远程线程注入是最常用的一种注入技术，在应用层注入是通过`CreateRemoteThread`这个函数实现的，该函数通过创建线程并调用`LoadLibrary`动态载入指定的DLL来实现注入，而在内核层同样存在一个类似的内核函数`RtlCreateUserThread`，但需要注意的是此函数未被公开，`RtlCreateUserThread`其实是对`NtCreateThreadEx`的包装，但最终会调用`ZwCreateThread`来实现注入，`RtlCreateUserThread`是`CreateRemoteThread`的底层实现。

内核级别的LoadLibrary实现DLL注入的过程与用户级别的LoadLibrary实现DLL注入的过程类似，只不过是在内核模式下实现。具体而言，实现内核级别的LoadLibrary实现DLL注入。

基于LoadLibrary实现的注入原理具体实现分为如下几步；

- 1.调用`AllocMemory`,在对端应用层开辟空间，函数封装来源于`《内核远程堆分配与销毁》`章节；
- 2.调用`MDLWriteMemory`，将DLL路径字符串写出到对端内存，函数封装来源于`《内核MDL读写进程内存》`章节；
- 3.调用`GetUserModuleAddress`，获取到`kernel32.dll`模块基址，函数封装来源于`《内核远程线程实现DLL注入》`章节；
- 4.调用`GetModuleExportAddress`，获取到`LoadLibraryW`函数的内存地址，函数封装来源于`《内核远程线程实现DLL注入》`章节；
- 5.最后调用本章封装函数`MyCreateRemoteThread`，将应用层DLL动态转载到进程内，实现DLL注入；

总结起来就是首先在目标进程申请一块空间，空间里面写入要注入的DLL的路径字符串或者是一段ShellCode，找到该内存中`LoadLibrary`的基址并传入到`RtlCreateUserThread`中，此时进程自动加载我们指定路径下的DLL文件。

注入依赖于`RtlCreateUserThread`这个未到处内核函数，该内核函数中最需要关心的参数是`ProcessHandle`用于接收进程句柄，`StartAddress`接收一个函数地址，`StartParameter`用于对函数传递参数，具体的函数原型如下所示；

```
typedef DWORD(WINAPI* pRtlCreateUserThread)(
    IN HANDLE                   ProcessHandle,          // 进程句柄
    IN PSECURITY_DESCRIPTOR     SecurityDescriptor,
    IN BOOL                     CreateSuspended,
    IN ULONG                    StackZeroBits,
    IN OUT PULONG               StackReserved,
    IN OUT PULONG               StackCommit,
    IN LPVOID                   StartAddress,          // 执行函数地址 LoadLibraryW
    IN LPVOID                   StartParameter,        // 参数传递
    OUT HANDLE                  ThreadHandle,          // 线程句柄
    OUT LPVOID                  ClientID
    );
```

由于我们加载DLL使用的是`LoadLibraryW`函数，此函数在运行时只需要一个参数，我们可以将DLL的路径传递进去，并调用`LoadLibraryW`以此来将特定模块拉起，该函数的定义规范如下所示；

