윈도우 플랫폼에서 네트워크 방화벽 개발

작성일: 2004년 11월 18일
작성자: 김기홍 (와우해커 & 세인트 시큐리티 대표이사)
www.wowhacker.com
www.stsc.co.kr

- 들어가면서

리눅스 플랫폼에서의 방화벽 개발은 많은 정보와 공개되어 있는 수 많은 소스 코드들을 참고 하면 쉽고 빠르게 성능이 좋은 방화벽을 개발 할 수 있다. 하지만 윈도우 플랫폼에서 방화벽을 개발하려고 한다면 그 개발의 한계를 쉽게 느낄 수 있을 것이다. 많은 부분 공개되어 있는 코드들이 없을 뿐더러 NDIS, TDI 등의 커널 레벨에서 방화벽을 개발하려다 보니 어려운 부분이 한, 두 가지가 있는 것이 아니기 때문이다. 하지만 요즘 개인용 방화벽에 대한 요구가 많아지고 있는 시점이고 그 방화벽 개발의 방향이 윈도우 2000 이상이라면 쉽게 개발을 할 수 있을 것이다.

- 환경

Windows 2000 DDK를 살펴 보게 되면 마이크로 소프트는 새로운 타입의 네트워크 드라이버를 포함하고 있는데 바로 Filter Hook Driver 이다. 이 드라이버를 이용하게 되면 보내고 받는 각종 패킷(TCP, UDP, ICMP) 등을 제어를 할 수 있게 되는 것이다. 반드시 모든 개발은 Windows 2000 이상에서 해야 하며 그 이하 버전에서 사용되는 방화벽 개발은 이 문서에서 열외로 하도록 하겠다.

- Filter Hook Driver?

글을 시작하면서 잠깐 언급을 했던 Filter Hook Driver 가 무엇인지 궁금해 하는 사람이 많을 것이다. Filter Hook Driver 라는 것은 Windows 2000 DDK 에 소개가 되어 있다. 따지고 보면 이 Filter Hook Driver 라는 것은 완전 새로운 네트워크 드라이버를 말하는 것이 아니다. 기존에 운영 체제가 가지고 있던 IP Filter Driver 의 하나의 확장 형태로 볼 수 있다. (이러한 확장 형태는 Windows 2000 이상에서만 가지고 있다.) Filter Hook Driver 는 NDIS 와 같은 네트워크 드라이버를 직접적으로 개발하는 것을 말하는 것은 아니다. 단지 그냥 커널 레벨에서 작동 하는 커널 레벨의 드라이버(커널 레벨에서 작동이 되는 일종의 프로그램) 를 말하는 것이다. Filter Hook Driver 는 Callback 함수를 가지고 있고 기존의 IP Filter Driver 의 함수에 Filter Hook Driver 의 함수를 등록을 시켜서 IP Filter Driver 내부에 있는 함수가 호출 될 때 우리가 지정한 Filter Hook Driver 의 Call Back 함수로 호출이 가능하게 하는 것이다. 그럼 기존의 IP Filter Driver 에서 send 와 receive 함수와 관계된 부분을 우리가 새롭게 만들 Filter Hook Driver 에 Callback 함수로 등록을 하게 되면 적당한 정보와 함께 비교를 하면서 차단을 할지 허용을 할지를 결정하게 되는 것이다. 위의 과정을 다시 한번 간단하게 정리를 해볼 수 있도록 하자.

1. 커널 레벨의 Filter Hook Driver를 만든다.
2. 필터 함수를 등록을 하기 위해서는 IP Filter Driver 의 포인터를 얻을 수 있도록 한

다.

3. IP Filter Driver 의 포인터를 알았으면 우리는 특정한 IRP를 보내면서 필터 함수를 설치하여 사용할 수 있다. 그렇게 되면 IP Filter Driver를 통과하는 모든 데이터들이 우리의 필터 함수에도 통과를 하게 된다.

4. 여기서 특정 정보와 비교를 해서 필터링을 할 수 있도록 한다.

5. 필터링 과정이 끝난 후에 원상태로 풀어주기 위해서는 NULL Pointer 의 또 다른 함수를 등록을 시키거나 바꾸게 되면 된다.

그렇다면 위와 같이 Filter Hook Driver 의 간단한 작동 원리를 알았다면 실질적으로 프로그램 소스 코드를 보면서 이해를 할 수 있도록 하자.

