在上一篇博文《驱动开发：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(R3)数据映射到内核层(R0)

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:

- IoAllocateMdl 该函数用于创建MDL(类似初始化)
- 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
- MmGetSystemAddressForMdlSafe 用于从MDL中得到映射内存地址
- RtlCopyMemory 用于内存拷贝,将DstAddr应用层中的数据拷贝到pMappedSrc中
- MmUnlockPages 拷贝结束后解锁pSrcMdl
- IoFreeMdl 释放MDL

内存拷贝SafeCopyMemory_R3_to_R0函数封装代码如下:

```
#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);
        }
```

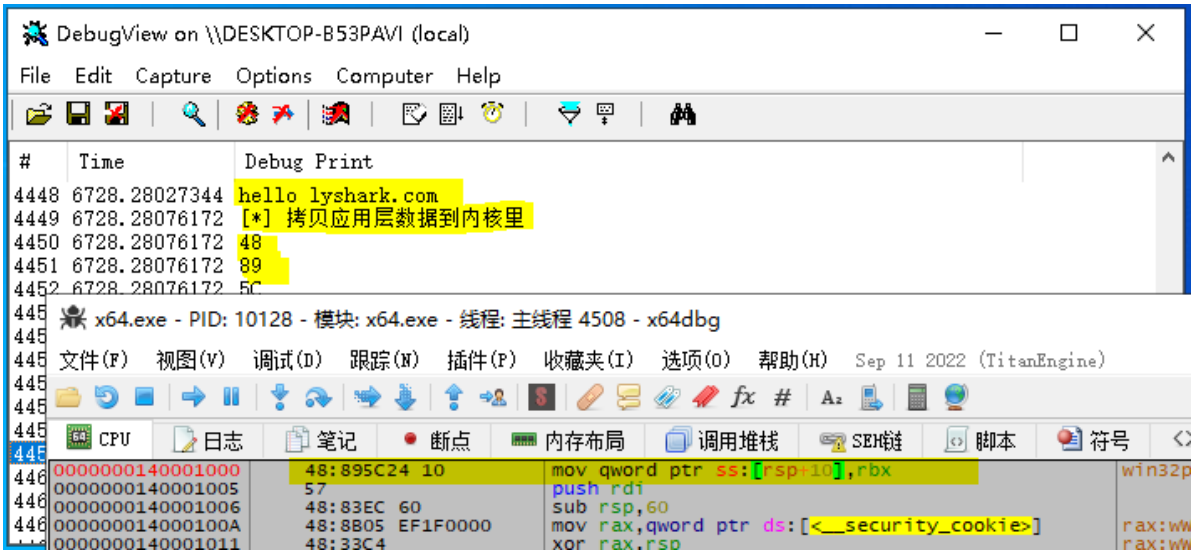
```
        if (nCopyedSize)
        {
            nRemainSize = PAGE_SIZE;
        }

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

    status = STATUS_SUCCESS;
    return status;
}
```

调用该函数实现拷贝，如下代码中首先 `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。

```
VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint(("Uninstall Driver Is OK \n"));
}

// lyshark.com
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(R0)数据映射到应用层(R3)

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，写出过程:

- IoAllocateMdl 分别调用MDL分配,源地址`SrcAddr`目标地址`DstAddr`均创建
- MmBuildMdlForNonPagedPool 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
- MmGetSystemAddressForMdlSafe 调用两次得到源地址，分别获取`pSrcMdl`,`pDstMdl`两个MDL的
- MmProbeAndLockPages 以写入方式锁定用户层中`pDstMdl`的地址

内存拷贝`SafeCopyMemory_R0_to_R3`函数封装代码如下:

```
// 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;
}
```

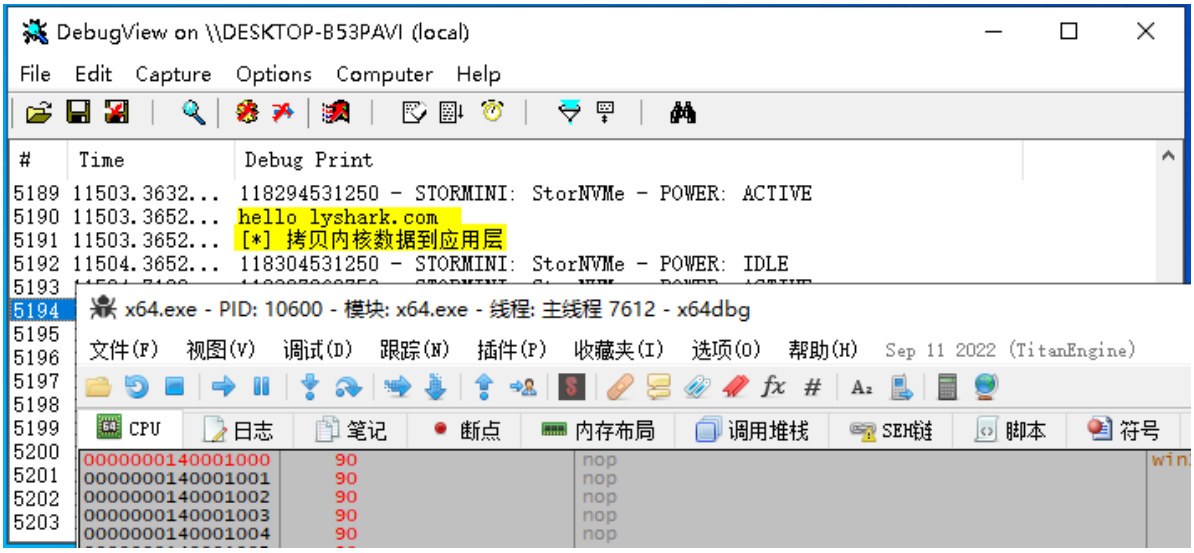
调用该函数实现拷贝，此处除去附加进程以外，在拷贝之前调用了`ZwAllocateVirtualMemory`将内存属性设置为`PAGE_EXECUTE_READWRITE`可读可写可执行状态，然后在向该内存中写出`pTempBuffer`变量中的内容，此变量中的数据是`0x90`填充的区域。

```
VOID UnDriver(PDRIVER_OBJECT driver)
```

```
{
    DbgPrint(("Uninstall Driver Is OK \n"));
}

// lyshark.com
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。



本书作者： 王瑞 (LyShark)
作者邮箱： me@lyshark.com
作者博客： https://lyshark.cnblogs.com
团队首页： www.lyshark.com