Banner Grabber em C utilizando conexões TCP

Renan Domingos Merlin Greca
Universidade Federal do Paraná
renangreca@gmail.com

José Robyson Aggio Molinari
Universidade Federal do Paraná
aggio13@hotmail.com

1. INTRODUÇÃO

Este relatório refere-se ao trabalho 1 da disciplina de Introdução a Segurança Computacional, cursada na Universidade Federal do Paraná no segundo semestre de 2016. O objetivo do trabalho é criar um programa em C que estabelece conexões com uma determinada sequência de endereços IP e portas para receber o *banner* dos serviços rodando nessas portas. Chamamos esse procedimento de *banner grabbing*. Tal operação é interessante para agentes maliciosos buscando possíveis vulnerabilidades em servidores remotos, pois eles saberão com precisão quais IPs e portas podem apresentar um ponto de entrada.

Os desafios para o desenvolvimento deste projeto incluíram: (1) a interpretação da entrada; (2) o estabelecimento de uma conexão TCP e o recebimento de um *banner*; (3) a iteração sobre toda a sequência de IPs/portas; e (4) a otimização da execução ao evitar conexões inválidas.

1.1 Banners

Ao estabelecer uma conexão com um servidor remoto em determinada porta, os primeiros bytes recebidos podem gerar uma *string* chamada *banner*. O *banner* contém detalhes (por exemplo, a versão) do serviço disponível na porta. No entanto, nem todo serviço possui um banner, então, mesmo que exista algo disponível em uma porta, pode ser que uma *string* vazia seja encontrada.

Suponha um servidor qualquer rodando um serviço de SSH na porta 22 (a porta padrão do SSH). Um exemplo de *banner* que pode ser encontrado nessa porta é:

```
SSH-1.99-OpenSSH_2.9p2
```

No caso, o *banner* acima indica que o servidor está rodando o serviço OpenSSH, versão 2.9, que utiliza a versão 1.99 do protocolo SSH. Com essas informações em mão, um atacante pode buscar, com relativa facilidade, vulnerabilidades da versão 2.9 do OpenSSH ou da versão 1.99 do SSH.

2. DESENVOLVIMENTO

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

Conforme mencionado na introdução, o desenvolvimento do projeto contou com quatro desafios principais. Nesta seção, esses desafios são detalhados e suas soluções são explicadas.

2.1 Entrada

O primeiro desafio, e o menos relevante para o conceito do projeto, foi interpretar a entrada do programa. O programa desenvolvido chama-se `recon` e aceita entrada no seguinte formato:

```
recon <ip> [port]
```

Tanto `ip` quanto `port` podem ser um valor único ou uma faixa separada por `-`. Adicionalmente, o parâmetro `port` é opcional; a ausência dele causará o uso de todas as portas, ou seja, de 1 a 65535. Portanto, alguns exemplos de chamadas do programa são:

```
recon 192.168.1.1 1-100
recon 192.168.1.1-10
recon 192.168.1.1-10 1-100
```

Dadas entradas nesses formatos, foi necessário utilizar as funções `strsep` e `strcat`, presentes na biblioteca `<string.h>`, para separar e concatenar *strings*. Adicionalmente, foi necessária a função `atoi` da biblioteca padrão, `<stdlib.h>`.

Programa 1 Segmento do programa que separa o endereço IP; uma operação similar é realizada para as portas.

```
while ((token = strsep(&ipstring, "-")) !=
    NULL) {
    iprange[i] = token;
    i++;
}
while (((token = strsep(&iprange[0], "."))
    != NULL) && i<3 ) {
    strcat(subnet, token);
    strcat(subnet, ".");
    i++;
}
int ip_min = atoi(token);
int ip_max;
if (iprange[1] != NULL) {
    ip_max = atoi(iprange[1]);
}
```

O Programa 1 mostra os laços que utilizam a função `strsep` para separar a primeira parte da entrada, a faixa de en-

dereços IP, usando os caracteres `-` e `..` A subrede do endereço é formada ao concatenar apenas as três primeiras partes do endereço IP; o quarto octeto será adicionado apenas durante a iteração sobre a faixa de IPs.

