

本章将继续探索内核中解析PE文件的相关内容，PE文件中FOA与VA,RVA之间的转换也是很重要的，所谓的FOA是文件中的地址，VA则是内存装入后的虚拟地址，RVA是内存基址与当前地址的相对偏移，本章还是需要用到《内核解析PE结构导出表》中所封装的 `KernelMapFile()` 映射函数，在映射后对其PE格式进行相应的解析，并实现转换函数。

首先先来演示一下内存VA地址与FOA地址互相转换的方式，通过使用WinHEX打开一个二进制文件，打开后我们只需要关注如下蓝色注释为映像建议装入基址，黄色注释为映像装入后的RVA偏移。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI	ASCII
000000B0	BA	0B	25	3C	62	F4	EE	3C	67	F4	EF	3C	50	F4	EE	3C	°	%<bôî<gôî<Pôî<
000000C0	6A	A6	0B	3C	65	F4	EE	3C	6A	A6	35	3C	66	F4	EE	3C	j	<eôî<j 5<fôî<
000000D0	67	F4	79	3C	66	F4	EE	3C	6A	A6	30	3C	66	F4	EE	3C	gôy<fôî<j 0<fôî<	
000000E0	52	69	63	68	67	F4	EE	3C	00	00	00	00	00	00	00	00	Richgôî<	
000000F0	00	00	00	00	00	00	00	00	50	45	00	00	4C	01	05	00	PE	L
00000100	0F	77	BD	5D	00	00	00	00	00	00	00	00	E0	00	02	01	w*]	à
00000110	0B	01	0C	00	00	0C	00	00	00	2E	00	00	00	00	00	00	.	
00000120	8B	15	00	00	00	10	00	00	00	20	00	00	00	00	40	00	<	@
00000130	00	10	00	00	00	02	00	00	06	00	00	00	00	00	00	00		
00000140	06	00	00	00	00	00	00	00	00	70	00	00	00	04	00	00	p	
00000150	00	00	00	00	02	00	40	81	00	00	10	00	00	10	00	00	@	
00000160	00	00	10	00	00	10	00	00	00	00	00	00	10	00	00	00	..	-

通过上方的截图结合PE文件结构图我们可得知 0000158B 为映像装入内存后的RVA偏移，紧随其后的 00400000 则是映像的建议装入基址，为什么是建议而不是绝对？别急后面慢慢来解释。

通过上方的已知条件我们就可以计算出程序实际装入内存后的入口地址了，公式如下：
 $VA(\text{实际装入地址}) = ImageBase(\text{基址}) + RVA(\text{偏移}) \Rightarrow 00400000 + 0000158B = 0040158B$

找到了程序的OEP以后，接着我们来判断一下这个 0040158B 属于那个节区，以.text节区为例，下图我们通过观察区段可知，第一处橙色位置 00000B44（节区尺寸），第二处紫色位置 00001000（节区RVA），第三处 00000C00（文件对齐尺寸），第四处 00000400（文件中的偏移），第五处 60000020（节区属性）。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI	ASCII
000001D0	00	20	00	00	E8	00	00	00	00	00	00	00	00	00	00	00	è	
000001E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
000001F0	2E	74	65	78	74	00	00	00	44	0B	00	00	00	10	00	00	.text	D
00000200	00	0C	00	00	00	04	00	00	00	00	00	00	00	00	00	00		
00000210	00	00	00	00	20	00	00	60	2E	72	64	61	74	61	00	00	.rdata	
00000220	9A	07	00	00	00	20	00	00	00	08	00	00	00	10	00	00	š	
00000230	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	40	@	@
00000240	2E	64	61	74	61	00	00	00	18	05	00	00	00	30	00	00	.data	0
00000250	00	02	00	00	00	18	00	00	00	00	00	00	00	00	00	00		
00000260	00	00	00	00	40	00	00	C0	2E	72	73	72	63	00	00	00	@	À.rsrc
00000270	B0	1D	00	00	00	40	00	00	00	1E	00	00	00	1A	00	00	°	@

