在上一篇博文《内核通过PEB得到进程参数》中我们通过使用KeStackAttachProcess附加进程的方式得到了该进程的PEB结构信息，本篇文章同样需要使用进程附加功能，但这次我们将实现一个更加有趣的功能，在某些情况下应用层与内核层需要共享一片内存区域通过这片区域可打通内核与应用层的隔离，此类功能的实现依附于MDL内存映射机制实现。

应用层数据映射到内核层

先来实现将R3内存数据拷贝到R0中，功能实现所调用的API如下:

- 调用IoAllocateMdl创建一个MDL结构体。这个结构体描述了一个要锁定的内存页的位置和大小。
- 调用MmProbeAndLockPages用于锁定创建的地址其中UserMode代表用户层,IoReadAccess以读取的方式锁定
- 调用MmGetSystemAddressForMdlSafe用于从MDL中得到映射内存地址
- 调用RtlCopyMemory用于内存拷贝,将DstAddr应用层中的数据拷贝到pMappedSrc中
- 调用MmUnlockPages拷贝结束后解锁pSrcMdl
- 调用IoFreeMdl释放之前创建的MDL结构体。

如上则是应用层数据映射到内核中的流程，我们将该流程封装成SafeCopyMemory_R3_to_R0方便后期的使用，函数对数据的复制进行了分块操作，因此可以处理更大的内存块。

下面是对该函数的分析：

- 1.首先进行一些参数的检查，如果有任何一个参数为0，那么函数就会返回 STATUS_UNSUCCESSFUL。
- 2.使用一个 while 循环来分块复制数据，每个块的大小为 PAGE_SIZE (通常是4KB)。这个循环在整个内存范围内循环，每次复制一个内存页的大小，直到复制完整个内存范围。
- 3.在循环内部，首先根据起始地址和当前要复制的大小来确定本次要复制的大小。如果剩余的内存不足一页大小，则只复制剩余的内存。
- 4.调用 IoAllocateMdl 创建一个 MDL，表示要锁定和复制的内存页。这里使用了 (PVOID)(SrcAddr & 0xFFFFFFFFFFFFF000) 来确定页的起始地址。因为页的大小为 0x1000，因此在计算页的起始地址时，将 SrcAddr 向下舍入到最接近的 0x1000 的倍数。
- 5.如果 IoAllocateMdl 成功，则调用 MmProbeAndLockPages 来锁定页面。这个函数将页面锁定到物理内存中，并返回一个虚拟地址，该虚拟地址指向已锁定页面的内核地址。
- 6.使用 MmGetSystemAddressForMdlSafe 函数获取一个映射到内核空间的地址，该地址可以直接访问锁定的内存页。
- 6.如果获取到了映射地址，则使用 RtlCopyMemory 函数将要复制的数据从应用层内存拷贝到映射到内核空间的地址。在复制结束后，使用 MmUnlockPages 函数解锁内存页，释放对页面的访问权限。

最后，释放 MDL 并更新 SrcAddr 和 DstAddr 以复制下一个内存块。如果复制过程中发生任何异常，函数将返回 STATUS_UNSUCCESSFUL。

总的来说，这个函数是一个很好的实现，它遵循了内核驱动程序中的最佳实践，包括对内存的安全处理、分块复制、错误处理等。

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

// 分配内存
```

```
void* RtlAllocateMemory(BOOLEAN InZeroMemory, SIZE_T InSize)
{
    void* Result = ExAllocatePoolWithTag(NonPagedPool, InSize, 'lysh');
    if (InZeroMemory && (Result != NULL))
        RtlZeroMemory(Result, InSize);
    return Result;
}

// 释放内存
void RtlFreeMemory(void* InPointer)
{
    ExFreePool(InPointer);
}

/*
将应用层中的内存复制到内核变量中

SrcAddr  r3地址要复制
DstAddr  R0申请的地址
Size     拷贝长度
*/
NTSTATUS SafeCopyMemory_R3_to_R0(ULONG_PTR SrcAddr, ULONG_PTR DstAddr, ULONG Size)
{
    NTSTATUS status = STATUS_UNSUCCESSFUL;
    ULONG nRemainSize = PAGE_SIZE - (SrcAddr & 0xFFF);
    ULONG nCopyedSize = 0;

    if (!SrcAddr || !DstAddr || !Size)
    {
        return status;
    }

    while (nCopyedSize < Size)
    {
        PMDL pSrcMdl = NULL;
        PVOID pMappedSrc = NULL;

        if (Size - nCopyedSize < nRemainSize)
        {
            nRemainSize = Size - nCopyedSize;
        }

