결여:
   - 포워드 패킷은 세 개의 체인을 거쳐 전달된다.
   - 이러한 체인을 무시하고 지나간 경우 알려줄 방법이 없다.

9. 라우팅 프로텍스로 인해 버려진 패킷을 관찰할 수 없다. (즉, Source Address Verification)
10. 패킷 필터링 룰에 대하여 자동으로 카운터를 읽어낼 방법이 없다.
11. CONFIG_IP_ALWAYS_DEFRAG은 컴파일할 때 주는 옵션이라서 일반적인 목적으로 원하는 커널을 배포판을 만들기가 어렵다.

1.1.2. 누구시죠?

나는 이런 짓을 하리만큼 바보스러운 사람이다. ipchains의 공동저자이고 현재 리눅스 커널 IP 방화벽의 메인터너로서, 현재의 시스템 때문에 사람들이 많은 문제를 격었다는 것 뿐만 아니라 그들이 시도하고 있는 것이 점점 더 노출되고 있다는 것을 알았다.

1.1.3. 그게 왜 폭주하죠?

이~~야!, 지난 주에 이 문서를 봤어야 하는 건데...

사실 나는 우리 모두가 되기 원하는 그런 훌륭한 프로그래머가 아니고, 시간, 장비 그리고 영감도 부족해서 시나리오 전체를 충분히 테스트 해보지 못했다. 그저 내가 해본 것이라고는 여러분들이 참여하기를 바라는 마음에서 만든 testsuite를 돌리는 것이 고작이었다.

2. 어디서 최신 버전을 구하죠?

최신의 HOWTO, userspace tools 그리고 testsuite를 가지고 있는 CVS 서버가 samba.org에 있다. 통상적인 브라우징 방법으로는, 웹 인터페이스 (http://cvs.samba.org/cgi-bin/cvsweb/netfilter/)를 사용할 수 있다. 최신의 소스를 얻으려면 다음과 같이 하면 된다:

1. anoymous로 SAMBA CVS 서버에 로그인한다:

   ```
   cvs -d :pserver:cvs@cvs.samba.org:/cvsroot login
   ```

2. 패스워드를 물어보면 'cvs'라고 친다.
3. 다음 명령을 이용하여 코드를 체크한다:

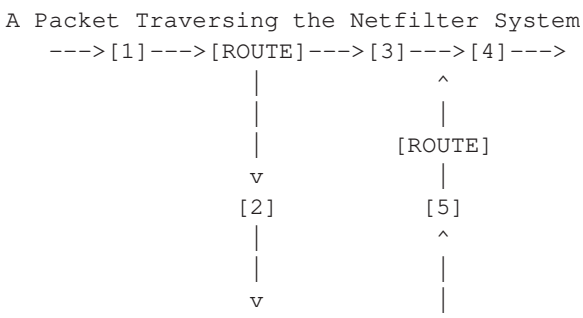
   ```
   cvs -d :pserver:cvs@cvs.samba.org:/cvsroot co netfilter
   ```

4. 최신 버전으로 업데이트하려면, 다음과 같이 한다.

   ```
   cvs update -d -P
   ```

3. 넷필터 아키텍처

넷필터는 단지 프로토콜 스택의 다양한 포인트에 존재하는 훅의 연속일 뿐이다. 이상적인 IPv4의 진행경로 다이어그램은 다음과 같다.



패킷은 그림의 좌측으로부터 들어와서 단순한 데이터 체크(즉, 데이터가 잘렸는지, 혹은 IP 체크 섬의 이상유무, 뒤죽박죽 되지는 않았는 지 등)를 거쳐, 넷필터 프레임웍의 NF_IP_PRE_ROUTING[1] 훅으로 전달된다.

다음으로, 패킷은 라우팅 코드로 들어가며, 여기서 패킷이 다른 인터페이스로 향하는지 또는 로컬 프로세스로 향하는지 결정된다. 패킷이 라우팅될 수 없는 경우, 라우팅 코드는 패킷을 버리기도 한다.

만일 패킷의 목적지가 들어온 박스라면, 넷필터 프레임웍은 패킷을 프로세스로 전달하기 전에 NF_IP_LOCAL_IN [2] 훅을 다시 한번 호출하게 된다.

만일 다른 인터페이스로 전달하고자 한다면, 넷필터 프레임웍은 NF_IP_FORWARD [3] 훅을 호출한다.

그리고 나서 패킷이 넷트웍 라인으로 보내지기 전에 마지막 넷필터 훅인 NF_IP_POST_ROUTING [4] 훅으로 전달된다.

로컬에서 생성된 패킷에 대해서는 NF_IP_LOCAL_OUT [5] 훅이 호출된다. 이 때, 이 훅이 호출된 후 라우팅이 발생하는 것을 알 수 있다. 실제로는 라우팅 코드가 소스 IP 주소와 몇 가지 IP 옵션을 확인하기 위해 라우팅 코드가 먼저 호출이 된다. 즉, 라우팅을 변경하고자 한다면, NAT 코드에 되어 있는 것처럼 여러분 스스로 'skb->dst' 필드를 변경해야만 한다.

3.1. 넷필터의 기초

이 절에서는 IPv4에 대한 넷필터의 예를 보일 것이며, 이를 통해 여러분들은 각각의 훅이 동작하는 시점을 이해하게 될 것이다. 이 것이 바로 넷필터의 기초이다.

커널 모듈은 앞서 언급한 어떠한 훅에 대해서 응답할 수 있도록 등록할 수 있으며, 어떤 함수를 등록한 모듈은 훅 내에서 함수의 우선순위에 대하여 반드시 명시하여야 한다. 코어 네트워킹 코드로부터 넷필터 훅이 호출되는 경우, 각 포인트에 등록된 각각의 모듈은 우선순위에 따라 호출이 되고, 패킷을 자유로이 다룰 수 있다. 모듈은 넷필터에게 다음의 다섯 가지 동작을 하도록 요청한다.

1. NF_ACCEPT: 보통처럼 계속 진행시킨다.
2. NF_DROP: 패킷을 버린다. 즉 계속 진행시키지 않는다.
3. NF_STOLEN: 패킷을 접수하겠다. 즉 계속 진행시키지 않는다.
4. NF_QUEUE: 패킷을 큐로 보낸다.(보통 사용자 공간에서의 처리를 목적으로 한다.)
5. NF_REPEAT: 현재 훅을 다시 호출한다.

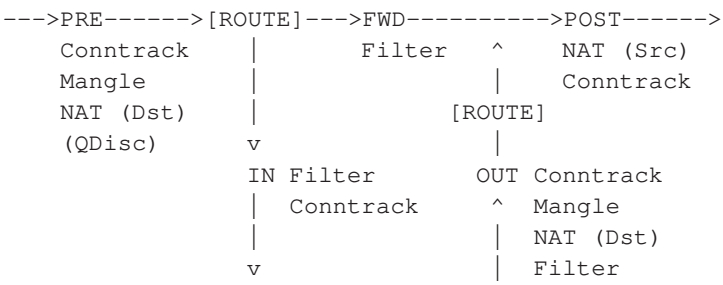
넷필터의 다른 부분은 뒤에 나오는 커널 부분에서 다루기로 한다.

이상과 같은 것이 기초가 되어, 저자들은 다음 두 절에 보인 것과 같은 복잡한 패킷 처리를 만들 수 있다.

3.2. 패킷 선택: IP Tables

IP table을 호출하는 패킷 선택 시스템은 넷필터 프레임웍을 기반으로 구성되었으며, 이는 확장성을 가지고 ipchains로부터 직접 물려받은 유산이다.(ipchains는 ipfwadm으로부터 물려받고, ipfwadm은 BSD의 ipfw IIRC로부터 물려받았다.) 커널 모듈은 새로운 테이블을 등록할 수 있으며, 임의의 패킷이 주어진 테이블을 통과하도록 요청할 수 있다. 이러한 패킷 선택 방법은 패킷 필터링(즉 'filter' 테이블), 네트웍 주소 변환('nat' 테이블) 그리고 종합적인 pre-route 패킷 mangling('mangling' 테이블')에 사용한다.

넷필터에 등록된 훅은 다음과 같다.(여기서는 각각의 함수가 실제로 호출되는 순서로 각각의 훅에 있는 함수와 같이 보였다.)



3.2.1. 패킷 필터링

'filter'라는 테이블은 패킷을 절대로 변경시키지 않고 단지 걸러내기만 한다.

ipchains와 비교했을 때, iptables filter의 장점 중 하나는 작고 빠르다는 것이며, NF_IP_LOCAL_IN과 NF_IP_FORWARD, NF_IP_LOCAL_OUT 시점에서 넷필터로 훅킹된다. 이는, 주어진 패킷에 대하여 이를 필터링 하는 위치가 오직 하나 뿐이라는 것을 의미한다. 결국 사용자들이 ipchains를 사용한 것보다 더 단순하게 만들며, 또한 넷필터 프레임웍이 NF_IP_FORWARD 훅에 대하여 입력과 출력 인터페이스를 제공한다는 사실은 다양한 필터링이 훨씬 단순해진다는 것을 의미한다.

주: 필자는 업그레이드의 필요 없이 기존의 ipfwadm과 ipchains를 사용 가능하도록 넷필터의 최상위에 모듈로서 ipchains와 ipfwadm의 커널부분을 포팅 하였다.

3.2.2. NAT

이는 'nat' 테이블의 영역으로, 두 가지의 넷필터 훅으로 패킷을 전달한다. 즉, non-local 패킷에 대해, 각각 목적지와 소스 전환에 대하여 NF_IP_PRE_ROUTING과 NF_IP_POST_ROUTING 훅이 완벽하게 동작한다. CONFIG_IP_NF_NAT_LOCAL이 정의된 경우, NF_IP_LOCAL_OUT과 NF_IP_LOCAL_IN 훅이 로컬 패킷의 목적지를 전환하기 위해 사용된다.

이 테이블은 'filter' 테이블과는 약간 다르며, 새로운 커넥션의 첫 번째 패킷만이 테이블로 전달된다. 따라서 이와 같은 전달의 결과는 동일한 커넥션에 있어서 향후 전달되는 모든 패킷에 적용된다.

3.2.3. 매스커레이딩, 포트 포워딩, 투명한 프락시

필자는 NAT를 출발지 NAT(즉 첫 번째 패킷이 출발지를 변경하는 경우)와 목적지 NAT(첫 번째 패킷이 목적지를 변경하는 경우)로 구분하였다.

매스커레이딩은 출발지 NAT의 특별한 경우이며, 포트 포워딩과 투명한 프락시는 목적지 NAT의 특별한 경우이다. 이와 같은 것은 서로 독립적인 엔티티를 가지고 NAT 프레임웍을 이용하여 구현되었다.

3.2.4. 패킷 맹글링(packet mangling)

패킷 맹글링 테이블('mangling' table)은 패킷의 정보를 실제로 변경하기 위해 사용되며, NF_IP_PRE_ROUTING과 NF_IP_LOCAL_OUT 시점에서 넷필터로 훅킹된다.

