# Provisão de Serviços Inseguros Usando Filtros de Pacotes com Estados

*Marcelo Barbosa Lima*

**Dissertação de Mestrado**

Instituto de Computação
Universidade Estadual de Campinas

# Provisão de Serviços Inseguros Usando Filtros de Pacotes com Estados

**Marcelo Barbosa Lima**[1]

Julho de 2000

**Banca Examinadora:**

- Prof. Dr. Paulo Lício de Geus (Orientador)
- Prof. Dr. Adriano Mauro Cansian
  DCCE, Instituto de Biociências, Letras e Ciências Exatas, UNESP
- Prof. Dr. Ricardo Dahab
  Instituto de Computação, UNICAMP
- Prof. Dr. Célio Cardoso Guimarães (Suplente)
  Instituto de Computação, UNICAMP

1. financiado por CNPq

# Provisão de Serviços Inseguros Usando Filtros de Pacotes com Estados

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Marcelo Barbosa Lima e aprovada pela Banca Examinadora.

Campinas, Outubro de 2000

Paulo Lício de Geus (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

# Resumo

As tecnologias mais tradicionais de *firewall* - filtros de pacotes e agentes *proxy* - apresentam algumas limitações que dificultam o suporte a certos serviços e protocolos TCP/IP, que neste trabalho são denominados de "serviços inseguros". Em geral, a única alternativa oferecida, para eliminar os riscos e prover um ambiente mais seguro, é o simples bloqueio incondicional de todo tráfego para estes serviços. Serviços baseados em RPC e UDP são exemplos importantes desta classe.

Entretanto, um grande número de novos e interessantes serviços—em sua maioria baseados em UDP e portanto inseguros—têm surgido muito rapidamente com o sucesso da Internet. Faz-se necessário, portanto, o uso de novas estratégias e/ou tecnologias para permitir a utilização de tais serviços sem comprometer a segurança do sistema. Este trabalho tem o objetivo de mostrar que filtros de pacotes com estados são uma excelente solução para a provisão destes serviços inseguros. Como aplicação do estudo realizado neste trabalho, são apresentados detalhes da implementação de um filtro de pacotes com estados capaz de suportar serviços baseados em RPC e que será oficialmente incorporado ao *kernel* do Linux (série 2.4).

# Abstract

Most tradicional firewall technologies, such as packet filters and proxy agents, present some limitations that make it difficult to support some TCP/IP services and protocols, hereafter called "unsecure services". In general, the only alternative offered to eliminate the risks and to provide a more secure environment, is simply to unconditionally deny all traffic for these services. UDP and RPC based services are important examples of this class of services.

However, a great deal of new and interesting services—mainly UDP based and as such unsecure—have appeared lately with the Internet's success. It is becoming necessary, therefore, the use of newer strategies and/or technologies to allow these services without jeopardizing the security of the network. The goal od this work is to show that stateful packet filters are an excellent solution for the provision of such unsecure services. As an application of this study, details of a stateful packet filter implementation are presented, one that is able to support RPC-based services, and that will be officially incorporated into the Linux kernel (2.4 series).

*Dedico aos meus pais, meus grandes amigos, companheiros e incentivadores, sem os quais este trabalho não teria sido realizado.*

# Agradecimentos

Ao professor Paulo Lício de Geus, por toda orientação, apoio e atenção fornecidas durante o desenvolvimento deste trabalho.

Ao Rusty Russel pelo apoio, esclarecimentos e sugestões que facilitaram o desenvolvimento de implementação neste trabalho.

Aos colegas do Instituto de Computação.

Ao CNPq pelo financiamento ao trabalho.

A Deus, pela oportunidade de realização de mais este importante passo em minha vida.

# Conteúdo

# Lista de Figuras

# Lista de Tabelas

**Ficha Catalográfica elaborada pela Biblioteca Central da UNICAMP**

Lima, Marcelo Barbosa
L628p Provisão de serviços inseguros usando filtros de pacotes com estados/ Marcelo Barbosa Lima. – Campinas, SP: [s.n.], 2000.

Orientador: Paulo Lício de Geus.
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Redes de computadores - Medidas de Segurança. 2. TCP/IP (Protocolos de rede de computação). I. Geus, Paulo Lício de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título

# Capítulo 1

# Introdução

Na década de 60, no auge da guerra fria, o DoD (Departamento de Defesa Americano) tinha o projeto de construir uma rede de telecomunicações capaz de sobreviver a uma guerra nuclear. As redes telefônicas tradicionais eram consideradas muito vulneráveis. Deste esforço, surgiu uma rede de computadores comutada por pacotes conhecida como ARPANET. A grande vantagem desta rede era a redundância, permitindo que falhas em alguns nós intermediários não significassem o fim de todas as comunicações entre pares de participantes da rede, ou seja, em casos de falhas envolvendo uma determinada rota, haveria provavelmente uma outra alternativa de roteamento para os pacotes.

Nos anos 70, surgiu a arquitetura de protocolos TCP/IP cujo objetivo era facilitar a interconexão das diferentes tecnologias físicas de rede que seriam usadas pela ARPANET, que nesta época ainda era uma rede relativamente pequena com alguns poucos computadores militares localizados nos EUA e em alguns países aliados, além de alguns outros em universidades americanas engajadas no projeto. O TCP/IP nasceu com poucos mecanismos de segurança nativos, mas isso não era problema para a ARPANET, visto que todas as máquinas na rede eram conhecidas e confiáveis entre si. Porém, graças às facilidades de interconexão trazidas pelo TCP/IP e ao fim da guerra fria, diversas outras redes menores (inclusive não militares) começaram a fazer parte da ARPANET, até chegarmos ao que hoje é a Internet.

A popularização e crescimento da Internet trouxeram a exploração comercial da rede e o surgimento de vários outros protocolos e serviços sobre a suíte TCP/IP original. Entretanto, também transformaram a rede em um ambiente inseguro. Por isso, os protocolos e serviços TCP/IP apresentam sérios problemas para esta nova funcionalidade. Além disso, qualquer computador

conectado à rede pode ser alvo de ataques de pessoas localizadas nos mais remotos lugares e, com isso, problemas de segurança em sistemas operacionais e demais *softwares* podem ser uma porta de entrada para estes atacantes. O resultado disto foi um esforço em busca de soluções para amenizar estes problemas de segurança e adequar a rede à esta nova realidade: *firewalls*, adoção de criptografia a serviços e protocolos (uso do PGP, por exemplo), dentre outras. Há ainda propostas e padrões para a criação de um modelo de arquitetura TCP/IP segura: DNSsec (DNS Seguro) e IPsec (Arquitetura IP segura) são exemplos.

O escopo deste trabalho é baseado exclusivamente em soluções de *firewalls*. A grosso modo, *firewalls* são elementos estruturais introduzidos na arquitetura TCP/IP com intuito específico para agregar segurança à Internet. Este trabalho tem como objetivo apresentar a mais nova geração de filtro de pacotes, filtro de pacotes com estados (*Stateful Packet Filter*, SPF), comparando-a com as tecnologias mais tradicionais de *firewall* (filtros de pacotes tradicionais e agentes *proxy*) e mostrando que filtros de pacotes com estados podem oferecer um tratamento mais conveniente para certos serviços e protocolos (que será definido, neste trabalho, como serviços inseguros). Para estes serviços e protocolos, as tecnologias mais tradicionais não oferecem outras alternativas além do simples bloqueio incondicional de todo tráfego, o que pode ser inaceitável em muitos casos. Entre os principais representantes destes serviços e protocolos encontramos os serviços baseados em RPC e UDP. Reconhecidamente, estes são problemas para as tecnologias tradicionais.

Atualmente, têm surgido vários novos protocolos e serviços que fazem uso do protocolo de transporte UDP, principalmente aqueles baseados em comunicação em tempo real na Internet: ICQ, *Netmeeting*, *CU-SeeMe* e similares. Além disso, ele é preferido ao TCP em sessões de curta duração com poucos dados envolvidos ou para eliminar o custo computacional associado ao processo de estabelecimento e encerramento de conexões. Portanto, a filtragem de todo tráfego UDP pode não ser sempre a melhor solução e será visto, ao longo do trabalho, que um SPF pode solucionar este problema.

Em resumo, o capítulo 2 apresenta as diversas vulnerabilidades existentes na arquitetura de protocolos TCP/IP e discute sobre falhas de segurança nas implementações dos sistemas operacionais e dos *softwares* que implementam serviços e protocolos TCP/IP, servindo como uma motivação para o uso de tecnologias de *firewall*. No capítulo 3 serão vistas as vantagens e desvantagens de cada uma das tecnologias tradicionais de *firewall*. Além disso, filtros de pacotes

com estados serão brevemente apresentados. O capítulo 4 traz uma discussão sobre serviços inseguros dentro do escopo das tecnologias tradicionais de *firewall*.

O capítulo 5 trata de filtros de pacotes com estados e mostra como eles podem lidar com estes protocolos e serviços vistos no capítulo 4. Esta nova geração de filtros de pacotes também será comparada com cada uma das tecnologias mais tradicionais discutidas no capítulo 3, revelando que, em geral, agentes *proxy* ainda são considerados soluções mais seguras.

Um mecanismo de filtragem de pacotes com estados foi implementado no *kernel* do Linux (versão 2.3.* em diante), cuja implementação e funcionamento são detalhados no capítulo 6. No capítulo 7 serão mostradas as idéias usadas na implementação de um filtro com estados que oferece suporte a serviços baseados em RPC. Será visto que este novo filtro foi construído usando o filtro analisado no capítulo 6 e aprovado para a inclusão na distribuição oficial de Linux, especificamente na funcionalidade de *firewall* dos *kernels* 2.3.x (desenvolvimento) e 2.4.x (produção). Finalmente, o capítulo 8 traz uma conclusão do trabalho.

# Capítulo 2

# Vulnerabilidades dos protocolos TCP/IP

Neste capítulo serão abordadas algumas das muitas vulnerabilidades que podem ser encontradas na arquitetura de protocolos TCP/IP. Além disso, serão discutidas também algumas falhas de segurança comuns em implementações dos diversos serviços e protocolos TCP/IP em sistemas operacionais. O objetivo deste capítulo é mostrar o porquê da necessidade de prover mecanismos e modelos de segurança capazes de amenizar os efeitos destas vulnerabilidades, motivando, desta forma, o uso de tecnologias de *firewall*, que serão tratadas detalhadamente nos próximos capítulos. Vale ressaltar ainda que a arquitetura TCP/IP evolui continuamente, novos protocolos surgem sobre esta suíte herdando os antigos problemas e adicionando outros particulares; algumas das suas antigas vulnerabilidades são corrigidas e, em paralelo, há trabalhos almejando um modelo de arquitetura TCP/IP segura que não são tratados no presente trabalho.

## 2.1 Introdução

Como já foi mencionado no capítulo anterior, a arquitetura de protocolos TCP/IP surgiu sem maiores preocupações com segurança. Em uma rede relativamente pequena, como era o caso da ARPANET, com algumas máquinas que honram os detalhes previstos nos padrões dos protocolos, segurança não é certamente a preocupação principal durante a especificação dos protocolos de rede. Mas a realidade que encontramos hoje na Internet é muito diferente. A Internet cresceu, continua crescendo muito rapidamente e deixou de ser um ambiente confiável onde se pode garantir que todos os computadores participantes da rede seguem perfeitamente as regras adotadas por estes padrões pré-estabelecidos. Além disso, o seu papel fundamental está sofrendo grandes e rápidas transformações, devido principalmente ao advento do comércio eletrônico e

à exploração com fins lucrativos da rede como um todo. Portanto, é necessário um esforço em busca de mecanismos e modelos que possam garantir um ambiente com um enfoque mais voltado à segurança para suportar estas novas funcionalidades da rede, o que não é encontrado na suíte TCP/IP original.

Uma das principais deficiências dos protocolos e serviços TCP/IP é usar esquemas baseados em endereços IP para autenticar máquinas na rede [1][4]. A autenticação é feita com base no endereço IP de origem de um pacote recebido, todavia é impossível determinar com certeza a identidade da máquina que o tenha originado. Isso torna muito fácil para atacantes personificarem outras máquinas da rede visando, por exemplo, abusar da relação de confiança entre algumas máquinas, em uma rede local, para obter informações sigilosas [59][60]. Esta técnica de personificação é conhecida como IP *spoofing*[1] e, geralmente, é utilizada com o objetivo de dificultar o rastreamento da verdadeira origem de um ataque, servindo como base para ataques mais sofisticados. Além deste grave problema, TCP/IP não oferece muitas garantias de que a privacidade dos dados dos pacotes tenha sido preservada ou de que estes dados não tenham sido alterados quando em trânsito na rede [1]. Estes dois últimos problemas podem ser resolvidos com o estabelecimento de canais seguros, fazendo uso de técnicas e algoritmos de criptografia, entre os participantes. Canais seguros estão fora do escopo deste trabalho. O leitor mais interessado pode buscar informações no trabalho de Keesje Duarte Pouw [33], que trata principalmente dos problemas relativos ao canal de comunicação.

Serão vistos nas próximas seções, maiores detalhes de problemas associados às diversas camadas da arquitetura TCP/IP e falhas comuns nas suas implementações em sistemas operacionais.

## 2.2 Vulnerabilidades do meio físico

A arquitetura TCP/IP não define nenhuma tecnologia básica no nível físico [22]. As tecnologias de redes locais apresentam problemas inerentes à sua própria natureza. Ataques que exploram estas vulnerabilidades são normalmente restritos às próprias redes locais, mas também podem ser usados como base para outros ataques a protocolos de mais alto nível na hierarquia TCP/IP.

---

1. Vale ressaltar que, em IP *spoofing*, um atacante não obterá diretamente as respostas da máquina alvo. Para este fim, deve-se usar alguma outra técnica para forçar que todo o tráfego de pacotes seja roteado para a máquina do atacante. Isto não é necessário se a localização do atacante facilitar a captura destes pacotes.

### 2.2.1 Grampeando a rede

Em tecnologias de rede onde o meio físico é compartilhado, como numa rede Ethernet, é possível configurar a interface de rede de uma máquina no modo "promíscuo", tornando possível a captura de todos os quadros transmitidos no meio [17]. Em ambientes UNIX, isso só pode ser feito com privilégios de super-usuário, mas em outros sistemas operacionais, como DOS e Windows, não existe nenhuma restrição de acesso à interface de rede. Programas de captura de pacotes ou *sniffers* usam estas características e o fato de não haver cifragem nos dados dos datagramas IP para obter informações privilegiadas, como por exemplo senhas dos usuários numa rede local.

### 2.2.2 ARP: Falso mapeamento em redes Ethernet

O protocolo ARP (*Address Resolution Protocol*) é responsável pelo mapeamento de endereços IP em endereços físicos [22]. Basicamente, ARP trabalha enviando uma mensagem de *broadcast* perguntando qual o endereço Ethernet (físico) da interface que possui um determinado endereço IP. A máquina que tiver este endereço IP procurado, ou alguma outra configurada para responder requisições ARP para ela (*Proxy* ARP), responde informando o endereço físico correspondente. O risco surge com a possibilidade de alguma máquina mal intencionada responder às requisições ARP com respostas falsas (ARP *spoofing*), desviando todo o tráfego para si [4].

## 2.3 UDP (*User Datagram Protocol*)

UDP é um protocolo de transporte simples, não orientado a conexões[1], sem estados, não confiável (não há retransmissões, tratamento de datagramas duplicados nem reordenamento dos datagramas, por exemplo) e que simplesmente oferece para a aplicação um serviço semelhante ao oferecido pelo nível IP [9][22]. Até mesmo a checagem do *checksum* em UDP é opcional. Além disso, UDP não oferece nenhum mecanismo para o controle de fluxo, podendo gerar congestionamento na rede ou inundação de uma máquina mais lenta e, por conseqüência, um aumento no número de pacotes perdidos na rede. Como vantagens do UDP temos a eliminação do custo computacional relacionado ao estabelecimento, controle e encerramento das conexões. É, por-

1. Como UDP é um protocolo não orientado a conexões, o leitor poderá observar, ao longo deste trabalho, o uso da palavra "sessão" ao invés de "conexão" quando UDP for o protocolo de transporte usado em uma comunicação.

tanto, preferido em aplicações onde desempenho é mais importante, como no caso do NFS (*Network File System*) [22].

Por não existir procedimento para o estabelecimento de conexão nem números de seqüência como no TCP, aplicações baseadas em UDP são mais suscetíveis à falsificação de endereços IP [4][17]. Consequentemente, pacotes UDP são muito interessantes para ataques de negação de serviço (DoS, *Denial of Service*) usando inundação de pacotes (*flooding*). Notar que ataques de negação de serviço não necessariamente dependem de IP *spoofing*, mas o uso de endereços forjados permite que o atacante esconda a origem do ataque e, até mesmo, dificulte a detecção de DoS por redes que usem sistemas de detecção de intrusão [72]. O UDP *flooding* consiste em enviar rapidamente uma grande quantidade de pacotes UDP para uma máquina alvo [46], escondendo a verdadeira origem do ataque com o IP *spoofing*. Desta forma, o alvo começa a descartar pacotes pela impossibilidade de tratar todas as requisições, negando serviço. Algumas aplicações baseadas em UDP, que mantém estados de suas *queries* por intermédio de números seqüenciais, podem ser menos vulneráveis a ataques de IP *spoofing*, desde que estes números sejam gerados aleatoriamente [17].

Será visto mais adiante, Seção 4.2, página 39, que estas aplicações, que usam UDP como protocolo de transporte, são difíceis de tratar com as tecnologias tradicionais de *firewall*. Na Seção 5.2, página 48, será discutido como filtros de pacotes com estados podem resolver esta limitação das tecnologias tradicionais para o tratamento de aplicações baseadas em UDP.

## 2.4 TCP (*Transport Control Protocol*)

TCP, ao contrário do UDP, é orientado a conexões e confiável (trata da perda de pacotes, retransmissões e reordenação dos pacotes, por exemplo). Além disso, TCP é mais bem comportado em redes congestionadas e na comunicação entre máquinas com grande diferença em desempenho (tem mecanismos próprios para controle de fluxo). Para que estas características sejam possíveis, este protocolo de transporte mantém estados de todas as conexões em andamento. O controle das conexões é feito, principalmente, com o uso de números de seqüência e números de reconhecimento (*acknowledgment numbers*) que são introduzidos no cabeçalho TCP dos pacotes [9][22]. Em conseqüência, TCP é menos sujeito a IP *spoofing* que UDP e ICMP. Entretanto, na próxima seção será visto que, principalmente em algumas implementações antigas, estes núme-

ros de seqüência são fáceis de prever e, com isso, esta proteção extra pode ser sobrepujada sem grande esforço [59].

### 2.4.1 Ataques de DoS ao TCP

Apesar da maior proteção oferecida pelos números de seqüência e pela manutenção dos estados das conexões, o primeiro pacote TCP enviado no processo de estabelecimento da conexão (*three-way handshaking*) pode facilmente sofrer *spoofing*; existem alguns ataques de negação de serviço que se aproveitam deste fato. Um dos exemplos mais interessantes é o ataque de SYN *flooding* [81]. No processo de estabelecimento de uma conexão [22], uma máquina recebendo um pacote TCP com a *flag* SYN ligada, deve responder com um pacote SYN/ACK à máquina que iniciou a conexão. A conexão é finalmente estabelecida quando a máquina, que enviou o SYN e recebeu o SYN/ACK, enviar um ACK, reconhecendo o pacote SYN/ACK recebido (ver Figura 2.1). Deve-se observar também, neste processo, a importância dos números de seqüência iniciais usados por cada máquina, mas este detalhe não será relevante nos ataques aqui descritos.

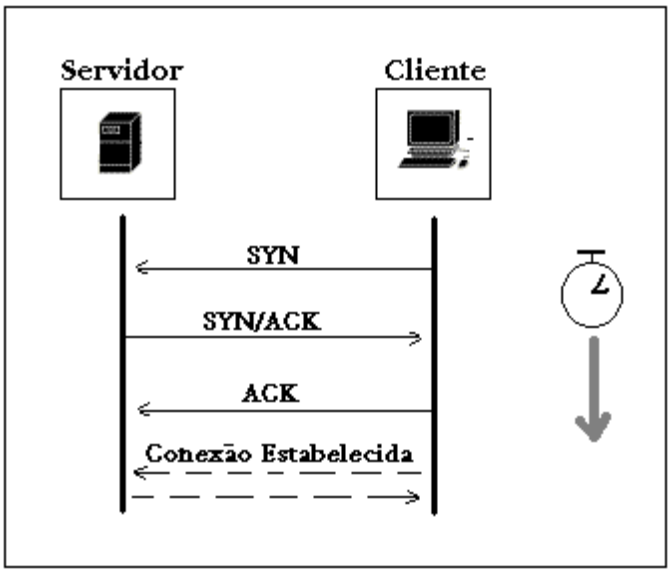


**Figura 2.1:** Three-Way Handshaking

No ataque de SYN *flooding*, o atacante envia uma série de pacotes, geralmente com o endereço de origem alterado para um endereço inexistente, para a máquina alvo com a *flag* SYN

ligada, indicando o pedido de conexão. Assim que estes pacotes são vistos pela máquina alvo, ela envia os respectivos pacotes com a *flag* SYN/ACK ligada e fica esperando as confirmações para estes pacotes que nunca virão. Neste caso, as conexões ficam em estado pendente na *cache* de conexões semi-abertas do *kernel* do sistema operacional da máquina alvo, evitando-se, possivelmente, que outras conexões possam ser estabelecidas. Eventualmente estas entradas da *cache* serão removidas (expiração de *timeouts*), mas um atacante pode ficar gerando novos pacotes SYN indefinidamente e rapidamente.

Existem algumas soluções [81] para tentar evitar os efeitos deste tipo de ataque: aumentar o tamanho da *cache* de conexões pendentes do *kernel*, diminuir o tempo de espera por confirmações para o estabelecimento das conexões e a implementação de SYN *cookies* em sistemas operacionais são alguns exemplos. As duas primeiras soluções somente aumentam o grau de dificuldade para o sucesso do ataque, mas não conseguem eliminar o problema por completo. O uso de SYN *cookies*, por sua vez, é uma solução bastante interessante para evitar este tipo de ataque.

Nesta solução [81], um servidor, quando recebe um pacote com a *flag* SYN ligada, fazendo o pedido de conexão, gera um *cookie* que será transmitido no pacote SYN/ACK resposta, mais especificamente codificado no campo reservado ao seu número de seqüência inicial (ISN, *Inicial Sequence Number*).

Na geração deste *cooki*e, é usado um algoritmo que calcula uma função *hash* criptográfica, como o MD5, tendo como entradas as informações obtidas no cabeçalho TCP do pacote recebido com o pedido de conexão (endereço e porta de origem, número de seqüência usado pela máquina de origem, endereço e porta de destino) e um segredo (conhecido somente por este servidor). Notar que estas mesmas informações, copiadas dos cabeçalhos deste pacote SYN, seriam mantidas na cache de conexões semi-abertas no *kernel* no modo de operação normal. Porém, com este novo esquema, estas informações simplesmente geram um código *hash*, com no máximo 32 bits[1], que é transmitido pela rede codificado junto ao número de seqüência inicial usado pelo servidor na nova conexão. O segredo é usado para que o servidor possa validar um *cookie* que ele tenha criado anteriormente.

1. Tamanho do campo reservado para números de seqüência.

Com o uso do *cookie*, nenhuma ou pouca informação precisa ser mantida no *kernel* do servidor sobre estas conexões semi-abertas. Desta forma, uma tentativa de ataque de SYN *flooding* não causará nenhum prejuízo. O *cookie*, junto ao número de seqüência, será incrementado pelo cliente e retornará ao servidor no campo reservado ao número de reconhecimento do pacote de ACK, concluindo o processo de *three-way handshaking*. Portanto, para o cliente tudo ocorrerá de maneira transparente, sem a necessidade de mudanças no protocolo TCP original. Este pacote ACK do cliente, ao chegar ao servidor, é verificado facilmente, usando a mesma função *hash,* as informações dos cabeçalhos no pacote e o segredo do servidor. Caso o código *hash* obtido neste processo seja idêntico ao *cookie* retornado pelo cliente, pode-se concluir que trata-se de um pacote ACK estabelecendo uma conexão. As informações relativas à esta nova conexão são extraídas deste pacote, mantidas na tabela de conexões do *kernel* do sistema operacional e a conexão prossegue normalmente. Caso contrário, o pacote será rejeitado e um pacote RST transmitido, como definido pelo protocolo TCP [22]. Vale mencionar aqui que o tratamento oferecido aos demais pacotes ACK do cliente é diferente, pois já existirá uma conexão estabelecida para eles, ou seja, já existirá uma entrada associada a estes pacotes na tabela de conexões. Atualmente esta solução tem sido implementada em muitos sistemas operacionais. SunOS e Linux são bons exemplos.

Um outro ataque de negação de serviço, usando os pacotes TCP de pedido de conexão, que merece destaque, é o Land [6]. Este ataque explora a facilidade de *spoofing* e um problema na implementação da pilha de protocolos TCP/IP existente em alguns sistemas operacionais. Basicamente, consiste em enviar para uma máquina alvo um pacote com SYN ligado (pedido de conexão) usando o endereço desta máquina como endereços de origem e destino e um mesmo número de porta como portas de origem e destino. Quando a máquina tenta responder ao pedido de conexão, ela gera um *loop* "derrubando" o sistema operacional.

### 2.4.2 Seqüestro de Sessão (*Session Hijacking*)

Em um ataque de seqüestro de sessão, um atacante procura por uma conexão TCP já existente entre dois hosts e tenta tomar o controle sobre ela [4]. Vale notar aqui, a necessidade de que o atacante esteja estrategicamente localizado, inspecionando todo fluxo de dados da comunicação. Para tal, ele pode tomar o controle de uma máquina (roteador ou um *firewall*) pela qual passa todo tráfego de pacotes desta conexão ou tentar desvirtuar roteadores (Seção 2.7, página 14),

criando rotas para o ataque. Isso torna difícil a implementação deste ataque. Ao atacante é requerida também a habilidade de acompanhar e controlar (dessincronizar) toda a sessão[85] ou, simplesmente, negar serviço a uma das partes participantes, usando pacotes ICMP (como será visto na Seção 2.6, página 12) ou pacotes TCP forjados com a *flag* RST ligada e roubar por completo a sessão.

A proteção contra este tipo de ataque é extremamente difícil. Até mesmo mecanismos de autenticação mais rígidos nem sempre têm êxito na tentativa de evitar tais ataques. O uso de um bom esquema de autenticação e canais seguros combinados podem ser uma boa solução para evitar estes ataques.

## 2.5 Ataques a Números de Seqüência (*Sequence Number Atacks*)

Para manter estados de suas conexões e sessões, alguns protocolos recorrem a números de seqüência. Tais números permitem que os pacotes sejam ordenados logicamente e o protocolo siga corretamente a especificação de alguma máquina de estados bem definida. Estes protocolos são, em geral, menos vulneráveis a IP *spoofing*. Obviamente, esta segurança extra somente existirá caso estes números usados não possam ser facilmente previstos[1]. O grande problema é que muitos dos protocolos de aplicação não têm esta preocupação. Um exemplo clássico é o caso de muitos *resolvers* (DNS) que sempre iniciam suas *queries* com um identificador igual a zero [24][45].

TCP também usa números de seqüência para manter estados das suas conexões. Quando um cliente envia um pacote SYN a um servidor, este pacote carrega também o número de seqüência inicial (ISN) que será usado pelo ciente nesta conexão. O pacote SYN/ACK correspondente, transmitido pelo servidor, transporta o ISN usado pelo servidor nesta nova conexão e reconhece o número de seqüência usado pelo cliente. Finalmente, a conexão é estabelecida quando o cliente retorna ao servidor um outro pacote ACK reconhecendo o número de seqüência inicial usado pelo servidor (ver Figura 2.2). Com a conexão estabelecida, os números de seqüência são usados para garantir a ordem de transferência dos dados para aplicação [9][22].

---

1. Como, em IP *spoofing,* o atacante não é capaz de obter as respostas enviadas pela máquina alvo, pode ser necessário prever o número de seqüência usado por esta máquina para o sucesso de um ataque.

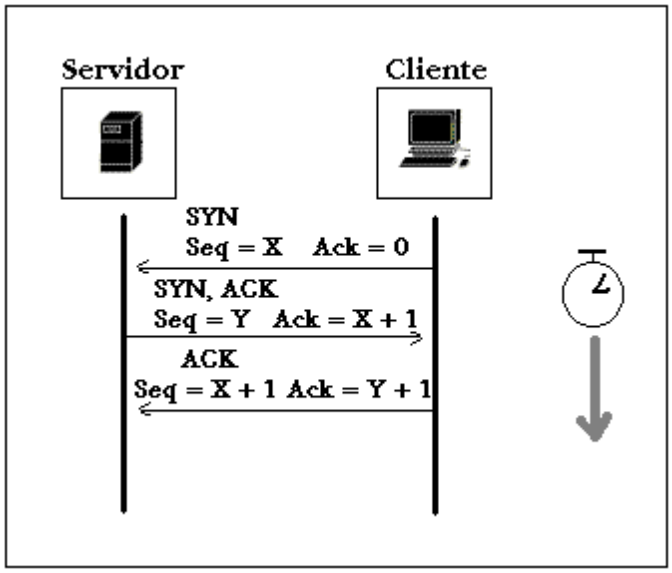


**Figura 2.2:** Números de Seqüência no TCP

Entretanto, implementações de TCP no *kernel* de alguns sistemas operacionais usam algoritmos simples para geração destes números e, portanto, permitem que atacantes facilmente descubram qual será o próximo número de seqüência inicial utilizado por uma máquina [59]. Apesar de previsto por Steve Bellovin [1] em 1989, somente após o Natal de 1994 este ataque se tornou famoso, quando Kevin Mitnick usou a predição dos números de seqüência usados em uma determinada implementação do TCP para invadir computadores de Tsutomu Shimomura. A Figura 2.3, página 13, mostra o esquema do ataque realizado pelo Mitnick usando IP *spoofing* para explorar, usando comandos "r" de Berkeley, a relação de confiança entre máquinas em uma rede local [60].

## 2.6 ICMP (*Internet Control Message Protocol*)

O ICMP é um protocolo geralmente considerado parte do nível IP [22], que influencia diretamente no comportamento das conexões TCP e sessões UDP. Ele pode informar qual a melhor rota para enviar pacotes a um certo destino, reportar problemas com uma rota, terminar uma conexão por detectar problemas na rede, entre outras funções de controle.

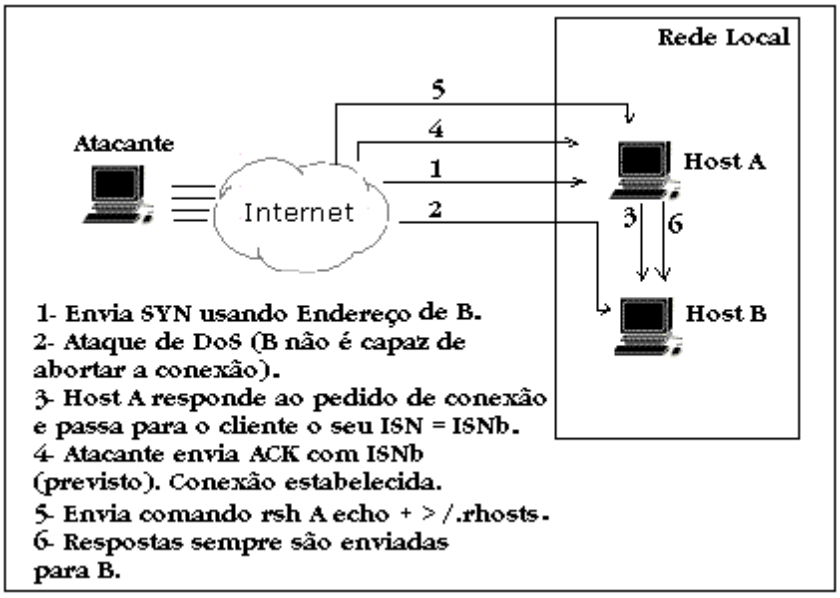


**Figura 2.3:** Ataque de IP *spoofing*

Pelas mesmas razões de UDP, pacotes ICMP são muito sujeitos a IP *spoofing* e são comuns ataques usando ICMP *flooding*. Um tipo de ataque interessante de ICMP *flooding* implementado com mensagens ICMP de *echo request* e *echo reply* (as mesmas usadas pelo programa Ping) é um ataque conhecido como "Smurf". Este ataque [12] utiliza o fato de que ao receber um pacote ICMP do tipo *echo request* outra mensagem do tipo *echo reply* é retornada à máquina de origem pela máquina destino. Portanto, uma mensagem de *echo request* pode ser enviada para o endereço de uma sub-rede, tendo o endereço de origem alterado para uma determinada máquina alvo. Neste caso, todas as máquinas da sub-rede irão responder com uma mensagem do tipo *echo reply* para a máquina alvo, inundando a máquina ou rede alvos. Existem ainda variações mais poderosas deste ataque. Além disso, por reportar falsos problemas com rotas, pacotes ICMP forjados podem abortar conexões válidas [46]. Este tipo de ataque é conhecido popularmente com o nome de "Nuke".

Muitas mensagens ICMP recebidas em uma dada máquina são atreladas a uma conexão ou sessão em particular ou em resposta a algum pacote enviando pela mesma. Em tais casos, o cabeçalho IP e os primeiros 8 bytes do cabeçalho de transporte (corresponde aos endereços das portas de origem e destino usadas na comunicação; este valor é de 64 bytes em Solaris da Sun

Microsystems, definido através da variável *icmp_return_data_bytes*) são incluídos na mensagem ICMP [22]. Desta forma, o sistema operacional pode identificar o escopo das mudanças ditadas pelo ICMP. Portanto, uma mensagem de *"Redirect"* ou uma *"Destination unreachable"* devem ser específicas a uma conexão ou sessão. Contudo, implementações ICMP antigas não fazem uso desta informação e todas as conexões e sessões entre duas máquinas são afetadas [4]. Estas implementações são mais vulneráveis a ataques de negação de serviços. Mensagens ICMP forjadas enviadas podem, portanto, derrubar todas as conexões e sessões legítimas existentes entre um par de *hosts*.

No entanto, o perigo maior reside na possibilidade do uso de mensagens ICMP para desvirtuar rotas através de mensagens *"Redirect"*. O que torna a implementação deste tipo de ataque mais difícil de ser implementado é que este tipo de mensagem só pode ser gerado em função de conexões existentes e deve ser originado de um *gateway* diretamente conectado à mesma sub-rede da máquina atacada [22]. Serão vistas, na próxima seção, outras estratégias que um atacante pode usar com a finalidade de criar rotas para um ataque. Tendo o controle das rotas tomadas pelos pacotes de uma sessão, qualquer atacante pode controlar por onde os pacotes de uma determinada conexão ou sessão devem fluir na rede e, consequentemente, fazer com que todo o tráfego passe por si mesmo ou realizar um ataque de negação de serviço induzindo o tráfego de pacotes a usar caminhos por roteadores que não tenham como direcionar pacotes para o destino desejado.