- 커널 모드 드라이버 만들기

Filter Hook Driver 는 앞에서도 충분히 언급을 했듯 Kernel Mode Driver 이다. 따라서 Filter Hook Driver를 사용하려면 Kernel Mode Driver를 만들어야 하는 것은 분명한 사실이다. 그렇다면 Kernel Mode 의 드라이버는 어떻게 만드는 것일까? 커널 모드의 드라이버를 제작하는 방법은 다른 책이나 참고 문헌에 많이 실려 있으니 그것을 참조 하도록 하고 여기서는 방화벽을 만들기 위해 최소한을 필요한 정보만 알면서 넘어 갈 수 있도록 하자.

Filter Hook Driver 는 아래와 같이 기본적인 Kernel Mode Driver 의 구조를 가진다.

1. 우리가 만들 커널 모드의 드라이버의 구조는 기본적으로 가져야 할 루틴들(Dispatch, load, unload, create 등)을 포함하고 있고 어플리케이션과 통신을 하기 위한 심볼릭 링크들을 가지고 있어야 한다.

2. 기본적인 루틴은 각종 IRP를 관리 할 수 있어야 한다. 즉, 각종 IOCTL 들은 외부의 프로그램과 통신을 하기 위한 일종의 외부 함수들로써 커널 모드 드라이버 내부에 미리 선언이 되어 있어야 한다. 또한 선언된 내용은 어플리케이션과도 연동을 하여 같이 작동이 될 수 있도록 한다.

3. 기본적으로 필터링 할 수 있는 함수를 커널 모드 드라이버에 등록을 할 수 있도록 한다.

그럼 위와 같은 대강의 구조의 구성을 알았다면 다음 소스 코드를 보면서 주석을 달아가면서 이해를 할 수 있도록 하자.

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
                     IN PUNICODE_STRING RegistryPath)
{
    // 디바이스 장치를 만든다
    RtlInitUnicodeString(&deviceNameUnicodeString, NT_DEVICE_NAME);

    ntStatus = IoCreateDevice(DriverObject,
                0,
                &deviceNameUnicodeString,
                FILE_DEVICE_DRVFLTIP,
                0,
                FALSE,
```

```
                &deviceObject);

if ( NT_SUCCESS(ntStatus) )
{
        // Win32 어플리케이션과 통신하기 위해서 심볼릭 링크를 만든다.
        // 이 커널 모드의 드라이버를 위해서 만든다.
        RtlInitUnicodeString(&deviceLinkUnicodeString, DOS_DEVICE_NAME);

        ntStatus = IoCreateSymbolicLink(&deviceLinkUnicodeString,
                                        &deviceNameUnicodeString);

        // Dispatch 포인터를 만든다. 디바이스 드라이버의 Create, Close, Control 등을 관리

        DriverObject->MajorFunction[IRP_MJ_CREATE]          =
        DriverObject->MajorFunction[IRP_MJ_CLOSE]           =
        DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DrvDispatch;
        DriverObject->DriverUnload                          = DrvUnload;
    }

    if ( !NT_SUCCESS(ntStatus) )
    {
        // 초기화를 실패 했을 경우 드라이버를 다시 언로드 시킨다.

        DrvUnload(DriverObject);
    }

    return ntStatus;
}

NTSTATUS DrvDispatch(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{

    // 위에서 등록한 디스패치 루틴에 따라 작동이 될 수 있도록 한다.

    switch (irpStack->MajorFunction)
    {
    case IRP_MJ_CREATE:

        break;

    case IRP_MJ_CLOSE:

        break;

    case IRP_MJ_DEVICE_CONTROL:

        ioControlCode = irpStack->Parameters.DeviceIoControl.IoControlCode;

        switch (ioControlCode)
        {
        // ioctl code를 필터링 하기 시작한다.
        case START_IP_HOOK:
        {
            SetFilterFunction(cbFilterFunction); // 필터링 함수 등록

            break;
        }

        // ioctl code 필터링을 NULL 포인터의 함수를 등록하면서 중지한다.
        // NULL 포인터 함수를 등록하게 되면 해당 함수의 내용이 없기 때문에 필터링을
```

```
        // 중지한 것과 마찬가지로 처리가 되기 때문이다.

        case STOP_IP_HOOK:
        {
            SetFilterFunction(NULL); // NULL 포인터 함수 등록

            break;
        }

        // Win32 어플리케이션으로부터 보내진 ioctl code - Filter 룰을 추가한다.
        case ADD_FILTER:
        {
            if(inputBufferLength == sizeof(IPFilter))
            {
                IPFilter *nf;

                nf = (IPFilter *)ioBuffer;

                AddFilterToList(nf);
            }

            break;
        }

        // Win32 어플리케이션으로부터 보내진 ioctl code - Filter 룰을 지운다.
        case CLEAR_FILTER:
        {
            ClearFilterList();

            break;
        }

        default:
            Irp->IoStatus.Status = STATUS_INVALID_PARAMETER;

            break;
    }

        break;
    }


    ntStatus = Irp->IoStatus.Status;

