UNIVERSIDADE TECNOLOGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA – DAINF
CURSO DE TECNOLOGIA EM INFORMÁTICA

PROJETO DE REDES
www.projetoderedes.com.br

TIAGO SALEM HERRMANN

RNAT: ROTEAMENTO ATRAVÉS DE NAT

CURITIBA – PR
2006

TIAGO SALEM HERRMANN

RNAT: ROTEAMENTO ATRAVÉS DE NAT

Trabalho de Diplomação apresentado à Universidade Tecnológica Federal do Paraná como requisito para obtenção do título de Tecnólogo em Informática.

Orientadora: Profa. MSc. Ana Cristina Kochem Vendramin.

CURITIBA
2006

AGRADECIMENTOS

Agradeço a todas as pessoas que de alguma forma estiveram envolvidas no desenvolvimento do presente trabalho, em especial aos amigos Tiago Almeida Barboza de Souza e Aristeu Sergio Rozanski Filho pelos conselhos e apoio. Agradeço também a professora Ana Cristina Kochem Vendramin pela orientação, apoio, conselhos e correções do trabalho. Por fim, agradeço ao Movimento *Open Source*, por permitir que trabalhos como este sejam possíveis.

“Progresso é a realização de utopias.”

Oscar Wilde

RESUMO

A Internet, com o passar dos tempos, se modificou e se adequou aos novos padrões tecnológicos, principalmente para permitir que pequenas e médias redes privadas pudessem se comunicar na Internet. A técnica de *Masquerading,* hoje presente em várias redes do mundo, permite que *hosts* com endereços IP reservados para redes privadas acessem a Internet, porém não permite que dois *hosts* que passam pelo processo de mascaramento possam se comunicar diretamente. O presente trabalho propõe um novo conceito de roteamento, chamado RNAT, que permite o acesso direto entre estes *hosts.*

Palavras-chave: NAT, roteamento, Connection Tracking, IPv4, redes privadas.

ABSTRACT

Since the beginning of the Internet, it has changed to aggregate the new standards, mainly to allow the Internet access to small and medium private networks. The Masquerading feature, common nowadays, has allowed the private networks to make Internet access, but it does not allow direct connection between two hosts located behind the mascaration process. This project intends to suggest a new routing concept, named RNAT, which allows the direct access between those *hosts.*

Keywords: NAT, routing, Connection Tracking, IPv4, Private Networks.

Sumário

1 INTRODUÇÃO 1
1.1 Objetivos 1
1.2 Trabalhos Relacionados 2
1.2.1 IPv4+4 2
1.2.2 IPNL 3
1.2.3 Extended IP 3
1.3 Organização do Documento 3
2 IPv4 4
2.1 IP válidos x IP inválidos 6
3 NAT 7
4 INFORMAÇÕES TÉCNICAS 11
4.1 Capacidades do Kernel do Sistema Operacional 11
4.2 IP Options 12
4.3 Netfilter 15
4.4 Source Route 16
4.5 DNS 17
5 IMPLEMENTAÇÃO DO RNAT (Routable NAT) 19
5.1 Opção RNAT 19
5.2 Protocolo RNAT 20
5.3 Módulo rnat_router 24
5.4 Resource Record RNAT 25
5.5 Biblioteca de funções RNAT 26
5.6 Informações importantes sobre o RNAT 27
6 FUNCIONAMENTO DO RNAT 29
6.1 RNAT em roteadores 29
6.2 RNAT em hosts 30
6.2.1 Kernel 30
6.2.2 OpenSSH 31
6.2.3 Putty 31
6.2.4 Ping 32
TRABALHOS FUTUROS 34
APÊNDICE A 35
APÊNDICE B 37
APÊNDICE C 47
APÊNDICE D 49
APÊNDICE E 51
APÊNDICE F 57
REFERÊNCIAS 63

Lista de Figuras

Figura 1 – Representação de um Pacote IPv4..... 4
Figura 2 – Processo de mascaramento ao enviar um pacote..... 8
Figura 3 – Processo de mascaramento ao receber pacote..... 8
Figura 4 - Estrutura do campo IP Options..... 13
Figura 5 – Utilização do campo IP Options com o comando ping..... 14
Figura 6 – Disposição das Hooks do Netfilter no kernel do Linux..... 16
Figura 7 – Estrutura da opção RNAT no campo IP Options..... 20
Figura 8 – Redes privadas com acesso à Internet..... 21
Figura 9 – Opção RNAT ao sair do host de origem..... 23
Figura 10 – Opção RNAT ao ser enviada para a Internet..... 23
Figura 11 – Opção RNAT ao ser enviado para o host destino..... 24
Figura 12 – Software putty com suporte a RNAT..... 32
Figura A1 - Tempo de resposta para um host com IP válido..... 36
Figura A2 - Tempo de resposta para um host com IP inválido de outra intranet..... 36

1 INTRODUÇÃO

Assim como qualquer outra tecnologia, a Internet que conhecemos hoje é resultado de uma evolução. Conforme as topologias de redes foram se moldando e o número de *hosts* conectados à grande rede foi aumentando, a Internet foi se modificando, principalmente para suprir a escassez de endereços IP (*Internet Protocol*) disponíveis, elemento que é essencial para comunicação entre *hosts.* Para amenizar esta escassez, novas técnicas foram surgindo, porém, sempre apresentando problemas quanto a compatibilidade e funcionalidades já existentes, como é o caso do *Masquerading*, uma técnica que permite que *hosts* com IP's inválidos acessem a Internet através da troca de endereços feita no roteador da rede (RFC 1631, 1994).

No estado atual, ainda existem restrições na conectividade direta entre dois *hosts* situados em redes privadas diferentes e que passam por um processo de mascaramento para acesso à Internet, como por exemplo em aplicações VoIP (*Voice over Internet Protocol*) e na transferência de arquivos.

Mesmo levando em consideração a existência de técnicas para fazer esta comunicação, nenhuma eliminou os problemas de o emissor poder escolher o *host* da outra rede privada que ele deseja acessar, o que em teoria faria com que os IP's, até então considerados inválidos, se tornassem IP's válidos, resolvendo em partes os problemas da escassez de IP's na Internet. Outro problema gerado pelo mascaramento é a falta de visibilidade, que não permite ao destinatário saber qual é o verdadeiro *host* que o está acessando.

1.1 Objetivos

O objetivo deste trabalho é apresentar um novo padrão, assim como o software necessário, para comunicação em redes IPv4, onde cada *host* terá uma identificação universal (composta pelo IP válido na Internet do seu *gateway* e pelo seu IP inválido) e poderá se comunicar com qualquer outro *host* na Internet que também tenha suporte a este padrão, resolvendo desta forma o problema da falta de

visibilidade causado pelo mascaramento.

O novo padrão será compatível com os padrões IPv4 existentes, será de fácil implementação, e fará uso de um novo registro DNS (*Domain Name System*) para se referenciar dois IP's ao invés de um, que são relativos ao IP externo e ao IP da rede privada do *host* que se deseja acessar. Serão fornecidos o código fonte para os roteadores com plataforma Linux implementarem as alterações necessárias, uma biblioteca de funções para fácil migração dos softwares clientes e modificações para alguns *softwares* de modo a demonstrar o funcionamento e o uso da biblioteca fornecida.

Para o presente trabalho o sistema operacional base será o GNU/Linux, devido à sua flexibilidade e por ser software livre, porém o padrão sugerido poderá ser aplicado em qualquer sistema operacional.

1.2 Trabalhos Relacionados

1.2.1 IPv4+4

O IPv4+4 sugere uma extensão do endereçamento de redes IPv4 baseadas no conceito de domínios (*realms)* públicos e privados. Os pacotes IPv4+4 possuem um cabeçalho a mais entre a camada de rede e a de transporte, indicando, entre outros campos, os endereços do domínio privado. O endereço do domínio público continua sendo referenciado no cabeçalho IPv4. Roteadores na Internet que não conheçam IPv4+4, considerariam o novo cabeçalho como parte do cabeçalho TCP (*Transmission Control Protocol*).

Como resultado, *hosts* origem, destino e roteadores que delimitam os dois domínios privados que estão se comunicando necessitam analisar os pacotes em níveis além da camada de rede. Roteadores necessitam desta análise para descobrir o endereço do *host* destino. Os *hosts* destino e origem necessitam desta análise para remover este cabeçalho antes de enviá-lo para a camada de transporte, evitando desta forma, por exemplo, a alteração da implementação TCP e UDP (TURÁNYI, VALKÓ, 2003).

1.2.2 IPNL

O IPNL, ou *NAT-Extended Internet Architecture*, assim como o IPv4+4, também trabalha com o conceito de domínios, porém o protocolo prevê a utilização de vários domínios privados, ao invés de um, aumentando o endereçamento disponível. Similarmente ao IPv4+4, um novo cabeçalho é inserido entre a camada de rede e a de transporte, servindo para indicar, entre outras informações, o IP e o domínio privado de destino. Para o uso do IPNL são necessárias modificações nos *hosts* e nos roteadores que limitam os domínios privados que estão se comunicando (FRANCIS, GUMMADI, 2001).

1.2.3 Extended IP

O *Extended* IP, ou EIP, é uma especificação criada em 1992 que propõe a extensão do endereçamento IP através de uma IP *Option* específica. Esta é uma proposta muito semelhante a apresentada no presente trabalho, porém sugere que a opção transporte mais informações além dos IP's de origem e destino das redes privadas. Sua implementação não foi encontrada (RFC 1385, 1992).

1.3 Organização do Documento

Este trabalho está dividido em cinco capítulos, sendo este utilizado para mostrar os objetivos, trabalhos relacionados e as motivações para o desenvolvimento do mesmo. O segundo capítulo faz uma descrição do IPv4 e demonstra as diferenças entre IP válido e inválido. O terceiro capítulo apresenta o NAT e sua importância na Internet atualmente. O quarto capítulo explica vários conceitos necessários para o desenvolvimento do RNAT, como *kernel capabilities* e o campo IP *Options* do cabeçalho IPv4. O quinto capítulo explica o desenvolvimento do RNAT propriamente dito, tanto o protocolo quanto os softwares necessários para seu funcionamento. Por fim, serão apresentadas demonstrações práticas da utilização do conceito RNAT no sexto capítulo e as conclusões obtidas.

2 IPv4

O IPv4 (RFC 791, 1981), ou IP (*Internet Protocol*) versão 4, é um protocolo utilizado para comunicação entre *hosts* tanto diretamente como inter-redes. Localizado na camada de rede do modelo OSI (*Open Systems Interconnection)*, é atualmente o protocolo mais utilizado para comunicação, principalmente por suportar roteamento de pacotes. Desenvolvido por Vint Cerf e Robert Kahn e adotado em 1 de janeiro de 1983 juntamente com o TCP, o IPv4 substituiu o antigo NCP (*Network Control Protocol*) (RFC 801, 1981), tornando-se até hoje o protocolo padrão para a Internet.

Dentre outras características, o IPv4 é um protocolo sem conexão (*connectionless*) e com suporte a fragmentação de pacotes, permitindo desta forma o seu transporte por mídias com MTU (*Maximum Transmission Unit*) pequenas. A Figura 1 mostra uma representação de um pacote IPv4 (RFC 791, 1981).

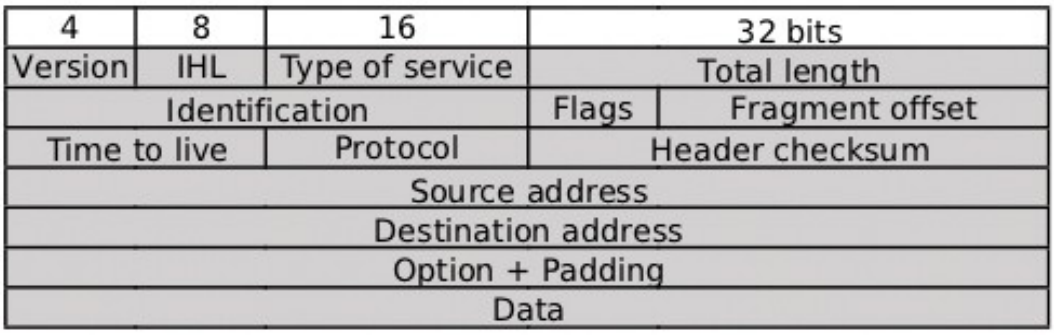

Figura 1 – Representação de um Pacote IPv4

Os seguinte campos fazem parte de um pacote IPv4 (RFC 791, 1981):

- ***Version*** (4 bits): Contém a versão do protocolo, neste caso 4;
- **IHL** (*IP Header Length* – 4 bits): Número de palavras de 4 bytes que o cabeçalho contém. O número mínimo é 5, o que indica um cabeçalho de 20 bytes e sem campo *Option*;
- **TOS** (*Type of Service* – 8 bits): Utilizado para fazer QoS (*Quality of Service*);
- ***Total length*** (16 bits): Tamanho total do pacote (cabeçalho e dados);
- ***Identification*** (16 bits): Utilizado para identificar os fragmentos de um

mesmo pacote;

- ***Flags*** (3 bits): Utilizado para controlar ou identificar fragmentos;
- ***Fragment offset*** (13 bits): Informa a localização do fragmento no datagrama IP original;
- **TTL** (*Time to live* - 8 bits): Utilizado para evitar rotas circulares. A cada roteador que o pacote passa, este número é decrementado, e ao chegar em zero o pacote é descartado;
- ***Protocol*** (8 bits): Número identificando o protocolo que o pacote carrega em sua área de dados. (1 para ICMP (*Internet Control Message Protocol*), 6 para TCP (*Transmission Control Protocol*), 17 para UDP (*User Datagram Protocol*) , etc);
- ***Header Checksum*** (16 bits): Soma de valores de todos os bytes do cabeçalho IP em palavras de 16 bits. A soma deve ser feita com o próprio campo *checksum* zerado e o resultado final deve ser o complemento de um da soma efetuada. A finalidade deste campo é checar inconsistências no cabeçalho do pacote;
- ***Source Address*** (32 bits): Endereço de origem do pacote;
- ***Destination Address*** (32 bits): Endereço do destinatário do pacote;
- ***Option+Padding*** (tamanho variável): Campo utilizado para opções customizadas. Somente existe caso o campo IHL seja maior do que 5. No caso da existência do campo, obrigatoriamente deverá acabar com a opção EOOL (*End-Of-Option-List*) e espaços não utilizados deverão ser preenchidos com zeros.

Exemplos de opções são:

  - *Record Route:* utilizado para armazenar os endereços de roteadores pelo qual o pacote passou para chegar ao destino e voltar à origem;
  - *Strict Source and Record Route:* opção que informa os roteadores pelos quais o pacote deve ser enviado para chegar ao destino.

- ***Data*** (tamanho variável): Área destinada aos dados propriamente ditos, vindos de protocolos das camadas superiores (TCP, UDP, etc).

2.1 IP válidos x IP inválidos

Utilizando o protocolo IPv4 é possível endereçar matematicamente até 4.294.967.296 (quatro bilhões, duzentos e noventa e quatro milhões, novecentos e sessenta e sete mil, duzentos e noventa e seis) endereços. Este número é obtido através do cálculo 256^4 (pois cada endereço IPv4 é formado por 4 octetos, ou 32 bits).

Nem todos estes endereços são utilizados na Internet, pois existe uma série deles reservados para utilização em redes privadas, *multicast*, além de vários outros escolhidos somente para uma utilização futura.