        // 创建MDL
        pSrcMdl = IoAllocateMdl((PVOID)(SrcAddr & 0xFFFFFFFFFFFFF000), PAGE_SIZE, FALSE,
FALSE, NULL);
        if (pSrcMdl)
        {
            __try
            {
                // 锁定内存页面(UserMode代表应用层)
                MmProbeAndLockPages(pSrcMdl, UserMode, IoReadAccess);

                // 从MDL中得到映射内存地址
```

```
                pMappedSrc = MmGetSystemAddressForMdlSafe(pSrcMdl, NormalPagePriority);
            }
            __except (EXCEPTION_EXECUTE_HANDLER)
            {
            }
        }

        if (pMappedSrc)
        {
            __try
            {
                // 将MDL中的映射拷贝到pMappedSrc内存中
                RtlCopyMemory((PVOID)DstAddr, (PVOID)((ULONG_PTR)pMappedSrc + (SrcAddr &
0xFFF)), nRemainSize);
            }
            __except (1)
            {
                // 拷贝内存异常
            }

            // 释放锁
            MmUnlockPages(pSrcMdl);
        }

        if (pSrcMdl)
        {
            // 释放MDL
            IoFreeMdl(pSrcMdl);
        }

        if (nCopyedSize)
        {
            nRemainSize = PAGE_SIZE;
        }

        nCopyedSize += nRemainSize;
        SrcAddr += nRemainSize;
        DstAddr += nRemainSize;
    }

    status = STATUS_SUCCESS;
    return status;
}
```

有了封装好的`SafeCopyMemory_R3_to_R0`函数，那么接下来就是使用该函数实现应用层到内核层中的拷贝，为了能实现拷贝我们需要做以下几个准备工作；

- 1.使用`PsLookupProcessByProcessId`函数通过进程ID查找到对应的`EProcess`结构体，以获取该进程在内核中的信息。
- 2.使用`KeStackAttachProcess`函数将当前进程的执行上下文切换到指定进程的上下文，以便能够访问该进程的内存。
- 3.使用`RtlAllocateMemory`函数在当前进程的内存空间中分配一块缓冲区，用于存储从指定进程中读取的数

据。

- 4.调用`SafeCopyMemory_R3_to_R0`函数将指定进程的内存数据拷贝到分配的缓冲区中。
- 5.将缓冲区中的数据转换为`BYTE`类型的指针，并将其输出。

PsLookupProcessByProcessId函数用于通过进程ID查找到对应的EProcess结构体，这个结构体是Windows操作系统内核中用于表示一个进程的数据结构。

```
NTSTATUS status = PsLookupProcessByProcessId(ProcessId, &ProcessObject);
if (!NT_SUCCESS(status)) {
    return status;
}
```

KeStackAttachProcess函数将当前进程的执行上下文切换到指定进程的上下文，以便能够访问该进程的内存。这个函数也只能在内核态中调用。

```
KeStackAttachProcess(ProcessObject, &ApcState);
```

RtlAllocateMemory函数在当前进程的内存空间中分配一块缓冲区，用于存储从指定进程中读取的数据。这个函数是Windows操作系统内核中用于动态分配内存的函数，其中第一个参数TRUE表示允许操作系统在分配内存时进行页面合并，以减少内存碎片的产生。第二个参数nSize表示需要分配的内存空间的大小。如果分配失败，就需要将之前的操作撤销并返回错误状态。

```
PVOID pTempBuffer = RtlAllocateMemory(TRUE, nSize);
if (pTempBuffer == NULL) {
    KeUnstackDetachProcess(&ApcState);
    ObDereferenceObject(ProcessObject);
    return STATUS_NO_MEMORY;
}
```

SafeCopyMemory_R3_to_R0函数将指定进程的内存数据拷贝到分配的缓冲区中。

