AGENTES PROXY: CONCEITOS E TÉCNICAS DE IMPLEMENTAÇÃO

João Porto de Albuquerque Pereira*
Instituto de Computação
Universidade Estadual de Campinas
13083-970 Campinas - SP
joao.porto@ic.unicamp.br

Paulo Lício de Geus
Instituto de Computação
Universidade Estadual de Campinas
13083-970 Campinas - SP
+55 19 3788-5865 *paulo@ic.unicamp.br*

RESUMO

Este trabalho concentra-se em Agentes "Proxy", programas utilizados como componentes de "firewalls" em sistemas de segurança de redes. São apresentados os principais conceitos relativos a agentes "proxy" e técnicas de implementação são descritas e analisadas, oferecendo parâmetros de escolha dependendo do tipo de aplicação à qual se direciona o agente.

ABSTRACT

This paper is about Proxy Agents, that are programs generally used as components of firewalls in security systems. The main concepts related with proxy agents are presented and implementation techniques are described and analyzed, giving choice parameters depending on the kind of application that the agent is made for.

1 INTRODUÇÃO

Agentes "proxy" são programas que aceitam solicitações de conexão a partir de máquinas clientes e efetuam conexão correspondente com o servidor final pretendido, repassando informações nos dois sentidos da comunicação. A forma de execução deste repasse caracteriza o agente proxy, podendo variar desde o simples baldeamento "cego" dos bits recebidos a complexas análises semânticas do tráfego.

Assim, o agente proxy atua duplamente como servidor, do ponto de vista do cliente; e como cliente, do ponto de vista do servidor, substituindo cada um dos agentes de comunicação, dependo do sentido em que se analisa a comunicação. Por esta razão, foi preferida a nomenclatura "agente proxy" neste trabalho em detrimento do termo também comumente utilizado "servidor proxy".

Os agentes proxy são geralmente utilizados em firewalls para restringir e controlar o tráfego de dados entre uma rede protegida e a rede pública, potencialmente perigosa.

2 CONCEITOS

A máquina que contém o agente proxy é, tipicamente, *dual homed*, ou seja, contém interfaces em duas redes distintas, mas não roteia pacotes. Dessa forma, a única maneira dos dados passarem de uma rede à outra deve ser através da aplicação proxy.

Como a maioria dos protocolos de redes utilizados é assimétrica (i. e., possui implementações distintas de cliente e servidor), a aplicação proxy deve atuar duplamente como servidor - respondendo às requisições do cliente na rede interna - e cliente - gerando pedidos ao servidor na rede externa.

Pode-se estabelecer duas abordagens para agentes proxy: os proxies tradicionais e os proxies transparentes.

2.1 Proxy Tradicional

Na abordagem tradicional, o agente proxy deve:

- Aceitar as conexões vindas do cliente (atuando como servidor);
- Receber do cliente o nome ou endereço final do servidor a ser contatado;
- Iniciar uma sessão para o servidor (atuando como cliente);
- Repassar as requisições, respostas e dados entre o cliente interno e o servidor externo, baseando-se em critérios e regras.

É nesta última atribuição dos proxies que está sua grande vantagem, pois se podem efetuar verificações de tipo de dados e restrições de acesso mais elaboradas do que através da filtragem, por exemplo. Como a maioria das aplicações proxy conhece o protocolo de uma aplicação específica, podemos registrar apenas informações relevantes sobre as conexões efetuadas, permitindo análise posterior (logging mais preciso).

Uma vantagem adicional do proxy é que ao receber os dados que vêm da rede externa (potencialmente perigosa), ele "abre" os datagramas, ou seja, extrai os dados de sua carga útil (*payload*), verifica os dados e constrói novos datagramas, enviando-os ao cliente na rede interna. Dessa forma, trafegam internamente apenas pacotes "seguros", gerados pelo proxy e, portanto, sem "vícios" nos cabeçalhos,

*Bolsista FAPESP

isto é, cabeçalhos sem opções maliciosas que visam explorar alguma vulnerabilidade da pilha TCP/IP.

Para que o agente proxy inicie uma conexão com um servidor da rede externa ele precisa receber do cliente interno o endereço final que este deseja contatar. Uma maneira de fornecer este endereço é a utilização de aplicações clientes modificadas. Ao invés de simplesmente iniciar a conexão ao endereço informado pelo usuário, esta aplicação deve primeiro contactar o agente proxy, enviar-lhe o endereço do servidor externo e só depois então proceder normalmente com o protocolo da aplicação.