得到了上方text节的相关数据，我们就可以判断程序的OEP到底落在了那个节区中，这里以.text节为例子，计算公式如下：

虚拟地址开始位置：节区基地址 + 节区RVA => 00400000 + 00001000 = 00401000
虚拟地址结束位置：text节地址 + 节区尺寸 => 00401000 + 00000B44 = 00401B44

经过计算得知 .text 节所在区间（401000 - 401B44）你的装入VA地址 0040158B 只要在区间里面就证明在本节区中，此处的VA地址是在 401000 - 401B44 区间内的，则说明它属于.text节。

经过上面的公式计算我们知道了程序的OEP位置是落在了.text节，此时你兴致勃勃的打开x64DBG想去验证一下公式是否计算正确不料，这地址根本不是400000开头啊，这是什么鬼？

CPU

流程图

日志

笔记

断点

内存布局

调用堆栈

SEH链

脚本

地址	大小	页面信息	内容	类型	页面保护
00AE0000	00001000	setup.exe		IMG	-R---
00AE1000	00001000	".text"	可执行代码	IMG	ER---
00AE2000	00001000	".rdta"	只读的已初始化数据	IMG	-R---
00AE3000	00001000	".data"	已初始化的数据	IMG	-RW--
00AE4000	00002000	".rsrc"	资源	IMG	-R---
00AE6000	00001000	".reloc"	基重定位数据	IMG	-R---
00DC0000	00010000			MAP	-RW--

上图中出现的这种情况就是关于随机基址的问题，在新版的VS编译器上存在一个选项是否要启用随机基址(默认启用)，至于这个随机基址的作用，猜测可能是为了防止缓冲区溢出之类的烂七八糟的东西。

为了方便我们调试，我们需要手动干掉它，其对应到PE文件中的结构为 IMAGE_NT_HEADERS ->

IMAGE_OPTIONAL_HEADER -> DllCharacteristics 相对于PE头的偏移为90字节，只需要修改这个标志即可，修改方式 x64: 6081 改 2081 相对于 x86: 4081 改 0081 以X86程序为例，修改后如下图所示。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI	ASCII
00000140	06	00	00	00	00	00	00	00	70	00	00	00	04	00	00	00		P
00000150	00	00	00	00	02	00	00	81	00	00	10	00	00	10	00	00		
00000160	00	00	10	00	00	10	00	00	00	00	00	00	10	00	00	00		
00000170	00	00	00	00	00	00	00	00	84	22	00	00	50	00	00	00		" P
00000180	00	40	00	00	B0	1D	00	00	00	00	00	00	00	00	00	00	@	°

经过上面对标志位的修改，程序再次载入就能够停在 0040158B 的位置，也就是程序的OEP，接下来我们将通过公式计算出该OEP对应到文件中的位置。

.text(节首地址) = ImageBase + 节区RVA => 00400000 + 00001000 = 00401000

VA(虚拟地址) = ImageBase + RVA(偏移) => 00400000 + 0000158B = 0040158B

RVA(相对偏移) = VA - (.text节首地址) => 0040158B - 00401000 = 58B

FOA(文件偏移) = RVA + .text节对应到文件中的偏移 => 58B + 400 = 98B

经过公式的计算，我们找到了虚拟地址 0040158B 对应到文件中的位置是 98B，通过WinHEX定位过去，即可看到OEP处的机器码指令了。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI	ASCII
00000970	00	75	0B	FF	15	68	20	40	00	A1	30	40	00	C7	45		u y h @ ; 0@ ÇE	
00000980	FC	FE	FF	FF	FF	E8	3B	05	00	00	C3	E8	E1	02	00	00	üpyÿyè; Àèá	
00000990	E9	49	FE	FF	FF	55	8B	EC	FF	15	18	20	40	00	6A	01	éIpyÿU<iÿ @ j	
000009A0	A3	54	33	40	00	E8	52	05	00	00	FF	75	08	E8	50	05	£T3@ èR yu èP	
000009B0	00	00	83	3D	54	33	40	00	00	59	59	75	08	6A	01	E8	f=T3@ YYu j è	
000009C0	38	05	00	00	59	68	09	04	00	C0	E8	39	05	00	00	59	8 Yh Àè9 Y	
000009D0	5D	C3	55	8B	EC	81	EC	24	03	00	00	6A	17	E8	5C	05]ÄU<i i\$ j è\	
000009E0	00	00	85	C0	74	05	6A	02	59	CD	29	A3	38	31	40	00	...Àt j YÍ)£81@	