```
if (!SafeCopyMemory_R3_to_R0(ModuleBase, pTempBuffer, nSize)) {
    RtlFreeMemory(pTempBuffer);
    KeUnstackDetachProcess(&ApcState);
    ObDereferenceObject(ProcessObject);
    return STATUS_UNSUCCESSFUL;
}
```

最后，将缓冲区中的数据转换为BYTE类型的指针，并将其输出。需要注意的是，在返回之前需要先将当前进程的执行上下文切换回原先的上下文。

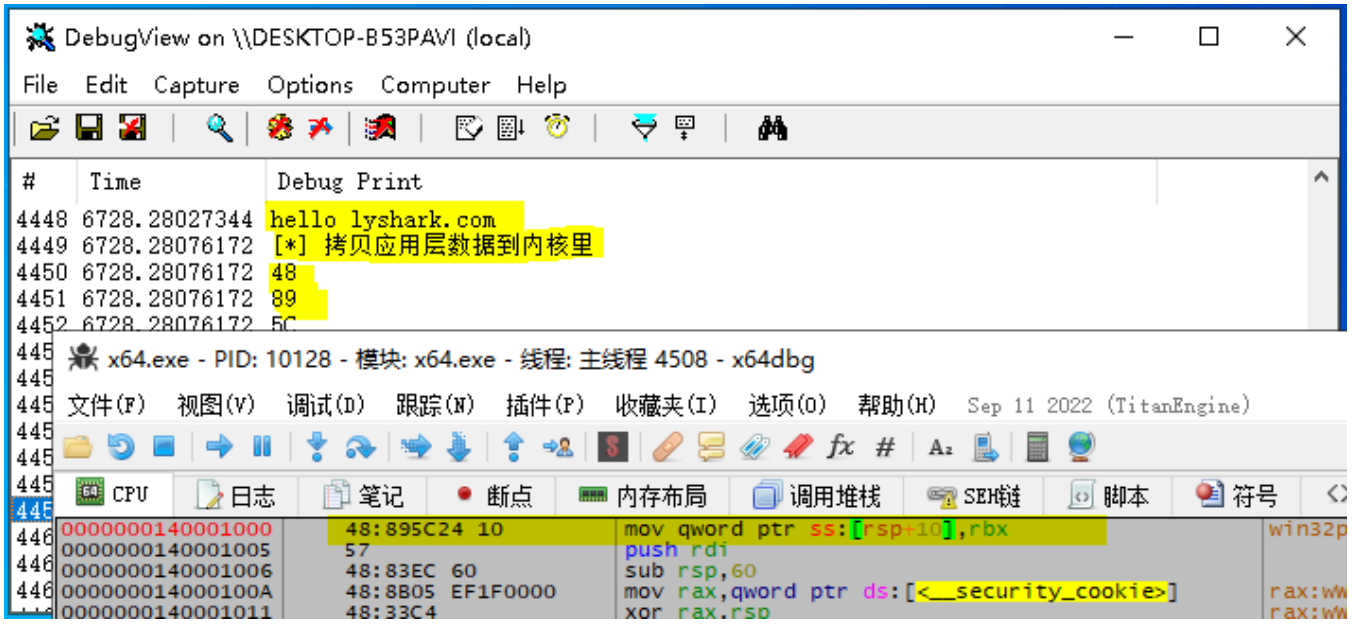
```
BYTE* data = (BYTE*)pTempBuffer;
KeUnstackDetachProcess(&ApcState);
ObDereferenceObject(ProcessObject);
return data;
```

如上实现细节用一段话总结，首先`PsLookupProcessByProcessId`得到进程`EProcess`结构，并`KeStackAttachProcess`附加进程，声明`pTempBuffer`指针用于存储`RtlAllocateMemory`开辟的内存空间，`nSize`则代表读取应用层进程数据长度，`ModuleBase`则是读入进程基址，调用`SafeCopyMemory_R3_to_R0`即可将应用层数据拷贝到内核空间，并最终`BYTE* data`转为BYTE字节的方式输出。这样就完成了将指定进程的内存数据拷贝到当前进程中的操作。

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark.com \n");

    NTSTATUS status = STATUS_UNSUCCESSFUL;
    PEPROCESS eproc = NULL;
    KAPC_STATE kpc = { 0 };

    __try
    {
        // HANDLE 进程PID
        status = PsLookupProcessByProcessId((HANDLE)4556, &eproc);

        if (NT_SUCCESS(status))
        {
            // 附加进程
            KeStackAttachProcess(eproc, &kpc);

            // ----------------------------------------------------------------------
            // 开始映射
            // ----------------------------------------------------------------------

            // 将用户空间内存映射到内核空间
            PVOID pTempBuffer = NULL;
            ULONG nSize = 0x1024;
            ULONG_PTR ModuleBase = 0x0000000140001000;

            // 分配内存
            pTempBuffer = RtlAllocateMemory(TRUE, nSize);
            if (pTempBuffer)
            {
                // 拷贝数据到R0
                status = SafeCopyMemory_R3_to_R0(ModuleBase, (ULONG_PTR)pTempBuffer, nSize);
                if (NT_SUCCESS(status))
                {
                    DbgPrint("[*] 拷贝应用层数据到内核里 \n");
                }

                // 转成BYTE方便读取
                BYTE* data = pTempBuffer;

                for (size_t i = 0; i < 10; i++)
                {
                    DbgPrint("%02X \n", data[i]);
                }
            }
```

```
            // 释放空间
            RtlFreeMemory(pTempBuffer);

            // 脱离进程
            KeUnstackDetachProcess(&kpc);
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
        Driver->DriverUnload = UnDriver;
        return STATUS_SUCCESS;
    }

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

代码运行后即可将进程中 `0x0000000140001000` 处的数据读入内核空间并输出：



内核层数据映射到应用层

与上方功能实现相反 `SafeCopyMemory_R0_to_R3` 函数则用于将一个内核层中的缓冲区写出到应用层中，`SafeCopyMemory_R0_to_R3` 函数接收源地址SrcAddr、要复制的数据长度 `Length` 以及目标地址 `DstAddr` 作为参数，其写出流程可总结为如下步骤：

- 1.使用 `IoAllocateMdl` 函数分别为源地址 `SrcAddr` 和目标地址 `DstAddr` 创建两个内存描述列表（MDL）。
- 2.使用 `MmBuildMdlForNonPagedPool` 函数，将源地址的 `MDL` 更新为描述非分页池的虚拟内存缓冲区，并更新 `MDL` 以描述底层物理页。
- 3.通过两次调用 `MmGetSystemAddressForMdlSafe` 函数，分别获取源地址和目标地址的指针，即 `pSrcMdl` 和 `pDstMdl`。
- 4.使用 `MmProbeAndLockPages` 函数以写入方式锁定用户空间中 `pDstMdl` 指向的地址，并将它的虚拟地址映射到物理内存页，从而确保该内存页在复制期间不会被交换出去或被释放掉。

- 5.然后使用`MmMapLockedPagesSpecifyCache`函数将锁定的用户空间内存页映射到内核空间，并返回内核空间中的虚拟地址。
- 6.最后使用`RtlCopyMemory`函数将源地址的数据复制到目标地址。
- 7.使用`MmUnlockPages`函数解除用户空间内存页的锁定，并使用`MmUnmapLockedPages`函数取消内核空间与用户空间之间的内存映射。
- 8.最后释放源地址和目标地址的`MDL`，使用`IoFreeMdl`函数进行释放。

内存拷贝`SafeCopyMemory_R0_to_R3`函数，函数首先分配源地址和目标地址的`MDL`结构，然后获取它们的虚拟地址，并以写入方式锁定目标地址的`MDL`，最后使用`RtlCopyMemory`函数将源地址的内存数据拷贝到目标地址。拷贝完成后，函数解锁目标地址的`MDL`，并返回操作状态。

封装代码`SafeCopyMemory_R0_to_R3()`功能如下:

```
// 分配内存
void* RtlAllocateMemory(BOOLEAN InZeroMemory, SIZE_T InSize)
{
    void* Result = ExAllocatePoolWithTag(NonPagedPool, InSize, 'lysh');
    if (InZeroMemory && (Result != NULL))
        RtlZeroMemory(Result, InSize);
    return Result;
}

// 释放内存
void RtlFreeMemory(void* InPointer)
{
    ExFreePool(InPointer);
}

/*
将内存中的数据复制到R3中

SrcAddr  R0要复制的地址
DstAddr  返回R3的地址
Size     拷贝长度
*/
NTSTATUS SafeCopyMemory_R0_to_R3(PVOID SrcAddr, PVOID DstAddr, ULONG Size)
{
    PMDL  pSrcMdl = NULL, pDstMdl = NULL;
    PUCHAR pSrcAddress = NULL, pDstAddress = NULL;
    NTSTATUS st = STATUS_UNSUCCESSFUL;

    // 分配MDL 源地址
    pSrcMdl = IoAllocateMdl(SrcAddr, Size, FALSE, FALSE, NULL);
    if (!pSrcMdl)
    {
        return st;
    }

    // 该 MDL 指定非分页虚拟内存缓冲区，并对其进行更新以描述基础物理页。
    MmBuildMdlForNonPagedPool(pSrcMdl);
```