Os IP's reservados para uso em redes privadas ficaram popularmente conhecidos como IP's inválidos, pelo fato de não ser possível trafegar diretamente na Internet com estes endereços. A técnica de *Masquerading* (ver Capítulo 3) possibilita até hoje que estes IP's inválidos possam acessar a Internet.

Os endereços que realmente podem se comunicar na Internet ficaram popularmente conhecidos como IP's válidos.

Estas terminologias não são tecnicamente corretas, pois IP's inválidos são endereços que não podem ser representados com um número de 32 bits, o que não reflete a realidade dos endereços IP reservados para uso em redes privadas. Apesar disto, estas terminologias serão utilizadas em alguns capítulos no presente trabalho por serem amplamente utilizadas e facilitarem o entendimento do propósito do mesmo.

Os endereços reservados para redes privadas são mostrados na Tabela 1.

Endereço de Rede	Endereço *Broadcast*	Prefixo
10.0.0.0	10.255.255.255	8
172.16.0.0	172.31.255.255	12
192.168.0.0	192.168.255.255	16

Tabela 1 – Endereços reservados para uso em redes privadas. (RFC 1918, 1996)

3 NAT

No início da Internet cada *host* possuía um número IP próprio, porém com o aumento do número de *hosts* e com a necessidade de conectar redes privadas que utilizavam IP's inválidos, uma nova técnica começou a ser usada.

O NAT (*Network Address Translator*) (RFC 1631, 1994) é uma técnica que se baseia na troca de endereços de origem ou destino de um pacote IP. É possível dividir o NAT em basicamente 2 tipos: DNAT (*Destination NAT*) e SNAT (*Source NAT*). A principal diferença diz respeito a qual endereço será trocado. No DNAT, o endereço do destinatário é alterado, já no SNAT quem é alterado é o endereço da origem.

O NAT permitiu o desenvolvimento de um tipo de SNAT chamado *Masquerading,* que, além de trocar o IP de origem dos pacotes vindos de uma rede interna com destino à Internet por um endereço válido, também mantém referência das conexões ativas utilizando peculiaridades dos protocolos das camadas superiores, como por exemplo a porta de origem em protocolos TCP e UDP. Esta referência serve basicamente para identificar o destino do pacote na rede privada quando este retorna em resposta a outro previamente enviado.

É possível verificar o funcionamento do *Masquerading* na comunicação exemplificada nas Figuras 2 e 3.

Na Figura 2 é possível ver uma comunicação entre um *host* localizado em uma rede privada com IP inválido (10.0.0.1) e um *host* com IP válido (200.200.200.200). Esta comunicação é possível graças ao *Masquerading*, efetuado no roteador 100.100.100.100, que troca o IP de origem, inválido na Internet, para o seu próprio IP válido. Este roteador ainda troca a porta de origem, assegurando que não existam duas conexões com a mesma porta de origem vindas da rede interna. Ao efetuar estas modificações (mais o decremento do TTL e o cálculo de um novo *checksum*), o roteador em questão armazena em uma tabela uma referência para esta conexão. A utilidade desta referência será explicada adiante.

Estas modificações permitem que a mensagem chegue ao destinatário e seja respondida corretamente ao remetente original, como ilustra a Figura 3.

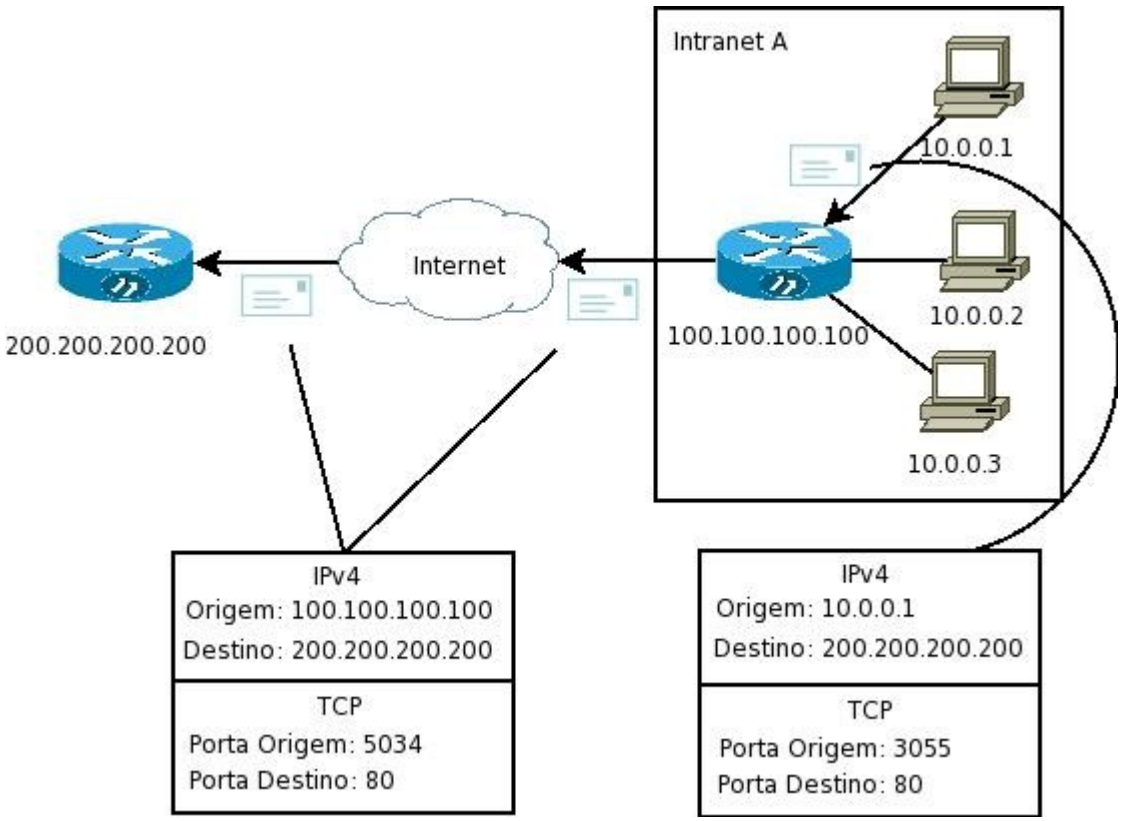

Figura 2 – Processo de mascaramento ao enviar um pacote

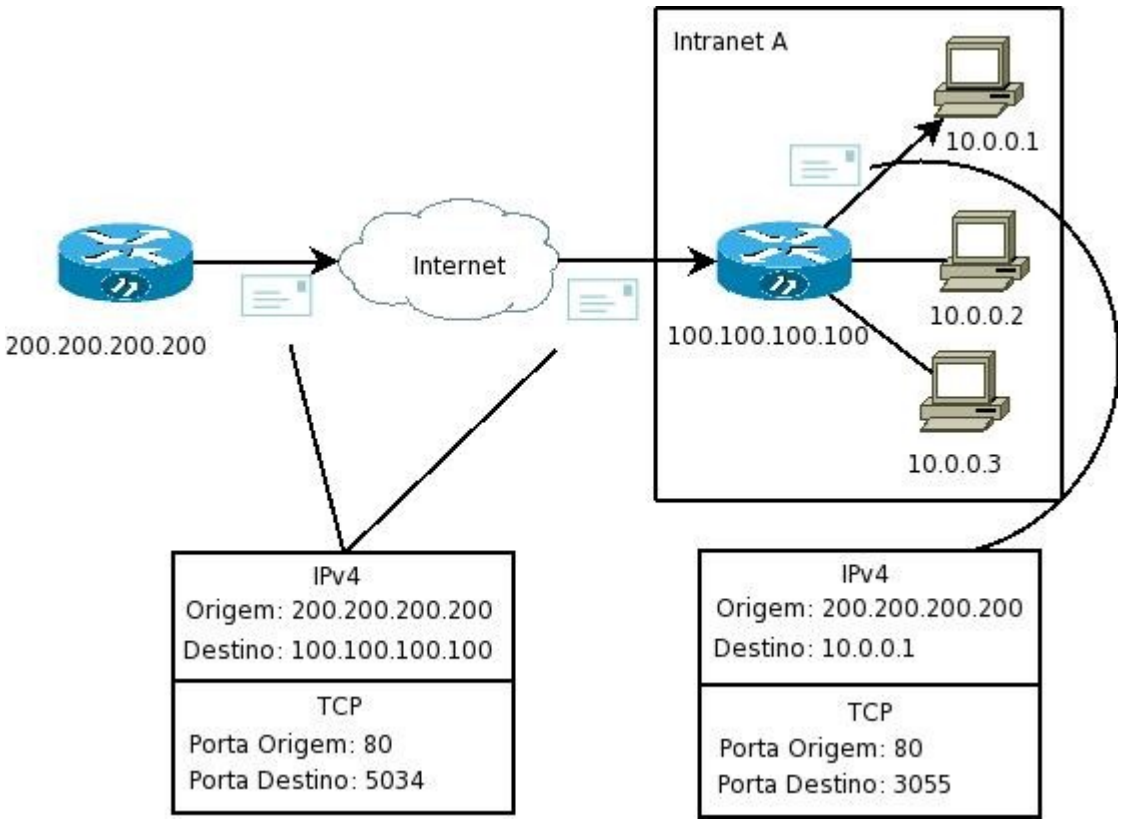

Figura 3 – Processo de mascaramento ao receber pacote

Quando o roteador 100.100.100.100 receber da Internet um pacote, será procurada em sua tabela, residente em memória, uma referência prévia, utilizando como base as informações de porta de destino, endereço IP de origem e destino. Caso não seja encontrada uma referência, o roteador processará o pacote como sendo para entrega local. Caso seja encontrada uma referência, o endereço IP e a porta destino serão trocados para os valores encontrados na tabela. Após a troca ter sido efetuada, o pacote é encaminhado para o destino final, ou seja, o host da intranet que aguarda esse pacote.

O *Masquerading* possibilita até hoje que redes privadas inteiras possam acessar a Internet sem estender o endereçamento provido pelo IPv4, o que seria praticamente impossível sem que alterações fossem feitas nos roteadores e nos softwares. Apesar de resolver a maioria dos problemas quanto a escassez de endereços IP válidos, o *Masquerading* apresenta uma limitação básica, que é permitir que um *host* com IP inválido somente se comunique na Internet caso a conexão seja iniciada a partir dele. Ainda existem muitos problemas relacionados com esta limitação, normalmente associados aos protocolos de camadas mais altas (no modelo TCP/IP) que levam em conta a conexão direta entre dois *hosts*, o que em tese deveria existir. É possível citar como exemplo o FTP (*File Transfer Protocol*), que, para permitir uma conexão não-passiva, parte da premissa de que a comunicação entre o servidor e o cliente é direta.

Para resolver os problemas destes protocolos com o *Masquerading,* outras técnicas foram desenvolvidas, como o *Connection Tracking*, o qual força o roteador executando o *Masquerading* a analisar os pacotes de uma conexão até o nível de camada de aplicação de modo a permitir que os dois *hosts* envolvidos no processo se comuniquem corretamente. Apesar de ser funcional para a maioria dos casos, há pelo menos três problemas no *Connection Tracking* (AYUSO, 2006):

- O roteador terá maior processamento, pois precisará analisar os pacotes até a camada de aplicação antes de roteá-los;
- Para cada protocolo é necessário um algoritmo que o analise e saiba tomar as decisões de roteamento e troca de endereços (NAT), quebrando o conceito de camadas existente na comunicação em rede;
- É necessário que já exista uma conexão iniciada para que o *tracking* seja feito.

Atualmente, existem outras soluções mais robustas quanto a conectividade de *hosts*, como o IPv6 (RFC 1752, 1995), já adotado pela IETF (IETF, 2006) como sucessor do IPv4. O IPv6, ou IPng (*IP Next Generation*), além de estender o endereçamento para 128 bits, extinguindo a necessidade do NAT, apresenta funcionalidades interessantes quanto à segurança de redes.

Em detrimento às suas vantagens, o IPv6 requer alterações em roteadores e *hosts* de toda a Internet. Além da necessidade de se manter os dois protocolos (IPv4 e IPv6) coexistindo em roteadores e *hosts* por questões de compatibilidade, softwares e alguns protocolos também devem estar preparados para o novo padrão sugerido pelo IPv6.

Hoje, grande parte dos softwares e roteadores na Internet têm suporte ao IPv6, porém a resistência à migração, mesmo depois de 10 anos de sua criação, ainda é grande.

4 INFORMAÇÕES TÉCNICAS

Para que seja possível explicar a opção, o protocolo, o módulo, o *Resource Record* e a biblioteca RNAT no Capítulo 5, o presente capítulo fornecerá uma breve descrição do funcionamento das *kernel capabilities,* campo IP *Options*, *Netfilter, Source Route e* DNS.

4.1 Capacidades do *Kernel* do Sistema Operacional

No padrão POSIX (*Portable Operating System Interface*) são definidas 26 capacidades que um software pode ter enquanto estiver em execução. Exemplos de capacidades são:

- CAP_CHOWN: permissão para mudar o dono de um arquivo;
- CAP_SYS_BOOT: permissão para reiniciar o sistema;
- CAP_NET_RAW: permissão para enviar pacotes RAW para a rede, ou seja, um pacote que não obedece os padrões definidos no protocolo de rede.

Toda vez que um programa invoca uma *system call*, por exemplo, para mudar o dono de um arquivo, é feita a seguinte checagem em *kernel space*:

```
if(capable(CAP_CHOWN)){....}
```

Esta checagem garante que um usuário qualquer não possa alterar o dono de um arquivo. Estas capacidades a princípio deveriam ser providas pelo sistema de arquivos, que as armazenaria em um espaço especial chamado *inode*. Atualmente, nenhum dos sistemas de arquivos suportados nativamente pelo Linux têm suporte a *capabilities*, apesar de o *kernel* tê-lo implementado.

4.2 IP *Options*

Em uma visão simplificada é possível definir o IP *Options* como um campo de utilização opcional e de implementação obrigatória, o que significa que sua presença no cabeçalho IPv4 não é exigida, porém todos os roteadores devem estar preparados para tratá-lo no caso de sua existência (ver Capítulo 2). Sua principal utilidade é permitir que opções personalizadas possam ser transmitidas sem a necessidade de alteração do protocolo.

Apesar de ser o único campo de tamanho variável do cabeçalho IPv4, é exigido que as opções nele inseridas sigam uma sintaxe pré-definida. Levando em consideração o tamanho, é possível classificar as opções em dois tipos:

- Opções de 1 byte;
- Opções com tamanho especificado no próprio campo IP *Options*;

As opções de 1 byte normalmente são utilizadas para controle do próprio IP *Options*. São definidas somente duas opções deste tipo (RFC 791, 1981):

- EOOL(*End of Option List*) – Representada pelo byte 0 (zero), deve delimitar o final das opções, ou seja, não deve existir entre uma opção e outra, caso exista mais de uma opção;
- NOOP (*No Operation*) – Representada pelo byte 1 (um), é uma opção que não causa efeito nenhum e normalmente é inserida entre duas opções somente para alinhamento de bytes.

As demais opções obrigatoriamente devem seguir o padrão demonstrado na Figura 4 (RFC 791, 1981).

