在笔者上一篇文章《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；

```
#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其中的关键部分已经做好了备注，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 内核反汇编

Capstone 是一款轻量级、多平台、多架构的反汇编引擎，旨在成为二进制分析和反汇编的终极工具。它支持多种平台和架构的反汇编，包括x86、ARM、MIPS等，并且可以轻松地集成到各种二进制分析工具中。Capstone的主要优点是它易于使用和快速的反汇编速度，而且由于其开源和活跃的社区支持，可以很容易地更新和维护。因此，Capstone被广泛用于二进制分析、安全研究和反汇编工作中。

- 反汇编引擎GitHub地址：https://github.com/capstone-engine

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()`打开一个句柄，这个打开功能的函数原型如下所示;

```
cs_err cs_open(
  cs_arch arch,
  cs_mode mode,
  csh *handle
);
```

- 参数 arch：指定架构类型，例如`CS_ARCH_X86`表示为 x86 架构。
- 参数 mode：指定模式，例如`CS_MODE_32`表示为 32 位模式。
- 参数 handle：打开的句柄，用于后续对引擎的调用。由于其是传递指针的方式，因此需要先分配好该指针的内存。函数执行成功后，该句柄将被填充，可以用于后续的反汇编操作。

函数`cs_open()`是`Capstone`反汇编引擎提供的，它用于初始化`Capstone`库并打开一个句柄，以便进行后续的反汇编操作。该函数有三个参数，分别是架构类型、执行模式和指向句柄的指针。

具体地说，第一个参数`CS_ARCH_X86`指定了反汇编的架构类型，这里表示为Windows平台；第二个参数`CS_MODE_32`或`CS_MODE_64`则指定了反汇编的执行模式，即32位模式或64位模式；第三个参数则是指向一个`Capstone`库句柄的指针，通过该指针可以进行后续的反汇编操作。

打开句柄后，我们可以使用其他的`Capstone`函数进行反汇编操作，比如`cs_disasm()`函数用于对二进制代码进行反汇编，反汇编后的结果可以用于分析和理解程序的行为。最后，我们还需要使用`cs_close()`函数关闭打开的句柄以释放资源。

第二步也是最重要的一步，调用`cs_disasm()`反汇编函数，函数返回实际反汇编的指令数，或者如果发生错误，则返回0。该函数的原型如下所示;