2.2 Conexão TCP

Para estabelecer a conexão TCP, primeiramente foi criado um programa separado que recebe apenas um endereço IP e uma porta e adquire o *banner*. Tutoriais para conexão TCP em C podem ser facilmente encontrados na Internet, então simplesmente foi utilizado um deles como base para o programa. O programa abre um *socket* utilizando os protocolos TCP e IP, estabelece uma conexão (por meio da função `connect`) com o servidor remoto usando o conjunto de endereço IP e porta. Por fim, ele utiliza a função `read` para obter uma sequência de 255 bytes, que, ao ser interpretado como uma *string*, é o *banner*.

Algumas bibliotecas do C são necessárias para sua execução: `<sys/types.h>`, `<sys/socket.h>`, `<netinet/in.h>`, `<netdb.h>` e `<unistd.h>`.

O Programa 2 mostra as principais operações para estabelecer essa conexão. No código completo há tratamento de erros e várias operações secundárias necessárias. A constante `BUFFER_SIZE` é 256.

Programa 2 As operações principais, fora de contexto, para receber os dados com TCP.

```
// Abre o socket
sockfd = socket(AF_INET, SOCK_STREAM, 0);
// Monta as structs nos formatos corretos
    para a função connect
server = gethostbyname(ip);
...
serv_addr.sin_port = htons(port);
// Tenta estabelecer conexão
connect(sockfd,(struct sockaddr *)&
    serv_addr,sizeof(serv_addr));
// Lê os primeiros bytes recebidos
n = read(sockfd,buffer,BUFFER_SIZE-1);
```

2.3 Iteração sobre vários endereços e portas

Programa 3 Laços para a iteração sobre todos os IPs e portas, mais a função de fechamento de socket que mostrou-se necessária.

```
for (int ipaddr=ip_min; ipaddr<=ip_max;
    ipaddr++) {
        for (int port=port_min; port<=
            port_max; port++) {
                /* conexão e banner grab */
                close(sockfd);
        }
}
```

A princípio, a iteração sobre diversos endereços IP e suas portas é trivial, quando consideradas as soluções dos desafios acima. Com as faixas de IPs e portas em mãos, basta utilizar dois laços aninhados e executar a conexão para cada combinação possível. No entanto, ao implementar essa solução, foi descoberto que as conexões deixavam de ocorrer após exatamente 254 conexões, ou tentativas, corretas.

A causa do problema era que, no programa que estabelecia uma conexão TCP isolada, um *socket* era aberto mas não era fechado, e essa falha foi replicada ao adaptar o programa para a forma iterativa. Portanto, foi necessário adicionar a função `close` usando o descritor do *socket* como parâmetro para fechá-lo após cada iteração. Desta forma, foi possível testar centenas de milhares de portas sequencialmente sem problemas.

Os laços são mostrados no Programa 3, mostrando a finalidade das variáveis `ip_min` e `ip_max` declaradas no Programa 1 e suas equivalentes para a faixa de portas.

2.4 Otimização das conexões

Durante os testes, foi observado que, sob determinadas circunstâncias, uma conexão era estabelecida com o servidor, mas passava alguns segundos tentando obter um *banner* sem sucesso. Aguardar alguns segundos é aceitável ao verificar uma sequência curta de portas, mas é inviável em uma sequência de milhares. Portanto, foi utilizada a biblioteca `<poll.h>`, que permite o uso de *timeout* para que, se a obtenção do *banner* demorar mais do que 0,1 segundo, assume-se que o *banner* é vazio e o programa pode continuar para a próxima porta.

O Programa 4 mostra como foi utilizada a biblioteca `poll`. A função `read`, vista no Programa 2, encontra-se no caso *default*.