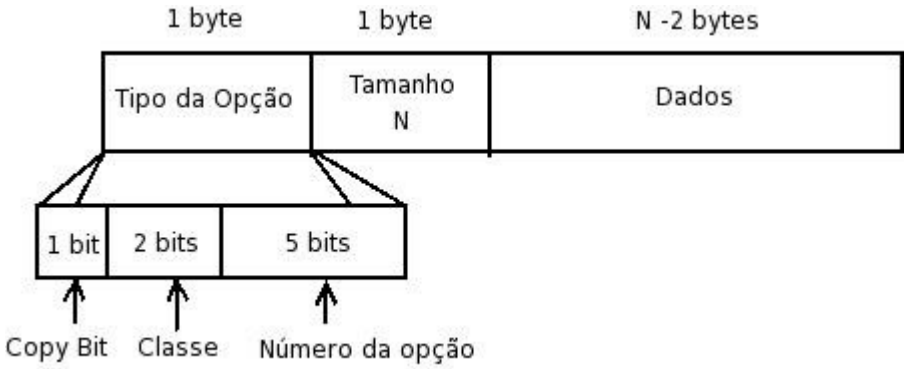

Figura 4 - Estrutura do campo IP *Options*.

- **Tipo da Opção** (8 bits): byte dividido em 3 subcampos:
  - **Copy bit** (1 bit): indica se a opção deve ser copiada para todos os fragmentos caso o pacote seja fragmentado;
  - **Classe** (2 bits) : indica se o campo é para controle ou para *debug*;
  - **Numero da opção** (5 bits): número de identificação da opção;
- **Tamanho** (8 bits): indica o tamanho total da opção, considerando o byte de "Tipo da Opção", o tamanho do campo de dados e o próprio campo "Tamanho";
- **Dados** (Tamanho – 2 bytes): Informações que se deseja transferir.

As opções, portanto, são identificadas pelo primeiro byte (Tipo da Opção). Os números de opção são controlados pelo IANA (*Internet Assigned Numbers Authority*), órgão responsável pela padronização dos números de protocolos utilizados na Internet.

O principal motivo de o IP *Options* seguir esta estrutura de 3 sub-campos para cada opção (exceto as de 1 byte) é permitir que um roteador que não conheça uma determinada opção possa repassá-la mesmo sem saber como tratá-la. Ao detectar um número de opção não conhecido durante a análise do IP *Options*, somente é necessário avançar no cabeçalho o número de bytes definido no campo "Tamanho", fazendo com que o byte atual seja o início da próxima opção, caso exista. Caso não exista, certamente será o byte 0 (zero), indicando o fim das opções.

Exemplos de opções são *Record Route, Source Route* e *Internet Timestamp* (RFC 791, 1981).

O número de bytes no campo IP *Options* é condicionado ao valor presente no campo IHL (*IP Header Length*) do cabeçalho IP, que informa o tamanho total do cabeçalho IP em palavras de 4 bytes. Somando todos os tamanhos dos campos do cabeçalho IPv4, tem-se o valor de 20 bytes, portanto o valor mínimo do IHL é 5 (4 x 5 = 20). Qualquer valor acima de 5 indica a existência do campo IP *Options,* e todo o espaço adicional, além dos 20 primeiros bytes, será atribuído a ele.

Devido ao fato de o tamanho do IP *Options* ser sempre um número múltiplo de 4, qualquer espaço não utilizado deve ser preenchido com zeros, tecnicamente chamado de *padding*.

Hoje, por exemplo, é possível visualizar o funcionamento do campo IP *Options* utilizando o comando *ping -R* (ver Figura 5). Este comando insere a opção RR (*Record Route*) no campo IP *Options*, que serve para gravar informações sobre as rotas por onde o pacote passou para alcançar o destino e voltar à origem. Estas rotas são armazenadas dentro do campo IP *Options*.

```
salem@rhapsody:/home/salem/rnat$ ping -R 192.168.1.35
        PING 192.168.1.35 (192.168.1.35) 56(124) bytes of data.
        64 bytes from 192.168.1.35: icmp_seq=1 ttl=64 time=0.468 ms
        RR:  192.168.1.33
             192.168.1.35
             192.168.1.35
             192.168.1.33
```

Figura 5 – Utilização do campo IP *Options* com o comando ping.

O cenário apresentado na Figura 5 é possível, pois a opção *Record Route* é conhecida pelo *kernel* do sistema operacional. O fato é que não é possível enviar um opção não conhecida pelo *kernel* dentro do IP *Options* sem a permissão necessária. Isto se deve ao fato de que, na hora em que a função *ip_options_compile()* (considerando um *kernel* Linux) é chamada e uma opção não conhecida é encontrada, uma checagem sobre as capacidades (ver Seção 4.1) do software em execução é feita.

Caso este software não tenha a capacidade CAP_NET_RAW, a opção é

ignorada e não enviada para a rede. A princípio somente o usuário root (administrador do sistema) executa os programas com esta capacidade ativada.

Vale lembrar que estas considerações são válidas somente para o *kernel* do *host* cliente, visto que os roteadores situados entre os *hosts* origem e destino, quando não sabem como tratar uma determinada opção, conseguem passá-la adiante.

4.3 Netfilter

Netfilter é um sistema para manuseio de pacotes localizado no *kernel* do Linux (RUSSEL,2002). O *Netfilter* mostra seu funcionamento através de *hooks* (ganchos) colocados em lugares específicos dentro do código do sistema operacional onde os pacotes podem ser manuseados. Para o IPv4 são definidos cinco *hooks:*

- INPUT: Trata pacotes com destino no *host* local;
- FORWARD: Trata pacotes com origem em um *host* (não o local) e com destino em outro;
- OUTPUT: Trata pacotes originados no *host* local;
- PREROUTING: Trata pacotes antes da tomada de decisão de roteamento;
- POSTROUTING: Trata pacotes depois da tomada de decisão de roteamento;

É possível entender o caminho percorrido pelo pacote no *kernel* do Linux e a ordem na qual as *hooks* são atingidas através da Figura 6 (RUSSEL, 2002).

Uma grande funcionalidade provida pelo Netfilter é permitir que novos códigos possam ser incorporados sem a alteração do *kernel*, normalmente através de módulos. É possível, por exemplo, escrever um novo módulo personalizado e associá-lo a uma determinada *hook*, de forma que qualquer pacote que passe por esta *hook* também passe pelo módulo. Na prática estas *hooks* são funções escritas em C (linguagem na qual o Linux é desenvolvido) e estão localizadas em lugares estratégicos na implementação da pilha TCP/IP. Para o *kernel*, a *hook* funciona

como uma caixa preta que receberá um pacote, podendo retorná-lo modificado, permitindo o seu trajeto normal no *kernel*, ou simplesmente descartá-lo.

Um software muito popular em ambiente Linux e que utiliza recursos do *netfilter* é o *iptables*, comumente utilizado para criar estruturas de *firewall* e, principalmente, executar *Masquerading* (ver Capítulo 3).

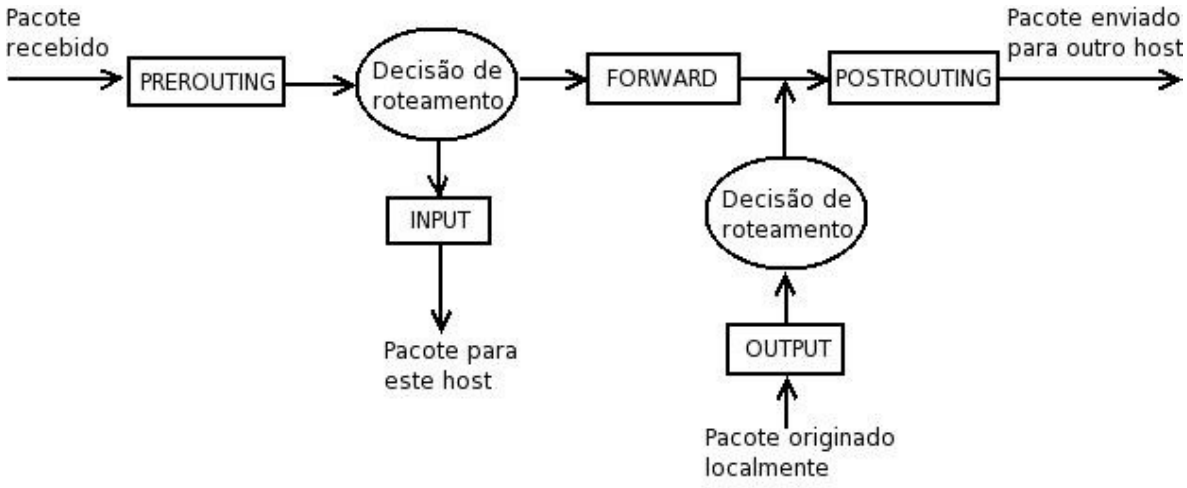

Figura 6 – Disposição das *Hooks* do *Netfilter* no kernel do Linux.

4.4 Source Route

O *Source Route* é uma técnica na qual o emissor de um pacote pode especificar alguns ou todos os roteadores pelo qual ele deverá passar. Isto é feito colocando em seu campo IP *Options* uma lista com os *hops*[1] até o destino. Devido ao desejo de desenvolver um esquema escalável (mais de um nível de roteamento) para o presente trabalho, a primeira tentativa foi a de utilizar o próprio *Source Route*, pois já está presente nos sistemas operacionais e roteadores. Existem basicamente dois tipos de *Source Route* (RFC 791, 1981):

- LSRR (*Loose Source and Record Route*): o próximo endereço IP na lista de roteadores por onde o pacote passará não precisa estar diretamente conectado à rede do roteador atual, o que significa que para atingir o próximo *hop* o pacote pode passar por vários roteadores intermediários;

1 Nome técnico dado para cada ponto de passagem de um pacote até seu destino.

- SSRR (*Strict Source and Record Route)*: toda a rota até o destino deve ser especificada.

O SSRR não serve para o presente trabalho, pois é inviável que um *host* tenha que conhecer previamente todos os endereços dos roteadores do caminho até o destino. O LSRR aparentemente seria a solução definitiva, porém no decorrer dos estudos apresentou uma série de problemas. O principal é que o *Source Route* foi praticamente eliminado da Internet por gerar problemas de segurança. Atualmente, muitos roteadores na Internet ao notar a presença de uma opção de *Source Route* descartam o pacote automaticamente. Como este não é um problema que tenha solução, os testes com S*ource Route* foram abortados.

Outro problema técnico encontrado, porém com menor impacto e fácil solução, é que o módulo *MASQUERADE* do *netfilter* (ver Seção 4.3) não mascara pacotes que contenham o *checksum* TCP errado. Quando um pacote TCP utilizando *Source Route* é enviado, seu *checksum* é calculado utilizando um pseudo-cabeçalho baseado em camada de rede, que contém, entre outros campos, o IP de destino, porém, no *Source Route* o pacote não é enviado diretamente para o destino, mas sim para um roteador intermediário (o especificado no campo IP O*ptions*), o que faz com que o *checksum* TCP trafegue errado na internet até que chegue em seu *host* final. O fato de o *checksum* estar incorreto é o suficiente para que o mascaramento não seja efetuado, tornando a idéia de usar *Source Route* novamente inválida.

4.5 DNS

O DNS (*Domain Name System*) é um mecanismo comumente utilizado para converter nomes de domínios em endereços IP. A existência deste sistema facilita o acesso a *hosts*, sendo que a origem não precisa conhecer previamente o endereço IP do destino, somente o seu domínio, que visivelmente é de mais fácil memorização, como por exemplo um domínio www.dainf.cefetpr.br em vez do endereço IP.

A arquitetura do DNS segue uma forma hierárquica, onde cada site pode ser autoridade para responder requisições DNS pelo seu próprio domínio.

O DNS funciona baseado no conceito de registros (*Resource Records ou*

RR's), que são os tipos de informações que podem ser traduzidas tomando como base um domínio. O cliente, ao fazer uma consulta ao servidor DNS, deve indicar qual o tipo de registro associado ao domínio consultado que ele deseja obter.
Atualmente, existem mais de 30 tipos de registros DNS cadastrados no IANA (IANA, 2006).

Exemplos de registros são:

- A – Endereço IPv4 genérico;
- AAAA – Endereço IPv6;
- MX – *Mail Exchanger* (Servidor responsável por receber e-mails para um domínio;
- TXT – Texto qualquer.

Cada consulta DNS pode retornar mais de um registro do mesmo tipo. Muitos sites utilizam este recurso como forma de equilibrar o acesso entre servidores, disponibilizando mais de um registro do tipo A para um mesmo domínio.

Em resumo, o DNS hoje é uma peça fundamental para o funcionamento da Internet, principalmente devido a sua vasta extensão e a inviabilidade de se decorar endereços IP.

5 IMPLEMENTAÇÃO DO RNAT (*Routable* NAT)

O principal objetivo do presente trabalho é alterar o IPv4 de forma que, com o menor número de modificações possível, um host de uma rede privada possa se conectar diretamente a um host de outra rede privada, ambos com IP's inválidos, utilizando os protocolos mais comuns na internet: UDP, TCP, ICMP (*Internet Control Message Protocol*). Faz parte da proposta achar uma identificação universal para que o endereço de cada *host* seja único no mundo, assim como desenvolver todas as modificações mantendo a máxima compatibilidade com o IPv4, aplicações e demais protocolos.

5.1 Opção RNAT

Como não é possível simplesmente aumentar o número de bytes no cabeçalho IP para estender os endereços de origem e destino, a identificação universal proposta é composta de dois endereços: um IP válido (referente ao roteador que executa o mascaramento) e um IP inválido (referente ao *host* final que passa pelo processo de mascaramento e que está localizado em uma rede privada).

Para manter a compatibilidade com o IPv4 criou-se uma nova opção para o campo IP *Options* (ver Capítulo 2) nomeada RNAT. O RNAT armazena os endereços IP da rede privada, ou seja, o IP *Options* é um campo que pode receber opções personalizadas do emissor do pacote, o qual pode inserir o endereço IP adicional, que seria o endereço IP do *host* da outra rede privada com o qual deseja se comunicar.

Como descrito no Capítulo 2, o campo IP *Options* deve seguir um formato bem definido. Para a opção RNAT, os campos do IP *Options* contêm os seguintes valores (representação em base binária):

- **Tipo da Opção**
  - **Copy bit**: 1 (opção deve ser copiada nos fragmentos);
  - **Classe**: 00 (classe de controle);
  - **Número da opção**: 11001 (25 em decimal) próximo número

disponível.

- **Tamanho**: 0xB (11 em base decimal);
- **Dados**: 4 bytes (IP da rede privada de origem), 4 bytes (IP da rede privada de destino), 1 byte (reservado para uso futuro). Os campos de origem e destino mudam de posição durante a comunicação.

Agrupando todos os oito bits iniciais tem-se o primeiro byte, número 153 em base decimal (ou 0x99 em hexadecimal). O segundo byte do campo IP *Options* é relativo ao tamanho total da opção, medido em bytes. Devido ao fato de ter sido incluída uma opção extra para uso futuro com o tamanho de 1 byte, o tamanho total será de 11 bytes (ou 0xB em hexadecimal), pois a opção deverá comportar 2 IP's de 32 bits (8 bytes) + 1 byte inicial + 1 byte de tamanho + 1 byte para uso futuro. Assim tem-se 1+1+8+1 = 11.