```
    // 获取源地址MDL地址
    pSrcAddress = MmGetSystemAddressForMdlSafe(pSrcMdl, NormalPagePriority);


    if (!pSrcAddress)
    {
        IoFreeMdl(pSrcMdl);
        return st;
    }

    // 分配MDL 目标地址
    pDstMdl = IoAllocateMdl(DstAddr, Size, FALSE, FALSE, NULL);
    if (!pDstMdl)
    {
        IoFreeMdl(pSrcMdl);
        return st;
    }

    __try
    {
        // 以写入的方式锁定目标MDL
        MmProbeAndLockPages(pDstMdl, UserMode, IoWriteAccess);

        // 获取目标地址MDL地址
        pDstAddress = MmGetSystemAddressForMdlSafe(pDstMdl, NormalPagePriority);
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
    }

    if (pDstAddress)
    {
        __try
        {
            // 将源地址拷贝到目标地址
            RtlCopyMemory(pDstAddress, pSrcAddress, Size);
        }
        __except (1)
        {
            // 拷贝内存异常
        }
        MmUnlockPages(pDstMdl);
        st = STATUS_SUCCESS;
    }

    IoFreeMdl(pDstMdl);
    IoFreeMdl(pSrcMdl);

    return st;
}
```

调用该函数实现拷贝，此处除去附加进程以外，在拷贝之前调用了`ZwAllocateVirtualMemory`将内存属性设置为`PAGE_EXECUTE_READWRITE`可读可写可执行状态，然后在向该内存中写出`pTempBuffer`变量中的内容，此变量中的数据是`0x90`填充的区域。

此处的`ZwAllocateVirtualMemory`函数，用于在进程的虚拟地址空间中分配一块连续的内存区域，以供进程使用。它属于Windows内核API的一种，与用户态的`VirtualAlloc`函数相似，但是它运行于内核态，可以分配不受用户空间地址限制的虚拟内存，并且可以用于在驱动程序中为自己或其他进程分配内存。

函数的原型为：

