# 在笔者上一篇文章 《驱动开发：内核MDL读写进程内存》 简单介绍了如何通过MDL映射的方式实现进程读写操作，本章将通过如上案例实现远程进程反汇编功能，此类功能也是ARK工具中最常见的功能之一，通 常此类功能的实现分为两部分，内核部分只负责读写字节集，应用层部分则配合反汇编引擎对字节集进 行解码，此处我们将运用 capstone 引擎实现这个功能。



首先是实现驱动部分，驱动程序的实现是一成不变的，仅仅只是做一个读写功能即可，完整的代码如下 所示；



// 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 <ntifs.h> #include <windef.h>

#define READ\_PROCESS\_CODE CTL\_CODE(FILE\_DEVICE\_UNKNOWN,0x800,METHOD\_BUFFERED,FILE\_ALL\_ACCESS)

#define WRITE\_PROCESS\_CODE CTL\_CODE(FILE\_DEVICE\_UNKNOWN,0x801,METHOD\_BUFFERED,FILE\_ALL\_ACCESS)

#define DEVICENAME L"\\Device\\ReadWriteDevice" #define SYMBOLNAME L"\\??\\ReadWriteSymbolName"

typedef struct

{

DWORD pid; // 进 程 PID UINT64 address; // 读写地址DWORD size; // 读 写 长 度

BYTE data; // 读写数据集

}ProcessData;

// MDL读取封装

BOOLEAN ReadProcessMemory(ProcessData ProcessData)

{

BOOLEAN bRet = TRUE;

PEPROCESS process = NULL;

// 将 PID 转 为 EProcess PsLookupProcessByProcessId(ProcessData->pid, &process); if (process == NULL)

{

return FALSE;



}

BYTE GetProcessData = NULL;

try

{

// 分配堆空间 NonPagedPool 非分页内存

GetProcessData = ExAllocatePool(NonPagedPool, ProcessData->size);

}

except (1)

{

return FALSE;

}

KAPC\_STATE stack = { 0 };

// 附加到进程

KeStackAttachProcess(process, &stack);

try

{

// 检查进程内存是否可读取

ProbeForRead(ProcessData->address, ProcessData->size, 1);

// 完成拷贝

RtlCopyMemory(GetProcessData, ProcessData->address, ProcessData->size);

}

except (1)

{

bRet = FALSE;

}

// 关闭引用

ObDereferenceObject(process);

// 解除附加

KeUnstackDetachProcess(&stack);

// 拷贝数据

RtlCopyMemory(ProcessData->data, GetProcessData, ProcessData->size);

// 释放堆

ExFreePool(GetProcessData); return bRet;

}

// MDL写入封装

BOOLEAN WriteProcessMemory(ProcessData ProcessData)

{

BOOLEAN bRet = TRUE;

PEPROCESS process = NULL;

// 将 PID 转 为 EProcess PsLookupProcessByProcessId(ProcessData->pid, &process);

if (process == NULL)

{

return FALSE;

}



BYTE GetProcessData = NULL;

try

{

// 分配堆

GetProcessData = ExAllocatePool(NonPagedPool, ProcessData->size);

}

except (1)

{

return FALSE;

}

// 循环写出

for (int i = 0; i < ProcessData->size; i++)

{

GetProcessData[i] = ProcessData->data[i];

}

KAPC\_STATE stack = { 0 };

// 附加进程

KeStackAttachProcess(process, &stack);

// 分配MDL对象

PMDL mdl = IoAllocateMdl(ProcessData->address, ProcessData->size, 0, 0, NULL);

if (mdl == NULL)

{

return FALSE;

}

MmBuildMdlForNonPagedPool(mdl);

BYTE ChangeProcessData = NULL;

try

{

// 锁定地址

ChangeProcessData = MmMapLockedPages(mdl, KernelMode);

// 开始拷贝

RtlCopyMemory(ChangeProcessData, GetProcessData, ProcessData->size);

}

except (1)

{

bRet = FALSE;

goto END;

}

// 结束释放MDL关闭引用取消附加

END:

IoFreeMdl(mdl); ExFreePool(GetProcessData); KeUnstackDetachProcess(&stack); ObDereferenceObject(process);



return bRet;

}

NTSTATUS DriverIrpCtl(PDEVICE\_OBJECT device, PIRP pirp)

{

PIO\_STACK\_LOCATION stack;

stack = IoGetCurrentIrpStackLocation(pirp); ProcessData ProcessData;

switch (stack->MajorFunction)

{

case IRP\_MJ\_CREATE:

{

break;

}

case IRP\_MJ\_CLOSE:

{

break;

}

case IRP\_MJ\_DEVICE\_CONTROL:

{

// 获取应用层传值

ProcessData = pirp->AssociatedIrp.SystemBuffer;

DbgPrint("进程ID: %d | 读写地址: %p | 读写长度: %d \n", ProcessData->pid, ProcessData->address, ProcessData->size);

switch (stack->Parameters.DeviceIoControl.IoControlCode)

{

// 读取函数

case READ\_PROCESS\_CODE:

{

ReadProcessMemory(ProcessData); break;

}

// 写入函数

case WRITE\_PROCESS\_CODE:

{

WriteProcessMemory(ProcessData); break;

}

}

pirp->IoStatus.Information = sizeof(ProcessData); break;

}

}

pirp->IoStatus.Status = STATUS\_SUCCESS; IoCompleteRequest(pirp, IO\_NO\_INCREMENT);

return STATUS\_SUCCESS;

}

VOID UnDriver(PDRIVER\_OBJECT driver)

{

if (driver->DeviceObject)

{

UNICODE\_STRING SymbolName; RtlInitUnicodeString(&SymbolName, SYMBOLNAME);

// 删除符号链接IoDeleteSymbolicLink(&SymbolName); IoDeleteDevice(driver->DeviceObject);

}

}

NTSTATUS DriverEntry(IN PDRIVER\_OBJECT Driver, PUNICODE\_STRING RegistryPath)

{

NTSTATUS status = STATUS\_SUCCESS; PDEVICE\_OBJECT device = NULL;

UNICODE\_STRING DeviceName;

DbgPrint("[LyShark] hello lyshark.com \n");

// 初始化设备名

RtlInitUnicodeString(&DeviceName, DEVICENAME);

// 创建设备

status = IoCreateDevice(Driver, sizeof(Driver->DriverExtension), &DeviceName, FILE\_DEVICE\_UNKNOWN, FILE\_DEVICE\_SECURE\_OPEN, FALSE, &device);

if (status == STATUS\_SUCCESS)

{

UNICODE\_STRING SymbolName; RtlInitUnicodeString(&SymbolName, SYMBOLNAME);

// 创建符号链接

status = IoCreateSymbolicLink(&SymbolName, &DeviceName);

// 失败则删除设备

if (status != STATUS\_SUCCESS)

{

IoDeleteDevice(device);

}

}

// 派遣函数初始化

Driver->MajorFunction[IRP\_MJ\_CREATE] = DriverIrpCtl; Driver->MajorFunction[IRP\_MJ\_CLOSE] = DriverIrpCtl;

Driver->MajorFunction[IRP\_MJ\_DEVICE\_CONTROL] = DriverIrpCtl;

// 卸载驱动

Driver->DriverUnload = UnDriver;

return STATUS\_SUCCESS;

}

# 上方的驱动程序很简单关键部分已经做好了备注，此类驱动换汤不换药没啥难度，接下来才是本节课的 重点，让我们开始了解一下 Capstone 这款反汇编引擎吧，Capstone是一个轻量级的多平台、多架构的反汇编框架。Capstone旨在成为安全社区中二进制分析和反汇编的终极反汇编引擎，该引擎支持多种平 台的反汇编，非常推荐使用。