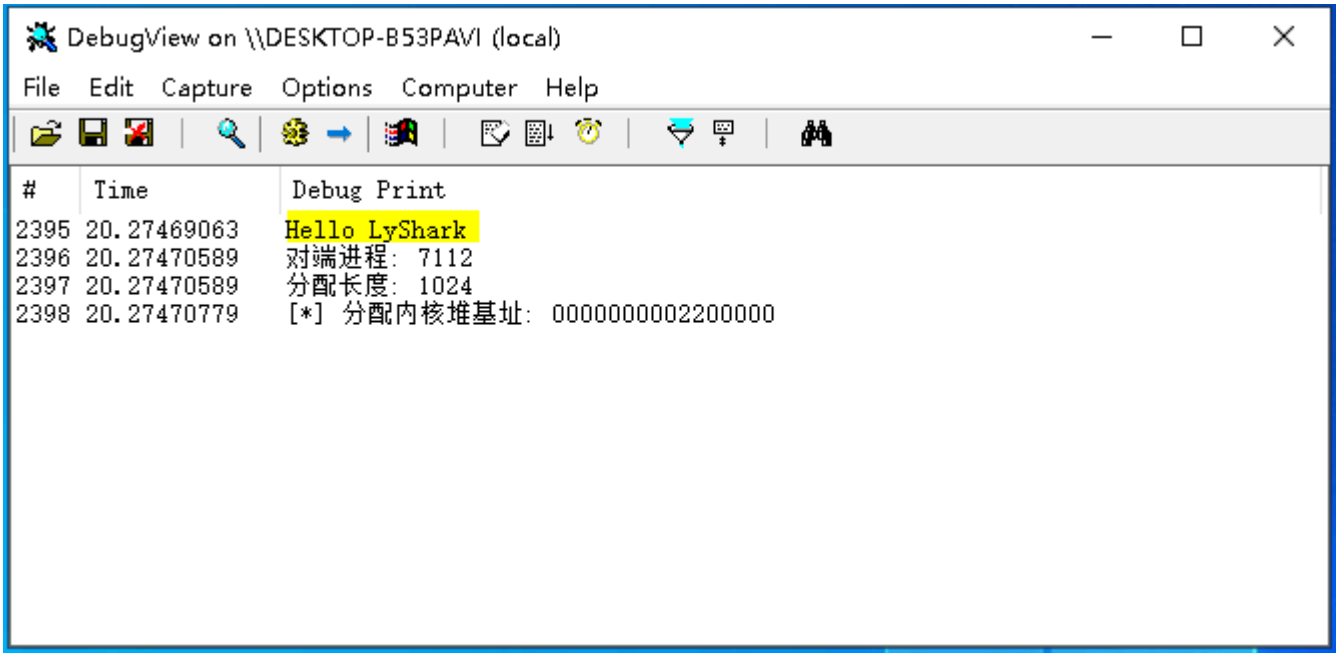
```
HMODULE LoadLibraryW(
  [in] LPCWSTR lpLibFileName
);
```

