

实验报告一

黄彦玮 (3180102067)

2019 年 11 月 6 日

1 推理过程

1.1 概述

在分别尝试三种方法之后，采用第一种思路，用 DestroyWindow 作切入点，OD 代码窗中按 ctrl+G 输入 DestroyWindow，按 F2 设断点，点击 Register 触发断点，按照 Ctrl+F9 快速运行至用户代码，利用 F7 和 F8 进行跟踪，通过观察上方代码进行推理，继续跟踪进行判断。

1.2 推理过程详细描述

1.2.1 寻找判断函数

1. OD 代码窗中按 ctrl+G 输入 DestroyWindow，按 F2 设断点，输入注册码“9876543287654321”，点击 Register，弹出“Bad”窗口。点击 ok 触发断点，如图 1 所示。



图 1: 触发断点

注意到当前断点地址为 77 开头，显然是系统内核函数，按 Ctrl+F9 快速运行。

2. 第一次跳转如图 2 所示，此时根据当前位置上一行判断应该是 Destroywindow 刚刚执行完毕处，继续按 Ctrl+F9。

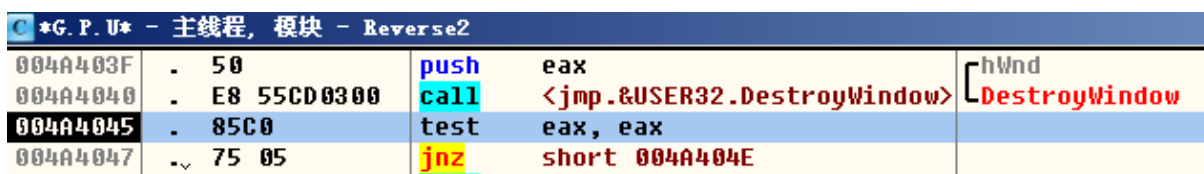


图 2: 第 1 次跳转

3. 第二次跳转如图 3 所示，发现位于 jmp 下方，判断应当是某处直接 call 或 jmp 而来，继续按 Ctrl+F9。

004A405E	. 66:8160 44 FF	and	word ptr [eax+0x44], 0xFDFF
004A4064	. C3	retn	
004A4065	~ E9 5ECB0200	jmp	004D0BC8
004A406A	. ^ EB EF	jmp	short 004A405B
004A406C	8B45 FC	mov	eax, [ebp-0x4]
004A406F	. 33D2	xor	edx, edx

图 3: 第 2 次跳转

4. 第三次跳转如图 4 所示，注意到上方有消息传输，但并没有明显的标志，所以暂时跳过，按 Ctrl+F9。

004901CF	. A1 AC7F4E00	mov	eax, [0x4E7FAC]	
004901D4	. 8B40 38	mov	eax, [eax+0x38]	
004901D7	. 8B80 3C020000	mov	eax, [eax+0x23C]	
004901DD	. 50	push	eax	
004901DE	. E8 570E0500	call	<jmp.&USER32.SendMessageA>	hWnd SendMessageA
004901E3	~ EB 07	jmp	short 004901EC	
004901E5	> 8BC3	mov	eax, ebx	
004901E7	. E8 2C3E0100	call	004A4018	
004901EC	> 33C0	xor	eax, eax	
004901EE	. 8983 3C020000	mov	[ebx+0x23C], eax	
004901F4	. 5B	pop	ebx	
004901F5	. C3	retn		

图 4: 第 3 次跳转

5. 第四次跳转如图 5 所示，注意到上方有条件跳转指令，但没有类似于将计算结果与某个字符串比较的函数，所以按 Ctrl+F9。（实际操作中保险起见我将上方的若干个 call 依次设置断点并验证，发现并没能阻值 Bad 的弹出，因此断定此处不是要找的地方。）

0048E0B1	. FF92 C4000000	call	[edx+0xC4]
0048E0B7	> 33D2	xor	edx, edx
0048E0B9	. 8B45 FC	mov	eax, [ebp-0x4]
0048E0BC	. E8 57260000	call	00490718
0048E0C1	. 8B45 FC	mov	eax, [ebp-0x4]
0048E0C4	. E8 278E0100	call	004A6EF0
0048E0C9	. 84C0	test	al, al
0048E0CB	~ 74 0B	je	short 0048E0D8
0048E0CD	. 8B45 FC	mov	eax, [ebp-0x4]
0048E0D0	. 8B10	mov	edx, [eax]
0048E0D2	. FF92 9C000000	call	[edx+0x9C]
0048E0D8	> 8B55 FC	mov	edx, [ebp-0x4]
0048E0DB	. 01 807E4E00	mov	eax, [0x4E7EB0]