反汇编引擎下载地址：https://cdn.lyshark.com/sdk/capstone\_msvc12.zip

这款反汇编引擎如果你想要使用它则第一步就是调用 cs\_open() 官方对其的解释是打开一个句柄，这个打开功能其中的参数如下所示；

参数1：指定模式 CS\_ARCH\_X86 表示为Windows平台

参数2：执行位数 CS\_MODE\_32为32位模式，CS\_MODE\_64为64位参数3：打开后保存的句柄&dasm\_handle

第二步也是最重要的一步，调用 cs\_disasm() 反汇编函数，该函数的解释如下所示； 参数1：指定dasm\_handle反汇编句柄

参数2：指定你要反汇编的数据集或者是一个缓冲区

参数3：指定你要反汇编的长度 64

参数4：输出的内存地址起始位置 0x401000

参数5：默认填充为0

参数6：用于输出数据的一个指针

这两个函数如果能搞明白，那么如下反汇编完整代码也就可以理解了。



#define \_CRT\_SECURE\_NO\_WARNINGS #include <Windows.h>

#include <iostream>

#include <inttypes.h>

#include <capstone/capstone.h>

#pragma comment(lib,"capstone64.lib")

#define READ\_PROCESS\_CODE CTL\_CODE(FILE\_DEVICE\_UNKNOWN,0x800,METHOD\_BUFFERED,FILE\_ALL\_ACCESS)

#define WRITE\_PROCESS\_CODE CTL\_CODE(FILE\_DEVICE\_UNKNOWN,0x801,METHOD\_BUFFERED,FILE\_ALL\_ACCESS)

typedef struct

{

DWORD pid; UINT64 address; DWORD size; BYTE data;

}ProcessData;

int main(int argc, char argv[])

{

// 连接到驱动

HANDLE handle = CreateFileA("\\??\\ReadWriteSymbolName", GENERIC\_READ | GENERIC\_WRITE, 0, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

ProcessData data;

DWORD dwSize = 0;



// 指定需要读写的进程data.pid = 6932; data.address = 0x401000; data.size = 64;

// 读取机器码到BYTE字节数组

data.data = new BYTE[data.size];

DeviceIoControl(handle, READ\_PROCESS\_CODE, &data, sizeof(data), &data, sizeof(data), &dwSize, NULL);

for (int i = 0; i < data.size; i++)

{

printf("0x%02X ", data.data[i]);

}

printf("\n");

// 开始反汇编

csh dasm\_handle; cs\_insn insn; size\_t count;

// 打开句柄

if (cs\_open(CS\_ARCH\_X86, CS\_MODE\_32, &dasm\_handle) != CS\_ERR\_OK)

{

return 0;

}

// 反汇编代码

count = cs\_disasm(dasm\_handle, (unsigned char )data.data, data.size, data.address, 0, &insn);

if (count > 0)

{

size\_t index;

for (index = 0; index < count; index++)

{

/

for (int x = 0; x < insn[index].size; x++)

{

printf("机器码: %d -> %02X \n", x, insn[index].bytes[x]);

}

/

printf(" 地 址 : 0x%"PRIx64" | 长 度 : %d 反 汇 编 : %s %s \n", insn[index].address, insn[index].size, insn[index].mnemonic, insn[index].op\_str);

}

cs\_free(insn, count);

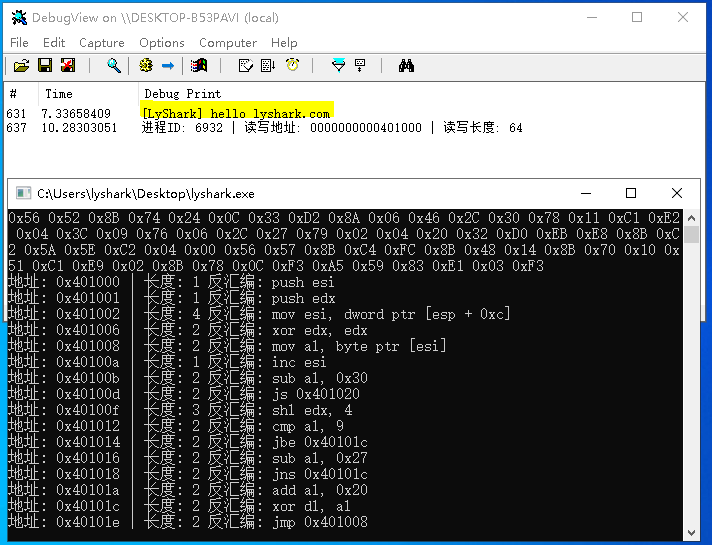
}

cs\_close(&dasm\_handle);

getchar(); CloseHandle(handle); return 0;

