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
#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；

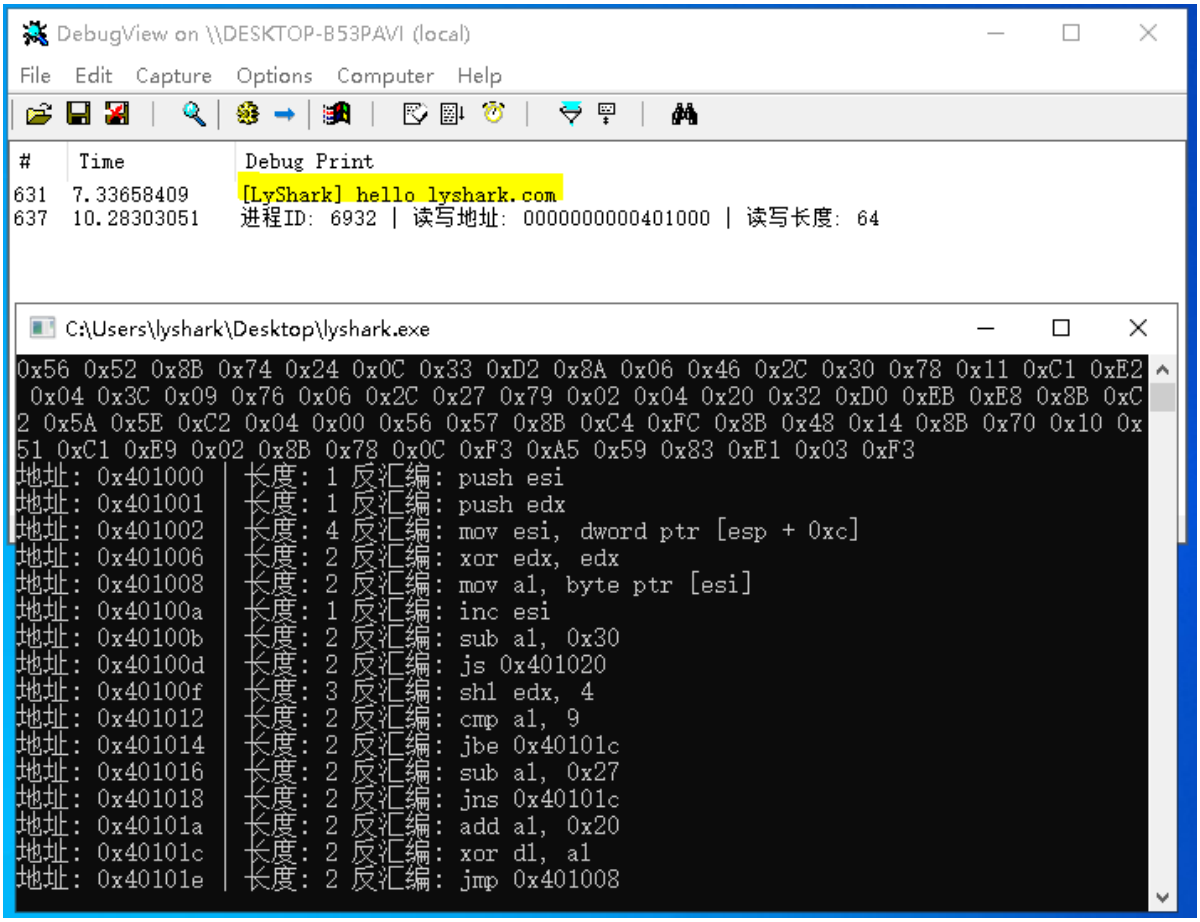
- 参数1：指定dasm_handle反汇编句柄
- 参数2：指定你要反汇编的数据集或者是一个缓冲区
- 参数3：指定你要反汇编的长度 64
- 参数4：输出的内存地址起始位置 0x401000
- 参数5：默认填充为0
- 参数6：用于输出数据的一个指针

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。



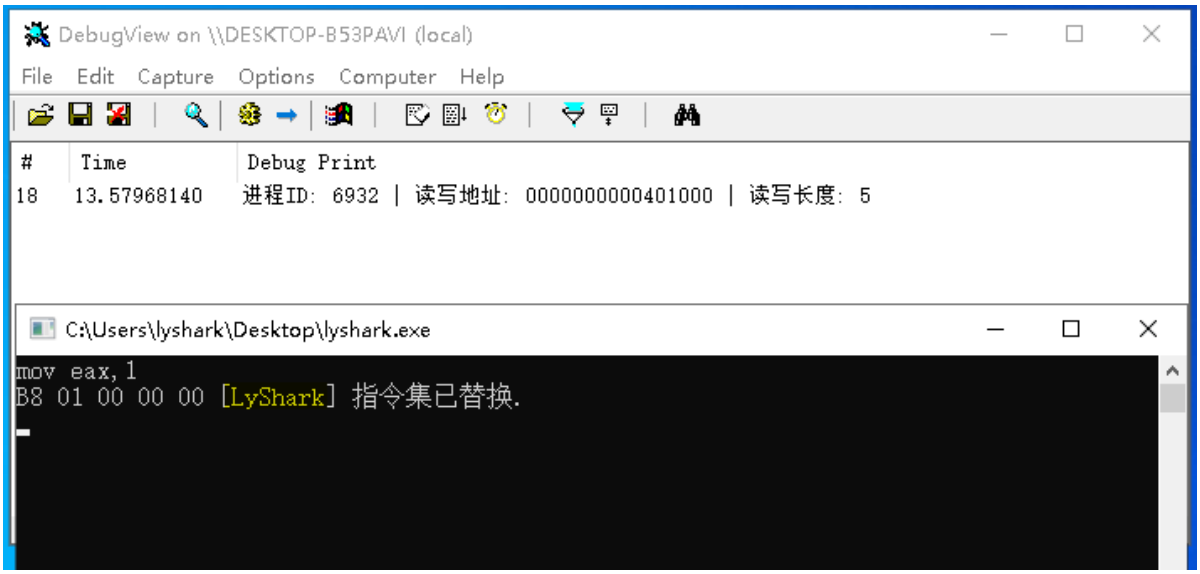
说完了反汇编接着就需要讲解如何对内存进行汇编操作了，汇编引擎这里采用了`XEDParse`该引擎小巧简洁，著名的`x64dbg`就是在运用本引擎进行汇编替换的，本引擎的使用非常简单，只需要向`XEDParseAssemble()`函数传入一个规范的结构体即可完成转换，完整代码如下所示。

- 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。