

在前几篇文章中 LyShark 通过多种方式实现了驱动程序与应用层之间的通信，这其中就包括了通过运用 `SystemBuf` 缓冲区通信，运用 `ReadFile` 读写通信，运用 `PIPE` 管道通信，以及运用 `ASYNC` 反向通信，这些通信方式在应对一收一发模式的时候效率极高，但往往我们需要实现一次性吐出多种数据，例如ARK工具中当我们枚举内核模块时，往往应用层例程中可以返回几条甚至是几十条结果，如下案例所示，这对于开发一款ARK反内核工具是必须要有的功能。

进程	驱动模块	内核层	内核钩子	应用层钩子	设置	监控	启动信息	注册表	服务	文件	网络	调试引擎
系统回调	过滤驱动	DPC定时器	IO定时器	系统线程	卸载的驱动							
回调入口	通知类型	模块路径				文件厂商	备注					
0xFFFFF8051C605110	CreateProcess	C:\Windows\system32\CI.dll				Microsoft Corporation	-					
0xFFFFF8051EDAB210	LoadImage	C:\Windows\system32\DRIVERS\ahcache.sy				Microsoft Corporation	-					
0xFFFFF8051C677220	CreateProcess	C:\Windows\System32\drivers\cng.sys				Microsoft Corporation	-					
0xFFFFF8051F908C60	CreateProcess	C:\Windows\System32\drivers\dxgkrnl.sys				Microsoft Corporation	-					
0xFFFFF8051DA6D930	CreateProcess	C:\Windows\system32\drivers\jorate.sys				Microsoft Corporation	-					
0xFFFFF8051C25B420	CreateProcess	C:\Windows\System32\drivers\ksecdd.sys				Microsoft Corporation	-					
0xFFFFF8051E843CF0	CreateProcess	C:\Windows\system32\drivers\peauth.sys				Microsoft Corporation	-					

- 那么如何实现如上述功能呢？

其实，实现这类功能可以从两个方面入手，但不论使用哪一种方式本质上都是预留一段缓冲区以此来给内核与应用层共享的区域，该区域内可用于交换数据，实现方式有两种要么在应用层分配空间，要么在内核中分配，LyShark先带大家在 内核层 实现，通过巧妙地运用 `MDL` 映射 机制来实现通信需求。

- MDL是什么呢？

MDL内存读写是最常用的一种读写模式，是用于描述物理地址页面的一个结构，简单的官方解释：内存描述符列表（MDL）是一个系统定义的结构，通过一系列物理地址描述缓冲区。执行直接I/O的驱动程序从I/O管理器接收一个MDL的指针，并通过MDL读写数据。一些驱动程序在执行直接I/O来满足设备I/O控制请求时也使用MDL。

通过运用MDL的方式对同一块物理内存同时映射到R0和R3，这样我们只需要使用 `DeviceIoControl` 向驱动发送一个指针，通过对指针进行读写就可以实现数据的交换，本人在网络上找到了如下两段被转载的烂大街的片段，这两段代码明显是存在缺陷的如果你也在寻找映射方法那么不要被这两段代码坑了，多数人也根本没有能力将其变为可用的，也就只能转载，不知道哪个大哥挖的坑。

用户态进程分配空间，内核态去映射。

```
// assume uva is a virtual address in user space, uva_size is its size
MDL * md1 = IoAllocateMdl(uva, uva_size, FALSE, FALSE, NULL);
ASSERT(md1);
__try {
    MmProbeAndLockPages(md1, UserMode, IoReadAccess);
} __except(EXCEPTION_EXECUTE_HANDLER) {
    DbgPrint("error code = %d", GetExceptionCode());
}
PVOID kva = MmGetSystemAddressForMdlSafe(md1, NormalPagePriority);
// use kva
// ...

MmUnlockPages(md1);
IoFreeMdl(md1);
```

内核态分配空间，用户态进程去映射。

```

VOID kva = ExAllocatePoolWithTag(NonPagedPool, 1024, (ULONG)'PMET');
MDL * md1 = IoAllocateMdl(uva, uva_size, FALSE, FALSE, NULL);
ASSERT(md1);
__try {
    MmBuildMdlForNonPagedPool(md1);
} __except(EXCEPTION_EXECUTE_HANDLER) {
    DbgPrint("error code = %d", GetExceptionCode());
}

VOID uva = MmMapLockedPagesSpecifyCache(md1, UserMode, MmCached, NULL, FALSE,
NormalPagePriority);

