SCOM: SCAN DE PORTAS DE COMUNICAÇÃO REMOTAS

Stéphany Moraes Martins
stephany@ulbra−to.br

Claudio de Castro Monteiro
ccm@ulbra−to.br

ULBRA − Universidade Luterana do Brasil
CEULP − Centro Universitário Luterano de Palmas
Curso de Bacharelado em Sistemas de Informação
Grupo de Pesquisa Aplicada em Redes de Computadores − GPARC
www.ulbra−to.br/~gparc
Palmas −Tocantins
Fone: (63) 223−2067

Resumo

Este trabalho é fruto das pesquisas realizadas no Grupo de Pesquisa Aplicada em Redes de Computadores −GPARC. A aplicação que desenvolvemos tem a finalidade de comunicar−se a um servidor para identificar as portas de comunicação existentes.

1. Introdução

É sabido que *sites* fornecedores de serviços pela Internet, tais como, lojas virtuais, provedores de *Internet* e correio eletrônico, enfrentam sérios problemas para garantir a segurança das informações que trafegam na rede. Isto ocorre devido ao fato de não haver uma cultura de segurança nas organizações e ferramentas suficientemente capazes de dar suporte aos administradores de redes de computadores na detecção ou prevenção de ataques de *hackers*.

O trabalho apresentado a seguir não buscou resolver o problema que os administradores de redes de computadores enfrentam, mas, abrir caminhos com a realização de estudos sobre a comunicação de dados. Os estudos realizados possibilitam a construção de aplicações capazes de buscar informações em diversos sites tais como, serviços disponíveis, versão de aplicações, fluxo de requisição de conexões. Essas informações, posteriormente, contribuirão para o desenvolvimento de um projeto maior intitulado: "AAMIGROOVE:“Um *framework* para Construção de Agentes Pró−ativos de Segurança de Sistemas", do qual esse trabalho faz parte.

No desenvolvimento dessa pesquisa, construímos uma aplicação chamada SCom (*Scan* de Portas de Comunicação Remota). Essa ferramenta tem finalidade acadêmica podendo auxiliar as disciplinas de Redes de Computadores I e II, assim como aqueles que estejam iniciando em projetos de pesquisas nessa área. Mas, também pode contribuir com o administrador do sistema no diagnóstico do funcionamento de seus serviços disponíveis e informar aos usuários do sistema quais serviços existem, antes que os solicite.

2. AAMIGROOVE: Utilização de Agentes para a Reconfiguração dos Sistemas

A utilização de entidades que realizem reconfigurações nos sistemas, ou mesmo uma auto−reconfiguração, indica a necessidade de utilizar e adaptar as características básicas de agentes inteligentes, definidos como entidades cognitivas, ativas e autônomas (monteiro, 2000).

Por serem entidades cognitivas, estes agentes possuem a capacidade de aprender com o meio, através da observação de determinadas características pré−definidas ou até mesmo de outras que o próprio agente julgue serem necessárias. Como entidades ativas, os agentes podem realizar determinadas tarefas necessárias para a configuração dos parâmetros que venham a ser necessários para uma rápida adaptação do(s) sistema(s) a uma nova situação, ou mesmo para o aprimoramento do mesmo. A autonomia do agente é que permite que, sem algo ou alguém que o guie, e através de mecanismos próprios de percepção, sejam tomadas decisões baseadas no conhecimento obtido e voltadas ao cumprimento dos objetivos propostos. É interessante ressaltar que objetivos “propostos” não necessariamente significam objetivos “pré−definidos”. Isto justamente porque os agentes, para atingir o que se propõe como um grande objetivo geral, podem, de forma autônoma, definir e redefinir objetivos específicos. Nesse sentido, o ambiente que propomos possui 3 grandes módulos a serem considerados:

- Módulo Gerente: administra a inicialização dos agentes
- Módulo Agente: que cria e cataloga os diferentes tipos de agentes
- Módulo de Definição de Padrões: cria padrões de reconhecimento que devem ser usados pelos agentes na captura de informações que irão servir como base para a reconfiguração do sistema.

O Módulo de Definição de Padrões utiliza nossa classificação de vulnerabilidades em sistemas, para determinar os padrões de reconhecimento usados pelos agentes durante suas pesquisas por sites Internet, visando buscar novas informações sobre vulnerabilidades existentes em sistemas operacionais e aplicações. Esse conhecimento, normalmente divulgado em formatos diferentes, terão que adotar os padrões sugeridos em, visando torná−los compatíveis com as "visões" de mundo que os agentes terão. A figura 1 abaixo, ilustra o funcionamento do ambiente (monteiro, 2000).