## 2.7 Roteadores, Rotas e Protocolos de Roteamento

Protocolos de roteamento são mecanismos usados por roteadores em uma rede para dinamicamente descobrirem as melhores rotas para cada destino e detectarem possíveis problemas relacionados a algumas de suas rotas [9]. Roteadores, portanto, podem usar algum algoritmo para a troca de informações com vistas a atualizar suas respectivas tabelas de rotas.

A dificuldade em personificar completamente uma máquina está em se fazer com que todos os pacotes enviados à máquina legítima cheguem ao falso destino. Usando deficiências existentes nos diversos protocolos de roteamento, um atacante pode tentar subverter os roteadores entre ele e o alvo do ataque. Certos protocolos de propagação de rotas, principalmente RIP (*Routing Information Protocol*), não têm como verificar se as informações recebidas, supostamente de roteadores vizinhos, são verdadeiras [4]. Desta forma, um atacante pode injetar informações de

roteamento, modificando de acordo com suas necessidades as tabelas de rotas dos roteadores. Alguns protocolos de roteamento, como RIP Versão 2 e OSPF (*Open Shortest Path First*), fornecem um campo de autenticação, além das informações de roteamento [22]. Entretanto, o mecanismo de autenticação definido usa simples *passwords*. Além disso, vale notar que ataques a roteadores usando o protocolo de roteamento podem propagar as informações forjadas pelos vários roteadores da rede.

Além de usar as deficiências dos protocolos de roteamento e de ICMP *redirects* (como visto na seção anterior), um atacante pode manipular rotas facilmente usando a opção *source routing* do protocolo IP. Pacotes IP com esta opção são roteados não com base nas tabelas de rotas, mas conforme o caminho especificado pelo originador dos mesmos [4]. Entretanto, este tipo de ataque às rotas pode ser facilmente evitado usando uma facilidade hoje existente na maioria dos roteadores que permite rejeitar pacotes IP contendo esta opção.

## 2.8 Nível de Aplicação TCP/IP

Os diferentes serviços e protocolos de aplicação TCP/IP herdaram vários dos problemas dos níveis inferiores da pilha de protocolos TCP/IP e acrescentaram outros particulares. FTP (*File Transfer Protocol*) e Telnet (*Telecommunications Network Protocol*), por exemplo, usam somente *passwords* como esquema de autenticação, não oferecem nenhuma garantia de que os dados terão sua privacidade preservada e não sofrerão alteração [1][4]. O HTTP (*Hypertext Transfer Protocol*), protocolo no qual é baseada a Web, abriu espaço para o transporte de diversos tipos de dados encapsulados nos pacotes do protocolo e, graças a isso, surgem várias outras aplicações que tornam a Internet ainda mais interessante. Por outro lado, não oferece nenhum esquema suficientemente seguro de autenticação, não há cifragem nos diferentes tipos de dados, programas (*applets* Java, Javascript e similares) podem ser transmitidos e executados em diferentes lugares sem a necessidade de verificação de assinaturas digitais e integridade, dentre outros problemas [4][10]. Aliás, esta grande diversidade de dados transmitidos no protocolo HTTP e a execução de programas armazenados remotamente são hoje um dos maiores problemas para os projetistas de firewalls [27].

## 2.9 DNS (*Domain Name System*)

O DNS é um serviço do nível de aplicação, que usa UDP ou TCP (quando o resultado de uma consulta excede o tamanho de 512 bytes), responsável pelo mapeamento de nomes em endereços IP e vice-versa [22]. Como DNS é um banco de dados distribuído, existe pouco controle sobre a veracidade das informações divulgadas pelos diferentes domínios na Internet. Um atacante, por exemplo, controlando o mapeamento inverso de algum domínio pode criar um mapeamento de um número IP local (domínio do atacante) em um nome de uma máquina confiável de um outro sistema qualquer (sistema alvo). Desta forma, ele pode explorar serviços que se baseiam em nomes de máquinas como esquema de autenticação, como é o caso dos comandos "r" do Unix [2][50]. Atualmente os mais novos sistemas são imunes a este tipo de ataque. Estes sistemas comparam o mapeamento inverso com o mapeamento direto (*cross-check*) em chamadas de sistema como *gethostbyname*.

A ausência de um mecanismo seguro de autenticação de servidores abre a possibilidade de ataques visando a "poluição" da cache de respostas de servidores DNS ou dos próprios *resolvers*[1]. Um atacante, por exemplo, pode simplesmente injetar informações falsas, dizendo ser o servidor DNS de um certo domínio. Pode-se usar, por exemplo, o campo de informações adicionais (introduzido com o objetivo de melhorar o desempenho [22], eliminando a necessidade de futuras consultas) de uma resposta verdadeira no processo de *cross-check*. Futuras consultas seriam desnecessárias e estas informações mantidas na *cache* seriam usadas. Um outro ataque que pode ser implementado, devido ao fraco esquema de autenticação, é o DNS *spoofing*. O atacante, usando alguma maneira para interceptar consultas de uma máquina alvo a servidores DNS de outros domínios ou simplesmente tentando prever números de identificação usados nas *queries* dos clientes DNS em uma máquina ou rede alvo, pode gerar respostas forjadas, passando-se pelo servidor real [45].

Além destes problemas citados acima, o DNS oferece, em geral, informações preciosas para atacantes a respeito de uma rede privada. Uma boa política é a de não autorizar transferências de zona (*zone tranfers*) entre servidores secundários [2]. Para isto, pode-se simplesmente usar alguma tecnologia de filtragem de pacotes no *firewall*, evitando a passagem dos pacotes TCP do

1. Em alguns casos, como na execução de *applets* Java, os *resolvers* tentam resolver nomes em outros domínios sem a ajuda de um servidor DNS local.

servidor DNS [10]. Entretanto esta solução não é suficiente, visto que um atacante paciente pode pesquisar todo o espaço de endereçamento via pedidos de resolução inversa, obtendo uma lista de nomes de máquinas numa rede local. A partir desta lista, eles podem deduzir e buscar outras informações.

## 2.10 Implementações dos Softwares TCP/IP

Além das várias deficiências de segurança existentes nos protocolos da pilha TCP/IP, que foram brevemente abordadas neste capítulo, as implementações destes protocolos e dos outros diferentes serviços e protocolos de aplicação TCP/IP, em sistemas operacionais, têm historicamente apresentado sempre várias falhas de segurança.

Ataques de *buffer overflow* têm sido o mais comum problema de segurança associado às implementações de serviços e protocolos de aplicação TCP/IP. Só para se ter uma idéia deste fato, 9 de 13 das "CERT *advisories*" de 1998 e pelo menos metade delas em 1999 envolviam *buffer overflow* [6][40]. Este tipo de ataque tem como objetivo geral subverter alguma função de um programa com privilégios especiais sobre o sistema operacional. Aproveita-se da ausência de verificação dos limites de tamanhos de *buffers* ou de problemas de *overflow* na alocação dinâmica de memória, tais que o atacante possa obter controle deste programa e herdar estes privilégios [40].

O uso de algumas linguagens de programação, como Java, pode amenizar problemas de *buffer overflow* em programas. Oferecem mecanismos de *garbage collection* (coleta de lixo), evitam que o programador manipule apontadores de memória e que realize outras tarefas que possam proporcionar o surgimento destes problemas. Além disso, já existem ferramentas que procuram eliminar os efeitos de *buffer overflows* em linguagens de programação que não possuem estas características presentes em Java. Um exemplo famoso é o Stackguard [40], que permite ao gcc (gnu C) gerar binários que façam verificações em tempo de execução. Evitam-se assim que funções sejam subvertidas usando estouro na pilha do processo para reescrever endereços de retorno. Em conseqüência, o problema com este tipo de ferramenta é que ela gera código menos eficiente. Além disso, ela somente impede que algum atacante obtenha controle do programa, parando a execução do programa imediatamente. Isso significa que neste caso, um atacante ainda pode causar prejuízos com ataques de DoS.

Algumas implementações da pilha de protocolos TCP/IP podem sofrer ataques explorando problemas no processo de remontagem de pacotes com seus fragmentos. Um ataque famoso de negação de serviço que se aproveita desta falha em alguns sistemas antigos é o *ping of death*. Este ataque consiste em enviar um pacote ICMP *echo request* suficientemente grande para uma máquina alvo. Como este pacote é fragmentado em trânsito, a máquina alvo não sabe como tratá-lo. Além disso, certos pacotes indevidamente ou particularmente construídos, que conseguem passar pelos testes de sanidade do sistema operacional, podem derrubar uma máquina [46].

Vale ressaltar ainda que atacantes podem descobrir qual o sistema operacional usado em uma determinada máquina alvo, utilizando certas decisões de projeto[1] tomadas na implementação da pilha de protocolos TCP/IP neste sistema [54][69]. Um atacante pode enviar uma seqüência de pacotes com certas características, como pacotes construídos com combinações atípicas de *flags* TCP ligadas, para uma determinada máquina e determinar o sistema operacional da máquina a partir dos pacotes recebidos como respostas. Esta informação pode ser muito interessante para um atacante, pois ele pode verificar as mais novas vulnerabilidades encontradas para esta plataforma e explorá-las em seu ataque, por exemplo. Algumas ferramentas de *scanning* (ver próxima seção), como o nmap [65][66], usam detalhes específicos de cada implementação para determinar qual o sistema operacional de uma máquina remota.

## 2.11 *Scanning*

Antes de realizar ataques a uma rede alvo, atacantes tentam coletar todas as informações possíveis sobre as máquinas da rede: a lista de máquinas, o sistema operacional de cada máquina, a topologia da rede, dentre outras. Isso permitirá que ele possa aprender quais os pontos fracos da mesma e como agir sem chamar a atenção. Uma farramenta útil que pode ser utilizada para este fim é o *Internet scanner*. *Scanners* [14] têm como objetivo procurar por serviços e falhas que possam comprometer a segurança de uma máquina ou rede.

Atualmente, os *scanners* mais usados são os chamados *stealth scanners*. Eles são conhecidos por este nome porque procuram esconder a origem do ataque e, principalmente, evitar sua detecção nos *logs* do sistema ou por ferramentas de detecção. Algumas ferramentas IDS

1. As especificações dos protocolos TCP/IP, em geral, deixam vários pontos em aberto. Isto permite comportamentos diferentes nas diversas implementações destes protocolos.

(*Intrusion Detection System*), como o scanlogd, verificam os cabeçalhos de transporte e de rede de pacotes recebidos por uma máquina e usam um conjunto de assinaturas destes ataques mais populares para detectar e registrar tentativas de *scanning*. Infelizmente, estas ferramentas não são perfeitas e falsos positivos e negativos são muito comuns. Além disso, alguns *scanners* são realizados lentamente, dificultando ainda mais sua detecção.

## 2.12 *Firewalls*

Para proteger uma rede privada contra as várias ameaças vistas neste capítulo, pode-se usar *firewalls*. *Firewall* é um conjunto de sistemas situado entre uma rede privada e a Internet, que têm como principal função interceptar todo tráfego entre ambos e fazer uma filtragem, bloqueando potenciais ataques [10]. Neste trabalho, serão abordadas as diversas tecnologias de *firewall*, ressaltando a capacidade de filtros de pacotes com estados de oferecerem suporte a uma classe de serviços que oferece problemas para as tecnologias mais tradicionais de *firewall*.

# Capítulo 3

# Tecnologias de *Firewall*

Este capítulo tem o objetivo de apresentar as tecnologias mais tradicionais de *firewall*, apontando alguns dos seus respectivos pontos positivos e negativos. Além disso, introduzirá a nova geração de filtros de pacotes, filtros de pacotes com estados, que é o foco principal deste trabalho e será explorado com mais detalhes nos próximos capítulos.

## 3.1 Introdução

No capítulo anterior, foi visto que a suíte de protocolos TCP/IP e implementações de alguns dos *softwares* envolvidos apresentam diversas vulnerabilidades já amplamente conhecidas e que podem ser exploradas por pessoas mal intencionadas [44]. No momento em que uma rede local é conectada à Internet, surge de imediato, portanto, a preocupação com a segurança das suas informações vitais. Além disso, em certos casos outros cuidados precisam ser tomados com relação à disponibilidade dos serviços oferecidos em uma rede local, à confidencialidade e integridade dos dados quando em trânsito pela Internet e em relação à autenticidade dos participantes em cada sessão. Estas preocupações com segurança, que têm se tornado ainda maior com a utilização da Internet para fins comerciais, provocam o aumento dos esforços em busca de soluções capazes de agregar um maior grau de segurança às diversas camadas de comunicação da suíte TCP/IP e às respectivas implementações dos *softwares*. Hoje, já é possível encontrar, no mercado, produtos das diversas tecnologias resultantes deste esforço: *firewalls*, IDS (*Intrusion Detection Systems*), VPN (*Virtual Private Network*), Anti-Vírus, dentre outras.

O escopo deste trabalho, como já foi afirmado anteriormente, compreende tecnologias de *firewalls*. O papel de um *firewall* é defender a rede privada das ameaças oriundas da rede

externa [10]. Por outro lado, *firewalls* não tratam dos problemas ligados ao canal lógico de comunicação, não conseguem deter ataques provenientes de usuários dentro da própria rede privada e não necessariamente incrementam o nível de proteção individual de cada máquina dentro da rede interna[1]. Algumas destas limitações podem ser eliminadas ou somente amortizadas com o uso de outras tecnologias de segurança específicas, que estão fora do escopo deste trabalho. Como exemplo, os problemas relacionados ao canal de comunicação podem ser resolvidos com a adição de produtos de VPN que criam canais seguros (a informação é toda cifrada antes de ser transmitida no canal inseguro) para a troca de informações entre os participantes. Isto pode ser uma das explicações para a tendência cada vez maior dos soluções comerciais estarem combinando diversas tecnologias de segurança em um mesmo produto [49], visto que cada uma delas tem suas próprias vantagens e desvantagens e tratam de problemas de segurança nos mais diferentes níveis da comunicação. Existe hoje no mercado o conceito de *firewall* pessoal, cujo objetivo é proteger uma máquina individualmente (o *software* de *firewall* pessoal é instalado e configurado na máquina a ser protegida). Entretanto, *firewall* pessoal usa as mesmas tecnologias básicas de *firewall* que serão apresentadas neste capítulo, geralmente combinadas com outras soluções de segurança. Não se trata, portanto, de nenhuma nova tecnologia. O emprego de um *firewall* pessoal é bastante comum para que usuários móveis possam usar remotamente uma VPN corporativa sem comprometer a segurança da mesma. Neste caso, evita-se que o computador do usuário seja usado como uma porta dos fundos (*backdoor*) para a rede corporativa.

Neste capítulo, serão vistas as tecnologias usadas para a implementação de *firewalls,* independente da topologia e dos outros componentes necessários para a implementação de *firewalls* em redes locais (*bastion hosts*, por exemplo). O leitor mais interessado em uma discussão mais completa sobre arquiteturas e componentes de *firewall* é convidado a ler a referência [10].

## 3.2 *Firewalls*

Um *firewall* é um sistema ou um conjunto de sistemas que implementam uma política de controle de acesso entre duas redes. Para tal finalidade, o *firewall* deve interceptar todo o tráfego de pacotes entre as duas redes e, com base na política de segurança interna, permitir ou bloquear

1. Entretanto, *firewalls* podem e são utilizados internamente a um *site* para compartimentalizar áreas/regiões/domínios administrativos.

a sua passagem [25]. Desta forma, o *firewall* pode reduzir os riscos de ataques às máquinas localizadas internamente, controlando quais máquinas na rede externa podem estabelecer sessões com quais servidores internos e, por outro lado, quais máquinas internas podem estabelecer sessões com quais computadores remotos. Como será visto nas próximas seções, dependendo da tecnologia empregada, o *firewall* pode também inspecionar e efetuar filtragem no fluxo de dados trocado durante a comunicação, aumentando consideravelmente o grau de segurança. Quando a rede externa em questão é a Internet, chama-se o *firewall* de "Internet *Firewall*".

## 3.3 Filtro de Pacotes Tradicional (*Packet Filter*)

A filtragem de pacotes é o processo de seletivamente permitir ou bloquear o tráfego de pacotes entre duas redes, fazendo uso de um conjunto de regras de filtragem. Estas regras são baseadas em informações existentes nos cabeçalhos de cada pacote e, logo, a filtragem é feita em cada pacote de uma sessão individualmente [10][42]. No caso de datagramas IP, a filtragem pode ser feita usando-se informações como endereços IP de origem e destino, portas de origem e destino, protocolo de transporte usado, tipo e código de pacotes ICMP, opções IP, dentre outras (a Tabela 3.1 mostra as regras de filtragem para o serviço SMTP). Pode-se concluir que filtros de pacotes fazem todo seu trabalho principalmente nos níveis de rede e de transporte e, desta forma, são comumente implementados junto com o processo de roteamento em roteadores ou no *kernel* dos sistemas operacionais. Algumas implementações de filtros de pacotes também fazem algum trabalho na parte de dados dos pacotes. Filtros dinâmicos (que serão apresentados na Seção 3.3.2, página 28), por exemplo, geralmente procuram por certas informações na parte de dados dos pacotes e as utilizam para criar outras regras de filtragem dinamicamente.

Por diversos anos, várias redes locais fizeram uso de roteadores escrutinadores (*screening routers*), implementados através de roteadores profissionais dedicados, para garantir a segurança de suas informações e controlar o acesso [10]. Estes roteadores simplesmente adicionam a tarefa de filtragem de pacotes à sua função normal de rotear o tráfego de pacotes pela rede. Atualmente, principalmente devido ao significativo aumento no desempenho dos processadores, muitos sistemas operacionais passaram a implementar também os processos de roteamento e de filtragem de pacotes.

Por serem implementados nas camadas mais baixas do sistema operacional[1] ou em *hardware*, filtros de pacotes apresentam sempre um bom desempenho. O baixo custo computacional

| Regras | Endereço Origem | Endereço Destino | Protocolo | Porta Origem | Porta Destino | ACK ligado? |
|---|---|---|---|---|---|---|
| ACEITO | Externo | Interno | TCP | >1023 | 25 | Não |
| ACEITO | Interno | Externo | TCP | 25 | >1023 | Sim |
| ACEITO | Interno | Externo | TCP | >1023 | 25 | Não |
| ACEITO | Externo | Interno | TCP | 25 | >1023 | Sim |

**Tabela 3.1:** Regras de Filtragem de Pacotes para o SMTP (*Simple Mail Transfer Protocol*)

dos filtros de pacotes permite ainda que eles sejam facilmente adequados ao crescimento do número de máquinas na rede interna (escalabilidade). Uma outra vantagem importante é a transparência, pois filtros não exigem nenhuma configuração especial nos *softwares* clientes e/ou servidores dentro da rede interna; os usuários só percebem que há alguma filtragem de pacotes quando tentam fazer algo não permitido pela política de segurança implementada [10][29].

Há uma série de ataques que se consegue evitar com a filtragem de pacotes. De uma forma geral, evitam-se todos os ataques aos serviços que não são permitidos pelas regras configuradas no filtro. Vale destacar aqui que todos os ataques mais sofisticados que se utilizam de IP *spoofing* também podem ser evitados com esta tecnologia de *firewall* [10]. Se a filtragem é realizada por interface de rede em ambos os sentidos e o atacante estiver tentando utilizar o endereço de alguma máquina dentro da rede interna (IP *spoofing*), este ataque não funcionaria porque, em condições normais, jamais um pacote pode chegar da rede externa tendo como endereço de origem o endereço de uma máquina interna (Figura 3.1). Com a correta configuração da filtragem, pacotes em tal situação poderão facilmente ser boqueados. Além destes, muitos daqueles

1. A implementação do filtro no *kernel* do sistema operacional é muito interessante devido ao significativo aumento no desempenho obtido; não há a necessidade de compartilhar o processador com processos do usuário, pois o *kernel* tem prioridade máxima. Como a filtragem envolve poucos dados, o consumo de memória do *kernel* é mínimo. Em contrapartida, *bugs* de implementação podem provocar *crashes* no sistema e danificar *filesystems*, dentre outros [82].

ataques que se baseiam em enviar pacotes mal formados (fragmentos com cabeçalho TCP truncado [21], por exemplo) ou *flooding* de pacotes não serão bem sucedidos, pois o processo de filtragem pode proibir que tais pacotes atinjam máquinas dentro da rede a ser protegida.

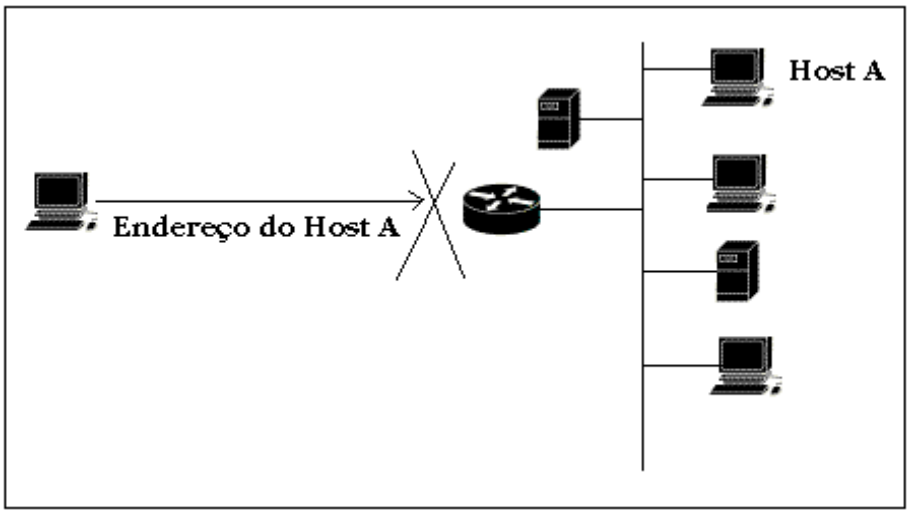


**Figura 3.1:** Evitando IP Spoofing com um roteador escrutinador

No entanto, por não entender o fluxo de dados dos pacotes de cada sessão, podem permitir ataques às vulnerabilidades particulares dos diversos protocolos e serviços do nível de aplicação [42] e, por este mesmo motivo, são incapazes de desempenhar autenticação do usuário e criar *logs* mais sofisticados de cada comunicação.

Filtros de pacotes carecem, principalmente em roteadores, de boas ferramentas de administração. As regras de filtragem podem ser muito difíceis de configurar e exigem o conhecimento de uma intricada sintaxe específica para o roteador. Além disso, a ordem em que as regras são definidas pode acarretar mudanças significativas no resultado do filtro de pacotes e, em muitos sistemas, a falta de ferramentas de edição torna ainda mais complicada a tarefa do administrador de segurança de editar as regras de filtragem a serem implementadas [42].

Outra desvantagem de filtros de pacotes é que eles não mantêm estados das conexões TCP ou sessões UDP e podem, com isso, permitir a passagem de pacotes que não pertençam a qualquer conexão ou sessão válida existente através do *firewall*, desde que eles satisfaçam as regras de filtragem [73][77]. Observar que esta vulnerabilidade diz respeito a pacotes ICMP também.

Como foi visto na Seção 2.6, página 12, alguns pacotes ICMP podem ser transmitidos para controlar o funcionamento de certas conexões ou sessões. Filtros de pacotes tradicionais não possuem as informações necessárias para relacionar os pacotes ICMP que chegam ao *firewall* às suas respectivas conexões ou sessões. Em alguns casos bloquear estes pacotes ICMP pode ser um grande problema. Um excelente exemplo desta deficiência é o processo de "*Path* MTU *Discovery*" [15][22]. Quando um cliente envia um datagrama IP com o bit DF (*Don't Fragment*) ligado, se este pacote precisar ser fragmentado em um roteador ao longo do caminho, este roteador retorna um pacote ICMP *Unreachable Error* (*Fragmentation Required*) ao cliente, com a MTU (*Maximum Transmission Unit*) do canal de comunicação que exige a fragmentação. Desta forma, o cliente pode reenviar pacotes menores, usando a MTU recebida. Se estes pacotes ICMP são filtrados no *firewall*, esta informação jamais chegará ao cliente e, portanto, o cliente nunca será capaz de se comunicar com máquinas na rede local destino. Por outro lado, criar regras para permitir pacotes ICMP atravessarem o *firewall* inadvertidamente também pode não ser uma boa idéia [39][54].

Esta escassez de informações de estados das conexões e sessões pode ser explorada em diversos ataques: *portscanning* (pacotes com o bit ACK ligado podem ser usados para descobrir que serviços são oferecidos dentro da rede interna [14]; portas ativas responderiam com pacotes com bit RST ligado e a ausência de resposta indicaria que o serviço não está disponível), ataques envolvendo fragmentação de pacotes (se fragmentos de pacotes são permitidos através do *firewall*, um atacante pode subverter a política de segurança implementada no filtro [21]. Para esta finalidade, basta usar a sobreposição de fragmentos para alterar os cabeçalhos de pacotes TCP no destino. Consequentemente, pacotes com pedido de conexão poderiam passar silenciosamente por filtros de pacotes), assinatura (*fingerprint*) do sistema operacional (Uma certa combinação de pacotes mais exóticos pode ser enviada para descobrir o sistema operacional [66] de um servidor interno) e "tunelamento" de dados (dados não permitidos podem ser transportados em pacotes permitidos [39], segundo análise do *firewall*), dentre outras possibilidades.

### 3.3.1 Filtros de Pacotes Estáticos (*Static Packet Filter*)

Os primeiros filtros de pacotes, por usarem somente regras de filtragem pré-definidas quando da configuração do filtro pelo administrador de segurança, ficaram conhecidos como filtros de pacotes estáticos [29]. Além das desvantagens comuns dos filtros de pacotes citadas anteriormente,

uma outra importante característica deste tipo de filtragem é a dificuldade para tratar alguns protocolos mais complexos, como por exemplo o FTP.

O FTP é um protocolo que usa duas conexões TCP em cada comunicação [22]: uma conexão de controle e uma conexão de dados. O uso de mais de uma conexão ou sessão é necessário em alguns protocolos, pois eles podem envolver tráfego de dados com características diferentes. Por esta mesma explicação, existem também outros protocolos e serviços que fazem uso simultaneamente de sessões UDP e conexões TCP. No caso do FTP, a conexão de controle transporta sempre poucos dados e exige um atraso mínimo, enquanto a conexão de dados geralmente transporta grandes volumes de dados e algum atraso pode ser tolerado. O problema para a filtragem estática deste protocolo de aplicação é que a conexão de dados é criada dinamicamente durante a conexão de controle. Quando um cliente interno, por exemplo, se comunica com um servidor remoto, ele inicialmente estabelece uma conexão de controle com a porta 21 do servidor, a partir de uma porta alta no cliente. Mas, no momento em que o cliente deseja iniciar uma transferência de dados, ele envia um comando PORT, para o servidor, notificando o número de porta do cliente que estará aguardando o pedido do servidor remoto para estabelecimento da conexão de dados. O servidor (usando sua porta 20) estabelece a conexão à porta do cliente recém informada e a transferência de dados pode então ser iniciada. Depois que os dados são transferidos entre os participantes, esta conexão de dados é encerrada. Portanto, várias conexões de dados (uma para cada transferência de dados) podem ser iniciadas e encerradas durante a existência da conexão de controle. Este modo de operação do FTP é conhecido como modo ativo (a Figura 3.2 ilustra este modo de operação do FTP).

Notar que estes protocolos tornam bastante complexa a atividade de criação de regras de filtragem estáticas em filtros de pacotes, pois normalmente envolvem um processo dinâmico de negociação das portas a serem usadas [10]. Como criar regras estáticas sem conhecer de antemão os números de portas a serem utilizados? Este é o mesmo problema existente em filtros de pacotes para serviços baseados em RPC, que serão abordados no próximo capítulo. Voltando ao exemplo do FTP, no caso de clientes localizados dentro da rede interna acessando servidores na Internet; o filtro de pacotes deverá permitir todos os pacotes com pedidos de conexão vindos de quaisquer máquinas remotas (supostos servidores FTP) destinados aos clientes internos (usando portas acima de 1023). Isto abre, portanto, espaço para ataques a servidores internos

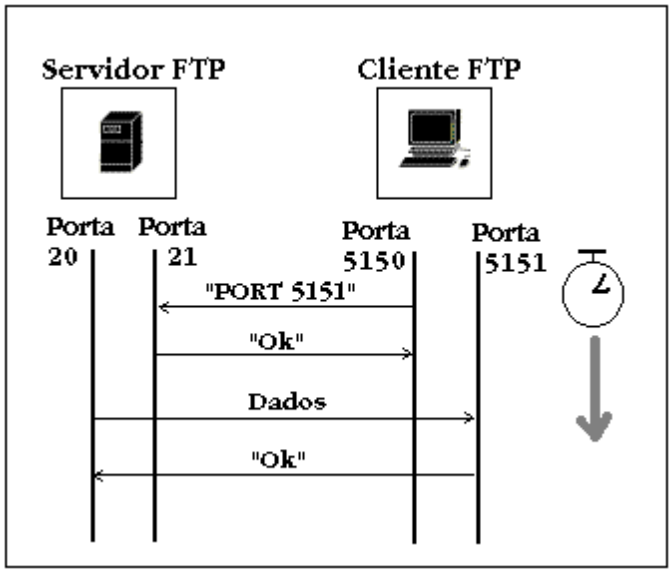


**Figura 3.2:** Modo Ativo do FTP

[42] usando números de portas não privilegiadas (valores acima de 1023), como serviços baseados em RPC ou o servidor X, a partir de clientes remotos usando o número de porta 20.

Uma solução para este problema, proposta por Steve Bellovin [87], é o uso de um outro modo de operação do protocolo, o FTP passivo (ver Figura 3.3). Basicamente, este recurso inverte o sentido da conexão de dados. Em vez do servidor iniciar a chamada para o cliente, o cliente iniciaria a conexão a uma porta do servidor (o cliente e o servidor já teriam negociado as suas respectivas portas, ambas não privilegiadas, durante a conexão de controle). A vantagem desta solução é que as regras de filtragem não precisam permitir pacotes, com pedidos de conexão vindos da porta 20 em máquinas remotas, com destino a serviços em máquinas internas, usando portas acima de 1023. Elimina-se, portanto, os riscos de algum ataque explorando o espaço deixado nas regras de filtragem para o modo ativo do FTP [10]. Entretanto, nem todos os clientes e servidores FTP suportam este modo de operação. Além disso, esta é uma solução exclusiva para o FTP; muitos outros protocolos com características similares não possuem modos alternativos de operação.

Uma outra grande limitação dos filtros de pacotes estáticos é a dificuldade para filtrar serviços baseados em UDP. Este protocolo de transporte não é orientado a conexões e, portanto, não existe um processo de estabelecimento de conexão. No TCP, que é orientado a conexão, a fil-

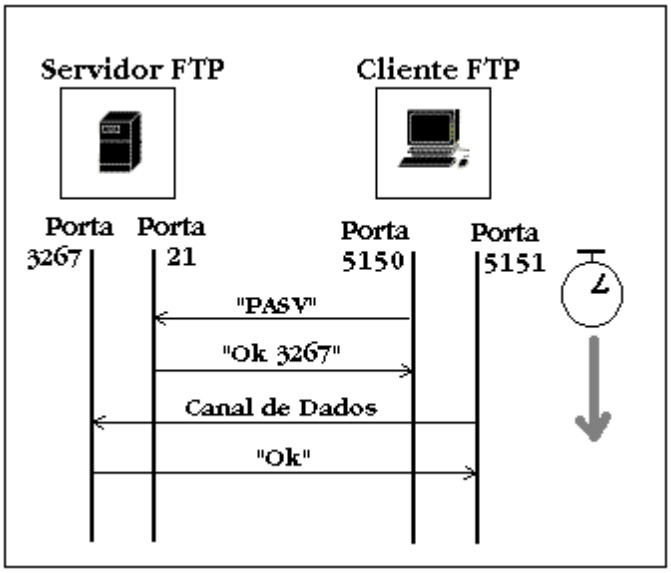


**Figura 3.3:** Modo Passivo do FTP

tragem pode ser feita simplesmente bloqueando ou permitindo a passagem do pacote com o pedido de estabelecimento da conexão [10]. Entretanto, esta solução simples para a filtragem dos serviços baseados em TCP ainda não é perfeita, pois neste caso ainda teremos a possibilidade de pacotes TCP quaisquer, não pertencendo a nenhuma conexão válida, atravessarem o *firewall*. Em UDP, não existe distinção entre uma requisição e uma resposta. Em filtros estáticos, as soluções utilizadas para serviços baseados em UDP consiste simplesmente em eliminar as sessões UDP inteiramente, ou ainda em abrir uma porção do espaço de portas UDP para a comunicação bi-direcional e, portanto, expor a rede interna [42].

### 3.3.2 Filtros de Pacotes Dinâmicos (*Dynamic Packet Filter*)

Surgiram depois filtros de pacotes mais inteligentes, capazes de criar regras de filtragem dinamicamente dependendo do andamento das comunicações e, por conseqüência, capazes de resolver os problemas existentes nos filtros estáticos vistos acima [29]. Estes são conhecidos como filtros de pacotes dinâmicos. Filtros dinâmicos, apesar de ainda apresentarem os problemas clássicos característicos dos filtros de pacotes mostrados anteriormente (ver Seção 3.3, página 22), já mantêm algumas poucas tabelas com informações de estados extraídas dos pacotes de algumas

sessões. Isto torna possível a criação de regras de filtragem dinâmicas para certos protocolos (como o FTP mostrado acima) mais complexos e para tratamento de serviços baseados em UDP [10], principalmente.

O tratamento de protocolos mais complexos é possível porque estes filtros acompanham o fluxo de dados de alguns protocolos especiais, como por exemplo o FTP, e extraem as informações necessárias para a criação de regras de filtragem "*on the fly*". Para o caso particular do FTP, um filtro dinâmico inspeciona o fluxo de dados dos pacotes da conexão de controle em busca de comandos PORT, para obter os números de portas que serão utilizados na conexão de dados. Esta estratégia também é aplicada em algumas implementações de filtros de pacotes com estados. Recentemente, foi apresentada uma séria vulnerabilidade com esta solução implementada em dois dos principais representantes dos filtros de pacotes com estados no mercado: o Firewall-1 da CheckPoint e o CISCO-PIX da Cisco. Este problema será discutido novamente na Seção 5.7, página 53. Vale ressaltar ainda que filtros de pacotes dinâmicos somente são capazes de criar regras dinamicamente para alguns serviços e protocolos muito utilizados, suportados pelo filtro. Para outros protocolos e serviços, a filtragem também é estática.

Filtros de pacotes dinâmicos geralmente têm a capacidade de "lembrar" os pacotes UDP que passam pelo *firewall* (são permitidos pela política de segurança implementada no filtro) e, com isso, somente aceitam pacotes no sentido inverso que correspondam a estes primeiros [10]. No próximo capítulo a filtragem de serviços baseados em UDP será discutida novamente, com mais detalhes.

## 3.4 Filtro de pacotes com Estados (SPF, *Stateful Packet Filter*)

O objetivo desta seção é introduzir brevemente esta nova geração de filtros de pacotes. Maiores detalhes serão apresentados nos próximos capítulos, principalmente no capítulo 5. Um filtro de pacotes com estados usa, no seu processo de filtragem, um conjunto de regras de filtragem, como no caso dos filtros tradicionais, e as informações de estados obtidas das conexões e sessões [32].