```
size_t cs_disasm(
  csh handle,
  const uint8_t *code,
  size_t code_size,
  uint64_t address,
  size_t count,
  cs_insn *insn
  );
```

其中各参数的含义为：

- 参数 handle：要使用的Capstone引擎的句柄，指定dasm_handle反汇编句柄
- 参数 code：要反汇编的二进制代码的指针，定你要反汇编的数据集或者是一个缓冲区
- 参数 code_size：要反汇编的二进制代码的大小（以字节为单位），指定你要反汇编的长度64
- 参数 address：要反汇编的二进制代码在内存中的地址（用于计算跳转目标地址），输出的内存地址起始位置0x401000
- 参数 count：要反汇编的指令数量限制。如果设置为0，则表示没有数量限制，将会反汇编所有有效的指令
- 参数 insn：用于存储反汇编结果的结构体数组。它是一个输出参数，由调用者分配内存。用于输出数据的一个指针

如上所示的`cs_open()`以及`cs_disasm()`两个函数如果能搞明白，那么反汇编完整代码即可写出来了，根据如下流程实现；

- 创建一个句柄`handle`，用于连接到驱动程序。
- 定义`ProcessData`结构体，包含需要读取的进程 ID、起始地址、读取的字节数以及存储读取结果的`BYTE`数组。
- 使用`DeviceIoControl()`函数从指定进程读取机器码，将结果存储到`data`结构体的`data`字段中。
- 使用`cs_open()`函数打开`Capstone`引擎的句柄`dasm_handle`，指定了架构为`x86`平台，模式为`32`位。
- 使用`cs_disasm()`函数将`data`结构体中的机器码进行反汇编，将结果存储到`insn`数组中，同时返回反汇编指令的数量 count。
- 循环遍历`insn`数组，将每个反汇编指令的地址、长度、助记符和操作数打印出来。
- 使用`cs_free()`函数释放`insn`数组占用的内存。
- 使用`cs_close()`函数关闭`Capstone`引擎的句柄 dasm_handle。
- 关闭连接到驱动程序的句柄`handle`。

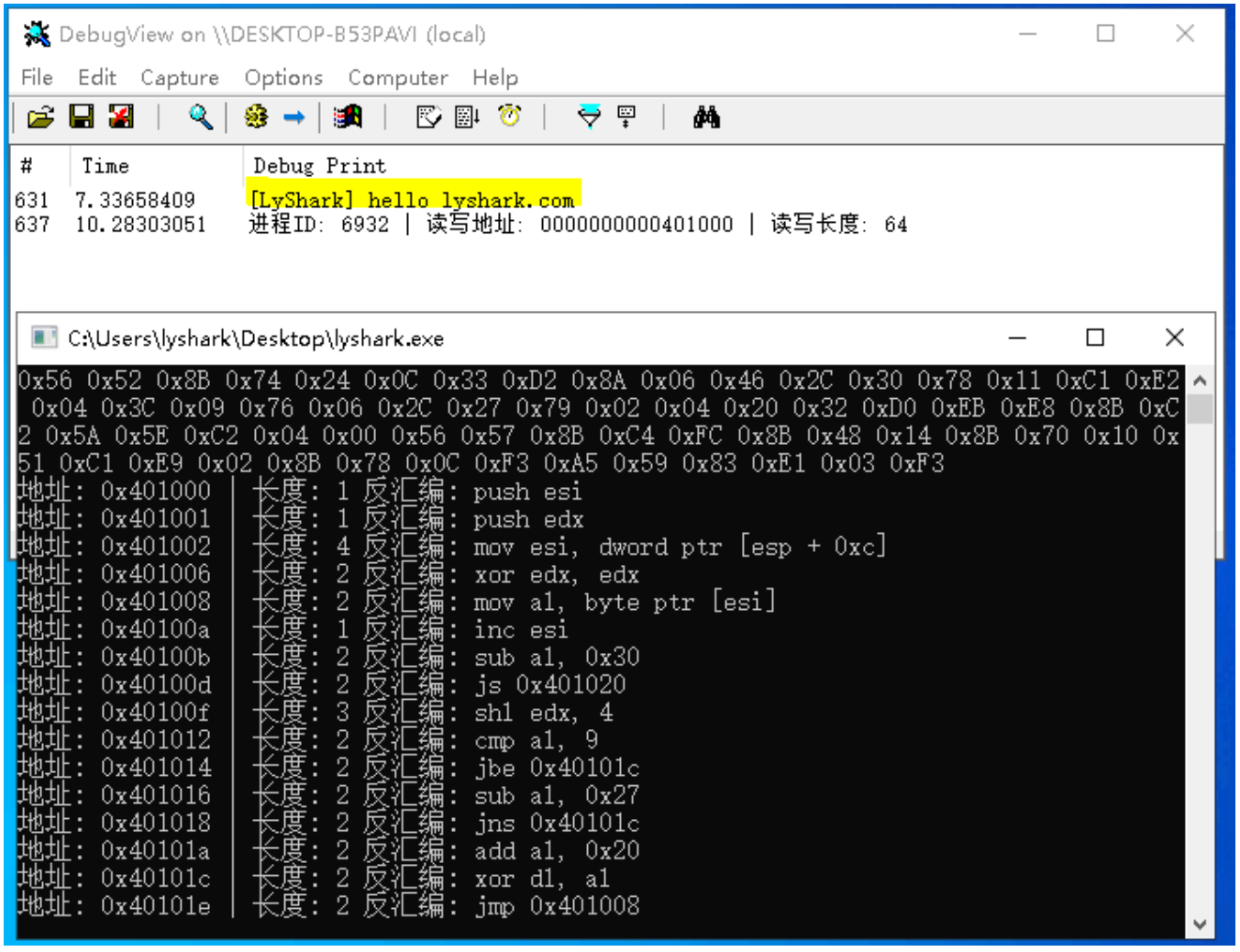
根据如上实现流程，我们可以写出如下代码片段；

