Windows网络编程之端口扫描实验

原创

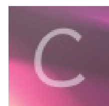
tianhaobing 于 2017-04-08 22:38:42 发布 3272 收藏 31

分类专栏： windows网络编程 文章标签： 网络编程 windows 计算机 工作 开放

版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。

本文链接：https://blog.csdn.net/tianhaobing/article/details/69788318

版权


windows网络编程 专栏收录该内容

4 篇文章 0 订阅

订阅专栏

实验三 端口扫描实验

1 实验类型

验证型实验

2 实验目的

1.了解端口扫描的基本概念和工作原理；

3 背景知识

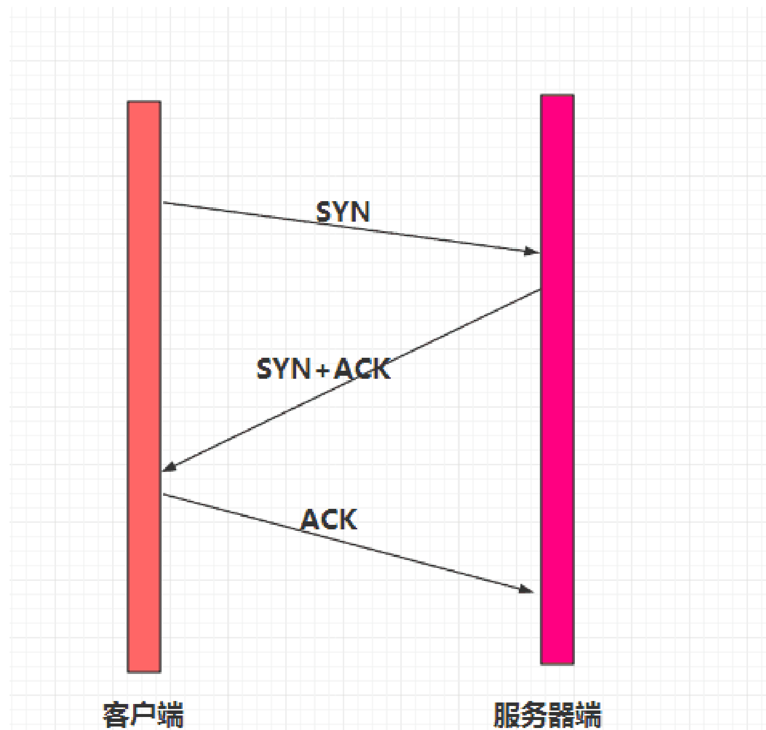
1. 端口扫描原理

在“计算机网络”课程中，我们知道完成一次TCP 连接需要完成三次握手才能建立。端
口扫描正是利用了这个原理，通过假冒正常的连接过程，依次向目标主机的各个端口发送连
接请求，并根据目标主机的应答情况判断目标主机端口的开放情况，从而分析并对一些重要
端口实施攻击。