A filtragem tradicional somente ocorre com o primeiro pacote da conexão ou sessão, ou seja, somente o primeiro pacote pertencendo a uma sessão ou conexão é verificado contra o conjunto de regras. Se este pacote é permitido, o SPF cria uma entrada para esta conexão ou sessão em

sua tabela de estados. Deste momento em diante, os demais pacotes de uma conexão ou sessão somente serão permitidos se existir alguma entrada na tabela de estados para esta conexão ou sessão [77]. Além disso, por manterem estados das conexões ou sessões, filtros de pacotes com estados são capazes de associar pacotes ICMP de controle às suas respectivas conexões. Isso garante também que somente pacotes ICMP válidos (associados a comunicações existentes) passem pelo *firewall*, eliminando a necessidade de criação de regras específicas para estes pacotes, que podem propiciar espaços perigosos no *firewall*. Desta forma, evita-se que pacotes (TCP, UDP ou ICMP) quaisquer, permitidos segundo as regras no filtro, entrem na rede interna, como acontecia nos filtros de pacotes tradicionais [73]. Portanto, há tabelas de estados que acompanham o andamento de cada conexão ou sessão atravessando o filtro em um dado momento. Algumas implementações, todavia, não acompanham rigorosamente o andamento das conexões e sessões, como será mostrado no capítulo 5.

Uma outra característica de filtros de pacotes com estados é que, ao contrário dos filtros tradicionais, geralmente fazem um trabalho muito maior no nível de aplicação [29][30]. Aliás, muitas das informações de estados mantidas pelo filtro são extraídas da parte de dados dos pacotes. SPFs mais sofisticados entendem certos protocolos e serviços de aplicação e chegam a fazer filtragem na parte de dados dos pacotes, eliminando alguns dos problemas de segurança existentes também no nível de aplicação. Alguns, inclusive, oferecem linguagens de script poderosas para a escrita de regras de filtragem usando informações do nível de aplicação [31]. Como será visto no capítulo 7, o resultado deste trabalho foi um filtro capaz de verificar os dados trocados na comunicação entre clientes RPC e o portmapper/rpcbind (servidor de nomes do RPC) e criar regras dinâmicas para acomodar serviços baseados em RPC que utilizam números de portas alocados dinamicamente.

Pode-se concluir, com as características mostradas acima, que filtros de pacotes com estados podem ser considerados simplesmente uma evolução dos filtros de pacotes dinâmicos [29]. Isto explica o porquê da facilidade de encontrar os termos “filtros dinâmicos” e “filtros de pacotes com estados” sendo usados como sendo uma mesma geração de filtros de pacotes.

## 3.5 Agentes Proxy

O *proxy* é uma outra tecnologia de *firewall* que tem como objetivo principal controlar todas as comunicações, para algum determinado serviço ou conjunto de serviços, entre as máquinas inter-

nas e externas [10]. Com esta tecnologia, quando um cliente na rede interna, por exemplo, deseja estabelecer uma conexão com algum servidor remoto, a comunicação deve ser feita sempre via um servidor *proxy* apropriado, como sugere a Figura 3.4. Portanto, este esquema envolve pelo menos duas conexões: uma entre o cliente interno e o servidor *proxy* e outra entre o servidor *proxy* e o servidor remoto. Os dados de uma conexão são repassados para a outra e vice-versa, pelo agente *proxy*; notar que cabeçalhos de rede e transporte não atravessam a máquina *proxy*. Para servidores remotos é como se todos os clientes estivessem na máquina *proxy*. Os clientes internos, por sua vez, se comunicam com o servidor *proxy* como se este fosse o próprio servidor remoto. Obviamente, como primeira desvantagem desta tecnologia, temos uma perda de desempenho quando comparada com as tecnologias de *firewall* anteriores apresentadas neste capítulo [29].

Na implementação de *firewalls* usando agentes *proxy*, alguma das tecnologias de filtragem de pacotes discutidas pode ser adicionada para garantir que todo tráfego de pacotes proveniente das máquinas internas sempre passe pelo servidor *proxy*. Em outras palavras, nenhuma máquina interna deve comunicar-se diretamente com a rede externa. Por outro lado, o filtro de pacotes deve evitar também que máquinas remotas possam se comunicar diretamente com as máquinas internas [10]. Desta forma, assegura-se que em nenhuma circunstância pacotes provenientes da rede externa entrarão na rede interna sem passar pelo *proxy* e vice-versa, protegendo completamente as máquinas internas de ataques originados externamente.

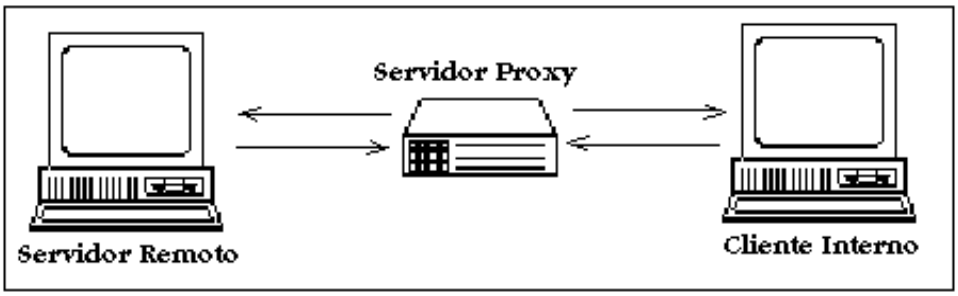


**Figura 3.4:** *Proxy*

Servidores *proxy* são implementados como processos do sistema operacional como qualquer outro serviço (aplicação), porém são normalmente mais simples (código fonte menor e constru-

ídos com maiores preocupações com segurança). Isto diminui em muito a possibilidade da existência de furos de segurança; além disso, os privilégios de tais processos sobre o sistema são geralmente limitados [41]. Por serem tratados como processos normais, dividirem o processador com os demais processos do sistema operacional e consumirem recursos da máquina, eles podem tornar-se um gargalo de comunicação, limitando também o crescimento do número de máquinas da rede (perda de escalabilidade) [29]. Além disso, uma máquina *proxy* pode ter sua segurança comprometida também por ataques a falhas de segurança no sistema operacional. São conhecidos hoje vários problemas em sistemas operacionais que podem, entre outras coisas, negar serviço, como por exemplo problemas na remontagem de pacotes com seus fragmentos. Então, alguma tecnologia de filtragem de pacotes é necessária para proteger o sistema operacional da máquina *proxy*. Outra limitação de servidores *proxy* é a dificuldade para lidar com protocolos UDP[1], por razões que serão discutidas no próximo capítulo.

### 3.5.1 Proxy de Aplicação ou Dedicado (*Dedicated Proxy*)

Um servidor *proxy* pode aumentar a segurança por examinar todo o fluxo de dados da comunicação na camada de aplicação. Realizando filtragem neste nível de comunicação, é possível bloquear ataques relacionados às vulnerabilidades próprias dos serviços e protocolos de aplicação [29]. Além disso, pode-se implementar *cache* de dados, prover autenticação de usuários e criar *logs* mais completos dos dados trocados. Neste caso, o *proxy* é conhecido como *proxy* de aplicação ou dedicado. Um *proxy* de FTP, por exemplo, é capaz de evitar que comandos GET (*download* de arquivos) sejam enviados em uma conexão de controle para um determinado servidor remoto, considerado inseguro pela política de segurança da rede local.

Como o *proxy* dedicado precisa conhecer uma aplicação em particular, deve existir um *proxy* para cada serviço desejado, transformando o suporte a novas aplicações um grande problema (extensibilidade). Para alguns serviços e protocolos pode ser difícil ou até mesmo impossível implementar *proxies* dedicados [32]. Além disso, com o crescimento de uso da Web, vários novos serviços e protocolos têm aparecido rapidamente, o que torna complexa a tarefa de criar *proxies* com a mesma rapidez.

---

1. ICMP também oferece problemas; a maioria dos pacotes ICMP são tratados no *kernel* do sistema operacional, ou seja, não atingem o espaço de processos.

### 3.5.2 Proxy de Circuito ou genérico (*Generic Proxy*)

Como nem sempre é possível contar com *proxies* dedicados para todos os serviços desejados, alguns produtos de *firewall* mais completos oferecem um tipo de *proxy* conhecido como *proxy* genérico ou de circuito. Este *proxy* não entende nenhuma aplicação em particular, mas simplesmente evita que as máquinas internas se comuniquem diretamente com servidores remotos, somente bombeando dados de pacotes de um lado para o outro [10]. Portanto, nenhuma filtragem é feita no nível de aplicação e, por isso, ele deve ser usado sempre temporariamente e com muita precaução[1].

O *proxy* genérico, assim como filtros estáticos, não consegue tratar serviços e protocolos mais complexos como o FTP, pois estes protocolos codificam certas informações (como números de porta, neste caso particular) na parte de dados dos pacotes e geralmente envolvem mais de uma sessão. Como um *proxy* genérico não entende os dados da aplicação, não será capaz de oferecer suporte a tais serviços e protocolos. Além disso, um *proxy* genérico não pode implementar *cache* de dados e não permite a criação de *logs* mais sofisticados dos dados trocados na comunicação.

### 3.5.3 Proxy Transparente (*Transparent Proxy*)

Um dos maiores problemas existentes em muitas implementações de agentes *proxy* é a perda de transparência. Os clientes na rede local devem ser configurados ou compilados para usarem um servidor *proxy* apropriado e não os servidores remotos diretamente; nos piores casos, o procedimento do usuário deve ser modificado para que tenha sucesso no uso do serviço [10]. Por anos, esta era uma importante desvantagem de *proxies* em relação aos filtros de pacotes.

Felizmente, esta deficiência já não existe nos *proxies* conhecidos como "transparentes" [29]. Para a implementação de um *proxy* transparente é preciso, porém, que todo tráfego de pacotes passe ou seja redirecionado de alguma maneira para a máquina *proxy*, pois o cliente não deve ter ciência que a comunicação será feita via *proxy*, ou seja, não há nenhuma configuração especial no *software* cliente ou no procedimento do usuário. É conveniente, portanto, que o servidor *proxy* fique em uma máquina *dual-homed* ou que seja usado algum filtro de pacotes que permita

1. No sentido de que não adiciona nenhuma segurança no nível de aplicação; não implica em nenhuma vulnerabilidade adicional.

redirecionar todo o tráfego, de um determinado serviço, para ser tratado pelo servidor proxy responsável.

Esta máquina *proxy*, para qual o tráfego será redirecionado, deve aceitar todos os pacotes (referentes ao serviço para o qual o servidor *proxy* oferece tratamento) vindos dos clientes internos, independente do endereço de destino do servidor real. Isso significa que o código de roteamento no *kernel* do sistema operacional do servidor *proxy* deve suportar *proxy* transparente. Depois de capturados os pacotes dos clientes e redirecionados às aplicações *proxy*, o *proxy* transparente funciona como um *proxy* tradicional para a comunicação com o servidor remoto. Para o cliente é tudo transparente, porque o servidor proxy usa, ao invés do seu próprio endereço, o endereço do servidor remoto como endereço de origem dos pacotes que ele envia de volta para o cliente.

## 3.6 NAT (*Network Address Translation*)

Não é comum tratar o processo de tradução de endereços de rede como uma tecnologia de *firewall*. Todavia, não se pode deixar de discutir NAT [71] neste capítulo, uma vez que algumas de suas implementações têm facilitado também a criação de *firewalls* bastante seguros. Nesta seção, não serão abordadas todas as potencialidades existentes para esta tecnologia, mas somente como NAT pode ser usado com a finalidade de prover segurança para uma rede local.

NAT, quando implementado com cuidados adequados [30], pode ser usado para isolar completamente as máquinas de uma rede local dos perigos externos, ou seja, ele pode evitar que máquinas na rede externa iniciem diretamente sessões com computadores dentro da rede interna. Uma boa alternativa para alcançar este resultado é fazer com que estas máquinas internas utilizem endereços IP não oficiais[1] e toda a comunicação delas com *hosts* em redes remotas seja feita via NAT [62], como pode ser visto na Figura 3.5. Neste texto chamar-se-á de "*gateway*" a máquina responsável pela tradução de endereços e pelo roteamento dos pacotes resultantes.

O *gateway* se encarrega de fazer a reescrita dos cabeçalhos dos pacotes passando por ele, modificando o endereço IP não oficial para algum endereço IP válido de um conjunto de endereços IP oficiais mantidos nesta máquina. Além disto, este processo de reescrita dos cabeçalhos

1. A RFC 1918 define três blocos do espaço de números IP que devem ser usados somente em redes privadas (redes isoladas da Internet): 10.0.0.0 a 10.255.255.255 (rede classe A), 172.16.00 a 172.31.255.255 (rede classe B) e 192.168.0.0 a 192.168.255.255 (rede classe C).

dos pacotes, em certos casos (mais seguros), também pode envolver os campos com os números de porta usados.

Para que o processo de tradução funcione corretamente, esta máquina deve manter uma tabela com o mapeamento dos endereços internos para os respectivos endereços usados na comunicação. Algumas implementações mais seguras de NAT (por isso mais indicadas para a implementação de *firewalls*) ainda mantém outras informações de estados das conexões ou sessões, como por exemplo números de seqüência do TCP e números de portas envolvidos, fazendo tradução diferenciada para cada comunicação e eliminando possíveis espaços que possam ser explorados por atacantes [30]. Graças a estas informações, o *gateway* pode fazer a tradução inversa com os pacotes que chegam da rede externa destinados às máquinas internas e que pertençam somente a conexões ou sessões válidas iniciadas internamente. É válido afirmar que todo este processo ocorre transparentemente; isto significa que nem servidores remotos nem clientes internos percebem que ocorre o processo de tradução de endereços.

Pacotes partindo de máquinas na Internet, com endereços IP de destino não oficiais, são descartados em trânsito e, consequentemente, nenhuma comunicação pode ser estabelecida com computadores da rede local protegida. Somente máquinas na rede local serão capazes de iniciar sessões com máquinas remotas, usando algum endereço válido atribuído pelo *gateway*. Pode-se concluir que o papel do NAT é semelhante ao desempenhado por um servidor *proxy* genérico: as máquinas ficam "escondidas" atrás do *firewall*, toda comunicação é feita via *gateway* e não há nenhuma filtragem no nível de aplicação. Aliás, o *proxy* é considerado um caso especial de NAT, já que todos os endereços internos são vistos como um único endereço (o endereço da máquina *proxy*), por servidores na rede externa [30].

A grande deficiência de NAT para a implementação de *firewalls* é que, assim como a maioria dos filtros de pacotes tradicionais, esta tecnologia não faz nenhuma filtragem no nível de aplicação. Um *applet* Java hostil, por exemplo, explorando algum *bug* na máquina virtual Java, pode provocar um *buffer overflow* e produzir grandes problemas para a rede local protegida.

Deve-se observar, contudo, que muitas implementações de NAT somente reescrevem os campos de endereços e não mantém muitas informações de estados das conexões ou sessões sendo mascaradas pelo NAT. Nestes casos, não é aconselhável usar-se NAT na criação de *firewalls*. Como um exemplo, suponha-se um *firewall* com uma implementação de NAT que simplesmente faça o mapeamento de endereços de rede. Quando uma máquina interna deseja fazer

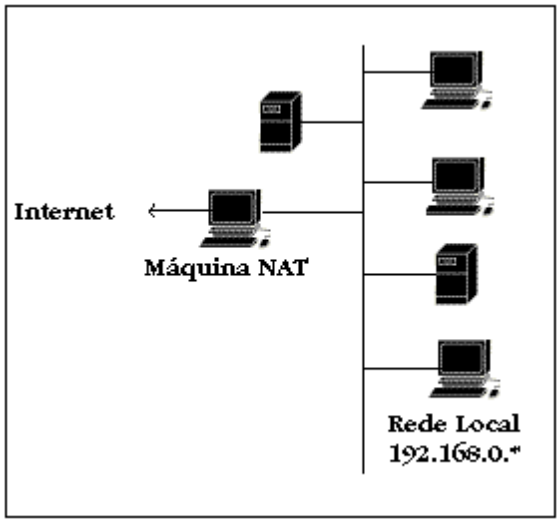


**Figura 3.5:** Firewall com NAT

uma comunicação com outra na rede externa, ela envia pacotes para o *gateway* realizar NAT e roteá-los para o destino. Na máquina *gateway*, será criada uma entrada mapeando o endereço interno ao endereço de saída utilizado. Um atacante pode tirar proveito desta janela de visibilidade na implementação do NAT e realizar, durante o tempo de existência desta entrada na tabela de mapeamento, ataques à máquina interna. Isto porque o *gateway*, neste caso, somente traduzirá o endereço dos pacotes enviados pelo atacante para o endereço da máquina interna, sem nenhuma checagem extra. Notar que implementações inseguras são bastante comuns, uma vez que NAT não foi originalmente concebido para prover segurança.

### 3.6.1 IP *Masquerading*

O IP *Masquerading* [62] é um caso especial de NAT que permite que computadores conectados internamente numa rede local, usando endereços IP não oficiais, comuniquem-se com servidores na Internet, compartilhando um mesmo endereço IP (o endereço da máquina *gateway*). O processo de *masquerading*, no *gateway*, reescreve os cabeçalhos dos pacotes provenientes das máquinas na rede interna, usando nos campos de endereço e porta de origem, respectivamente, o seu endereço IP válido e um número de porta local disponível, que serve para separar os tráfegos entre as diferentes sessões das várias máquinas da rede local. Por ser implementado no

*kernel* de sistemas operacionais como Linux e FreeBSD, este recurso tem sido bastante utilizado atualmente.

## 3.7 Firewalls Híbridos (*Hybrid Firewalls*)

*Firewalls* híbridos são aqueles que adicionam *proxies* dedicados a filtros de pacotes. Eles, portanto, tentam aliar o que há de melhor em ambas as tecnologias. O "*proxy* adaptativo" Gauntlet [76], da Network Associates, é um produto comercial que implementa um *firewall* híbrido. O Gauntlet combina filtros de pacotes com estados e *proxies* dedicados numa mesma caixa. A grande inovação deste produto de *firewall* é que ele permite que o administrador de segurança ajuste o processo de filtragem nas diversas camadas de comunicação de acordo com os riscos oferecidos por um serviço. Desta forma, o administrador pode configurar o *firewall* para fazer uma filtragem nos comandos da conexão de controle FTP, mas diminuir o atraso relacionado à conexão de dados FTP, fazendo uma filtragem somente nos níveis de rede e transporte para esta última conexão. Um outro produto comercial, que combina filtros de pacotes com estados e proxies dedicados, é o Firewall-1 da CheckPoint [31]. Este será visto em mais detalhes no capítulo 5.

Além disso, é cada vez mais comum o surgimento de produtos juntando tecnologias de *firewall* com outras tecnologias de segurança, criando diversos novos termos, como se tratassem de idéias revolucionárias [49]. NAT e VPN, por exemplo, têm sido oferecidos na maioria dos produtos de *firewall*. Outro bom exemplo disso são os chamados "*firewalls* reativos" [75]. Estes nada mais são que a união de tecnologias de *firewall* com sistemas de detecção de intrusão capazes de executar certas ações, como reconfigurar o *firewall* para bloquear certos pacotes, quando detectada alguma assinatura de ataque.

# Capítulo 4

# Serviços Inseguros

Neste capítulo serão abordadas, com mais detalhes, algumas importantes deficiências dos filtros de pacotes tradicionais e dos agentes *proxy*. Será visto que, para certos serviços, a melhor solução oferecida por estas tecnologias tradicionais de *firewall*, para eliminar riscos e prover um ambiente mais seguro, pode ser simplesmente evitar o uso de tais serviços. O objetivo aqui é mostrar que o simples bloqueio incondicional de todo o tráfego relativo a um serviço não é a melhor solução a ser adotada; novas estratégias e/ou tecnologias devem ser encontradas para que o *firewall* tenha um comportamento mais aceitável diante destes problemas. Filtros de pacotes com estados surgem, então, como uma excelente alternativa para a provisão de muitos destes serviços.

## 4.1 Introdução

Existem diversos serviços e protocolos de aplicação que podem ser responsabilizados pelo crescimento surpreendente da Internet. Muitos outros estão surgindo rapidamente sobre a infra-estrutura de rede atual como resultado deste sucesso, impulsionados pela tecnologia e devido à pressão de usuários cada vez mais exigentes. Além disso, o aumento na velocidade das linhas de comunicação e no desempenho das máquinas têm facilitado o desenvolvimento de protocolos mais complexos. Tais protocolos envolvem a movimentação de uma grande diversidade de dados pela rede, em quantidades variáveis e com exigências diferentes na qualidade de serviço. Por outro lado, a maioria destes serviços não trazem muitos mecanismos de segurança embutidos e, geralmente, herdam muitos dos problemas discutidos no capítulo 2. É importante, portanto, que as tecnologias de segurança sejam capazes de agregar segurança a estes serviços para que

usuários, em redes locais com maiores requisitos de segurança, possam usufruir destes atraentes serviços e protocolos.

A segurança de um serviço ou protocolo pode ser avaliada usando pontos de vista diferentes. Aquelas aplicações que lidam com informações confidenciais (senhas, números de cartão de crédito e outras), por exemplo, podem ser consideradas seguras, caso usem algoritmos criptográficos para a cifragem dos dados. Isto vale quando se avalia a segurança com base nos problemas relativos a um canal de comunicação inseguro. No entanto, estas mesmas aplicações podem não oferecer nenhum esquema seguro para gerenciamento e distribuição de chaves públicas no sistema e, com isso, seriam consideradas inseguras caso estivéssemos avaliando problemas referentes à autenticação. No restante deste trabalho, serão tratados como serviços inseguros aqueles serviços que oferecem problemas para serem filtrados pelas tecnologias tradicionais de *firewall*.

As tecnologias tradicionais de *firewall*, vistas no capítulo anterior juntamente com filtros de pacotes com estados e NAT, têm suas limitações e nem sempre conseguem tratar convenientemente todos estes serviços "inseguros" existentes na rede. Para muitos deles, a única alternativa para prover segurança é bloquear todo tráfego de pacotes relativo a tais serviços. Mas, com certeza, esta não é a solução desejada. Neste capítulo serão abordados, de forma geral, alguns destes serviços inseguros e os problemas relacionados à filtragem usando as tecnologias tradicionais.

## 4.2 Serviços Baseados em UDP

Como já visto na Seção 2.3, página 6, UDP é um protocolo de transporte não orientado a conexões e não confiável. A grande vantagem oferecida pelo UDP, em relação ao TCP, é o aumento no desempenho obtido com a eliminação dos processos de estabelecimento e encerramento das conexões. Além disso, o cabeçalho de pacotes UDP é mais simples, significando um menor número de *bytes* por pacote gasto com dados de controle do protocolo. Isto sem falar no tráfego extra dos pacotes para realizar os atributos de uma conexão. Em contrapartida, a aplicação fica com o trabalho mais árduo de garantir a reordenação e a retransmissão dos pacotes perdidos e de cuidar das duplicatas, dentre outras possibilidades existentes devido às características do protocolo de rede IP [22].

### 4.2.1 Filtros de Pacotes

Todo pacote UDP é independente, ou seja, não é parte de nenhum circuito virtual, como é o caso dos pacotes TCP. Portanto, não existem maneiras de distinguir, somente observando cabeçalhos dos pacotes UDP, uma requisição e uma resposta, sem consultar também alguma informação extra dos estados das sessões correspondentes [10]. Os cabeçalhos destes pacotes (ver Figura 4.1) transportam simplesmente as informações das portas de origem e destino envolvidas, o comprimento total e um *checksum*[1] do pacote UDP, não tendo nenhuma *flag* de sincronismo e números de seqüência para controlar a comunicação. Serviços baseados em UDP são, por este motivo, muito difíceis de filtrar com filtros de pacotes estáticos.

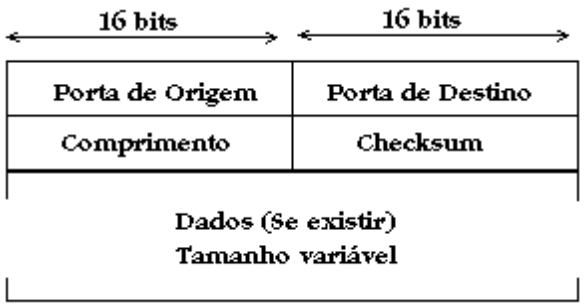


**Figura 4.1:** Formato de um Pacote UDP

Filtros de pacotes estáticos não mantém informações de estados. Toda filtragem é realizada usando as informações dos cabeçalhos de rede e transporte de cada pacote individualmente, como visto no capítulo anterior. No caso do TCP, filtros de pacotes estáticos podem filtrar serviços verificando se um pacote TCP é um pedido de conexão, ou seja, se ele tem o *bit* de ACK desligado [10]. Todos os outros pacotes TCP pertencentes a uma conexão estabelecida têm este *bit* ligado (Figura 4.2). Desta forma, para bloquear um serviço usando estes filtros, basta bloquear o primeiro pacote da conexão. Entretanto, para o caso de um pacote UDP, não existe maneira destes filtros descobrirem se este pacote se trata de uma requisição ou resposta. Neste caso, o

1. O valor do *checksum* pode ser zero, caso este não tenha sido calculado; no UDP o cálculo do *checksum* é opcional.

filtro pode permitir todo fluxo UDP, ou evitar as sessões UDP inteiramente para alguma porção do espaço de portas.

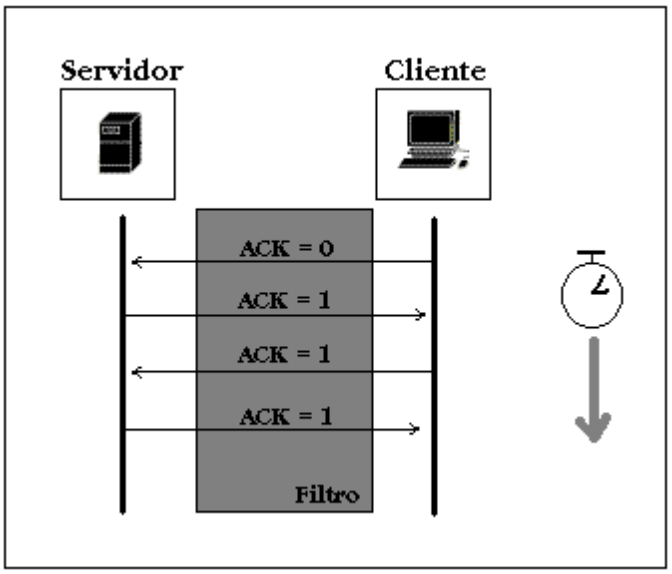


**Figura 4.2:** Filtragem de Pacotes TCP

Alguns filtros dinâmicos, como as primeiras implementações do Ipfilter[1], têm a capacidade de lembrar os pacotes UDP que saem da rede interna e só aceitam pacotes entrando nesta rede que correspondam a estes primeiros. Esta correspondência é feita mantendo-se algumas informações presentes nos cabeçalhos de rede e de transporte dos pacotes UDP saindo da rede protegida: porta de origem, endereço de origem, porta de destino e endereço de destino. Assim, se um pacote UDP saindo da rede interna tem endereço de origem A, porta de origem X, endereço de destino B e porta de destino Y, o filtro permitirá que qualquer pacote UDP entre na rede, caso ele tenha endereço de origem B, porta de origem Y, endereço de destino A e porta de destino X.

Infelizmente, apesar desta solução ser bastante interessante, ela ainda apresenta uma vulnerabilidade. Caso algum atacante descubra estas informações dos cabeçalhos dos pacotes UDP saindo da rede interna, ele pode negar serviços, criando respostas forjadas para estes pacotes.

1. Filtro de domínio público que tem, em suas últimas versões, características de um filtro de pacotes com estados.

Isso acontece porque a regra criada, para permitir estes pacotes no sentido inverso, é removida depois que a resposta atravessa o filtro [58]. Pode-se imaginar que isto seja tarefa difícil, a menos que o atacante tenha acesso ao tráfego UDP. Porém, na verdade ela pode ser bastante simplificada pelo modo de operação de diversos protocolos e serviços.

No DNS, por exemplo, servidores DNS dentro de uma rede local se comunicam com servidores DNS remotos para resolver nomes para os clientes internos. Observar que os endereços IP dos servidores podem ser facilmente conhecidos e que ambos usam a porta de número 53 na comunicação. Uma solução para este problema, adotada em alguns filtros dinâmicos mais recentes e filtros de pacotes com estados, é associar *timeouts* a estas entradas na tabela de regras dinâmicas, impedindo que pacotes forjados sejam capazes de remover as regras para os pacotes com respostas verdadeiras. As regras existiriam temporariamente, mesmo após a chegada do pacote com a suposta resposta.

### 4.2.2 Agentes *Proxy*

Implementar *proxies* para serviços e protocolos de aplicação também não é uma tarefa fácil. Como afirmado no início desta seção, todo o trabalho para suprir a falta de confiabilidade do protocolo de transporte UDP deve ser feito no nível de aplicação. Portanto, um servidor *proxy* para UDP, além de inspecionar e filtrar todo o fluxo de dados do serviço, deve tratar também do ordenamento e retransmissões dos pacotes e da tarefa de eliminar pacotes duplicados, tanto na comunicação com o servidor quanto na outra com o cliente. Isso pode tornar o *proxy* muito complexo e, portanto, sujeito a falhas de segurança na implementação do *software* [41]. Além disso, algumas aplicações, como o NFS, usam UDP pelo maior desempenho oferecido e o uso de um agente *proxy* pode comprometer bastante este desempenho. Devido a estes fatores, implementações de *proxies* de aplicação para serviços baseados em UDP são incomuns. Existem, porém, algumas implementações de *proxies* genéricos para serviços baseados em UDP, como SOCKS e UDP *Packet Relayer* [10], mas, como vimos na Seção 3.5.2, página 33, *proxies* genéricos não oferecem nenhuma filtragem no nível de aplicação, pondo em risco uma rede local.

Atualmente, o DNS é o serviço baseado em UDP mais usado em redes locais. Mais recentemente, muitos protocolos que fazem uso de UDP surgiram na Internet, principalmente aqueles baseados em comunicação em tempo real, os chamados "*Instant messages*". Alguns outros protocolos, bem mais importantes para redes corporativas, já têm sido propostos e implementados,

como é o caso do H.323 [56], que é um protocolo para telefonia sobre IP (VoIP, *Voice-over-IP*) definido pelo ITU-T, que usa ambos UDP e TCP. Portanto, é essencial a busca de melhores mecanismos de filtragem destes serviços.

## 4.3 Serviços Baseados em RPC

Existem vários protocolos de chamada a procedimentos remotos conhecidos como RPC. O mais popular e que será usado em nossa discussão, nesta seção, é o Sun RPC [20], que foi originalmente desenvolvido pela Sun Microsystems. Pode-se afirmar que o protocolo RPC estaria mais próximo do que seria a camada de sessão do modelo OSI; isso porque ele trabalha entre a camada de transporte e o nível de aplicação.

Nos protocolos UDP e TCP, os cabeçalhos dos pacotes reservam somente 2 *bytes* para os campos com números de portas, ou seja, existem somente 65.536 possíveis portas para todos os serviços TCP e UDP (65.536 para cada). Se tivesse que reservar um número de porta bem definido para cada serviço existente, ter-se-ia um número de serviços limitado por este valor. Entre outras coisas, RPC oferece uma boa solução para este problema [10]. Cada serviço baseado em RPC recebe um número de programa único de quatro *bytes* (isto permite 4.294.967.296 serviços diferentes). Como RPC é implementado acima do nível de transporte, deve existir algum mecanismo de mapeamento entre os números dos serviços RPC, oferecidos em uma máquina, e os números de porta que estes serviços estão usando em um dado momento. No Sun RPC, o Portmapper/Rpcbind é o responsável por este mapeamento e é o único servidor baseado em RPC que usa um número de porta pré-definido[1].

Quando um servidor RPC é iniciado, ele aloca um número de porta qualquer e registra-se junto ao Portmapper/Rpcbind. Nesta comunicação, ele passa o seu número de programa, a porta usada, o número de versão e o número do protocolo para o Portmapper/Rpcbind a ser usado na comunicação com os clientes. Vale comentar que o servidor usa o próprio protocolo RPC nesta comunicação [22]. Estas informações são, na verdade, os parâmetros para um procedimento PMAPPROC_SET a ser executado pelo Portmapper/Rpcbind. Um cliente, desejando comunicar-se com este servidor RPC, comunica-se com o Portmapper/Rpcbind para consultar o número de porta usado pelo serviço. Na verdade, ele também faz uma chamada remota ao procedimento

1. Porta 111, em ambos TCP ou UDP.

PROC_GETPORT, passando como parâmetros o número de programa, a versão e o protocolo usado. No momento em que o cliente obtém a porta usada pelo servidor, ele pode comunicar-se diretamente com o serviço RPC, sem nenhum envolvimento do Portmapper/Rpcbind. A Figura 4.3 apresenta todos os passos citados acima.

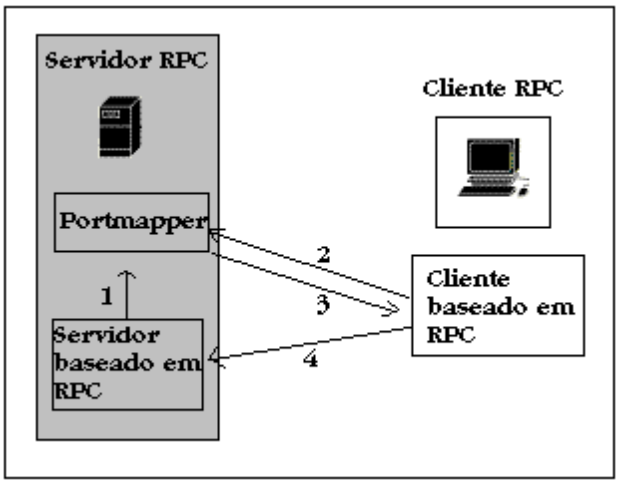


**Figura 4.3:** RPC e Portmapper/Rpcbind

Por não usarem números de portas pré-definidos, não se tem como criar regras estáticas para filtragem de serviços baseados em RPC [42]. A alocação de portas é feita dinamicamente e pode mudar com o passar do tempo (máquina ou servidor podem ser reinicializados, por exemplo). Bloquear tráfego de pacotes para o Portmapper/Rpcbind não resolve, pois um atacante pode descobrir o número da porta usada por algum serviço baseado em RPC, fazendo tentativas para todas as portas não reservadas. Outros serviços RPC podem usar números de portas pré-definidos, como é o caso do NFS que usa a porta 2049. Mesmo assim, é aconselhável bloquear acesso ao Portmapper/Rpcbind para evitar que eles sejam usados para ataques à própria máquina. O Portmapper/Rpcbind tem um recurso chamado *proxy forwarder*, que permite que clientes façam uma chamada ao procedimento PMAPPROC_CALLIT, passando como parâmetro o número de um programa RPC, sua versão, um número de procedimento deste serviço e os parâmetros para este procedimento. O Portmapper/Rpcbind faz a chamada ao procedimento do serviço, caso esteja registrado, e retorna os resultados obtidos para o cliente. Notar que isto pode ser perigoso

para alguns serviços, uma vez que o Portmapper/Rpcbind tem privilégios de super-usuário e a requisição parece vir de uma máquina confiável (a própria máquina) [10].