Uma desvantagem deste tipo de conexão acontece quando o usuário tem que se conectar algumas vezes a máquinas na rede interna e outras vezes a servidores externos. Nesta situação é necessário manter dois tipos de aplicação cliente: o cliente modificado, para conexões externas; e o cliente padrão, para conexões locais. Existem alguns clientes, atualmente, que suprimem a necessidade da manutenção de clientes duplos através de configurações de endereços que não devem ser contatados via proxy (e. g., Netscape Navigator, Microsoft Internet Explorer), entretanto estas configurações não são muito flexíveis e estão disponíveis em poucas aplicações.

Outra forma de passar o endereço externo final ao servidor proxy é alterando o procedimento do usuário. Neste caso, o usuário, utilizando o software cliente padrão, deve conectar-se primeiramente ao servidor proxy e informar a ele o endereço do servidor externo desejado. Este é o procedimento adotado por muitos agentes proxies largamente utilizados, como os proxies de FTP. Quando o usuário deseja contactar um servidor da rede externa, ele utiliza o cliente padrão FTP e inicia uma conexão com o servidor proxy. Quando o proxy requisitar um nome de usuário, ele deve passar a seguinte informação: *usuário@servidor_externo*, onde *usuário* é o nome do usuário a ser utilizado na conexão com o servidor externo e *servidor_externo*, o nome (ou endereço) do servidor destino final.

Não se deve desconsiderar o custo com treinamento dos usuários finais ao escolher essa abordagem. Dependendo de fatores como tamanho da organização, número de funcionários e grau de familiaridade destes com o sistema, os custos podem tornar-se proibitivos.

2.2 Proxy Transparente

O proxy transparente surgiu da idéia de evitar as alterações - na aplicação cliente ou no procedimento do usuário - necessárias para a abordagem tradicional, mantendo as vantagens de segurança proporcionadas por um agente proxy.

Nesta abordagem, os clientes devem ser configurados de maneira a utilizar a máquina que executa o agente proxy como roteador para os endereços externos (*default gateway*). A conexão através do proxy transparente é, então, realizada da seguinte forma:

- O cliente envia o pacote à máquina proxy, com o endereço de destino contendo o servidor externo desejado;
- A máquina proxy "captura" o pacote e o entrega à aplicação proxy oculta;
- A aplicação proxy inicia uma conexão para o servidor externo (como num proxy tradicional);
- A aplicação proxy repassa as requisições, respostas e dados entre o cliente interno e o servidor externo, baseando-se em critérios e regras.

A grande vantagem desta abordagem é a completa transparência do processo proxy para os clientes, eliminando a necessidade de alteração em sua rotina de trabalho. Do ponto de vista dos clientes, o proxy é visto como um roteador (possivelmente com filtragem de pacotes), e para o servidor externo, atua como um proxy tradicional.

A aplicação proxy transparente deve ter a mesma complexidade de uma aplicação proxy tradicional, entretanto a pilha TCP/IP da máquina proxy não pode seguir totalmente o padrão (pelo menos o padrão atual) . Segundo a especificação padrão, quando uma máquina recebe um datagrama IP cujo endereço de destino não pertence a ela mesma, há dois comportamentos possíveis: descartar (*drop*) ou repassar (*forward*). Porém, em uma máquina proxy, deve ser possível especificar endereços (e portas) que não pertencem à mesma, mas que, ao receber pacotes para estes, a pilha deve "capturá-los" e entregá-los ao agente proxy. Adicionalmente, a pilha TCP deve fornecer um meio para que o agente proxy recupere o endereço original para o qual era destinado o datagrama (ferindo, obviamente, o isolamento entre as camadas de rede). Outro requisito fora do padrão é a capacidade de estabelecer uma conexão com o cliente utilizando um endereço "local" que não pertence ao servidor ("apropriando-se" do endereço do servidor externo final).

2.3 Proxies de Circuito *versus* Proxies de Aplicação

Um proxy de circuito ("*circuit proxy*") realiza apenas a retransmissão dos dados contidos nos pacotes recebidos sem avaliar o seu conteúdo. Dessa forma, efetua a verificação dos dados que serão passados através de um conjunto estático de regras, independente do estado da conexão. Como exemplo pode-se citar um proxy de circuito de FTP que sempre repassa o tráfego na porta 20 (*ftp-data*) mesmo que não haja conexão de controle na porta 21 (*ftp-control*).

Um proxy de aplicação ("*application proxy*") conhece o protocolo da aplicação, podendo efetuar verificações no tráfego de dados baseadas em critérios específicos desta aplicação. Dessa forma, pode-se proteger a rede de ataques no nível da aplicação, que geralmente seriam ignorados por um firewall baseado apenas em filtragem de pacotes.

