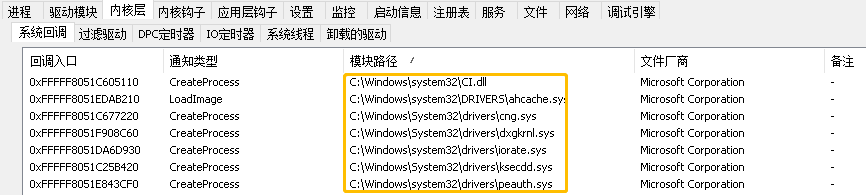
# 在前几篇文章中 LyShark 通过多种方式实现了驱动程序与应用层之间的通信，这其中就包括了通过运用

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。



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 mdl = IoAllocateMdl(uva, uva\_size, FALSE, FALSE, NULL); ASSERT(mdl);

try {

MmProbeAndLockPages(mdl, UserMode, IoReadAccess);

} except(EXCEPTION\_EXECUTE\_HANDLER) {

DbgPrint("error code = %d", GetExceptionCode);

}

PVOID kva = MmGetSystemAddressForMdlSafe(mdl, NormalPagePriority);

// use kva

// …

MmUnlockPages(mdl); IoFreeMdl(mdl);

内核态分配空间，用户态进程去映射。



PVOID kva = ExAllocatePoolWithTag(NonPagedPool, 1024, (ULONG)'PMET'); MDL mdl = IoAllocateMdl(uva, uva\_size, FALSE, FALSE, NULL); ASSERT(mdl);

try {

MmBuildMdlForNonPagedPool(mdl);

} except(EXCEPTION\_EXECUTE\_HANDLER) {

DbgPrint("error code = %d", GetExceptionCode);

}

PVOID uva = MmMapLockedPagesSpecifyCache(mdl, UserMode, MmCached, NULL, FALSE, NormalPagePriority);

如上的代码看看就好摘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, "lyshark");

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：** 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，首先当用户调用派遣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(ptr[0]) \* 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



// 署名权

// right to sign one's name on a piece of work

// PowerBy: LyShark

// Email: [me@lyshark.com](mailto:me@lyshark.com)

// 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。



// 署名权

// right to sign one's name on a piece of work

// PowerBy: LyShark

// Email: [me@lyshark.com](mailto:me@lyshark.com)

// -------------------------------------------------

// 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；

// 署名权

// right to sign one's name on a piece of work

// PowerBy: LyShark

// Email: [me@lyshark.com](mailto:me@lyshark.com)

#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 szUname[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);

NULL);

// 创建MDL

PMDL pShareMM\_MDL = IoAllocateMdl(pShareMM\_SYS, pool\_size, FALSE, FALSE,

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，驱动讲被清除出内核空间）

return STATUS\_SUCCESS;

}

应用层客户端程序 lyshark.exe 完整代码；

// 署名权

// right to sign one's name on a piece of work

// PowerBy: LyShark

// Email: [me@lyshark.com](mailto:me@lyshark.com)

#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，此时屏幕中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。