```

如上的代码看看就好搞出来只是要提醒大家这个是无法使用的，如下将进入本篇文章的正题。

以内核中开辟空间为例，首先在代码中要做的就是定义一段非分页内存 `#define FILE_DEVICE_EXTENSION 4096` 这段区域用于给全局变量使用，其次我们需要传输结构体那么结构体中的成员就要事先定义好，例如此处使用 `StructAll` 来定义结构体成员变量如下所示，通过使用 `static` 将结构体定义为静态，预先空出 1024 的内存空间并初始化为0，当然了这种方式是存在弊端的，例如最大只支持1024个结构如果超过了则可能会溢出，当然最好的办法是用户空间开辟，在下次章节中再介绍。

```

// -----
// MDL数据传递变量
// -----


// 保存一段非分页内存,用于给全局变量使用
#define FILE_DEVICE_EXTENSION 4096

// 定义重复结构(循环传递)
typedef struct
{
    char username[256];
    char password[256];
    int count;
}StructAll;

static StructAll ptr[1024] = { 0 };

```

为了能够达到输出结构体的效果这里我定义一个 `ShowProcess` 用于模拟当前系统内进程数，并自动填充为特定的数据，此处结构体内部 `count` 成员则用于标注当前共有多少个结构体，用于在用户层读取判断，当然了这种方式的另一个弊端就是浪费空间，因为每一个结构体中都存在一个被填充为0的整数类型。但如果只是实现功能的话其实也不是那么重要。

```

// 模拟进程列表赋值测试
int ShowProcess(int process_count)
{
    memset(ptr, 0, sizeof(StructAll) * process_count);
    int x = 0;

    for (; x < process_count + 1; x++)
    {
        strcpy_s(ptr[x].username, 256, "Tyshark");
    }
}

```

```

    strcpy_s(ptr[x].password, 256, "123456");
}

// 设置总共有多少个结构体，并返回结构体个数
ptr[0].count = x;
return x;
}

```

## 内核态映射数据

当定义好如上这些方法时，接下来就是最重要的驱动映射部分了，如下代码所示，首先当用户调用派发时第一个执行的函数是 `ShowProcess()` 它用于获取到当前系统中有多少个进程，接着通过 `sizeof(MyData) * count` 计算出当前 `MyData` 需要分配的内存池大小并返回给 `pool_size`，调用 `ExAllocatePool` 分配一块非分页内核空间，创建 `IoAllocateMdl` MDL映射，将数据 `MmMapLockedPagesSpecifyCache` 映射到用户空间，最后将指针 `pShareMM_User` 返回给用户态。

- `ShowProcess(715)` 获取当前进程数，并返回数量
- `sizeof(MyData) * count` 计算得到结构体长度
- `ExAllocatePool(NonPagedPool, pool_size)` 分配非分页内存，长度是`pool_size`
- `IoAllocateMdl()` 分配MDL空间，并放入内核态
- `MmMapLockedPagesSpecifyCache()` 将内核态指针映射到用户态
- `RtlCopyMemory(pShareMM_SYS, &ptr, sizeof(ptr0) * count)` 将总进程数放入到`count`计数变量内
- `*(PVOID *)pIrp->AssociatedIrp.SystemBuffer = pShareMM_User` 直接将指针传递给用户态

```

// 获取到当前列表数据
int count = ShowProcess(715);
long pool_size = sizeof(MyData) * count;

DbgPrint("总进程数: %d 分配内存池大小: %d \n", count, pool_size);

__try
{
    // 分配内核空间
    PVOID pShareMM_SYS = ExAllocatePool(NonPagedPool, pool_size);
    RtlZeroMemory(pShareMM_SYS, pool_size);

    // 创建MDL
    PMDL pShareMM_MDL = IoAllocateMdl(pShareMM_SYS, pool_size, FALSE, FALSE, NULL);
    MmBuildMdlForNonPagedPool(pShareMM_MDL);

    // 将内核空间映射到用户空间
    PVOID pShareMM_User = MmMapLockedPagesSpecifyCache(pShareMM_MDL, UserMode, MmCached, NULL,
FALSE, NormalPagePriority);

    // 拷贝发送数据
    RtlCopyMemory(pShareMM_SYS, &ptr, sizeof(ptr[0]) * count);

    DbgPrint("[lyshark] 用户地址空间: 0x%x \n", pShareMM_User);
    DbgPrint("[lyshark] 内核地址空间: 0x%p \n", pShareMM_SYS);
}