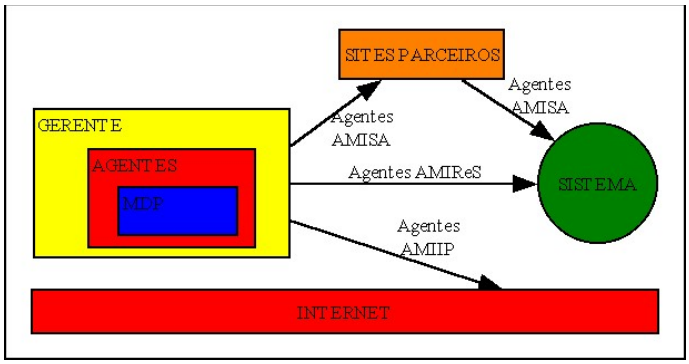

Figura 1:Descrição dos Módulos Funcionais do AAMIGROOVE.

Antes de iniciarmos a descrição funcional do ambiente, precisamos definir 3 tipos de agentes considerados em nosso trabalho e conceituar o que chamamos de sites parceiros.

AMIIP: Agentes Móveis Inteligentes Identificadores de Padrão: São agentes que são disparados de tempos em tempos pelo módulo gerente e que levam em suas classes de dados, informações sobre os padrões a serem reconhecidos nos sites Internet que divulgam problemas de segurança;

AMISA: Agentes Móveis Inteligentes Simuladores de Ataques: São agentes que são disparados pelo módulo gerente, sempre que um agente AMIIP chegar com informações novas. Esses agentes tem em sua classe de dados informações sobre sites Internet Parceiros, os quais serão utilizados por esses agentes para simular o ataque ao site local; O Scom e caracterizado entao como um embriao de um AMISA.

AMIReS: Agentes Móveis Inteligentes Reconfiguradores de Sistema: São agentes que também são disparados pelo módulo gerente, após o resultado de uma simulação feita por um agente AMISA. Esse agente tem como funcionalidade aprender como o sistema local funciona, sua configurações e variáveis, suas características físicas e lógicas. Esse tipo de agente irá conhecer, aos poucos, todas as características operacionais do sistema local, visando alterar o seu estado sempre que possível, com base nas premissas apontadas pelos agentes AMISA.

SITES PARCEIROS: São sites na Internet, localizados fora do seguimento de rede onde se encontra o sistema a ser testado e que estabelecem uma relação de confiança com o referido sistema, ou seja, é necessário que esses sites saibam e concordem em servir de base para as simulações dos ataques remotos ao sistema em questão.

Em casos clássicos de ataques a sistemas, os atacantes utilizam problemas nas implementações dos protocolos de comunicação e/ou das aplicações existentes. Esses problemas poderão ser identificados por agentes autônomos (os AMIIP), gerados por nosso ambiente, através da aquisição do conhecimento de tais vulnerabilidades e da posterior simulação de um ataque que possa explorar tais problemas. Após identificado o problema através da simulação do ataque (usando os AMISA), outro tipo de agente seria então disparado pelo módulo gerente. Esse agente, conceituado como agente reconfigurador (os AMIReS), faria as alterações necessárias no referido sistema, incluindo aí as aplicações de patches em aplicações e protocolos (monteiro, 2000).

3. Scan de Portas

Um *scan* tem como principal objetivo a verificação de quais portas de serviços estão disponíveis local ou remotamente. Para que essa verificação seja feita remotamente é necessário que haja um sistema de comunicação.

Um sistema de comunicação baseia−se num conjunto de componentes de *software* interconectados que oferecem uma *interface* para a comunicação e são regidos por protocolos de comunicação de dados que viabilizam a troca de mensagens ou informações entre processos existentes em máquinas diferentes (arnett, 1997).

Os protocolos são um conjunto de regras formais de comportamento, que regem a comunicação entre os diferentes sistemas interconectados (arnett, 1997).

A comunicação é realizada através de um processo em *front−end* responsável pela solicitação de serviços, recebidos do usuário e outro em b*ack−end* que processa e responde ao pedido do processo cliente (dumas, 1995).