端口扫描的方式有两种，一种称为完整扫描，一次连接过程如下图所示：

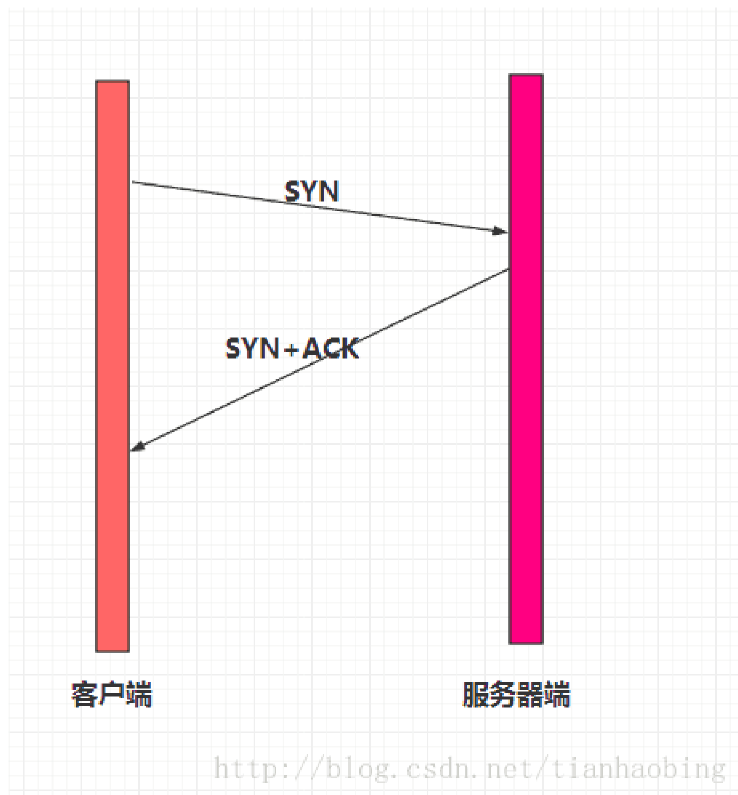
完整扫描连接过程示意图

另一种扫描方式称为半开扫描，出于欺骗的目的，半开扫描在收到服务端的应答信号（SYN+ACK）后，不再发送响应信号（ACK）。一次连接过程如下图所示：



http://blog.csdn.net/tianhaobing

半开扫描连接过程示意图



4 实验内容

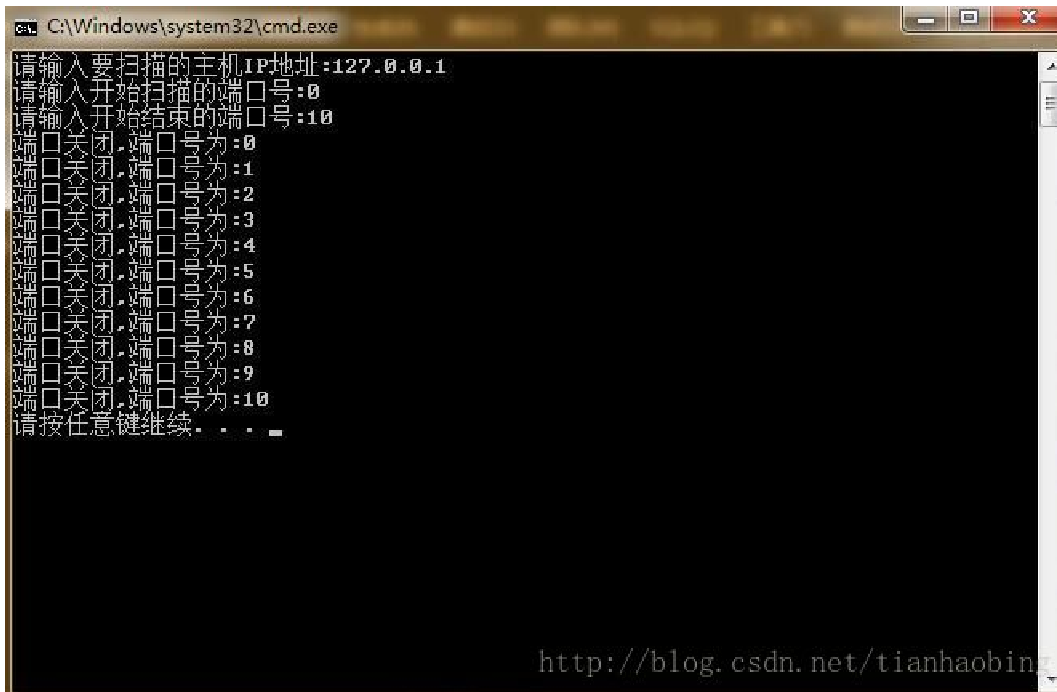
1、编写一个利用全连接的端口扫描程序，能显示目标主机的端口开放情况。要求能在命令行输入要扫描的目标主机和端口范围。比如：scan ... nnnn-mmmm。