3.3. 연결 추적

연결추적(connection tracking)은 NAT의 기본이지만, 모듈로 분리되어 구현된다. 이는 연결추적을 단순하고 명확하게 사용할 수 있도록 패킷 필터링 코드에 대한 확장성을 제공한다.('state' 모듈)

3.4. 그 이외 추가된 사항

새로운 유연성 때문에 정말로 무서운 일을 겪을 기회가 많아졌지만, 다른 사람들이 향상된 코드를 작성하거나 기존의 코드를 완전히 대체해 버릴 수 있는 기회 역시 많아졌다.

4. 프로그래머들을 위한 정보

비밀을 하나 말씀드리겠습니다. 뭐냐하면, 제가 기르는 햄스터가 모든 코드를 작성했습니다. 저는 단지 전달하는 역할만 했고, 모든 계획은 제 애완동물이 했습죠. 그러니 버그가 생기더라도 저를 원망하지 마시고, 귀여운 털북숭이를 원망하시기 바랍니다.

4.1. ip_tables의 이해

iptables는 메모리 내에 있는 규칙의 명명된 배열과 각각의 훅으로부터 패킷이 전달되기 시작해야 하는 정보를 단순히 제공만 하는 것이다. 어떤 테이블이 등록되고 나면, 사용자 공간은 getsockopt()과 setsockopt()를 이용하여 그 내용을 읽고 변경할 수 있다.

iptables는 어떠한 넷필터 훅에도 등록하지 않으며, 이를 수행하는 다른 모듈에 의존하고 적절히 패킷을 모듈에 전달한다. 다시 말해, 하나의 모듈은 넷필터 훅과 ip_tables에 따로따로 등록해야한 하고, 훅이 발생하면 ip_tables를 호출하는 메커니즘을 제공한다.

4.1.1. ip_tables의 데이터 구조

편리성을 위해, 동일한 데이터 구조를 사용하여 사용자 공간에 의한 규칙과 커널내부의 규칙을 표현하였다. 이렇게 표현된 데이터 구조 중 아주 일부분만이 커널 내부에서 사용된다.

각각의 규칙은 다음과 같은 부분으로 구성된다.

1. 'struct ipt_entry'
2. zero 또는 그 이상의 'struct ipt_entry_match' 구조로, 각각은 여기에 추가 가능한 데이터의 크기를 변경할 수 있다.
3. 'struct ipt_entry_target' 구조: 추가 가능한 데이터 크기 변화 가능

규칙의 변화 가능한 특성은 확장성에 대하여 상당한 유연성을 제공하며, 특히 각각의 match 혹은 타깃이 임의 크기의 데이터를 전달할 수 있도록 한다. 반면 이는 몇 가지 함정을 만들게 되는 데, 반드시 정렬(alignment)에 주의해야만 한다. 'ip_entry'와 'ipt_entry_match', 'ipt_entry_target' 구조가 크기 변경이 편리하도록 하고, IPT_ALIGN() 매크로를 이용하여 장비의 최대 정렬(alignment)까지 모든 데이터들을 모으는 것 등을 확실하게 함으로써 정렬을 구현하였다.

'struct ipt_entry'는 다음과 같은 필드를 포함한다.

1. 'struct ipt_ip' : IP header에 대한 세부항목을 포함
2. 'nf_cache' : 현재의 규칙을 검사해야하는 패킷의 부분을 알려주는 비트 필드
3. 'target_offset' : ipt_entry_target 구조가 시작하는 현재 규칙의 시작점으로부터의 offset을 알려주는 필드
4. 'next_offset' : 현재 규칙의 최대 크기를 알려주는 필드로 match와 target을 포함한다. 이 것 역시 IPT_ALIGN 매크로를 이용하여 정렬되어야 한다.

5. ‘comefrom’ : 패킷의 경로를 추적하기 위해 커널이 사용하는 필드
6. ‘struct ipt_counters’ : 현재 규칙에 일치하는 패킷에 대한 바이트 카운터와 패킷을 포함하는 필드

‘struct ipt_entry_match’와 ‘struct ipt_entry_target’은 상당히 유사하며, 전체(IPT_ALIGN으로 정렬된) 길이 필드(각각 ‘match_size’와 ‘target_size’)와, match와 target(사용자 공간에 대한) 명칭을 포함하는 구조체, 그리고 (커널에 대한) 포인터를 포함한다.

4.1.2. ip_tables의 사용과 진행

커널은 특정한 훅에 의해 지시된 위치에서 관찰을 시작하여, 그에 관련한 규칙을 검사하고, ‘struct ipt_ip’의 element가 일치하는 경우, 차례로 각각의 ‘struct ipt_entry_match’를 검사한다(match가 호출된 곳과 관련한 match function을 수행한다). match function이 0을 돌려주는 경우, 현재 규칙에 대한 반복을 중단한다. ‘hotdrop’ 파라미터가 1로 설정된 경우, 현재 패킷은 즉시 폐기된다(tcp match 함수와 같은 곳에서 조금이라도 수상한 패킷에 대해 사용한다).

수행반복이 끝까지 진행된 경우, 카운터가 증가하고, ‘struct ipt_entry_target’이 검사된다. 표준 타깃인 경우, ‘verdict’ 필드를 읽는다. ‘verdict’ 필드가 음인 경우 패킷이 결정된 것을 의미하고 양인 경우는 이동해야할 offset을 의미한다. 응답이 양이고 offset이 다음 규칙을 가리키지 않으면, ‘back’ 이라는 변수가 세트되고 이전의 ‘back’ 값이 현재 규칙의 ‘comefrom’ 필드에 설정된다.

비표준 타깃에 대해서는 target 함수가 호출되어, 결정을 알려주게 된다.(비표준 타깃은 정적 루프 검출 코드를 위반하기 때문에 이동할 수 없다) 그 결정은, 다음 규칙으로 계속 진행하기 위해서는 IPT_CONTINUE가 될 것이다.

4.2. iptables 확장하기

전 무지하게 게으른 넘이라서, iptables는 얼마든지 확장 가능합니다. 다시 말하면 제 손을 떠나 다른 사람에게 넘어간, 오픈소스 그 이상이라는 거죠.

iptables를 확장한다는 것은 다음의 두 부분을 포함한다. 즉, 새로운 모듈을 작성하여 커널을 확장하는 것과 새로운 공유 라이브러리를 작성하여 사용자 차원의 프로그램인 iptables를 확장하는 것이다.

4.2.1. 커널

예제를 보신 사람들은 알겠지만, 커널 모듈을 작성한다는 것 자체는 상당히 단순하다. 한가지 알아야 할 것은 여러분의 코드가 재진입 가능해야 한다는 것이다. 예를 들어보면, 사용자 공간으로부터 들어오는 어떤 패킷이 존재할 수 있을 것이며, 동시에 또 다른 패킷이 인터럽트에 의해 들어 올 수도 있을 것이다. 하지만, 커널 2.3.4이상에서 SMP를 사용할 경우, CPU당 하나의 인터럽트에 대해 하나의 패킷만 존재하게 된다.

여러분들이 알아야 하는 함수는 다음과 같다.

init_module()

모듈의 진입 포인트로 에러가 발생한 경우 음수를 넘겨주며 넷필터에 성공적으로 등록이 된 경우 0을 돌려준다.

cleanup_module()

모듈의 종료 포인트로 넷필터에서 모듈자체를 등록해제한다.

ipt_register_match()

새로운 match 타입을 등록하기 위하여 사용하며, 이 것을 'struct ipt_match'로 전달해야 한다. 통상 'struct ipt_match'는 정적 변수로 정의한다.

ipt_register_target()

새로운 타입을 등록하기 위해 사용하며, 이 것을 'struct ipt_target'으로 전달해야 한다. 보통 'struct ipt_target'은 정적 변수로 전달된다.

ipt_unregister_target()

target을 등록해제하기 위해 사용한다.

ipt_unregister_match()

match를 등록해제하기 위해 사용한다.

여러분이 작성한 새로운 match나 target에 대한 새로운 공간 내에서의 편법 사용(카운터 기능 제공 같은)에 대한 한가지 경고를 하겠다. SMP 머신의 경우 각각의 CPU에 대하여 전체 테이블을 memcpy()를 이용하여 복사한다. 즉 중심이 되는 정보를 보존하기 바란다면, 'limit' match에 사용된 방법을 찾아봐야만 할 것이다.

4.2.1.1. 새로운 **Match** 함수

새로운 match function은 일반적으로 독립모듈로 작성한다. 바꾸어 말하면, 비록 통상적으로 필요하지 않더라도, 이러한 모듈에 확장성을 제공하는 것이 가능하다는 것이다. 따라서, 사용자들이 직접 여러분이 작성한 모듈과 통신할 수 있도록 넷필터 프레임웍의 'nf_register_sockopt' 함수를 사용하는 것이 하나의 방법이 될 것이다. 또 다른 방법은 넷필터 모듈과 ip_tables에 구현된 것과 동일한 방법으로, 다른 모듈이 자신을 등록하도록 심벌을 export하는 것이다.

여러분 작성한 새로운 함수의 핵심은 ipt_register_match()로 전달되는 ipt_match 구조체이다. 이 구조체는 다음과 같은 필드를 포함한다.

list

임의의 값으로 설정되는 필드이다. 즉 '{ NULL, NULL }'

name

사용자 공간에서 참조되는 match함수의 이름을 저장하는 필드이다. 자동 로딩 기능이 동작하기 위해서 함수의 이름은 모듈의 이름과 일치하여야 한다. 예를 들면, 함수이름이 “mac”인 경우, 모듈이름은 반드시 “ipt_mac.o”이어야 한다.

match

match 함수의 포인터를 저장하는 필드로, skb, 입/출력 장치의 포인터(훅에 따라 둘 중 하나는 NULL이 될 수도 있다), 동작시킬 룰에 해당하는 match 데이터의 포인터, IP 오프셋(non-zero는 non-head 프래그먼트를 의미), 프로토콜 헤더에 대한 포인터(과거의 IP header), 데이터의 길이(패킷 길이에서 IP 헤더의 길이를 뺀 크기) 그리고 ‘hotdrop’ 변수에 대한 포인터를 취하게 된다. 패킷이 일치하면 non-zero를 돌려야 주어야 하고, 0을 돌려주는 경우 ‘hotdrop’은 1로 설정할 수 있으며 이는 패킷을 바로 버렸다는 것을 알리기 위한 것이다.

checkentry

룰에 대한 세부명세를 확인하는 함수의 포인터를 저장한다. 등록된 함수가 0을 돌려주는 경우, 현재의 룰은 사용자로부터 받아들여지지 않을 것이다. 예를 들면, “tcp” match 타입은 오직 tcp 패킷만 받아들일 것이고, 따라서 룰의 ‘struct ipt_ip’ 부분에 프로토콜은 반드시 tcp이어야 한다고 명시되어 있지 않는 한 0이 리턴 될 것이다. tablename 인자는 사용자의 match가 어떤 테이블을 사용할 수 있는지를 허가하고, ‘hook_mask’는 사용자의 룰이 호출될 수 있는 훅에 대한 비트매스크이다. 만일 여러분의 match가 넷필터의 일부 훅에 대하여 아무런 의미가 없다면, 그 match를 이 위치에서 없앨 수 있다.

destroy

현재의 match가 삭제될 경우 호출되는 함수의 포인터를 저장하며, 사용자로 하여금 checkentry에서 동적으로 리소스를 재배치하고 또 이를 없앨 수 있도록 한다.

me

‘THIS_MODULE’로 설정되며, 이는 여러분의 모듈에 대한 포인터를 돌려준다. 어떤 타입의 규칙이 생성되거나 소멸되는 경우 usage-count를 증가 혹은 감소시킨다. 어떤 규칙이 이 것을 참조하고 있음에도 불구하고 사용자가 모듈을 제거하고자 하는 경우, 사용자가 모듈을 제거하지 못하도록 한다.