```

```

// 将字符串指针发送给应用层
*(PVOID *)pIrp->AssociatedIrp.SystemBuffer = pShareMM_User;

// ExFreePool(pshareMM_SYS);
}

__except (EXCEPTION_EXECUTE_HANDLER)
{
    break;
}

status = STATUS_SUCCESS;
break;

```

## 用户态读取数据

与内核层一致，用户层同样需要定义 `structAll` 结构体用于接收内核中返回过来的结构，而重要的代码则是接收部分，通过 `IoControl` 发送控制码，并得到 `ptr` 内存指针，此处区域就是内核态分配过的指针，用户只需要通过循环的方式依次读出里面的数据即可。

```

// -----
// MDL数据传递变量
// -----
// 定义重复结构(循环传递)
typedef struct
{
    char username[256];
    char password[256];
    int count;
}StructAll;

// 直接输出循环结构体
StructAll *ptr;

// 派遣命令
DriveControl.IoControl(IOCTL_IO_MDLStructAll, 0, 0, &ptr, sizeof(PVOID), 0);

printf("共享内存地址: %x \n", ptr);

long size = ptr[0].count;

std::cout << "得到结构体总数: " << size << std::endl;

for (int x = 0; x < size; x++)
{
    std::cout << "计数器: " << x << std::endl;
    std::cout << "用户名: " << ptr[x].username << std::endl;
    std::cout << "密码: " << ptr[x].password << std::endl;
    std::cout << std::endl;
}

```

如上就是内核层与应用层的部分代码功能分析，接下来我将完整代码分享出来，大家可以自行测试效果。

驱动程序 WinDDK.sys 完整代码；

```
#define _CRT_SECURE_NO_WARNINGS
#include <ntifs.h>
#include <windef.h>

// 定义符号链接，一般来说修改为驱动的名字即可
#define DEVICE_NAME          L"\Device\WinDDK"
#define LINK_NAME             L"\DosDevices\WinDDK"
#define LINK_GLOBAL_NAME      L"\DosDevices\Global\WinDDK"

// 定义驱动功能号和名字，提供接口给应用程序调用
#define IOCTL_IO_MDLStructAll CTL_CODE(FILE_DEVICE_UNKNOWN, 0x805, METHOD_BUFFERED,
FILE_ANY_ACCESS)

// 保存一段非分页内存，用于给全局变量使用
#define FILE_DEVICE_EXTENSION 4096

// 定义传递结构体
typedef struct
{
    int uuid;
    char szName[1024];
} MyData;

// -----
// MDL数据传递变量
// -----

// 定义重复结构(循环传递)
typedef struct
{
    char username[256];
    char password[256];
    int count;
} StructAll;

static StructAll ptr[1024] = { 0 };

// 模拟进程列表赋值测试
int ShowProcess(int process_count)
{
    memset(ptr, 0, sizeof(StructAll) * process_count);
    int x = 0;

    for (; x < process_count + 1; x++)
    {
        strcpy_s(ptr[x].username, 256, "hello lyshark.com");
        strcpy_s(ptr[x].password, 256, "123456");
    }

    // 设置总共有多少个结构体，并返回结构体个数
    ptr[0].count = x;
```

```

    return x;
}

// 驱动绑定默认派遣函数
NTSTATUS DefaultDispatch(PDEVICE_OBJECT _pDeviceObject, PIRP _pIrp)
{
    _pIrp->IoStatus.Status = STATUS_NOT_SUPPORTED;
    _pIrp->IoStatus.Information = 0;
    IoCompleteRequest(_pIrp, IO_NO_INCREMENT);
    return _pIrp->IoStatus.Status;
}

// 驱动卸载的处理例程
VOID DriverUnload(PDRIVER_OBJECT pDriverObj)
{
    if (pDriverObj->DeviceObject)
    {
        UNICODE_STRING strLink;

        // 删除符号连接和设备
        RtlInitUnicodeString(&strLink, LINK_NAME);
        IoDeleteSymbolicLink(&strLink);
        IoDeleteDevice(pDriverObj->DeviceObject);
        DbgPrint("[kernel] # 驱动已卸载 \n");
    }
}

// IRP_MJ_CREATE 对应的处理例程，一般不用管它
NTSTATUS DispatchCreate(PDEVICE_OBJECT pDevObj, PIRP pIrp)
{
    DbgPrint("[kernel] # 驱动处理例程载入 \n");
    pIrp->IoStatus.Status = STATUS_SUCCESS;
    pIrp->IoStatus.Information = 0;
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}

// IRP_MJ_CLOSE 对应的处理例程，一般不用管它
NTSTATUS DispatchClose(PDEVICE_OBJECT pDevObj, PIRP pIrp)
{
    DbgPrint("[kernel] # 关闭派遣 \n");
    pIrp->IoStatus.Status = STATUS_SUCCESS;
    pIrp->IoStatus.Information = 0;
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}

// IRP_MJ_DEVICE_CONTROL 对应的处理例程，驱动最重要的函数
NTSTATUS DispatchIoctl(PDEVICE_OBJECT pDevObj, PIRP pIrp)
{
    NTSTATUS status = STATUS_INVALID_DEVICE_REQUEST;
    PIO_STACK_LOCATION pIrpStack;
    ULONG uIoControlCode;
}