É possível entender melhor a representação da opção analisando a Figura 7.

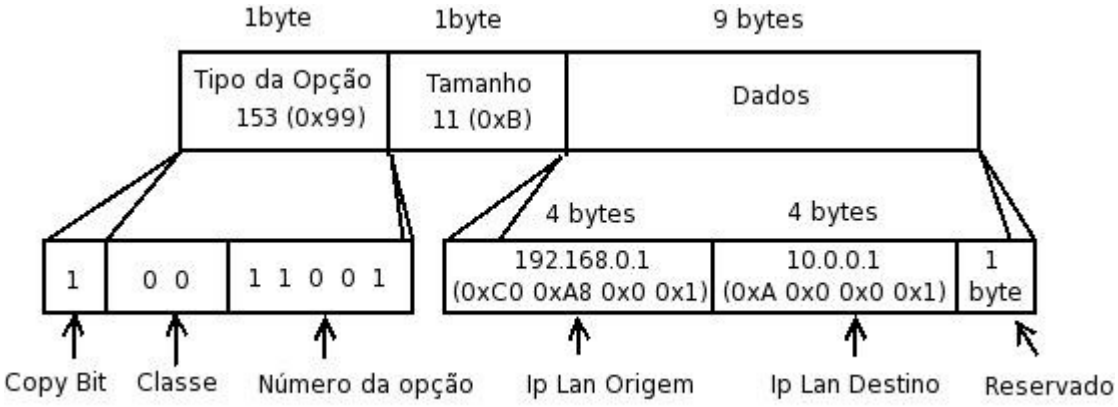

Figura 7 – Estrutura da opção RNAT no campo IP *Options*

O desempenho da comunicação IPv4, mesmo após a adição dos 11 bytes da opção RNAT ao cabeçalho, pode ser conferido no Apêndice A. A forma como esta opção será utilizada está descrita a seguir.

5.2 Protocolo RNAT

Como o S*ource Route* não foi a melhor escolha como base para o presente trabalho, desenvolveu-se um protocolo próprio, nomeado *Routable NAT*

(RNAT), pelo fato de permitir o acesso direto através da Internet entre *hosts* que estão passando pelo processo de mascaramento.

Este protocolo dita as regras de como tratar a nova opção para o campo IP *Options* descrita no capítulo 5.1. A idéia é que toda vez que um roteador receber um pacote e detectar a presença da opção RNAT, tomará uma decisão de roteamento baseada nela. Nesta opção sempre estarão os endereços IP referentes aos *hosts* que passam pelo processo de mascaramento. Em uma visão macro, a comunicação é sempre feita entre os dois roteadores, porém estes últimos ao analisar a opção conseguem rotear corretamente os pacotes para os *hosts* das redes privadas e com IP's inválidos que estão se comunicando realmente.

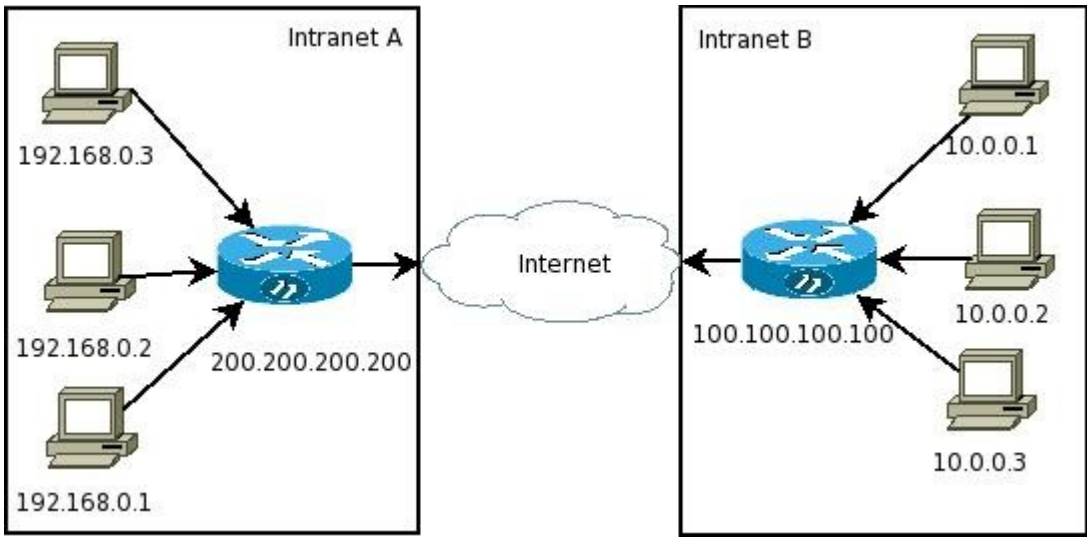

Figura 8 – Redes privadas com acesso à Internet

O cenário demonstrado na Figura 8, redes privadas com acesso à Internet, é algo muito comum tanto em ambientes corporativos como em ambientes residenciais. Vamos supor que o *host* 192.168.0.1 da rede privada A deseje se comunicar com o roteador 100.100.100.100. Esta é uma comunicação possível graças ao NAT efetuado no servidor 200.200.200.200, que mascara a origem. Ou seja, para o *host* 100.100.100.100, a comunicação é feita entre ele e o *host* 200.200.200.200, sendo que na verdade o *host* com o qual ele se comunica é o 192.168.0.1.

O grande problema é quando o *host* 192.168.0.1 deseja se comunicar com um *host* que está depois do roteador 100.100.100.100, como, por exemplo, o *host* 10.0.0.1. Supondo que o acesso esteja sendo feito na porta TCP 80, este

cenário é possível somente se o roteador da rede privada B possuir uma regra baseada na camada de transporte para direcionar todo o tráfego recebido na porta TCP 80 para o *host* 10.0.0.1 (DNAT). Esta solução tem pelo menos dois problemas graves:

- O emissor não sabe previamente qual o *host* que ele irá acessar, pois é feito um redirecionamento no destino;
- Não é possível atingir outro *host* nesta mesma rede privada (10.0.0.2, por exemplo) que atenda na porta 80, pois todas as conexões nesta porta serão redirecionadas para 10.0.0.1.

O segundo problema é o mais comum e, atualmente, pode-se citar muitos exemplos de como isto interfere na comunicação entre dois *hosts*, como em uma transferência de arquivos via um IM (*Instant Messenger*) qualquer. A comunicação direta entre estes *hosts* não pode acontecer, devido aos dois *hosts* estarem atrás de um roteador executando *Masquerading*. Isso obriga o envolvimento de um terceiro *host* na Internet com IP válido para servir como ponte, fazendo com que a segurança seja comprometida (pelo fato de enviar os dados para um *host* intermediário, o qual pode capturar o tráfego) e a velocidade reduzida devido ao consumo de banda duplicado.

A idéia é utilizar a opção RNAT para contornar os problemas expostos acima. A explicação do seu funcionamento será baseada no cenário demonstrado na Figura 8, levando em consideração uma comunicação entre o *host* 192.168.0.1 da rede privada A e o *host* 10.0.0.1 da rede B.

O *host* 192.168.0.1 inicia o processo de conexão fazendo uma solicitação ao sistema operacional para abrir um *socket*[2] com o *host* 100.100.100.100. Porém, antes de chamar a função *connect(), a* qual efetivamente manda o primeiro pacote para o destino, a função *setsockopt()* deve ser invocada passando para ela o formato da opção RNAT (descrito na Seção 5.1). O *host* 192.168.0.1 não deverá informar o IP da rede privada de origem, ou seja, o seu próprio IP. Por questões de segurança, este endereço será preenchido no roteador 200.200.200.200 antes de enviar o pacote para a Internet. Desta forma, qualquer valor preenchido no campo destinado ao IP da rede privada será sobrescrito.

2 Referência de uma conexão. Um socket para comunicação em rede pode ser considerado como uma associação lógica entre um IP e uma porta TCP ou UDP.

Para seguir um padrão é sugerido que o *host* cliente sempre preencha este campo com o endereço IP 255.255.255.255. A Figura 9 mostra como a opção RNAT estará disposta no pacote ao sair do *host* de origem.

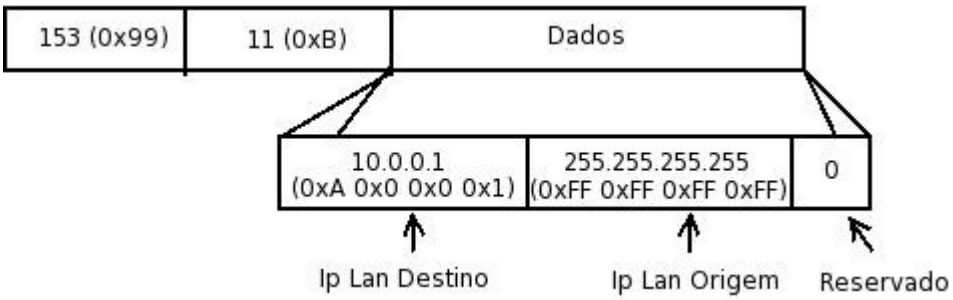

Figura 9 – Opção RNAT ao sair do *host* de origem

Ao chegar no roteador 200.200.200.200, o sistema detectará que o pacote teve origem em sua rede privada e fará o NAT normalmente, porém, trocará o conteúdo do campo destinado ao IP da LAN (*Local Area Network*) de origem pelo verdadeiro IP do *host* que enviou o pacote. A Figura 10 demonstra o conteúdo da opção RNAT no momento em que o pacote deixa o roteador 200.200.200.200.

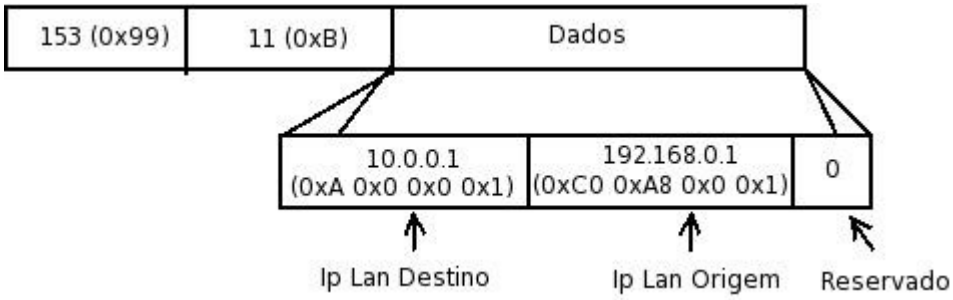

Figura 10 – Opção RNAT ao ser enviada para a Internet

Quando o pacote chegar no roteador 100.100.100.100, o mesmo deverá detectar a presença da opção RNAT e, como o pacote teve origem externa, assumirá que este pacote deverá ser roteado para sua rede privada. Para saber qual será o *host* destino, consultará a opção RNAT e colocará no endereço destino do cabeçalho IP o verdadeiro endereço, neste caso 10.0.0.1. Além desta alteração, os endereços de origem e destino da opção RNAT são trocados de posição. Isto é feito para que, no momento em que a opção chegue no *host* destino, somente seja necessário copiá-la de volta ao pacote sem precisar efetuar nenhuma alteração. A Figura 11 demonstra a opção RNAT no momento em que chega ao *host* destino.

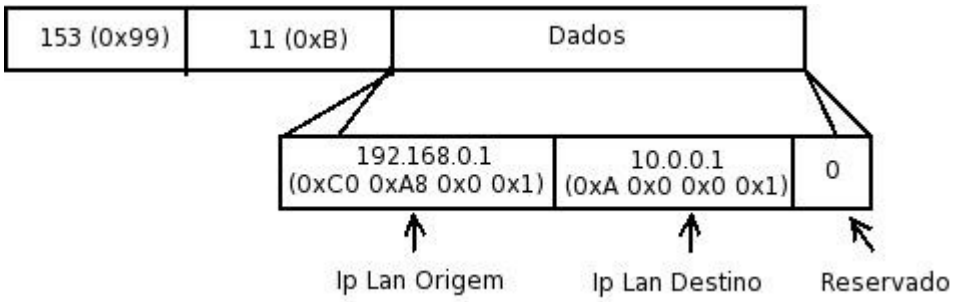

Figura 11 – Opção RNAT ao ser enviado para o *host* destino.

Neste ponto é necessário considerar dois cenários:

a) Se o *kernel* do *host* destino não conhecer a opção RNAT, esta não será copiada de volta para o pacote, porém a comunicação é efetuada com sucesso, pois o *host* destino retornará o pacote para o *host* 200.200.200.200. Este por sua vez já manteve uma referência previamente em sua tabela NAT ao processar o pacote, e desta forma o roteará normalmente para o verdadeiro *host* emissor.
b) Se o *kernel* do *host* destino conhecer a opção, esta será copiada de volta para o pacote. No momento em que este pacote passar pelo *host* 100.100.100.100 com destino ao *host* 200.200.200.200, será detectada que existe a opção RNAT e que o pacote teve origem em sua rede privada, assumindo desta forma que o endereço de origem na opção RNAT deve ser preenchido com o endereço do verdadeiro emissor (10.0.0.1), fechando o ciclo de comunicação.

Com estas regras a comunicação entre os dois *hosts* (192.168.0.1 e 10.0.0.1) torna-se possível, mesmo considerando que ambos tenham IP's inválidos e passem pelo processo de mascaramento.

5.3 Módulo rnat_router

Para que a comunicação proposta no presente trabalho, entre dois *hosts* que passam pelo processo de mascaramento, seja possível, é necessário que os roteadores saibam como tratar os pacotes contendo a IP *Option* RNAT.

Normalmente, o código que trata estes pacotes está localizado

internamente no sistema operacional dos roteadores, o que dificulta a sua alteração.

Felizmente, o *kernel* Linux provê uma forma simples para que novas funcionalidades sejam agregadas sem a necessidade de recompilação: *Kernel Modules* (Módulos do *kernel*). Um módulo é um trecho de código compilado em linguagem C e formatado de forma específica, podendo se agregar ao *kernel* sem a necessidade de reinicialização do sistema.

Para tratar a opção RNAT foi desenvolvido um módulo para o *kernel* nomeado *rnat_router*. Este módulo se registra em duas *hooks* do *netfilter* (ver Seção 4.3): PREROUTING e POSTROUTING. Este registro permite que todos os pacotes sejam tratados por este módulo sempre que atingirem as *hooks* especificadas. A criação do módulo *rnat_router* elimina a necessidade de ter uma versão modificada do *kernel* Linux nos roteadores em um primeiro momento, porém a idéia é que, posteriormente, os roteadores implementem nativamente o suporte à opção RNAT. O código fonte do módulo rnat_router é mostrado no Apêndice B.

O funcionamento interno do módulo segue exatamente a especificação do protocolo RNAT (ver Seção 5.2).

5.4 Resource Record RNAT

Para o presente trabalho, percebeu-se que o padrão atual de DNS (descrito na Seção 4.5) não era suficiente para se referenciar 2 IP's ao invés de um, ou seja, os endereços IP do roteador e do *host* da rede privada. Apesar de o DNS suportar mais de um registro do mesmo tipo por domínio, a idéia atual é que, quando o DNS retornar mais de um endereço por domínio, o software cliente escolha um destes endereços para fazer o acesso (ver Seção 4.5), o que não é verdade considerando o sistema RNAT.