3 TÉCNICAS DE IMPLEMENTAÇÃO

A estrutura básica de uma implementação de agentes proxy consiste num laço (loop) de espera por pedidos de conexão de clientes em uma determinada porta da máquina proxy e, quando isto acontece, o atendimento a esta conexão. Esta estrutura é comum a qualquer servidor de aplicações TCP/IP, evidenciando a porção servidor do agente proxy.

3.1 Aguardando conexões

Para que o agente "escute" determinada porta da máquina proxy é necessário criar um `socket` e associá-lo a esta porta através da chamada `bind`. Esta associação pode ser feita especificando o endereço IP de uma interface ou a constante INADDR_ANY, que faz com que valha para qualquer endereço.

Num ambiente de segurança é recomendável a associação apenas ao endereço no qual o agente receberá os pedidos de conexão, evitando assim que ele seja utilizado maliciosamente em um sentido diferente do originalmente pretendido. Um *firewall* funcionando corretamente bloquearia esta utilização através da filtragem de pacotes, porém se um atacante conseguir burlar o filtro pode tentar utilizar o proxy para acessar a rede interna protegida. Se for necessário que o agente esteja associado a mais de um endereço, é aconselhável criar um `socket` para cada um deles.

Segue um trecho de implementação em C para esta seção:

```
struct sockaddr_in laddr;
int lfd;

//cria socket
lfd = socket(AF_INET),
             SOCK_STREAM, 0);

// prepara socket address
laddr.sin_family = AF_INET;
inet_pton( AF_INET,
   "10.1.1.1", &laddr.sin_addr );
laddr.sin_port = htons();

//associa socket ao endereço
bind(lfd, &laddr, sizeof(laddr));

// Aloca buffers para
   "escutar" porta local
listen( lfd, 10 );
```

3.2 Controle de Processos

Para que o agente proxy possa atender diversas conexões simultaneamente, uma escolha natural é a utilização de algum método de controle de processos (*process control*) para tratar as conexões que estão sendo atendidas.

O método mais simples para o tratamento destas conexões é um servidor iterativo, que utilizaria a chamada de sistema `select` para as conexões ativas. Ao retornar, o programa verificaria cada conexão uma por vez, atendendo àquelas que tiverem recebido dados. Por atendimento aqui entende-se o repasse lógico dos dados, como explicado em seções anteriores. No entanto este método é extremamente ineficiente pois não tira proveito dos recursos de *time-sharing* presentes nos sistemas operacionais modernos, enfrentando, portanto, todos os conhecidos problemas de um controle de processos rudimentar como este (*first come first serviced*[1]).

A primeira alternativa para implementação do controle de processamento concorrente beneficiando-se dos recursos de escalonamento de processos do sistema operacional é utilizar a seguinte estratégia: para cada nova conexão recebida executa-se uma chamada `fork` criando novo processo filho para tratar esta conexão. O processo pai volta, então, a aguardar por novas conexões. Este é um paradigma padrão do modelo Unix, largamente utilizado em servidores de aplicação TCP/IP.

Outra opção para este gerenciamento é a utilização de *threads* para tratar cada uma das conexões ativas. O ganho de performance das *threads*, relativamente leves, sobre o caro processo de `fork` pode ser interessante em agentes proxy com alta demanda, justificando sua adoção. Uma vantagem adicional do uso de *threads* acontece em agentes proxy que necessitam de comunicação entre seus processos. Por exemplo, em agentes proxy que mantêm uma tabela de estados (*stateful proxying*), as *threads* filhas podem realizar alterações na tabela global (utilizando, é claro, mecanismos para garantir a exclusão mútua no acesso), que será consultada pela *thread* principal para decidir sobre a aceitação de uma nova conexão. A implementação desta estratégia utilizando `fork` seria bem mais complexa.

Há, entretanto, uma desvantagem na utilização de *threads*. Como todas as *threads* compartilham o mesmo espaço de memória (instruções, dados globais, arquivos abertos, etc.), o agente fica mais vulnerável ao ataque de cada uma das conexões ativas, que pode afetar todas as outras. Num agente que utiliza `fork`, por exemplo, ao derrubar uma conexão, explorando alguma vulnerabilidade do código, as demais não são afetadas. Já num agente baseado em *threads*, todo o conjunto cairia.

Independente do método de controle de processos (`fork` ou *threads*), é sempre interessante restringir o número de conexões simultâneas, através da não aceitação de novas conexões depois de um número máximo. Desta forma, os recursos do sistema utilizados pelo agente são delimitados e os efeitos de ataques *DoS (Denial of Service)* são minimizados sobre a máquina como um todo.