根据上一篇文章中针对注入头文件`lyshark.h`的封装，本章将继续使用这个头文件中的函数，首先我们实现这样一个功能，将一段准备好的`UCHAR`字符串动态的写出到应用层进程内存，并以宽字节模式写出在对端内存中，这段代码可以写为如下样子；

```
#include "lyshark.h"

// 驱动卸载例程
VOID UnDriver(PDRIVER_OBJECT driver)
{
  DbgPrint("Uninstall Driver \n");
}

// 驱动入口地址
NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
  DbgPrint("Hello LyShark \n");

  DWORD process_id = 7112;
  DWORD create_size = 1024;
  DWORD64 ref_address = 0;

  // 分配内存堆 《内核远程堆分配与销毁》 核心代码
  NTSTATUS Status = AllocMemory(process_id, create_size, &ref_address);

  DbgPrint("对端进程: %d \n", process_id);
  DbgPrint("分配长度: %d \n", create_size);
  DbgPrint("[*] 分配内核堆基址: %p \n", ref_address);

  UCHAR DllPath[256] = "C:\\hook.dll";
  UCHAR Item[256] = { 0 };

  // 将字节转为双字
  for (int x = 0, y = 0; x < strlen(DllPath) * 2; x += 2, y++)
  {
    Item[x] = DllPath[y];
  }

  // 写出内存 《内核MDL读写进程内存》 核心代码
  ReadMemoryStruct ptr;

  ptr.pid = process_id;
  ptr.address = ref_address;
  ptr.size = strlen(DllPath) * 2;

  // 需要写入的数据
  ptr.data = ExAllocatePool(PagedPool, ptr.size);

  // 循环设置
  for (int i = 0; i < ptr.size; i++)
  {
    ptr.data[i] = Item[i];
  }
```

```
    // 写内存
    MDLWriteMemory(&ptr);

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

运行如上方所示的代码，将会在目标进程 `7112` 中开辟一段内存空间，并写出 `C:\hook.dll` 字符串，运行效果图如下所示；



此处你可以通过 `x64dbg` 附加到应用层进程内，并观察内存 `0000000002200000` 会看到如下字符串已被写出，双字类型则是每一个字符空一格，效果图如下所示；

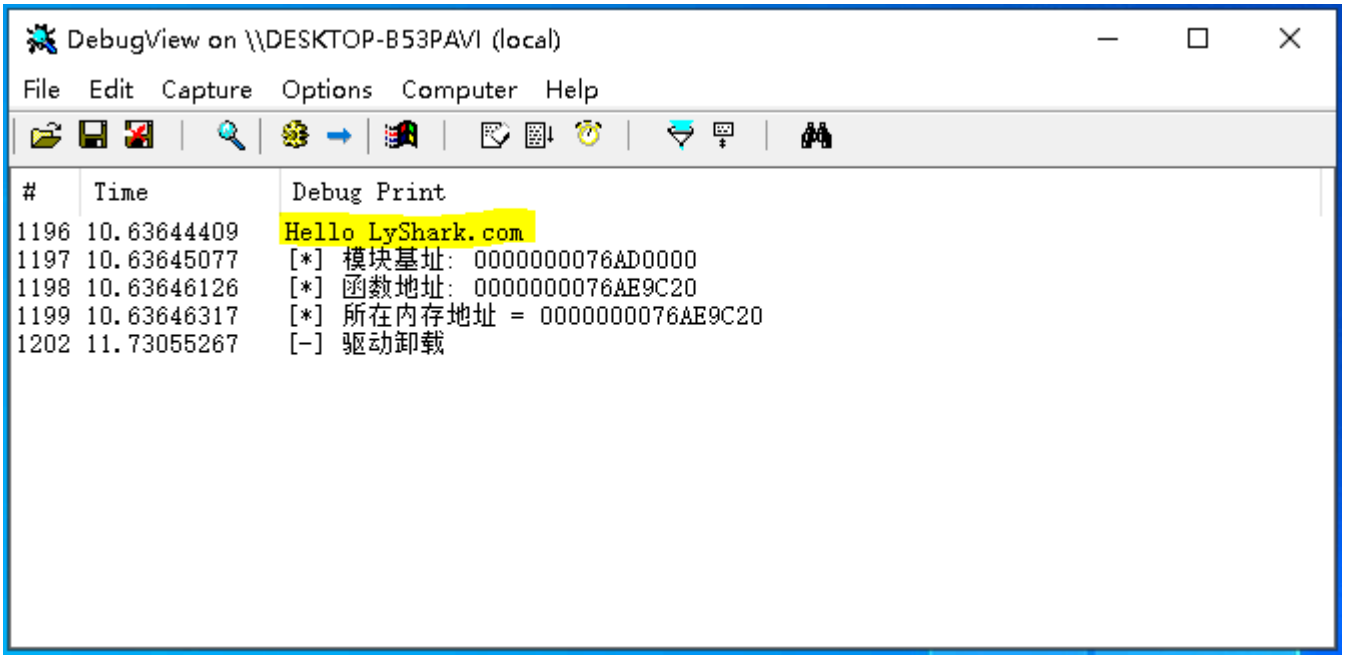
继续实现所需要的子功能，实现动态获取 `Kernel32.dll` 模块里面 `LiadLibraryW` 这个导出函数的内存地址，这段代码相信你可以很容易的写出来，根据上节课的知识点我们可以二次封装一个 `GetProcessAddress` 来实现对特定模块基址的获取功能，如下是完整代码案例；

```
#include "lyshark.h"

// 实现取模块基址
PVOID GetProcessAddress(HANDLE ProcessID, PWCHAR DllName, PCCHAR FunctionName)
{
    PEPROCESS EProcess = NULL;
    NTSTATUS Status = STATUS_SUCCESS;
    KAPC_STATE ApcState;
    PVOID RefAddress = 0;

    // 根据PID得到进程EProcess结构
```

```
  Status = PsLookupProcessByProcessId(ProcessID, &EProcess);
  if (Status != STATUS_SUCCESS)
  {
    return Status;
  }