代码如下:

```
#include<stdio.h>
#include<WINSOCK2.H>
#include<string.h>
#include<iphlpapi.h>
#pragma comment(lib,"WS2_32.lib")
int main(){
    WSADATA WSAData;
    sockaddr_in addr; //用来创建socket的结构体
    char IpAddress[100]; //待扫描的主机IP地址
    char startPort[10],endPort[10]; //开始和结束的端口号
    printf("请输入要扫描的主机IP地址:");
    gets(IpAddress);
    printf("请输入开始扫描的端口号:");
    gets(startPort);
    printf("请输入开始结束的端口号:");
    gets(endPort);

    if(WSAStartup(MAKEWORD(2,2),&WSAData)!=0){ //初始化Winsock2.2
          printf("无法完成初始化...");
          return 0;
    }
    addr.sin_family=AF_INET;
    addr.sin_addr.S_un.S_addr=inet_addr(IpAddress); //将点分十进制的IP地址转换为网络字节序
    for(int i=atoi(startPort);i<=atoi(endPort);i++){ //atoi函数将字符串型端口转换为int型的值
        SOCKET s=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);//和每一个端口建立相关的socket,用于TCP连接
        if(s==INVALID_SOCKET){  //若端口建立失败，则程序结束
            printf("建立Socket失败,错误代码:%d",WSAGetLastError());
            return 0;
        }
        addr.sin_port=(htons(i));//将端口设置为遍历的每一个端口，用于TCP连接
        if((connect(s,(sockaddr*)&addr,sizeof(sockaddr)))==-1){
               printf("端口关闭,端口号为:%d\n",i);
        }else{
                      printf("端口开启,端口号为:%d\n",i);
        }
        closesocket(s); //得到端口是否开启后关闭socket
    }
    if(WSACleanup()==SOCKET_ERROR){
           printf("WSACleanUp出错!!!");
    }  //做一些清除工作
    system("pause");
    return 0;
}
```

实验结果:



以上代码效率非常低,提高效率可以通过多开几个线程来同时扫描。多线程下的扫描这里就不写了,下面把相关的函数放在下面,可以字节动手写下:

在windows下，我们可以调用SDK win32 api来编写多线程的程序，下面就此简单的讲一下：
创建线程的函数
代码如下:
HANDLE CreateThread(
LPSECURITY_ATTRIBUTES lpThreadAttributes, // SD
SIZE_T dwStackSize, // initial stack size
LPTHREAD_START_ROUTINE lpStartAddress, // thread function
LPVOID lpParameter, // thread argument
DWORD dwCreationFlags, // creation option
LPDWORD lpThreadId // thread identifier
);
在这里我们只用到了第三个和第四个参数，第三个参数传递了一个函数的地址，也是我们要指定的新的线程，第四个参数是传给新线程的参数指针。eg1:

复制代码 代码如下:

```
#include <iostream>
#include <windows.h>
using namespace std;
DWORD WINAPI Fun(LPVOID lpParamter)
{
      while(1) { cout<<"Fun display!"<<endl; }
}
int main()
{
    HANDLE hThread = CreateThread(NULL, 0, Fun, NULL, 0, NULL);
    CloseHandle(hThread);
    while(1) { cout<<"main display!"<<endl;  }
    return 0;
}
```

我们可以看到主线程（main函数）和我们自己的线程（Fun函数）是随机地交替执行的，但是两个线程输出太快，使我们很难看清楚，我们可以使用函数

VOID Sleep(

DWORD dwMilliseconds // sleep time

);

来暂停线程的执行，dwMilliseconds表示千分之一秒，所以Sleep(1000);表示暂停1秒

eg2:

复制代码 代码如下:

```
#include <iostream>
#include <windows.h>
using namespace std;
DWORD WINAPI Fun(LPVOID lpParamter)
{
      while(1) { cout<<"Fun display!"<<endl; Sleep(1000);}
}
int main()
{
      HANDLE hThread = CreateThread(NULL, 0, Fun, NULL, 0, NULL);
      CloseHandle(hThread);
      while(1) { cout<<"main display!"<<endl;  Sleep(2000);}
      return 0;
}
```

执行上述代码，这次我们可以清楚地看到在屏幕上交错地输出Fun display!和main display!，我们发现这两个函数确实是并发运行的，细心的读者可能会发现我们的程序是每当Fun函数和main函数输出内容后就会输出换行，但是我们看到的确是有的时候程序输出换行了，有的时候确没有输出换行，甚至有的时候是输出两个换行。这是怎么回事？下面我们把程序改一下看看：

eg3:

代码如下:

```
#include <iostream>
#include <windows.h>
using namespace std;
DWORD WINAPI Fun(LPVOID lpParamter)
{
      while(1) { cout<<"Fun display!\n"; Sleep(1000);}
}
int main()
{
      HANDLE hThread = CreateThread(NULL, 0, Fun, NULL, 0, NULL);
      CloseHandle(hThread);
      while(1) { cout<<"main display!\n";  Sleep(2000);}
      return 0;
}
```

