# 在前面的文章 《驱动开发：运用MDL映射实现多次通信》 LyShark教大家使用 MDL 的方式灵活的实现了内核态多次输出结构体的效果，但是此种方法并不推荐大家使用原因很简单首先内核空间比较宝贵，其次内核里面不 能分配太大且每次传出的结构体最大不能超过 1024 个，而最终这些内存由于无法得到更好的释放从而导致坏堆的产生，这样的程序显然是无法在生产环境中使用的，如下 LyShark 将教大家通过在应用层申请空间来实现同等效果，此类传递方式也是多数ARK反内核工具中最常采用的一种。

与MDL映射相反，MDL多数处理流程在内核代码中，而应用层开堆复杂代码则在应用层，但内核层中同样还 是需要使用指针，只是这里的指针仅仅只是保留基本要素即可，通过 EnumProcess() 模拟枚举进程操作，传入的是 PPROCESS\_INFO 进程指针转换，将数据传入到 PPROCESS\_INFO 直接返回进程计数器即可。



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

// R3传输结构体

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

// 进程指针转换

typedef struct

{

DWORD PID; DWORD PPID;

}PROCESS\_INFO, PPROCESS\_INFO;

// 数据存储指针

typedef struct

{

ULONG\_PTR nSize; PVOID BufferPtr;

}BufferPointer, pBufferPointer;

// 模拟进程枚举

ULONG EnumProcess(PPROCESS\_INFO pBuffer)

{

ULONG nCount = 0;

for (size\_t i = 0; i < 10; i++)

{

pBuffer[i].PID = nCount 2; pBuffer[i].PPID = nCount 4;

nCount = nCount + 1;

}

return nCount;

}

**内核层核心代码：** 内核代码中是如何通信的，首先从用户态接收 pIoBuffer 到分配的缓冲区数据，并转换为 pBufferPointer 结构， ProbeForWrite 用于检查地址是否可写入，接着会调用 EnumProcess() 注意传入的其实是应用层的指针，枚举进程结束后，将进程数量 nCount 通过 (PULONG)pIrp-



>AssociatedIrp.SystemBuffer = (ULONG)nCount 回传给应用层，至此内核中仅仅回传了一个长度，其

# 他的都写入到了应用层中。

// 署名权



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

// PowerBy: LyShark

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

pBufferPointer pinp = (pBufferPointer)pIoBuffer;

try

{

DbgPrint("缓冲区长度: %d \n", pinp->nSize);

DbgPrint("缓冲区基地址: %p \n", pinp->BufferPtr);

// 检查地址是否可写入

ProbeForWrite(pinp->BufferPtr, pinp->nSize, 1);

ULONG nCount = EnumProcess((PPROCESS\_INFO)pinp->BufferPtr);

DbgPrint("进程计数 = %d \n", nCount); if (nCount > 0)

{

// 将进程数返回给用户

(PULONG)pIrp->AssociatedIrp.SystemBuffer = (ULONG)nCount; status = STATUS\_SUCCESS;

}

}

except (1)

{

status = GetExceptionCode(); DbgPrint("IOCTL\_GET\_EPROCESS %x \n", status);

}

// 返回通信状态

status = STATUS\_SUCCESS; break;

**应用层核心代码：** 通信的重点在于应用层，首先定义 BufferPointer 用于存放缓冲区头部指针，定义

PPROCESS\_INFO 则是用于后期将数据放入该容器内，函数 HeapAlloc 分配一段堆空间，并

HEAP\_ZERO\_MEMORY 将该堆空间全部填空，将这一段初始化后的空间放入到 pInput.BufferPtr 缓冲区内， 并计算出长度放入到 pInput.nSize 缓冲区内，一切准备就绪之后，再通过 DriveControl.IoControl 将

BufferPointer 结构传输至内核中，而 bRet 则是用于接收返回长度的变量。

当收到数据后，通过 (PPROCESS\_INFO)pInput.BufferPtr 强制转换为指针类型，并依次

pProcessInfo[i] 读出每一个节点的元素，最后是调用 HeapFree 释放掉这段堆空间。至于输出就很简单了 vectorProcess[x].PID 循环容器元素即可。

// 署名权

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

// PowerBy: LyShark

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

// 应用层数据结构体数据

BOOL bRet = FALSE;

BufferPointer pInput = { 0 };

PPROCESS\_INFO pProcessInfo = NULL;



// 分配堆空间

pInput.BufferPtr = (PVOID)HeapAlloc(GetProcessHeap(), HEAP\_ZERO\_MEMORY, sizeof(PROCESS\_INFO) 1000);

pInput.nSize = sizeof(PROCESS\_INFO) 1000;

ULONG nRet = 0;

if (pInput.BufferPtr)

