在笔者上一篇文章《内核取应用层模块基地址》中简单为大家介绍了如何通过遍历PLIST_ENTRY32链表的方式获取到32位应用程序中特定模块的基地址，由于是入门系列所以并没有封装实现太过于通用的获取函数，本章将继续延申这个话题，并依次实现通用版GetUserModuleBaseAddress()取远程进程中指定模块的基址和GetModuleExportAddress()取远程进程中特定模块中的函数地址，此类功能也是各类安全工具中常用的代码片段。

首先封装一个lyshark.h头文件，此类头文件中的定义都是微软官方定义好的规范，如果您想获取该结构的详细说明文档请参阅微软官方，此处不做过多的介绍。

```
#include <ntifs.h>
#include <ntimage.h>
#include <ntstrsafe.h>

// 导出未导出函数
NTKERNELAPI PPEB NTAPI PsGetProcessPeb(IN PEPROCESS Process);
NTKERNELAPI PVOID NTAPI PsGetProcessWow64Process(IN PEPROCESS Process);

typedef struct _PEB32
{
  UCHAR InheritedAddressSpace;
  UCHAR ReadImageFileExecOptions;
  UCHAR BeingDebugged;
  UCHAR BitField;
  ULONG Mutant;
  ULONG ImageBaseAddress;
  ULONG Ldr;
  ULONG ProcessParameters;
  ULONG SubSystemData;
  ULONG ProcessHeap;
  ULONG FastPebLock;
  ULONG AtlThunkSListPtr;
  ULONG IFEOKey;
  ULONG CrossProcessFlags;
  ULONG UserSharedInfoPtr;
  ULONG SystemReserved;
  ULONG AtlThunkSListPtr32;
  ULONG ApiSetMap;
} PEB32, *PPEB32;

typedef struct _PEB_LDR_DATA
{
  ULONG Length;
  UCHAR Initialized;
  PVOID SsHandle;
  LIST_ENTRY InLoadOrderModuleList;
  LIST_ENTRY InMemoryOrderModuleList;
  LIST_ENTRY InInitializationOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;

typedef struct _PEB
{
  UCHAR InheritedAddressSpace;
  UCHAR ReadImageFileExecOptions;
```

```
  UCHAR BeingDebugged;
  UCHAR BitField;
  PVOID Mutant;
  PVOID ImageBaseAddress;
  PPEB_LDR_DATA Ldr;
  PVOID ProcessParameters;
  PVOID SubSystemData;
  PVOID ProcessHeap;
  PVOID FastPebLock;
  PVOID AtlThunkSListPtr;
  PVOID IFEOKey;
  PVOID CrossProcessFlags;
  PVOID KernelCallbackTable;
  ULONG SystemReserved;
  ULONG AtlThunkSListPtr32;
  PVOID ApiSetMap;
} PEB, *PPEB;

typedef struct _PEB_LDR_DATA32
{
  ULONG Length;
  UCHAR Initialized;
  ULONG SsHandle;
  LIST_ENTRY32 InLoadOrderModuleList;
  LIST_ENTRY32 InMemoryOrderModuleList;
  LIST_ENTRY32 InInitializationOrderModuleList;
} PEB_LDR_DATA32, *PPEB_LDR_DATA32;

typedef struct _LDR_DATA_TABLE_ENTRY32
{
  LIST_ENTRY32 InLoadOrderLinks;
  LIST_ENTRY32 InMemoryOrderLinks;
  LIST_ENTRY32 InInitializationOrderLinks;
  ULONG DllBase;
  ULONG EntryPoint;
  ULONG SizeOfImage;
  UNICODE_STRING32 FullDllName;
  UNICODE_STRING32 BaseDllName;
  ULONG Flags;
  USHORT LoadCount;
  USHORT TlsIndex;
  LIST_ENTRY32 HashLinks;
  ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY32, *PLDR_DATA_TABLE_ENTRY32;

typedef struct _LDR_DATA_TABLE_ENTRY
{
  LIST_ENTRY InLoadOrderLinks;
  LIST_ENTRY InMemoryOrderLinks;
  LIST_ENTRY InInitializationOrderLinks;
  PVOID DllBase;
  PVOID EntryPoint;
  ULONG SizeOfImage;
```

```
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    ULONG Flags;
    USHORT LoadCount;
    USHORT TlsIndex;
    LIST_ENTRY HashLinks;
    ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

取进程中模块基址

实现取进程中模块基址，该功能在`《内核取应用层模块基地址》`中详细介绍过原理，这段代码核心原理如下所示，此处最需要注意的是如果是`32位进程`则我们需要得到`PPEB32 Peb32`结构体，该结构体通常可以直接使用`PsGetProcessWow64Process()`这个内核函数获取到，而如果是`64位进程`则需要将寻找PEB的函数替换为`PsGetProcessPeb()`，其他的枚举细节与上一篇文章中的方法一致。