4.2.1.2. 새로운 **Targets**

여러분의 타깃이 패킷(헤더나 바디)을 변경시킨다면, 패킷이 복제되는 시점에서 패킷을 복사하기 위하여 skb_unshare()를 호출해야만 한다. 그렇지 않으면 skbuff에 복제된 패킷이 있는 어떠한 raw socket이라도 변경된 사항을 알아차리게 된다.

새로운 target은 통상적으로 단독 모듈로 작성된다. ‘New Match Functions’의 절에서 언급한 바와 동일한 내용이 여기서도 적용된다.

여러분의 새로운 target의 핵심은 ipt_register_target()으로 전달되는 struct ipt_target으로, 이 구조체는 다음과 같은 필드를 갖고 있다.

list

임의의 값으로 설정되는 필드이다. 즉 '{ NULL, NULL }'

name

사용자 공간에서 참조되는 target 함수의 이름을 저장하는 필드이다. 자동 로딩 기능이 동작하기 위해서 함수의 이름은 모듈의 이름과 일치하여야 한다. 예를 들면, 함수이름이 "REJECT"인 경우, 모듈이름은 반드시 "ipt_REJECT.o"이어야 한다.

target

target 함수의 포인터를 저장하는 필드로, skbuff, 훅 넘버, 입/출력 장치의 포인터(훅에 따라 둘 중 하나는 NULL이 될 수도 있다), target 데이터의 포인터, 테이블에 있는 룰의 위치를 값으로 갖게 된다. 패킷이 계속 진행해야 한다면 target 함수는 IPT_CONTINUE(-1)을 돌려주고, 그렇지 않은 경우는 NF_DROP, NF_ACCEPT, NF_STOLEN등을 돌려주어 패킷의 운명을 결정하게 된다.

checkentry

룰에 대한 세부명세를 확인하는 함수의 포인터를 저장한다. 등록된 함수가 0을 돌려주는 경우, 현재의 룰은 사용자로부터 받아들여지지 않을 것이다.

destroy

현재의 target이 사용중인 entry가 삭제될 경우 호출되는 함수의 포인터를 저장하며, 사용자로 하여금 checkentry에서 동적으로 리소스를 재배치하고 또 이를 없앨 수 있도록 한다.

me

'THIS_MODULE'로 설정되며, 이는 여러분의 모듈에 대한 포인터를 돌려준다. 어떤 타입의 규칙이 생성되거나 소멸되는 경우 usage-count를 증가 혹은 감소시킨다. 어떤 규칙이 이 것을 참조하고 있음에도 불구하고 사용자가 모듈을 제거하고자 하는 경우, 사용자가 모듈을 제거하지 못하도록 한다.

4.2.1.3. 새로운 **Tables**

여러분들의 목적에 맞는 새로운 table을 작성할 수 있으며, 이를 위해서는 'ipt_register_table()' 함수를 호출해야한다. 이 함수는 전달인자로 'struct ipt_table'을 받으며 그 구조는 다음과 같다.

list

임의의 값으로 설정되는 필드이다. 즉 '{ NULL, NULL }'

name

사용자 공간에서 참조되는 table 함수의 이름을 저장하는 필드이다. 자동 로딩 기능이 동작하기 위해서 함수의 이름은 모듈의 이름과 일치하여야 한다. 예를 들면, 함수이름이 "nat"인 경우, 모듈이름은 반드시 "ipt_nat.o"이어야 한다.

table

'struct ipt_replace'로 가득 찬 필드로, 테이블을 교체하기 위하여 사용자 공간에서 사용된다. 'counters' 포인터는 NULL로 설정되어야 한다. 이 구조체는 '__initdata'로 선언되기 때문에 부팅 후에는 초기화된다.

valid_hooks

테이블로 진입하고자 하는 IPv4 넷필터 훅의 비트매스크이다. 진입엔트리가 유효한지 확인하고, ipt_match와 ipt_target의 'checkentry()'함수에 대해 사용 가능한 훅을 계산하기 위해 사용한다.

lock

전체 테이블에 대하여 읽고 쓰기가 가능한 spinlock이며, RW_LOCK_UNLOCKED로 초기화 된다.

private

ip_tables 코드에 의해 내부적으로 사용된다.

4.2.2. 사용자공간 도구(Userpace Tool)

이제 여러분들이 직접 커널 모듈을 작성했고, 사용자 공간에서 이에 대한 옵션을 조정하기를 원할 것이다. 필자는 각각 확장된 버전에 대하여 새로이 파생된 버전을 만드는 것보다는 아주 최신의 90년대 기술을 사용한다. 즉 furbies이다. 쏴리... 공유라이브러리를 말하는 것이다.

새로운 테이블을 사용하고자 하는 경우 iptables를 확장할 필요는 없고, '-t' 옵션만 주면 된다.

공유라이브러리는 '_init()'함수를 포함해야하며, 이 함수는 모듈이 로딩 되는 시점에서 자동으로 호출된다. 여러분이 작성한 공유라이브러리가 새로운 match나 새로운 target을 포함하느냐에 따라 _init()함수가 'register_match()'나 'register_target()'함수를 호출한다.

공유라이브러리를 제공해야할 필요도 있으며, 구조체의 일부를 초기화하거나 추가 옵션을 제공하는 데 공유라이브러리를 사용할 수도 있기 때문이다. 필자는 공유라이브러리가 아무것도 안 할지라도 공유라이브러리는 라이브러리가 없을 때 발생하는 문제를 줄일 수 있기 때문에 반드시 공유라이브러리를 사용하기를 주장한다.

'iptables.h' 헤더에 유용한 함수들이 정의되어 있으며, 그중 다음과 같은 것들이 상당히 유용하다.

chech_inverse()

전달인자가 '!'인지 검사하고, 맞으면 'invert' 플랙이 설정되고, 그렇지 않으면, 'invert' 플랙을 설정한다. true가 리턴된 경우 예제에서 보인 바와 같이 optiond를 증가시켜야한다.

string_to_number()

스트링을 주어진 범위 내의 숫자로 변환한다. 형식이 잘 못 되었거나 범위를 벗어나면, -1이 리턴된다. 'string_to_number'는 'strtol'을 사용한다. 다시 말하면, 선행문자열 "0x"는 16진수라는 것을 의미하고, "0"은 8진수라는 것을 의미하게 된다.

exit_error()

에러가 검출된 경우 호출되는 함수이다. 일반적으로 첫 번째 인수는 'PARAMETER_PROBLEM'이며, 사용자가 커맨드 라인을 정확하기 사용하지 않았다는 것을 의미한다.

4.2.2.1. 새로운 **Match** 함수

여러분들이 작성한 공유라이브러리의 _init() 함수는 'register_match()'에 정적 구조체인 'struct iptables_match'에 대한 포인터를 넘겨준다. 이 구조체는 다음과 같은 필드를 포함한다.

next

match의 링크드 리스트를 만들기 위해 사용되는 포인터로, 초기에는 NULL로 설정된다.

name

match 함수의 이름으로, 라이브러리의 이름과 일치해야한다.(즉, 'libipt_tcp.so'에 대해서는 "tcp"와 같이...)

version

일반적으로 NETFILTER_VERSION 매크로로 설정되며, iptables 바이너리파일이 실수로 엉뚱한 공유라이브러리를 선택하지 않도록 하기 위하여 사용된다.

size

현재 사용하는 match에 대한 match 데이터의 크기로, 정확하게 정렬하기 위해서는 IPT_ALIGN() 매크로를 사용하여야 한다.

userpacesize

일부 match에 대해, 커널은 일부 필드를 내부적으로 변경한다. 'limit' target이 좋은 예이다. 이는 단순한 'memcmp()'함수로는 두개의 룰을 비교하기에는 부족하다는 것을 의미한다. 만일 이러한 경우가 발생하면 구조체의 시작점에서 변경되지 않는 모든 필드를 위치시키고, 변경되지 않은 필드의 크기를 여기에 집어넣게 된다. 그러나 일반적인 경우, 이 필드는 'size'필드와 동일한 값을 갖는다.

help

옵션의 사용법을 화면에 출력한다.

init

ipt_entry_match 구조체에 존재하는 별도의 공간(만일 존재한다면)을 초기화하는 데 사용할 수 있고, 어떠한 nfcache 비트도 1로 설정한다. 'linux/include/netfilter_ipv4.h'의 내용을 이용하여 표현할 수 없는 어떤 것을 검사하고있다면, 간단히 NFC_UNKNOWN 비트와 OR를 취하면 된다. 이 함수는 'parse()'보다 먼저 호출되어야 한다.

parse

커맨드 라인에 알 수 없는 옵션이 주어진 경우 호출되며, 여러분이 작성한 라이브러리에 주어진 옵션이 존재하는 경우 non-zero를 리턴 한다. '!'이 이미 나타난 경우는 'invert'가 TRUE로 설정된다. 'flags' 포인터는 여러분들이 작성한 라이브러리의 배타적 사용을 위한 것이며, 특별히 명시된 옵션의 비트매스크를 저장하기 위하여 사용한다. 여러분들은 ncfcache 필드를 확실히 조정할 수 있도록 해야하며, 필요한 경우에는 'ipt_entry_match' 구조체의 크기를 확장할 수 있어야 한다. 다만 그 크기는 반드시 IPT_ALIGN 매크로를 거쳐서 전달되어야 한다.

final_check

커맨드 라인이 파싱된 후 호출되는 함수이며, 여러분의 라이브러리를 위해 예약된 'flags' 정수를 다루게 된다. 이를 이용하면 어떤 강제적인 옵션이 명시되었는 지 확인할 수 있다. 이러한 경우가 발생하면 'exit_error()'을 호출해야 한다.

print

어떤 룰에 대하여 부가적인 match 정보를 출력하기 위해 chain listing 코드에서 사용하는 함수이다. 사용자가 '-n' 플랙을 명시한 경우 numeric flag이 설정된다.

extra_opts

여러분의 라이브러리가 제공하는 부가 옵션의 null-terminated 리스트이다. 이 옵션은 현재 사용 중인 옵션에 더해져서 getopt_long으로 전달된다. 보다 자세한 것은 man 페이지를 보는 것이 좋을 것이다. getopt_long에 대한 리턴 값은 여러분이 작성한 'parse()' 함수에 대한 첫 번째 인자가 된다.

iptables에 의해 내부적으로 사용하기 위한 이 구조체의 마지막 부분에 부가적인 필드가 존재하지만 그 값을 설정할 필요는 없다.