接着我们来计算一下.text节区的结束地址，通过文件的偏移加上文件对齐尺寸即可得到.text节的结束地址 400+c00=1000，那么我们主要就在文件偏移为(98B - 1000)在该区间中找空白的地方，此处我找到了在文件偏移为1000之前的位置有一段空白区域，如下图：

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI	ASCII
00000F40	14	20	40	00	00	00	00	00	00	00	00	00	00	00	00	00	@	
00000F50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
00000F60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
00000F70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
00000F80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
00000F90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
00000FA0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		

接着我么通过公式计算一下文件偏移为 0xF43 的位置，其对应到VA虚拟地址是多少，公式如下：

.text(节首地址) = ImageBase + 节区RVA => 00400000 + 00001000 = 00401000
VPK(实际大小) = (text节首地址 - ImageBase) - 实际偏移 => 401000-400000-400 = C00
VA(虚拟地址) = FOA(.text节) + ImageBase + VPK => F43+400000+C00 = 401B43

.text(节首地址) = ImageBase + 节区RVA => 00400000 + 00001000 = 00401000
VPK(实际大小) = (text节首地址 - ImageBase) - 实际偏移 => 401000-400000-400 = C00
VA(虚拟地址) = FOA(.text节) + ImageBase + VPK => F43+400000+C00 = 401B43

.text(节首地址) = ImageBase + 节区RVA => 00400000 + 00001000 = 00401000
VPK(实际大小) = (text节首地址 - ImageBase) - 实际偏移 => 401000-400000-400 = C00
VA(虚拟地址) = FOA(.text节) + ImageBase + VPK => F43+400000+C00 = 401B43

.text(节首地址) = ImageBase + 节区RVA => 00400000 + 00001000 = 00401000
VPK(实际大小) = (text节首地址 - ImageBase) - 实际偏移 => 401000-400000-400 = C00
VA(虚拟地址) = FOA(.text节) + ImageBase + VPK => F43+400000+C00 = 401B43

计算后直接X64DBG跳转过去，我们从 00401B44 的位置向下全部填充为90(nop)，然后直接保存文件。

CPU	流程图	日志	笔记	断点	内存布局	调用堆栈	SEH链	脚本
00401B32	▼	FF25 28204000						
00401B38	▼	FF25 24204000						
00401B3E	▼	FF25 14204000						
00401B44		90						
00401B45		90						
00401B46		90						
00401B47		90						
00401B48		90						
00401B49		90						
00401B4A		90						
00401B4B		90						
00401B4C		90						
00401B4D		90						
00401B4E		90						
00401B4F		90						
00401B50		90						

再次使用WinHEX查看文件偏移为 0xF43 的位置，会发现已经全部替换成了90指令，说明计算正确。

[illegible]

到此文件偏移与虚拟偏移的转换就结束了，那么这些功能该如何实现呢，接下来将以此实现这些转换细节。

FOA转换为VA

首先来实现将 FOA 地址转换为 VA 地址，这段代码实现起来很简单，如下所示，此处将 dwFOA 地址 0x84EC00 转换为对应内存的虚拟地址。

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

    NTSTATUS status = STATUS_SUCCESS;
    HANDLE hFile = NULL;
    HANDLE hSection = NULL;
    PVOID pBaseAddress = NULL;
    UNICODE_STRING FileName = { 0 };

    // 初始化字符串
    RtlInitUnicodeString(&FileName, L"\\??\\C:\\windows\\System32\\ntoskrnl.exe");

    // 内存映射文件
```

```

status = KernelMapFile(FileName, &hFile, &hSection, &pBaseAddress);
if (!NT_SUCCESS(status))
{
    return 0;
}

// 获取PE头数据集
PIMAGE_DOS_HEADER pDosHeader = (PIMAGE_DOS_HEADER)pBaseAddress;
PIMAGE_NT_HEADERS pNtHeaders = (PIMAGE_NT_HEADERS)((PUCHAR)pDosHeader + pDosHeader->e_lfanew);
PIMAGE_SECTION_HEADER pSection = IMAGE_FIRST_SECTION(pNtHeaders);
PIMAGE_FILE_HEADER pFileHeader = &pNtHeaders->FileHeader;