A solução encontrada para a filtragem de serviços baseados em RPC, com filtros de pacotes estáticos, é simplesmente bloquear todo tráfego de pacotes UDP, pois a grande maioria destes serviços usam UDP como protocolo de transporte [10]. Outro detalhe que se pode ressaltar neste momento é que, sendo a maioria destes serviços também baseados em UDP, apresentam todos os problemas da seção anterior.

No capítulo 7 serão discutidos com mais detalhes serviços baseados em RPC e a implementação de um filtro de pacotes com estados capaz de acompanhar e controlar toda a comunicação entre clientes e o Portmapper/Rpcbind. Desta forma, ele pode criar regras dinâmicas para permitir que estes clientes comuniquem-se com serviços RPC.

## 4.4 Outros Protocolos mais Complexos

Como já dito em outras oportunidades, têm surgido, com bastante rapidez, diversos protocolos importantes, que podem fazer uso de ambos UDP e TCP, envolvendo diversas sessões simultâneas. Um exemplo é o novo protocolo H.323 para o transporte de voz sobre o protocolo IP. Sem muitos detalhes (a compreensão deste protocolo exige conhecimentos que fogem ao escopo deste trabalho), este protocolo utiliza quatro sessões UDP diferentes para controle e uma conexão TCP para a comunicação entre as partes [56].

Estes protocolos, por usarem UDP além do TCP, têm todos os problemas de filtragem estática e de *proxies* discutidos na Seção 4.2, página 39. Um outro agravante para a implementação de agentes *proxy* para estes serviços é a complexidade. Um dos requisitos principais destas implementações é que elas sejam simples, minimizando as chances da existência de falhas de segurança [41]. Além disso, para o caso de várias sessões e conexões simultâneas, um *proxy* pode ficar mais rapidamente sobrecarregado com um número menor de clientes.

Portanto, as limitações das tecnologias tradicionais de *firewall* oferecem bons argumentos à busca de outras soluções para a provisão destes serviços inseguros. Filtros de pacotes com estados podem desempenhar muito bem este papel de prover com segurança serviços mais complexos. No próximo capítulo será apresentado, com mais detalhes, a tecnologia de filtragem com estados, explicando-se como eles podem oferecer suporte à serviços inseguros.

# Capítulo 5

# Filtros de Pacotes com Estados (*Stateful Packet Filter*)

Neste capítulo será apresentado, em mais detalhes, como funciona a filtragem de pacotes com estados e em que ela difere das tecnologias tradicionais de *firewall*. Ao longo do capítulo, será mostrado, entre outras coisas, como o SPF pode ser usado para prover com segurança alguns daqueles serviços inseguros vistos no capítulo anterior.

## 5.1 Introdução

Como discutido na Seção 3.3, página 22, filtros de pacotes tradicionais não mantêm informações de estados das conexões ou sessões que passam pelo *firewall*. Pacotes são analisados independentemente, usando somente as informações existentes nos seus cabeçalhos de rede e transporte e regras de filtragem. Se um pacote qualquer com as flags de sincronismo SYN e ACK ligadas chegar ao filtro de pacotes, o filtro supõe que este seja parte de algum canal virtual existente entre a rede interna e a rede externa. Silenciosamente, este o aceita, caso esteja de acordo com o conjunto de regras que implementa a política de segurança local.

Entretanto, um bom *firewall* não deve permitir que pacotes não válidos entrem na rede a ser protegida [48]. Pacotes não válidos são aqueles que não satisfazem as regras de filtragem e/ou não pertencem a nenhuma conexão ou sessão existente através do *firewall* [67]. Isto, além de garantir a perfeita implementação da política de segurança da rede interna, evita diversos ataques já bem conhecidos e outros que poderão aparecer no futuro usando pacotes devidamente construídos e/ou mal formados. Um bom exemplo disto são os pacotes ou fragmentos de pacotes,

que satisfazem às regras de filtragem, mas não pertencem a nenhuma conexão ou sessão existente entre a rede interna e a externa, enviados para derrubar uma máquina alvo, explorando alguma falha na implementação da pilha de protocolos TCP/IP de um determinado sistema operacional existente na rede interna. Aliás, um atacante pode descobrir os sistemas operacionais usados em máquinas internas, enviando alguma seqüência bem definida de pacotes com características não especificadas nos padrões dos protocolos e, de acordo com as respostas obtidas, inferir sobre o sistema usado pela máquina [66].

Em um filtro de pacotes com estados, quando um pacote SYN/ACK chega ao filtro, ele normalmente verifica, em suas tabelas de estados das conexões e sessões, se este pertence a alguma conexão em andamento através do *firewall*. Caso não exista uma conexão TCP válida para este pacote, ele é descartado e, possivelmente, registrado nos *logs*. Portanto, mesmo que o filtro esteja configurado com uma única regra permitindo todo tráfego de pacotes entre as redes, estes pacotes continuarão sendo bloqueados no *firewall*, já que não pertencem a nenhuma conexão válida segundo suas tabelas de estados [77]. Esta solução garante sempre uma maior segurança, visto que somente pacotes autorizados pelas regras de filtragem ou pertencentes às sessões e conexões válidas entre as redes podem atravessar o filtro. Uma outra vantagem desta solução é a melhora no desempenho, pois somente os primeiros pacotes das conexões ou sessões são verificados com as regras de filtragem, que geralmente é uma tarefa de mais alto custo computacional [34].

Vale ressaltar aqui que este é o comportamento mais normal das atuais implementações de filtros de pacotes com estados. Será visto mais adiante, neste capítulo, que nem sempre este processo de filtragem com estados trabalha exatamente desta forma em todas as implementações [77]. Portanto, não existe uma especificação padrão para a implementação de um SPF, ou seja, o termo "*stateful*" especifica que o filtro deve manter informações de estados das sessões e conexões, mas não como estas informações devem ser mantidas nem como elas devem ser usadas no processo de filtragem. O que será visto nas próximas seções são as características mais desejáveis e esperadas em filtros de pacotes com estados. Mas algumas exceções e suas respectivas soluções serão discutidas, sempre que possível. As informações aqui discutidas são baseadas principalmente nos produtos comerciais mais usados e conhecidos: Firewall-1 da CheckPoint e CISCO-PIX da Cisco e nas implementações de domínio público: Iptables (parte do *kernel* no *Linux* das séries 2.3 e 2.4) e Ipfilter.

## 5.2 UDP

Como em UDP não existem conexões e, por conseqüência, não há um processo de estabelecimento de conexão, para todos os pacotes UDP que chegam ao SPF verifica-se primeiramente se já existe, em suas tabelas de estados, alguma sessão para este pacote. Se este pacote UDP puder ser associado a alguma sessão existente, então ele é permitido continuar seu percurso entre as redes. Entretanto, se não existe uma sessão para este pacote, verifica-se, usando as regras de filtragem do filtro, se o mesmo é permitido passar pelo *firewall*, como ocorre na filtragem tradicional [77]. Em caso positivo, além de permitir a sua passagem, o filtro assume que este pacote é o primeiro pacote de uma sessão e cria uma nova entrada na tabela de estados do SPF.

Cada entrada para uma sessão UDP usa normalmente as informações existentes nos cabeçalhos dos pacotes da sessão: endereços de origem e destino, números de portas de origem e destino e um *timeout* associado. Portanto, todo pacote recebido com estas informações no cabeçalho são assumidos pertencer a tal sessão. Este *timeout* evita que a entrada fique indefinidamente na tabela de estados (que em muitas implementações tem tamanho limitado) e é reinicializado toda vez que um pacote para esta sessão passa pelo *firewall*. Um outro importante detalhe que deve ser lembrado é que este *timeout* deve ser suficientemente longo para evitar o bloqueio das comunicações válidas. No caso do Firewall-1 (FW-1) da CheckPoint, por exemplo, este *timeout* é de 40 segundos [77].

Caso o tempo de vida de uma entrada na tabela seja muito curto, poderá acontecer de um cliente ficar reenviando alguma requisição a um servidor remoto e jamais receber a resposta correspondente. Isso acontece porque quando a requisição é enviada, ao passar pelo *firewall*, o filtro cria uma entrada na tabela de estados das sessões e inicializa o *timeout* associado. Entretanto, este *timeout* expira antes que a resposta do servidor remoto chegue e, consequentemente, todas as outras sucessivas tentativas do cliente terão o mesmo resultado. Algum tempo depois, quando a resposta do servidor chegar ao filtro, a entrada para a sessão não mais existirá na tabela de estados e, como este pacote não é permitido pelas regras de filtragem, ele será descartado.

Um outro aspecto importante em filtros de pacotes com estados, tratado mais adiante, é que algumas implementações conseguem associar pacotes ICMP às suas sessões e conexões [80]. No caso do UDP, pacotes com ICMP *port unreachable*, por exemplo, poderiam chegar ao seu des-

tino correto e evitar que um cliente interno ficasse fazendo tentativas sucessivas, sem a necessidade de uma regra específica para isso.

Como UDP também não tem nenhum processo de encerramento de conexões, o filtro não tem como saber quando a sessão termina exatamente e, portanto, a entrada para uma sessão é removida da tabela de estados somente quando expirar o tempo de vida da entrada na tabela.

## 5.3 TCP

A filtragem de conexões TCP é levemente diferente de sessões UDP. Como TCP é orientado a conexões, é suficiente verificar o primeiro pacote do processo de estabelecimento da conexão, junto ao conjunto de regras de filtragem (pacote com o *bit* ACK desligado). Caso esta conexão seja permitida, uma entrada na tabela de estados é criada para permitir a passagem dos demais pacotes da conexão [73]. Portanto, para pacotes pertencentes a uma conexão já estabelecida, verifica-se se o canal virtual a que este pacote está associado tem uma entrada na tabela de estados das conexões. Somente se existir uma entrada é permitida a passagem do pacote.

Entretanto, no FW-1 este processo é um pouco diferente. Neste produto, quando qualquer pacote chega ao filtro e não pertence a nenhuma conexão existente na tabela de estados, ele verifica o pacote usando as regras de filtragem e, caso seja permitido, cria uma entrada para uma nova conexão, como se este pacote realmente fosse o primeiro pacote da conexão. Só que neste caso, o filtro retira a parte de dados do pacote e modifica números de seqüência antes de permitir a sua passagem. Se a resposta para este pacote for um pacote com bit RST ligado, ele descarta este último e remove a entrada da conexão da tabela de estados [73]. Desta forma, o pacote não causaria prejuízos ao destino e nenhuma informação poderia ser obtida sobre a máquina interna. Isso não somente é válido para pacotes SYN/ACK, mas também para pacotes Null, FIN/ACK e vários outros pacotes mais exóticos, com exceção, obviamente, para os pacotes com FIN ou RST válidos [77]. Estes últimos são usados para encerrar uma conexão e, portanto, ao chegarem ao filtro, o FW-1 reduz o valor do *timeout* para a conexão de 3600 segundos para 50 segundos, simulando um estado semelhante ao de TIME_WAIT do TCP.

A explicação para este estranho comportamento que foi dada em [77], é a disponibilidade: em caso de falhas ou quando as regras de filtragem são atualizadas durante o funcionamento normal do filtro, deseja-se que nenhuma conexão em andamento seja afetada. Portanto, esta solução possibilita evitar que as conexões válidas sejam abortadas. Notar que UDP não tem uma

solução similar, já que os clientes internos têm a responsabilidade de retransmitir suas requisições, recriando a entrada na tabela de estados das sessões UDP, se não receberem a resposta esperada após algum tempo.

Quanto às entradas na tabela de estados relativas às conexões TCP, as soluções usadas pelas diversas implementações diferem um pouco. O caso mais simples é semelhante ao que se adota para as sessões UDP: endereços de origem e destino, portas de origem e destino e um *timeout*. O CISCO-PIX da Cisco, a versão atual do Iptables (versão 1.0), que será examinada em detalhes no próximo capítulo, e o FW-1 mantêm estados das conexões TCP com estas informações. Isso significa que pacotes TCP que tenham estas mesmas informações (com exceção do *timeout*, é claro) nos seus cabeçalhos são considerados parte da conexão.

As últimas versões do Ipfilter (versões 3.3.0 em diante), assim como as futuras versões do Iptables adotam uma solução mais segura para evitar ataques de DoS ao filtro [86]. Mantém-se, além daquelas informações, os números de seqüência e tamanhos das janelas de recepção (parâmetro "*window*" do TCP) usadas na conexão. Portanto, há um acompanhamento mais rigoroso do andamento das conexões. Entretanto, o emprego destas informações consome mais recursos e uma inteligência maior do filtro. Neste caso, o filtro conclui que os pacotes fazem parte de uma conexão, considerando também os números de seqüência e o tamanho da janela do receptor, além de simplesmente checar endereços e números de portas [73]. Além disso, os números de seqüência e os tamanhos das janelas de recepção nas tabelas de estados vão sendo convenientemente atualizados com o passar do tempo, acompanhando o protocolo de janela deslizante usado pelo TCP. Cuidados especiais, porém, devem ser tomados na implementação do filtro para evitar que alguns pacotes atrasados ou perdidos[1] na rede bloqueiem todo o tráfego válido de uma conexão. Um problema como este, por exemplo, existia em antigas implementações do Ipfilter [73].

Diferentemente do que acontecia para o caso das sessões UDP, o filtro pode remover uma entrada na tabela de estados quando chegar um pacote com as *flags* RST ou FIN ligados em um pacote da conexão. Neste caso, o pacote atravessa o firewall e a entrada para a conexão é removida da tabela de estados. Isso ocorre nas atuais versões do CISCO-PIX [86]. Outras implementa-

1. Alguns pacotes que passam pelo filtro podem não chegar ou podem chegar fora de ordem ao seu destino.

ções optam por manter a entrada ainda por um período mais curto de tempo para dificultar ataques de DoS, como visto no caso do FW-1.

## 5.4 ICMP

Para pacotes ICMP saindo da rede protegida com alguma requisição, como *echo request* ou *timestamp request*, um filtro de pacotes com estados usa a mesma estratégia adotada no caso das sessões UDP. Desta forma, é possível que usuários internos possam fazer *ping* para redes remotas, criando somente uma regra para permitir que pacotes *echo request* atravessem o *firewall*. Esta é uma outra importante vantagem de filtros de pacotes com estados em relação a filtros tradicionais: o conjunto de regras é menor e mais simples [77].

Por manter estados das conexões e sessões, um filtro de pacotes com estados permite também associar pacotes ICMP de erro ou controle à sua devida conexão ou sessão [80]. Como visto na Seção 2.6, página 12, estes pacotes trazem informações (ver o formato destes pacotes na Figura 5.1) que permitem identificar a conexão ou sessão da máquina à que se refere a mensagem ICMP. Usando estas informações, o filtro pode verificar a existência desta sessão ou conexão em suas tabelas de estados e permitir a passagem. O Iptables e o Ipfilter são exemplos de SPFs que fazem esta verificação.

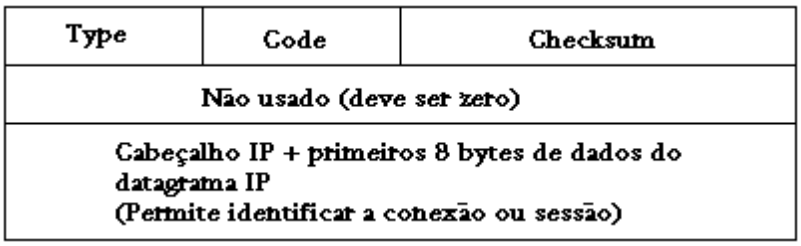


**Figura 5.1:** Formato de Pacotes ICMP de Erro

## 5.5 Filtragem no Nível de Aplicação

Para alguns serviços e protocolos de aplicação, como visto no caso do FTP, algum trabalho extra precisa ser feito no nível de aplicação. Para estes casos especiais, filtros de pacotes com estados podem acompanhar o fluxo de dados da aplicação e extrair as informações necessárias para suas

tabelas de estados. No caso do FTP, o filtro pode extrair, por exemplo, a porta que o cliente usará para uma conexão de dados. Desta forma, ele cria uma regra dinâmica ou, como no caso do Iptables, cria uma entrada em uma lista de conexões aguardadas.

Por já fazerem algum trabalho nesta camada de comunicação para alguns serviços, alguns SPFs, como o CISCO-PIX e o FW-1, permitem também alguma filtragem no nível de aplicação, como em agentes *proxy* [30]. Entretanto, a filtragem neste caso é geralmente bastante simples, verificando somente a parte de dados dos pacotes individualmente, sem qualquer remontagem do fluxo de dados do serviço [29]. Este tópico retornará à discussão na Seção 5.8, página 55.

Apesar disso, muitos serviços de aplicação transmitem comandos do protocolo e suas respectivas respostas em unidades pequenas, que cabem em um único datagrama IP e podem perfeitamente ser filtrados por filtros de pacotes com estados[1]. Um bom exemplo é o caso da comunicação entre clientes RPC e o Portmapper. As mensagens enviadas pelos clientes, requisitando o número da porta usada por algum servidor RPC, ocupam somente 56 *bytes*, enquanto uma resposta com o número da porta usada ocupa 28 *bytes* da parte de dados[2]. Nestes casos, não há a necessidade de remontagem dos dados dos pacotes de uma comunicação para filtrar comandos do serviço ou protocolo de aplicação. A grande vantagem obtida com esta filtragem é que o SPF desempenha o papel de um agente *proxy* dedicado sem comprometer o desempenho.

## 5.6 Ataques de DoS a Filtros de Pacotes com Estados

Como visto na Seção 5.3, página 49, o FW-1 tem um comportamento um pouco diferente para prover uma disponibilidade mais alta. Entretanto, descobriu-se que a possibilidade de criação de uma entrada na tabela de estados das conexões, com a chegada de um pacote SYN/ACK ou qualquer outro, pode trazer sérias implicações para ataques de DoS neste produto [77]. Sempre que uma conexão é acrescentada à tabela de estados, o *timeout* associado é automaticamente inicializado para 3600 segundos (por *default*). Ao enviar vários destes pacotes para endereços que

1. Neste caso, a fragmentação e qualquer tentativa de manipulação de certos atributos do TCP, capazes de forçar a troca de *streams* de dados cuidadosamente truncados, pode ser encarado como uma tentativa de ataque. Pacotes com estas características devem ser descartados e logados. Estes cuidados evitam que a filtragem com estados seja subvertida por pacotes maliciosamente truncados.
2. Usando o protocolo UDP.

não existem (não obterão pacotes RST de volta e, portanto, o timeout continuará sendo de 1 hora), pode-se lotar a tabela de estados do SPF. Portanto, implementações de filtros de pacotes com estados têm que dar atenção especial aos *timeouts* usados e ao tamanho das tabelas de estados das conexões para minimizar estas ameaças.

Algumas implementações de SPFs podem sofrer ataques de DoS usando pacotes RST forjados. O CISCO-PIX [86] e o Iptables[1] são vulneráveis a este ataque. Para estes filtros, a verificação de que um pacote pertence a uma determinada conexão baseia-se somente em endereços de origem e destino e portas de origem e destino no cabeçalho do pacote. Isso permite que um atacante crie, mais facilmente, pacotes RST forjados para remover (no caso do CISCO-PIX) ou reduzir o *timeout* (no caso do Iptables 1.0), impedindo o andamento normal das conexões e, portanto, negando serviço. O FW-1 não tem este problema, pois os pacotes válidos podem reconstruir normalmente a entrada para a conexão na tabela de estados, caso ela tenha sido removida [77]. Mas a melhor solução para este problema é incluir também, nesta verificação de pertinência, números de seqüências e tamanhos das janelas de recepção como é feito no Ipfilter, dificultando bastante a criação destes pacotes RST [73].

## 5.7 Ataques no Nível de Aplicação

Na Seção 5.5, página 51, foi citado que filtros de pacotes com estados fazem somente filtragens simples no nível de aplicação, sem, por exemplo, a tarefa de remontagem dos dados da aplicação. Para alguns serviços e protocolos, isso já é suficiente mas, em outros casos, como o FTP, isso pode trazer problemas, como será visto nesta seção.

Recentemente descobriu-se uma vulnerabilidade existente no CISCO-PIX, FW-1 e no Iptables que permite a um atacante usar um servidor FTP, protegido pelo *firewall*, para subverter o SPF e induzí-lo a permitir tráfego de pacotes para qualquer porta desejada em máquinas internas. Este ataque usa o fato destes filtros de pacotes com estados não remontarem todo o fluxo de dados para a aplicação FTP e simplesmente basearem-se na busca por certas seqüências de caracteres ASCII na parte de dados de cada pacote individualmente [52].

Como já mencionado neste trabalho, o protocolo FTP tem um modo de operação alternativo chamado FTP passivo. Neste modo, o servidor passa para o cliente o número da porta que usará

1. Esta vulnerabilidade não mais estará presente nas futuras versões deste SPF.

na conexão de dados. Nesta comunicação, o servidor envia sempre o padrão de caracteres ASCII "*227 Entering Passive Mode*", antecedendo as informações de endereço IP do servidor e a porta a ser usada. Desta forma, a conexão de dados é iniciada pelo cliente usando este número de porta no servidor. SPFs monitoram a conexão de dados em busca deste padrão e, desta forma, podem obter o número da porta que o servidor usará na conexão de dados com o cliente.

Entretanto, um atacante pode simplesmente forçar o servidor a enviar alguma mensagem de erro, na qual este padrão ASCII estaria presente, e o *firewall* interpretaria como sendo informações válidas do servidor para o modo passivo. Por exemplo:

/* Comando do atacante */
GET AAAA[ vários caracteres "A"]AAAAA**227 Entering Passive Mode** 140, 252, 13, 34, 4, 150

Neste caso , o servidor retornará uma mensagem de erro, na conexão de controle, dizendo que "AAAA[ vários caracteres "A"]AAAAA227 Entering Passive Mode 140, 252, 13, 34, 4, 150" é um nome de arquivo inválido. Se o atacante for cuidadoso na escolha da quantidade de caracteres "A" e usar um parâmetro TCP MSS[1] (*Maximum Segment Size*) apropriado, o SPF pode acreditar que a mensagem se trata realmente de uma resposta do servidor para entrar no modo passivo.

Neste caso, manipulando estes valores, um atacante pode fazer com que o servidor envie esta mensagem de erro truncada em diversos pacotes. O sucesso do ataque depende de que um destes pacotes apresente somente "227 Entering Passive Mode 140, 252, 13, 34, 4, 150" na sua parte de dados. Então, o filtro de pacotes com estados, fazendo o *match* pelo padrão *"227 Entering Passive Mode"*, cria uma regra dinâmica para permitir que o cliente se comunique usando a porta 1174[2] na máquina interna 140.252.13.34.

Isso prova que estas implementações comerciais de SPFs não remontam completamente o fluxo de dados dos serviços e protocolos de aplicação e, portanto, não podem fazer todas as tarefas realizadas em agentes *proxy* dedicados [29]. Além disso, a ausência de remontagem dos dados exige que o SPF mantenha mais informações de contexto para evitar que mensagens de erro, dentre outras, possam ser incorretamente interpretadas pelo filtro. Uma das soluções ado-

1. Este valor é informado por cada participante durante o *setup* da conexão. Este atributo indica o tamanho do buffer de dados reservado para a conexão.
2. 1174 = 4 x 256 + 150

tadas, para este problema do FTP, é acompanhar cada *carriage return* na parte de dados dos pacotes, para facilitar uma correta delimitação dos dados da aplicação.

## 5.8 Comparação entre SPFs e as Tecnologias Tradicionais de Firewall

Teoricamente, é possível construir um *proxy* completo em um filtro de pacotes com estados [30]. Mas, diferentemente de um *proxy*, ele envolve no seu processo de filtragem todas as camadas de comunicação, desde os cabeçalhos de rede e transporte até a parte de dados do nível de aplicação, permitindo aliar as principais vantagens de filtros de pacotes e agentes *proxy*. Desta forma, o processo de filtragem torna-se agora mais homogêneo: uma só entidade vai fazer a análise completa do tráfego de dados. Entretanto, na prática, os vendedores optam por produtos mais "leves" e, quando um trabalho maior precisa ser feito na camada de aplicação, eles recorrem a *proxies* para o serviço [29].

Tome-se o Firewall-1 como exemplo, embora o mesmo possa ser observado em outros produtos de filtros de pacotes com estados. Neste, existem dois módulos básicos [31]: o Firewall-1 Inspection Module (também chamado "INSPECT engine") e o security server (ver Figura 5.2). No primeiro, reside o filtro de pacotes com estados propriamente dito. Ele é implementado no *kernel* do sistema operacional entre as camadas de enlace de dados e rede. Com o INSPECT engine é possível filtrar pacotes usando todas as camadas de comunicação e estados de comunicações passadas. Muito do que pode ser feito em alguns *proxies* pode ser feito também neste módulo, porque ele implementa alguma filtragem, similar a de *proxies* mais simples para os serviços. Entretanto, se o usuário quiser fazer um trabalho maior na camada de aplicação, ele precisa usar o segundo módulo do produto. Notar que este trabalho pode ser a tarefa de remontar e inspecionar todo o fluxo de dados. Os security servers rodam no espaço de processos e existem para FTP, HTTP, SMTP e também para fins de autenticação. Na verdade, eles apresentam as mesmas características que *proxies* para estes serviços [30].

Assim, quando o usuário desejar autenticação extra de usuários, fazer inspeção no fluxo de dados à procura de vírus, bloquear *applets* Java ou fazer análises mais sofisticadas no fluxo de dados, o Firewall-1 deixa estes security servers realizarem o trabalho. A vantagem desta solução é que o filtro no *kernel* fica mais simples, eliminando o *overhead* de um *proxy* implementado junto ao filtro e garantindo, portanto, um bom desempenho com segurança [30].

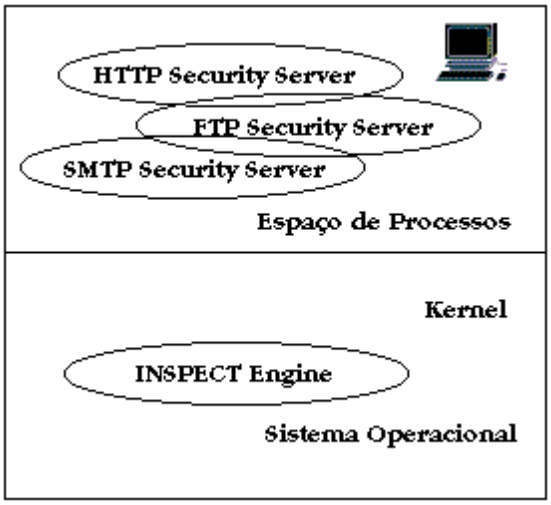


**Figura 5.2:** INSPECT Engine e Security Servers

Uma boa prova disto é a vulnerabilidade mostrada na seção anterior. Aquele problema revela que o fluxo de dados para FTP não é completamente remontado. Quando se deseja fazer alguma filtragem maior no nível de aplicação, como filtrar os comandos PUT para o servidor FTP interno, todo o tráfego precisa ser remontado/normalizado e o FTP security server se encarrega da tarefa.

Na prática, portanto, a maioria dos produtos que implementam filtros de pacotes com estados recorrem a soluções semelhantes a *proxies* (os security servers, no caso particular do Firewall-1) sempre que é necessário fazer um trabalho maior com os dados da aplicação. Por este motivo, muitos acreditam que *proxies* de aplicação são mais seguros que filtros de pacotes com estados [29].

Por outro lado, é inegável que filtros de pacotes com estados podem ser uma solução suficiente em diversos ambientes computacionais onde os riscos são baixos e em ambientes com exigências de desempenho. Se algum trabalho mais sofisticado deve ser feito como, por exemplo, bloquear instâncias específicas de Java ou buscar vírus no fluxo de dados, deve-se usar proxies dedicados a este trabalho, em adição à filtragem por estados. No caso de alguns produtos comerciais como o Firewall-1, é possível que estes *proxies* já sejam incluídos no próprio produto.

No caso do suporte a serviços e protocolos novos, as poderosas linguagens de *scripts* incluídas nestes produtos comerciais que implementam filtros de pacotes com estados, para a escrita de regras de filtragem, podem permitir uma solução mais segura que *proxies* genéricos ou neu-

tros [32]. Além disso, para aplicações desenvolvidas dentro de um ambiente corporativo, por exemplo, para as quais não existem *proxies*, pode ser bastante interessante usar um filtro de pacotes com estados para estabelecer regras de filtragem para tal serviço.

É válido ressaltar ainda que, por seu baixo custo computacional e por sua segurança adicional, filtros de pacotes com estados são sempre preferidos a filtros de pacotes estáticos e dinâmicos. Como já foi dito anteriormente, um filtro de pacotes com estados pode ser considerado uma evolução dos filtros de pacotes dinâmicos.

## 5.9 Serviços Inseguros

No capítulo anterior, viu-se que certos serviços, definidos aqui como serviços inseguros, oferecem problemas para serem tratados pelas tecnologias tradicionais. Com filtros de pacotes tradicionais e agentes *proxy*, a provisão destes serviços pode abrir espaços no *firewall*, que tornam a rede local menos segura.

Para serviços baseados em UDP, SPFs podem adicionar alguma noção de estados, ou seja, somente respostas a requisições válidas, segundo análise do *firewall*, são permitidas trafegar entre as redes. Esta solução é bem mais interessante do que a utilizada em filtros de pacotes estáticos, que simplesmente abrem parte do espaço de portas, expondo a rede interna. Além disso, o SPF permite que a filtragem também seja feita no nível de aplicação sem comprometimento no desempenho [32]. No capítulo 8, será visto como isso pode ser feito na comunicação entre um cliente RPC com o portmapper/rpcbind usando UDP.

A grande maioria dos serviços baseados em UDP envolve a troca de pequenas unidades de dados na comunicação e, portanto, podem ser perfeitamente filtrados na camada de aplicação por SPFs[1]. Um bom exemplo é o DNS. Este serviço baseia-se somente em um pacote UDP com a requisição e um outro com a resposta (TCP é usado somente quando esta resposta é maior que 512 bytes [22] ). Muitas outras aplicações, como "*Instant messages*", usam UDP somente para tarefas mais simples, como transmitir ao servidor poucas informações sobre o *status* do cliente. Nestes casos, UDP é preferido ao TCP, por eliminar os processos de estabelecimento e encer-

1. Nao esquecendo, obviamente, de tomar cuidados especiais contra ataques usando fragmentos e/ou pacotes truncados.

mento da conexão. Agentes *proxy* para estes serviços também seriam mais simples de implementar, porém o SPF, pelo menor custo computacional, é uma melhor opção.

Para serviços mais complexos, que usem sessões UDP e TCP simultaneamente, filtros de pacotes com estados são muito interessantes. Um bom exemplo é o serviço Real Audio [27], para transmissão de áudio na Internet. Neste serviço, como pode ser visto na Figura 5.3, o cliente estabelece uma conexão TCP (uma espécie de conexão de controle) usando a porta 7070 do servidor. Usando esta conexão, ele envia suas requisições e números de portas UDP[1] que ficarão esperando os dados do servidor. Múltiplas sessões UDP serão usadas para aumentar a vazão, garantindo a qualidade do som. Esta quantidade de portas usadas pode sobrecarregar muito mais rapidamente *proxies* dedicados. Além disso, o aumento no *overhead* pode prejudicar a qualidade do som. Um SPF, por outro lado, pode acompanhar a conexão TCP e permitir as sessões UDP muito facilmente.

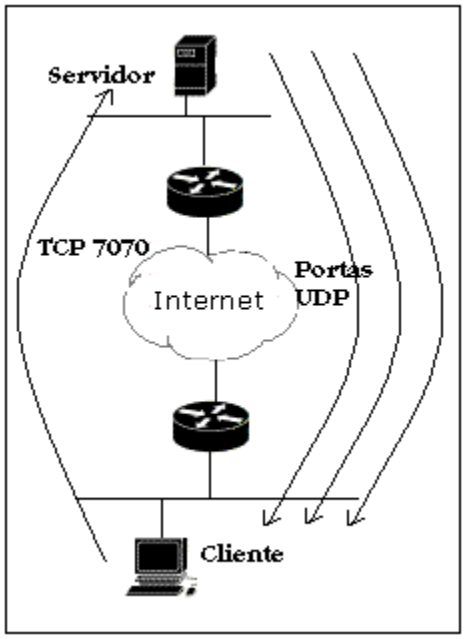


**Figura 5.3:** Real Audio

1. TCP também poderia ser usado, mas causaria uma degradação considerável na qualidade do som.

Quanto aos serviços baseados em RPC, nos próximos capítulos será mostrado como este autor implementou um filtro de pacotes com estados capaz de oferecer suporte a estes serviços. Neste trabalho, foi usado o Iptables como ponto de partida e criados módulos auxiliares para efetuar a filtragem com estados de serviços inseguros que utilizam o protocolo RPC.

# Capítulo 6

# Iptables: Um Filtro de Pacotes com Estados no *Kernel* do Linux

Como visto no capítulo anterior, o Iptables é o filtro de pacotes com estados implementado no *kernel* do Linux nas séries 2.3 de desenvolvimento e 2.4 de produção. Neste capítulo, serão mostrados mais detalhes sobre o mecanismo de filtragem com estados realizado por este SPF. Será visto que, na verdade, um outro módulo do *kernel* é o responsável por manter os estados de todas as conexões e sessões através do *firewall*. Estas informações de estados são então usadas de maneira independente pelos processos de tradução de endereços e filtragem de pacotes, dentre outros.

## 6.1 Introdução

Na Seção 3.3, página 22, foi visto que alguns sistemas operacionais trazem o processo de filtragem de pacotes implementado no núcleo do sistema. Nas versões 2.0.x e 2.2.x de produção do *kernel* do Linux, por exemplo, pode-se encontrar respectivamente os filtros Ipfwadm e Ipchains, que implementam simplesmente uma filtragem estática dos pacotes. Neste trabalho, entretanto, tem-se mais interesse na nova infra-estrutura [18] usada para tratamento e filtragem dos pacotes que estará presente na série de produção 2.4 do *kernel*, neste sistema operacional.

Como ainda não há uma versão estável deste novo *kernel*, muito do que será apresentado neste capítulo é baseado nas versões 2.3.x de desenvolvimento e poderá sofrer pequenas mudanças até a sua versão final. Contudo, as idéias básicas usadas na implementação certamente permanecerão as mesmas. Esta arquitetura surge para resolver antigos problemas existentes nas

versões anteriores do núcleo deste sistema operacional, que fogem ao escopo deste trabalho, e trazem outras grandes melhorias.

Entre as novas características que poderão ser encontradas no *kernel* 2.4.x do sistema operacional Linux e que conduziram à escolha desta plataforma para o desenvolvimento deste trabalho, merecem destaque a implementação do processo de filtragem de pacotes com estados, a modularidade e a facilidade para implementar e/ou estender as diversas funcionalidades que realizam seu trabalho com os pacotes que chegam ao *kernel* [57]. Estas funções são implementadas como módulos independentes, os quais podem ser acoplados ao *kernel* de acordo com as necessidades do usuário. Este capítulo estará concentrado nos módulos de acompanhamento das conexões e sessões e de filtragem de pacotes, que juntos possibilitam uma filtragem também baseada em informações de estados. As estruturas de dados citadas neste capítulo e algumas outras usadas na implementação destes módulos podem ser encontradas no apêndice A.