4.2.2.2. 새로운 **Targets**

여러분의 공유라이브러리의 _init() 함수는 'register_target()' 함수로 정적으로 선언된 'struct iptables_target'에 대한 포인터를 전달하며, 이는 앞서 언급한 iptables_match 구조체와 유사한 필드를 포함한다.

4.2.3. '**libiptc**' 사용하기

libiptc는 iptable 제어 라이브러리로 iptable 커널 모듈에서 룰을 나열하고 처리하기 위하여 설계되었다. 이 라이브러리가 현재 사용되고 있는 곳은 iptables 프로그램뿐이지만, 다른 툴을 개발하는 곳에도 쉽게 사용할 수 있다. 이 함수를 사용하기 위해서는 루트 권한이 필요하다.

이 함수가 제공하는 표준 target은 ACCEPT, DROP, QUEUE, RETURN, 그리고 JUMP이다. ACCEPT, DROP, QUEUE는 NF_ACCEPT, NF_DROP과 NF_QUEUE로 번역되고, RETURN은 ip_tables가 처리하는 특별한 IPT_RETURN 값으로, JUMP는 chain name으로부터 table내의 실제 오프셋으로 번역된다.

'iptc_init()' 함수가 호출되면, counter를 포함한 테이블이 읽혀지고, 이 테이블은 'iptc_insert_entry()', 'iptc_replace_entry()', 'iptc_append_entry()', 'iptc_delete_entry()', 'iptc_delete_num_entry()', 'iptc_flush_entries()', 'iptc_zero_entries()', 'iptc_create_chain()', 'iptc_delete_chain()' 그리고 'iptc_set_policy()' 함수에 의해 처리된다.

'iptc_commit()' 함수가 호출되기 전까지는 테이블의 변화가 기록되지 않는다. 따라서, 라이브러리를 사용하는 두 명의 유저가 동일한 chain을 조작하고자 하기 위해 레이스(race)를 하는 경우가 발생할 수 있으며, 이를 막기 위해서는 locking을 사용해야 하지만, 현재는 구현되어 있지 않다.

하지만, counters에 대해서는 레이스(race)가 발생하지 않으며, 이는 tables의 읽기와 쓰기 중에 발생하는 counters의 증가가 새로운 table에 나타나는 방식을 이용하여 counter가 커널에서 기록되기 때문이다.

여기에는 다음과 같은 다양한 helper 함수가 있다.

iptc_first_chain()

table 내의 첫 번째 chain의 이름을 리턴 한다.

iptc_next_chain()

table 내의 다음 chain의 이름을 리턴하며, NULL은 더 이상 chain이 없다는 것을 의미한다.

iptc_builtin()

주어진 chain name이 builtin chain name이면 TRUE를 리턴 한다.

iptc_first_rule()

주어진 chain name내의 첫 번째 룰의 포인터를 리턴하며, NULL인 경우는 chain이 비었다는 것을 뜻한다.

iptc_next_rule()

chain내의 다음 룰에 대한 포인터를 리턴하며, NULL인 경우는 chain의 끝이라는 것을 알려준다.

iptc_get_target()

주어진 룰의 target을 가져온다. 확장된 target이라면, 그에 해당하는 target의 name을 돌려준다. 다른 chain으로의 jump인 경우는 그 chain의 이름을 돌려준다. 또 결정(DROP 같은)

인 경우, 그 name을 돌려준다. accounting-style rule처럼 target이 없는 경우는 null string을 돌려준다.

이 함수가 표준 결정(판정)에 대하여 보다 확장된 해석을 제공하기 때문에, ipt_entry 구조체의 'verdict' 필드의 값을 직접 사용하는 대신에 이 함수를 사용하는 것이다.

iptc_get_policy()

builtin chain의 정책을 가져오고, 그 정책에 대한 적중 통계치를 'counters' 인자에 채운다.

iptc_strerror()

iptc 라이브러리내의 failure code에 대하여 보다 의미 있는 해석을 리턴 한다. 어떤 함수가 에러를 발생한 경우, 항상 errno가 설정되며, 이 값은 iptc_strerror() 함수로 전달되어 에러 메시지로 변환된다.

4.3. NAT의 이해

이제 커널의 NAT(Network Address Translation)까지 오셨군요. 여기서 제공되는 하부구조는 효율보다는 완벽성에 중점을 두고 설계된 것이며, 향후 개조를 통해 효율성이 현저하게 증가될지도 모릅니다. 현재 저는 이 넘이 동작한다는 것만으로도 행복합니다.

NAT는 패킷을 전혀 처리하지 않는 connection tracking과 NAT 코드 자체로 분리되었다. connection tracking 역시 iptables 모듈에서 사용하기 위해 설계되었으며, 따라서 NAT가 관심을 두지 않는 상태(state)에 대해서 미묘한 차이를 보이게 된다.

4.3.1. 연결 추적

연결추적(connection tracking)은 우선 순위가 높은 NF_IP_LOCAL_OUT과 NF_IP_PRE_ROUTING 훅으로 훅킹 되며, 이는 패킷이 시스템으로 진입하기 전 그 패킷을 살펴보기 위함이다.

skb에 있는 nfct 필드는 struct ip_conntrack의 내부에 대한 포인터이며, 배열 infos[] 중 한 곳에 존재한다. 즉, 이 배열내의 어떤 요소를 가리키게 함으로써 skb의 상태를 말할 수 있다. 다시 말해, 이 포인터는 state structure와 그 상태에 대한 skb의 관계 모두를 알려주게 된다.

'nfct' 필드를 추출하는 최선의 방법은 'ip_conntrack_get()'을 호출하는 것으로, 'nfct' 필트가 세트되어 있으며 connection 포인터를 돌려주고 그렇지 않은 경우는 NULL을 돌려주며, 현재의 연결에 대한 패킷의 관계를 표현하는 ctinfo을 채운다. 'nfct'의 값들은 수치화(enumerate)되어 있으며, 다음과 같은 값을 갖는다.

IP_CT_ESTABLISHED

원래 방향에 대하여 established connection의 일부인 패킷이다.

IP_CT_RELATED

connection에 관련된 패킷으로, 원래의 방향으로 전달되고 있다.

IP_CT_NEW

새로운 connection을 생성하고자 하는 패킷이다.(분명히, 원래 진행방향에 존재한다)

IP_CT_ESTABLISHED + IP_CT_IS_REPLY

established connection에 관련된 패킷으로, 응답방향(reply direction)으로 향하고 있다.

IP_CT_RELATED + IP_CT_IS_REPLY

connection에 관련된 패킷으로, 응답방향(reply direction)으로 향하고 있다.

즉, 응답패킷(reply packet)은 nfct를 검사하여 그 값이 IP_CT_IS_REPLY 보다 크거나 같은 값인 가로 확인할 수 있다.

4.4. Connection Tracking/NAT 확장하기

이 방식은 여러 가지 프로토콜과 서로 다른 맵핑 타입을 조절하기 위하여 설계된 것으로, 맵핑 타입 중 일부는 부하분산(load-balancing)이나 fail-over 맵팽 타입처럼 상당히 구체적인 것도 있다.

내부적으로는, connection tracking은 일치하는 룰이나 binding을 검색하기 전에 패킷을 "tuple"로 변환시킨다. 여기서 "tuple"이란 패킷 중 관심의 대상이 되는 부분을 말한다. "tuple"은 처리 가능한 부분과 그렇지 못한 부분으로 구분되며, 각각 "src"와 "dst"라고 한다. 이는 Source NAT의 입장에서 첫 번째 패킷에 대한 관점이다. 동일한 방향에 있어 동일한 패킷 스트림의 각 패킷에 대한 tuple은 모두 동일하다.

예를 들어보면, TCP 패킷의 tuple은 처리 가능한 부분을 포함하는 데 이는 source IP와 source PORT이며, 처리 불가능한 부분은 목적지 IP와 목적지 port이다. 처리 가능한 부분과 그렇지 못한 부분은 반드시 같은 타입이어야 할 필요는 없다. 다시 말하면 ICMP 패킷의 tuple은 source IP와 ICMP port 같은 처리 가능한 부분을 포함하며, 처리 불가능한 부분은 목적지 IP와 ICMP 타입과 코드이다.

각각의 패킷은 inverse를 가지고 있으며, 이는 스트림에 있어서 응답패킷의 tuple이다. 이를테면, icmp id가 12345이고 192.168.1.1에서 1.2.3.4로 가는 ICMP ping 패킷의 inverse는 icmp id 12345이고 1.2.3.4에서 192.168.1.1이 된다.

'struct ip_conntrack_tuple'로 표현되는 tuple은 널리 사용되며, 실제로 패킷이 들어오는 훅과 디바이스를 포함하여 패킷에 대한 완전한 정보를 제공한다.

대부분의 tuple은 'struct ip_conntrack_tuple_hash'에 포함되며, 더블링크드 리스트와 tuple이 포함된 connection에 대한 포인터를 추가한다.

connection은 'struct ip_conntrack'에 의해 표현되며, 이 구조체는 'struct ip_conntrack_tuple_hash' 필드를 두개 포함한다. 하나는 원본 패킷에 대한 방

향(tuplehash[IP_CT_DIR_ORIGINAL])이며, 다른 하나는 응답방향에 대한 패킷(tuplehash[IP_CT_DIR_REPLY])에 대한 것이다.

아무튼, NAT 코드가 하는 첫 번째 일은 skbuff의 nfct 필드를 참조하여 connection tracking 코드로 tuple을 추출할 수 있는 지 확인하고 이미 존재하는 connection을 찾는 것이다. 이것이 의미하는 바는 현재 connection이 새로이 시도된 것인지 아닌지, 그리고 어떤 방향인지를 알려주는 것이다. 여기서 후자인 경우, 그러니까 이미 연결이 된 상태라면, 이전에 결정된 처리방법이 적용된다.

새로운 connection이 시작되면, 표준 iptable 진행 메커니즘을 이용하여 tuple에 대한 룰을 'nat' table에서 검색한다. 이 때 룰이 일치하는 경우, 보통의 경우 진행방향과 응답방향 모두에 대하여 처리를 초기화한다. 즉, 기대하고 있는 응답이 변경되어 버렸다는 것을 connection-tracking 코드가 알 수 있게 된다. 그리고 나서 앞서 언급한 바와 같이 처리된다.

만일 적용할 룰이 없는 경우 'null' 바인딩이 생성된다. 이 바인딩이 패킷과 맵핑되지 않지만, 기존의 다른 스트림과 맵핑되지 않도록 주의해야 한다. 어떤 경우는 null 바인딩이 생성될 수 없는 경우도 발생하며 이 경우는 null 바인딩을 기존의 스트림으로 이미 맵핑을 해버렸기 때문이다. 이러한 경우는 정상적인 'null' 바인딩이라 하더라도 per-protocol으로 이를 새로 맵핑해야 할 것이다.