Para o RNAT, obrigatoriamente um dos endereços IP deve ser o do roteador e o outro do *host* da rede privada. Como não é possível fazer esta especificação através do DNS, criou-se um novo registro chamado RNAT. Este novo RR é na verdade uma cópia do RR A, somente com algumas informações trocadas. Desta forma é possível que o software cliente consulte o servidor DNS solicitando o RR RNAT para obter o IP da outra rede privada e, posteriormente, faça a mesma

solicitação com o RR A para obter o IP do roteador.

A criação do RR RNAT provê uma forma elegante de se referenciar um *host* de outra rede privada, porém para poder utilizar este RR é necessário que o servidor DNS tenha suporte à ele. Este suporte exige que o servidor DNS seja alterado. Para que esta alteração seja evitada, um método de compatibilidade foi planejado, utilizando um RR hoje já presente nos servidores DNS. O RR escolhido para este fim foi o TXT, ou seja, em vez de retornar o endereço IP, é retornado um texto com uma formatação bem definida representando o endereço IP.

Quando utilizado o RR na forma TXT, deve-se seguir o seguinte padrão:

- "RNAT 192.168.0.1".

Ou seja, a *string* "RNAT" deve ser inserida para que o software cliente reconheça esta opção como sendo um IP inválido, seguida do IP propriamente dito. Um exemplo de arquivo para configuração de um domínio para o servidor DNS *BIND* (ISC, 2006) é mostrado no Apêndice C.

No software cliente é sugerido que primeiro seja feita uma consulta pelo RR RNAT. Caso não seja retornado um resultado, a consulta deve ser feita pelo RR TXT. Como dito anteriormente, consultas com o RR TXT somente devem ser utilizadas para manter a compatibilidade com as implementações de servidores DNS atuais, porém tendem a ficar obsoletas no caso de o RR RNAT fazer parte nativamente de implementações posteriores destes servidores.

5.5 Biblioteca de funções RNAT

Como explicado na Seção 5.2, os softwares clientes necessitam de uma pequena alteração em seu código, mais especificamente na solicitação de abertura de um *socket* ao sistema operacional.

Para facilitar a migração dos programas ao novo padrão, uma biblioteca escrita em C foi desenvolvida. Atualmente, possui somente duas funções, que estão descritas abaixo.

Devido ao fato de a estrutura da opção RNAT no campo IP *Options* ser sempre a mesma, apenas alterando o IP inválido de destino, foi criada uma função chamada *do_rnat (int socket, unsigned int ip),* a qual recebe como parâmetros a referência do *socket* (retornada pela função *socket()*) e o IP propriamente dito. A

função *do_rnat()* se encarregará de fazer as modificações necessárias internamente através da *system call setsockopt()*.

A outra função presente na biblioteca chama-se *ns_rnat_query (char *domain).* Esta função é capaz de consultar o servidor DNS pedindo por um RR do tipo RNAT e em caso de falha consultar o RR TXT (modo de compatibilidade).

É possível ver um exemplo de uso da biblioteca RNAT no Apêndice D.

5.6 Informações importantes sobre o RNAT

Algumas informações relevantes sobre a arquitetura das redes utilizando RNAT são fornecidas abaixo:

- Roteadores de redes privadas que não implementam o RNAT permitem que seus *hosts* acessem *hosts* de outras redes privadas utilizando RNAT, porém não permitem que *hosts* de outras redes privadas acessem os *hosts* de sua própria rede privada*;*
- Softwares servidores não necessitam de alterações para se comunicar utilizando RNAT;
- Softwares clientes necessitam de uma pequena alteração na abertura do *socket* para poder inserir a opção RNAT nos pacotes. Esta alteração pode ser feita com ajuda da biblioteca RNAT, desde que o software esteja escrito em linguagem C;
- Devido ao fato de o *kernel* do *host* cliente não conhecer a opção RNAT, não é possível inserí-la nos pacotes utilizando um usuário comum, somente com um usuário privilegiado (root em sistemas Linux), pois o pacote é considerado RAW, ou seja, não obedece os padrões de rede impostos pelo *kernel*. Esta situação pode ser solucionada utilizando um *kernel* alterado para reconhecer a opção RNAT, ou utilizando o módulo do *kernel* chamado rnat_host, fornecido junto com o software RNAT, que sobrescreve a função que impede a inserção da opção RNAT (ip_options_compile), fazendo com que o *kernel* passe a reconhecê-la sem a necessidade de ser recompilado;
- Caso o roteador da rede conectado à Internet não seja um computador com o *kernel* Linux e se deseje receber conexões externas para os

hosts da rede privada, é possível redirecionar as portas todas TCP e UDP via NAT para um computador com Linux localizado entre a rede privada e o roteador, que por sua vez poderá ter suporte ao RNAT.

6 FUNCIONAMENTO DO RNAT

O presente capítulo tem como objetivo demostrar o funcionamento do RNAT de forma prática. Para que isto seja possível, modificações (*patches*) para alguns softwares populares foram feitas, utilizando como base a biblioteca RNAT (Seção 5.5). As modificações disponibilizadas visam apenas provar a funcionalidade do RNAT, e não devem ser consideradas como as modificações oficiais. Modificações oficiais devem ser disponibilizadas pelos mantenedores do software em questão.

6.1 RNAT em roteadores

Para que roteadores das redes privadas permitam que seus *hosts* sejam acessados a partir de outras redes privadas, é necessário que ele utilize o módulo *rnat_router*. Este módulo pode ser adquirido na página oficial do projeto RNAT (RNAT, 2006). Para compilar o módulo é necessário ter o código fonte do *kernel* utilizado. Distribuições Linux normalmente disponibilizam este código em um pacote chamado kernel-sources.

Para executar a compilação do módulo somente é necessário digitar 'make' no diretório 'rnat_router' dentro do código fonte do RNAT. Este processo, ao ser executado com sucesso, criará um arquivo chamado *rnat_router.ko*, que é o módulo propriamente dito. Para carregá-lo no kernel basta executar o comando '*insmod rnat_router.ko*'. É possível certificar-se que seu carregamento ocorreu com sucesso analisando a saída do comando '*dmesg* | *grep RNAT*', que deverá retornar o seguinte texto:

```
root@rhapsody:~/rnat# insmod rnat_router.ko
root@rhapsody:~/rnat# dmesg | grep RNAT
RNAT: Masquerade interface for rnat is eth0
RNAT: Intranet interface for rnat is eth1
RNAT: This is an experimental module, use at your own risk.
root@rhapsody:~/rnat#
```

Neste ponto, todos os pacotes recebidos pelo roteador estão passando pelo módulo *rnat_router*, porém ele só processará pacotes RNAT que cheguem pela interface de rede eth0 e que o *host* com IP inválido de destino seja acessível através da interface eth1. Estes valores são atribuídos automaticamente quando nenhum parâmetro é passado ao módulo no momento do seu carregamento.

Como nem sempre esta configuração de interfaces de rede é igual em roteadores, há duas forma de mudar os valores padrão definidos pelo módulo:

- Passagem de parâmetros na hora da carga do módulo;
- Utilizar arquivos especiais para alterar o funcionamento do módulo em tempo de execução, sem necessitar descarregá-lo e carregá-lo novamente.

Para o primeiro caso, considerando que as interfaces de rede para a Internet e para a rede privada são respectivamente eth0 e eth3, é possível carregar o módulo da seguinte forma:

```
root@rhapsody:~/rnat# insmod rnat_router.ko dev_ext=eth0 dev_lan=eth3
```

Para o segundo caso, considerando a mesma configuração de interfaces de rede do exemplo anterior, é possível obter o mesmo resultado através dos comandos:

```
root@rhapsody:~/rnat# insmod rnat_router.ko
root@rhapsody:~/rnat# echo -n eth0 > /sys/module/rnat_router/parameters/dev_ext
root@rhapsody:~/rnat# echo -n eth3 > /sys/module/rnat_router/parameters/dev_lan
```

6.2 RNAT em hosts

6.2.1 Kernel

Para que usuários sem privilégios especiais possam enviar a opção RNAT dentro de pacotes, é necessário que o *kernel* dos *hosts* clientes tenham o módulo *rnat_host* carregado. O código fonte é mostrado no Apêndice E. O processo

de compilação é exatamente igual ao do módulo *rnat_router* para o *kernel* dos roteadores (seção 6.1). O carregamento não exige nenhum parâmetro especial.

6.2.2 OpenSSH

O software OpenSSH é amplamente utilizado para efetuar acesso a *hosts* remotos. A alteração efetuada em seu código basicamente adiciona suporte a consultas DNS pela RR RNAT e a inserção da opção RNAT nos pacotes enviados por ele.

Uma sessão ssh utilizando RNAT pode ser conferida abaixo:

```
salem@rhapsody:~$ ./ssh
usage: ssh [-1246AaCfgkMNnqsTtVvXxY] [-b bind_address] [-c cipher_spec]
        [-D [bind_address:]port] [-e escape_char] [-F configfile]
        [-i identity_file] [-L [bind_address:]port:host:hostport]
        [-l login_name] [-m mac_spec] [-O ctl_cmd] [-o option] [-p port]
        [-R [bind_address:]port:host:hostport] [-S ctl_path]
        [-w tunnel:tunnel] [user@]hostname [command]
        [-r RNAT host IP]
salem@rhapsody:~$ ./ssh 200.200.200.200 -r 192.168.0.1 -l usuario
Password:
usuario@host:~/$
```

6.2.3 Putty

O Putty é um software popular por permitir conexões ssh (*Secure Shell*) utilizando uma interface gráfica. As modificações efetuadas foram as mesmas do software OpenSSH (Seção 6.2.2), porém com alterações em sua interface, conforme ilustrado na Figura 12.

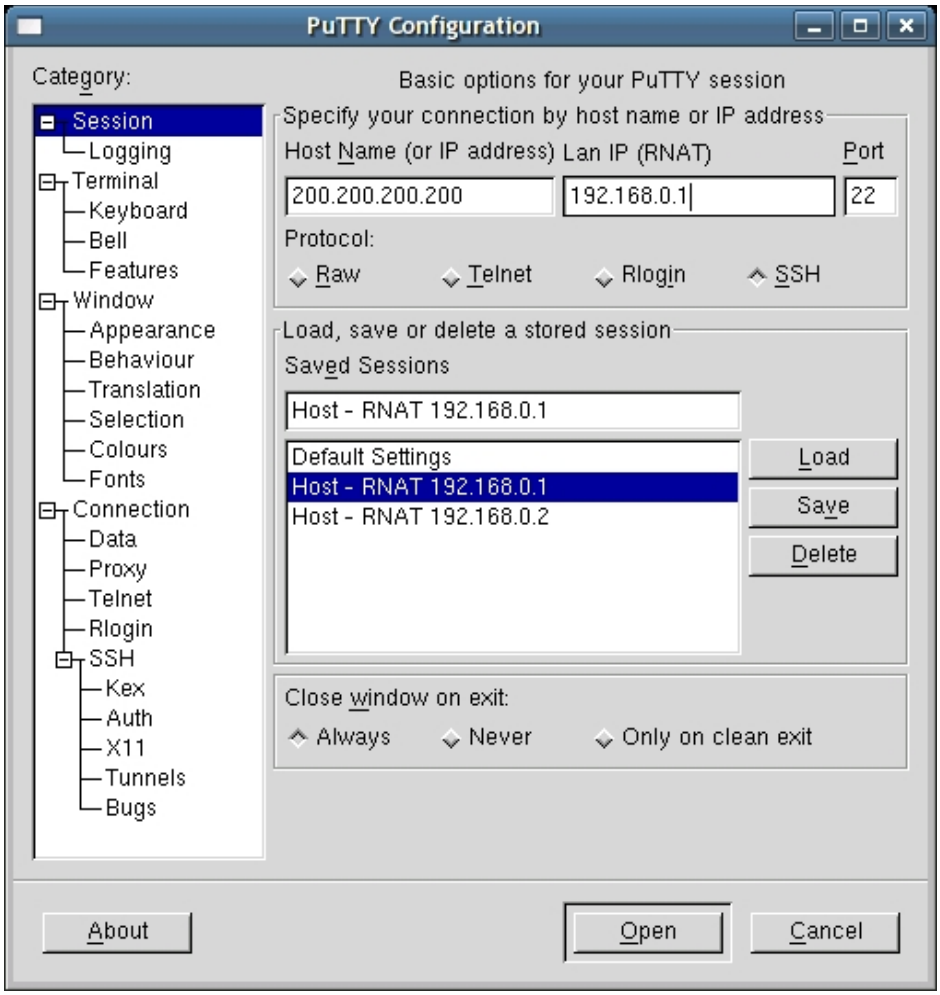

Figura 12 – Software putty com suporte a RNAT.

6.2.4 Ping

O software ping é popular em diversas plataformas. Comumente utilizado para testar a conectividade entre dois *hosts.* Para isto utiliza mensagens ICMP do tipo *echo request-reply.* É possível encontrar a modificação completa do seu código fonte para suportar RNAT no Apêndice F. Um exemplo de sua utilização com RNAT pode ser conferido abaixo.

```
root@rhapsody:~/netkit-base-0.17/ping# ./ping 200.200.200.200 -a 192.168.0.1 -c 3
PING 200.200.200.200 (200.200.200.200 - 192.168.0.1): 56 octets data
64 octets from 200.200.200.200 - 192.168.0.1: icmp_seq=0 ttl=254 time=34.1 ms
64 octets from 200.200.200.200 - 192.168.0.1: icmp_seq=1 ttl=254 time=32.1 ms
64 octets from 200.200.200.200 - 192.168.0.1: icmp_seq=2 ttl=254 time=37.1 ms

3 packets transmitted, 3 packets received, 0% packet loss
root@rhapsody:~/netkit-base-0.17/ping#
```

CONCLUSÃO

Com a utilização do RNAT, o acesso direto a *hosts* que passam por um processo de mascaramento torna-se possível. Apesar de todo o estudo ter sido baseado em plataforma GNU/Linux, o conceito foi desenvolvido de forma genérica e tende a ser compatível com qualquer sistema operacional. Uma grande vantagem em sua utilização é permitir que redes privadas consigam se comunicar com outras sem alterações em suas topologias. É importante enfatizar que o RNAT não pretende ser uma alternativa ao IPv6, mas apenas um conceito que adiciona uma funcionalidade não provida pelo NAT: a conexão direta. Apesar de o endereçamento com RNAT ser estendido (por se tratar de dois endereços de 32 bits), ainda é notavelmente pequeno comparado aos 128 bits do IPv6.

O RNAT provê alta compatibilidade com *softwares* servidores já existentes e requer pequenas alterações em *softwares* clientes, tornando-se assim uma solução a curto prazo para a conexão direta entre redes privadas enquanto um protocolo mais robusto, como o IPv6, não é utilizado em toda a Internet.

TRABALHOS FUTUROS

Como trabalho futuro propõe-se a modificação na implementação de *sockets* dos sistemas operacionais, fazendo com que o endereço RNAT seja reconhecido nativamente como um identificador, além das portas TCP/UDP e do endereço IP do roteador de origem. Desta forma softwares servidores poderiam identificar o IP da rede privada de origem através de uma chamada de sistema e o NAT não se faria mais necessário a longo prazo. Outra proposta seria adicionar todo o código que hoje está localizado nos módulos *rnat_router* e *rnat_host* internamente no *kernel*, fazendo com que ele reconheça a opção nativamente e evite que *hosts* clientes precisem ter privilégios especiais ou utilizem o módulo rnat_host para poder enviar a opção RNAT nos pacotes.