## 6.2 Netfilter

O Netfilter é um *framework* independente da interface normal de Berkeley *Sockets*, que gerencia e organiza como os pacotes que chegam ao *kernel* do Linux devem ser tratados, nas séries de desenvolvimento 2.3 e de produção 2.4 [57]. Ele define, por exemplo, que partes do *kernel* podem realizar algum trabalho com o pacote quando este atingir certos pontos do núcleo do sistema operacional.

Nesta nova infra-estrutura, cada protocolo define "*hooks*" (IPv4 define 5), que são pontos bem definidos na pilha de protocolos por onde passam os pacotes. Quando um pacote chega a um determinado *hook*, o Netfilter checa que módulos do *kernel* estão registrados para oferecer algum tratamento especial naquele ponto. Caso existam, cada módulo seqüencialmente recebe o pacote (desde que, obviamente, ele não seja descartado ou passado para o espaço de processos por um módulo anterior) e pode, portanto, examiná-lo, alterá-lo, descartá-lo, injetá-lo de volta ao *kernel* ou passá-lo para o espaço de processos, dependendo da funcionalidade de rede que esteja implementando. Estes módulos geralmente registram-se em mais de um *hook* ao mesmo tempo, podendo, até mesmo, fazer trabalhos diferentes com os pacotes em cada um deles.

Desta forma, pode-se definir quais módulos devem ser carregados e registrados perante o Netfilter, sempre que algum tratamento especial com os pacotes for necessário. Além disso, esta

interessante infra-estrutura facilita a implementação de novos módulos e permite que os módulos pré-existentes sejam facilmente estendidos, usando interfaces de comunicação bem definidas. Alguns dos módulos nativos são o NAT, o Iptables e o módulo de *Connection tracking*.

Notar que, diferentemente das versões mais antigas do *kernel* do Linux, as diversas funções de rede são implementadas em módulos independentes. Nas versões 2.0.x e 2.2.x, o IP *masquerading*, por exemplo, era implementado junto ao processo de filtragem de pacotes e das funções de roteamento no *kernel* [55]. Isso tornava o código confuso e acarretava alguma perda no desempenho. Com este novo esquema, IP *masquerading*, por ser um caso especial de NAT, é implementado usando o módulo de NAT. Este último, por sua vez, é totalmente independente do módulo de filtragem.

### 6.2.1 Netfilter e IPv4

Na Figura 6.1, página 63, pode-se verificar que o IPv4 define 5 *hooks* [57]. Neste caso, quando um pacote chega ao kernel, ele passa inicialmente por uma série de testes de sanidade. Verifica-se aqui, entre outras coisas, se o pacote é truncado e se há problemas com o *checksum* do cabeçalho IP. A seguir, o pacote é passado para o *hook* NF_IP_PRE_ROUTING. Ele recebe este nome porque o pacote, ao atingir este ponto na pilha de protocolos, não terá ainda passado pelo processo de roteamento no *kernel*. Em alguns casos, como no processo de tradução de endereços, já é necessário realizar alguma tarefa com os pacotes neste ponto. O módulo de NAT, portanto, deve ser registrado neste *hook* para reescrever endereços nos pacotes entrando no sistema.

Em seguida o pacote chega ao código de roteamento, onde se verifica se ele é destinado a algum processo local ou algum outro *host*. Vale mencionar que o pacote pode também ser descartado neste momento, caso não exista uma rota para ele. Se o pacote é destinado a algum processo local na máquina, o mesmo é passado para o *hook* NF_IP_LOCAL_IN, antes de ser entregue ao processo destino (se existir). Caso o destino seja um outro *host*, o pacote chega ao *hook* NF_IP_FORWARD e, por fim, o pacote é entregue ao *hook* NF_IP_POST_ROUTING antes de ser enviado pela rede. Para tratar os pacotes originados por algum processo local, antes de atingirem o processo de roteamento[1], existe ainda o *hook* NF_IP_LOCAL_OUT.

---

1. A partir do roteamento, o pacote originado por um processo local segue o mesmo percurso dos outros pacotes vindos de alguma interface de rede.

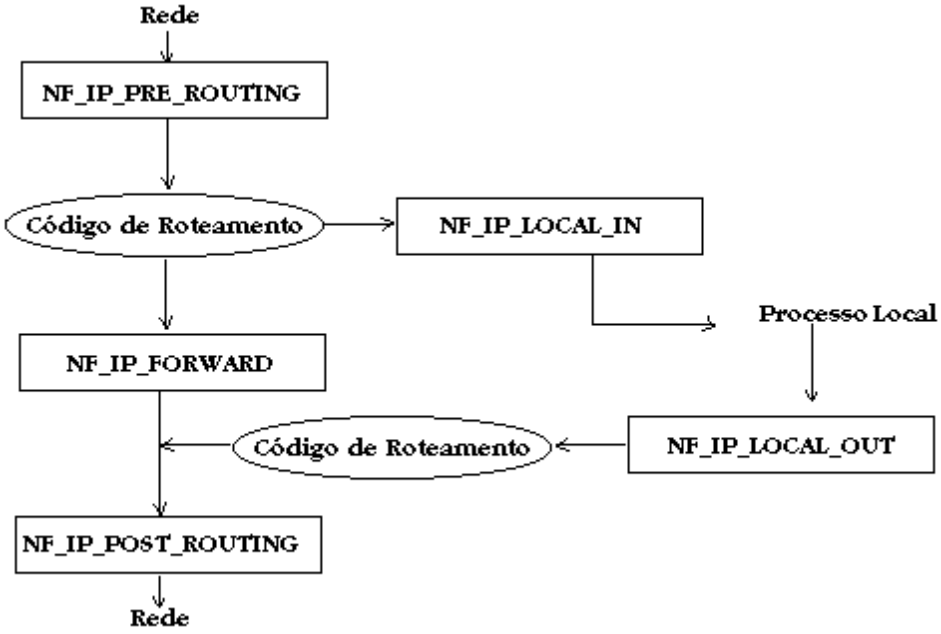


**Figura 6.1:** IPv4 e seus *Hooks*

## 6.3 Módulo de *Connection Tracking* (Conntrack)

Este módulo foi originalmente implementado para ser usado pelo módulo de NAT. Todavia, possibilita também que os outros módulos no *kernel* possam utilizá-lo. O papel do módulo Conntrack é manter informações de estados de todas as conexões e sessões [57]. O módulo Iptables, em suas versões iniciais, não fazia filtragem com estados, mas foi rapidamente e facilmente estendido para basear seu processo de decisão também nas informações de estados mantidas pelo módulo Conntrack [80]. Portanto, a grosso modo, este módulo de *Connection tracking* é o verdadeiro responsável pela parte "*stateful*" do filtro de pacotes implementado no *kernel* do Linux.

Este módulo deve oferecer tratamento para os pacotes antes que entrem realmente no sistema e, desta forma, ele se registra aos *hooks* NF_IP_LOCAL_OUT (para tratar os pacotes originados localmente) e NF_IP_PRE_ROUTING (para tratar os pacotes chegando por uma das interfaces de rede). Assim, é possível manter estados das sessões e conexões independente das outras partes do *kernel*.

Cada pacote chegando ao *kernel* é associado a uma estrutura de dados (skb_buff). Todos os componentes do núcleo do sistema operacional, que realizam algum trabalho com um determinado pacote, recebem o apontador para a estrutura de dados correspondente. Desta forma, os cabeçalhos e dados do pacote não precisam ser recopiados durante sua travessia pela pilha de protocolos [82]. Portanto, esta estrutura de dados, dentre outras coisas, tem apontadores para os cabeçalhos dos diferentes níveis de rede e para a parte de dados do pacote. Nas séries 2.3 e 2.4, ela apresenta um campo extra (nfmark), no qual o módulo Conntrack codifica o estado para o pacote em sua sessão ou conexão. Vale notar que este módulo não deve descartar pacotes, com exceção de alguns casos excepcionais (pacotes mal formados). Esta função é deixada para o módulo de filtragem. Portanto, depois de passar pelo módulo Conntrack, um pacote atravessa o *kernel* com o seu respectivo estado, que pode ser:

- ***INVALID***: para pacotes que não pertencem e não criam uma conexão ou sessão. Pacotes ICMP não válidos são um bom exemplo;
- ***NEW***: para pacotes que iniciam uma nova conexão ou sessão;
- ***ESTABLISHED***: para pacotes pertencendo a uma conexão já estabelecida;
- ***RELATED***: para pacotes relacionados a alguma conexão ou sessão já estabelecida. Por exemplo, pacotes ICMP relacionados a alguma conexão existente.

### 6.3.1 Tabela de Estados

Como visto na seção anterior, o módulo de *Connection tracking* atribui um estado a cada pacote que chega ao *kernel*, de acordo com o andamento da comunicação a que pertence. Para isso, ele usa uma tabela *hash* (conhecida como *Separate Chaining*, Figura 6.2) para manter as informações de estados de todas as conexões e sessões passando pelo núcleo do sistema operacional [57]. Nesta tabela, cada posição guarda uma lista encadeada, onde cada nó representa uma conexão ou sessão (estrutura ip_conntrack_tuple_hash), contendo as informações necessárias para identificá-la unicamente (os endereços IP de origem e destino, os números de portas de origem e destino, o protocolo[1] usado), apontadores para os seus vizinhos na lista e para uma outra estrutura de dados (ip_conntrack) com informações de estados propriamente ditas referentes à comu-

1. O protocolo usado pode ser TCP, UDP ou ICMP.

nicação. Usando estas informações, o módulo de Contrack consegue atribuir o estado adequado para cada pacote dentro de sua respectiva conexão ou sessão.

A grande vantagem da tabela *hash* é a sua capacidade de armazenar elementos mesmo quando o seu número é maior que o número total de posições alocadas, sem comprometer o desempenho [82].

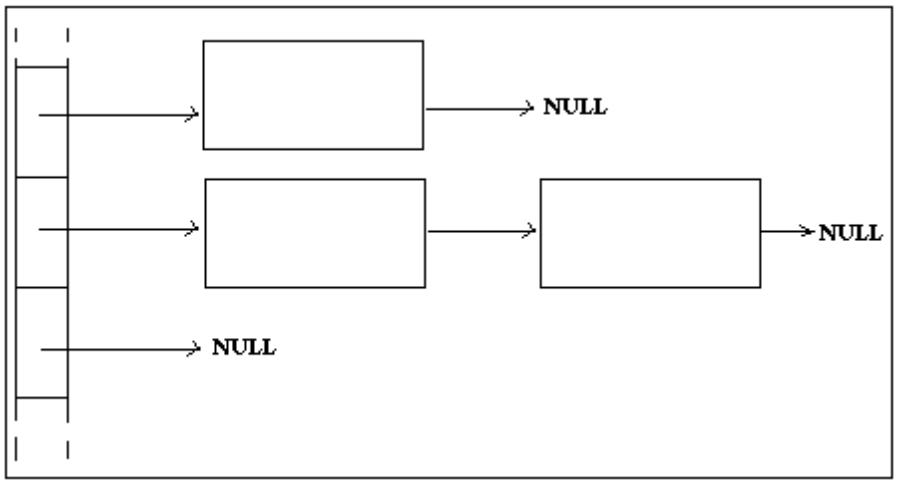


**Figura 6.2:** Tabela *Hash - Separate Chaining*

## 6.3.2 Tratamento dos Pacotes[1]

Quando um pacote chega para ser tratado pelo módulo Conntrack, verifica-se sempre se este é oriundo da interface de *loopback*, significando que já recebeu o tratamento adequado (já tem seu estado definido anteriormente) por este módulo e pode ser ignorado, ou seja, pode continuar sua travessia normalmente.

### 6.3.2.1 ICMP

Se o pacote recebido se trata de uma mensagem ICMP de erro ou controle (*destination unreachable*, *source quench*, *time exceeded*, *parameter problem* ou *redirect*), o módulo tenta associá-lo a alguma das sessões UDP ou conexões TCP existentes na tabela de estados. Caso este pacote

1. Vale ressaltar aqui que estas informações foram extraídas diretamente do código fonte implementando os módulos Conntrack e Iptables.

possa ser atrelado a alguma sessão ou conexão, ele recebe o estado de *RELATED*. Porém, se este não for o caso, recebe *INVALID*[1] como estado.

Para os demais tipos de pacotes ICMP, verifica-se somente se já existe alguma sessão ICMP para o mesmo na tabela de estados e, caso exista, se já houve tráfego em ambos os sentidos nesta comunicação. Caso não exista uma sessão para o pacote, o módulo cria uma nova entrada na tabela de estados (inicializa o *timeout* para a entrada) e atribui o estado de *NEW* para o pacote ICMP. Além disso, ele guarda a informação do sentido deste primeiro pacote, que permite tratar corretamente as retransmissões dentro da sessão. Portanto, quando um outro pacote ICMP chega e existe uma sessão para ele na tabela de estados, o módulo Conntrack observa se já há tráfego em ambos os sentidos nesta sessão. Quando já existe tráfego em ambos os sentidos, o pacote recebe *ESTABLISHED* como estado. Em caso contrário, ele é tratado como uma retransmissão do primeiro pacote e recebe o estado *NEW*. A Figura 6.3, página 67 apresenta o fluxograma para o tratamento de pacotes ICMP no módulo de *Connection Tracking*.

#### 6.3.2.2 UDP e TCP

Para o caso de um pacote UDP ou TCP, verifica-se se este pertence a alguma comunicação existente na tabela de estados. Em caso positivo, este pacote recebe o estado *ESTABLISHED* (se já existe tráfego em ambos os sentidos na sessão). Do contrário, o pacote inicia uma nova sessão ou conexão na tabela de estados (*timeout* é inicializado) e o estado do pacote é considerado *NEW*. Todo o processo, portanto, é idêntico para pacotes TCP e UDP. A única diferença é o *timeout* usado para a entrada de uma conexão TCP na tabela de estados. Para uma conexão TCP, existe um conjunto de valores de *timeout* que são atribuídos pelo módulo Conntrack dependendo do estado da conexão TCP; já em sessões UDP, o *timeout* é reinicializado usando sempre o mesmo valor quando algum pacote da sessão é recebido, como em sessões ICMP.

O leitor mais atento pode perceber que, para pacotes TCP, o comportamento é bastante parecido com o do FW-1. Qualquer pacote (não somente o pacote com a *flag* SYN ligada) pode criar uma entrada na tabela de estados. Entretanto, não há os cuidados de segurança adotados no caso do FW-1 (vide Seção 5.3, página 49, para maiores detalhes) e o administrador de segurança deve estar ciente disto, quando configurando o filtro de pacotes, para evitar que pacotes indesejáveis atravessem o *firewall*. Um outro aspecto interessante é que o módulo Conntrack antecipa o esta-

1. Estes pacotes devem ser filtrados pelo módulo Iptables.

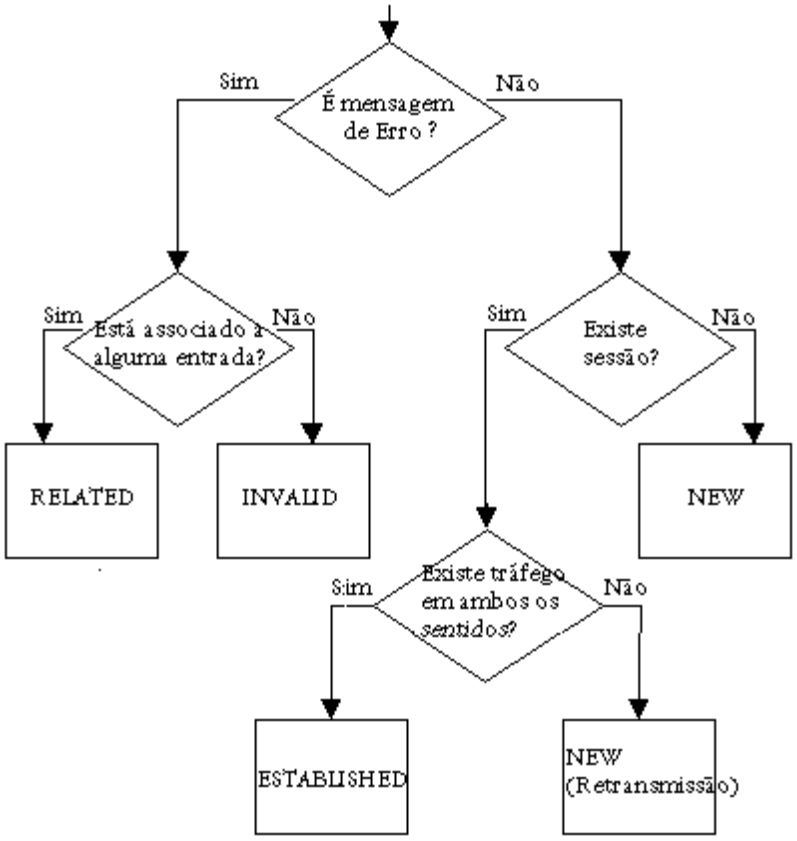


**Figura 6.3:** Pacotes ICMP

belecimento da conexão TCP, ou seja, um pacote pode receber o estado de *ESTABLISHED* desde que já exista uma entrada para a conexão na tabela de estados e tráfego em ambos os sentidos.

### 6.3.3 Lista de Conexões ou Sessões aguardadas

Além da tabela de estados das conexões e sessões, o módulo Conntrack também mantém uma lista de conexões ou sessões aguardadas (expect_list). Cada nó nesta lista guarda as informações dos endereços IP de origem e destino envolvidos, o número da porta destino e o protocolo a ser usado. Se um pacote tiver as mesmas informações em seus cabeçalhos de rede e transporte de uma das entradas nesta lista, o módulo assume que este pertence a esta conexão ou sessão esperada e atribui o estado *RELATED* ao pacote. Dai em diante, todos os pacotes desta conexão ou sessão relacionada recebem sempre o estado de *RELATED*. Além disso, cria-se uma entrada na

tabela de estados para esta nova conexão ou sessão e retira a entrada correspondente da lista de conexões ou sessões aguardadas. Isto é equivalente a criar uma regra dinâmica para permitir a comunicação.

Esta lista é, portanto, essencial para permitir o funcionamento correto de certos serviços e protocolos TCP/IP. Como discutido em outras oportunidades neste trabalho, o FTP é um bom exemplo destes protocolos, pois define o número de porta a ser usado na conexão de dados durante a conexão de controle. Para acomodar bem tais serviços e protocolos, o módulo Conntrack oferece módulos auxiliares para alguns protocolos (conhecidos como "*helpers*") e permite que outros, para protocolos ainda não suportados, sejam facilmente implementados e agrupados ao módulo de *Connection tracking*. Os *helpers* acompanham os dados trocados em uma comunicação envolvendo algum determinado protocolo e extraem as informações necessárias para permitir as conexões ou sessões relacionadas. Com estas informações, este módulo auxiliar pode criar uma entrada para a conexão ou sessão na lista.

Cada uma destas entradas permanece nesta lista enquanto durar a conexão ou sessão que a originou ou até que o primeiro pacote da sessão ou conexão relacionada chegue. Isto significa que não há *timeouts* associados, podendo dificultar um pouco o suporte a certos protocolos. Pode ser necessário, dependendo do protocolo, que um determinado *helper* gerencie uma lista de sessões ou conexões aguardadas particular para o correto tratamento de conexões ou sessões relacionadas. Esta foi a solução adotada para o caso do RPC que será visto na Seção 7.4.1, página 76.

## 6.4 Iptables (Módulo de Filtragem)

O Iptables é considerado um filtro de pacotes com estados porque ele permite o uso, no seu processo de filtragem, do estado atribuído pelo módulo de *Connection tracking* aos pacotes das diferentes sessões ou conexões. O módulo Iptables sozinho implementa somente uma filtragem estática dos pacotes como seus antecessores: Ipfwadm e Ipchains. Entretanto, quando o módulo Conntrack está presente no *kernel*, mantendo os estados das conexões ou sessões, o Iptables permite que o administrador de segurança crie regras baseadas nos estados dos pacotes dentro de suas respectivas conexões ou sessões. Desta forma, é possível a criação de regras permitindo a passagem, através do *firewall*, de todos os pacotes com estados *ESTABLISHED* e *RELATED*, sem comprometimento da segurança requerida.

# Capítulo 7

# Suporte a Serviços Inseguros Baseados em RPC

Neste capítulo serão mostrados detalhes da implementação do suporte a serviços inseguros baseados em RPC no filtro de pacotes com estados apresentado no capítulo anterior. Os módulos resultantes deste trabalho foram aprovados e serão incorporados ao próximo núcleo de produção do sistema operacional Linux (série 2.4).

## 7.1 Introdução

Na Seção 4.3, página 43, foram discutidos os principais problemas que dificultam a filtragem de serviços baseados em RPC. Viu-se que a melhor alternativa, a ser usada com as tecnologias mais tradicionais de *firewall*, é simplesmente bloquear incondicionalmente todo o tráfego de pacotes UDP, pois a maioria destes serviços RPC utiliza UDP como protocolo de transporte. Mas esta não é certamente a solução mais desejável sempre, como também já foi amplamente discutido.

Será visto, neste capítulo, como o iptables e o módulo de *Connection tracking* foram estendidos para implementar uma filtragem com estados de serviços baseados em RPC. O objetivo é inspecionar os dados trocados entre os clientes RPC e o portmapper/rpcbind, extraindo as informações relevantes para possibilitar uma filtragem com estados destes serviços inseguros. Além disso, serão discutidos alguns dos cuidados especiais tomados para evitar que este mecanismo de filtragem, baseado nestas informações de estados, seja facilmente subvertido com a utilização de pacotes ou fragmentos de pacotes devidamente construídos com esta finalidade. As estruturas de dados usadas e citadas no texto podem ser encontradas no apêndice B.

## 7.2 Sun RPC

Antes de tratar os detalhes de como a nova infra-estrutura no *kernel* do Linux 2.4.x e os módulos necessários foram usados para a implementação do suporte a serviços inseguros baseados em RPC, é válido observar algumas informações importantes sobre o funcionamento do protocolo Sun RPC [20], sobretudo os detalhes referentes à comunicação entre clientes RPC e o portmapper/rpcbind.

### 7.2.1 Funcionamento Básico

Quando um servidor RPC é iniciado em uma determinada máquina, caso ele não use um número de porta pré-definido (o NFS [22] usa sempre o número de porta 2049), ele aloca uma porta não reservada dinamicamente junto ao sistema operacional. Este servidor inicia também uma comunicação com o portmapper/rpcbind, na mesma máquina, para informar o número desta porta que estará usando para aguardar as requisições dos clientes.

Nesta comunicação com o servidor de nomes, o servidor RPC faz, na verdade, uma chamada remota ao procedimento PMAPPROC_SET do portmapper/rpcbind. Como parâmetros para este procedimento são passados: o número do programa que identifica unicamente o servidor RPC desejado, a versão deste servidor, o protocolo a ser usado na comunicação e o número da porta alocada. Por outro lado, o servidor de nomes retorna somente uma mensagem informando o sucesso ou insucesso do procedimento executado. Portanto, o próprio servidor de nomes portmapper/rpcbind é um servidor baseado em RPC [22]. Ele aguarda requisições usando a porta reservada 111, em ambos UDP e TCP. Mais adiante neste capítulo será visto em que situações cada protocolo de transporte é normalmente usado.

No momento em que um cliente deseja se comunicar com algum servidor RPC, ele tenta descobrir, junto ao servidor de nomes na máquina oferecendo o serviço, qual o número da porta que está sendo usada pelo servidor. Para tal, o cliente faz uma chamada remota ao procedimento PMAPPROC_GETPORT do portmapper/rpcbind, passando como parâmetros: o número de programa do servidor RPC, a versão do servidor e o protocolo a ser usado. O portmapper/rpcbind retorna, para o cliente, o número da porta sendo usada pelo servidor RPC (caso o servidor tenha se registrado anteriormente). Deste momento em diante, a comunicação entre o cliente e o servidor RPC ocorre normalmente; o cliente envia suas requisições diretamente para o servidor RPC.

Um outro aspecto interessante é que o cliente RPC geralmente mantém uma *cache* destas consultas e suas respectivas respostas aos servidores de nomes [3]. Desta forma, ele não precisa fazer sempre uma nova consulta ao portmapper/rpcbind para requisitar o número da porta usada por um determinado servidor.

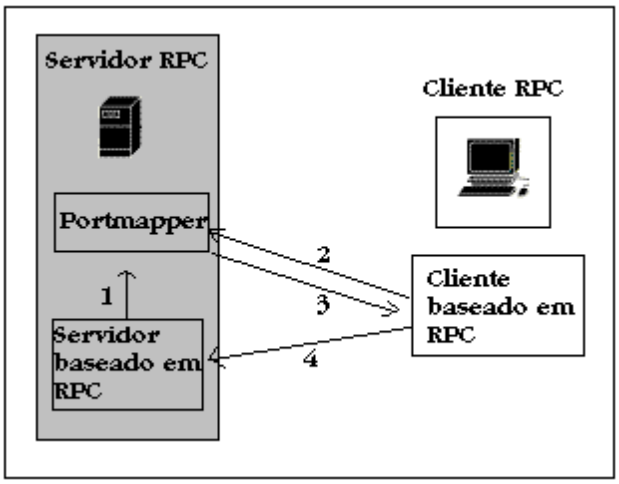


**Figura 7.1:** Comunicação RPC

Como mencionado acima, o servidor de nomes portmapper/rpcbind é também um servidor RPC. Na referência [20], o leitor mais interessado pode encontrar uma descrição completa dos outros procedimentos implementados pelo portmapper/rpcbind.

### 7.2.2 Mensagens RPC

O protocolo RPC envolve basicamente dois tipos de mensagens: mensagens *"call"* e mensagens *"reply"*. Uma mensagem *call* é originada quando um cliente RPC envia uma requisição para a execução de um procedimento em particular. Depois que o procedimento é executado, o servidor RPC envia de volta uma mensagem *reply*, contendo os resultados retornados na execução do procedimento [20][22]. A Figura 7.2, página 72, mostra o formato de uma mensagem *call*, quando encapsulada dentro de um datagrama UDP.

A mensagem *call* inicia com o campo XID (*Transaction* ID) que é um valor inteiro estabelecido pelo cliente e retornado sem nenhuma alteração pelo servidor. Quando o cliente recebe

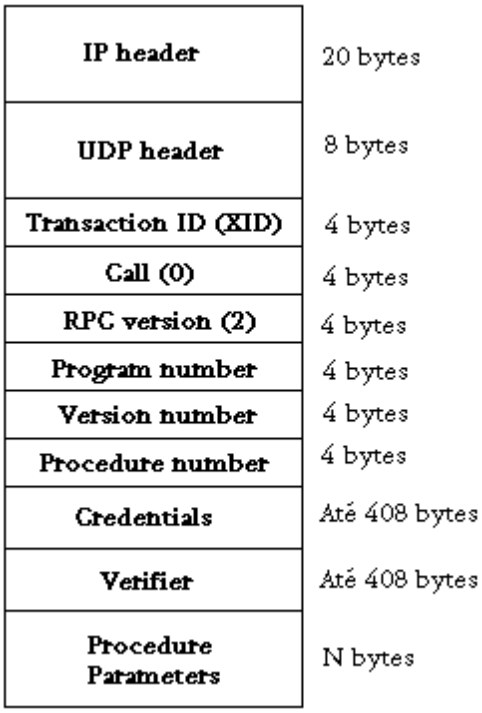


**Figura 7.2:** Mensagem *Call* com UDP

uma mensagem *reply* do servidor, ele compara o campo XID desta última mensagem (observar a existência do campo também na Figura 7.3, página 74) com o XID enviado. Desta forma, o cliente RPC pode verificar se a mensagem recebida se trata de uma resposta válida à sua requisição. É interessante ressaltar ainda que, caso o cliente retransmita a sua mensagem *call*, o valor usado anteriormente no campo XID, na mensagem original, não é alterado. Isto evita problemas com mensagens duplicadas ou atrasadas na rede.

O campo seguinte serve para distinguir entre mensagens *call* (valor inteiro 0) e mensagens *reply* (valor inteiro 1). A versão corrente do protocolo Sun RPC é 2 e, portanto, é este valor que é sempre transmitido no campo "RPC *version*". A seguir pode-se ver os campos *"program number"*, *"version number"* e *"procedure number"* que unicamente identificam um procedimento específico no servidor. Os campos *"credentials"* e *"verifier"* são usados para fins de autenticação e não serão tratados em detalhes neste trabalho. Na comunicação entre um cliente e o portmapper/rpcbind, que é mais interessante para este trabalho, não existe nenhum esquema de autenticação, no modo de operação normal, sendo usado. Embora estes campos tenham tamanhos

variáveis, o número de *bytes* usados é codificado como parte do campo: os primeiros quatro *bytes*, em cada um destes dois campos, representam o tamanho total dos mesmos. Além disso, mesmo que nenhum esquema de autenticação esteja sendo usado, este campo também deve transmitir um valor inteiro 0, que indica a ausência de um esquema de autenticação.

Finalmente, tem-se o campo *"procedure parameters"* com os parâmetros que devem ser passados ao procedimento. O formato deste campo depende da definição de cada procedimento remoto. Deve-se notar aqui que este campo também tem tamanho variado. Para o caso de um datagrama UDP não existem problemas, pois o tamanho deste campo é igual ao tamanho total do pacote UDP subtraído do comprimento total de todos os outros campos mostrados acima. Entretanto, quando TCP estiver sendo usado, um campo extra de quatro *bytes* entre o cabeçalho TCP e o XID é introduzido, codificando o tamanho total da mensagem RPC [22].

A Figura 7.3, página 74, mostra o formato de uma mensagem *reply* também encapsulada em um pacote UDP. O XID é copiado da mensagem *call* correspondente, como já foi discutido nesta seção. O campo seguinte tem valor 1 indicando que a mensagem se trata de um *reply*. A versão do protocolo RPC é 2, como no caso das mensagens *call*. O campo *"status"* recebe o valor 0 se a mensagem foi aceita (ela pode ser rejeitada, no caso de se estar usando outra versão do RPC ou de alguma falha na autenticação, e o valor atribuído, neste caso, será 1). O *"verifier"* tem o mesmo papel do caso anterior para mensagens *call*. O campo *"accept status"* recebe o valor 0 quando o procedimento é executado com sucesso. Neste caso, um valor diferente de zero identifica um determinado caso de erro, como por exemplo, um número de procedimento inválido sendo enviado pelo cliente. Como em mensagens *call*, se o protocolo de transporte TCP estiver sendo usado, um campo extra de quatro *bytes* é introduzido entre o cabeçalho TCP e o XID. Isto é útil para o cliente descobrir o tamanho correto do campo *"procedure results"*, com os resultados retornados pelo procedimento.

Os campos destas mensagens são codificados usando o padrão de representação XDR, possibilitando que clientes e servidores RPC usem arquiteturas diferentes. XDR [79] define como os vários tipos de dados são representados e transmitidos em uma mensagem RPC (ordem dos *bits*, ordem dos *bytes*, etc.). Portanto, o transmissor cria mensagens codificando todos os dados usando XDR e, do outro lado, o receptor decodifica os dados da mensagem usando este mesmo padrão de representação. Nas figuras 7.2 e 7.3, os campos usados transmitem principalmente

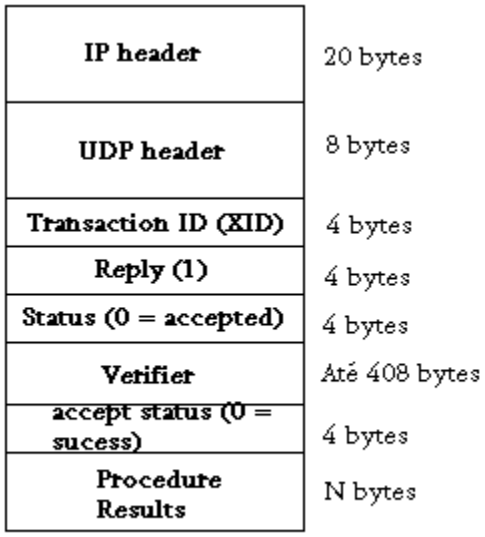


**Figura 7.3:** Mensagem *Reply* com UDP

valores inteiros. Portanto, pode-se concluir que um inteiro em XDR é codificado usando quatro *bytes* de dados [22].

### 7.2.3 Comunicação entre o Cliente RPC e o Portmapper/ Rpcbind

Na seção anterior, foram apresentados os formatos das mensagens RPC. Estas mensagens podem ser transferidas usando UDP ou TCP, como protocolo de transporte. TCP geralmente é usado quando o número de *bytes* dos parâmetros e/ou dos resultados, em um procedimento, é muito grande para ser enviado num único pacote UDP [20]. Na comunicação entre os clientes RPC e o portmapper/rpcbind, este limite é excedido quando os procedimentos PMAPPROC_DUMP e PMAPPROC_CALLIT são usados. O primeiro destes procedimentos é executado quando a ferramenta "rpcinfo" [3] é utilizada para verificar as informações de todos os serviços registrados junto ao portmapper/rpcbind em um dado momento. Já o procedimento PMAPPROC_CALLIT, como foi visto na Seção , página 44, é usado pelo recurso de *proxy forwarder*. Portanto, somente nestes dois casos um cliente realmente precisa usar TCP na comunicação com o servidor de nomes RPC.

Quando o cliente envia uma requisição para que o procedimento PMAPPROC_GETPORT seja executado no portmapper/rpcbind[1], somente 16 *bytes* da mensagem *call* são consumidos pelos parâmetros. Vale lembrar que cada valor inteiro, transferido usando XDR, ocupa 4 *bytes* da mensagem. Esta mensagem *call* envia 3 valores inteiros como parâmetros: número do programa, versão do programa e o protocolo (total de 12 *bytes*). Os 4 *bytes* restantes carregam sempre, neste caso, o valor zero [20]. Somando-se a este valor o número de *bytes* consumidos pelos outros campos na mensagem *call*, tem-se sempre 56 *bytes* na parte de dados dos pacotes.

A mensagem *reply* do portmapper/rpcbind carrega no campo dos resultados somente um valor inteiro (número de porta usado pelo servidor RPC desejado), ou seja, 4 *bytes* apenas são necessários no campo de *"procedure results"*. Portanto, estes pacotes têm sempre um tamanho de 28 *bytes* na parte de dados da aplicação. Vale ressaltar aqui que os campos de *"credentials"* e *"verifiers"*, neste caso, não codificam nenhum esquema de autenticação em particular, mas mesmo assim cada um ocupa sempre 2 *bytes* de dados (1 *byte* para o tamanho total do campo e 1 *byte* informando o esquema de autenticação NULL). Pode-se concluir, desta forma, que estas mensagens RPC são geralmente encapsuladas em datagramas UDP (acrescentando-se mais 20 *bytes* de cabeçalho IP e 8 *bytes* de cabeçalho UDP [9][22]). Além disso, faz pouco sentido criar e encerrar conexões TCP para a simples troca de alguns poucos *bytes* de dados. Os processos de estabelecimento e encerramento de conexões TCP utilizam três pacotes cada, gerando um *overhead* indesejável e desnecessário em casos como este.