4.4.1. 표준 NAT Targets

NAT target은 'nat' 테이블 내에서만 사용한다는 것만 제외하면, 여타의 iptables target extension과 상당히 유사하다. SNAT와 DNAT 모두 부가 데이터로서 'struct ip_nat_multi_range'를 취하고, 이 데이터는 맵핑으로 바인딩 하는 주소의 범위를 명시하게 된다. 범위 요소인 'struct ip_nat_range'는 최소/최대 IP와 최대/최소 프로토콜 값(예:TCP 포트)으로 구성된다. 플랙을 위한 공간도 있으며, 어떤 플랙은 IP주소가 맵팽될 수 있는 지 없는지 알려주고, 어떤 것은 범위의 protocol-specific 부분이 유효한지 알려주는 역할을 한다.

다중 범위는 'struct ip_nat_range'의 배열이며, 범위를 "1.1.1.1-1.1.1.2 ports 50-55 AND 1.1.1.3 port 80"과 같이 설정할 수 있다는 것을 의미한다.

4.4.2. 새로운 Protocols

4.4.2.1. 커널 내부

새로운 프로토콜을 구현한다는 것은 tuple의 처리가능 부분과 그렇지 못한 부분을 결정하는 것이다. tuple에 포함된 모든 것은 스트림을 구별할 수 있는 특성을 가지고 있다. tuple의 처리가능부분은 NAT를 적용할 수 있는 부분으로 TCP인 경우는 소스 포트가 되며, ICMP인 경우는 icmp ID가 된다. 즉, 스트림 구별자로 사용할 수 있다는 말이다. 처리 불가능한 부분은 패킷의 나머지 부분으로 스트림을 구별할 수 있지만, 이것을 마음대로 처리할 수 없다. (예: TCP 목적지 포트, ICMP 타입)

한가지가 결정되었으면, connection tracking 코드를 작성할 수 있고, 'ip_conntrack_register_protocol()'로 전달하기 위하여 'ip_conntrack_protocol' 구조체를 다루어야 할 것이다.

'struct ip_conntrack_protocol'의 필드는 다음과 같다.

list

{ NULL, NULL }로 설정한다. 리스트를 확보한다.

proto

프로토콜 번호이며 자세한 것은 '/etc/protocols'를 참조하기 바란다.

name

사용자가 보게 될 프로토콜 명칭이다. '/etc/protocols'에 있는 명칭을 사용하는 것이 제일 좋을 것이다.

pkt_to_tuple

주어진 패킷에 대한 tuple의 프로토콜 상세부분을 채우는 함수이다. 'datah'라는 포인터는 IP 헤더의 시작부분을 가리키며, datalen은 패킷을 길이이다. 패킷이 헤더 정보를 저장하기에 충분히 길지 않으면, 0을 돌려준다. datalen은 최소 8바이트이다.

invert_tuple

tuple의 프로토콜 상세부분을 단순히 패킷에 대한 응답으로 변경하는 데 사용한다.

print_tuple

tuple의 프로토콜 상세부분을 출력하는 데 사용하는 함수로, sprintf() 함수를 이용한다. 사용된 버터 캐릭터의 수가 리턴된다. /proc 엔트리에 대한 상태를 출력하기 위해 사용하기도 한다.

print_conntrack

conntrack 구조체의 공개되지 않은 부분을 출력하는 데 사용하는 함수로, 간혹 /proc에 있는 상태를 출력하기 위해 사용하기도 한다.

packet

established connection의 일부를 보고자 할 때 호출하는 함수이다. conntrack 구조체, IP 헤더, 길이 그리고 ctinfo에 대한 포인터를 얻게 된다. 패킷에 대한 판결로 통상 NF_ACCEPT를 돌려주며, connection이 유효하지 않은 패킷에 대해서는 -1을 돌려준다. 원한다면 이 함수 내부에서 connection을 제거할 수도 있지만, 레이스(race)를 방지하기 위해서는 다음과 같은 방법을 사용해야만 한다.

```
if (del_timer(&ct->timeout))
ct->timeout.function((unsigned long)ct);
```

new

패킷이 최초로 연결을 맺을 때 호출되는 함수로, ctinfo 인자는 없다. 그 이유는 최초의 패킷은 정의에 의해 ctinfo IP_CT_NEW이기 때문이다. 연결을 맺는데 실패하면 0을 돌려주고, 성공한 경우는 순간적으로 connection timeout을 돌려준다.

코드를 작성하고 새로운 프로토콜에 대한 추적을 테스트 했으면, 이제는 NAT에게 이러한 것을 어떻게 해석할 것인지를 알려주어야 한다. 다시 말하면 새로운 모듈을 작성해야 한다는 것이다. 즉, NAT 코드에 대한 확장 및 'ip_nat_protocol_register()'로 전달하고자 하는 'ip_nat_protocol' 구조체를 사용하는 것이다.

list

{ NULL, NULL }로 설정한다. 리스트를 확보한다.

name

사용자가 보게 될 프로토콜 명칭이다. 사용자 공간에 자동으로 로딩 되기 위해서는 '/etc/protocols'에 있는 명칭을 사용하는 것이 제일 좋다.

protonum

프로토콜 번호이며 자세한 것은 '/etc/protocols'를 참조하기 바란다.

manip_pkt

connection tracking의 pkt_to_tuple 함수의 다른 반쪽으로, "tuple_to_pkt"라고 생각해도 무방하다. 약간 다르게 고려해야 할 점은 다음과 같다. IP 헤더의 시작위치에 대한 포인터와 전체 패킷의 길이를 얻는다는 점으로, 일부 프로토콜(UDP, TCP)이 IP 헤더를 알아야 할 필요가 있기 때문이다. 패킷 전체가 아닌 tuple(즉, "src" 필드)로부터 ip_nat_tuple_manip 필드와 수행하고자 하는 처리에 대한 타입을 받아오게 된다.

in_range

주어진 패킷의 처리 가능한 부분이 주어진 범위에 속해있는 지를 알려주는 함수이다. bit tricky 함수로, tuple에 적용할 처리 타입(manipulation type)을 돌려주며, 범위를 어떻게 해석할 것인 가, 즉 처리하고자 하는 것이 소스 범위인지 목적지 범위인가 하는 가를 알려준다.

기존의 맵핑이 올바른 범위에 속해 있는 지 확인하는 데 사용되며, 또한 전혀 처리할 필요가 없는 지 확인하는 데 사용한다.

unique_tuple

NAT의 핵심이 되는 함수이다. tuple과 범위가 주어지면, tuple의 per-protocol 부분을 이 범위에 속하는 tuple로 변경하고, 이 것을 유일하게(unique) 만들어버린다. 주어진 범위에서 사용하지 않는 tuple을 찾아내지 못한 경우 0을 리턴 한다. 또한 ip_nat_used_tuple()에 필요한 conntrack 구조체의 포인터를 얻어낸다.

통상의 방법은 tuple에 대하여 'ip_nat_used_tuple()'을 확인하면서 false가 리턴 될 때까지 범위에서 tuple의 per-protocol 부분을 반복한다.

null-mapping인 경우는 이미 확인이 된 것으로, 주어진 범위 밖에 있거나, 이미 취해진 경우이다.

IP_NAT_RANGE_PROTO_SPECIFIED가 설정되어 있지 않으면, 사용자가 NAPT가 아니라 NAT를 수행하고 있는 것을 의미한다. 즉, 범위 내에서 무엇인가를 한다는 것이다. 맵핑이 필요하지 않다면, 0을 돌려준다.

print

문자버퍼와 범위가 주어진 경우, 그 범위의 per-protocol 부분을 출력하고, 사용된 버퍼의 길이를 돌려준다. IP_NAT_RANGE_PROTO_SPECIFIED 플랙이 주어진 범위에 대해 설정되어 있지 않으면 호출되지 않는다.

4.4.3. 새로운 NAT Targets

매우 흥미로운 부분으로, 새로운 맵핑 타입을 제공하는 새로운 NAT target을 여러분들이 작성할 수 있다. 기본 패키지에는 추가 target은 MASQUERADE와 REDIRECT으로 새로운 NAT target을 작성하기에 충분하리 만큼 쉽게 설명되어 있다.

위의 두 target은 다른 iptables target처럼 작성되어 있지만, 내부적으로는 connection을 추출하고 'ip_nat_setup_info()'를 호출한다.

4.4.4. 프로토콜 도우미(protocol helper)

connection tracking에 대한 protocol helper는 connection code가 다중 네트웍 connection을 사용하는 프로토콜을 알아차리고 초기 연결에 관련된 'child' connection을 표시할 수 있도록 하며, 일반적으로 이와 같은 과정은 data stream 외부의 관련된 주소를 읽음으로써 수행된다.

NAT에 대한 protocol helper는 다음과 같은 두 가지 작업을 수행한다. 첫 째로는 NAT 코드가 데이터 스트림을 포함하는 주소를 변경하도록 데이터 스트림을 처리할 수 있도록 한다. 두 번째로는 데이터 스트림이 들어올 때 그와 연관된 connection에 대하여 원래의 connection을 기초로 하여 NAT를 수행한다.

4.4.5. 연결 추적 도우미 모듈(Connection Tracking Helper Modules)

4.4.5.1. 설명

connection tracking 모듈의 임무는 어떤 패킷이 이미 이루어진 connection에 속해 있는 지를 명시하는 것으로, 다음과 같은 일을 한다.

- 우리 모듈이 어떤 패킷에 관심을 가지고 있는 가를 netfilter에게 알려준다.(대부분의 helper는 특정한 포트에 대해 작업을 한다.)
- netfilter에 함수를 등록한다. 앞서 언급한 범주에 속하는 모든 패킷에 대하여 등록된 함수가 호출된다.

- 등록된 곳으로부터 호출되는 'ip_conntrack_expect_related()' 함수는 netfilter에게 연관된 connection을 예측할 수 있도록 알려준다.

4.4.5.2. 사용가능한 구조체와 함수

여러분들이 작성한 커널 모듈의 init 함수는 'struct ip_conntrack_helper'에 대한 포인터를 가지고 'ip_conntrack_helper_register()' 함수를 호출해야만 한다. 이 구조체는 다음과 같은 필드를 가지고 있다.

list

링크드 리스트에 대한 헤더로, 넷필터가 내부적으로 다루는 리스트이다. '{ NULL, NULL}'로 초기화시킨다.

tuple

'struct ip_conntrack_tuple'로, 우리가 작성한 conntrack helper 모듈이 관심을 갖는 패킷을 명시한 것이다.

mask

역시 'struct ip_conntrack_tuple'이며, tuple의 어느 비트가 유효한지를 명시하고 있는 매스크이다.

help

tuple+mask에 부합하는 각 패킷에 대하여 넷필터가 호출해야하는 함수이다.