Os serviços para a comunicação inter−processos podem ser de dois tipos: orientados a conexão (confiável) e não orientados a conexão (não confiável ou *datagramas*). Os serviços não confiáveis são aqueles que não possuem dependência entre as operações realizadas. Esses serviços possuem como principais vantagens a rapidez na transmissão dos dados e o baixo *overhead*[*] na rede. Em contrapartida, o usuário não recebe nenhuma informação sobre a operação realizada, os pacotes podem ser perdidos ou chegar fora de seqüência. Os serviços de *datagramas* são utilizados, principalmente, em aplicações de *broadcast* (emissão de mensagem para uma aplicação específica em todas as máquinas da rede com o objetivo de verificar quais estão respondendo) e *chat* (salas de bate−papo).

Nos serviços confiáveis, é estabelecida uma relação de confiança entre as máquinas que desejam conversar e um canal lógico de comunicação é criado entre ambas, através do qual as informações trafegarão. Por exemplo, quando uma discagem a um telefone remoto é completada, foi estabelecido um canal lógico para a transferência de dados do telefone que discou para o telefone discado. A esse tipo de comunicação chamamos fim−a −fim porque um caminho específico é criado entre uma origem e um destino (duarte, 2000).

Os serviços ou aplicações existentes em um servidor são identificados por um número inteiro de 16 bits, chamado de porta, que dá acesso a um serviço. Os números no intervalo de 0 a 1024 foram padronizados para serviços específicos preexistentes. Qualquer número acima de 1204 é utilizado como porta de retorno ou para o desenvolvimento de aplicações de usuários. Por exemplo, a aplicação Pop3, sistema de correio eletrônico da *internet* responsável pelo gerenciamento dos *mails* recebidos, atende na porta 110 (duarte, 2000).

Atrás de todo esse processo de comunicação entre máquinas, existe um tipo de programação conhecida como programação de *sockets*. É através dela que são abertos canais de comunicação.

Socket é uma API (*Application Programing Interface*), padrão em sistemas *UNIX*, para a programação sobre os protocolos de comunicação (dumas, 1995). Essa programação é a base para transmissão e recepção de dados entre processos. A comunicação de processos remotos consiste na criação de um canal virtual de comunicação. (duarte, 2000). Para a comunicação entre processos locais existe um mecanismo chamado *pipe* que garante a entrega confiável dos pacotes, mas não garante que serão entregues seqüencialmente (comer, 1998).

3. Resultados Alcançados

O desenvolvimento do trabalho foi dividido em etapas seguidas seqüencialmente para que o objetivo fosse alcançado com êxito.

A realização de cada etapa trouxe dificuldades e, conseqüentemente, muitos estudos para solucionar os problemas detectados.

A discussão também será seqüêncialmente na mesma ordem em que foram desenvolvidas as atividades facilitando o entendimento.

Para entendermos melhor a codificação do processo de comunicação entre computadores é necessário sabermos como referenciá−los. Há duas maneiras de referenciarmos uma máquina localizada em uma rede baseada em TCP/IP: através de um número IP, por exemplo, 10.241.55.33, que identifique uma única máquina da rede ou um nome de *host*, por exemplo, **www.qualquerservidor.com.br**, associado ao endereço IP.

Quando executamos o SCom deve−se especificar o endereço do servidor a ser verificado, mas, antes que o endereço, fornecido pelo cliente, seja usado, averigua−se se ele é válido através da função *gethostbyname()*, mostrada no trecho de código da figura 2.

```
if(gethostbyname(argv[1])==NULL)
{
        printf("O Host %s eh invalido!!",argv[1]);
        return −1;
}
```

Figura 2:Verificação do endereço de destino

A função *gethostbyname()* retornará um endereço no formato de bytes entendidos pela rede, caso o endereço seja válido, do contrário, retornará *NULL* e a execução é abortada.

Em seguida, é feita a indicação da família de protocolos utilizada pela aplicação, a porta de serviço a qual queremos nos conectar, e a criação do socket. Isto pode ser observado na figura 3.

[*] Overhead e o super congestionamento na rede causado pelo elevado número de dados sendo processado simultaneamente.

```
sin.sin_family=AF_INET;
in.sin_port=htons(port);
sock=socket(AF_INET, SOCK_STREAM, 0);
```

Figura 3:Identificação do protocolo, porta e criação do socket

Observe na figura 3 que para a criação do *socket* são exigidos alguns parâmetros tais como, o tipo de família de protocolos utilizada (*AF_INET*), o tipo de fluxo de conexão (orientado ou não a conexão) e qual o protocolo.