Programa 4 O `switch-case` acionado pela biblioteca `poll` que permite um *timeout* nas conexões.

```
fd.fd = sockfd;
fd.events = POLLIN;
ret = poll(&fd, 1, 100);
      /* 0.1 second timeout */
switch (ret) {
    case -1:
        // Erro
        break;
    case 0:
        // Timeout
        break;
    default:
        // Conexão; pegar dados
        break;
}
```

Adicionalmente, quando algum dos endereços IP estava completamente inativo (ou seja, não havia servidor rodando com aquele endereço), o programa levava alguns segundos para tentar estabelecer uma conexão com cada uma das possíveis portas. Assim como acima, isso era admissível. Para evitar isso, foi utilizada a biblioteca `<errno.h>`, que permite uma identificação precisa do motivo pelo qual uma conexão falhou.

Foi descoberto que, quando um endereço IP está indisponível, a função `connect` falha com o erro `51`, então o restante das portas daquele IP podem ser ignoradas. Outros erros geralmente indicam a indisponibilidade apenas da porta tentada.

Programa 5 Captação de erro de conexão; erro `51` indica que todo o endereço IP está inativo.

```
if (connect(sockfd,(struct sockaddr *)&
    serv_addr,sizeof(serv_addr)) < 0) {
    close(sockfd);

    if (errno == 51) {
        // Erro 51 indica endereço IP
            indisponível
        port = port_max;
    } // Outros erros podem ser porta
        inativa, timeout da conexão, etc.
    continue;
}
```

3. RESULTADOS

Mesmo com as otimizações acima, a execução do programa pode ser consideravelmente demorada. Como solicitado no enunciado do projeto, o teste final foi feito usando a seguinte entrada:

```
recon 200.238.144.20-29
```

Ou seja, uma faixa de 10 endereços IP e todas as suas portas.

Como pode ser visto abaixo, o programa descobriu uma série de informações sobre os servidores rodando em IPs finalizados em 27, 28 e 29. No entanto, como os endereços IPs finalizados com os números entre 20 e 26 estavam inativos, eles foram pulados após a verificação de apenas uma porta ao receber um erro `51`.

Log 1 log do programa

```
Varredura iniciada em Thu Sep  1 19:25:24
    2016
IP: 200.238.144.20-29
Portas: (null)
---
200.238.144.27      21
200.238.144.27      22        SSH-1.99-
    OpenSSH_2.9p2
200.238.144.27      111
200.238.144.27      32768
200.238.144.28      22        SSH-2.0-OpenSSH_6
    .6.1p1 Ubuntu-2ubuntu2.6
200.238.144.28      5001      ¬í
200.238.144.28      8080
200.238.144.29      21        220 Welcome to
    the ftp service
200.238.144.29      22        SSH-2.0-OpenSSH_5
    .9p1 Debian-5ubuntu1.9
200.238.144.29      42
200.238.144.29      80
200.238.144.29      443
200.238.144.29      1433
200.238.144.29      3306      4
200.238.144.29      5060
200.238.144.29      5061
```

Como pode ser visto, os serviços de FTP e SSH são os mais fáceis de detectar. Outros serviços, como aqueles rodando na porta 5001 do IP 28 e na porta 3306 do IP 29, podem ter oferecido bytes que deveriam ser interpretados de forma diferente, portanto não fazem sentido quando vistos como uma *string*.

Os serviços restantes têm *banners* vazios. Isso pode ocorrer porque o serviço instanciado naquela porta utiliza um protocolo (como, por exemplo, SMTP) que utiliza pacotes mais elaborados, portanto o *banner* pode não estar nos primeiros bytes. Outra possibilidade é que seja um serviço de HTTP, que retorna pacotes apenas após receber alguma solicitação e, portanto, ocorre um *timeout* na conexão.

De qualquer maneira, essa execução pode resumir as centenas de milhares de portas inicialmente recebidas para apenas dezesseis. Com essa lista de IPs e portas em mão, é possível construir um algoritmo que tente efetuar conexões de outros tipos com cada uma delas, ou mesmo tentar acessá-las manualmente.