}

通过驱动加载工具加载 WinDDK.sys 然后在运行本程序，你会看到正确的输出结果，反汇编当前位置处向下 64 字节。



# 说完了反汇编接着就需要讲解如何对内存进行汇编操作了，汇编引擎这里采用了 XEDParse 该引擎小巧简洁，著名的 x64dbg 就是在运用本引擎进行汇编替换的，本引擎的使用非常简单，只需要向

XEDParseAssemble() 函数传入一个规范的结构体即可完成转换，完整代码如下所示。

# 汇编引擎下载地址：https://cdn.lyshark.com/sdk/XEDParse.zip

#define \_CRT\_SECURE\_NO\_WARNINGS #include <Windows.h>

#include <iostream>

extern "C"

{

#include "D:/XEDParse/XEDParse.h"

#pragma comment(lib, "D:/XEDParse/XEDParse\_x64.lib")

}

using namespace std;

#define READ\_PROCESS\_CODE CTL\_CODE(FILE\_DEVICE\_UNKNOWN,0x800,METHOD\_BUFFERED,FILE\_ALL\_ACCESS)

#define WRITE\_PROCESS\_CODE CTL\_CODE(FILE\_DEVICE\_UNKNOWN,0x801,METHOD\_BUFFERED,FILE\_ALL\_ACCESS)

typedef struct

{

DWORD pid;

UINT64 address; DWORD size; BYTE data;



}ProcessData;

int main(int argc, char argv[])

{

// 连接到驱动

HANDLE handle = CreateFileA("\\??\\ReadWriteSymbolName", GENERIC\_READ | GENERIC\_WRITE, 0, NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

ProcessData data; DWORD dwSize = 0;

// 指定需要读写的进程

data.pid = 6932; data.address = 0x401000; data.size = 0;

XEDPARSE xed = { 0 }; xed.x64 = FALSE;

// 输入一条汇编指令并转换

scanf\_s("%llx", &xed.cip); gets\_s(xed.instr, XEDPARSE\_MAXBUFSIZE);

if (XEDPARSE\_OK != XEDParseAssemble(&xed))

{

printf("指令错误: %s\n", xed.error);

}

// 生成堆

data.data = new BYTE[xed.dest\_size];

// 设置长度

data.size = xed.dest\_size;

for (size\_t i = 0; i < xed.dest\_size; i++)

{

// 替换到堆中

printf("%02X ", xed.dest[i]); data.data[i] = xed.dest[i];

}

// 调用控制器，写入到远端内存

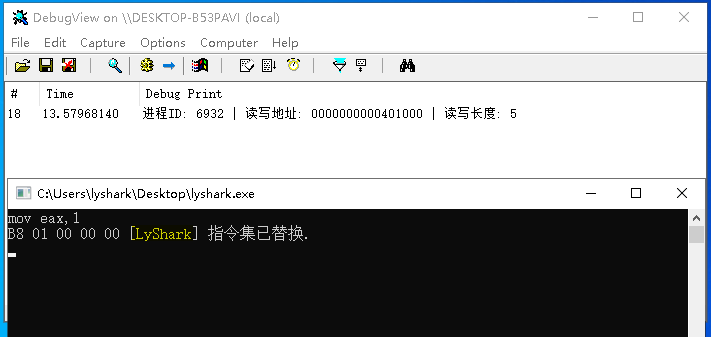
DeviceIoControl(handle, WRITE\_PROCESS\_CODE, &data, sizeof(data), &data, sizeof(data), &dwSize, NULL);

printf("[LyShark] 指令集已替换. \n"); getchar();

CloseHandle(handle); return 0;

}

# 通过驱动加载工具加载 WinDDK.sys 然后在运行本程序，你会看到正确的输出结果，可打开反内核工具验证是否改写成功。



打开反内核工具，并切换到观察是否写入了一条 mov eax,1 的指令集机器码，如下图已经完美写入。