A seguir um exemplo de implementação de con-

[1] mais detalhes podem ser obtidos em Silberschatz[6]

trole de processos, utilizando `fork`:

```
for (;;) {

  len = sizeof(client_addr);
  cfd = accept(lfd, &client_addr, &len);
  if (cfd < 0) {
    if (errno == EINTR) continue;
    else {
      syslog(LOG_ERR,
        "unexpected error- aborted");
      exit(1);
    }
  }

  /* Cria processo filho
     para tratar conexao */
  ch_pid = fork();
  if (ch_pid == 0) {
    close(lfd);
    handle_session(cfd,&client_addr,len);
    exit(0);
  }

  close(cfd);
}
```

Neste código, a função `handle_session` recebe o *file descriptor* de uma conexão do cliente e seu endereço, efetua, então, a conexão com o servidor e o tratamento do tráfego desta sessão (repasse). Foram omitidas as declarações de variáveis cujos tipos são imediatamente dedutíveis.

Maiores detalhes e exemplos sobre implementação de processamento concorrente podem ser obtidos em *Unix Network Programming*[1].

3.3 Tratamento de Sessão

Até este ponto, evidenciou-se a porção "servidor" de um agente proxy. Nesta seção será abordada sua porção "cliente", responsável pelo tratamento de um pedido de sessão a partir de um cliente, direcionado a determinado processo (ou *thread*), como analisado na seção anterior. No exemplo de implementação, a função `handle_session` desempenha esta atribuição.

O tratamento de uma sessão é o coração do agente proxy, que consiste em receber os dados do cliente e do servidor e efetuar seu repasse lógico. É nele que residem as regras específicas da aplicação coberta pelo agente proxy, caso seja ele um proxy de aplicação.

Entretanto, o modo de operação deste repasse, ou seja, as técnicas utilizadas para efetuá-lo, são genéricas e comuns a qualquer tipo de agente proxy TCP. Elas constituirão o objeto de análise desta seção, exemplificadas através de um proxy de circuito TCP.

Como visto anteriormente, uma sessão proxy é composta por duas conexões TCP: uma entre o cliente e o agente proxy e outra entre o agente proxy e o servidor. Uma forma simples de efetuar o repasse lógico é a utilização da chamada de sistema `select` para bloquear o processo e aguardar recebimento de dados em alguma das duas conexões. Ao retornar desta chamada, o agente verifica a existência de dados recebidos em cada uma das direções. Em caso positivo efetua uma leitura (através da chamada `recv`, por exemplo) e envia os dados recebidos para a outra conexão (chamada `write`).

A seguir trecho de implementação utilizando a técnica que acaba de ser descrita:

```
FD_ZERO(&fds);
FD_SET(cfd,&fds);
FD_SET(lfd,&fds);

while(FD_ISSET(cfd,&fds) ||
      FD_ISSET(lfd,&fds)) {
  bcopy(&fds,&rds,sizeof(fd_set));
  bcopy(&fds,&xds,sizeof(fd_set));

  tv.tv_sec = MAX_TIMEOUT;
  tv.tv_usec = 0;

  ret = select(cfd>lfd ? cfd+1:lfd+1,
                &rds,NULL,&xds,&tv);

  /* Verifica TIMEOUT */
  if (!ret) {
    syslog(LOG_WARNING,
      "(timeout) connection closed");
    exit(1);
  }

  /* Repassando dados do
     cliente -> servidor */
  if (FD_ISSET(cfd,&rds)) {
    n = recv(cfd,buff,sizeof(buff),0);
    if (n<=0) FD_CLR(cfd,&fds);
    else n = write (lfd,buff,n);
  }

  /* Repassando dados do
     servidor -> cliente */
  if (FD_ISSET(lfd,&rds)) {
    n = recv(lfd,buff,sizeof(buff),0);
    if (n<=0) FD_CLR(lfd,&fds);
    else n = write(cfd,buff,n);
  }
}
close(lfd);
close(cfd);
return 0;
```

3.4 Implementando Transparência

A implementação proxies transparentes (descritos na seção 2.2), possui apenas uma fase adicional em relação à abordagem tradicional: a obtenção do endereço do servidor final com o qual o cliente deseja conectar-se, que é o endereço IP ao qual o datagra-

ma foi originalmente destinado. Como dito anteriormente, esta obtenção não faz parte do padrão da pilha TCP/IP e desta forma varia bastante com o sistema operacional utilizado, divergindo muitas vezes até mesmo em diferentes versões de um mesmo sistema.