A partir desse momento poderemos solicitar uma conexão ao endereço fornecido, através da função *connect()*, como é mostrado na figura 4.

```
X=connect(sock,(struct sockaddr *)&sin,sizeof(sin));
```

Figura 4:A conexão

Para utilização da função *connect()* é preciso especificar alguns parâmetros, são eles: o descritor de socket, a estrutura *sockaddr* (que contém dados tais como, o endereço de destino e porta de serviço) e o tamanho dessa estrutura.

É nesse momento que o servidor saberá qual o serviço que queremos utilizar e caso ele tenha possibilidades de atender a solicitação então ocorre o *Three Way Handshake*, descrito anteriormente.

A função *connect()* possui retorno −1 quando ocorre erros, por exemplo, quando a máquina está desligada, e qualquer número diferente deste para indicação de sucesso. Essa checagem é mostrada abaixo.

```
if (X!=−1)
{
    if (!find_service(port));
            fprintf(file,"%s\t%d\n",nome,port);
        flag=1;
}
```

Figura 5:Checagem de sucesso ou não da conexão

Na figura 5 é mostrado qual o tratamento dado quando a conexão é realizada com sucesso. Inicialmente é feita uma verificação do nome do serviço indicado pela porta que foi aberta a conexão, através da consulta a uma tabela de correspondência existente no arquivo /etc/services, depois o resultado será impresso no arquivo indicado pela variável *file*.

A narrativa, até o memento, mostrou os principais pontos da codificação de uma aplicação cliente, o SCom. Agora mostraremos alguns trechos de código que diferenciam o cliente e o servidor no tangente a sua estrutura e em seguida as suas funcionalidades.

Um programa servidor difere de um cliente no seguinte: o servidor libera um *socket* ao cliente e associa o *socket* criado para o servidor ao seu endereço utilizando *bind()*, permanece escutando por novas requisições com *listen()*, limitadas pelo segundo parâmetro exigido, e aceita a conexão através da função *accept()*.

```
if(−1==bind(sock,(struct sockaddr *)&me, tame))
        exit(1);
listen(sock,5);
clientsock=accept(sock, (struct sockaddr *)&from, &tame);
```

Figura 6:Principais funções do programa servidor

A figura 6 mostra as principais funções da estrutura de um servidor com seus respectivos parâmetros.

Em seguida, discutiremos sobre as tarefas específicas do cliente e servidor.

O cliente é responsável por enviar o SCom ao servidor quando uma solicitação de localização de serviços ocorrer e mostrar ao usuário a resposta recebida do servidor. O servidor se encarregará de pegar o SCom, executá−lo e enviar a resposta gerada ao cliente.

A primeira tarefa realizada pelo cliente é a transferência do Scom.

```
if ((file=fopen("./SCom.c","r"))!=NULL)
{
        bzero(buff,3000);
        while(!feof(file))
        {
                sprintf(buff,"%s%c",buff,getc(file));
        }
        fclose(file);
        buff[strlen(buff)−1]='@';
        write(s,buff,3000);
}
```

Figura 7:Despachando o SCom

A figura 7 mostra como é o processo de transferência: inicialmente o arquivo que contém a codificação do SCom é aberto com a função *fopen()* no modo de leitura. Caso o arquivo tenha sido aberto com êxito todo o conteúdo do arquivo é passado para uma variável utilizada para transmitir os dados ao servidor. Os dados são transmitidos através da chamada à função *write()* tendo como parâmetros o *socke*t, a *string* que contém os dados e a quantidade de dados a ser enviada.

O servidor, ao receber os dados, cria um arquivo com o mesmo nome, o modo de acesso ao arquivo, no caso é o de escrita, e coloca os dados no mesmo.

```
if((file=fopen("./SCom.c","w"))!=NULL)
{
        bzero(buff,3000);
        read(clientsock,buff,3000);
        i=0;
        while(buff[i]!='@')
        {
            fputc(buff[i],file);
                i++;
        }
        fputc(buff[i−1],file);
        flag=1;
   fclose(file);
}
```

Figura 8:Armazenamento do SCom

Temos na figura acima o código para criação de arquivo. Se o arquivo for criado com sucesso os dados são lidos utilizando a função *read()* que exige os mesmos parâmetros da função *write()*.

```
If (flag)
{
      system("gcc −o SCom SCom.c");
      system("./SCom localhost pesquisa");
      if (mountHTML())
      {
         sendResult(clientsock);
      }
}
```

Figura 9:Execução do SCom

É mostrado, na figura acima, que são feitas duas chamadas ao sistema através da função *system()*. Na primeira são passados como parâmetros os comandos necessários para geração de um código executável (primeira chamada a *system()*) e em seguida sua execução fornecendo o *host* e o arquivo de retorno chamado "pesquisa".