```

```
PVOID pIoBuffer;
ULONG uInSize;
ULONG uOutSize;

// 获得IRP里的关键数据
pIrpStack = IoGetCurrentIrpStackLocation(pIrp);

// 获取控制码
uIoControlCode = pIrpStack->Parameters.DeviceIoControl.IoControlCode;

// 输入和输出的缓冲区 (DeviceIoControl的InBuffer和OutBuffer都是它)
pIoBuffer = pIrp->AssociatedIrp.SystemBuffer;

// EXE发送传入数据的BUFFER长度 (DeviceIoControl的nInBufferSize)
uInSize = pIrpStack->Parameters.DeviceIoControl.InputBufferLength;

// EXE接收传出数据的BUFFER长度 (DeviceIoControl的nOutBufferSize)
uOutSize = pIrpStack->Parameters.DeviceIoControl.OutputBufferLength;

// 对不同控制信号的处理流程
switch (uIoControlCode)
{
    // 测试MDL传输多次结构体
    case IOCTL_IO_MDLSTRUCTALL:
    {
        // 获取到当前列表数据
        int count = ShowProcess(715);
        Long pool_size = sizeof(MyData) * count;

        DbgPrint("总进程数: %d 分配内存池大小: %d \n", count, pool_size);

        __try
        {
            // 分配内核空间
            PVOID pShareMM_SYS = ExAllocatePool(NonPagedPool, pool_size);
            RtlZeroMemory(pShareMM_SYS, pool_size);

            // 创建MDL
            PMDL pShareMM_MDL = IoAllocateMdl(pShareMM_SYS, pool_size, FALSE, FALSE, NULL);
            MmBuildMdlForNonPagedPool(pShareMM_MDL);

            // 将内核空间映射到用户空间
            PVOID pShareMM_User = MmMapLockedPagesSpecifyCache(pShareMM_MDL, UserMode,
                MmCached, NULL, FALSE, NormalPagePriority);

            // 拷贝发送数据
            RtlCopyMemory(pShareMM_SYS, &ptr, sizeof(ptr[0]) * count);

            DbgPrint("[lyshark.com] 用户地址空间: 0x%x \n", pShareMM_User);
            DbgPrint("[lyshark.com] 内核地址空间: 0x%p \n", pShareMM_SYS);

            // 将字符串指针发送给应用层
            *(PVOID *)pIrp->AssociatedIrp.SystemBuffer = pShareMM_User;
        }
    }
}
```

```

        // ExFreePool(pShareMM_SYS);
    }
__except (EXCEPTION_EXECUTE_HANDLER)
{
    break;
}
status = STATUS_SUCCESS;
break;
}
}

// 设定DeviceIoControl的*lpBytesReturned的值（如果通信失败则返回0长度）
if (status == STATUS_SUCCESS)
{
    pIrp->IoStatus.Information = uOutSize;
}
else
{
    pIrp->IoStatus.Information = 0;
}

// 设定DeviceIoControl的返回值是成功还是失败
pIrp->IoStatus.Status = status;
IoCompleteRequest(pIrp, IO_NO_INCREMENT);
return status;
}

// 驱动的初始化工作
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObj, PUNICODE_STRING pRegistryString)
{
    NTSTATUS status = STATUS_SUCCESS;
    UNICODE_STRING ustrLinkName;
    UNICODE_STRING ustrDevName;
    PDEVICE_OBJECT pDevObj;

    // 初始化其他派遣
    for (ULONG i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
    {
        DbgPrint("初始化派遣: %d \n", i);
        pDriverObj->MajorFunction[i] = DefaultDispatch;
    }

    // 设置分发函数和卸载例程
    pDriverObj->MajorFunction[IRP_MJ_CREATE] = DispatchCreate;
    pDriverObj->MajorFunction[IRP_MJ_CLOSE] = DispatchClose;
    pDriverObj->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DispatchIoctl;
    pDriverObj->DriverUnload = DriverUnload;

    // 创建一个设备
    RtlInitUnicodeString(&ustrDevName, DEVICE_NAME);
}