  // 判断目标进程是32位还是64位
  BOOLEAN IsWow64 = (PsGetProcessWow64Process(EProcess) != NULL) ? TRUE : FALSE;

  // 验证地址是否可读
  if (!MmIsAddressValid(EProcess))
  {
    return NULL;
  }

  // 将当前线程连接到目标进程的地址空间(附加进程)
  KeStackAttachProcess((PRKPROCESS)EProcess, &ApcState);

  __try
  {
    UNICODE_STRING DllUnicodeString = { 0 };
    PVOID BaseAddress = NULL;

    // 得到进程内模块基地址
    RtlInitUnicodeString(&DllUnicodeString, DllName);

    BaseAddress = GetUserModuleAddress(EProcess, &DllUnicodeString, IsWow64);

    if (!BaseAddress)
    {
      return NULL;
    }

    DbgPrint("[*] 模块基址: %p \n", BaseAddress);

    // 得到该函数地址
    RefAddress = GetModuleExportAddress(BaseAddress, FunctionName, EProcess);
    DbgPrint("[*] 函数地址: %p \n", RefAddress);
  }
  __except (EXCEPTION_EXECUTE_HANDLER)
  {
    return NULL;
  }

  // 取消附加
  KeUnstackDetachProcess(&ApcState);
  return RefAddress;
}

VOID Unload(PDRIVER_OBJECT pDriverObj)
{
  DbgPrint("[-] 驱动卸载 \n");
}
```

```
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegPath)
{
  DbgPrint("Hello LyShark.com \n");

  // 取模块基址
  PVOID pLoadLibraryW = GetProcessAddress(5200, L"kernel32.dll", "LoadLibraryW");

  DbgPrint("[*] 所在内存地址 = %p \n", pLoadLibraryW);

  DriverObject->DriverUnload = Unload;
  return STATUS_SUCCESS;
}
```

编译并运行如上驱动代码，将自动获取 `PID=5200` 进程中 `Kernel32.dll` 模块内的 `LoadLibraryW` 的内存地址，输出效果图如下所示；



实现注入的最后一步就是调用自定义函数 `MyCreateRemoteThread` 该函数实现原理是调用 `RtlCreateUserThread` 开线程执行，这段代码的最终实现如下所示；

```
#include "lyshark.h"

// 定义函数指针
typedef PVOID(NTAPI* PfnRtlCreateUserThread)
(
  IN HANDLE ProcessHandle,
  IN PSECURITY_DESCRIPTOR SecurityDescriptor,
  IN BOOLEAN CreateSuspended,
  IN ULONG StackZeroBits,
  IN OUT size_t StackReserved,
  IN OUT size_t StackCommit,
  IN PVOID StartAddress,
  IN PVOID StartParameter,
  OUT PHANDLE ThreadHandle,
  OUT PCLIENT_ID ClientID
```

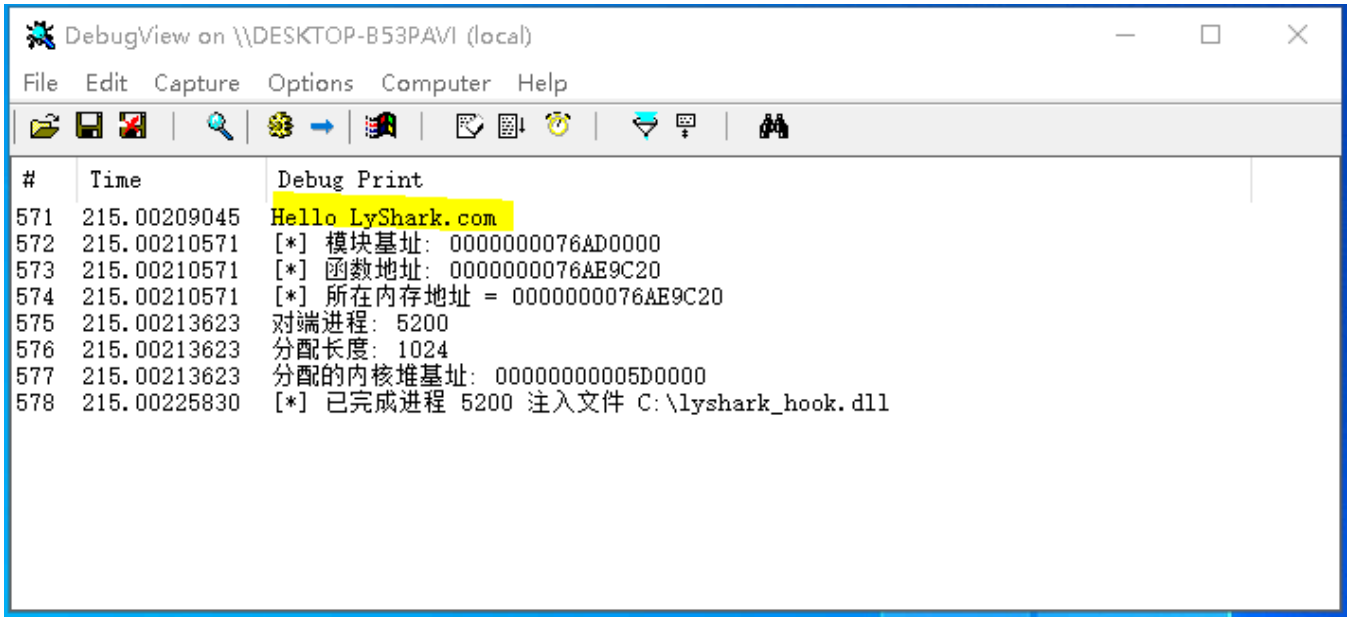
```
);