我们再次运行这个程序，我们发现这时候正如我们预期的，正确地输出了我们想要输出的内容并且格式也是正确的。下面我就来讲一下此前我们的程序为什么没有正确的运行。多线程的程序时并发地运行的，多个线程之间如果公用了一些资源的话，我们并不能保证这些资源都能正确地被利用，因为这个时候资源并不是独占的，举个例子吧：

eg4:
加入有一个资源 int a = 3
有一个线程函数 selfAdd() 该函数是使 a += a;
又有一个线程函数 selfSub() 该函数是使a -= a;
我们假设上面两个线程正在并发欲行，如果selfAdd在执行的时候，我们的目的是想让a编程6，但此时selfSub得到了运行的机会，所以a变成了0，等到selfAdd的到执行的机会后，a += a ，但是此时a确是0，并没有如我们所预期的那样的到6，我们回到前面EG2，在这里，我们可以把屏幕看成是一个资源，这个资源被两个线程所共用，加入当Fun函数输出了Fun display!后，将要输出endl（也就是清空缓冲区并换行，在这里我们可以不用理解什么事缓冲区），但此时main函数确得到了运行的机会，此时Fun函数还没有来得及输出换行就把CPU让给了main函数，而这时main函数就直接在Fun display!后输出main display!，至于为什么有的时候程序会连续输出两个换行，读者可以采用同样的分析方法来分析，在这里我就不多讲了，留给读者自己思考了。
那么为什么我们把eg2改成eg3就可以正确的运行呢？原因在于，多个线程虽然是并发运行的，但是有一些操作是必须一气呵成的，不允许打断的，所以我们看到eg2和eg3的运行结果是不一样的。
那么，是不是eg2的代码我们就不可以让它正确的运行呢？答案当然是否，下面我就来讲一下怎样才能让eg2的代码可以正确运行。这涉及到多线程的同步问题。对于一个资源被多个线程共用会导致程序的混乱，我们的解决方法是只允许一个线程拥有对共享资源的独占，这样就能够解决上面的问题了。
代码如下:
HANDLE CreateMutex(
LPSECURITY_ATTRIBUTES lpMutexAttributes, // SD
BOOL bInitialOwner, // initial owner
LPCTSTR lpName // object name
);

该函数用于创造一个独占资源，第一个参数我们没有使用，可以设为NULL，第二个参数指定该资源初始是否归属创建它的进程，第三个参数指定资源的名称。
代码如下:

HANDLE hMutex = CreateMutex(NULL,TRUE,”screen”);这条语句创造了一个名为screen并且归属于创建它的进程的资源
代码如下:
BOOL ReleaseMutex(
HANDLE hMutex // handle to mutex
);
该函数用于释放一个独占资源，进程一旦释放该资源，该资源就不再属于它了，如果还要用到，需要重新申请得到该资源。申请资源的函数如下
代码如下:
DWORD WaitForSingleObject(
HANDLE hHandle, // handle to object
DWORD dwMilliseconds // time-out interval
);

第一个参数指定所申请的资源的句柄，第二个参数一般指定为INFINITE，表示如果没有申请到资源就一直等待该资源，如果指定为0，表示一旦得不到资源就返回，也可以具体地指定等待多久才返回，单位是千分之一秒。好了，该到我们来解决eg2的问题的时候了，我们可以把eg2做一些修改，如下
eg5：
代码如下:

```
#include <iostream>
#include <windows.h>
using namespace std;
HANDLE hMutex;
DWORD WINAPI Fun(LPVOID lpParamter)
{
       while(1) {
                 WaitForSingleObject(hMutex, INFINITE);
                 cout<<"Fun display!"<<endl;
                 Sleep(1000);
                 ReleaseMutex(hMutex);
        }
  }
int main()
{
      HANDLE hThread = CreateThread(NULL, 0, Fun, NULL, 0, NULL);
      hMutex = CreateMutex(NULL, FALSE, "screen");
      CloseHandle(hThread);
      while(1) {
               WaitForSingleObject(hMutex, INFINITE);
               cout<<"main display!"<<endl;
               Sleep(2000);
               ReleaseMutex(hMutex);
      }
      return 0;
}
```