{

bRet = DriveControl.IoControl(IOCTL\_IO\_R3StructAll, &pInput, sizeof(BufferPointer), &nRet, sizeof(ULONG), 0);

}

std::cout << "返回结构体数量: " << nRet << std::endl; if (bRet && nRet > 0)

{

pProcessInfo = (PPROCESS\_INFO)pInput.BufferPtr; std::vector<PROCESS\_INFO> vectorProcess;

for (ULONG i = 0; i < nRet; i++)

{

vectorProcess.push\_back(pProcessInfo[i]);

}

// 释放空间

bRet = HeapFree(GetProcessHeap(), 0, pInput.BufferPtr); std::cout << "释放状态: " << bRet << std::endl;

// 输出容器内的进程ID列表

for (int x = 0; x < nRet; x++)

{

std::cout << "PID: " << vectorProcess[x].PID << " PPID: " << vectorProcess[x].PPID << std::endl;

}

}

// 关闭符号链接句柄

CloseHandle(DriveControl.m\_hDriver);

# 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\_R3StructAll CTL\_CODE(FILE\_DEVICE\_UNKNOWN, 0x806, 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

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

// R3传输结构体

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

// 进程指针转换

typedef struct

{

DWORD PID; DWORD PPID;

}PROCESS\_INFO, PPROCESS\_INFO;

// 数据存储指针

typedef struct

{

ULONG\_PTR nSize; PVOID BufferPtr;

}BufferPointer, pBufferPointer;

// 模拟进程枚举

ULONG EnumProcess(PPROCESS\_INFO pBuffer)

{

ULONG nCount = 0;

for (size\_t i = 0; i < 10; i++)

{

pBuffer[i].PID = nCount 2; pBuffer[i].PPID = nCount 4;

nCount = nCount + 1;

}

return nCount;

}

// 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R3传输多次结构体

case IOCTL\_IO\_R3StructAll:

{

pBufferPointer pinp = (pBufferPointer)pIoBuffer;

try

{

DbgPrint("[lyshark] 缓冲区长度: %d \n", pinp->nSize);

DbgPrint("[lyshark] 缓冲区基地址: %p \n", pinp->BufferPtr);

// 检查地址是否可写入

ProbeForWrite(pinp->BufferPtr, pinp->nSize, 1);

ULONG nCount = EnumProcess((PPROCESS\_INFO)pinp->BufferPtr);

DbgPrint("[lyshark.com] 进程计数 = %d \n", nCount); if (nCount > 0)

{

// 将进程数返回给用户

(PULONG)pIrp->AssociatedIrp.SystemBuffer = (ULONG)nCount; status = STATUS\_SUCCESS;

}

}

except (1)

{

status = GetExceptionCode(); DbgPrint("IOCTL\_GET\_EPROCESS %x \n", status);

}

// 返回通信状态

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("[hello LyShark.com] # 驱动初始化完毕 \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\_R3StructAll 0x806

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;

}

}

}

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



// R3数据传递变量

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

// 进程指针转换

typedef struct

{

DWORD PID; DWORD PPID;

}PROCESS\_INFO, PPROCESS\_INFO;

// 数据存储指针

typedef struct

{

ULONG\_PTR nSize; PVOID BufferPtr;

}BufferPointer, pBufferPointer;

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");

// 应用层数据结构体数据

BOOL bRet = FALSE;

BufferPointer pInput = { 0 }; PPROCESS\_INFO pProcessInfo = NULL;

// 分配堆空间

pInput.BufferPtr = (PVOID)HeapAlloc(GetProcessHeap(), HEAP\_ZERO\_MEMORY, sizeof(PROCESS\_INFO) 1000);

pInput.nSize = sizeof(PROCESS\_INFO) 1000;

ULONG nRet = 0;

if (pInput.BufferPtr)

{

bRet = DriveControl.IoControl(IOCTL\_IO\_R3StructAll, &pInput, sizeof(BufferPointer), &nRet, sizeof(ULONG), 0);

}

std::cout << "[LyShark.com] 返回结构体数量: " << nRet << std::endl;

if (bRet && nRet > 0)

{

pProcessInfo = (PPROCESS\_INFO)pInput.BufferPtr; std::vector<PROCESS\_INFO> vectorProcess;

for (ULONG i = 0; i < nRet; i++)

{

vectorProcess.push\_back(pProcessInfo[i]);

}

// 释放空间

bRet = HeapFree(GetProcessHeap(), 0, pInput.BufferPtr); std::cout << "释放状态: " << bRet << std::endl;

// 输出容器内的进程ID列表

for (int x = 0; x < nRet; x++)

{

std::cout << "PID: " << vectorProcess[x].PID << " PPID: " << vectorProcess[x].PPID << 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。