```
#include <ntifs.h>
#include <windef.h>
#include "lyshark.h"

// 获取特定进程内特定模块的基址
PVOID GetUserModuleBaseAddress(IN PEPROCESS EProcess, IN PUNICODE_STRING ModuleName, IN
BOOLEAN IsWow64)
{
    if (EProcess == NULL)
        return NULL;
    __try
    {
        // 设置延迟时间为250毫秒
        LARGE_INTEGER Time = { 0 };
        Time.QuadPart = -250ll * 10 * 1000;

        // 如果是32位则执行如下代码
        if (IsWow64)
        {
            // 得到PEB进程信息
            PPEB32 Peb32 = (PPEB32)PsGetProcessWow64Process(EProcess);
            if (Peb32 == NULL)
            {
                return NULL;
            }

            // 延迟加载等待时间
            for (INT i = 0; !Peb32->Ldr && i < 10; i++)
            {
                KeDelayExecutionThread(KernelMode, TRUE, &Time);
            }

            // 没有PEB加载超时
            if (!Peb32->Ldr)
            {
                return NULL;
```

```
            }

            // 搜索模块 InLoadOrderModuleList
            for (PLIST_ENTRY32 ListEntry = (PLIST_ENTRY32)((PPEB_LDR_DATA32)Peb32->Ldr)->InLoadOrderModuleList.Flink; ListEntry != &((PPEB_LDR_DATA32)Peb32->Ldr)->InLoadOrderModuleList; ListEntry = (PLIST_ENTRY32)ListEntry->Flink)
            {
                UNICODE_STRING UnicodeString;
                PLDR_DATA_TABLE_ENTRY32 LdrDataTableEntry32 = CONTAINING_RECORD(ListEntry, LDR_DATA_TABLE_ENTRY32, InLoadOrderLinks);
                RtlUnicodeStringInit(&UnicodeString, (PWCH)LdrDataTableEntry32->BaseDllName.Buffer);

                // 找到了返回模块基址
                if (RtlCompareUnicodeString(&UnicodeString, ModuleName, TRUE) == 0)
                {
                    return (PVOID)LdrDataTableEntry32->DllBase;
                }
            }
        }
        // 如果是64位则执行如下代码
        else
        {
            // 同理,先找64位PEB
            PPEB Peb = PsGetProcessPeb(EProcess);
            if (!Peb)
            {
                return NULL;
            }

            // 延迟加载
            for (INT i = 0; !Peb->Ldr && i < 10; i++)
            {
                KeDelayExecutionThread(KernelMode, TRUE, &Time);
            }

            // 找不到PEB直接返回
            if (!Peb->Ldr)
            {
                return NULL;
            }

            // 遍历链表
            for (PLIST_ENTRY ListEntry = Peb->Ldr->InLoadOrderModuleList.Flink; ListEntry != &Peb->Ldr->InLoadOrderModuleList; ListEntry = ListEntry->Flink)
            {
                // 将特定链表转换为PLDR_DATA_TABLE_ENTRY格式
                PLDR_DATA_TABLE_ENTRY LdrDataTableEntry = CONTAINING_RECORD(ListEntry, LDR_DATA_TABLE_ENTRY, InLoadOrderLinks);