```
#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。



XEDParse 内核汇编

实现了反汇编接着就需要讲解如何对内存进行汇编操作，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，XEDParse 是一个开源的汇编引擎，用于将汇编代码转换为二进制指令。它基于`Intel`的XED库，并提供了一些易于使用的接口。

- 汇编引擎GitHub地址：https://github.com/x64dbg/XEDParse

一般而言，再进行汇编转换之前需要做如下几个步骤的工作；

1.定义`xed_state_t`结构体，该结构体包含有关目标平台的信息，例如处理器架构和指令集。可以使用`xed_state_zero()`函数来初始化该结构体。

```
xed_state_t state;
xed_state_zero(&state);
state.mmode = XED_MACHINE_MODE_LONG_64;
state.stack_addr_width = XED_ADDRESS_WIDTH_64b;
```

2.定义`xed_error_enum_t`类型的变量来接收转换过程中可能出现的错误信息。

```
xed_error_enum_t error = XED_ERROR_NONE;
```

3.定义`xed_encoder_request_t`结构体，该结构体包含要转换的汇编指令的信息，例如操作码和操作数。

```
xed_encoder_request_t request;
xed_encoder_request_zero_set_mode(&request, &state);
request.iclass = XED_ICLASS_MOV;
request.operand_order[0] = 0;
request.operand_order[1] = 1;
request.operands[0].name = XED_REG_RAX;
request.operands[1].name = XED_REG_RBX;
```

4.使用`XEDParseAssemble()`函数将汇编代码转换为二进制指令，并将结果存储在`xed_uint8_t`类型的数组中。此函数返回转换后的指令长度。

```
xed_uint8_t binary[15];
xed_uint_t length = XEDParseAssemble(&request, binary, sizeof(binary), &error);
if (error != XED_ERROR_NONE) {
  // handle error
}
```

5.使用转换后的二进制指令进行后续操作。

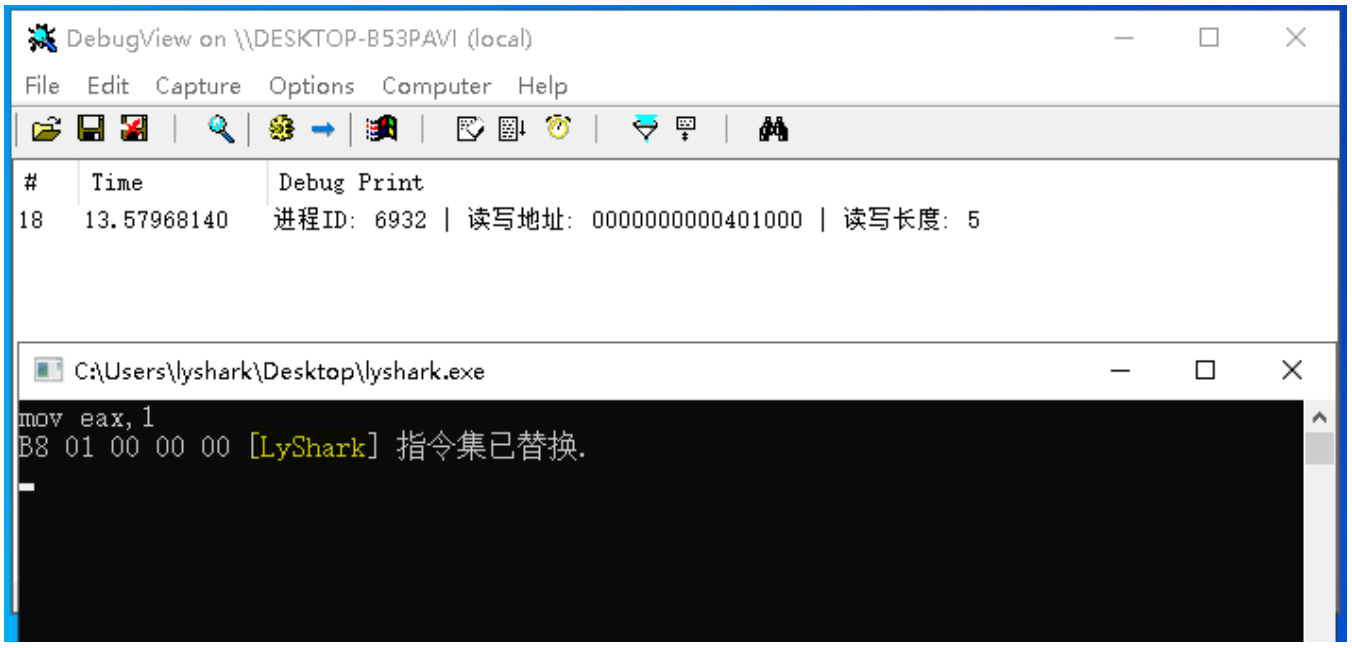
```
typedef int (*func_t)(void);
func_t func = (func_t)binary;
int result = func();
```