4.4.6. conntrack 도우미 모듈의 예제

```
#define FOO_PORT 111
static int foo_nat_expected(struct sk_buff **pksb,
unsigned int hooknum,
struct ip_conntrack *ct,
struct ip_nat_info *info,
struct ip_conntrack *master,
struct ip_nat_info *masterinfo,
unsigned int *verdict)
/* called whenever a related packet (as specified in the connec-
tion tracking
   module) arrives
   params: pksb packet buffer
hooknum HOOK the call comes from (POST_ROUTING, PRE_ROUTING)
ct information about this (the related) connection
info &ct->nat.info
master information about the master connection
```

```
masterinfo &master->nat.info
verdict what to do with the packet if we return 1.
{
        /* Check that this was from foo_expect, not ftp_expect, etc */
/* Then just change ip/port of the packet to the masqueraded
     values (read from master->tuplehash), to map it the same way,
           call ip_nat_setup_info, set *verdict, return 1. */
}
static int foo_help(struct ip_conntrack *ct,
struct ip_nat_info *info,
enum ip_conntrack_info ctinfo,
unsigned int hooknum,
struct sk_buff  **pksb)
/* called for the packet causing related packets
   params: ct information about tracked connection
info (STATE: related, new, established, ... )
hooknum HOOK the call comes from (POST_ROUTING, PRE_ROUTING)
pksb packet buffer
*/
{
/* extract information about future related packets (you can
   share information with the connection tracking's foo_help).
   Exchange address/port with masqueraded values, insert tuple
   about related packets */
}
static struct ip_nat_expect foo_expect = {
{ NULL, NULL },
foo_nat_expected };
static struct ip_nat_helper hlpr;
static int __init(void)
{
int ret;
if ((ret = ip_nat_expect_register(&foo_expect)) == 0) {
memset(&hlpr, 0, sizeof(struct ip_nat_helper));
hlpr.list = { NULL, NULL };
hlpr.tuple.dst.protonum = IPPROTO_TCP;
hlpr.tuple.dst.u.tcp.port = htons(FOO_PORT);
hlpr.mask.dst.protonum = 0xFFFF;
hlpr.mask.dst.u.tcp.port = 0xFFFF;
hlpr.help = foo_help;
ret = ip_nat_helper_register(hlpr);
if (ret != 0)
ip_nat_expect_unregister(&foo_expect);
}
return ret;
}
static void __exit(void)
{
ip_nat_expect_unregister(&foo_expect);
```

```
ip_nat_helper_unregister(&hlpr);
}
```

4.4.7. NAT 도우미 모듈

4.4.7.1. 설명

NAT helper 모듈은 특정 응용프로그램에 적합한 NAT 핸들링을 수행한다. 이 함수는 데이터에 대한 on-the-fly 처리를 포함하고 있다. FTP의 포트 명령을 고려해보자. 이 때, 클라이언트는 서버에게 어떤 IP와 포트로 연결을 해야 하는 가를 물어보게 되고, FTP 제어 연결에서 PORT 명령이 수행된 후 FTP helper 모듈은 IP/port를 교체한다.

TCP를 다루는 경우는 사뭇 복잡해진다. 이유는 패킷 크기가 변하기 때문이다. FTP의 예를 다시 들어보면, PORT 명령이 수행된 후 IP/port tuple을 나타내는 스트링의 길이가 변할 것이다. 결국 패킷 크기가 변경되면, NAT 박스의 좌측과 우측간에 syn/ack의 차이가 발생할 것이다. 예를 들어보면, 패킷을 4 octet만큼 확장했다면, 이후 계속되는 패킷의 TCP sequence number에 앞서 확장시킨 만큼의 offset를 더해야만 한다.

연관된 모든 패킷에 대하여 특별한 NAT 처리가 필요한 경우도 있다. 다시금 FTP를 예로 들어보자. control connection의 PORT명령을 수행하는 클라이언트에 의해 얻어진 IP/port에 대해 DATA connection의 모든 입력 패킷은 정상적인 table lookup을 거치는 것보다는 반드시 NAT되어야만 한다.

- 연관된 connection을 유발하는 패킷에 대한 callback(foo_help)
- 연관된 모든 패킷에 대한 callback(foo_nat_expected)

4.4.7.2. 사용가능한 구조체와 함수

여러분이 작성한 nat helper 모듈의 'init()' 함수는 'struct ip_nat_helper'에 대한 포인터를 인자로 하여 'ip_nat_helper_register()' 함수를 호출한다. 인자가 되는 구조체는 다음과 같은 멤버 변수를 포함한다.

list

netfilter에서 내부적으로 사용하는 list 헤더로 { NULL, NULL }로 초기화시킨다.

tuple

'struct ip_conntrack_tuple'로, 우리가 작성한 NAT helper가 관심을 갖는 패킷을 명시한 것이다.

mask

'struct ip_conntrack_tuple'이며, tuple의 어느 비트가 유효한지를 netfilter에게 알려준다.

help

tuple+mask에 부합하는 각 패킷에 대하여 호출해야하는 함수이다.

name

NAT help로 구별이 되는 유일한 name

이상은 connection tracking helper를 작성하는 방법과 완전히 동일하다. 이제 여러분들이 작성한 모듈은 'ip_nat_register()' 함수를 이용하여 예측되는 임의의 connection의 NAT를 처리할 준비가 되어 있다고 말할 수 있다. 이때, 'ip_nat_register()' 함수는 'struct ip_nat_expect'를 인자로 취하게 되며, 그 멤버 변수는 다음과 같다.

list

netfilter에서 내부적으로 사용하는 list 헤더로 { NULL, NULL }로 초기화시킨다.

expect

예견된 connection에 대하여 NAT를 수행하는 함수이다. connection을 처리하면 true를 리턴하고, 다음에 등록된 expect 함수가 호출되어 패킷을 처리할 수 있는 지 검사하게 된다. true가 리턴된 경우, 이 함수는 반드시 판결(verdict)을 알려주어야 한다.

4.4.7.3. NAT 도우미 모듈 예제

```
#define FOO_PORT 111
static int foo_nat_expected(struct sk_buff **pksb,
unsigned int hooknum,
struct ip_conntrack *ct,
struct ip_nat_info *info,
struct ip_conntrack *master,
struct ip_nat_info *masterinfo,
unsigned int *verdict)
/* called whenever a related packet (as specified in the connec-
tion tracking
   module) arrives
   params: pksb packet buffer
hooknum HOOK the call comes from (POST_ROUTING, PRE_ROUTING)
ct information about this (the related) connection
info &ct->nat.info
master information about the master connection
masterinfo &master->nat.info
verdict what to do with the packet if we return 1.
{
        /* Check that this was from foo_expect, not ftp_expect, etc */
/* Then just change ip/port of the packet to the masqueraded
     values (read from master->tuplehash), to map it the same way,
           call ip_nat_setup_info, set *verdict, return 1. */
```

```
}
static int foo_help(struct ip_conntrack *ct,
struct ip_nat_info *info,
enum ip_conntrack_info ctinfo,
unsigned int hooknum,
struct sk_buff  **pksb)
/* called for the packet causing related packets
   params: ct information about tracked connection
info (STATE: related, new, established, ... )
hooknum HOOK the call comes from (POST_ROUTING, PRE_ROUTING)
pksb packet buffer
*/
{
/* extract information about future related packets (you can
   share information with the connection tracking's foo_help).
   Exchange address/port with masqueraded values, insert tuple
   about related packets */
}
static struct ip_nat_expect foo_expect = {
{ NULL, NULL },
foo_nat_expected };
static struct ip_nat_helper hlpr;
static int __init(void)
{
int ret;
if ((ret = ip_nat_expect_register(&foo_expect)) == 0) {
memset(&hlpr, 0, sizeof(struct ip_nat_helper));
hlpr.list = { NULL, NULL };
hlpr.tuple.dst.protonum = IPPROTO_TCP;
hlpr.tuple.dst.u.tcp.port = htons(FOO_PORT);
hlpr.mask.dst.protonum = 0xFFFF;
hlpr.mask.dst.u.tcp.port = 0xFFFF;
hlpr.help = foo_help;
ret = ip_nat_helper_register(hlpr);
if (ret != 0)
ip_nat_expect_unregister(&foo_expect);
}
return ret;
}
static void __exit(void)
{
ip_nat_expect_unregister(&foo_expect);
ip_nat_helper_unregister(&hlpr);
}
```

4.5. Netfilter의 이해

Netfiler는 상당히 단순하며, 앞 절에서 꽤 상세히 설명하였다. 그러나, 간혹 NAT나 ip_tables 하부 구조가 제공하는 것 이외의 것 또는 여러분들이 전부 바꾸고 싶은 것에 대하여 알아볼 필요가 있다.

미래의 이야기가 되겠지만, netfilter가 지향하고 있는 중요한 쟁점 중 하나는 캐슁이다. 각각의 skb는 'nfcache' 필드를 가지고 있으며, 이는 헤더의 어떤 필드를 검사하고 패킷을 변경할 것인지 말 것인지에 대한 비트 매스크이다. 각각의 훅이 그와 연관된 비트에 대한 netfilter의 OR를 0으로 만드는 것이 아이디어로, 이렇게 함으로써 패킷이 netfilter를 거쳐야 할 이유가 전혀 없다는 것을 알아차릴 수 있는 아주 훌륭한 캐쉬 시스템을 향후 작성할 수 있다.

가장 중요한 비트는 NFC_ALTERD와 NFC_UNKNOWN으로, NFC_ALTERED는 패킷이 변경되었다는 것을 의미하며 이 비트는 변경된 패킷을 다시 라우팅하기 위해 IPv4의 NF_IP_LOCAL_OUT 훅에 대하여 이미 적용되었다. NFC_UNKNOWN은, 표현할 수 없는 어떤 특성이 검출되어 캐슁이 수행되지 않았다는 것을 의미한다. 만일 의심이 가는 경우가 발생하면, 여러분의 훅 내부에 있는 skb의 nfcache 필드에 대해 NFC_UNKNOWN 플랙을 설정하기만 하면 된다.

4.6. 새로운 Netfilter 모듈 작성

4.6.1. Netfilter 훅에 연결하기

커널 내부에서 패킷을 줄이거나 조각내기 위해서는 'nf_register_hook' 함수와 'nf_unregister_hook' 함수를 사용하면 된다. 이들 각각은 다음과 같은 필드를 포함하는 'struct nf_hook_ops'에 대한 포인터를 인자로 취한다.

list

링크드 리스트로 '{ NULL, NULL }'로 설정된다.

hook

패킷이 이 훅 포인트에 걸리면 호출되는 함수로, NF_ACCEPT, NF_DROP 또는 NF_QUEUE 중 하나를 반드시 리턴 해야 한다. NF_ACCEPT인 경우는 현재의 포인터 다음에 오는 훅이 호출된다. NF_DROP인 경우는 패킷이 DROP되고, NF_QUEUE인 경우는 대기열로 들어 간다. skb 포인터에 대한 포인터를 돌려 받아서 원하는 경우에는 skb를 전부 바꾸어 버릴 수도 있다.

flush

현재는 사용하지 않고 있다. 캐쉬가 지워지는 경우 패킷 적중률을 전달하기 위해 설계되었으나, 전혀 구현될 일이 없을 것이다. 그냥 NULL로 설정하기 바란다.

pf

프로토콜 패밀리, 즉 IPv4에 대해서는 'PF_INET'이 된다.

hooknum

관심을 가지고 있는 훅의 수, 즉 'NF_IP_LOCAL_OUT'이다.

4.6.2. 큐된 패킷의 처리

ip_queue에 의해 사용되는 인터페이스로, 주어진 프로토콜에 대하여 대기된 패킷을 처리하기 위해 등록할 수 있다. 패킷을 처리하는 것을 방지할 수 있다는 것만 빼고는 훅을 등록하는 것과 유사한 의미를 가지며, 훅이 'NF_QUEUE'로 응답하는 패킷을 확인만 할 수 있다.