                // 找到了则返回地址
                if (RtlCompareUnicodeString(&LdrDataTableEntry->BaseDllName, ModuleName, TRUE) == 0)
```

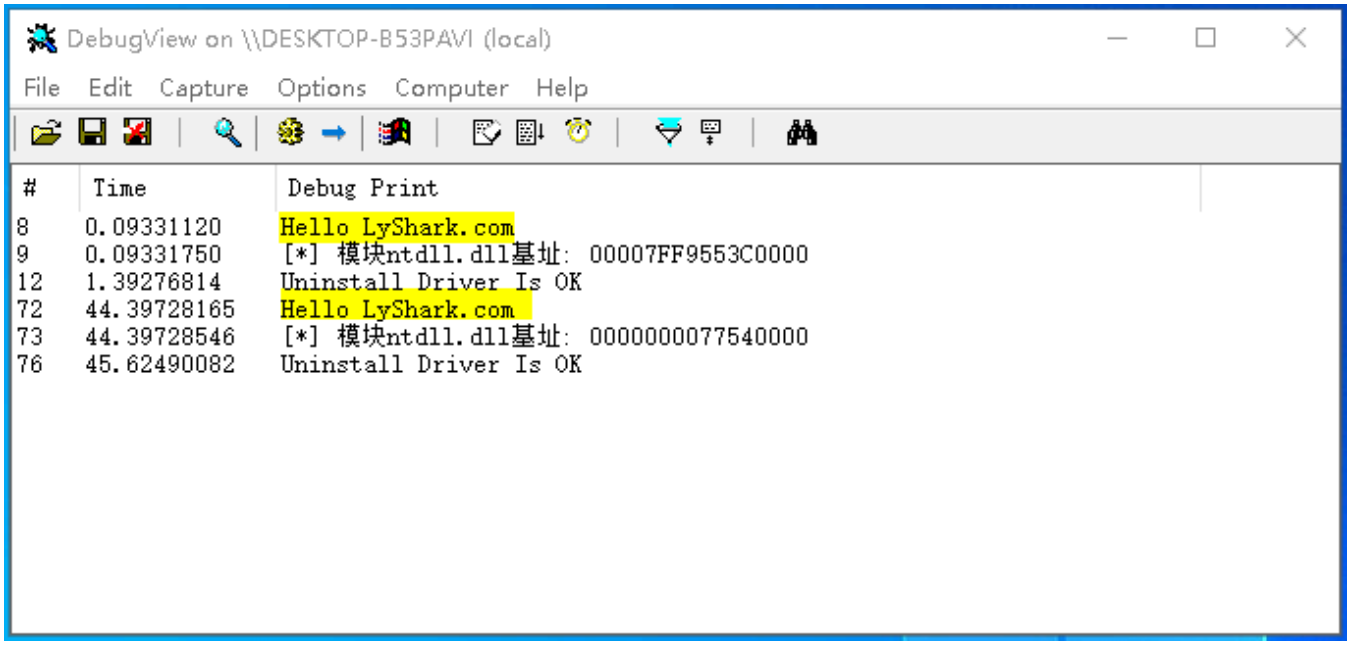
```
                {
                    return LdrDataTableEntry->DllBase;
                }
            }
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
        return NULL;
    }
    return NULL;
}
```

那么该函数该如何调用传递参数呢，如下代码是`DriverEntry`入口处的调用方法，首先要想得到特定进程的特定模块地址则第一步就是需要`PsLookupProcessByProcessId`找到模块的`EProcess`结构，接着通过`PsGetProcessWow64Process`得到当前被操作进程是32位还是64位，通过调用`KeStackAttachProcess`附加到进程内存中，然后调用`GetUserModuleBaseAddress`并传入需要获取模块的名字得到数据后返回给`NtdllAddress`变量，最后调用`KeUnstackDetachProcess`取消附加即可。

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    HANDLE ProcessID = (HANDLE)7924;

    PEPROCESS EProcess = NULL;
    NTSTATUS Status = STATUS_SUCCESS;
    KAPC_STATE ApcState;

    DbgPrint("Hello LyShark.com \n");

    // 根据PID得到进程EProcess结构
    Status = PsLookupProcessByProcessId(ProcessID, &EProcess);
    if (Status != STATUS_SUCCESS)
    {
        DbgPrint("获取EProcessID失败 \n");
        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))
    {
        DbgPrint("地址不可读 \n");
        Driver->DriverUnload = UnDriver;
        return STATUS_SUCCESS;
    }

    // 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 NtdllUnicodeString = { 0 };
        PVOID NtdllAddress = NULL;

        // 得到进程内ntdll.dll模块基地址
        RtlInitUnicodeString(&NtdllUnicodeString, L"Ntdll.dll");
        NtdllAddress = GetUserModuleBaseAddress(EProcess, &NtdllUnicodeString, IsWow64);
        if (!NtdllAddress)
        {
            DbgPrint("没有找到基址 \n");
            Driver->DriverUnload = UnDriver;
            return STATUS_SUCCESS;
        }

        DbgPrint("[*] 模块ntdll.dll基址: %p \n", NtdllAddress);
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
    }

    // 取消附加
    KeUnstackDetachProcess(&ApcState);

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

替换 `DriverEntry` 入口函数处的 `ProcessID` 并替换为当前需要获取的应用层进程PID，运行驱动程序即可得到该进程内 `Ntdll.dll` 的模块基址，输出效果如下；



取模块中函数地址

实现获取特定模块中特定函数的基地址，通常我们通过 `GetUserModuleBaseAddress()` 可得到进程内特定模块的基址，然后则可继续通过 `GetModuleExportAddress()` 获取到该模块内特定导出函数的内存地址，至于获取导出表中特定函数的地址则可通过如下方式循环遍历导出表函数获取。

```
// 获取特定模块下的导出函数地址
PVOID GetModuleExportAddress(IN PVOID ModuleBase, IN PCCHAR FunctionName, IN PEPROCESS
EProcess)
{
    PIMAGE_DOS_HEADER ImageDosHeader = (PIMAGE_DOS_HEADER)ModuleBase;
    PIMAGE_NT_HEADERS32 ImageNtHeaders32 = NULL;
    PIMAGE_NT_HEADERS64 ImageNtHeaders64 = NULL;
    PIMAGE_EXPORT_DIRECTORY ImageExportDirectory = NULL;
    ULONG ExportDirectorySize = 0;
    ULONG_PTR FunctionAddress = 0;