DWORD64 dwFOA = 0x84EC00;

DWORD64 ImageBase = pNtHeaders->OptionalHeader.ImageBase;
DWORD NumberOfSectinsCount = pNtHeaders->FileHeader.NumberOfSections;
DbgPrint("镜像基址 = %p | 节表数量 = %d \n", ImageBase, NumberOfSectinsCount);

for (int each = 0; each < NumberOfSectinsCount; each++)
{
    DWORD64 PointerRawStart = pSection[each].PointerToRawData;
    // 文件偏移开始位置
    DWORD64 PointerRawEnds = pSection[each].PointerToRawData +
pSection[each].SizeOfRawData; // 文件偏移结束位置
    // DbgPrint("文件开始偏移 = %p | 文件结束偏移 = %p \n", PointerRawStart,
PointerRawEnds);

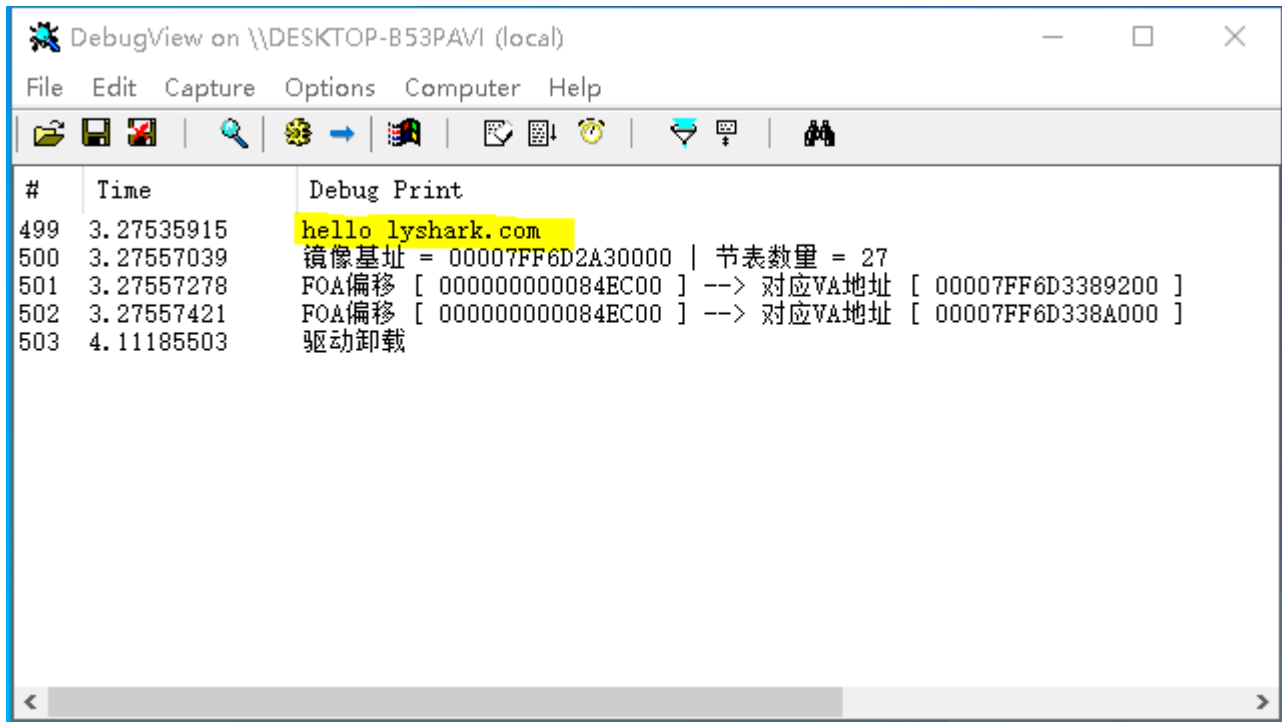
    if (dwFOA >= PointerRawStart && dwFOA <= PointerRawEnds)
    {
        DWORD64 RVA = pSection[each].VirtualAddress + (dwFOA -
pSection[each].PointerToRawData); // 计算出RVA
        DWORD64 VA = RVA + pNtHeaders->OptionalHeader.ImageBase;
        // 计算出VA
        DbgPrint("FOA偏移 [ %p ] --> 对应VA地址 [ %p ] \n", dwFOA, VA);
    }
}