대기된 패킷에 대한 관심을 등록하기 위해 사용하는 두개의 함수는 'nf_register_queue_handle()'과 'nf_unregister_queue_handler()'이다. 여러분이 등록하고자 하는 함수는 'nf_register_queue_handler()' 함수로 전달되는 'void *' 포인터와 함께 호출된다.

프로토콜을 처리하기 위해 등록된 함수가 없는 경우는, NF_QUEUE를 리턴 하는 것은 NF_DROP를 리턴하는 것과 마찬가지가 된다.

대기된 패킷에 대한 관심을 등록했으면, 패킷이 큐잉 되기 시작한다. 이제 큐잉 된 패킷을 가지고 무엇을 하던 그건 여러분들의 맘이지만, 처리가 끝난 경우에는 반드시 'nf_reject()' 함수를 호출해야 한다(단순히 kfree_skb()를 호출해서는 안 된다). skb를 재 도입하는 경우는, 큐잉 된 패킷을 skb, 큐잉 된 패킷에 할당된 'struct nf_info'와 판결(verdict)을 전달한다. 그 이유는, NF_DROP은 패킷을 DROP시킬 것이고, NF_ACCEPT는 훅을 통해 계속 반복되도록 할 것이고, NF_QUEUE는 패킷을 다시 대기 시킬 것이고 NF_REPEAT는 패킷을 대기시킨 훅이 또 다시 검사하도록 만들 것이기 때문이다(이 때, 무한루프에 빠지지 않도록 조심할 것).

'struct nf_info'를 살펴보면, 패킷에 대한 부가적인 정보, 즉 패킷이 존재했던 인터페이스와 훅 같은 것을 얻을 수 있다.

4.6.3. 사용자 공간으로부터 명령어 전달받기

Netfilter의 구성요소들이 사용자공간과 상호작용을 필요로 한다는 것은 아주 당연한 일이다. 방법은 setsockopt 메커니즘을 사용하여 이런 작용을 구현할 수 있다. 여기서 주의할 점은 각 프로토콜이 이해하지 못하는 setsockopt 넘버에 대해 Nf_setsockop()를 호출할 수 있도록 각 프로토콜이 수정되어야만하며, 이는 IPv4까지 이고, IPv6와 DECnet은 이미 변경되어 있다.

최근에 알려진 기술을 사용하면, nf_register_sockopt() 함수를 사용하여 'struct nf_sockopt_ops'를 등록하며, 이 구조체는 다음과 같은 필드로 구성되어 있다.

list

링크드 리스트를 사용하기 위한 것으로, '{ NULL, NULL }'로 설정된다.

get_optmin
get_optmax

처리해야 할 getsockopt의 개수의 범위를 지정한다. 즉 0과 0을 사용하면 처리해야 할 getsockopt의 개수가 없다는 것을 의미한다.

get

사용자들이 여러분들이 작성한 getsockopt 중 하나를 호출한 경우 호출되는 함수이다. 이 함수 내부에서 사용자들이 NET_ADMIN의 권한을 가지고 있는 지 확인해야 한다.

나머지 두개의 필드는 내부적으로 사용된다.

4.7. 사용자 공간에서 패킷 처리

libipq 라이브러리와 'ip_queue' 모듈을 사용하면, 커널에서 할 수 있는 대부분의 것들을 사용자 공간에서 수행할 수 있다. 이것이 다음과 같은 것을 의미한다. 속도에 대한 문제가 발생한다면, 사용자 공간에서 완전히 여러분들만의 코드를 개발할 수 있다. 개발하고자 하는 여러분들이 큰 대역을 필터링 하고자 하지만 않는다면, 커널 내부의 패킷 맹글링에 비해 이 방법이 월등하다는 것을 알게 될 것이다.

netfilter 초창기에, 필자는 iptables의 초기 버전을 포팅 하여 이를 증명하였다. netfilter는, 개발자들이 원하는 언어가 무엇이던 간에, 개발자 자신의 코드와 고효율의 넷맹글링 모듈을 개발하고자 하는 사람들에게 오픈 되어 있다.

5. 커널 2.0/2.2 패킷 필터 모듈 변환

포팅을 아주 쉽게 할 수 있는 단순한 계층에 대해서는 ip_fw_compat.c를 보기 바란다.

6. 터널 코드 개발자를 위한 Netfilter 훅

Tunnel 드라이버 제작자들은 2.4 커널에 대해서는 다음에 보이는 단순한 두개의 규칙을 따르기 바란다.

- 패킷을 인식할 수 없도록 하려면 skb->nfct를 릴리즈 해야한다(즉, decapsulating/encapsulating). 만일 패킷을 *new* skb로 감싸지 않으려면 릴리즈를 하지 않아도 되지만, 적절한 곳에서 수행하고자 한다면, 반드시 릴리즈 해야 한다.

  그렇지 않은 경우는, NAT 코드는 패킷을 맹글하기 위해 과거의 connection tracking 정보를 이용할 것이고, 이로 인해 순서가 엉망이 될 것이다.

- encapsulated 패킷은 반드시 LOCAL_OUT 훅을 통과하도록 해야하며, decapsulated 패킷은 PRE_ROUTING 훅을 통과해야만 한다. 이를 수행하기 위해 대부분의 tunnel은 ip_rcv()를 사용한다.

  그렇지 않은 경우는, 사용자들이 tunnel을 이용하여 원하는 대로 필터링할 수 없을 것이다.

첫 번째를 수행하는 표준 방법은 패킷을 wrap하거나 unwrap하기 전에 다음과 유사한 코드를 삽입하는 것이다.

```
/* Tell the netfilter framework that this packet is not the
          same as the one before! */
#ifdef CONFIG_NETFILTER
nf_conntrack_put(skb->nfct);
skb->nfct = NULL;
#ifdef CONFIG_NETFILTER_DEBUG
skb->nf_debug = 0;
#endif
#endif
```

두 번째를 수행하기 위해 필요로 하는 것은 새로이 encapsulated된 패킷이 “ip_send()”로 들어가는 위치를 찾아내고, 이를 다음과 같은 것으로 대체하는 것이다.

```
/* Send "new" packet from local host */
NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL, rt->u.dst.dev, ip_send);
```

다음에 보인 룰이 의미하는 것은 tunnel 박스에 패킷 필터링 룰을 적용하고자 하는 사람이 tunnel 되고 있는 패킷에 대해 다음과 같은 절차를 보게 될 것이다.

1. FORWARD hook : normal packet (from eth0 -> tunl0)
2. LOCAL_OUT hook : encapsulated packet (to eth1)

응답 패킷에 대해서는

1. LOCAL_IN hook: encapsulated reply packet (from eth1)
2. FORWARD hook: reply packet (from eth1 -> eth0).

7. 시험도구(Test Suite)

test suite는 CVS 저장소에 있으며, test suite가 적용되는 범위가 넓을 수록, 여러분의 코드가 오동작을 하는 경우가 거의 없을 수 있다는 자신감은 더 커지게 된다. 사소한 테스트(trivial test)는 상당히 어려운 테스트(tricky test) 만큼이나 중요하며, 이는 복잡한 시험을 단순화한 것이 사소한 테스트이기 때문이다. 일단 복잡한 테스트를 수행하기 전에 기본적인 작업은 만족한다는 것을 알 수 있기 때문이다.

테스트는 단순하며, 그저 testsuite/의 서브디렉토리에 있는 쉘스크립트이다. 실행되는 순서는 알파벳 순서다. 따라서 ‘01test’가 ‘02test’보다 먼저 실행된다. 현재는 다섯 개의 테스트 디렉토리가 있다.

00netfilter/

General netfilter framework tests.

01iptables/

iptables tests.

02conntrack/

connection tracking test.

03NAT/

NAT tests.

04ipchains-compat/

ipchains/ipfwadm compatibility tests.

testsuite/ 디렉토리 내부에는 'test.sh'라는 쉘 스크립트가 존재하며, 두개의 더미 인터페이스(tap0와 tap1)를 구성하여, forwarding을 on 시키고, 모든 netfilter 모듈을 제거하는 역할을 한다. 그리고 나서 하위 디렉토리에 존재하는 모든 쉘 스크립트를 수행하는 데 이중 하나라도 오류가 발생하면 수행을 멈추게 된다. 'test.sh'는 두개의 옵션을 넘겨받을 수 있는 데, '-v'는 수행하는 각각의 테스트를 출력하며, 추가적인 test 이름이 주어지면 해당하는 테스트가 발견될 때까지 다른 테스트는 모두 건너뛰게 된다.

7.1. 테스트를 위한 스크립트 작성

우선 적당한 디렉토리에 새로운 파일을 생성하고, 적절한 시기에 실행할 수 있도록 번호를 부여한다. 예를 들어보면, ICMP reply tracking (02conntrack/02reply.sh)을 테스트하기 위해서는 외부로 나가는 ICMP가 적절하게 추적되고 있는 지를 먼저 확인해야한다(02conntrack/01simple.sh).

작은 크기의 파일을 많이 만들어서, 각각의 파일은 한 가지 분야만 담당하도록 하는 것이 좋을 것이며, 그 이유는 testsuite를 실행시킨 사람들이 문제를 즉각 알아 낼 수 있도록 할 수 있기 때문이다.

테스트 도중 문제가 발생하면, 그냥 'exit 1'을 하면 되고, 그러면 failure가 발생한다. 즉, 여러분들이 문제가 발생한 것을 감지할 수 있게 되면, 적절한 메시지를 출력하는 것이 좋다. 문제가 발생하지 않았다면, 'exit 0'로 종료하면 된다. 스크립트의 최상위에서 'set -e'를 사용하거나, 각 명령의 마지막에 '|| exit 1'을 추가하여 각 명령의 성공여부를 확인해야 한다.

helper 함수인 'load_module'와 'remove_module'는 모듈을 올리거나 내리는 데 사용할 수 있는 데, 특별히 테스트라고 지정하지 않는 한 testsuite에서는 autoloading을 지원하지 않는다.

7.2. 변수와 환경

일단 tap0와 tap1이라는 두개의 인터페이스를 가지고 있다. 각 인터페이스의 주소는 $TAP0와 $TAP1이라는 변수에 저장되어 있으며, 넷매스크는 모두 255.255.255.0이다. network은 $TAP0NET과 $TAP1NET에 저장되어 있다.

$TMPFILE은 임시파일로, 테스트가 종료되는 시점에서 삭제된다.

여러분의 스크립트는 testsuite/ 디렉토리부터 시작해서, 스크립트가 존재하는 모든 디렉토리를 찾아가게 된다. 즉 여러분들은 iptables같은 도구를 실행시키기 위해서는 '../userspace'로 시작하는 path를 사용해야 한다.

$VERBOSE를 설정해 놓으면 여러분의 스크립트는 보다 많은 정보를 출력하게된다. 즉, command line에서 '-v' 옵션을 준 것과 동일한 효과가 된다.

7.3. 유용한 도구들

"tools" 디렉토리에는 쓸만한 testsuite 도구가 몇 개 있으며, 각 도구는 문제가 발생한 경우 non-zero exit status를 발생시키며 종료한다.