```
NTSYSAPI NTSTATUS NTAPI ZwAllocateVirtualMemory(
  _In_    HANDLE    ProcessHandle,
  _Inout_ PVOID     *BaseAddress,
  _In_    ULONG_PTR ZeroBits,
  _Inout_ PSIZE_T   RegionSize,
  _In_    ULONG     AllocationType,
  _In_    ULONG     Protect
);
```

其中`ProcessHandle`参数是进程句柄，`BaseAddress`是分配到的内存区域的起始地址，`ZeroBits`指定保留的高位，`RegionSize`是分配内存大小，`AllocationType`和`Protect`分别表示内存分配类型和内存保护属性。

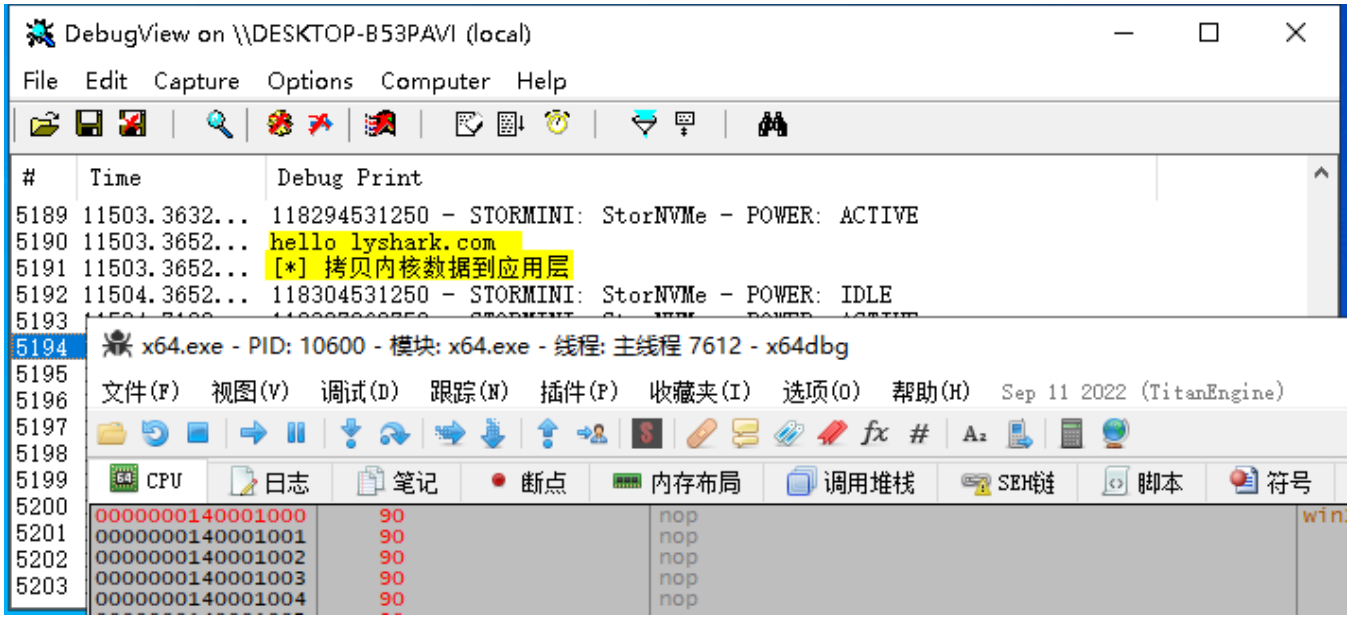
ZwAllocateVirtualMemory函数成功返回`NT_SUCCESS`，返回值为0，否则返回相应的错误代码。如果函数成功调用，会将`BaseAddress`参数指向分配到的内存区域的起始地址，同时将`RegionSize`指向的值修改为实际分配到的内存大小。

当内存属性被设置为`PAGE_EXECUTE_READWRITE`之后，则下一步直接调用`SafeCopyMemory_R0_to_R3`进行映射即可，其调用完整案例如下所示；

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark.com \n");

    NTSTATUS status = STATUS_UNSUCCESSFUL;
    PEPROCESS eproc = NULL;
    KAPC_STATE kpc = { 0 };

    __try
    {
        // HANDLE 进程PID
        status = PsLookupProcessByProcessId((HANDLE)4556, &eproc);

        if (NT_SUCCESS(status))
        {
            // 附加进程
            KeStackAttachProcess(eproc, &kpc);

            // ------------------------------------------------------------------
            // 开始映射
            // ------------------------------------------------------------------

            // 将用户空间内存映射到内核空间
```

```
            PVOID pTempBuffer = NULL;
            ULONG nSize = 0x1024;
            PVOID ModuleBase = 0x0000000140001000;

            // 分配内存
            pTempBuffer = RtlAllocateMemory(TRUE, nSize);
            if (pTempBuffer)
            {
                memset(pTempBuffer, 0x90, nSize);

                // 设置内存属性 PAGE_EXECUTE_READWRITE
                ZwAllocateVirtualMemory(NtCurrentProcess(), &ModuleBase, 0, &nSize,
MEM_RESERVE, PAGE_EXECUTE_READWRITE);
                ZwAllocateVirtualMemory(NtCurrentProcess(), &ModuleBase, 0, &nSize,
MEM_COMMIT, PAGE_EXECUTE_READWRITE);

                // 将数据拷贝到R3中
                status = SafeCopyMemory_R0_to_R3(pTempBuffer, &ModuleBase, nSize);
                if (NT_SUCCESS(status))
                {
                    DbgPrint("[*] 拷贝内核数据到应用层 \n");
                }
            }

            // 释放空间
            RtlFreeMemory(pTempBuffer);

            // 脱离进程
            KeUnstackDetachProcess(&kpc);
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
        Driver->DriverUnload = UnDriver;
        return STATUS_SUCCESS;
    }

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

拷贝成功后，应用层进程内将会被填充为`Nop`指令。


DebugView on \\DESKTOP-B53PAVI (local)
File Edit Capture Options Computer Help
Time Debug Print
5189 11503.3632... 118294531250 - STORMINI: StorNVMe - POWER: ACTIVE
5190 11503.3652... hello lyshark.com
5191 11503.3652... [*] 拷贝内核数据到应用层
5192 11504.3652... 118304531250 - STORMINI: StorNVMe - POWER: IDLE
5193
5194
5195
5196
5197
5198
5199
5200
5201
5202
5203
x64.exe - PID: 10600 - 模块: x64.exe - 线程: 主线程 7612 - x64dbg
文件(F) 视图(V) 调试(D) 跟踪(N) 插件(P) 收藏夹(I) 选项(O) 帮助(H) Sep 11 2022 (TitanEngine)
CPU 日志 笔记 断点 内存布局 调用堆栈 SEH链 脚本 符号
0000000140001000 90 nop
0000000140001001 90 nop
0000000140001002 90 nop
0000000140001003 90 nop
0000000140001004 90 nop