```

```

// FILE_DEVICE_EXTENSION 创建设备时，指定设备扩展内存的大小，传一个值进去，就会给设备分配一块非页面
内存。
status = IoCreateDevice(pDriverObj, sizeof(FILE_DEVICE_EXTENSION), &ustrDevName,
FILE_DEVICE_UNKNOWN, 0, FALSE, &pDevObj);
if (!NT_SUCCESS(status))
{
    return status;
}

// 判断支持的WDM版本，其实这个已经不需要了，纯属WIN9X和WINNT并存时代的残留物
if (IoIsWdmVersionAvailable(1, 0x10))
{
    RtlInitUnicodeString(&ustrLinkName, LINK_GLOBAL_NAME);
}
else
{
    RtlInitUnicodeString(&ustrLinkName, LINK_NAME);
}

// 创建符号连接
status = IoCreateSymbolicLink(&ustrLinkName, &ustrDevName);
if (!NT_SUCCESS(status))
{
    DbgPrint("创建符号链接失败 \n");
    IoDeleteDevice(pDevObj);
    return status;
}
DbgPrint("[kernel] # hello lyshark.com \n");

// 返回加载驱动的状态（如果返回失败，驱动将被清除出内核空间）
return STATUS_SUCCESS;
}

```

应用层客户端程序 lyshark.exe 完整代码；

```

#include <iostream>
#include <windows.h>
#include <vector>

#pragma comment(lib,"user32.lib")
#pragma comment(lib,"advapi32.lib")

// 定义驱动功能号和名字，提供接口给应用程序调用
#define IOCTL_IO_MDLStructAll 0x805

class cDrvCtrl
{
public:
    cDrvCtrl()
    {
        m_pSysPath = NULL;
        m_pServiceName = NULL;
        m_pDisplayName = NULL;
    }
}
```

```

    m_hSCManager = NULL;
    m_hService = NULL;
    m_hDriver = INVALID_HANDLE_VALUE;
}
~cDrvCtrl()
{
    CloseServiceHandle(m_hService);
    CloseServiceHandle(m_hSCManager);
    CloseHandle(m_hDriver);
}

// 安装驱动
BOOL Install(PCHAR pSysPath, PCHAR pServiceName, PCHAR pDisplayName)
{
    m_pSysPath = pSysPath;
    m_pServiceName = pServiceName;
    m_pDisplayName = pDisplayName;
    m_hSCManager = OpenSCManagerA(NULL, NULL, SC_MANAGER_ALL_ACCESS);
    if (NULL == m_hSCManager)
    {
        m_dwLastError = GetLastError();
        return FALSE;
    }
    m_hService = CreateServiceA(m_hSCManager, m_pServiceName, m_pDisplayName,
        SERVICE_ALL_ACCESS, SERVICE_KERNEL_DRIVER, SERVICE_DEMAND_START,
        SERVICE_ERROR_NORMAL,
        m_pSysPath, NULL, NULL, NULL, NULL, NULL);
    if (NULL == m_hService)
    {
        m_dwLastError = GetLastError();
        if (ERROR_SERVICE_EXISTS == m_dwLastError)
        {
            m_hService = OpenServiceA(m_hSCManager, m_pServiceName, SERVICE_ALL_ACCESS);
            if (NULL == m_hService)
            {
                CloseServiceHandle(m_hSCManager);
                return FALSE;
            }
        }
        else
        {
            CloseServiceHandle(m_hSCManager);
            return FALSE;
        }
    }
    return TRUE;
}

// 启动驱动
BOOL Start()
{
    if (!StartServiceA(m_hService, NULL, NULL))
    {