7.3.1. gen_ip

IP 패킷을 생성하는 함수로, IP 패킷을 표준 입력으로 출력한다. 여러분들은 표준 출력을 /dev/tap0와 /dev/tap1으로 전송하여 tap0와 tap1에 패킷을 보낼 수 있다. (/dev/tap0와 /dev/tap1이 존재하지 않으면 testsuite가 최초 실행 될 때 생성된다.)

FRAG=offset,length

패킷을 생성하고, offset과 length 만큼 패킷을 조각 낸다.

MF

패킷의 'More Fragments' 비트를 설정한다.

MAC=xx:xx:xx:xx:xx:xx

패킷에 소스 MAC 주소를 설정한다.

TOS=tos

패킷에 대하여 TOS 필드를 설정한다. 0-255.

다음은 강제 옵션이다.

source ip

패킷의 소스 IP 주소

dest ip

패킷의 목적지 주소

length

헤더를 포함한 패킷의 길이

protocol

패킷의 프로토콜 번호, (예: 17 = UDP)

각 전달인자는 프로토콜에 따라 달라진다. UDP(17)인 경우 소스와 목적지 포트 번호가 된다. ICMP(1)인 경우는 ICMP 메시지의 코드가 된다. 타입이 0 또는 8(ping-reply 또는 ping)인 경우는, 두개의 인자로 ID와 sequence 필드가 더 필요하다. TCP인 경우는 소스와 목적지 포트 그리고 플랙(“SYN”, “SYN/ACK”, “ACK”, “RST” 또는 “FIN”)이 필요하다. 세개의 추가 옵션이 있는 데, “OPT=” 다음에는 컴마로 분리된 옵션들이 오고, “SYN=” 다음에는 시퀀스 번호, “ACK=” 다음에도 시퀀스 번호가 호다. 마지막으로, “DATA=”이라는 추가 옵션은 표준입력의 내용을 가득 채운 TCP 패킷의 부하를 의미한다.

7.3.2. rcv_ip

‘rcv_ip’를 사용하면 IP 패킷을 볼 수 있으며, 가능한 get_ip로 들어오는 원래의 값에 가깝게 명령 행에 출력한다.(fragments는 제외)

이 함수는 패킷을 분석하는 데 매우 유용하며, 다음과 같은 강제 옵션이 두개 있다.

wait time

표준입력으로부터 패킷을 기다리는 최대 시간으로 초단위로 표시된다.

iterations

수신하고자 하는 패킷의 수

“DATA”라는 추가 옵션이 있으며, 패킷 헤더 다음에 패킷의 부하를 표준출력으로 출력한다.

다음 스크립트는 ‘rcv_ip’를 사용하는 표준을 보인 것이다.

```
# Set up job control, so we can use & in shell scripts.
set -m
# Wait two seconds for one packet from tap0
../tools/rcv_ip 2 1 < /dev/tap0 > $TMPFILE &
# Make sure that rcv_ip has started running.
sleep 1
# Send a ping packet
../tools/gen_ip $TAP1NET.2 $TAP0NET.2 100 1 8 0 55 57 > /dev/tap1 || exit 1
# Wait for rcv_ip,
if wait %../tools/rcv_ip; then :
else
    echo rcv_ip failed:
    cat $TMPFILE
    exit 1
fi
```

7.3.3. get_err

표준입력으로부터 패킷(예를 들면, gen_ip로 생성된 패킷)을 취하고, ICMP 에러로 변환시킨다.

세개의 전달인자로 소스 IP 주소, 타입과 코드를 받는다. 목적지 입력은 표준입력으로부터 받은 소스 IP 주소로 설정된다.

7.3.4. local_ip

표준입력으로부터 패킷을 얻어서 raw 소켓으로 전달한다. 로컬에서 생성된 패킷의 모양을 보여준다(ethertap 장치 중 하나로 입력된 패킷으로부터 분리하는 경우, 외부에서 생성된 패킷처럼 보인다).

7.4. 생각나는 대로 하는 충고

모든 도구는 읽고 쓰기를 한번 할 때 모든 것을 할 수 있다고 가정한다. 즉 ethertap 장치에 대해서는 사실이지만, pipe를 이용하여 약간 복잡한 작업을 할 때는 반드시 옳다고 말할 수는 없다.

패킷을 자르기 위해서 dd를 사용할 수 있다. dd는 일회성 작업으로 패킷을 출력하기 위해 사용할 수 있는 obs(output block size)라는 옵션이 있다.

첫 번째에 성공하도록 테스트하는 것이 좋다. 즉, 패킷이 성공적으로 블럭 되도록 테스트해야 한다. 처음에는 패킷이 정상적으로 통과하도록 테스트를 하고 나서, 일부 패킷이 블럭 되도록 테스트해야 한다. 그렇지 않으면, 엉뚱한 오류로 인해 패킷이 진행하는 것을 중지시킬 수 도 있다.

아무거나 막 보내고 나서 어떤 결과가 일어나는 확인하지 말고, 정확한 테스트를 할 수 있도록 해야한다. 정확한 테스트가 잘 못된 경우라면, 무엇이 잘 못 되었는지 알 수 있지만, 랜덤 테스트가 잘못 되면 전혀 도움이 되지 않는다.

아무런 메시지도 남기지 않고 테스트가 실패한 경우, 어떤 명령이 실행되고 있는지 알아보기 위해 스크립트의 맨 위에 '-x'를 추가 할 수 있다. 즉, '#!/bin/sh -x'.

테스트가 불규칙적으로 실패하면, 모든 외부 장치를 차단하거나 하는 식으로, 랜덤 네트웍 트래픽에 대하여 확인해 보는 것이 좋다. 예를 들면,Andrew Tridgell처럼 동일한 네트웍 상에 있으면, 윈도우의 브로드 캐스팅에 의해 수많은 간섭을 받게 된다.

8. 개발 동기

필자가 ipchains를 개발하고 나서, 시드니의 중국식당 입구에 들어서려는 순간 불현듯 머리는 스치는 생각에 패킷 필터링이 엉뚱한 곳에서 이루어 지고 있다는 사실을 깨달았다. 지금은 어디인지 찾을 수 없지만, 한가지 기억하는 것은 Alan Cox에게 메일을 보냈고, 친절하게도 그

는 '처음엔 아무리 당신이 옳았다고 할지라도 한일을 끝내는 것이 어떤가요?'라는 답장을 보냈다. 이 짧은 말 한마다에서 실용주의가 정도를 이겨버렸다.

최초에는 ipfwadm의 커널 파트의 마이너 수정이었지만 결국에는 많은 부분을 다시 작성하여 ipchains를 마무리짓고 HOWTO를 쓰고 나서야, 패킷 필터링, 매스커레이딩, 포트포워딩과 유상한 여러 이슈에 대하여 리눅스 커뮤니티에서 많은 혼란이 있었다는 것을 알게 되었다.

이런 것들이 바로 여러분들의 도움으로부터 얻는 즐거움이다. 여러분들은 사용자들이 무엇을 하려고 하고, 또 무엇을 가지고 논쟁하는 지 보다 가까이서 느낄 수 있다. 프리소프트웨어라는 것은 많은 사용자들이 사용하고 있을 때 가치가 있고, 보다 쉽게 만들 수 있다는 것을 의미한다. 문서가 아닌 구조(architecture)가 최대 결함이었다.

그래서 필자는 ipchains 코드를 가지고 사람들이 하려고 했던 아이디어와 경험을 얻었다. 문제는 단지 둘 뿐이었다.

첫 번째로, 필자는 보안 쪽으로 다시 돌아가고 싶지 않았다. 보안 컨설턴트가 된다는 것은 여러분의 양심과 부의 사이에 있는 영원한 도덕적인 전쟁이다. 기본적인 수준에서, 여러분들은 실제 보안이 아닌 보안에 대한 분위기를 팔고 있는 것이다. 보안을 이해하고 있는 군대에서 군사용으로 작업을 하더라고, 다를 바는 없다.

두 번째 문제는 새로운 사용자들은 관심 밖의 문제였다. 큰 회사와 ISP들이 앞다퉈 이 물건을 사용하고 있다. 내가 필요한 것은, 물론 그들이 내일의 홈유저의 규모가 될 수도 있지만, 이런류의 사용자들 부터의 신뢰를 받는 것이었다.

WatchGuard에 대한 노출이 내게 필요한 대규모의 클라이언트의 노출을 가져다 주었고, 그들로부터 독립하고자 하는 것이 모든 사용자들에게 동일하게 지원되도록 하였다.

그래서 필자는 쉽게 작성된 netfilter를 가질 수 있었고, 최상위에 ipchains를 포팅 하여, 이를 가지고 모든 것을 해냈다. 불행하게도 커널에서 모든 매스커레이딩 코드가 사라져 버렸다. 즉, 필터링으로부터 매스커레이딩을 독립시키고자 하는 것이 패킷 필터링으로 이동하게된 주된 이유이지만, 그렇게 함으로써 매스커레이딩 역시 netfilter 프레임 웍으로 이동할 필요가 있었다.

ipfwadm의 'interface-address'(ipchains에서 필자가 삭제한 것 중 하나)의 특성에 대한 나의 경험으로 배운 것은, 단순히 매스커레이딩 코드를 분리하고 나를 대신하여 netfilter에 포팅해 줄 사람을 기다리는 것은 아무런 희망이 없다는 것이다.

그래서 필자는 최소한 현재 코드만큼 많은 특성을 확보할 필요가 있었다. 초기에 도입한 niche users를 북돋기 위해, 정확히 말하면 조금 더 필요했다. 이는 투명한 프락시, 매스커레이딩과 포트 포워딩을 대체한다는 것을 의미한다. 바꿔 말하면, 완벽한 NAT 계층이 되는 것이다.

일반적인 NAT 시스템을 작성하는 대신, 기존의 매스커레이딩 계층을 포팅하기로 결정했었지만, 매스커레이딩은 연륜과 유지보수의 부재가 문제시되었다. 그러니까 매스커레이딩 메인터너가 없다는 이야기다. 순수한 사용자들은 매스커레이딩을 사용하지 않고, 유지보수 작업을 하려고 하는 홈 유저들도 많지 않다. Juan Ciarlante 같이 용감한 사람들이 버그를 고쳤지만, 결국 다시 작성해야 하는 수준에 이르게 되었다.

필자는 NAT를 다시 작성한 사람이 아니라는 것을 알아주기 바란다. 즉 필자는 더 이상 매스커레이딩을 사용하지 않고, 당시 기존의 코드를 살펴보지도 않았다. 아마도 이것이 내가 생각한 것보다 많은 시간이 소요된 이유일 것이다. 하지만 필자의 견해로는 결과는 꽤 좋았고, 많

은 것을 배울 수 있었다. 얼마나 많은 사람들이 사용하고 있는 지 알게 되면, 두 번째 버전은 훨씬 더 좋아 질 것이라는 점은 믿어 의심치 않는다.

9. 감사의 말

도와주신 모든 분들에게 감사 드리며, 특히 protocol helpers를 작성해 주신 Harald Welte에게 감사 드립니다.