ZwUnmapViewOfSection(NtCurrentProcess(), pBaseAddress);
ZwClose(hSection);
ZwClose(hFile);

Driver->DriverUnload = UnDriver;
return STATUS_SUCCESS;
}

```

运行效果如下所示，此处之所以出现两个结果是因为没有及时返回，一般我们取第一个结果就是最准确的；



VA转换为FOA

将VA内存地址转换为FOA文件偏移，代码与如上基本保持一致。

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

    NTSTATUS status = STATUS_SUCCESS;
    HANDLE hFile = NULL;
    HANDLE hSection = NULL;
    PVOID pBaseAddress = NULL;
    UNICODE_STRING FileName = { 0 };

    // 初始化字符串
    RtlInitUnicodeString(&FileName, L"\\??\\C:\\Windows\\System32\\ntoskrnl.exe");

    // 内存映射文件
    status = KernelMapFile(FileName, &hFile, &hSection, &pBaseAddress);
    if (!NT_SUCCESS(status))
    {
        return 0;
    }

    // 获取PE头数据集
    PIMAGE_DOS_HEADER pDosHeader = (PIMAGE_DOS_HEADER)pBaseAddress;
    PIMAGE_NT_HEADERS pNtHeaders = (PIMAGE_NT_HEADERS)((PUCHAR)pDosHeader + pDosHeader->e_lfanew);
    PIMAGE_SECTION_HEADER pSection = IMAGE_FIRST_SECTION(pNtHeaders);
    PIMAGE_FILE_HEADER pFileHeader = &pNtHeaders->FileHeader;

    DWORD64 dwVA = 0x00007FF6D3389200;
    DWORD64 ImageBase = pNtHeaders->OptionalHeader.ImageBase;
```

```

DWORD NumberOfSectinsCount = pNtHeaders->FileHeader.NumberOfSections;
DbgPrint("镜像基址 = %p | 节表数量 = %d \n", ImageBase, NumberOfSectinsCount);

for (DWORD each = 0; each < NumberOfSectinsCount; each++)
{
    DWORD Section_Start = ImageBase + pSection[each].VirtualAddress;
    // 获取节的开始地址
    DWORD Section_Ends = ImageBase + pSection[each].VirtualAddress +
pSection[each].Misc.VirtualSize; // 获取节的结束地址

    DbgPrint("Section开始地址 = %p | Section结束地址 = %p \n", Section_Start,
Section_Ends);

    if (dwVA >= Section_Start && dwVA <= Section_Ends)
    {
        DWORD RVA = dwVA - pNtHeaders->OptionalHeader.ImageBase;
        // 计算RVA
        DWORD FOA = pSection[each].PointerToRawData + (RVA -
pSection[each].VirtualAddress); // 计算FOA

        DbgPrint("VA偏移 [ %p ] --> 对应FOA地址 [ %p ] \n", dwVA, FOA);
    }
}

ZwUnmapViewOfSection(NtCurrentProcess(), pBaseAddress);
ZwClose(hSection);
ZwClose(hFile);

Driver->DriverUnload = UnDriver;
return STATUS_SUCCESS;
}