## 7.3 Filtragem de Serviços Baseados em RPC

Na Seção 4.3, página 43, foi visto que a característica de não usar números de porta pré-definidos é o principal problema para a filtragem estática destes serviços. Um servidor RPC aloca o número de porta dinamicamente, junto ao sistema operacional, e registra-se ao servidor de nomes portmapper/rpcbind, informando a porta sendo usada em um determinado momento. Isto impossibilita a criação de regras estáticas para estes serviços usando número de portas fixos e previamente conhecidos. Outro problema existente com serviços baseados em RPC, também abordado na Seção 4.3, é que eles usam UDP, na grande maioria dos casos, como protocolo de

1. Assumindo que UDP seja o protocolo de transporte usado na comunicação.

transporte. Isso introduz todos aqueles problemas particulares do UDP relacionados à filtragem estática.

Uma boa solução para estes problemas é utilizar informações de estados extraídas da comunicação entre um cliente RPC e o portmapper/rpcbind para possibilitar a criação de regras dinâmicas para tais serviços. Um filtro de pacotes com estados pode simplesmente acompanhar uma sessão UDP (ou uma conexão TCP) envolvendo clientes RPC e o portmapper/rpcbind, dando atenção especial aos dados trocados em todas as chamadas ao procedimento PMAPPROC_GETPORT. Associando requisições às suas respostas e extraindo as informações relevantes, é possível manter as informações das portas sendo usadas e, consequentemente, permitir a comunicação entre clientes e servidores RPC.

## 7.4 Estendendo a Filtragem de Pacotes com Estados do Iptables

A solução, apresentada na seção anterior, de acompanhar todas as conexões e sessões entre clientes RPC e o portmapper/rpcbind e extrair as informações necessárias foi implementada neste trabalho. Foi usado, como ponto de partida, os módulos de *Connection tracking* e o Iptables presentes no *kernel* de desenvolvimento 2.3.x do Linux.

### 7.4.1 Estendendo o Módulo de Connection Tracking

Inicialmente foi implementado um módulo auxiliar (ip_conntrack_rpc) para inspecionar o andamento de todas as tentativas de comunicação com o portmapper/rpcbind. Este módulo segue diretamente as informações que foram vistas nas seções anteriores. Somente mensagens RPC, usando UDP ou TCP, requisitando a execução de procedimentos PMAPPROC_GETPORT e suas respectivas respostas são acompanhadas pelo módulo. O objetivo deste módulo é manter uma lista interna (lista de conexões ou sessões RPC aguardadas) com informações de estados extraídas da parte de dados dos pacotes e permitir que um outro módulo (ipt_rpc_known), estendendo o módulo de filtragem Iptables, possa verificar esta lista e, com base nestas informações, fazer a filtragem corretamente dos serviços baseados em RPC. Cada entrada na lista, representada por uma estrutura de dados (expect_rpc), mantém: os endereços IP do cliente e do servidor RPC,

o protocolo a ser usado na comunicação, o número da porta usada pelo servidor RPC e um timeout associado à entrada.

Como visto nas seções anteriores, o funcionamento do protocolo RPC inicia quando o cliente se comunica com o portmapper/rpcbind para determinar o número da porta usada por um determinado servidor RPC naquela máquina. Nesta comunicação, o cliente passa como parâmetros o número de programa do serviço, o protocolo e a versão desejada. O módulo Conntrack responsável pelo tratamento do RPC acompanha todas estas requisições. Ele, na verdade, cria uma lista de requisições, para manter o histórico de todas as consultas ao portmapper/rpcbind. Cada entrada na lista de requisições (estrutura de dados request_p) mantém: o endereço IP do cliente, o XID, o número de porta do cliente, o protocolo a ser usado na comunicação com o servidor RPC e um *timeout* associado à entrada. Este *timeout* evita que consultas forjadas criem um número indefinido de novas entradas em ataques de DoS, consumindo recursos na máquina filtradora.

Caso este servidor esteja registrado, o portmapper/rpcbind retornará o número da porta usada pelo servidor naquela máquina. Quando esta mensagem *reply* chega ao módulo Conntrack, este pode verificar se existe uma entrada na lista de requisições e, caso exista, pode criar uma entrada na lista de conexões ou sessões RPC aguardadas. Nesta verificação, além das informações mantidas na lista de requisições, o módulo observa o XID, o endereço IP e a porta do cliente nestas mensagens. Isso é necessário, pois a informação do protocolo a ser usado não está disponível na mensagem *reply*. Para que uma entrada seja criada na lista de conexões ou sessões RPC aguardadas esta informação também é necessária.

Se não existir uma entrada na lista de requisições correspondendo à mensagem *reply*, o módulo Conntrack simplesmente descarta a mensagem para um acompanhamento mais correto das comunicações. O leitor pode imaginar uma situação em que o portmapper/rpcbind, por algum motivo, atrasa a execução do procedimento PMAPPROC_GETPORT e o cliente retransmite a requisição. Pode-se supor então um cenário onde o *timeout* para entrada na lista de requisições expira e a mensagem de requisição retransmitida pelo cliente é perdida em trânsito e, portanto, não seria vista pelo módulo Conntrack. Caso esta mensagem *reply* chegue ao cliente, ele tentará fazer a comunicação com o servidor RPC normalmente. Entretanto, esta comunicação não será permitida devido a inexistência de uma entrada para ela na lista de conexões ou sessões RPC aguardadas.

Um outro aspecto importante é o tratamento das possíveis comunicações subseqüentes dos clientes com os servidores RPC. O *timeout* associado a cada entrada na lista de conexões ou sessões aguardadas tem como objetivo básico permitir futuras tentativas de comunicação de clientes usando sua respectiva *cache* de respostas anteriores do portmapper/rpcbind. Se a entrada fosse excluída imediatamente após o fim da comunicação, futuras tentativas não seriam permitidas. Obviamente, clientes bem implementados devem prover um esquema de recuperação para falhas deste tipo [3]. Esta falha seria similar ao caso de um servidor registrado ser reinicializado e o cliente, usando as informações da *cache*, tentar usar o serviço.

### 7.4.2 Estendendo o Iptables (Módulo de Filtragem)

Foi visto na Seção 6.4, página 68, que o módulo de filtragem, nas séries 2.3 e 2.4 do *kernel*, baseia sua decisão também em informações de estados dos pacotes dentro de suas respectivas sessões ou conexões. Um módulo estendendo o Iptables foi criado para tratar todos os pacotes RPC com estado *NEW* (iniciando uma sessão ou conexão) chegando ao módulo de filtragem. A Figura 7.4, página 79 ilustra detalhes do algoritmo implementado neste módulo. Desta forma, o Iptables permite todas as tentativas de comunicação com o portmapper/rpcbind com consultas aos números de portas sendo usadas pelos servidores registrados. Portanto, não há a necessidade de uma regra permitindo explicitamente a comunicação com o servidor de nomes do RPC. Notar ainda que as perigosas mensagens *call* com requisições para a execução de PMAPPROC_CALLIT (ver Seção 4.3, página 43) não são permitidas pelo filtro. Este novo módulo do Iptables também é o responsável em verificar as entradas da lista de conexões e sessões aguardadas e permitir que clientes se comuniquem normalmente com servidores RPC.

### 7.4.3 Testes de Sanidade e Cuidados Adicionais com Fragmentos e Pacotes Truncados

Para um correto acompanhamento das mensagens RPC, alguns cuidados devem ser tomados com respeito à sanidade dos pacotes. Além disso, os módulos devem tomar cuidados extras para evitar que pessoas mal intencionadas possam subverter o mecanismo de filtragem de pacotes. O módulo Conntrack, antes de inspecionar a parte de dados de todos os pacotes RPC, verifica se há uma má formação e calcula o *checksum* do cabeçalho de transporte no pacote. Se UDP estiver sendo usado, esta última operação somente é feita se o campo de *checksum* tiver um valor dife-

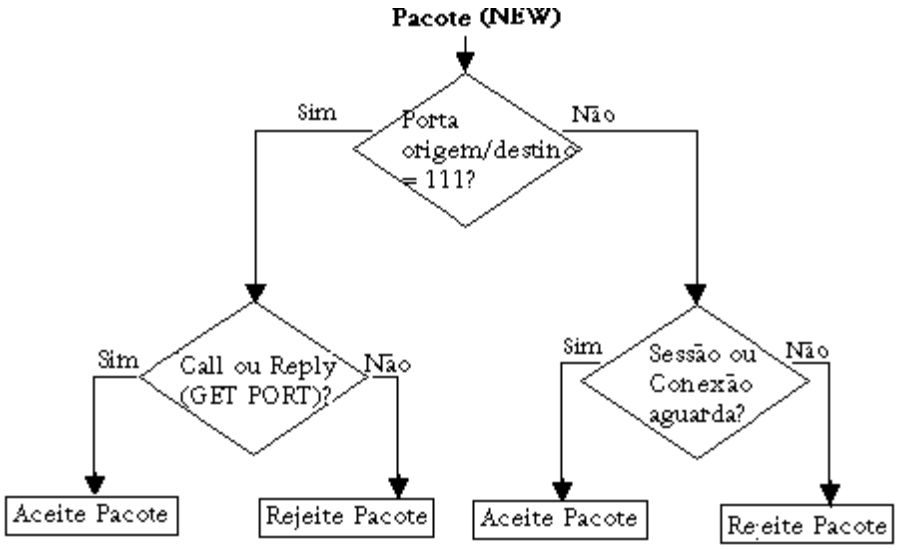


**Figura 7.4:** Filtragem dos Pacotes RPC

rente de zero, indicando que ele foi calculado, já que a checagem em UDP é opcional e percebida através do valor diferente de zero [22].

Pacotes e fragmentos de pacotes cuidadosamente construídos foram previstos durante a implementação, tendo-se em mente, principalmente, a vulnerabilidade recente encontrada no FW-1 e Cisco-PIX no suporte ao FTP, mostrada em detalhes na Seção 5.7, página 53. Com este problema, um atacante é capaz de subverter o filtro de pacotes com estados. Para isto, bastava forçar que os pacotes do protocolo FTP fossem devidamente truncados e, explorando um problema existente na inspeção da parte de dados, abrir brechas no *firewall* [52].

Como já mencionado na Seção 7.2.3, página 74, as mensagens RPC trocadas entre clientes e o portmapper/rpcbind são extremamente pequenas. Desta forma, cabem facilmente em qualquer MTU existente nas interfaces de rede atuais. Desta forma, pacotes RPC fragmentados são tratados como um comportamento anormal e simplesmente ignorados pelo módulo Conntrack. Além disso, pacotes RPC, onde a parte de dados é maior que o esperado, são imediatamente descartados.

Se um cliente usar TCP para a comunicação com o portmapper/rpcbind, os valores dos atributos da conexão devem ser sempre suficientemente grandes para não truncar a parte de dados nos pacotes. Em geral, o valor mínimo atribuído ao parâmetro MSS, por exemplo, é idêntico à

MTU usada pelo meio físico para evitar a indesejável fragmentação de pacotes [22][78]. Pode-se concluir, com isso, que não há boas razões para usar valores muito pequenos para este parâmetro e, portanto, qualquer tentativa disto é vista como um ataque contra o filtro de pacotes com estados.

# Capítulo 8

# Conclusão

Ao longo deste trabalho, com mais ênfase no capítulo 2, foram apresentados diversos problemas de segurança existentes na arquitetura de protocolos TCP/IP e as falhas mais comuns encontradas nas implementações dos diversos serviços e protocolos TCP/IP em sistemas operacionais. Viu-se que estes protocolos nasceram com poucos recursos de segurança nativos; foram especificados tendo como base uma realidade completamente diferente da que encontramos na Internet atualmente. Existiam conectados somente alguns poucos computadores confiáveis entre si e que perfeitamente honravam todos os detalhes previstos nos padrões destes protocolos.

Entretanto, a rede cresceu bastante e deixou de ser um ambiente seguro. Novos protocolos e serviços surgiram e continuam surgindo muito rapidamente, como resultado do avanço tecnológico e devido à pressão de usuários cada vez mais exigentes, herdando os antigos problemas de segurança da infra-estrutura original e introduzindo outros novos. Além disso, muitos *softwares* são pobremente desenvolvidos do ponto de vista de segurança e/ou não seguem por completo as especificações dos protocolos. Estes *softwares* têm aberto perigosas brechas de segurança que podem ser facilmente exploradas.

No primeiro semestre do ano 2000, grandes portais da Internet foram vítimas de ataques de DDoS (*Distributed Denial of Service*), revelando, principalmente àqueles mais crédulos, que a Internet não é um ambiente totalmente seguro e apresenta sérios problemas na arquitetura de protocolos TCP/IP. Mais recentemente, pragas eletrônicas (vírus, cavalos de Tróia, dentre outros), disseminadas por correio eletrônico, demonstram que muitos *softwares* são desenvolvidos sem uma preocupação adequada com segurança. Isto sem falar dos diversos problemas encontrados diariamente nas diferentes implementações de serviços e protocolos TCP/IP.

Estas vulnerabilidades e as novas funcionalidades, exigida pela Internet, motivam um esforço, cada vez maior, em busca de soluções e tecnologias capazes de amortizar e, até mesmo, eliminar por completo a maioria destes problemas. Bons exemplos são: *firewalls*, IDS, VPN, adoção de algoritmos e técnicas de criptografia, dentre outras. Há ainda trabalhos que objetivam um modelo de arquitetura TCP/IP segura, que não foram tratados neste trabalho.

Este trabalho abordou somente as diferentes tecnologias de *firewall*, mostradas no capítulo 3, destacando a utilização da filtragem de pacotes com estados para um tratamento mais conveniente dos chamados "serviços inseguros". Estes serviços, de acordo com a definição dada no capítulo 4, são aqueles que não podem ser facilmente tratados pelas tecnologias mais tradicionais de *firewall*. Com tais tecnologias de *firewall* - filtros de pacotes estáticos ou agentes *proxy* - estes serviços devem ser totalmente bloqueados para garantir uma maior segurança a uma rede local.

Neste trabalho, mais especificamente no capítulo 4, viu-se que alguns serviços - os que usam UDP como protocolo de transporte e os baseados em RPC - são representantes importantes desta classe, aqui chamada de serviços inseguros. Atualmente, um grande número de novos protocolos e serviços, que fazem uso do UDP[1] principalmente, têm surgido. Faz-se necessário, portanto, que outras alternativas mais flexíveis sejam adotadas para permitir que tais serviços sejam oferecidos numa rede local, sem um comprometimento da segurança do sistema.

No capítulo 5, foram apresentadas as características básicas encontradas nas diversas implementações correntes de filtros de pacotes com estados. Por manter informações de estados das conexões ou sessões através do filtro, elevando assim o processo de filtragem timidamente ao nível de aplicação, é possível oferecer uma provisão segura de tais serviços inseguros. Entretanto, foi visto também que a filtragem feita na camada de aplicação, nas implementações atuais desta nova tecnologia de filtragem de pacotes, é bem simples quando comparada à realizada por agentes *proxy* dedicados. Nestes agentes *proxy* há uma completa remontagem e inspeção do fluxo de dados de uma aplicação.

Uma implementação de filtragem com estados estará presente no próximo *kernel* de produção (série 2.4) do Linux. Trata-se do Iptables, que foi apresentado em detalhes no capítulo 6. Este filtro de pacotes com estados foi usado como ponto de partida para o trabalho de implementação do suporte a serviços inseguros baseados em RPC. Serviços baseados em RPC são difí-

1. *Real Audio/Video, Netmeeting, Cu-SeeMe* são alguns exemplos.

ceis de filtrar, pois normalmente usam portas alocadas dinamicamente, junto ao sistema operacional, no momento em que são inicializados. Clientes RPC recorrem a um servidor de nomes RPC (portmapper/rpcbind, no caso específico do protocolo SunRPC) para descobrir o número da porta sendo usado por um determinado servidor RPC em um determinado momento.

Com este trabalho, o Iptables foi então estendido para manter informações de estados relevantes, extraídas destas comunicações entre os clientes RPC e o portmapper/rpcbind. Possibilitou-se, desta forma, observar as respostas do servidor de nomes RPC às consultas dos diferentes clientes e determinar quais portas foram alocadas pelos diferentes servidores RPC registrados. Com estas informações é possível permitir dinamicamente todas as comunicações entre clientes e servidores RPC. Detalhes desta implementação foram mostrados no capítulo 7. Além disso, foi mostrada a preocupação da realização de testes de sanidade, no filtro de pacotes, para evitar que pacotes e/ou fragmentos cuidadosamente construídos pudessem sobrepujar o mecanismo de filtragem com estados. Portanto, pacotes mal formados jamais podem ser permitidos através do *firewall* e descartados pelo processo de filtragem.

Como principais contribuições deste trabalho, podem ser destacados o estudo pioneiro das principais características desta novíssima geração de filtros de pacotes e a criação de um filtro de pacotes com estados capaz de prover suporte a serviços inseguros baseados em RPC. O resultado deste trabalho de implementação será incorporado à próxima versão oficial do *kernel* (série 2.4 de produção) do sistema operacional Linux. Nos apêndices A e B pode-se encontrar as estruturas de dados usadas neste trabalho de implementação. O leitor mais interessado pode encontrar ainda, no apêndice C, uma listagem das mais populares implementações de filtros de pacotes com estados.

# Bibliografia


[1] Steven M. Bellovin. *Security Problems in the TCP/IP Protocol Suite*. ACM Computer Communications Review, Vol. 19, No. 2, March 1989.

[2] Steven M. Bellovin. *Using the Domain Name System for System Break-ins*. Proceedings of the Fifth Usenix UNIX Security Symposium, Salt Lake City, UT. AT&T Bell Laboratories, 1995.

[3] John Bloomer. *Power Programming with RPC*. O'Reilly & Associates, Inc. 1991.

[4] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security: Repelling the Willy Hacker*, Addison-Wesley, Reading, MA, 1994.

[5] Terry Bernstein, Anish B. Bhimani, Eugene Schultz and Carol A. Siegel. *Segurança na Internet*, Editora Campus, Rio de Janeiro, 1997.

[6] *Computer Emergency Response Team*. CERT FTP Archive. URL: ftp://ftp.cert.org/pub/cert_advisories/.

[7] *Bugtraq Mailing List Archive*. URL: http://www.bugtraq.net.

[8] *Firewall-Wizard Mailing List Archive*. URL: http://www.nfr.net.

[9] Douglas E. Comer. *Internetworking with TCP/IP: Principles, Protocols and Architecture, Volume I*. Prentice-Hall, Englewood Cliffs, New Jersey, Third Edition, 1995.

[10] D. Brent Chapman and Elizabeth D. Zwicky. *Building Internet Firewalls*. O' Reilly & Associates, Inc. 1995.

[11] Simson Garfinkel and Gene Spafford. *Practical Unix Security*. O' Reilly & Associates, Inc. 1991.

[12] Craig A. Huegenchuegen. *The Latest in Denial of Service Attacks: Smurf*. July 1998.

[13] John Ioannidis and Matt Blaze. *Architecture and Implementation of Network-layer Security Under Unix*. Proceedings of USENIX Security Symposium, Santa Clara, CA, October 1993.

[14] Christopher William Klaus. *Stealth Scanning - bypassing Firewalls and SATAN Detectors*. Internet Security Systems, Inc, 1995.

[15] J. Mogul and S. Deering. *Path MTU Discovery*. RFC 1191, DECWRL, Stanford University, November 1990.

[16] Marcelo Barbosa Lima and Paulo Lício de Geus. *Comparação entre Filtros de Pacotes com Estados e Tecnologias Tradicionais de Firewalls*. SSI99 (Simpósio sobre Segurança em Informática), São José dos Campos, Setembro, 1999.

[17] Keesje Duarte Pouw and Paulo Lício de Geus. *Uma Análise das Vulnerabilidades dos Firewalls*. WAIS´96 (Workshop sobre Administração e Integração de Sistemas), Fortaleza, Maio, 1996.

[18] Rusty Russell. *Netfilter Home Page*. URL: http://netfilter.kernelnotes.org. July 2000.

[19] Darren Reed. *Darren Reed home page*. ULR: http://coombs.anu.edu.au/~avalon/ip-filter.html. Version 3.4.7. July 2000.

[20] Sun Microsystems. *RPC: Remote Procedure Call protocol specification: Version 2*. RFC 1057, June 1988.

[21] Dareen Reed, Tom Fitzgerald and Paul Traina. *RFC 1858: Security Considerations - IP Fragment Filtering*. October, 1995.

[22] William Richard Stevens. *TCP/IP Illustrated*. Addison Wesley, 1994.

[23] Wietse Venema. *TCP WRAPPER: Network Monitoring, Access Control and Booby Traps*. Proceedings of the Third Usenix UNIX Security Symposium, Baltimore, Maryland, September 1992.

[24] Paul Vixie. *DNS and BIND security issues*. In Proceedings of the Fifth Usenix UNIX Security Symposium, Salt Lake City, UT, 1995.

[25] Marcus J. Ranum. *Firewalls FAQ—Frequently Asked Questions*. 1995.

[26] Simson Garfinkel and Gene Spafford. *Practical Unix Security & Internet Security, 2nd Edition*. O' Reilly & Associates, Inc., 1996.

[27] Marcus Gonçalves. *Firewalls Complete*. McGraw-Hill, 1997.

[28] Christopher Klaus. *Backdoors*. Internet Security Systems. April 1997.

[29] Frederick M. Avolio. *Application Gateways and Stateful Inspection: Brief Note Comparing and Contrasting*. Trusted Information System, Inc. January, 1998.

[30] Ryan Russell. *Proxies Vs. Stateful Packet Filters*. Version 0.9. Initial Draft RLR. June 1997.

[31] CheckPoint Software Technologies Ltd. *CheckPoint Firewall-1 white paper*, Version 3.0. June 1995.

[32] CheckPoint Software Technologies Ltd. *Stateful Inspection Firewall Technology*, Tech Note, 1998.

[33] Keesje Duarte Pouw. *Segurança na Arquitetura TCP/IP: de Firewalls a Canais Seguros*. Campinas, janeiro 1999. Tese de Mestrado.

[34] Marco Aurélio Spohn. *Desenvolvimento e análise de desempenho de um "Packet Session Filter"*. Porto Alegre, 1997. Tese de Mestrado.

[35] ISS X-Force. *Back Orifice 2000 Backdoor Program white paper*. 2000.

[36] Paul A. Henry. *Buffer Overrun Attacks*. Cyberguard Corp, 1999.

[37] Aleph One. *Smashing the Stack for Fun and Profit*. Phrack 49. Volume 7, article 14.

[38] Stefan Savage, Neal Cardwell, David Wetherall and Tom Anderson. *TCP congestion Control with a Misbehaving Receiver*. Department of Computer Science and Engineering. University of Washington, Seattle, 1999.

[39] Craig H. Rowland. *Covert Channels in the TCP/IP Protocol*. 1996.

[40] Crispin Cowan, Perry Wagle, Calton Pu, Seteve Beatte and Jonathan Walple. *Buffer Overflows: Attacks and Defenses for the Vulnerability of the decade*. SANS, 2000.

[41] Bill Cheswick. *The Design of a Secure Internet Gateway*. Proceedings of the USENIX Summer' 90 Conference.

[42] D. Brent Chapman. *Network (In) Security through IP packet filtering*. In proceedings of the Third USENIX Unix Security Symposium, pages 63-67, September 1992.

[43] Marcus J. Ranum. *Thinking About Firewalls*. Trusted Information Systems, Inc. 1992.

[44] Steve M. Bellovin. *There Be Dragons*. AT&T Bell Laboratories, Murray Hill, NJ., Septemper 1992.

[45] Aleph One. *DNS ID Hacking*. Phrack Magazine, Volume 8, Issue 52. January 1998.

[46] Mika Silander. *Denial of Service Attacks*. Department of Computer Science, Helsinki University of Technology, March 1999.

[47] Scott C. Sanchez. *What Firewalls Will Look in the Year 2003*. CISSP, April 2000.

[48] Robert Graham. *FAQ - Firewalls: what am I seeing?*. March 2000.

[49] Tim Clark. *Softwares Extras Add Fuel to Firewalls*. CNET News.com, March 1999.

[50] Christoph L. Schuba and Eugene H. Spafford. *Addresing Weaknesses in the Domain Name System Protocol*. August 1993.

[51] David Sacerdote. *Some Problems with File Transfer Protocol. A Failure of Commom Implepentations, and Sugestions for Repair*. April 1996.

[52] Mikael Olson. *Breaking Through FTP ALGs -- Is it Possible?*. Exploit-Dev List, Message-ID:<389FEB7B.AA290CC7@enternet.se. February 2000.

[53] Van Haussen. *Placing Backdoors Through Firewalls*. THC, April 1998.

[54] Ofir Arkin. *ICMP Usage in Scanning*. Publicom Communications Solutions, July 2000.

[55] Paul Russel. *Linux IPCHAINS-HOWTO*. Version 1.0.5, October 1998.

[56] Dan Moniz. *Alternative Thinking in H.323 Capable Firewall Design*. Phrack Magazine, Volume 9, Issue 55, Article 17. September 1999.

[57] Rusty Russell. *Linux Netfilter Hacking HOWTO*. Version 0.0.1, August 1999.

[58] Brendan Conoboy and Erik Fichtner. *IP Filter Based Firewalls HOWTO*. October 1999.

[59] Robert T. Morris. *A Weakness in the 4.2BSD Unix TCP/IP Software*. AT&T Bell LAboratories, February 1985.

[60] Daemon, Route and Infinity. *IP spoofing Demystified (Trust-Relationship Exploitation)*. Phrack Magazine, Volume 7, Issue 48, article 14, June 1996.

[61] Al Berg. *Net App Opens Doors for Hackers*. LAN Times, August 1996.

[62] Trevor Marshall. *The Firewall Masquerade*. Byte Magazine, July 1999.

[63] Trevor Marshall. *Increasing Your Masquerading Gateway Security*. Byte Magazine, September 1999.

[64] Ofir Arkin. *Network Scanning Techniques - Understanding How It is Done*. Publicom Communications Solutions, November 1999.

[65] Fyodor. *The Art of Port Scanning*. Phrack Magazine, Volume 7, Issue 51, Article 11. September 1997.

[66] Fyodor. *Remote OS Detection via TCP/IP Stack fingerprinting*. October 1998.

[67] Steve M. Bellovin. *Packets found on an Internet*. Computer Communications Review, Vol. 23, Number 3, pages 26-31. July 1993.

[68] Ron Gula. *How to Handle and Identify Network Probes*. April 1999.

[69] Douglas E. Comer and John Lin. *Probing TCP implementations*. Proceedings of USENIX Summer 1994 Conference.

[70] SANS Institute. *How Eliminate the Ten Most Critical Internet security Threats*. June 2000.

[71] K. Egevang and P. Francis. *The IP Network Address Translator (NAT)*. RFC 1631, May 1994.

[72] P. Ferguson and D. Senie. *Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing*. RFC 2267, January 1998.

[73] Guido Van Rooj. *Real Stateful TCP Packet Filtering in Ipfilter*. Proceedings of SANE2000.

[74] Network Associates, Inc. *TCP Spoofing Attack*. Security Advisory #7, February 1997.

[75] Network Associates, Inc. *The Active Firewall. The End of the Passive Firewall Era*. Executive White Paper.

[76] Network Associates, Inc. *Adaptive Proxy Firewalls. The Next Generation Firewall Architecture*. Gauntlet Firewall Executive White Paper.

[77] Lance Spitzner. *Understanding the FW-1 State Table*. Work in Progress. URL: http://www.enteract.com/~lspitz/fwtable.html.

[78] R. Atkinson. *Default IP MTU for use over ATM ALL 5*. RFC 1626, May 1994.

[79] R. Srinivasan. *XDR: External Data Representation Standard*. RFC 1832, August 1995.

[80] Rusty Russell. *Linux Iptables HOWTO*. Version 1.0, July 2000.

[81] NeonSurge/Rhino9 Publications. *SYN Floods and SYN cookies. The Cause and Cure*. 1996.

[82] David A. Rusling. *The Linux Kernel*. Review, Version 0.8-3, January 1999.

[83] Trusted Information Systems, Inc. *TIS Firewall Toolkit - Configuration and Administration*. February 1994.

[84] NEC USA, Inc. *Introduction to Socks*. 1999.

[85] Laurent Joncheray. *A simple Active Attack Against TCP*. Merit Network, Inc. April 1995.

[86] Cisco Systems Products Security Incident Team. *Cisco Secure PIX Firewall TCP Reset Vulnerability*. Revision 1.0, July 2000.

[87] S. Bellovin. Firewall-Friendly FTP. RFC 1579. February 1994.

# Apêndice A

# Estrutura de Dados mais Importantes Usadas pelo Módulo de *Connection Tracking*

```
/* Expect List */
struct list_head {
        struct list_head *next, *prev;
};
struct list_head expect_list;
/* Connection state tracking for netfilter. This is separated from, but required by, the NAT layer;
it can also be used by an iptables extension. */
/* A `tuple' is a structure containing the information to uniquely identify a connection. ie. if two
packets have the same tuple, they are in the same connection; if not, they are not. We divide the
structure along "manipulatable" and "non-manipulatable" lines, for the benefit of the NAT code.
*/
/* The protocol-specific manipulable parts of the tuple. */
union ip_conntrack_manip_proto
{
        /* Add other protocols here. */
        u_int16_t all;
        struct {
                u_int16_t port;
```

```
        } tcp;
        struct {
                u_int16_t port;
        } udp;
        struct {
                u_int16_t id;
        } icmp;
};
/* The manipulable part of the tuple. */
struct ip_conntrack_manip
{
        u_int32_t ip;
        union ip_conntrack_manip_proto u;
        u_int16_t pad; /* Must be set to 0 for memcmp. */
};
/* This contains the information to distinguish a connection. */
struct ip_conntrack_tuple
{
        struct ip_conntrack_manip src;
        /* These are the parts of the tuple which are fixed. */
        struct {
                u_int32_t ip;
                union {
                        /* Add other protocols here. */
                        u_int16_t all;
                        struct {
                                u_int16_t port;
                        } tcp;
                        struct {
                                u_int16_t port;
                        } udp;
```

```
                        struct {
                                u_int8_t type, code;
                        } icmp;
                } u;
                /* The protocol. */
                u_int16_t protonum;
        } dst;
};
enum ip_conntrack_info
{
        /* Not part of an established, and can't be used to make a new
        one; eg. bogus ICMP error.  Conntrack pointer invalid. */
        IP_CT_INVALID = -1,
        /* Started a new connection to track (only
         IP_CT_DIR_ORIGINAL); may be a retransmission. */
        IP_CT_NEW,
        /* Part of an established connection (either direction). */
        IP_CT_ESTABLISHED,
        /* Like NEW, but related to an existing connection, or ICMP error
           (in either direction). */
        IP_CT_RELATED,
        /* Specifies a related connection: in established conn
           (either direction). */
        IP_CT_ESTABLISHED_SPECIFIES_RELATED,
        IP_CT_NUMBER
};
enum ip_conntrack_dir
{
        IP_CT_DIR_ORIGINAL,
        IP_CT_DIR_REPLY,
        IP_CT_DIR_MAX
```

```
};
/* Connections have two entries in the hash table: one for each way */
struct ip_conntrack_tuple_hash
{
        struct list_head list;
        struct ip_conntrack_tuple tuple;
        struct ip_conntrack *ctrack;
};
/* Carefully managed bitset for atomic updates. */
enum ip_conntrack_status {
        /* Normal connection: bit 0 set. */
        IPS_NORMAL = 0,
        /* It's an expected connection: bit 1 set */
        IPS_EXPECTED = 1,
        /* We've seen packets both ways: bit 2 set. */
        IPS_SEEN_REPLY = 2
};
struct ip_conntrack_expect
{
        /* Internal linked list */
        struct list_head list;
        /* We expect this tuple, but DON'T CARE ABOUT THE SOURCE per-protocol part. */
        struct ip_conntrack_tuple tuple;
};
struct ip_conntrack
{
        /* These are my tuples; original and reply */
        struct ip_conntrack_tuple_hash tuplehash[2];
        /* Have we seen traffic both ways yet? (bitset) */
        unsigned int status;
        /* If it's dead, ignore it in the hash. */
```

```
        int dead;
        /* Usage count. */
        atomic_t use;
        /* Timer function; deletes us when we go off. */
        struct timer_list timeout;
        /* If we're expecting another related connection, this will be
         in expected linked list */
        struct ip_conntrack_expect expected;
        /* Helper, if any. */
        struct ip_conntrack_helper *helper;
};
struct ip_conntrack_protocol
{
        /* Next pointer. */
        struct list_head list;
        /* Protocol number. */
        u_int8_t proto;
        /* Protocol name */
        const char *name;
        /* Try to fill in the third arg; return true if possible. */
        int (*pkt_to_tuple)(const void *datah, size_t datalen,
                            struct ip_conntrack_tuple *tuple);
        /* Invert the per-proto part of the tuple: ie. turn xmit into reply.
         * Some packets can't be inverted: return 0 in that case.
         */
        int (*invert_tuple)(struct ip_conntrack_tuple *inverse,
                            const struct ip_conntrack_tuple *orig);
        /* Print out the per-protocol part of the tuple. */
        unsigned int (*print_tuple)(char *buffer,
                                    const struct ip_conntrack_tuple *);
        /* Print out the private part of the conntrack. */
```

```
        unsigned int (*print_conntrack)(char *buffer,
                                        const struct ip_conntrack *);
        /* Returns verdict for packet, and may modify conntracktype */
        unsigned int (*established)(struct ip_conntrack *conntrack,
                                    struct iphdr *iph, size_t len,
                                    enum ip_conntrack_dir direction,
                                    enum ip_conntrack_info *conntrackinfo);
        /* Called when a new connection for this protocol found; returns TRUE if it's OK. */
        int (*new)(struct ip_conntrack *conntrack,
                   struct iphdr *iph, size_t len);
        /* Module (if any) which this is connected to. */
        struct module *me;
        /* Time (in HZ) until expiry. */
        unsigned long expiry;
        /* unregister() attaches here waiting to complete deletion. */
        wait_queue_head_t wait;
};
struct ip_conntrack_helper
{
        /* Internal use. */
        struct list_head list;
        /* Here's the protocol and src (of reply) we care about. */
        u_int16_t protocol;
        union ip_conntrack_manip_proto protocol_reply;
        /* Function to call when data passes; return verdict. */
        unsigned int (*help)(const struct iphdr *, size_t len,
                             struct ip_conntrack_tuple_hash *h,
                             enum ip_conntrack_info *conntrackinfo);
};
```

# Apêndice B

# Principais Estruturas de Dados usadas no Tratamento de Serviços Inseguros baseados em RPC (Módulo Conntrack Estendido)

```
/* This identifies each particular request and stores protocol */
struct request_p {
   struct list_head list;
   u_int32_t xid;
   u_int32_t ip;
   u_int16_t port;
   /* Protocol */
   u_int16_t proto;
   struct timer_list timeout;
};
/* request_p list */
struct list_head {
          struct list_head *next, *prev;
     };
struct list_head request_p_list;
/* It permits RPC communications */
```

```
struct expect_rpc {
   /* list */
   struct list_head list;
   /* binding */
   u_int32_t ip;
   u_int16_t port;
   u_int32_t ip_rmt;
   u_int16_t proto;
   /* Expect RPC conection for long time, but this time expires */
   struct timer_list timeout;
};
/* RPC Expect List */
struct list_head expect_rpc_list;
```

# Apêndice C

# Algumas Implementações de Filtros de Pacotes com Estados.

| Implementação | Endereço na Internet |
|---|---|
| Iptables (domínio público) | http://netfilter.kernelnotes.org |
| T-REX (domínio público) | http://www.opensourcefirewall.com/trex.html |
| Firewall-1 (CheckPoint) | http://www.checkpoint.com |
| CISCO-PIX (Cisco) | http://www.cisco.com |
| Gauntlet (NAI) | http://www.nai.com |
| Guardian (Netguard) | http://www.netguard.com |
| WatchGuard (WatchGuard) | http://www.sealabs.com |
| Solstice Firewall-1 (Sun Microsystems) | http://www.sun.com |
| Firewall/Plus (Network-1) | http://www.network-1.com |
| CyberGuard Firewall (Cyberguard) | http://www.cybg.com |
| Ipfilter (domínio público) | http://coombs.anu.edu.au/~avalon/ip-filter.html |

**Tabela C.1:** Implementações de Filtros de Pacotes com Estados

# Apêndice D

# Comparação entre Filtros de Pacotes com Estados e Tecnologias Tradicionais de Firewall[1]

## D.1 Resumo

Este trabalho compara as vantagens e desvantagens de filtros de pacotes com estados (*Stateful Packet Filter*) em relação às tecnologias tradicionais de *firewall*: filtros de pacotes e *proxies*. Será mostrado aqui, que um filtro de pacotes com estados é uma solução que em alguns casos (pequenas *Intranets* onde os riscos são pequenos e redes locais onde o desempenho é muito importante), poderia substituir de maneira segura as tecnologias tradicionais de *firewall* (sem a necessidade de filtros de pacotes e agentes *proxy*). Entretanto, em ambientes onde os requisitos de segurança são maiores, *proxies* específicos para determinados protocolos e serviços críticos devem ser adicionados à filtragem.