    IoCompleteRequest(Irp, IO_NO_INCREMENT);

    return ntStatus;
}


VOID DrvUnload(IN PDRIVER_OBJECT DriverObject)
{
    UNICODE_STRING deviceLinkUnicodeString;

    // 드라이버 언로딩 함수
    // NULL 포인터 함수 등록
    SetFilterFunction(NULL);

    // 필터 리스트 초기화
    ClearFilterList();
```

```
    // win32 어플리케이션과 통신하기 위한 심볼릭 링크 삭제
    RtlInitUnicodeString(&deviceLinkUnicodeString, DOS_DEVICE_NAME);
    IoDeleteSymbolicLink(&deviceLinkUnicodeString);

    // 디바이스 객체 삭제
    IoDeleteDevice(DriverObject->DeviceObject);
}
```

- 필터링 함수 등록하기

앞에서 살펴본 소스 코드를 보게 되면 SetFilterFunction 이라는 함수를 호출 하도록 되어 있다. 이 함수는 IP Filter Driver 에 우리가 원하는 Callback 함수로 올 수 있도록 해당 내용을 등록하는 함수라고 볼 수 있다. 이 함수를 등록하는 과정은 다음과 같다.

1. 첫 번째, 반드시 IP Filter Driver의 포인터를 가지고 있어야 한다. 또한 IP Filter Driver가 설치되고 정상 작동이 되고 있어야 하며 우리가 원하는 커널 레벨의 드라이버 보다 먼저 작동이 되어있어야지만 된다.
2. 두 번째, 특별한 IRP를 만들어서 IO Control Code 만들어 줘야 한다. 또한 PF_SET_EXTENSION_HOOK_INFO 구조체를 만들어서 필터 함수에 필터링 할 수 있는 정보를 보내줘야 한다.
3. 디바이스 드라이버에 IRP를 보내준다.

하지만 여기서 문제점이 들어난다. 바로 단 하나의 필터 함수만 등록이 가능 하다는 것이다. 만약에 같은 Filter 함수를 사용하는 프로그램이 실행되어져 있다면 우리의 프로그램은 실행되지 않을 것이다.

아래의 코드는 필터 함수를 등록하는 함수이다.

```
NTSTATUS SetFilterFunction
            (PacketFilterExtensionPtr filterFunction)
{
    NTSTATUS status = STATUS_SUCCESS, waitStatus=STATUS_SUCCESS;
    UNICODE_STRING filterName;
    PDEVICE_OBJECT ipDeviceObject=NULL;
    PFILE_OBJECT ipFileObject=NULL;

    PF_SET_EXTENSION_HOOK_INFO filterData;

    KEVENT event;
    IO_STATUS_BLOCK ioStatus;
    PIRP irp;

    // IpFilterDriver 장치의 포인터를 가져온다.
    RtlInitUnicodeString(&filterName, DD_IPFLTRDRVR_DEVICE_NAME);
    status = IoGetDeviceObjectPointer(&filterName,STANDARD_RIGHTS_ALL,
                                      &ipFileObject, &ipDeviceObject);
```

```
if(NT_SUCCESS(status))
{
    // 함수의 인자들과 함께 구조체를 초기화 시켜준다.
    filterData.ExtensionPointer = filterFunction;

    // 이벤트를 초기화 시켜준다.
    // 그러면 IpFilterDriver 로부터 이벤트가 세팅되면
    // 우리는 작업 완료
    KeInitializeEvent(&event, NotificationEvent, FALSE);

    // irp를 만들어서 필터 함수를 내보낼 수 있도록 한다.
    irp = IoBuildDeviceIoControlRequest(IOCTL_PF_SET_EXTENSION_POINTER,
                                                    ipDeviceObject,
    if(irp != NULL)
    {
        // irp 보냄
        status = IoCallDriver(ipDeviceObject, irp);

        // IpFilterDriver 로부터 이벤트 셋 기다림
        if (status == STATUS_PENDING)
        {
            waitStatus = KeWaitForSingleObject(&event,
                        Executive, KernelMode, FALSE, NULL);

            if (waitStatus != STATUS_SUCCESS )
                // 에러
        }

        status = ioStatus.Status;

        if(!NT_SUCCESS(status))
            // 에러
    }

    else
    {
        // 실패할 경우 에러를 리턴한다.
        status = STATUS_INSUFFICIENT_RESOURCES;