APÊNDICE A

ANÁLISE DE DESEMPENHO DO RNAT

As Figuras A1 e A2 ilustram os testes realizados para analisar o desempenho da comunicação IPv4 sem e com o uso do RNAT, respectivamente. Os testes foram efetuados calculando o tempo de resposta de 60 mensagens ICMP do tipo *echo request-reply* (ping). Vale lembrar que todo teste de desempenho efetuado via Internet não garante resultados precisos, devido ao fato de não ser um ambiente totalmente controlado e imune a interferências externas.

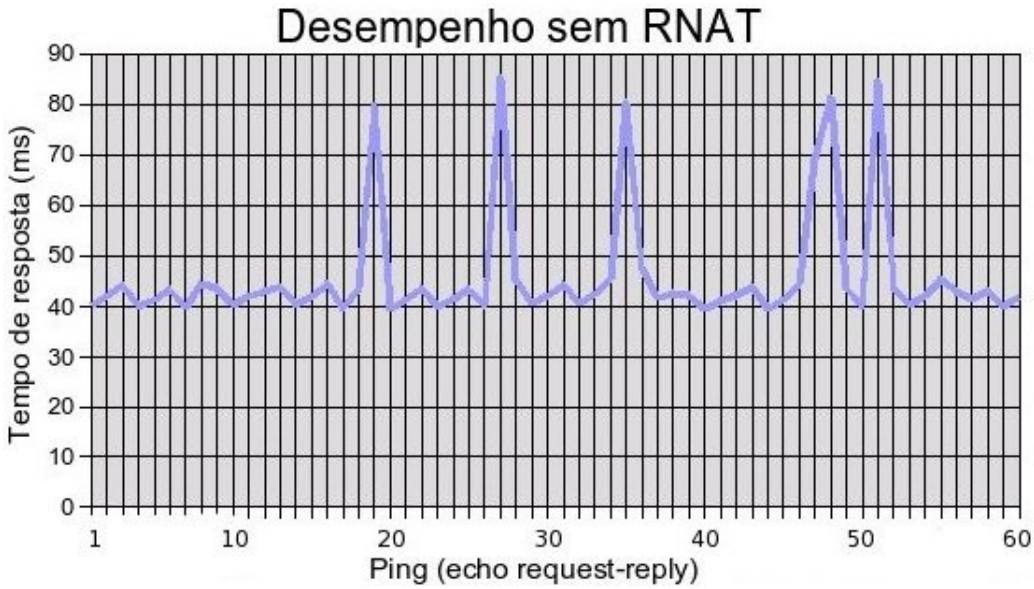

Figura A1 - Tempo de resposta para um *host* com IP válido.

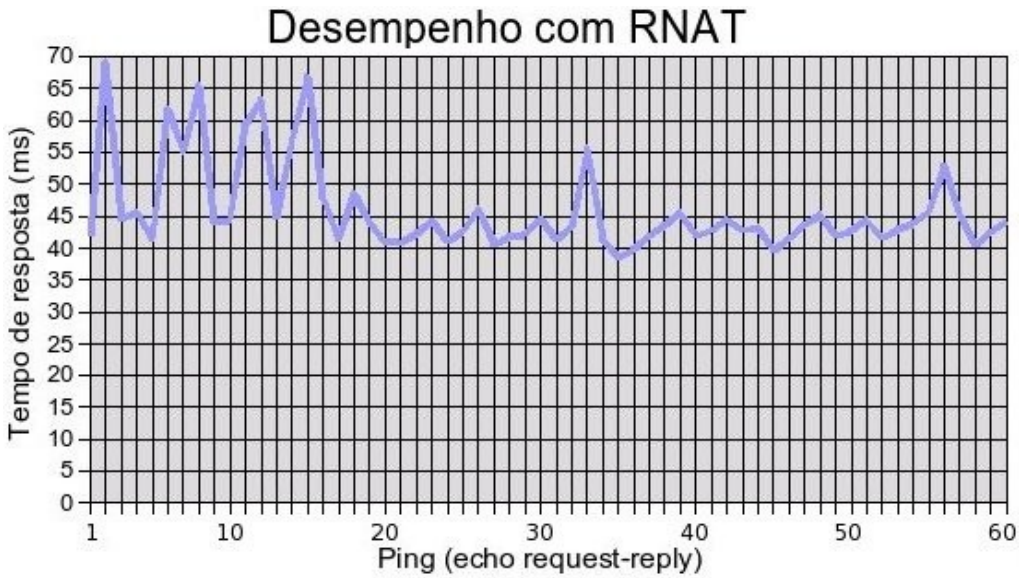

Figura A2 - Tempo de resposta para um *host* com IP inválido de outra intranet.

APÊNDICE B

CÓDIGO FONTE DO MÓDULO RNAT_ROUTER

```
/*
 * Copyright (C) 2006 Tiago Salem Herrmann <tiagosh@gmail.com>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; only version
 * 2 of the License.
 *
 */

#include <linux/module.h>
#include <linux/skbuff.h>
#include <linux/tcp.h>
#include <linux/inetdevice.h>
#include <linux/icmp.h>
#include <linux/udp.h>
#include <net/ip.h>
#include <net/route.h>
#include <linux/moduleparam.h> // sysfs
#include <linux/netfilter_ipv4.h>

#ifdef CONFIG_PROC_FS // only if we have procfs
#include <linux/proc_fs.h>
#endif

// define the rnat option number.
// | 8 bits   | 8 bits |  4 bytes  |  4 bytes  | 1 byte |
// -----------------------------------------------------
// |1|00|11001|  0x0B  |  X.X.X.X  |  X.X.X.X  |    X   |
// |-----------------------------------------------------
// | 153 class|  size  |   ip 1    |    ip 2   |extended|
//
#define IPOPT_RNAT      (25|IPOPT_CONTROL|IPOPT_COPY)
#define IPOPT_RNAT_SIZE 0xb
#define DEFAULT_IFACE_EXT    "eth0"
#define DEFAULT_IFACE_LAN    "eth1"

/* NEVER USE GLOBAL VARIABLES, OK? */

static struct nf_hook_ops netfilter_ops_in; /* NF_IP_PRE_ROUTING */
static struct nf_hook_ops netfilter_ops_out; /* NF_IP_PRE_ROUTING */

#ifdef CONFIG_PROC_FS
static struct proc_dir_entry *proc_dev_ext;
static struct proc_dir_entry *proc_dev_lan;
#endif

static char dev_ext[10];
static char dev_lan[10];

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Tiago Salem Herrmann <tiagosh@gmail.com>");
MODULE_DESCRIPTION("Netfilter module which allows routable NAT");

module_param_string(dev_ext,dev_ext, sizeof(dev_ext), 0600);
module_param_string(dev_lan,dev_lan, sizeof(dev_lan), 0600);

MODULE_PARM_DESC(dev_ext, "Internet output NIC");
MODULE_PARM_DESC(dev_lan, "Intranet output NIC");
```

```
/* stolen from net/ipv4/ip_options.c
 * This is a modified version of ip_options_compile()
 * only to allow us to know if there is a rnat option.
 * This function returns 0 if there isn't a rnat option,
 * or if we find some error in the IP options.
 * Any other value is the offset of rnat option from ip
 * header.
*/

int ip_options_rnat(struct sk_buff * skb)
{
      int l;
      unsigned char * iph;
      unsigned char * optptr;
      int optlen;
      int is_rnat=0;
      int rnat_offset=0;
      unsigned char * pp_ptr = NULL;
      struct ip_options *opt=NULL;
      opt=kmalloc(sizeof(struct ip_options),GFP_KERNEL);
      if (opt) {
            memset(opt,0,sizeof(struct ip_options));
            iph = skb->nh.raw;
            opt->optlen = ((struct iphdr *)iph)->ihl*4 - sizeof(struct
iphdr);
            optptr = iph + sizeof(struct iphdr);
            opt->is_data = 0;
      }else{
            return 0;
      }

      for (l = opt->optlen; l > 0; ) {

            switch (*optptr) {
                  case IPOPT_END:
                  for (optptr++, l--; l>0; optptr++, l--) {
                        if (*optptr != IPOPT_END) {
                              *optptr = IPOPT_END;
                              opt->is_changed = 1;
                        }
                  }
                  goto eol;
                  case IPOPT_NOOP:
                  l--;
                  optptr++;
                  continue;
            }
            optlen = optptr[1];
            if (optlen<2 || optlen>l) {
                  pp_ptr = optptr;
                  goto error;
            }
            if (*optptr == IPOPT_RNAT){
                  // the size is fixed
                  if(optlen!=IPOPT_RNAT_SIZE)
                        goto error;
                  is_rnat = 1;
                  rnat_offset = (optptr - iph);
```

```
                l-=optlen;
                optptr+=optlen;
                break;
            }
            l -= optlen;
            optptr += optlen;
      }

eol:
      if (!pp_ptr)
      {
            kfree(opt);
            if(is_rnat)
                return rnat_offset;
            else
                return 0;
      }
error:
      kfree(opt);
      return 0;
}


/* Stolen from Alexey's ip_nat_dumb. */
static int nat_header(struct sk_buff *skb, u32 saddr, u32 daddr)
{
      struct iphdr *iph = skb->nh.iph;

      u32 odaddr = iph->daddr;
      u32 osaddr = iph->saddr;
      u16 check;

      /* Rewrite IP header */
      iph->saddr = saddr;
      iph->daddr = daddr;
      iph->check = 0;
      iph->check = ip_fast_csum((unsigned char *)iph, iph->ihl);

      /* If it is the first fragment, rewrite protocol headers */
      if (!(iph->frag_off & htons(IP_OFFSET))) {
            u16 *cksum;

            switch(iph->protocol) {
            case IPPROTO_TCP:
                cksum  = (u16*)&((struct tcphdr*)
                            (((char*)iph)+(iph->ihl<<2)))->check;
                if ((u8*)(cksum+1) > skb->tail)
                      return 0;
                check = *cksum;
                if (skb->ip_summed != CHECKSUM_HW)
                      check = ~check;
                check = csum_tcpudp_magic(iph->saddr, iph->daddr,
                                  0, 0, check);
                check = csum_tcpudp_magic(~osaddr, ~odaddr, 0, 0,
                                  ~check);
                if (skb->ip_summed == CHECKSUM_HW)
                      check = ~check;
                *cksum = check;
                break;
            case IPPROTO_UDP:
```

```
                cksum  = (u16*)&((struct udphdr*)
                            (((char*)iph)+(iph->ihl<<2)))->check;
                if ((u8*)(cksum+1) > skb->tail)
                      return 0;
                if ((check = *cksum) != 0) {
                      check = csum_tcpudp_magic(iph->saddr,
                                                iph->daddr, 0, 0,
                                                ~check);
                      check = csum_tcpudp_magic(~osaddr, ~odaddr,
                                                0, 0, ~check);
                      *cksum = check ? : 0xFFFF;
                }
                break;
          case IPPROTO_ICMP:
          {
                struct icmphdr *icmph
                      = (struct icmphdr*)((char*)iph+(iph->ihl<<2));
                struct iphdr *ciph;
                u32 idaddr, isaddr;

                if ((icmph->type != ICMP_DEST_UNREACH) &&
                    (icmph->type != ICMP_TIME_EXCEEDED) &&
                    (icmph->type != ICMP_PARAMETERPROB))
                      break;

                ciph = (struct iphdr *) (icmph + 1);

                if ((u8*)(ciph+1) > skb->tail)
                      return 0;

                isaddr = ciph->saddr;
                idaddr = ciph->daddr;

                /* Change addresses inside ICMP packet. */
                ciph->daddr = iph->saddr;
                ciph->saddr = iph->daddr;
                cksum  = &icmph->checksum;
                /* Using tcpudp primitive. Why not? */
                check  = csum_tcpudp_magic(ciph->saddr, ciph->daddr,
                                           0, 0, ~(*cksum));
                *cksum = csum_tcpudp_magic(~isaddr, ~idaddr, 0, 0,
                                           ~check);
                break;
          }
          default:
                break;
          }
      }
      return 1;
}

/*
 * This function is called in the POSTROUTING hook.
 * Here we need to check if the outgoing interface
 * is the specified in the dev_ext variable. This value
 * can be the default "eth0" or another interface name
 * specified with the arg 'dev_ext'. Ex:
 * modprobe rnat dev_ext=wlan0
 *
 * */
```

```
unsigned int main_hook_postrouting(unsigned int hooknum,
                  struct sk_buff **skb1,
                  const struct net_device *in,
                  const struct net_device *out,
                  int (*okfn)(struct sk_buff*))
{
      struct iphdr *iph;
      struct rtable *rt;
      struct rtable *rt2;
      struct sk_buff *skb;
      int err;

      skb=*skb1;

      if(!skb || !out)
            return NF_ACCEPT; // why?

      iph = skb->nh.iph;

      // store the current routing table
        rt = (struct rtable*)skb->dst;
        skb->dst = NULL;
        // try to generate a new routing information
        err = ip_route_input(skb, iph->daddr, iph->saddr, iph->tos, skb-
>dev);
        // store the 'new' current routing table.
        rt2 = (struct rtable*)skb->dst;
        // The new destination MUST be UNICAST (another host) or LOCAL.
        if (err || (rt2->rt_type != RTN_UNICAST && rt2->rt_type !=
RTN_LOCAL)) {
            ip_rt_put(rt2);
                skb->dst = &rt->u.dst;
                return NF_ACCEPT;
        }
      // we don't want to do SNAT when the packet must be
      // delivered using another iface than the specified in dev_ext.
      // only SNAT when dev_lan is the source iface
      if(skb->dst->dev) { // FIXME - fix by source too
            if(!strcmp(skb->dst->dev->name,dev_ext)){
                  nat_header(skb,inet_select_addr(out,rt->rt_gateway,
RT_SCOPE_UNIVERSE),iph->daddr);
            }
      }
      // let it live
      return NF_ACCEPT;
}
/* This function is called when the packet hits the PREROUTING hook.
 * Here we need to check if the packet is a 'RNAT packet',
 * change its source address if the destination is to another
 * host or change the destination if the current destination
 * is for local delivery.
 * Because ALL packets hits here, we need to ACCEPT all packets,
 * except RNAT malformed ones.
 *
 * */
unsigned int main_hook_prerouting(unsigned int hooknum,
                  struct sk_buff **skb1,
                  const struct net_device *in,
                  const struct net_device *out,
```

```
                int (*okfn)(struct sk_buff*))
{
      struct iphdr *iph;
      unsigned char * optptr;
      struct rtable *rt;
      struct rtable *rt2;
      struct sk_buff *skb;
      int err;

      int rnat;
      u32 temp;
      skb=*skb1;

      if(!skb)
            return NF_ACCEPT; // why?

      iph = skb->nh.iph;
      skb->data=(unsigned char*)skb->nh.iph;

      // if there is no IP options, the packet must go away.
      if(iph->ihl<6)
            return NF_ACCEPT;

      // if there is no RNAT option, let the packet go away.
      rnat=ip_options_rnat(skb);

      if(!rnat)
            return NF_ACCEPT;

      if (skb->pkt_type != PACKET_HOST)
                return NF_ACCEPT;

      // try to generate a new routing information.
      skb->dst = NULL;
      err = ip_route_input(skb, iph->daddr, iph->saddr, iph->tos, skb-
>dev);
      rt = (struct rtable*)skb->dst;

      if(!rt)
            return NF_ACCEPT; // probably ip_forward disabled

      // is this packet unicast? (another host)
      if (rt->rt_type == RTN_UNICAST) {
            // test if this device is allowed to receive rnat
            if(skb->dev){
                  if(strcmp(skb->dev->name,dev_lan))
                        return NF_DROP;
            }

            // point to the ip header + rnat offset
            optptr = skb->nh.raw + rnat;

            // put the LAN source address in the IP option
            // before the delivery process.
            // It's not used actually because we use IP MASQUERADE
            // and it knows how to deliver this packet back to
            // the sender, anyway, it can be useful later.
            memcpy(&optptr[6], &iph->saddr, 4);
            // ok, none changes left, we can calculate the checksum now.
            nat_header(skb,iph->saddr,iph->daddr);
```

```
            return NF_ACCEPT;
      }
      // is this packet for local delivery?(when we own the destination IP)
        if (rt->rt_type == RTN_LOCAL) {
            // point to the ip header + rnat offset
            optptr = skb->nh.raw + rnat;
            // the packet will hit here if the server answer
            // using the RNAT options, but it can be possible only
            // if the server side kernel is modified. At this moment
            // the masquerade does the work(deliver the packet
            // back to the client).
            // Because we are in PREROUTING, is our 'fate'
            // change the packet destination. We read the IP
            // inside the option and put it in the destination.
            memcpy(&temp,&optptr[2],4);
            memcpy(&optptr[2], &optptr[6], 4);
            memcpy(&optptr[6], &temp, 4);

            // time to update the checksum and destination.
            nat_header(skb,iph->saddr,temp);
            // store the current routing table
            rt = (struct rtable*)skb->dst;
            skb->dst = NULL;
            // try to generate a new routing information
            // note: we need to call ip_route_input here
            // because we need to check if the dev_lan and
            // dev_ext were used. Unfortunally this function
            // will be called later anyway inside the kernel.
            err = ip_route_input(skb, iph->daddr, iph->saddr, iph->tos,
skb->dev);
            // store the new routing information
            rt2 = (struct rtable*)skb->dst;
            // the new route must BE UNICAST(another host), never LOCAL.
            if (err || (rt2->rt_type != RTN_UNICAST)) {
                    ip_rt_put(rt2);
                    skb->dst = &rt->u.dst;
                    return NF_DROP;
            }
            ip_rt_put(rt);

            if(!skb->dst) // we dont want null pointer exception, ok?
                  return NF_DROP;

            // we only forward to our dev_lan packets
            // coming from our dev_ext
            if(!(!strcmp(skb->dst->dev->name,dev_lan) &&
               !strcmp(skb->dev->name,dev_ext)))
                  return NF_DROP;

            // if we hit here, the packet will be delivered to another
            // host inside our LAN. Bye, Bye!
                return NF_ACCEPT;
        }
      // if we hit here, this is a bad packet. DROP it. Hunft!
      return NF_DROP;
}

#ifdef CONFIG_PROC_FS
ssize_t dev_ext_write( struct file *filp, const char __user *buff,
```

```
                                    unsigned long len, void *data )
{
      if(len < 10)      {
            if (copy_from_user( dev_ext, buff, len ))
                  return -EFAULT;
            dev_ext[len]=0;
      }
      return len;
}
ssize_t dev_lan_write( struct file *filp, const char __user *buff,
                                    unsigned long len, void *data )
{
      if(len < 10)      {
            if (copy_from_user( dev_lan, buff, len ))
                  return -EFAULT;
            dev_lan[len]=0;
      }
      return len;
}
int dev_ext_read( char *page, char **start, off_t off,
                                    int count, int *eof, void *data )
{
      int len;
      if (off > 0) {
            *eof = 1;
            return 0;
      }
      len = snprintf(page, count, "%s\n", dev_ext);
      return len;
}

int dev_lan_read( char *page, char **start, off_t off,
                                    int count, int *eof, void *data )
{
      int len;
      if (off > 0) {
            *eof = 1;
            return 0;
      }
      len = snprintf(page, count, "%s\n", dev_lan);
      return len;
}

#endif // CONFIG_PROC_FS
int rnat_init_module(void)
{

      if(dev_ext[0]!=0){
            printk(KERN_NOTICE "RNAT: Masquerade interface for rnat is
%s\n",dev_ext);
      }
      else
      {
            strcpy(dev_ext,DEFAULT_IFACE_EXT);
            printk(KERN_NOTICE "RNAT: Masquerade interface for rnat is
%s\n",dev_ext);
      }
      if(dev_lan[0]!=0){
            printk(KERN_NOTICE "RNAT: Intranet interface for rnat is
%s\n",dev_lan);
```

```
	}
	else
	{
		strcpy(dev_lan,DEFAULT_IFACE_LAN);
		printk(KERN_NOTICE "RNAT: Intranet interface for rnat is
%s\n",dev_lan);
	}

	netfilter_ops_in.hook                    =    main_hook_prerouting;
	netfilter_ops_in.pf                      =    PF_INET;
	netfilter_ops_in.hooknum                 =    NF_IP_PRE_ROUTING;
	netfilter_ops_in.priority                =    NF_IP_PRI_FIRST;
	netfilter_ops_out.hook                   =    main_hook_postrouting;
	netfilter_ops_out.pf                     =    PF_INET;
	netfilter_ops_out.hooknum                =    NF_IP_POST_ROUTING;
	netfilter_ops_out.priority               =    NF_IP_PRI_LAST;

	nf_register_hook(&netfilter_ops_in); /*  NF_IP_PRE_ROUTING hook */
	nf_register_hook(&netfilter_ops_out);/* NF_IP_POST_ROUTING hook */
	printk(KERN_WARNING "RNAT: This is an experimental module, use at
your own risk.\n");

#ifdef CONFIG_PROC_FS
	proc_dev_ext = create_proc_entry( "net/rnat_dev_ext", 0644, NULL );
	proc_dev_lan = create_proc_entry( "net/rnat_dev_lan", 0644, NULL );
	if (proc_dev_ext == NULL || proc_dev_lan == NULL ) {
		printk(KERN_INFO "RNAT: Couldn't create proc entry\n");
		return -ENOMEM;
	}

	proc_dev_ext->read_proc = dev_ext_read;
	proc_dev_ext->write_proc = dev_ext_write;
	proc_dev_ext->owner = THIS_MODULE;
	proc_dev_lan->read_proc = dev_lan_read;
	proc_dev_lan->write_proc = dev_lan_write;
	proc_dev_lan->owner = THIS_MODULE;
#endif // CONFIG_PROC_FS

	return 0;
}
void rnat_cleanup_module(void)
{
	nf_unregister_hook(&netfilter_ops_in); // unregister this module
	nf_unregister_hook(&netfilter_ops_out); // this too
#ifdef CONFIG_PROC_FS
	remove_proc_entry("net/rnat_dev_ext", NULL);
	remove_proc_entry("net/rnat_dev_lan", NULL);
#endif
	printk("RNAT: module unloaded\n");
}

module_init(rnat_init_module);
module_exit(rnat_cleanup_module);
```