// 实现取模块基址
PVOID GetProcessAddress(HANDLE ProcessID, PWCHAR DllName, PCCHAR FunctionName)
{
  PEPROCESS EProcess = NULL;
  NTSTATUS Status = STATUS_SUCCESS;
  KAPC_STATE ApcState;
  PVOID RefAddress = 0;

  // 根据PID得到进程EProcess结构
  Status = PsLookupProcessByProcessId(ProcessID, &EProcess);
  if (Status != STATUS_SUCCESS)
  {
    return Status;
  }

  // 判断目标进程是32位还是64位
  BOOLEAN IsWow64 = (PsGetProcessWow64Process(EProcess) != NULL) ? TRUE : FALSE;

  // 验证地址是否可读
  if (!MmIsAddressValid(EProcess))
  {
    return NULL;
  }

  // 将当前线程连接到目标进程的地址空间(附加进程)
  KeStackAttachProcess((PRKPROCESS)EProcess, &ApcState);

  __try
  {
    UNICODE_STRING DllUnicodeString = { 0 };
    PVOID BaseAddress = NULL;

    // 得到进程内模块基地址
    RtlInitUnicodeString(&DllUnicodeString, DllName);

    BaseAddress = GetUserModuleAddress(EProcess, &DllUnicodeString, IsWow64);

    if (!BaseAddress)
    {
      return NULL;
    }

    DbgPrint("[*] 模块基址: %p \n", BaseAddress);

    // 得到该函数地址
    RefAddress = GetModuleExportAddress(BaseAddress, FunctionName, EProcess);
    DbgPrint("[*] 函数地址: %p \n", RefAddress);
  }
  __except (EXCEPTION_EXECUTE_HANDLER)
  {
    return NULL;
```

```
  }

  // 取消附加
  KeUnstackDetachProcess(&ApcState);
  return RefAddress;
}

// 远程线程注入函数
BOOLEAN MyCreateRemoteThread(ULONG pid, PVOID pRing3Address, PVOID PParam)
{
  NTSTATUS status = STATUS_UNSUCCESSFUL;
  PEPROCESS pEProcess = NULL;
  KAPC_STATE ApcState = { 0 };

  PfnRtlCreateUserThread RtlCreateUserThread = NULL;
  HANDLE hThread = 0;