        // IRP 생성 오류
    }

    if(ipFileObject != NULL)
        ObDereferenceObject(ipFileObject);

    ipFileObject = NULL;
    ipDeviceObject = NULL;
}

else
    // 포인터 가져 오는 동안 오류
```

```
    return status;
}
```

위의 소스를 보면 알 수 있듯이 필터 함수를 등록하는 프로세스를 완료하게 되면 우리가 처음에 얻었던 Device Driver와 다른 포인터의 객체를 얻을 수 있다. 그리고 우리의 네트워크 필터 드라이버가 IRP 처리를 끝내면 이벤트를 일으킬 수 있도록 하였다.

- 필터 함수

우리는 지금 까지 어떻게 커널 모드의 드라이버를 제작하고 어떻게 Hooking 이라는 과정을 통해서 패킷을 필터링 하는지 또 기존의 커널 모드의 드라이버에 Hooking 함수를 등록을 하는지 알 수 있게 되었다. 하지만 정말 중요하다고 할 수 있는 어떻게 들어오고 나가는 패킷을 차단하고 허용을 하는지에 대한 본격적인 이야기는 하지 않았다. 우리가 등록한 필터 함수는 그 필터 함수의 결과 값에 따라 어떻게 해당 패킷을 처리를 할지를 결정하게 되는데 다음을 살펴보도록 하자.

```
typedef  PF_FORWARD_ACTION (*PacketFilterExtensionPtr)(
  // IP 패킷 헤더
  IN unsigned char *PacketHeader,
  // 헤더를 포함하지 않는 패킷
  IN unsigned char *Packet,
  // IP 패킷 헤더의 길이를 제외한 패킷 길이
  IN unsigned int PacketLength,
  // 장치 인덱스(몇 번째 장치 인지)
  // 받은 패킷에 대해서
  IN unsigned int RecvInterfaceIndex,
  // 장치 인덱스(몇 번째 장치 인지)
  // 보내는 패킷에 대해서
  IN unsigned int SendInterfaceIndex,
  // IP 주소 형태
  // 장치가 받은 주소
  IN IPAddr RecvLinkNextHop,
  // IP 주소 형태
  // 장치가 보낼 주소
  IN IPAddr SendLinkNextHop
  );
```

PF_FORWARD_ACTION 은 아래와 같은 결과 값을 가질 수 있게 된다.

• PF_FORWARD

패킷을 정상적으로 처리 하기 위해 시스템 상의 IP Stack 에 값을 넣는다. 넣게 되면 해당 패킷은 처리를 하기 위한 어플리케이션으로 넘어가게 되며 해당 어플리케이션에서는 받은 정보를 가지고 적절한 처리를 하게 된다.

• PF_DROP

패킷을 드롭 하게 된다. 시스템 상의 IP Stack 에 해당 포인터를 넘겨 주지 않고 폐기를

함으로써 어플리케이션은 해당 패킷을 받지 못하게 된다.

• PF_PASS
패킷을 그냥 통과 시킨다. IP Stack 에 넣지는 않지만 시스템 드라이버 내부는 통과 하게 된다. 하지만 IP Stack 에 값을 넣지 않기 때문에 어플리케이션에서는 정상적인 패킷 데이터를 받지 못하는 것으로 나온다.

기본적으로 DDK 에서 선언되어 있는 값은 위의 3개이다. 물론 추가 적인 값도 설정이 되어 있지만 여기서는 예외로 친다. (더 많은 정보를 보고 싶으면 DDK를 설치하고 pfhook.h 파일을 살펴 보면 될 것이다.) ICMP 패킷 관련하여 PF_ICMP_ON_DROP 함수가 따로 있다. 이 함수는 ICMP 패킷에 대해서 따로 처리를 하는 부분이라고 볼 수 있다.

필터 함수를 보면 알 수 있듯이 패킷과 그 패킷의 헤더는 포인터 형태로 필터 함수를 통과하게 된다. 따라서 패킷의 헤더 정보를 변경하는 작업 또한 필터 함수에서 진행을 할 수 있게 되는 것이다.

- 마치면서
Filter Hook Driver를 이용해서 네트워크 패킷 필터 드라이버를 작성할 수 있는 방법에 대해서 이야기를 해보았다. 물론 강력하고 성능이 좋은 방화벽을 만드려면 더욱더 깊은 커널 레벨에서 NDIS, TDI 등을 직접 건드려 보면서 작성하는 것이 좋겠지만 기본적인 방화벽 기능을 하는 어플리케이션을 작성하고자 한다면 위의 방법을 이용해도 충분히 가능할 것이다. 방화벽 작성에 필요한 모든 소스는 이야기 하지 않았다. 하지만 가장 중요하다고 볼 수 있는 핵심 부분을 모두 이야기 하였고 위의 기능을 충분히 이해하고 작성하고자 한다면 어느정도 필요한 네트워크 방화벽은 만들 수 있게 될 것이다.