在本次转换流程中我们只需要向`XEDParseAssemble()`函数传入一个规范的结构体即可完成转换，通过向`XEDPARSE`结构传入需要转换的指令，并自动转换为机器码放入到`data.data`堆中，实现核心代码如下所示；

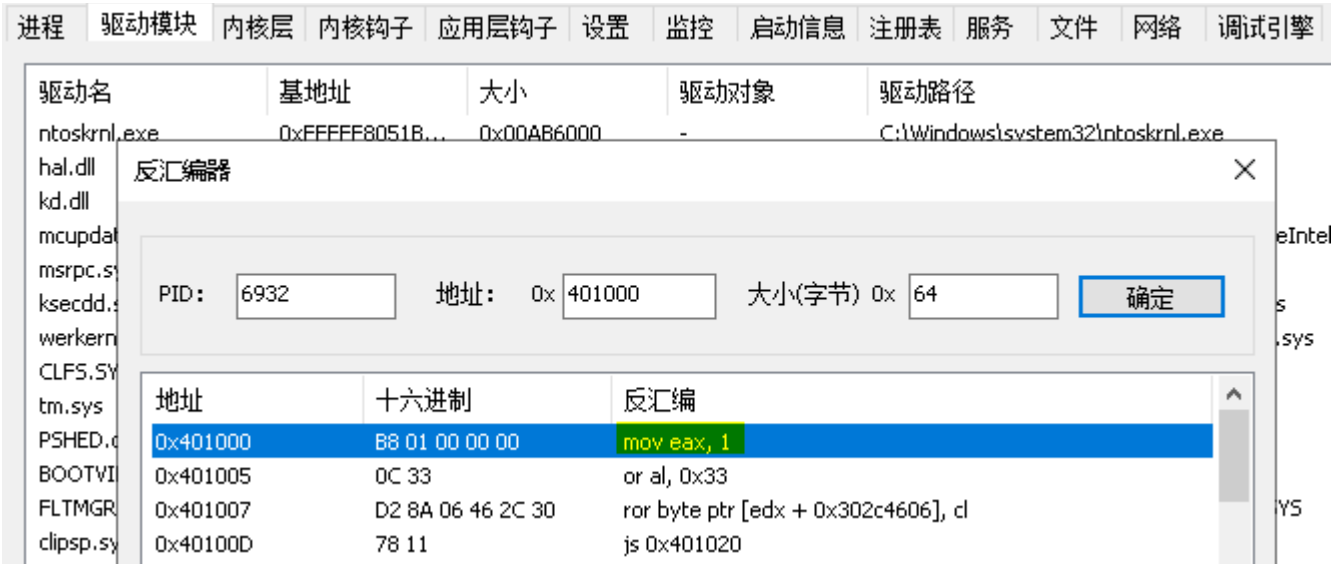
```
#define _CRT_SECURE_NO_WARNINGS
#include <Windows.h>
#include <iostream>

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。


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
驱动名
基地址
大小
驱动对象
驱动路径
ntoskrnl.exe
0xFFFFF8051B...
0x00AB6000
-
C:\Windows\system32\ntoskrnl.exe
hal.dll
kd.dll
反汇编器
PID:
6932
地址:
0x
401000
大小(字节) 0x
64
确定
地址
十六进制
反汇编
0x401000
B8 01 00 00 00
mov eax, 1
0x401005
0C 33
or al, 0x33
0x401007
D2 8A 06 46 2C 30
ror byte ptr [edx + 0x302c4606], cl
0x40100D
78 11
js 0x401020