## D.2 Abstract

This work compares the advantages and disadvantages of stateful packet filters in relation to the-traditional technologies of firewall: packet filters and proxies. It will be shown here, that a stateful packet filter is a solution that in some cases (little Intranets where the risks are small and local networks where the throughput is very important), could safely to substitute traditional techno-

1. Artigo publicado nos anais do simpósio segurança em informática (SSI), São José dos Campos: CTA/ITA/IEC, Setembro,1999.

logies of firewall (without no need for packet filters and proxy agents). However, in rooms where security requirements are higher, specific proxies for certain critical protocols and services should be added to the filtering.

## D.3 Introdução

Nos últimos anos, a indústria de segurança na Internet tem crescido rapidamente e vários produtos estão disponíveis na classe de *firewalls*, gerando uma certa confusão para quem procura uma solução que melhor se enquadre às suas necessidades de segurança.

Neste artigo, é feita uma comparação entre as diferentes tecnologias de *firewall* que fornecem a base usada na implementação desses produtos, ajudando a esclarecer as suas características importantes, vantagens e desvantagens, ajudando portanto a descobrir qual produto é o mais conveniente, dependendo dos requisitos de segurança desejados. Alguns produtos são híbridos, ou seja, apresentam características de tecnologias diferentes. Além das tradicionais, filtros de pacotes e agentes *proxy*, uma nova alternativa (se é que pode ser considerada nova, uma vez que para muitos, não passa de uma nova geração de filtros de pacotes – Avolio, 1998) está sendo usada: filtros de pacotes com estados (*stateful packet filter*, SPF), que já obtém boa parcela do mercado de *firewalls* e tem o Firewall-1 da CheckPoint como seu principal representante. Será mostrado que filtros de pacotes com estados podem ser suficientes em alguns ambientes de computação, mas em outros, onde os requisitos de segurança são maiores para alguns serviços, será necessário o uso de proxies específicos para estes serviços, possivelmente complementando a atuação do SPF.

A seção 2 apresenta as tecnologias tradicionais (filtros de pacotes e *proxies*), com suas principais vantagens e deficiências. A seção 3 apresenta filtros de pacotes com estados, como eles tentam aproveitar algumas das principais vantagens de filtros de pacotes e de proxies e as suas desvantagens em relação a um *proxy* dedicado a um determinado protocolo ou serviço. A seção 4 mostra que um filtro com estados pode ser uma solução suficientemente segura em certos ambientes, mas *proxies* podem ser necessários (em adição à filtragem) em outros onde os requisitos de segurança sejam maiores. A seção 5 vem com uma breve discussão sobre limitações de *firewalls*. A seção 6 traz as considerações finais e as conclusões deste trabalho.

# D.4 Tecnologias Tradicionais

Nesta seção veremos as duas tecnologias tradicionais de *firewalls* (Cheswick e Bellovin, 1994; Chapman, e Zwick, 1995; Spohn, 1997; Gonçalves, 1997; Pouw, 1999), que são os filtros de pacotes e *proxies*.

## D.4.1 Filtros de Pacotes

Filtragem de pacotes é o processo de permitir ou evitar tráfego de pacotes entre redes com base nas informações existentes nos cabeçalhos de cada pacote e em um conjunto de regras de filtragem. Em produtos que implementam filtragem de pacotes as informações utilizadas são aquelas existentes nos cabeçalhos dos níveis de rede e de transporte de cada pacote.

Por diversos anos, várias redes locais fizeram uso de roteadores filtradores ou escrutinadores (*screening routers*) para garantir a segurança de suas informações e controlar acesso. Estes roteadores implementavam o processo de filtragem juntamente com o processo de roteamento de pacotes. Esta arquitetura de *firewall* com um único roteador filtrador pode ser bastante atraente, uma vez que estes *hardware* e *software* de roteamento já são mesmo necessários para que a rede local seja conectada no nível de rede.

Por concentrar-se nas informações das camadas de rede e de transporte, um filtro de pacotes pode ser implementado também no núcleo do sistema operacional, o que lhe garante sempre um bom desempenho. Além disso, por não depender do nível de aplicação, tem ainda como vantagens a transparência (não precisa de configurações especiais nos clientes) e a escalabilidade (novos serviços são facilmente suportados).

Estes primeiros filtros de pacotes, por usarem regras de filtragem estáticas, são classificados como filtros de pacotes estáticos. Tais filtros de pacotes não conseguem tratar convenientemente serviços que usem *back-channels*, como o FTP (*File Transfer Protocol*). Estes protocolos exigem um pedido de conexão entrando na rede protegida, em resposta a um pedido de conexão saindo, feito por uma máquina desta rede. Por usar regras estáticas, este filtro de pacotes pode ser obrigado a não implementar perfeitamente a política de segurança da rede local para possibilitar estas últimas conexões, permitindo também a entrada de certos pacotes que deveriam ser filtrados no *firewall*.

Surgiram depois filtros mais inteligentes, capazes de criar regras dinamicamente. Estes filtros são conhecidos como filtros de pacotes dinâmicos. Os filtros de pacotes com estados, que serão apresentados na próxima seção, são considerados por muitos somente um avanço de filtros de pacotes dinâmicos (Avolio, 1998) e, por isso, não se tratam de uma nova tecnologia de *firewall*, como falam outros (CheckPoint, 1998).

Entretanto, os filtros de pacotes apresentam algumas desvantagens. Aplicações baseadas no protocolo UDP (*User Datagram Protocol*) são difíceis de filtrar, porque no UDP não existe distinção entre uma requisição e uma resposta. Nestes filtros, as soluções utilizadas para tais serviços consiste simplesmente em eliminar as sessões UDP inteiramente, ou ainda em abrir uma porção do espaço de portas UDP para comunicação bi-direcional e, portanto, expor a rede interna. Alguns filtros dinâmicos têm a capacidade de "lembrar" os pacotes UDP que saem da rede interna e, com isso, só aceitam pacotes entrando nesta rede que correspondam a estes primeiros. Esta correspondência é feita mantendo-se algumas informações presentes nos cabeçalhos de rede e de transporte dos pacotes UDP saindo da rede protegida: porta de origem, endereço de origem, porta de destino e endereço de destino. Então, somente pacotes UDP entrando na rede interna, vindos destas portas e endereços do destino original para as suas respectivas portas e endereços da origem, são aceitos. Aplicações baseadas em RPC (*Remote Procedure Call*) são ainda mais complicadas de serem filtradas, por não usarem números de portas pré-definidos. A alocação de portas é feita de maneira dinâmica e geralmente muda com o tempo. O portmapper/rpcbind é o único servidor relacionado ao RPC que usa um número de porta pré-definido. Ele é o responsável em manter o mapeamento entre os serviços RPC oferecidos e os números de portas que foram alocados para eles. O que filtros tradicionais geralmente fazem é simplesmente bloquear todos os pacotes UDP, porque a grande maioria destes serviços baseados em RPC utilizam como protocolo de transporte o UDP.

Por excluírem do processo de filtragem os dados da aplicação, tais filtros podem permitir ataques às vulnerabilidades destes protocolos e dos serviços do nível de aplicação. Além disso, filtros de pacotes permitem que máquinas, dentro da rede a ser protegida, recebam diretamente os pacotes enviados pelas máquinas externas, tornando possíveis ataques que se utilizam de vulnerabilidades e falhas no sistema operacional destas máquinas, como por exemplo *teardrop*, *ping of death* e outros.

Como os filtros de pacotes tradicionais não fazem nenhuma filtragem na camada de aplicação, alguma outra tecnologia de *firewall* também deve ser utilizada para eliminar esta deficiência.

### D.4.2 Proxies

Um *proxy* ou *gateway* de aplicação é uma função de *firewall* no qual um serviço de rede é fornecido por um processo que mantém um completo estado da comunicação, examinando todo o fluxo de dados. Um *proxy*, na verdade, evita que hosts na rede interna sejam acessíveis de fora da rede protegida, usando duas conexões: uma entre o cliente (máquina interna) e o servidor *proxy* e outra entre o servidor *proxy* e o servidor real (máquina remota).

Um servidor *proxy* pode melhorar a segurança por examinar o fluxo da conexão na camada de aplicação, mantendo informações de contexto para o processo de decisão. Com isso, ele permite fazer uma filtragem na camada de aplicação evitando ataques relacionados às vulnerabilidades dos protocolos de aplicação, possíveis em filtros de pacotes. Além disso, pode ser também utilizado para implementar *cache* de dados para um determinado serviço e autenticação de usuários. Entretanto, há uma queda de desempenho em relação a filtros de pacotes, porque este processo requer duas conexões e geralmente é implementado como um processo do sistema operacional. Como deve existir um *proxy* para cada serviço diferente, há uma perda na escalabilidade, transformando o suporte a novas aplicações um grande problema. Para amenizar este problema, alguns desses produtos de *firewall* fornecem ferramentas para a criação de *proxies* genéricos, que fazem na realidade, pouco mais que um filtro de pacotes estático e, portanto, não são muito diferentes de filtros de pacotes dinâmicos.

Um outro problema de muitos produtos que implementam *proxies* é a perda de transparência. Os clientes atrás do *firewall* devem ser configurados para usarem o *proxy* do serviço e não os servidores remotos diretamente. Este problema não existe mais nos *proxies* conhecidos como transparentes. Nestes *proxies* transparentes, os clientes não precisam de nenhuma configuração especial para fazerem a conexão direta ao *proxy*.

Em NAT (*Network Address Translation*) não há consumo de recursos da máquina responsável pela tradução de endereços, por não manter estados de conexões, ao contrário de *proxies* transparentes. Neste caso, há somente uma reescrita nos campos de endereços dos pacotes que entram e saem da rede interna. Para a implementação de um *proxy* transparente é preciso que todo tráfego de pacotes passe ou seja redirecionado para a máquina *proxy*, antes de pesquisar o seu

destino. É conveniente, portanto, que o servidor *proxy* fique em uma máquina *dual-homed*. Esta máquina deve aceitar todos os pacotes, saindo da rede interna, vindos dos clientes, independente do endereço do destino. O *proxy* transparente funciona como um proxy tradicional, fazendo a comunicação com o servidor remoto. Para o cliente, tudo é transparente porque o servidor *proxy* usa o endereço do servidor remoto como origem dos pacotes que ele envia para o cliente. Portanto, *proxies* transparentes não podem ser confundidos com máquinas que fazem NAT.

Além destes problemas, se um protocolo é muito complexo, a implementação do *proxy* pode ficar com código fonte muito grande e conter *bugs*. Por confiar no sistema operacional (um *proxy* está no espaço de processos do sistema operacional), uma máquina *proxy* pode ter sua segurança comprometida também, por ataques ao sistema operacional. São conhecidos hoje vários problemas em sistemas operacionais que podem entre outras coisas, negar serviços, como por exemplo problemas na remontagem de pacotes com seus fragmentos.

Para alguns serviços pode ser difícil ou até mesmo impossível implementar proxies. *Proxies* para serviços baseados em UDP, ICMP (*Internet Control Message Protocol*) e RPC, por exemplo, são muito difíceis de implementar. Além disso, com o crescimento de uso do WWW (*World Wide Web*), vários novos serviços têm aparecido rapidamente, o que torna complexa a tarefa de criar *proxies* para estes novos serviços com a mesma rapidez.

## D.5 Filtros de Pacotes com Estados

Visando aumentar a segurança, um *firewall* deve guardar informações e controlar todo fluxo de comunicação passando através dele. Para fazer decisões de controle para serviços baseados em TCP/IP (aceitar ou rejeitar pedidos de conexões, autenticar, cifrar, etc), um *firewall* deve obter, armazenar, recuperar e manipular informação derivada de todas as camadas de comunicação incluindo a de aplicação.

Não é suficiente examinar pacotes isoladamente. Informações de estados, derivada de comuncações passadas e de outras aplicações, é fator essencial para a decisão de controle em novas tentativas de comunicação. Estas são as idéias básicas de um filtro de pacotes com estados (CheckPoint, 1998).

Um filtro com estados potencialmente pode fazer qualquer coisa que um *proxy* pode fazer (Russel, 1997). Mas diferentemente de um *proxy*, ele envolve no seu processo de filtragem todas as camadas de comunicação, desde os cabeçalhos de rede e transporte até a parte de dados do

nível de aplicação. Desta forma o processo de filtragem torna-se agora mais homogêneo: uma só entidade vai fazer a análise completa do tráfego de dados. Veremos depois que, embora seja possível fazer o mesmo que quaisquer proxies, implementações correntes de filtros com estados não o fazem por completo.

Por manter estados do andamento da comunicação, um filtro de pacotes com estados permite o suporte a protocolos sem conexão, tais como UDP, e a serviços baseados em RPC, que usam portas alocadas dinamicamente. Com estes filtros é possível extrair informações de contexto (o campo de dados) dos pacotes UDP, mantendo desta forma uma conexão virtual de forma a fazer uma filtragem mais inteligente. Para o caso de serviços baseados em RPC, uma *cache* pode ser mantida nas tabelas de estados do filtro de pacotes com estados, usando temporariamente um processo de realimentação, para um portmapper/rpcbind "virtual". Este, por sua vez, pode mapear os números de portas aos serviços RPC oferecidos na rede interna, podendo estabelecer regras dinâmicas para estes serviços. O processo de realimentação contínuo é necessário porque um serviço RPC em algum host da rede interna pode, eventualmente, alocar um outro número de porta depois de uma reinicialização do *host*, por exemplo.

No filtro de pacotes com estados, portanto, temos a segurança de um *proxy* e o desempenho e escalabilidade de filtros de pacotes. Deve-se lembrar, contudo, que a escalabilidade não ocorrerá inicialmente com o mesmo grau de segurança que os serviços previamente cobertos. Os níveis de rede e transporte podem receber regras de filtragem apropriadas em questão de minutos, por um administrador de segurança competente. Contudo, o nível de aplicação, para um novo serviço, será geralmente estranho ao administrador e o desenvolvimento de filtragem apropriada poderá levar tempo. Assim, é possível disponibilizar o novo serviço imediatamente, mas em um grau menor de segurança (caso o novo serviço venha realmente exigir filtragem específica no nível de aplicação) até que regras mais sofisticadas sejam disponibilizadas. Por fazer seu trabalho com todos os *bits* do pacote, tais filtros são implementados nas camadas mais baixas do sistema operacional, geralmente entre as camadas de comunicação de enlace de dados e rede. Isso pode garantir mais segurança por evitar que pacotes "mal formados" cheguem ao sistema operacional da máquina filtradora antes de serem interceptados pelo filtro. Entretanto, ainda permitem, como nos filtros tradicionais, que as máquinas internas sejam acessadas diretamente.

Uma vantagem, que os implementadores de filtros de pacotes com estados alegam que osseus produtos têm em relação a *proxies*, é a de ser mais fácil e rápido o suporte a novos serviços

e protocolos. Geralmente estes produtos incluem linguagens de *scripts* para permitir ao usuário do produto criar regras de filtragem para estas novas tecnologias. A verdade é que estas linguagens apenas tornam possível construir mais facilmente *proxies* (podemos pensar que filtros de pacotes com estados implementam *proxies* em filtros de pacotes), melhores que os *proxies* genéricos. *Proxies* genéricos, por não realizarem nenhum tipo de filtragem, devem ser apenas temporários, até que *proxies* dedicados sejam disponibilizados.

Como já foi mencionado antes, um filtro de pacotes com estados pode potencialmente fazer qualquer coisa que um *proxy* faz, ou seja, ele pode implementar um *proxy* em suas regras de filtragem. Entretanto, a realidade, que encontramos nos diversos produtos que implementam filtros de pacotes com estados, é que os mesmos não o fazem por completo. Usaremos o Firewall-1 (CheckPoint, 1997) como exemplo, embora o mesmo possa ser observado em outros produtos de filtros de pacotes com estados. No Firewall-1, existem dois módulos básicos: o Firewall-1 *Inspection Module* (também chamado INSPECT *engine*) e o *security server*. No primeiro, reside o filtro de pacotes com estados propriamente dito. Ele é implementado no *kernel* do sistema operacional entre as camadas de enlace de dados e rede. Com o INSPECT *engine* é possível filtrar pacotes usando todas as camadas de comunicação e estados de comunicações passadas. Muito do que pode ser feito em *proxies* pode ser feito também neste módulo, porque ele implementa, com suas regras de filtragem, proxies mais simples para os serviços. Entretanto, se o usuário quiser fazer um trabalho maior na camada de aplicação, ele precisa usar o segundo módulo do produto. Os *security servers* rodam no espaço de processos e existem para FTP, HTTP (*HyperText Transmission Protocol*), SMTP (*Simple Mail Transfer Protocol*) e também para fins de autenticação. Na verdade, eles apresentam as mesmas características que *proxies* para estes serviços. Assim, quando o usuário desejar autenticação extra de usuários, fazer inspeção no fluxo de dados à procura de vírus, bloquear *applets* Java ou fazer análises mais sofisticadas no fluxo de dados, o Firewall-1 deixa estes *security servers* realizarem o trabalho. A vantagem desta solução é que o filtro no *kernel* fica mais simples, eliminando o *overhead* de um *proxy* implementado junto do filtro e garantindo, portanto, um bom desempenho com segurança, além de regras de filtragens mais simples, o que diminui as chances de erros.

Na prática, portanto, a maioria dos produtos que implementam filtros de pacotes com estados recorrem a *proxies* (os *security servers*, no caso particular do Firewall-1) sempre que é necessário fazer um trabalho maior com os dados da aplicação. Por este motivo, muitos acreditam que *prox-*

*ies* são mais seguros que filtros de pacotes com estados. Além disso, suspeita-se que os produtos comerciais que implementam filtros de pacotes com estados, sequer fazem uma análise mais detalhada no fluxo de dados para alguns serviços e protocolos, funcionando como um simples filtro de pacotes tradicional (Avolio, 1998). Na próxima seção voltaremos a esta discussão.

Finalizando o tema sobre "filtros de pacotes com estados", falta mencionar que a implementação de NAT no mesmo produto, junto com filtros de pacotes com estados, é bastante facilitada; por isso, a maioria dos produtos desta classe oferece este recurso. NAT, para a maioria dos protocolos e serviços, deve trabalhar no nível de rede, mas para alguns outros, ele deve fazer algum trabalho também na camada de aplicação. Este é o caso do FTP (se houver tradução de endereços somente no nível de rede, o FTP não funcionará, uma vez que ele usa endereços de rede em seu processamento). Deste modo, NAT assim como filtros de pacotes com estados, também realiza seu trabalho nas camadas de rede e aplicação e pode ser implementado usando-se os recursos da aplicação de filtragem, sem grande esforço adicional. Além disso, usando NAT e as informações de estados, filtros de pacotes com estados podem oferecer suporte à distribuição de processamento entre diversos servidores de algum serviço de maneira transparente, dividindo o carregamento entre os diversos servidores e provendo escalabilidade e maximizando o desempenho. Isso é muito útil, principalmente em redes locais que permitem acesso a suas informações através de *sites* Web, onde é grande o número de acessos. Esta característica também pode ser encontrada na maioria dos produtos que implementam filtros de pacotes com estados.

## D.6 Filtros de Pacotes com Estados vs. *Proxies*

Como visto na seção anterior, um filtro de pacotes com estados geralmente pode implementar um *proxy* mais simples no *kernel* do sistema operacional e, ao mesmo tempo, funcionar como um filtro de pacotes tradicional. Além disso, alguns participantes em listas de discussão da área acreditam que, para alguns protocolos e serviços, os produtos que implementam filtros de pacotes com estados não fazem nenhuma análise no fluxo de dados, funcionando como um filtro de pacotes tradicional (Avolio, 1998). Mas será que por isso é sempre verdade falar que *proxies* são soluções melhores que filtros de pacotes com estados? A resposta para esta pergunta é não. Dependendo de que requisitos devem ser satisfeitos em uma rede local, filtros de pacotes com estados, sozinhos, podem ser suficientes para prover segurança.

Teoricamente, é possível construir um *proxy* completo em um filtro de pacotes com estados (Russel, 1997). Desta forma, poder-se-ia afirmar que um *proxy* é um caso especial de um filtro de pacotes com estados. Entretanto na prática como vimos anteriormente, os vendedores optam por produtos mais leves e, quando um trabalho maior precisa ser feito na camada de aplicação, eles recorrem a *proxies* para o serviço. Isso pode ser explicado pelo fato de que é uma tarefa mais complexa escrever regras de filtragem que implementem um *proxy* do que escrever um *proxy* equivalente. Mas, se o ambiente computacional não exige, em nenhum serviço ou protocolo, um trabalho maior na camada de aplicação, um filtro de pacotes com estados é uma solução suficiente. Se este não é o caso, pode ser necessário usar um *proxy* para o serviço junto com o filtro de pacotes com estados ou com um filtro de pacotes tradicional.

O filtro de pacotes com estados também tem a vantagem de evitar ataques às vulnerabilidades do sistema operacional, por interceptar os pacotes antes de atingirem o sistema operacional, evitando assim os diversos ataques de baixo nível na arquitetura TCP/IP, como por exemplo, *stealth scanning*, SYN *flood*, etc. Além disso, serviços baseados em UDP e RPC podem ser tratados convenientemente neste tipo de filtros de pacotes, como foi discutido na seção anterior. O NFS (*Network File System*), por exemplo, é um serviço baseado em RPC e UDP que poderia ser mais facilmente provido por um filtro de pacotes com estados. Construir um *proxy* para NFS é ainda mais complicado que em outros serviços baseados em UDP e RPC, porque neste serviço o desempenho é fundamental e o volume de dados é grande.

É, portanto, inegável que filtros de pacotes com estados podem ser uma solução suficiente em diversos ambientes computacionais onde os riscos são baixos e em ambientes com exigências de desempenho. Se algum trabalho mais sofisticado deve ser feito como, por exemplo, bloquear instâncias específicas de Java ou buscar vírus no fluxo de dados, deve-se usar *proxies* dedicados a este trabalho, em adição à filtragem por estados. No caso de alguns produtos comerciais como o Firewall-1, vimos que é possível que estes *proxies* já sejam incluídos no próprio produto. Esta necessidade pode ocorrer, na verdade, com serviços novos ou com serviços e protocolos mais complexos, onde proxies mais poderosos sejam a única maneira de garantir segurança ao serviço. Como exemplos, cita-se o caso de clientes WWW e o de propagação de vírus e cavalos de Tróia (BackOriffice, Netbus e semelhantes).

No caso do suporte a serviços e protocolos novos, as linguagens de *scripts* nestes produtos comerciais que implementam filtros de pacotes com estados, para a escrita de regras de filtragem,

podem permitir uma solução mais segura que *proxies* genéricos ou neutros. Além disso, para aplicações desenvolvidas dentro de um ambiente corporativo, por exemplo, para as quais não existem *proxies*, pode ser bastante interessante usar um filtro de pacotes com estados para estabelecer regras de filtragem para este serviço.

É válido ressaltar ainda que, por seu baixo custo computacional e por sua segurança adicional, filtros de pacotes com estados são sempre preferidos a filtros de pacotes estáticos e dinâmicos. Em resumo: o filtro de pacotes com estados sempre é uma boa solução, mesmo que seja apenas em substituição aos filtros tradicionais em ambientes onde os riscos são altos. *Proxies*, sempre em adição à filtragem, serão acrescentados sempre que algum tratamento mais sofisticado precisar ser realizado em algum serviço. No âmbito de *firewalls*, já existem vários produtos na categoria de filtros de pacotes com estados. Firewall-1 da CheckPoint, CISCO-PIX da Cisco, o Firewall/Plus da Network-1, o Guardian da NetGuard e o Gauntlet da TIS, são alguns dos que implementam filtros de pacotes com estados.

## D.7 Limitações das Tecnologias de *Firewall*

Todas estas tecnologias de *firewall* infelizmente ainda apresentam algumas limitações (Pouw e Geus, 1997). Dentre as limitações, o surgimento destas linguagens de *scripts*, como Java e Javacript, que fazem a Web tão atrativa, são um grande problema para os *firewalls*. Bloquear tais programas não é a melhor solução, mas é a única que há no momento para os projetistas de *firewalls*. Seria interessante algo que permitisse uma filtragem seletiva destes programas no *firewall*. Infelizmente, isso é impossível em alguns casos. Talvez uma solução distribuída (no *firewall* e nos clientes atrás do *firewall*, por exemplo) resolva este problema, mas até o momento as soluções existentes não são 100% seguras.

Um outro grande problema para *firewalls* é uma classe de ataques que usa o "tunelamento" de pacotes, isto é, dados não permitidos dentro de pacotes permitidos, segundo a análise do *firewall*. Já são conhecidos ataques que usam esta deficiência dos protocolos TCP/IP. Como um exemplo, vamos imaginar um programa "cavalo de Tróia" que consegue privilégios de superusuário dentro de uma rede local. O atacante pode simplesmente aguardar que este programa atrás do *firewall* tente estabelecer uma conexão para um servidor do atacante e este servidor, na verdade, envia para ele os comandos a serem executados dentro da rede local. Para que os pacotes passem desapercebidos no *firewall*, o programa atacante atrás do *firewall* e o servidor do

atacante podem simplesmente encapsular seus dados dentro de pacotes HTTP que seriam permitidos pelo *firewall*. Não existe nenhuma tecnologia de *firewall* que possa evitar este tipo de ataque. Algo que poderia ser feito seria exigir que as tentativas de conexões saindo da rede fossem autenticadas, mas se aquele programa sendo executado atrás do *firewall* for um pouco mais sofisticado, esta solução não será suficiente.

Portanto, é importante que mesmo com um *firewall*, defina-se políticas de segurança para os usuários da rede local (evitar a execução de programas recebidos por e-mail e de *applets* Java em páginas WWW não confiáveis, por exemplo) e usar outras ferramentas de segurança (IDS, anti-vírus atualizados, entre outras) minimizando estas limitações das tecnologias de *firewalls*.

## D.8 Conclusão

Filtros de pacotes com estados podem ser considerados um avanço ou uma nova geração de filtros de pacotes; em alguns ambientes podem ser suficientes para prover segurança. Estes ambientes são ambientes onde os riscos são baixos ou deseja-se eliminar os gargalos de comunicação via *proxy*, provendo transparência aos usuários e garantindo segurança maior que filtragem simples.

Se o ambiente computacional quer minimizar riscos de segurança, algum trabalho extra deve ser feito para alguns serviços isolados, usando-se então *proxies* dedicados e bastante focados. Estas tecnologias de *firewall* não conseguem, por si só, garantir um ambiente totalmente seguro. Políticas de segurança para os usuários devem ser bem estabelecidas para garantir que certos problemas explorando limitações dos *firewalls* sejam evitados.

## D.9 Agradecimentos

Os autores agradecem ao CNPq, à FAEP-UNICAMP e ao Pronex-SAI pelo financiamento ao trabalho e às muitas pessoas das listas de discussão de segurança, que direta e indiretamente nos ajudaram com referências e informações sobre o assunto.

# D.10 Referência Bibliográficas

AVOLIO, Frederick M. Application Gateways and Stateful Inspection: Brief Note Comparing and Contrasting. Trusted Information System, Inc. (TIS). Janeiro, 1998.

CHAPMAN, D. Brent; ZWICKY, D. Elizabeth. Building Internet Firewalls. O' Reilly & Associates, Inc. 1995.

CHECKPOINT, Software Technologies Ltd. CheckPoint Firewall-1 White Paper, Version 3.0, junho, 1997.

CHECKPOINT, Software Technologies Ltd. Stateful Inspection Firewall Technology, Tech Note, 1998.

CHESWICK, William ; BELLOVIN, Steven M. Firewalls and Internet Security – Repelling the wily hacker. Addison Wesley, 1994.

GONÇALVES, Marcus. Firewalls Complete. McGraw-Hill, 1997.

POUW, Keesje Duarte. Segurança na Arquitetura TCP/IP: de Firewalls a Canais Seguros. Campinas: IC-Unicamp, Janeiro, 1999. Tese de Mestrado.

POUW, Keesje Duarte; GEUS, Paulo Licio de. Uma Análise das Vulnerabilidades dos Firewalls. WAIS'96 (Workshop sobre Administração e Integração de Sistemas), Fortaleza. Maio, 1997.

RUSSEL, Ryan. Proxies vs. Stateful Packet Filters. http://futon.sfsu.edu/~rrussell/spfvprox.htm, Junho 1997.

SPOHN, Marco Aurélio. Desenvolvimento e análise de desempenho de um "Packet Session Filter". Porto Alegre – RS: CPGCC/UFRGS, 1997. Tese de Mestrado.

# Apêndice E

# Filtragem com Estados de Serviços baseados em RPC no *Kernel* do Linux[1]

## E.1 Resumo

As tecnologias mais tradicionais de *firewall* – filtros de pacotes e agentes *proxy* – apresentam limitações que dificultam o suporte a certos protocolos e serviços TCP/IP, principalmente aqueles serviços que usam portas alocadas dinamicamente. Filtros de pacotes com estados são uma boa alternativa, pois permitem a criação de regras de filtragem dinâmicas para tais serviços. Este trabalho mostra detalhes da implementação de um filtro de pacotes com estados capaz de tratar serviços baseados em RPC. Foi usado como ponto de partida o filtro com estados implementado no novo *kernel* do Linux (série 2.4). O resultado deste trabalho foi aprovado e será incorporado à próxima versão oficial do núcleo deste sistema operacional.

## E.2 Abstract

Most traditional firewall technologies, such as packet filters and proxy agents, present limitations that make it difficult to support some TCP/IP protocols and services, mainly those services that use ports dynamically allocated. Stateful Packet Filter are a good alternative, because permits the setup of dynamic filtering rules for such services. This work shows details of the implementation of a stateful packet filter able to handle RPC-based services. It was used like start point the stateful

1. Artigo submetido ao simpósio segurança em informática (SSI) 2000.

filter implemented in new Linux kernel (2.4 series). The result of this work was approved and will be officially incorporated into the next operating system version.

## E.3 Introdução

Existem diversos serviços e protocolos de aplicação que podem ser responsabilizados pelo crescimento surpreendente da Internet. Muitos outros estão surgindo rapidamente sobre a infra-estrutura de rede atual como resultado deste sucesso, impulsionados pela tecnologia e devido à pressão de usuários cada vez mais exigentes. Além disso, o aumento na velocidade das linhas de comunicação e no desempenho das máquinas têm facilitado o desenvolvimento de protocolos mais complexos. Tais protocolos envolvem a movimentação de uma grande diversidade de dados pela rede, em quantidades variáveis, com exigências diferentes na qualidade de serviço e, em geral, usam várias portas alocadas dinamicamente para as diferentes sessões ou conexões necessárias.

Por outro lado, a maioria destes serviços não trazem muitos mecanismos de segurança embutidos e, geralmente, herdam muitos dos problemas da suíte TCP/IP original. É importante, portanto, que as tecnologias de segurança sejam capazes de agregar segurança a estes serviços para que usuários, em redes locais com maiores requisitos de segurança, possam usufruir destes atraentes serviços e protocolos. Infelizmente, as tecnologias tradicionais de *firewall* têm suas limitações e nem sempre conseguem tratar convenientemente todos estes interessantes serviços existentes na rede. Serviços baseados em UDP e RPC são representantes importantes desta classe de serviços. Neste trabalho serão tratados exclusivamente os problemas relacionados a serviços baseados em RPC. O leitor mais interessado em uma discussão mais geral sobre o assunto pode buscar informações na referência [12].

Na seção 2, serão discutidos os problemas existentes para a filtragem de serviços baseados em RPC. A seção 3 apresenta brevemente a nova geração de filtros de pacotes – filtros de pacotes com estados (*Stateful Packet Filter, SPF*) – e como filtros com estados podem solucionar os problemas mencionados na seção 2. A seção 4 mostra detalhes da implementação de um SPF no *kernel* de desenvolvimento 2.3.x do Linux. Na seção 5 será visto como este filtro da seção 4 foi estendido para oferecer suporte a serviços baseados em RPC. Finalmente, a seção 6 traz algumas considerações finais e a conclusão do trabalho.

## E.4 Serviços Baseados em RPC

Existem vários protocolos de chamada a procedimentos remotos conhecidos como RPC. O mais popular e que será usado em nossa discussão, neste trabalho, é o Sun RPC, que foi originalmente desenvolvido pela Sun Microsystems [15].

Nos protocolos UDP e TCP, os cabeçalhos dos pacotes reservam somente 2 *bytes* para os campos com números de portas, ou seja, existem somente 65.536 possíveis portas para todos os serviços TCP e UDP (65.536 para cada). Se tivesse que reservar um número de porta bem definido para cada serviço existente, ter-se-ia um número de serviços limitado por este valor. Entre outras coisas, RPC oferece uma boa solução para este problema [2]. Cada serviço baseado em RPC recebe um número de programa único de quatro *bytes* (isto permite 4.294.967.296 serviços diferentes). Como RPC é implementado acima do nível de transporte, deve existir algum mecanismo de mapeamento entre os números dos serviços RPC, oferecidos em uma máquina, e os números de porta que estes serviços estão usando em um dado momento. No Sun RPC, o Portmapper/Rpcbind é o responsável por este mapeamento e é o único servidor baseado em RPC que usa um número de porta pré-definido (porta 111, em ambos TCP e UDP).

Quando um servidor RPC é iniciado, ele aloca um número de porta qualquer e registra-se junto ao Portmapper/Rpcbind. Nesta comunicação, ele passa o seu número de programa, a porta usada, o número de versão e o número do protocolo para o Portmapper/Rpcbind a ser usado na comunicação com os clientes. Vale comentar que o servidor usa o próprio protocolo RPC nesta comunicação. Estas informações são, na verdade, os parâmetros para um procedimento PMAPPROC_SET a ser executado pelo Portmapper/Rpcbind. Um cliente, desejando comunicar-se com este servidor RPC, comunica-se com o Portmapper/Rpcbind para consultar o número de porta usado pelo serviço. Na verdade, ele também faz uma chamada remota ao procedimento PROC_GETPORT, passando como parâmetros o número de programa, a versão e o protocolo usado. No momento em que o cliente obtém a porta usada pelo servidor, ele pode comunicar-se diretamente com o serviço RPC, sem nenhum envolvimento do Portmapper/Rpcbind.