    // 为空则返回
    if (ModuleBase == NULL)
    {
        return NULL;
    }

    // 是不是PE文件
    if (ImageDosHeader->e_magic != IMAGE_DOS_SIGNATURE)
    {
        return NULL;
    }

    // 获取NT头
    ImageNtHeaders32 = (PIMAGE_NT_HEADERS32)((PUCHAR)ModuleBase + ImageDosHeader->e_lfanew);
    ImageNtHeaders64 = (PIMAGE_NT_HEADERS64)((PUCHAR)ModuleBase + ImageDosHeader->e_lfanew);

    // 是64位则执行
    if (ImageNtHeaders64->OptionalHeader.Magic == IMAGE_NT_OPTIONAL_HDR64_MAGIC)
    {
        ImageExportDirectory = (PIMAGE_EXPORT_DIRECTORY)(ImageNtHeaders64-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress +
(ULONG_PTR)ModuleBase);
        ExportDirectorySize = ImageNtHeaders64-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].Size;
    }
    // 是32位则执行
    else
    {
        ImageExportDirectory = (PIMAGE_EXPORT_DIRECTORY)(ImageNtHeaders32-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress +
(ULONG_PTR)ModuleBase);
        ExportDirectorySize = ImageNtHeaders32-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].Size;
    }
```

```
    // 得到导出表地址偏移和名字
    PUSHORT pAddressOfOrds = (PUSHORT)(ImageExportDirectory->AddressOfNameOrdinals +
(ULONG_PTR)ModuleBase);
    PULONG  pAddressOfNames = (PULONG)(ImageExportDirectory->AddressOfNames +
(ULONG_PTR)ModuleBase);
    PULONG  pAddressOfFuncs = (PULONG)(ImageExportDirectory->AddressOfFunctions +
(ULONG_PTR)ModuleBase);

    // 循环搜索导出表
    for (ULONG i = 0; i < ImageExportDirectory->NumberOfFunctions; ++i)
    {
        USHORT OrdIndex = 0xFFFF;
        PCHAR  pName = NULL;

        // 搜索导出表下标索引
        if ((ULONG_PTR)FunctionName <= 0xFFFF)
        {
            OrdIndex = (USHORT)i;
        }
        // 搜索导出表名字
        else if ((ULONG_PTR)FunctionName > 0xFFFF && i < ImageExportDirectory-
>NumberOfNames)
        {
            pName = (PCHAR)(pAddressOfNames[i] + (ULONG_PTR)ModuleBase);
            OrdIndex = pAddressOfOrds[i];
        }
        else
        {
            return NULL;
        }

        // 找到设置返回值并跳出
        if (((ULONG_PTR)FunctionName <= 0xFFFF && (USHORT)((ULONG_PTR)FunctionName) ==
OrdIndex + ImageExportDirectory->Base) || ((ULONG_PTR)FunctionName > 0xFFFF && strcmp(pName,
FunctionName) == 0))
        {
            FunctionAddress = pAddressOfFuncs[OrdIndex] + (ULONG_PTR)ModuleBase;
            break;
        }
    }
    return (PVOID)FunctionAddress;
}
```

如何调用此方法，首先将 `ProcessID` 设置为需要读取的进程PID，然后将上图中所输出的 `0x00007FF9553C0000` 赋值给 `BaseAddress` 接着调用 `GetModuleExportAddress()` 并传入 `BaseAddress` 模块基址，需要读取的 `LdrLoadDll` 函数名，以及当前进程的 `EProcess` 结构。

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    HANDLE ProcessID = (HANDLE)4144;
    PEPROCESS EProcess = NULL;
    NTSTATUS Status = STATUS_SUCCESS;
```

```
    // 根据PID得到进程EProcess结构
    Status = PsLookupProcessByProcessId(ProcessID, &EProcess);
    if (Status != STATUS_SUCCESS)
    {
        DbgPrint("获取EProcessID失败 \n");
        return Status;
    }

    PVOID BaseAddress = (PVOID)0x00007FF9553C0000;
    PVOID RefAddress = 0;

    // 传入Ntdll.dll基址 + 函数名 得到该函数地址
    RefAddress = GetModuleExportAddress(BaseAddress, "LdrLoadDll", EProcess);
    DbgPrint("[*] 函数地址: %p \n", RefAddress);

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

运行这段程序，即可输出如下信息，此时也就得到了`x64.exe`进程内`ntdll.dll`模块里面的`LdrLoadDll`函数的内存地址，如下所示；