  __try
  {
    // 获取RtlCreateUserThread函数的内存地址
    UNICODE_STRING ustrRtlCreateUserThread;
    RtlInitUnicodeString(&ustrRtlCreateUserThread, L"RtlCreateUserThread");
    RtlCreateUserThread =
(PfnRtlCreateUserThread)MmGetSystemRoutineAddress(&ustrRtlCreateUserThread);
    if (RtlCreateUserThread == NULL)
    {
      return FALSE;
    }

    // 根据进程PID获取进程EProcess结构
    status = PsLookupProcessByProcessId((HANDLE)pid, &pEProcess);
    if (!NT_SUCCESS(status))
    {
      return FALSE;
    }

    // 附加到目标进程内
    KeStackAttachProcess(pEProcess, &ApcState);

    // 验证进程是否可读写
    if (!MmIsAddressValid(pRing3Address))
    {
      return FALSE;
    }

    // 启动注入线程
    status = RtlCreateUserThread(ZwCurrentProcess(),
      NULL,
      FALSE,
      0,
      0,
      0,
      pRing3Address,
```

```
      PParam,
      &hThread,
      NULL);
    if (!NT_SUCCESS(status))
    {
      return FALSE;
    }

    return TRUE;
  }

  __finally
  {
    // 释放对象
    if (pEProcess != NULL)
    {
      ObDereferenceObject(pEProcess);
      pEProcess = NULL;
    }

    // 取消附加进程
    KeUnstackDetachProcess(&ApcState);
  }

  return FALSE;
}

VOID Unload(PDRIVER_OBJECT pDriverObj)
{
  DbgPrint("[-] 驱动卸载 \n");
}

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegPath)
{
  DbgPrint("Hello LyShark.com \n");

  ULONG process_id = 5200;
  DWORD create_size = 1024;
  DWORD64 ref_address = 0;

  // ------------------------------------------------------
  // 取模块基址
  // ------------------------------------------------------

  PVOID pLoadLibraryW = GetProcessAddress(process_id, L"kernel32.dll", "LoadLibraryW");
  DbgPrint("[*] 所在内存地址 = %p \n", pLoadLibraryW);

  // ------------------------------------------------------
  // 应用层开堆
  // ------------------------------------------------------

  NTSTATUS Status = AllocMemory(process_id, create_size, &ref_address);
```

```
    DbgPrint("对端进程: %d \n", process_id);
    DbgPrint("分配长度: %d \n", create_size);
    DbgPrint("分配的内核堆基址: %p \n", ref_address);

    // 设置注入路径,转换为多字节
    UCHAR DllPath[256] = "C:\\lyshark_hook.dll";
    UCHAR Item[256] = { 0 };

    for (int x = 0, y = 0; x < strlen(DllPath) * 2; x += 2, y++)
    {
      Item[x] = DllPath[y];
    }

    // ------------------------------------------------------
    // 写出数据到内存
    // ------------------------------------------------------

    ReadMemoryStruct ptr;

    ptr.pid = process_id;
    ptr.address = ref_address;
    ptr.size = strlen(DllPath) * 2;

    // 需要写入的数据
    ptr.data = ExAllocatePool(PagedPool, ptr.size);

    // 循环设置
    for (int i = 0; i < ptr.size; i++)
    {
      ptr.data[i] = Item[i];
    }

    // 写内存
    MDLWriteMemory(&ptr);

    // ------------------------------------------------------
    // 执行开线程函数
    // ------------------------------------------------------

    // 执行线程注入
    // 参数1: PID
    // 参数2: LoadLibraryW内存地址
    // 参数3: 当前DLL路径
    BOOLEAN flag = MyCreateRemoteThread(process_id, pLoadLibraryW, ref_address);
    if (flag == TRUE)
    {
      DbgPrint("[*] 已完成进程 %d 注入文件 %s \n", process_id, DllPath);
    }

    DriverObject->DriverUnload = Unload;
    return STATUS_SUCCESS;
}
```

编译这段驱动程序，并将其放入虚拟机中，在C盘下面放置好一个名为 `lyshark_hook.dll` 文件，运行驱动程序将自动插入DLL到 `Win32Project` 进程内，输出效果图如下所示；



回到应用层进程，则可看到如下图所示的注入成功提示信息；