Por não usarem números de portas pré-definidos, não se tem como criar regras estáticas para filtragem de serviços baseados em RPC. A alocação de portas é feita dinamicamente e pode mudar com o passar do tempo (máquina ou servidor podem ser reinicializados, por exemplo). Bloquear tráfego de pacotes para o Portmapper/Rpcbind não resolve, pois um atacante pode

descobrir o número da porta usada por algum serviço baseado em RPC, fazendo tentativas para todas as portas não reservadas. Outros serviços RPC podem usar números de portas pré-definidos, como é o caso do NFS que usa a porta 2049 [14]. Mesmo assim, é aconselhável bloquear acesso ao Portmapper/Rpcbind para evitar que eles sejam usados para ataques à própria máquina. O Portmapper/Rpcbind tem um recurso chamado *proxy forwarder*, que permite que clientes façam uma chamada ao procedimento PMAPPROC_CALLIT, passando como parâmetro o número de um programa RPC, sua versão, um número de procedimento deste serviço e os parâmetros para este procedimento. O Portmapper/Rpcbind faz a chamada ao procedimento do serviço, caso esteja registrado, e retorna os resultados obtidos para o cliente. Notar que isto pode ser perigoso para alguns serviços, uma vez que o Portmapper/Rpcbind tem privilégios de super-usuário e a requisição parece vir de uma máquina confiável (a própria máquina).

A solução encontrada para a filtragem de serviços baseados em RPC, com filtros de pacotes estáticos, é simplesmente bloquear todo tráfego de pacotes UDP, pois a grande maioria destes serviços usam UDP como protocolo de transporte [2].

## E.5 Filtros de Pacotes com Estados (*Stateful Packet Filters*)

Filtros de pacotes tradicionais não mantêm informações de estados das conexões ou sessões que passam pelo *firewall* [12]. Pacotes são analisados independentemente, usando somente as informações existentes nos seus cabeçalhos de rede e transporte e regras de filtragem. Se um pacote qualquer com as *flags* de sincronismo SYN e ACK ligadas chegar ao filtro de pacotes, o filtro supõe que este seja parte de algum canal virtual existente entre a rede interna e a rede externa. Silenciosamente, este o aceita, caso esteja de acordo com o conjunto de regras que implementa a política de segurança local.

Entretanto, um bom *firewall* não deve permitir que pacotes não válidos entrem na rede a ser protegida. Pacotes não válidos são aqueles que não satisfazem as regras de filtragem e/ou não pertencem a nenhuma conexão ou sessão existente através do *firewall*. Isto, além de garantir a perfeita implementação da política de segurança da rede interna, evita diversos ataques já bem conhecidos e outros que poderão aparecer no futuro usando pacotes devidamente construídos e/ou mal formados. Um bom exemplo destes ataques são os pacotes ou fragmentos de pacotes, que satisfazem às regras de filtragem, mas não pertencem a nenhuma conexão ou sessão exis-

tente entre a rede interna e a externa, enviados para derrubar uma máquina alvo, explorando alguma falha na implementação da pilha de protocolos TCP/IP de um determinado sistema operacional existente na rede interna. Aliás, um atacante pode descobrir os sistemas operacionais usados em máquinas internas, enviando alguma seqüência bem definida de pacotes com características não especificadas nos padrões dos protocolos e, de acordo com as respostas obtidas, inferir sobre o sistema usado pela máquina.

Em um filtro de pacotes com estados, quando um pacote SYN/ACK chega ao filtro, ele normalmente verifica, em suas tabelas de estados das conexões e sessões, se este pertence a alguma conexão em andamento através do *firewall*. Caso não exista uma conexão TCP válida para este pacote, ele é descartado e, possivelmente, registrado nos *logs*. Portanto, mesmo que o filtro esteja configurado com uma única regra permitindo todo tráfego de pacotes entre as redes, estes pacotes continuarão sendo bloqueados no *firewall*, já que não pertencem a nenhuma conexão válida segundo suas tabelas de estados. Esta solução garante sempre uma maior segurança, visto que somente pacotes autorizados pelas regras de filtragem ou pertencentes às sessões e conexões válidas entre as redes podem atravessar o filtro. Uma outra vantagem desta solução é a melhora no desempenho, pois somente os primeiros pacotes das conexões ou sessões são verificados com as regras de filtragem, que geralmente é uma tarefa de mais alto custo computacional.

Vale ressaltar aqui que este é o comportamento mais normal das atuais implementações de filtros de pacotes com estados. Nem sempre este processo de filtragem com estados trabalha exatamente desta forma em todas as implementações. Portanto, não existe uma especificação padrão para a implementação de um SPF, ou seja, o termo "*stateful*" especifica que o filtro deve manter informações de estados das sessões e conexões, mas não como estas informações devem ser mantidas nem como elas devem ser usadas no processo de filtragem.

Para alguns serviços e protocolos de aplicação, como o caso do FTP, algum trabalho extra precisa ser feito também no nível de aplicação. Estes codificam certas informações dos níveis de rede e transporte, como números de portas e endereços IP, na parte de dados dos pacotes. Para estes casos especiais, filtros de pacotes com estados podem acompanhar o fluxo de dados da aplicação, extrair as informações necessárias para suas tabelas de estados e para criação de regras dinâmicas. No caso do FTP, o filtro pode extrair, por exemplo, a porta que o cliente usará para

uma conexão de dados. Desta forma, é possível garantir o correto funcionamento destes protocolos e serviços.

Por já fazerem algum trabalho nesta camada de comunicação para alguns serviços, alguns SPFs, como o CISCO-PIX e o FW-1, permitem também alguma filtragem no nível de aplicação, como em agentes *proxy*. Entretanto, a filtragem neste caso é geralmente bastante simples, verificando somente a parte de dados dos pacotes individualmente, sem qualquer remontagem do fluxo de dados do serviço.

Apesar disso, muitos serviços de aplicação transmitem comandos do protocolo e suas respectivas respostas em unidades pequenas, que cabem em um único datagrama IP e podem perfeitamente ser filtrados por filtros de pacotes com estados. Um bom exemplo é o caso da comunicação entre clientes RPC e o Portmapper, que será destacada neste trabalho. As mensagens enviadas pelos clientes, requisitando o número da porta usada por algum servidor RPC, ocupam somente 56 *bytes*, enquanto uma resposta com o número da porta usada ocupa 28 *bytes* da parte de dados. Nestes casos, não há a necessidade de remontagem dos dados dos pacotes de uma comunicação para filtrar comandos do serviço ou protocolo de aplicação. A grande vantagem obtida com esta filtragem é que o SPF desempenha o papel de um agente *proxy* dedicado sem comprometer o desempenho.

Por manter estados das conexões e sessões, um filtro de pacotes com estados permite também associar pacotes ICMP de erro ou controle à sua devida conexão ou sessão. Estes pacotes trazem informações que permitem identificar a conexão ou sessão da máquina à que se refere a mensagem ICMP. Usando estas informações, o filtro pode verificar a existência desta sessão ou conexão em suas tabelas de estados e permitir a passagem dos mesmos.

## E.6 Filtragem com Estados de Serviços Baseados em RPC

Foi visto na seção 2 que a característica de não usar números de porta pré-definidos é o principal problema para a filtragem estática de serviços baseados em RPC. Um servidor RPC aloca o número de porta dinamicamente, junto ao sistema operacional, e registra-se ao servidor de nomes portmapper/rpcbind, informando a porta sendo usada em um determinado momento. Isto impossibilita a criação de regras estáticas para estes serviços usando número de portas fixos e previamente conhecidos.

Uma boa solução para este problema é utilizar informações de estados extraídas da comunicação entre um cliente RPC e o portmapper/rpcbind para possibilitar a criação de regras dinâmicas para tais serviços. Um filtro de pacotes com estados pode simplesmente acompanhar uma sessão UDP (ou uma conexão TCP) envolvendo clientes RPC e o portmapper/rpcbind, dando atenção especial aos dados trocados em todas as chamadas ao procedimento PMAPPROC_GETPORT. Associando requisições às suas respostas e extraindo as informações relevantes, é possível manter as informações das portas sendo usadas e, consequentemente, permitir a comunicação entre clientes e servidores RPC.

Esta solução de acompanhar todas as sessões entre clientes RPC e o portmapper/rpcbind e extrair as informações necessárias foi implementada neste trabalho, como será visto mais adiante na seção 5.

## E.7 IPtables: Um Filtro de Pacotes com Estados no *Kernel* do Linux

Com o aumento no desempenho dos processadores, alguns sistemas operacionais passaram a implementar o processo de filtragem de pacotes no núcleo do sistema. Nas versões 2.0.x e 2.2.x de produção do *kernel* do Linux, por exemplo, pode-se encontrar respectivamente os filtros Ipfwadm e Ipchains, que implementam simplesmente uma filtragem estática dos pacotes. Neste trabalho, entretanto, tem-se mais interesse na nova infra-estrutura usada para tratamento e filtragem dos pacotes que estará presente na série de produção 2.4 do *kernel*, neste sistema operacional.

Como ainda não há uma versão estável deste novo *kernel*, muito do que será apresentado neste documento é baseado nas versões 2.3.x de desenvolvimento e poderá sofrer pequenas mudanças até a sua versão final. Contudo, as idéias básicas usadas na implementação certamente permanecerão as mesmas.

### E.7.1 Netfilter

O Netfilter é um *framework* independente da interface normal de Berkeley *Sockets*, que gerencia e organiza como os pacotes que chegam ao *kernel* do Linux devem ser tratados, nas séries de desenvolvimento 2.3 e de produção 2.4. Ele define, por exemplo, que partes do *kernel* podem

realizar algum trabalho com o pacote quando este atingir certos pontos do núcleo do sistema operacional [13].

Nesta nova infra-estrutura, cada protocolo define "*hooks*" (IPv4 define 5), que são pontos bem definidos na pilha de protocolos por onde passam os pacotes. Quando um pacote chega a um determinado *hook*, o Netfilter checa que módulos do *kernel* estão registrados para oferecer algum tratamento especial naquele ponto. Caso existam, cada módulo seqüencialmente recebe o pacote (desde que, obviamente, ele não seja descartado ou passado para o espaço de processos por um módulo anterior) e pode, portanto, examiná-lo, alterá-lo, descartá-lo, injetá-lo de volta ao *kernel* ou passá-lo para o espaço de processos, dependendo da funcionalidade de rede que esteja implementando. Estes módulos geralmente registram-se em mais de um *hook* ao mesmo tempo, podendo, até mesmo, fazer trabalhos diferentes com os pacotes em cada um deles.

Desta forma, pode-se definir quais módulos devem ser carregados e registrados perante o Netfilter, sempre que algum tratamento especial com os pacotes for necessário. Além disso, esta interessante infra-estrutura facilita a implementação de novos módulos e permite que os módulos pré-existentes sejam facilmente estendidos, usando interfaces de comunicação bem definidas. Alguns dos módulos nativos são o NAT, o Iptables e o módulo de *Connection tracking*.

Notar que, diferentemente das versões mais antigas do *kernel* do Linux, as diversas funções de rede são implementadas em módulos independentes. Nas versões 2.0.x e 2.2.x, o IP *masquerading*, por exemplo, era implementado junto ao processo de filtragem de pacotes e das funções de roteamento no *kernel*. Isso tornava o código confuso e acarretava alguma perda no desempenho. Com este novo esquema, IP *masquerading*, por ser um caso especial de NAT, é implementado usando o módulo de NAT. Este último, por sua vez, é totalmente independente do módulo de filtragem.

### E.7.2 Módulo de *Connection Tracking* (Conntrack)

Este módulo foi originalmente implementado para ser usado pelo módulo de NAT. Todavia, possibilita também que os outros módulos no *kernel* possam utilizá-lo. O papel do módulo Conntrack é manter informações de estados de todas as conexões e sessões. O módulo Iptables, em suas versões iniciais, não fazia filtragem com estados, mas foi rapidamente e facilmente estendido para basear seu processo de decisão também nas informações de estados mantidas pelo módulo

Conntrack. Portanto, a grosso modo, este módulo de *Connection tracking* é o verdadeiro responsável pela parte "*stateful*" do filtro de pacotes implementado no kernel do Linux.

Cada pacote chegando ao *kernel* é associado a uma estrutura de dados (skb_buff). Todos os componentes do núcleo do sistema operacional, que realizam algum trabalho com um determinado pacote, recebem o apontador para a estrutura de dados correspondente. Desta forma, os cabeçalhos e dados do pacote não precisam ser recopiados durante sua travessia pela pilha de protocolos. Portanto, esta estrutura de dados, dentre outras coisas, tem apontadores para os cabeçalhos dos diferentes níveis de rede e para a parte de dados do pacote. Nas séries 2.3 e 2.4, ela apresenta um campo extra (nfmark), no qual o módulo Conntrack codifica o estado para o pacote em sua sessão ou conexão. Vale notar que este módulo não deve descartar pacotes, com exceção de alguns casos excepcionais (pacotes mal formados). Esta função é deixada para o módulo de filtragem (Iptables). Portanto, depois de passar pelo módulo Conntrack, um pacote atravessa o *kernel* com o seu respectivo estado, que pode ser:

- ***INVALID***: para pacotes que não pertencem e não criam uma conexão ou sessão. Pacotes ICMP não válidos são um bom exemplo;
- ***NEW***: para pacotes que iniciam uma nova conexão ou sessão;
- ***ESTABLISHED***: para pacotes pertencendo a uma conexão já estabelecida;
- ***RELATED***: para pacotes relacionados a alguma conexão ou sessão já estabelecida. Por exemplo, pacotes ICMP relacionados a alguma conexão existente.

O módulo de *Connection tracking* atribui um estado a cada pacote que chega ao *kernel*, de acordo com o andamento da comunicação a que pertence. Para isso, ele usa uma tabela *hash* (conhecida como *Separate Chaining*) para manter as informações de estados de todas as conexões e sessões passando pelo núcleo do sistema operacional. Nesta tabela, cada posição guarda uma lista encadeada, onde cada nó representa uma conexão ou sessão (estrutura ip_conntrack_tuple_hash), contendo as informações necessárias para identificá-la unicamente (os endereços IP de origem e destino, os números de portas de origem e destino, o protocolo usado), apontadores para os seus vizinhos na lista e para uma outra estrutura de dados (ip_conntrack) com informações de estados propriamente ditas referentes à comunicação. Usando estas informações, o módulo de Contrack consegue atribuir o estado adequado para cada pacote dentro de sua respectiva conexão ou sessão.

Além da tabela de estados das conexões e sessões, o módulo Conntrack também mantém uma lista de conexões ou sessões aguardadas (expect_list). Cada nó nesta lista guarda as informações dos endereços IP de origem e destino envolvidos, o número da porta destino e o protocolo a ser usado. Se um pacote tiver as mesmas informações em seus cabeçalhos de rede e transporte de uma das entradas nesta lista, o módulo assume que este pertence a esta conexão ou sessão esperada e atribui o estado *RELATED* ao pacote. Dai em diante, todos os pacotes desta conexão ou sessão relacionada recebem sempre o estado de *RELATED*. Além disso, cria-se uma entrada na tabela de estados para esta nova conexão ou sessão e retira a entrada correspondente da lista de conexões ou sessões aguardadas. Isto é equivalente a criar uma regra dinâmica para permitir a comunicação.

Esta lista é, portanto, essencial para permitir o funcionamento correto de certos serviços e protocolos TCP/IP. O FTP é um bom exemplo destes protocolos, pois define o número de porta a ser usado na conexão de dados durante a conexão de controle [14]. Para acomodar bem tais serviços e protocolos, o módulo Conntrack oferece módulos auxiliares para alguns protocolos (conhecidos como "*helpers*") e permite que outros, para protocolos ainda não suportados, sejam facilmente implementados e agrupados ao módulo de *Connection tracking*. Os *helpers* acompanham os dados trocados em uma comunicação envolvendo algum determinado protocolo e extraem as informações necessárias para permitir as conexões ou sessões relacionadas. Com estas informações, este módulo auxiliar pode criar uma entrada para a conexão ou sessão na lista.

Cada uma destas entradas permanece nesta lista enquanto durar a conexão ou sessão que a originou ou até que o primeiro pacote da sessão ou conexão relacionada chegue. Isto significa que não há *timeouts* associados, podendo dificultar um pouco o suporte a certos protocolos. Pode ser necessário, dependendo do protocolo, que um determinado *helper* gerencie uma lista de sessões ou conexões aguardadas particular para o correto tratamento de conexões ou sessões relacionadas. Esta foi a solução adotada para o caso do RPC que será visto na próxima seção.

### E.7.3 Iptables

O Iptables [13] é considerado um filtro de pacotes com estados porque ele permite o uso, no seu processo de filtragem, do estado atribuído pelo módulo de *Connection tracking* aos pacotes das diferentes sessões ou conexões. O módulo Iptables sozinho implementa somente uma filtragem estática dos pacotes como seus antecessores: Ipfwadm e Ipchains. Entretanto, quando o módulo

Conntrack está presente no *kernel*, mantendo os estados das conexões ou sessões, o Iptables permite que o administrador de segurança crie regras baseadas nos estados dos pacotes dentro de suas respectivas conexões ou sessões. Desta forma, é possível a criação de regras permitindo a passagem, através do *firewall*, de todos os pacotes com estados *ESTABLISHED* e *RELATED*, sem comprometimento da segurança requerida.

# E.8 Estendendo o Iptables

## E.8.1 Mensagens RPC

O protocolo RPC [14] envolve basicamente dois tipos de mensagens: mensagens "*call*" e mensagens "*reply*". Uma mensagem *call* é originada quando um cliente RPC envia uma requisição para a execução de um procedimento em particular. Depois que o procedimento é executado, o servidor RPC envia de volta uma mensagem *reply*, contendo os resultados retornados na execução do procedimento. A Figura 1 mostra o formato de uma mensagem *call*, quando encapsulada dentro de um datagrama UDP.

A mensagem *call* inicia com o campo XID (*Transaction* ID) que é um valor inteiro estabelecido pelo cliente e retornado sem nenhuma alteração pelo servidor. Quando o cliente recebe uma mensagem *reply* do servidor, ele compara o campo XID desta última mensagem (observar a existência do campo também na Figura 2) com o XID enviado. Desta forma, o cliente RPC pode verificar se a mensagem recebida se trata de uma resposta válida à sua requisição. É interessante ressaltar ainda que, caso o cliente retransmita a sua mensagem *call*, o valor usado anteriormente no campo XID, na mensagem original, não é alterado. Isto evita problemas com mensagens duplicadas ou atrasadas na rede.

O campo seguinte serve para distinguir entre mensagens *call* (valor inteiro 0) e mensagens *reply* (valor inteiro 1). A versão corrente do protocolo Sun RPC é 2 e, portanto, é este valor que é sempre transmitido no campo "RPC *version*". A seguir pode-se ver os campos "*program number*", "*version number*" e "*procedure number*" que unicamente identificam um procedimento específico no servidor. Os campos "*credentials*" e "*verifier*" são usados para fins de autenticação e não serão tratados em detalhes neste trabalho. Na comunicação entre um cliente e o portmapper/rpcbind, que é mais interessante para este trabalho, não existe nenhum esquema de autenticação, no modo de operação normal, sendo usado. Embora estes campos tenham tamanhos vari-

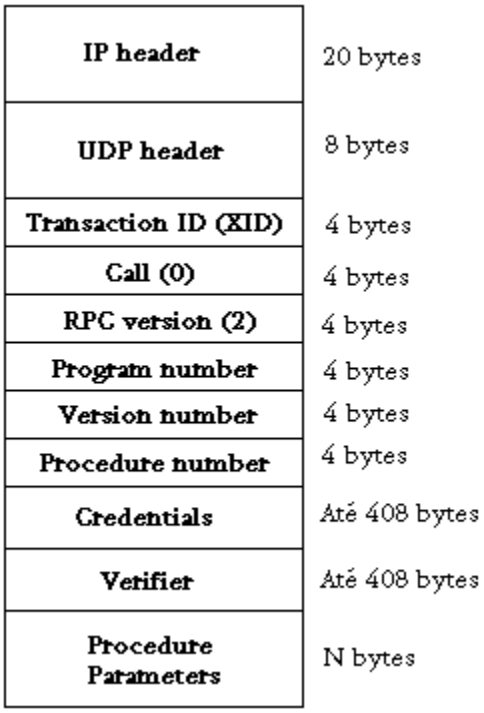


**Figura E.1:** Mensagem *call* com UDP

áveis, o número de *bytes* usados é codificado como parte do campo: os primeiros quatro *bytes*, em cada um destes dois campos, representam o tamanho total dos mesmos. Além disso, mesmo que nenhum esquema de autenticação esteja sendo usado, este campo também deve transmitir um valor inteiro 0, que indica a ausência de um esquema de autenticação.

Finalmente, tem-se o campo "*procedure parameters*" com os parâmetros que devem ser passados ao procedimento. O formato deste campo depende da definição de cada procedimento remoto. Deve-se notar aqui que este campo também tem tamanho variado. Para o caso de um datagrama UDP não existem problemas, pois o tamanho deste campo é igual ao tamanho total do pacote UDP subtraído do comprimento total de todos os outros campos mostrados acima. Entretanto, quando TCP estiver sendo usado, um campo extra de quatro *bytes* entre o cabeçalho TCP e o XID é introduzido, codificando o tamanho total da mensagem RPC.

A Figura 2 mostra o formato de uma mensagem *reply* também encapsulada em um pacote UDP. O XID é copiado da mensagem *call* correspondente, como já foi discutido nesta seção. O campo seguinte tem valor 1 indicando que a mensagem se trata de um *reply*. A versão do pro-

tocolo RPC é 2, como no caso das mensagens *call*. O campo "*status*" recebe o valor 0 se a mensagem foi aceita (ela pode ser rejeitada, no caso de se estar usando outra versão do RPC ou de alguma falha na autenticação, e o valor atribuído, neste caso, será 1). O "*verifier*" tem o mesmo papel do caso anterior para mensagens *call*. O campo "*accept status*" recebe o valor 0 quando o procedimento é executado com sucesso. Neste caso, um valor diferente de zero identifica um determinado caso de erro, como por exemplo, um número de procedimento inválido sendo enviado pelo cliente. Como em mensagens *call*, se o protocolo de transporte TCP estiver sendo usado, um campo extra de quatro *bytes* é introduzido entre o cabeçalho TCP e o XID. Isto é útil para o cliente descobrir o tamanho correto do campo "*procedure results*", com os resultados retornados pelo procedimento.

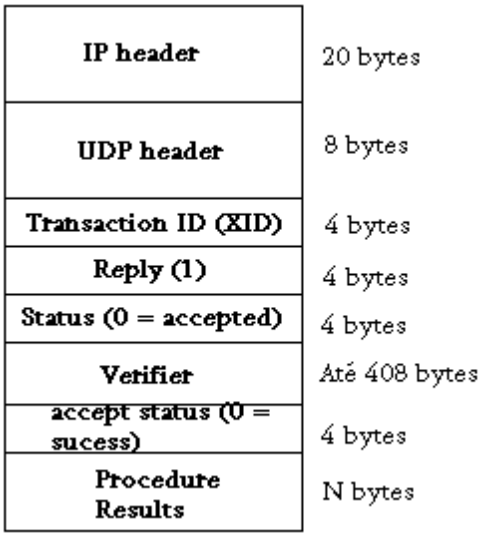


**Figura E.2:** Mensagem reply com UDP

Os campos destas mensagens são codificados usando o padrão de representação XDR, possibilitando que clientes e servidores RPC usem arquiteturas diferentes. XDR define como os vários tipos de dados são representados e transmitidos em uma mensagem RPC (ordem dos *bits*, ordem dos *bytes*, etc.). Portanto, o transmissor cria mensagens codificando todos os dados usando XDR e, do outro lado, o receptor decodifica os dados da mensagem usando este mesmo padrão de representação. Nas figuras 1 e 2, os campos usados transmitem principalmente valores

inteiros. Portanto, pode-se concluir que um inteiro em XDR é codificado usando quatro bytes de dados.

### E.8.2 Estendendo o Módulo de *Connection Tracking*

Inicialmente foi implementado um módulo auxiliar (ip_conntrack_rpc) para inspecionar o andamento de todas as tentativas de comunicação com o portmapper/rpcbind. Somente mensagens RPC, usando UDP ou TCP, requisitando a execução de procedimentos PMAPPROC_GETPORT e suas respectivas respostas são acompanhadas pelo módulo. O objetivo deste módulo é manter uma lista interna (lista de conexões ou sessões RPC aguardadas) com informações de estados extraídas da parte de dados dos pacotes e permitir que um outro módulo (ipt_rpc_known), estendendo o módulo de filtragem Iptables, possa verificar esta lista e, com base nestas informações, fazer a filtragem corretamente dos serviços baseados em RPC. Cada entrada na lista, representada por uma estrutura de dados (expect_rpc), mantém: os endereços IP do cliente e do servidor RPC, o protocolo a ser usado na comunicação, o número da porta usada pelo servidor RPC e um *timeout* associado à entrada.

Como visto anteriormente, o funcionamento do protocolo RPC inicia quando o cliente se comunica com o portmapper/rpcbind para determinar o número da porta usada por um determinado servidor RPC naquela máquina. Nesta comunicação, o cliente passa como parâmetros o número de programa do serviço, o protocolo e a versão desejada. O módulo Conntrack responsável pelo tratamento do RPC acompanha todas estas requisições. Ele, na verdade, cria uma lista de requisições, para manter o histórico de todas as consultas ao portmapper/rpcbind. Cada entrada na lista de requisições (estrutura de dados request_p) mantém: o endereço IP do cliente, o XID, o número de porta do cliente, o protocolo a ser usado na comunicação com o servidor RPC e um *timeout* associado à entrada. Este timeout evita que consultas forjadas criem um número indefinido de novas entradas em ataques de DoS (*Denial of Service*), consumindo recursos na máquina filtradora.

Caso este servidor esteja registrado, o portmapper/rpcbind retornará o número da porta usada pelo servidor naquela máquina. Quando esta mensagem *reply* chega ao módulo Conntrack, este pode verificar se existe uma entrada na lista de requisições e, caso exista, pode criar uma entrada na lista de conexões ou sessões RPC aguardadas. Nesta verificação, além das informações mantidas na lista de requisições, o módulo observa o XID, o endereço IP e a porta do cliente

nestas mensagens. Isso é necessário, pois a informação do protocolo a ser usado não está disponível na mensagem *reply*. Para que uma entrada seja criada na lista de conexões ou sessões RPC aguardadas esta informação também é necessária.

Se não existir uma entrada na lista de requisições correspondendo à mensagem *reply*, o módulo Conntrack simplesmente descarta a mensagem para um acompanhamento mais correto das comunicações. O leitor pode imaginar uma situação em que o portmapper/rpcbind, por algum motivo, atrasa a execução do procedimento PMAPPROC_GETPORT e o cliente retransmite a requisição. Pode-se supor então um cenário onde o *timeout* para entrada na lista de requisições expira e a mensagem de requisição retransmitida pelo cliente é perdida em trânsito e, portanto, não seria vista pelo módulo Conntrack. Caso esta mensagem *reply* chegue ao cliente, ele tentará fazer a comunicação com o servidor RPC normalmente. Entretanto, esta comunicação não será permitida devido a inexistência de uma entrada para ela na lista de conexões ou sessões RPC aguardadas.

Um outro aspecto importante é o tratamento das possíveis comunicações subseqüentes dos clientes com os servidores RPC. O *timeout* associado a cada entrada na lista de conexões ou sessões aguardadas tem como objetivo básico permitir futuras tentativas de comunicação de clientes usando sua respectiva *cache* de respostas anteriores do portmapper/rpcbind. Se a entrada fosse excluída imediatamente após o fim da comunicação, futuras tentativas não seriam permitidas. Obviamente, clientes bem implementados devem prover um esquema de recuperação para falhas deste tipo. Esta falha seria similar ao caso de um servidor registrado ser reinicializado e o cliente, usando as informações da *cache*, tentar usar o serviço.

### E.8.3 Estendendo o Iptables

Como já foi mencionado, o módulo de filtragem, nas séries 2.3 e 2.4 do *kernel*, baseia sua decisão também em informações de estados dos pacotes dentro de suas respectivas sessões ou conexões. Um módulo estendendo o Iptables foi criado para tratar todos os pacotes RPC com estado *NEW* (iniciando uma sessão ou conexão) chegando ao módulo de filtragem. Desta forma, o Iptables permite todas as tentativas de comunicação com o portmapper/rpcbind com consultas aos números de portas sendo usadas pelos servidores registrados. Portanto, não há a necessidade de uma regra permitindo explicitamente a comunicação com o servidor de nomes do RPC. Este novo

módulo do Iptables também é o responsável em verificar as entradas da lista de conexões e sessões aguardadas e permitir que clientes se comuniquem normalmente com servidores RPC.

### E.8.4 Testes de Sanidade e Cuidados Adicionais com Fragmentos e Pacotes Truncados

Para um correto acompanhamento das mensagens RPC, alguns cuidados devem ser tomados com respeito à sanidade dos pacotes. Além disso, os módulos devem tomar cuidados extras para evitar que pessoas mal intencionadas possam subverter o mecanismo de filtragem de pacotes. O módulo Conntrack, antes de inspecionar a parte de dados de todos os pacotes RPC, verifica se há uma má formação e calcula o *checksum* do cabeçalho de transporte no pacote. Se UDP estiver sendo usado, esta última operação somente é feita se o campo de *checksum* tiver um valor diferente de zero, indicando que ele foi calculado, já que a checagem em UDP é opcional e percebida através do valor diferente de zero.

Pacotes e fragmentos de pacotes cuidadosamente construídos foram previstos durante a implementação, tendo-se em mente, principalmente, a vulnerabilidade recente encontrada no FW-1 e Cisco-PIX no suporte ao FTP. Com este problema, um atacante é capaz de subverter o filtro de pacotes com estados. Para isto, bastava forçar que os pacotes do protocolo FTP fossem devidamente truncados e, explorando um problema existente na inspeção da parte de dados, abrir brechas no *firewall*.

Como já mencionado, as mensagens RPC trocadas entre clientes e o portmapper/rpcbind são extremamente pequenas. Desta forma, cabem facilmente em qualquer MTU (*Maximum Transmission Unit*) existente nas interfaces de rede atuais. Desta forma, pacotes RPC fragmentados são tratados como um comportamento anormal e simplesmente ignorados pelo módulo Conntrack. Além disso, pacotes RPC, onde a parte de dados é maior que o esperado, são imediatamente descartados.

Se um cliente usar TCP para a comunicação com o portmapper/rpcbind, os valores dos atributos da conexão devem ser sempre suficientemente grandes para não truncar a parte de dados nos pacotes. Em geral, o valor mínimo atribuído ao parâmetro MSS (*Maximum Segment Size*), por exemplo, é idêntico à MTU usada pelo meio físico para evitar a indesejável fragmentação de pacotes. Pode-se concluir, com isso, que não há boas razões para usar valores muito pequenos

para este parâmetro e, portanto, qualquer tentativa disto é vista como um ataque contra o filtro de pacotes com estados.

## E.9 Conslusão

Este trabalho mostrou que filtros de pacotes com estados oferecem condições para a filtragem de serviços baseados em RPC. Vale notar ainda que soluções semelhantes podem ser adotadas para outros protocolos e serviços mais complexos. Um bom exemplo é o serviço Real Audio, para transmissão de áudio na Internet. Neste serviço, o cliente estabelece uma conexão TCP usando a porta 7070 do servidor. Usando esta conexão, ele envia suas requisições e números de portas UDP que ficarão esperando os dados do servidor. Múltiplas sessões UDP serão usadas para aumentar a vazão, garantindo a qualidade do som. Esta quantidade de portas usadas pode sobrecarregar muito mais rapidamente *proxies* dedicados. Além disso, o aumento no *overhead* pode prejudicar a qualidade do som. Um SPF, por outro lado, pode acompanhar a conexão TCP e permitir as sessões UDP muito facilmente. Vale ressaltar, porém, que agentes *proxy* dedicados devem ser adicionados sempre que o fluxo de dados para algum serviço precisar ser remontado/ normalizado e inspecionado mais detalhadamente.

## E.10 Agradecimentos

Os autores agradecem ao CNPq, à FAEP- UNICAMP pelo financiamento ao trabalho e, principalmente, ao Rusty Russel por todos os esclarecimentos e sugestões a respeito da nova infra-estrutura de tratamento e filtragem de pacotes no *kernel* do Linux.

## E.11 Referências Bibliográficas

[1] AVOLIO, Frederick M. Application Gateways and Stateful Inspection: Brief Note Comparing and Contrasting. Trusted Information System, Inc. (TIS). Janeiro, 1998.

[2] CHAPMAN, D. Brent; ZWICKY, D. Elizabeth. Building Internet Firewalls. O' Reilly & Associates, Inc. 1995.

[3] CHECKPOINT, Software Technologies Ltd. CheckPoint Firewall-1 White Paper, Version 3.0, junho, 1997.

[4] CHECKPOINT, Software Technologies Ltd. Stateful Inspection Firewall Technology, Tech Note, 1998.

[5] CHESWICK, William ; BELLOVIN, Steven M. Firewalls and Internet Security – Repelling the wily hacker. Addison Wesley, 1994.

[6] GONÇALVES, Marcus. Firewalls Complete. McGraw-Hill, 1997.

[7] POUW, Keesje Duarte. Segurança na Arquitetura TCP/IP: de Firewalls a Canais Seguros. Campinas: IC-Unicamp, Janeiro, 1999. Tese de Mestrado.

[8] POUW, Keesje Duarte; GEUS, Paulo Licio de. Uma Análise das Vulnerabilidades dos Firewalls. WAIS'96 (Workshop sobre Administração e Integração de Sistemas), Fortaleza. Maio, 1997.

[9] RUSSEL, Ryan. Proxies vs. Stateful Packet Filters. http://futon.sfsu.edu/~rrussell/spfvprox.htm, Junho 1997.

[10] SPONPH, Marco Aurélio. Desenvolvimento e análise de desempenho de um "Packet Session Filter". Porto Alegre – RS: CPGCC/UFRGS, 1997. Tese de Mestrado.

[11] LIMA, Marcelo Barbosa; GEUS, Paulo Lício de. Comparação entre Filtro de Pacotes com Estados e Tecnologias Tradicionais de Fiewall. SSI99, São José dos Campos, 1999.

[12] LIMA, Marcelo Barbosa. Provisão de Serviços Inseguros Usando Filtros de Pacotes com Estados. Campinas: IC-Unicamp, Setembro 2000.

[13] Netfilter Home Page. URL: http://netfilter.kernelnotes.org. Julho, 2000.

[14] STEVENS, W. Richard. TCP/IP Illustrated. Addison Wesley, 1994.

[15] Sun Microsystems. RPC: Remote Procedure Call Protocol Specification. Version 2. RFC 1057, Junho, 1988.