```

```
        m_dwLastError = GetLastError();
        return FALSE;
    }

    return TRUE;
}

// 关闭驱动
BOOL Stop()
{
    SERVICE_STATUS ss;
    GetSvcHandle(m_pServiceName);
    if (!ControlService(m_hService, SERVICE_CONTROL_STOP, &ss))
    {
        m_dwLastError = GetLastError();
        return FALSE;
    }
    return TRUE;
}

// 移除驱动
BOOL Remove()
{
    GetSvcHandle(m_pServiceName);
    if (!DeleteService(m_hService))
    {
        m_dwLastError = GetLastError();
        return FALSE;
    }
    return TRUE;
}

// 打开驱动
BOOL Open(PCHAR pLinkName)
{
    if (m_hDriver != INVALID_HANDLE_VALUE)
        return TRUE;
    m_hDriver = CreateFileA(pLinkName, GENERIC_READ | GENERIC_WRITE, 0, 0,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
    if (m_hDriver != INVALID_HANDLE_VALUE)
        return TRUE;
    else
        return FALSE;
}

// 发送控制信号
BOOL IoControl(DWORD dwIoCode, PVOID InBuff, DWORD InBuffLen, PVOID OutBuff, DWORD
OutBuffLen, DWORD *RealRetBytes)
{
    DWORD dw;
    BOOL b = DeviceIoControl(m_hDriver, CTL_CODE_GEN(dwIoCode), InBuff, InBuffLen,
OutBuff, OutBuffLen, &dw, NULL);
    if (RealRetBytes)
        *RealRetBytes = dw;
```

```

        return b;
    }

private:

    // 获取服务句柄
    BOOL GetSvcHandle(PCHAR pServiceName)
    {
        m_pServiceName = pServiceName;
        m_hSCManager = OpenSCManagerA(NULL, NULL, SC_MANAGER_ALL_ACCESS);
        if (NULL == m_hSCManager)
        {
            m_dwLastError = GetLastError();
            return FALSE;
        }
        m_hService = OpenServiceA(m_hSCManager, m_pServiceName, SERVICE_ALL_ACCESS);
        if (NULL == m_hService)
        {
            CloseServiceHandle(m_hSCManager);
            return FALSE;
        }
        else
        {
            return TRUE;
        }
    }

    // 获取控制信号对应字符串
    DWORD CTL_CODE_GEN(DWORD lngFunction)
    {
        return (FILE_DEVICE_UNKNOWN * 65536) | (FILE_ANY_ACCESS * 16384) | (lngFunction * 4)
    | METHOD_BUFFERED;
    }

public:
    DWORD m_dwLastError;
    PCHAR m_pSysPath;
    PCHAR m_pServiceName;
    PCHAR m_pDisplayName;
    HANDLE m_hDriver;
    SC_HANDLE m_hSCManager;
    SC_HANDLE m_hService;
};

void GetAppPath(char *szCurFile)
{
    GetModuleFileNameA(0, szCurFile, MAX_PATH);
    for (SIZE_T i = strlen(szCurFile) - 1; i >= 0; i--)
    {
        if (szCurFile[i] == '\\')
        {
            szCurFile[i + 1] = '\0';
            break;
        }
    }
}

```

```
    }
}

// -----
// MDL数据传递变量
// -----
// 定义重复结构(循环传递)
typedef struct
{
    char username[256];
    char password[256];
    int count;
}StructAll;

int main(int argc, char *argv[])
{
    cDrvCtrl DriveControl;

    // 设置驱动名称
    char szSysFile[MAX_PATH] = { 0 };
    char szSvcLnkName[] = "WinDDK";
    GetAppPath(szSysFile);
    strcat(szSysFile, "WinDDK.sys");

    // 安装并启动驱动
    DriveControl.Install(szSysFile, szSvcLnkName, szSvcLnkName);
    DriveControl.Start();

    // 打开驱动的符号链接
    DriveControl.Open("\\\\.\\"WinDDK");

    // 直接输出循环结构体
    StructAll *ptr;

    // 派遣命令
    DriveControl.IoControl(IOCTL_IO_MDLStructAll, 0, 0, &ptr, sizeof(PVOID), 0);

    printf("[Lyshark.com] 共享内存地址: %x \n", ptr);

    long size = ptr[0].count;

    std::cout << "得到结构体总数: " << size << std::endl;

    for (int x = 0; x < size; x++)
    {
        std::cout << "计数器: " << x << std::endl;
        std::cout << "用户名: " << ptr[x].username << std::endl;
        std::cout << "密码: " << ptr[x].password << std::endl;
        std::cout << std::endl;
    }

    // 关闭符号链接句柄
    CloseHandle(DriveControl.m_hDriver);
```

```
// 停止并卸载驱动
DriveControl.Stop();
DriveControl.Remove();

system("pause");
return 0;
}
```

手动编译这两个程序，将驱动签名后以管理员身份运行 lyshark.exe 客户端，此时屏幕上即可看到滚动输出效果，如此一来就实现了循环传递参数的目的。