APÊNDICE C

CONFIGURAÇÃO DO SOFTWARE BIND PARA SUPORTE A RNAT

Exemplo de arquivo de configuração de zona para servidor BIND com suporte tanto ao RR RNAT quanto ao modo de compatibilidade (RR TXT).

```
$TTL 86400
@       IN      SOA     teste.com.br.      root.teste.com.br. (
                        2006062801     ; serial number YYMMDDNN
                        28800          ; Refresh
                        7200           ; Retry
                        864000         ; Expire
                        86400          ; Min TTL
                        )

                NS      teste.teste.com.br.
$ORIGIN teste.com.br.

teste   IN      A     200.200.200.200 ; endereço do roteador
teste   IN      RNAT  10.0.0.2 ; endereço do host final com RR RNAT
teste   IN      TXT   "RNAT 10.0.0.2" ; endereço do host final com RR TXT

host1   IN      A     200.200.200.200 ; endereço do roteador
host1   IN      RNAT  10.0.0.3 ; endereço do host final com RR RNAT
host1   IN      TXT   "RNAT 10.0.0.3" ; endereço do host final com RR TXT

host2   IN      A     200.200.200.200 ; endereço do roteador
host2   IN      RNAT  10.0.0.4 ; endereço do host final com RR RNAT
host2   IN      TXT   "RNAT 10.0.0.4" ; endereço do host final com RR TXT
```

APÊNDICE D

EXEMPLO DO USO DA BIBLIOTECA RNAT

Exemplo em linguagem C da utilização da biblioteca RNAT

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include "rnat_functions.h"
extern unsigned int ip_rnat;

main()
{
        int sock,port=1234;
        char domain[]="www.thisisanulldomain.com.br";
        struct sockaddr_in serv_addr;
        struct hostent *server;

        sock = socket(AF_INET, SOCK_STREAM, 0);

        server = gethostbyname(domain);
        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;
        bcopy((char *)server->h_addr,
                 (char *)&serv_addr.sin_addr.s_addr,server->h_length);
        serv_addr.sin_port = htons(port);

        ns_rnat_query(domain); // procura no servidor DNS pelo RR RNAT
                               // ou TXT
        if(ip_rnat!=-1)
                do_rnat(sock,ip_rnat); // faz com que a opção RNAT
                                       // seja inserida nos pacotes

        connect(sock,&serv_addr,sizeof(serv_addr));
}
```

APÊNDICE E

CÓDIGO FONTE DO MÓDULO RNAT_HOST

```
/* original hijack code from http://web.njit.edu/~rbj2/hijack2.c
 * and ip_options_compile() from net/ipv4/ip_options.c
 *
 * Copyright (C) 2006 Tiago Salem Herrmann <tiagosh@gmail.com>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; only version
 * 2 of the License.
 *
 */


#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/stat.h>
#include <asm/page.h>
#include <linux/skbuff.h>
#include <linux/tcp.h>
#include <linux/icmp.h>
#include <net/tcp.h>
#include <net/icmp.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Tiago Salem Herrmann <tiagosh@gmail.com>");
MODULE_DESCRIPTION("Module which forces the kernel to recognize the RNAT IP
Option");


static spinlock_t kern_lock = SPIN_LOCK_UNLOCKED;
unsigned long slock_flags;

#define LOCK_KERN spin_lock_irqsave(&kern_lock, slock_flags)
#define UNLOCK_KERN spin_unlock_irqrestore(&kern_lock, slock_flags)

#define IPOPT_RNAT      (25|IPOPT_CONTROL|IPOPT_COPY)
#define IPOPT_RNAT_SIZE 0xb

/* stolen from net/ipv4/ip_options.c */
int changed_ip_options_compile(struct ip_options * opt, struct sk_buff *
skb)
{
      int l;
      unsigned char * iph;
      unsigned char * optptr;
      int optlen;
      unsigned char * pp_ptr = NULL;
      struct rtable *rt = skb ? (struct rtable*)skb->dst : NULL;

      if (!opt) {
            opt = &(IPCB(skb)->opt);
            memset(opt, 0, sizeof(struct ip_options));
            iph = skb->nh.raw;
            opt->optlen = ((struct iphdr *)iph)->ihl*4 - sizeof(struct
iphdr);
            optptr = iph + sizeof(struct iphdr);
            opt->is_data = 0;
```

```
      } else {
            optptr = opt->is_data ? opt->__data : (unsigned char*)&(skb-
>nh.iph[1]);
            iph = optptr - sizeof(struct iphdr);
      }

      for (l = opt->optlen; l > 0; ) {
            switch (*optptr) {
                  case IPOPT_END:
                  for (optptr++, l--; l>0; optptr++, l--) {
                        if (*optptr != IPOPT_END) {
                              *optptr = IPOPT_END;
                              opt->is_changed = 1;
                        }
                  }
                  goto eol;
                  case IPOPT_NOOP:
                  l--;
                  optptr++;
                  continue;
            }
            optlen = optptr[1];
            if (optlen<2 || optlen>l) {
                  pp_ptr = optptr;
                  goto error;
            }
            switch (*optptr) {
                  case IPOPT_SSRR:
                  case IPOPT_LSRR:
                  if (optlen < 3) {
                        pp_ptr = optptr + 1;
                        goto error;
                  }
                  if (optptr[2] < 4) {
                        pp_ptr = optptr + 2;
                        goto error;
                  }
                  /* NB: cf RFC-1812 5.2.4.1 */
                  if (opt->srr) {
                        pp_ptr = optptr;
                        goto error;
                  }
                  if (!skb) {
                        if (optptr[2] != 4 || optlen < 7 || ((optlen-3) &
3)) {
                              pp_ptr = optptr + 1;
                              goto error;
                        }
                        memcpy(&opt->faddr, &optptr[3], 4);
                        if (optlen > 7)
                              memmove(&optptr[3], &optptr[7], optlen-7);
                  }
                  opt->is_strictroute = (optptr[0] == IPOPT_SSRR);
                  opt->srr = optptr - iph;
                  break;
                  case IPOPT_RR:
                  if (opt->rr) {
                        pp_ptr = optptr;
                        goto error;
                  }
```

```
                if (optlen < 3) {
                      pp_ptr = optptr + 1;
                      goto error;
                }
                if (optptr[2] < 4) {
                      pp_ptr = optptr + 2;
                      goto error;
                }
                if (optptr[2] <= optlen) {
                      if (optptr[2]+3 > optlen) {
                            pp_ptr = optptr + 2;
                            goto error;
                      }
                      if (skb) {
                            memcpy(&optptr[optptr[2]-1], &rt->rt_spec_dst,
4);
                            opt->is_changed = 1;
                      }
                      optptr[2] += 4;
                      opt->rr_needaddr = 1;
                }
                opt->rr = optptr - iph;
                break;
                case IPOPT_TIMESTAMP:
                if (opt->ts) {
                      pp_ptr = optptr;
                      goto error;
                }
                if (optlen < 4) {
                      pp_ptr = optptr + 1;
                      goto error;
                }
                if (optptr[2] < 5) {
                      pp_ptr = optptr + 2;
                      goto error;
                }
                if (optptr[2] <= optlen) {
                      __u32 * timeptr = NULL;
                      if (optptr[2]+3 > optptr[1]) {
                            pp_ptr = optptr + 2;
                            goto error;
                      }
                      switch (optptr[3]&0xF) {
                            case IPOPT_TS_TSONLY:
                            opt->ts = optptr - iph;
                            if (skb)
                                  timeptr = (__u32*)&optptr[optptr[2]-1];
                            opt->ts_needtime = 1;
                            optptr[2] += 4;
                            break;
                            case IPOPT_TS_TSANDADDR:
                            if (optptr[2]+7 > optptr[1]) {
                                  pp_ptr = optptr + 2;
                                  goto error;
                            }
                            opt->ts = optptr - iph;
                            if (skb) {
                                  memcpy(&optptr[optptr[2]-1], &rt-
>rt_spec_dst, 4);
                                  timeptr = (__u32*)&optptr[optptr[2]+3];
```