A partir do resultado gerado é montado um arquivo *html*, com a chamada a *mountHTML()*, que apresentará ao cliente o resultado da consulta.

O servidor envia o arquivo *html* ao cliente e ele o mostra ao usuário da aplicação chamando o *browser netscape*.

Em fim, quando todo o fluxo de transferência de dados cessar os *sockets* abertos devem ser fechados para que sejam utilizados por outras aplicações.

```
Close(sock);
```

Figura 10: Fechando o socket

O fechamento do *socket*, como é mostrado na figura 10, é feito através da função *close()*.

Quando o SCom é executado ele gera a saída de sua pesquisa que é apresentada no browser como mostra a figura abaixo.

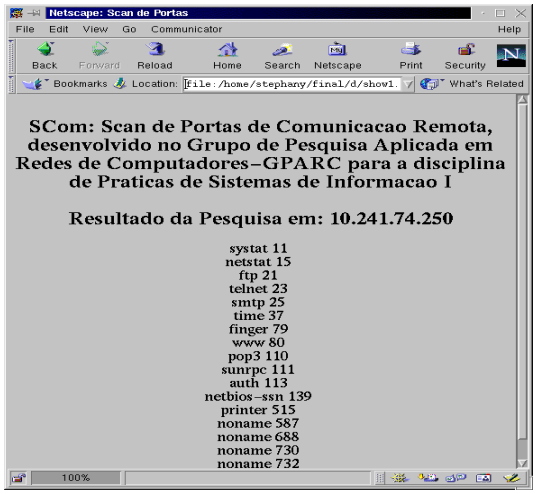

Figura 11:Apresentação dos Resultados

O problema encontrado nessa etapa foi detectado no momento da execução no ponto onde é feita a primeira chamada à função *system()*. O compilador gcc, aparentemente, não reconhecia o fim do arquivo do SCom e por isso, não gerava o executável. Resolvemos o problema de maneira simples: utilizando nosso próprio limitador de arquivo (@, arroba). O símbolo arroba é acrescento ao final do arquivo quando é enviado e quando recebido é substituído pelo EOF (*end of file*) conhecido no ambiente de programação C.

4. Conclusão

Atualmente não há como falar em *sites Internet* sem mencionarmos segurança, mas, o questionamento sobre o grau de segurança que os desenvolvedores podem alcançar na construção de suas aplicações, geralmente, inexiste devido a ética de alguns profissionais que estudam constantemente para criar técnicas de burlar os sistemas para fins ilícitos. Portanto, é preciso, que os profissionais desenvolvam uma cultura de realizar seu trabalho de forma ética, buscando a segurança de suas aplicações, tanto locais quanto remotas e ainda, promovendo juntos com usuários, a discussão sobre o assunto, com o objetivo de descobrirmos como lutar contra invasores e como evitar que as técnicas de ataque cresçam na mesma proporção que os avanços computacionais. Precisamos mostrar−lhes que nós somos o diferencial nesse mundo cibernético.

5. Referências Bibliográficas

(arnett, 1997) ARNETT, Mattew Flint, DULANEY, Emmett, et al. **Desvendando o TCP/IP**. Rio de Janeiro: Campus, 1997. 543p.

(comer, 1998) COMMER, Douglas E. **Interligação em rede com TCP/IP**. Vol. I. Rio de Janeiro: Campus, 1998. 672p.

(duarte, 2000) DUARTE, Fabiano Caixeta, MARTINS, Stéphany Moraes. **API de sockets: os segredos da comunicação cliente/servidor**. Artigo publicado e defendido no II Encontro de Estudantes de Informática do Estado do Tocantins. 2000.

(dumas, 1995) DUMAS, Arthur. **Programando Winsock**. Rio de Janeiro: Axcel Books, 1995. 358p.

(monteiro, 2000) MONTEIRO, Claudio de Castro e FAGUNDES, Fabiano. **AAMIGROOVE: Um Framework para a Construçao de Agentes Móveis e Inteligentes para a Reconfiguraçao de Sistemas com Falhas de Segurança.** Computaçao Reconfiguravél: Experiências e Perspectivas. Rio de Janeiro: Brasport.