Como exemplo, no sistema Linux, kernel 2.2, a função a ser utilizada para obter este recurso era `getsockname`, passando o *file descriptor* do `socket` através do qual o pacote foi recebido. Já no kernel 2.4, para obter o endereço original deve-se utilizar a chamada `getsockopt`, passando o parâmetro SO_ORIGINAL_DST e o *file decriptor*. Os programadores do Linux consideraram esta uma forma mais elegante do que a chamada não documentada oficialmente de `getsockname` utilizada anteriormente.

Um exemplo de implementação utilizando o kernel 2.4:

```
/* file descriptor do socket
   que recebeu o pedido de conexão /*
int fd;
struct sockaddr *sa;
int salen = sizeof(sa);
getsockopt(fd, SOL_IP,
   SO_ORIGINAL_DST, sa, &salen)
// endereço está em sa
```

3.5 Sintonia Fina

Nesta seção serão apresentados dois recursos extras para aprimorar a segurança de agentes proxy: `setuid` e `chroot`.

A chamada `chroot` é utilizada para alterar o diretório raiz (*root*) do sistema e torna-se interessante para limitar o acesso do processo (neste caso o agente proxy) apenas à árvore de diretórios abaixo de determinado diretório (que é passado como parâmetro), "engaiolando-o" apenas com os arquivos realmente necessários a seu funcionamento. Dessa forma, evita-se que através da exploração de alguma vulnerabilidade do agente proxy, um atacante consiga alterar arquivos vitais do sistema. Há, entretanto, possibilidade de quebrar esta "gaiola" se o usuário do agente proxy for "*root*". Este problema pode ser minimizado através da utilização conjunta da chamada `setuid`, descrita a seguir.

Os agentes proxy geralmente são postos em execução durante a carga do sistema e ficam carregados na memória como "*daemons*". Por este motivo eles geralmente são invocados com privilégios de super-usuário (*root*), o que é potencialmente perigoso. Embora um agente proxy seja, por definição, um programa simples, robusto e, assim sendo, possua poucas vulnerabilidades a serem exploradas, pode-se minimizar os possíveis danos decorrentes de um ataque ao agente executando-o com menos privilégios. Para isto, utiliza-se a chamada `setuid`, que altera o usuário "*owner*" do processo para outro possuindo apenas os privilégios necessários para o serviço proxy (geralmente bem restritos).

Uma vantagem adicional da utilização de `setuid` é poder ajustar as regras de filtragem de maneira mais precisa, permitindo a passagem apenas de pacotes gerados por programas locais que tenham como "*owner*" o usuário proxy (sendo necessário para isto que o agente proxy esteja sendo executado na mesma máquina que efetua filtragem). Dificulta-se, dessa maneira, a evasão do firewall através desta regra em particular, ou seja, torna-se mais difícil que através da regra seja permitido tráfego diferente daquele originalmente pretendido. Na plataforma Linux, kernel 2.4, este recurso pode ser obtido através do módulo *owner*, da ferramenta *iptables* (para maiores detalhes consultar [5]).

4 CONCLUSÃO

Neste trabalho foram apresentados os principais conceitos relativos a agentes proxy que podem ser encontrados dispersos nas referências especializadas. As técnicas de implementação descritas são as mais comumente utilizadas em cada caso analisado. Possíveis extensões a este trabalho poderiam envolver a análise comparativa quantitativa de outras técnicas de implementação (e.g. *non-blocking I/O, pre-forking*) e abranger outros casos particulares (como os proxies *UDP*). Além disso, outra linha de pesquisa interessante é a constituição de um "esqueleto" contendo as técnicas comuns de implementação de agentes proxy que possa ser utilizado por diferentes tipos de proxy. Dessa forma, o desenvolvimento de novos agentes utilizando esta ferramenta concentrar-se-ia apenas nos detalhes específicos de cada aplicação à que se destina o proxy.

Referências

[1] Stevens, R. W. (1994). *Unix Network Programming, Volume I* . 2nd Edition, Addison-Wesley.

[2] Stevens, R. W. (1994). *TCP/IP Illustrated, Volume I* . Addison-Wesley.

[3] Zwicky, E. D., Cooper, S., Chapman, D. B. (2000). *Building Internet Firewalls*. 2nd Edition, O'Reilly and Associates.

[4] Chatel, M. (1996). *Classical versus Transparent IP Proxies*. RFC 1919.

[5] Russel, R. (2001). *IPTABLES HOWTO (Linux Documentation Project)* http://netfilter.samba.org/.

[6] Silberschatz, A., Galvin, P. B. (1998). *Operating System Concepts*. 5th Edition, Addison-Wesley.