```

运行效果如下所示，此处没有出现想要的结果是因为我们当前的VA内存地址并非实际装载地址，仅仅是PE磁盘中的地址，此处如果换成内存中的PE则可以提取出正确的结果；

#	Time	Debug Print
10	0.00177320	hello lyshark.com
11	0.00191340	镜像基址 = 00007FF6D2A30000 节表数量 = 27
12	0.00191550	Section开始地址 = 00000000D2A31000 Section结束地址 = 00000000D2D80000
13	0.00191680	Section开始地址 = 00000000D2D81000 Section结束地址 = 00000000D2D83000
14	0.00191800	Section开始地址 = 00000000D2D84000 Section结束地址 = 00000000D2D84000
15	0.00191930	Section开始地址 = 00000000D2D85000 Section结束地址 = 00000000D2D9E000
16	0.00192050	Section开始地址 = 00000000D2D9F000 Section结束地址 = 00000000D2D9F000
17	0.00192170	Section开始地址 = 00000000D2DA0000 Section结束地址 = 00000000D2E56000
18	0.00192290	Section开始地址 = 00000000D2E56000 Section结束地址 = 00000000D2F40000
19	0.00192410	Section开始地址 = 00000000D2F41000 Section结束地址 = 00000000D2F9F000
20	0.00192540	Section开始地址 = 00000000D2FA0000 Section结束地址 = 00000000D2FA2000
21	0.00192670	Section开始地址 = 00000000D2FA3000 Section结束地址 = 00000000D2FBA000
22	0.00192790	Section开始地址 = 00000000D2FBB000 Section结束地址 = 00000000D2FBC000
23	0.00192910	Section开始地址 = 00000000D2FBD000 Section结束地址 = 00000000D2FC5000
24	0.00193030	Section开始地址 = 00000000D2FC6000 Section结束地址 = 00000000D2FC6000
25	0.00193150	Section开始地址 = 00000000D2FC7000 Section结束地址 = 00000000D2FE1000
26	0.00193280	Section开始地址 = 00000000D2FE2000 Section结束地址 = 00000000D2FE2000

RVA转换为FOA

将相对偏移地址转换为FOA文件偏移地址，此处仅仅只是多了一步 `pntHeaders->OptionalHeader.ImageBase + dwRVA` RVA转换为VA的过程其转换结果与VA转FOA一致。

```

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

    NTSTATUS status = STATUS_SUCCESS;
    HANDLE hFile = NULL;
    HANDLE hSection = NULL;
    PVOID pBaseAddress = NULL;
    UNICODE_STRING FileName = { 0 };

    // 初始化字符串
    RtlInitUnicodeString(&FileName, L"\\??\\C:\\windows\\System32\\ntoskrnl.exe");

    // 内存映射文件
    status = kernelMapFile(FileName, &hFile, &hSection, &pBaseAddress);
    if (!NT_SUCCESS(status))
    {
        return 0;
    }

    // 获取PE头数据集
    PIMAGE_DOS_HEADER pDosHeader = (PIMAGE_DOS_HEADER)pBaseAddress;
    PIMAGE_NT_HEADERS pntHeaders = (PIMAGE_NT_HEADERS)((PUCHAR)pDosHeader + pDosHeader->e_lfanew);
    PIMAGE_SECTION_HEADER psection = IMAGE_FIRST_SECTION(pntHeaders);
    PIMAGE_FILE_HEADER pFileHeader = &pntHeaders->FileHeader;

```

```

DWORD64 dwRVA = 0x89200;
DWORD64 ImageBase = pNtHeaders->OptionalHeader.ImageBase;
DWORD NumberOfSectinsCount = pNtHeaders->FileHeader.NumberOfSections;
DbgPrint("镜像基址 = %p | 节表数量 = %d \n", ImageBase, NumberOfSectinsCount);

for (DWORD each = 0; each < NumberOfSectinsCount; each++)
{
    DWORD Section_Start = pSection[each].VirtualAddress;
    // 计算RVA开始位置
    DWORD Section_Ends = pSection[each].VirtualAddress +
pSection[each].Misc.VirtualSize; // 计算RVA结束位置

    if (dwRVA >= Section_Start && dwRVA <= Section_Ends)
    {
        DWORD VA = pNtHeaders->OptionalHeader.ImageBase + dwRVA;
        // 得到VA地址
        DWORD FOA = pSection[each].PointerToRawData + (dwRVA -
pSection[each].VirtualAddress); // 得到FOA
        DbgPrint("RVA偏移 [ %p ] --> 对应FOA地址 [ %p ] \n", dwRVA, FOA);
    }
}

ZwUnmapViewOfSection(NtCurrentProcess(), pBaseAddress);
ZwClose(hSection);
ZwClose(hFile);

Driver->DriverUnload = UnDriver;
return STATUS_SUCCESS;
}

```

运行效果如下所示;