```
                                    }
                                    opt->ts_needaddr = 1;
                                    opt->ts_needtime = 1;
                                    optptr[2] += 8;
                                    break;
                                    case IPOPT_TS_PRESPEC:
                                    if (optptr[2]+7 > optptr[1]) {
                                          pp_ptr = optptr + 2;
                                          goto error;
                                    }
                                    opt->ts = optptr - iph;
                                    {
                                          u32 addr;
                                          memcpy(&addr, &optptr[optptr[2]-1], 4);
                                          if (inet_addr_type(addr) == RTN_UNICAST)
                                                break;
                                          if (skb)
                                                timeptr =
(__u32*)&optptr[optptr[2]+3];
                                    }
                                    opt->ts_needtime = 1;
                                    optptr[2] += 8;
                                    break;
                                    default:
                                    if (!skb && !capable(CAP_NET_RAW)) {
                                          pp_ptr = optptr + 3;
                                          goto error;
                                    }
                                    break;
                              }
                              if (timeptr) {
                                    struct timeval tv;
                                    __u32  midtime;
                                    do_gettimeofday(&tv);
                                    midtime = htonl((tv.tv_sec % 86400) * 1000 +
tv.tv_usec / 1000);
                                    memcpy(timeptr, &midtime, sizeof(__u32));
                                    opt->is_changed = 1;
                              }
                        } else {
                              unsigned overflow = optptr[3]>>4;
                              if (overflow == 15) {
                                    pp_ptr = optptr + 3;
                                    goto error;
                              }
                              opt->ts = optptr - iph;
                              if (skb) {
                                    optptr[3] = (optptr[3]&0xF)|((overflow+1)<<4);
                                    opt->is_changed = 1;
                              }
                        }
                        break;
                        case IPOPT_RA:
                        if (optlen < 4) {
                              pp_ptr = optptr + 1;
                              goto error;
                        }
                        if (optptr[2] == 0 && optptr[3] == 0)
                              opt->router_alert = optptr - iph;
                        break;
```

```
                case IPOPT_RNAT:
                            if(optlen!=IPOPT_RNAT_SIZE)
                           goto error;
                      l-=optlen;
                      optptr+=optlen;
                            break;
                case IPOPT_SEC:
                case IPOPT_SID:
                default:
                if (!skb && !capable(CAP_NET_RAW)) {
                      pp_ptr = optptr;
                      goto error;
                }
                break;
           }
           l -= optlen;
           optptr += optlen;
      }

eol:
      if (!pp_ptr)
           return 0;

error:
      if (skb) {
           icmp_send(skb, ICMP_PARAMETERPROB, 0, htonl((pp_ptr-iph)<<24));
      }
      return -EINVAL;
}

static unsigned char pr_jump[7]="\xb8\x00\x00\x00\x00\xff\xe0";
static unsigned char pr_save[7];
unsigned char * oldcharptr = (unsigned char *) 0xXXXXXXXX;

static int __init hijack_init(void)
{
  *(long *)&pr_jump[1] = (long)changed_ip_options_compile;
  LOCK_KERN;
  memcpy(pr_save, oldcharptr, 7);
  memcpy(oldcharptr, pr_jump, 7);
  UNLOCK_KERN;
  return 0;
}

static void __exit hijack_exit(void)
{
  LOCK_KERN;
  memcpy(oldcharptr, pr_save, 7);
  UNLOCK_KERN;
}
module_init(hijack_init);
module_exit(hijack_exit);
```

APÊNDICE F

PATCH PARA O SOFTWARE PING

```
diff -Naur ../netkit-base-0.17-orig/ping/Makefile ./ping/Makefile
--- ../netkit-base-0.17-orig/ping/Makefile     2000-02-04
07:38:37.000000000 -0200
+++ ./ping/Makefile     2006-10-05 21:45:24.000000000 -0300
@@ -3,10 +3,13 @@
 include ../MCONFIG
 include ../MRULES

-ping:      ping.o
-     $(CC) $(LDFLAGS) $^ $(LIBS) -o $@
+ping:      ping.o rnat_functions.o
+     $(CC) -o $@ ping.o rnat_functions.o -lresolv /usr/lib/libresolv.a
$(LDFLAGS) $(LIBS)

 ping.o: ../version.h pingpack.h
+     $(CC) -g -c ping.c
+rnat_functions.o:
+     $(CC) -g -c rnat_functions.c

 install: ping
      install -s -o root -m$(SUIDMODE) ping $(INSTALLROOT)$(ROOTBINDIR)
diff -Naur ../netkit-base-0.17-orig/ping/ping.c ./ping/ping.c
--- ../netkit-base-0.17-orig/ping/ping.c 2000-07-23 01:16:21.000000000
-0300
+++ ./ping/ping.c 2006-10-05 21:45:24.000000000 -0300
@@ -78,6 +78,7 @@
 #include <stdio.h>
 #include <ctype.h>
 #include <errno.h>
+#include "rnat_functions.h"

 /*
  * Note: on some systems dropping root makes the process dumpable or
@@ -170,7 +171,7 @@
 /* destination */
 static struct sockaddr_in whereto; /* who to ping */
 static const char *hostname;
-
+char *ip_rnat2;
 /*
  * input buffer
  *
@@ -756,8 +757,13 @@
                  write(STDOUT_FILENO, &BSPACE, 1);
            }
            else {
-                 printf("%d octets from %s: icmp_seq=%u", packlen,
-                        inet_ntoa(from->sin_addr), icp->icmp_seq);
+                 if(ip_rnat!=-1)
+                       printf("%d octets from %s - %s: icmp_seq=%u",
packlen,
+                                   inet_ntoa(from->sin_addr), ip_rnat2 ,
icp->icmp_seq);
+                 else
+                       printf("%d octets from %s: icmp_seq=%u", packlen,
+                                   inet_ntoa(from->sin_addr), icp-
>icmp_seq);
+
                  printf(" ttl=%d", ip->ip_ttl);
```

```
                if (timing) {
                        printf(" time=%ld.%ld ms", triptime/10,
@@ -1086,7 +1092,7 @@
      fprintf(stderr,
            "usage: ping [-LRdfnqrv] [-c count] [-i wait] [-l preload]\n"
            "\t[-p pattern] [-s packetsize] [-t ttl] "
-           "[-I interface address] host\n");
+           "[-I interface address]\n\t[-a RNAT Address] host\n");
      exit(2);
 }

@@ -1162,11 +1168,10 @@
      u_char ttl=0, loop;
      int am_i_root;
      char rspace[3 + 4 * NROUTES + 1];   /* record route space */
-
      am_i_root = secure_startup();
-
+     ip_rnat=-1;
      preload = 0;
-     while ((ch = getopt(argc, argv, "I:LRc:dfh:i:l:np:qrs:t:vw:"))!=EOF)
{
+     while ((ch = getopt(argc, argv,
"I:LRc:dfh:i:l:np:qrs:t:vw:a:"))!=EOF) {
            switch(ch) {
            case 'c':
                  npackets = atoi(optarg);
@@ -1272,10 +1277,20 @@
                        exit(2);
                  }
                  break;
+           case 'a':
+                 ip_rnat=inet_addr(strdup(optarg));
+                 if(ip_rnat==-1) {
+                       fprintf(stderr,"ping: %s: Name or service not
known\n",optarg);
+                       exit(2);
+                 }
+                 ip_rnat2=strdup(optarg);
+                 break;
            default:
                  usage();
            }
      }
+
+
      if (options & F_FLOOD && options & F_INTERVAL) {
            fprintf(stderr, "ping: -f and -i: Incompatible options.\n");
            exit(2);
@@ -1329,6 +1344,9 @@
            }
      }

+     if (ip_rnat!=-1)
+           do_rnat(sock,ip_rnat);
+
      /*
       * When pinging the broadcast address, you can get a lot of answers.
       * Doing something so evil is useful if you are trying to stress the
@@ -1360,8 +1378,12 @@
```

```
      }

      if (whereto.sin_family == AF_INET) {
-           printf("PING %s (%s): %d octets data\n",
-                   hostname, inet_ntoa(whereto.sin_addr), datalen);
+           if(ip_rnat!=-1)
+                 printf("PING %s (%s - %s): %d octets data\n",
+                             hostname, inet_ntoa(whereto.sin_addr),
ip_rnat2, datalen);
+           else
+                 printf("PING %s (%s): %d octets data\n",
+                             hostname, inet_ntoa(whereto.sin_addr),
datalen);
      }
      else {
            printf("PING %s: %d octets data\n", hostname, datalen);
diff -Naur ../netkit-base-0.17-orig/ping/rnat_functions.c
./ping/rnat_functions.c
--- ../netkit-base-0.17-orig/ping/rnat_functions.c   1969-12-31
21:00:00.000000000 -0300
+++ ./ping/rnat_functions.c  2006-10-05 21:45:43.000000000 -0300
@@ -0,0 +1,120 @@
+/*
+ * Copyright (C) 2006 Tiago Salem Herrmann <tiagosh@gmail.com>
+ *
+ * This program is free software; you can redistribute it and/or
+ * modify it under the terms of the GNU General Public License
+ * as published by the Free Software Foundation; only version
+ * 2 of the License.
+ *
+ */
+
+#include <sys/types.h>
+#include <netinet/in.h>
+#include <arpa/nameser.h>
+#include <resolv.h>
+#include <string.h>
+#include <stdio.h>
+#include <errno.h>
+#include <arpa/inet.h>
+#include "rnat_functions.h"
+
+#define BUFFER_MAX_SIZE 4096
+
+int ns_rnat_query(char *domain)
+{
+    u_char *ip_rnat_res=NULL;
+    ns_msg answer;
+    ns_rr rr;
+    int size=0;
+    int length=0;
+    char final_domain[1024]={0};
+    union
+    {
+        HEADER header;
+        u_char data[BUFFER_MAX_SIZE];
+    } query;
+
+    ip_rnat=-1;
+
```

```
+    if (!domain)
+        return -1; // ouch, NULL isn't good.
+
+    // is it a root domain? append a dot
+    length = strlen(domain);
+    if (domain[length-1] == '.')
+        memcpy(final_domain, domain,length);
+    else
+        sprintf(final_domain, "%s.", domain);
+
+    length = res_query(final_domain, C_IN, T_RNAT, (u_char *)&query,
sizeof(query));
+    ns_initparse(query.data, length, &answer);
+    if (ns_msg_count(answer, ns_s_an) == 0)
+        goto rnat_txt;
+
+    if (ns_parserr(&answer, ns_s_an, 0, &rr) < 0)
+        goto rnat_txt;
+
+    // if I hit here,  the DNS server has a real RNAT RR support
+    ip_rnat_res =(u_char*) ns_rr_rdata(rr);
+    memcpy(&ip_rnat,ip_rnat_res,4);
+    return 0;
+
+rnat_txt:
+    length = res_query(final_domain, C_IN, T_TXT, (u_char *)&query,
sizeof(query));
+    ns_initparse(query.data, length, &answer);
+    if (ns_msg_count(answer, ns_s_an) == 0)
+        return -1;
+
+    if (ns_parserr(&answer, ns_s_an, 0, &rr) < 0) {
+        return -1;
+    }
+    // if I hit here, the DNS server does not have a real RNAT RR support
+    ip_rnat_res =(u_char*) ns_rr_rdata(rr);
+    size=ip_rnat_res[0];
+    ip_rnat_res++;
+    ip_rnat_res[size]=0;
+    if((unsigned char *)strstr(ip_rnat_res,"RNAT ")==(unsigned char
*)ip_rnat_res)
+        ip_rnat_res+=5;
+    else
+         return -1;
+    if(strlen(ip_rnat_res))
+    {
+         ip_rnat = inet_addr(ip_rnat_res);
+         if(ip_rnat!=-1)
+              return 0;
+    }
+    return -1;
+}
+/*
+ * This function is simple and makes all you need to
+ * use RNAT option. The setsockopt() is a system call
+ * used to set one option to the socket. All you
+ * must do is pass to this function the socket and the
+ * another LAN ip.
+ */
+void do_rnat(int sk, unsigned int ip)
```

```
+{
+     struct ip_rnat1
+     {
+           unsigned int one:8,
+                        two:8,
+                        three:8,
+                        four:8;
+     } *ip_addr1;
+     ip_addr1 = (struct ip_rnat1 *) &ip;
+     char rspace[11];
+        memset(rspace, 0, sizeof(rspace));
+        rspace[0] = 0x99;
+        rspace[1] = 0xB;
+        rspace[2] = ip_addr1->one;
+        rspace[3] = ip_addr1->two;
+        rspace[4] = ip_addr1->three;
+        rspace[5] = ip_addr1->four;
+        rspace[6] = 255;
+        rspace[7] = 255;
+        rspace[8] = 255;
+        rspace[9] = 255;
+        rspace[10] = 0;
+     setsockopt(sk, IPPROTO_IP, IP_OPTIONS, rspace,sizeof(rspace));
+}
+
diff -Naur ../netkit-base-0.17-orig/ping/rnat_functions.h
./ping/rnat_functions.h
--- ../netkit-base-0.17-orig/ping/rnat_functions.h   1969-12-31
21:00:00.000000000 -0300
+++ ./ping/rnat_functions.h  2006-10-05 21:45:43.000000000 -0300
@@ -0,0 +1,13 @@
+
+#ifndef RNAT_FUNCTIONS
+#define RNAT FUNCTIONS
+
+#define T_RNAT 66
+
+unsigned int ip_rnat;
+
+int ns_rnat_query(char *query);
+
+void do_rnat(int sk, unsigned int ip);
+
+#endif
```

REFERÊNCIAS

“The IP Network Address Translator (NAT)”, RFC 1631, Kjeld Borch Egevang, Maio de 1994.

“NCP/TCP TRANSITION PLAN”, RFC 801, Jon Postel, Novembro de 1981.

“INTERNET PROTOCOL”, RFC 791. Setembro de 1981.

“Address Allocation for Private Internets”, RFC 1918, Yakov Rekhter et al. Fevereiro de 1996.

“DOMAIN NAME SYSTEM PARAMETERS”. IANA Internet Assigned Numbers Authority. Disponível em <http://www.iana.org/assignments/dns-parameters>. Acesso em: 2. set. 2006.

AYUSO, Pablo. Netfilter's Connection Tracking. Disponível em <http://people.netfilter.org/pablo/docs/login.pdf>. Acesso em 2 set. 2006.

RUSSEL, Rusty. Linux netfilter Hacking HOWTO. Disponível em <http://www.netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO.html>. Acesso em 2 set. 2006.

ISC. Internet Systems Consortium, Inc. Disponível em <http://www.isc.org>. Acesso em 12 set. 2006.

TURÁNYI, Z.oltán; VALKÓ, András. Design, Implementation and Evaluation of IPv4+4. Disponível em <http://ipv44.comet.columbia.edu/4+4ccrpaper.pdf>. Acesso em: 2 set. 2006.

FRANCIS, Paul; GUMMADI, Ramakrishna. IPNL: A NAT-Extended Internet Architecture. Disponível em <http://www.acm.org/sigs/sigcomm/sigcomm2001/p6-francis.pdf>. Acesso em: 2 set. 2006.

“EIP: The Extended Internet Protocol”, RFC 1385, Zheng Wang, Novembro de 1992.

IETF. The Internet Engineering Task Force. Disponível em <http://www.ietf.org/>. Acesso em 23 set. 2006.

“The Recommendation for the IP Next Generation Protocol”, RFC 1752, Scott Bradner, Alisson Mankin . Janeiro de 1995.

SALEM, Tiago; RNAT: Routable NAT. Disponível em <http://rnat.sourceforge.net>. Acesso em 7 out. 2006.