堆栈 ss:[0012FAA4]=00AD2EF4
edx=00000000
跳转来自 0048E0CB

图 5: 第 4 次跳转

6. 第五次跳转如图 6 所示，显然是某处跳转而来，继续跳转。

0048E136	~ E9 8D2A0400	jmp	004D0BC8
0048E13B	^ EB EC	jmp	short 0048E129
0048E13D	. 807D FB 00	cmp	byte ptr [ebp-0x5], 0x0
0048E141	. 7E 08	jle	short 0048E148

图 6: 第 5 次跳转

7. 第六次跳转如图 7 所示，注意到上方有条件跳转指令，但同样没有类似于将计算结果与某个字符串比较的函数，所以按 Ctrl+F9。（实际操作中也对出现的 call 设置断点进行了验证，但没有发现）

00401AC7	. E8 A9EC0D00	call	004E0775
00401ACC	. 8BDA	mov	ebx, edx
00401ACE	. 8945 FC	mov	[ebp-0x4], eax
00401AD1	. B8 9C284E00	mov	eax, 004E289C
00401AD6	. E8 51220D00	call	004D3D2C
00401ADB	. C745 F4 0E0000	mov	dword ptr [ebp-0xC], 0xE
00401AE2	. 84DB	test	bl, bl
00401AE4	. 7C 14	j1	short 00401AFA
00401AE6	. 66:C745 E8 08	mov	word ptr [ebp-0x18], 0x8
00401AEC	. 836D F4 0E	sub	dword ptr [ebp-0xC], 0xE
00401AF0	. 33D2	xor	edx, edx
00401AF2	. 8B45 FC	mov	eax, [ebp-0x4]
00401AF5	. E8 72C50800	call	0048E06C
00401AFA	> 8B55 D8	mov	edx, [ebp-0x28]

图 7: 第 6 次跳转

8. 第七次跳转如图 8 所示，这一次代码上方出现的 call 和上一次一样，果断跳过。

00401BDF	. E8 91EB0D00	call	004E0775
00401BE4	. 8855 DB	mov	[ebp-0x25], dl
00401BE7	. 8BD8	mov	ebx, eax
00401BE9	. B8 B0284E00	mov	eax, 004E28B0
00401BEE	. E8 39210D00	call	004D3D2C
00401BF3	. 807D DB 00	cmp	byte ptr [ebp-0x25], 0x0
00401BF7	. 7C 0D	j1	short 00401C06
00401BF9	. 836D F8 0F	sub	dword ptr [ebp-0x8], 0xF
00401BFD	. 33D2	xor	edx, edx
00401BFF	. 8BC3	mov	eax, ebx
00401C01	. E8 BAFEFFFF	call	00401AC0
00401C06	> 8B55 DC	mov	edx, [ebp-0x24]

图 8: 第 7 次跳转

9. 第八次跳转如图 9 所示，上方的代码非常的长，有非常多可疑的 call，并且有 0x57、0x42、0xDEADBEEF 等类似于人为定义的变量，且有许多 cmp（虽然暂时还不能确定与字符串比较有关），于是我们在最上方设置断点跟踪进行观察。至此，我们找到了正确的位置。（实际操作中失败了数次，在后续的跳转中也反复尝试，但都没有结果）

00401708	. E8 CFFFFFFFFF	call	004015DC
0040170D	. 83C4 08	add	esp, 0x8
00401710	. 8843 01	mov	[ebx+0x1], al
00401713	. 8043 02 42	add	byte ptr [ebx+0x2], 0x42
00401717	. 806B 03 57	sub	byte ptr [ebx+0x3], 0x57
0040171B	. 8B4D C4	mov	ecx, [ebp-0x3C]
0040171E	. 81F1 EFBADDE	xor	ecx, 0xDEADBEEF
00401724	. 3B0D 40254E00	cmp	ecx, [0x4E2540]
0040172A	. 75 0C	jnz	short 00401738
0040172C	. C705 AC7D4E00	mov	dword ptr [0x4E7DAC], 004E2
00401736	. EB 0A	jmp	short 00401742
00401738	> C705 AC7D4E00	mov	dword ptr [0x4E7DAC], 004E2
0040173D	. 01 00704E00	mov	dword ptr [0x4E7000], 004E2

图 9: 第 8 次跳转

1.2.2 字符串分解

1. 在最上方的一个 `call:004015FF call 004D3D2C` 处设置断点,重新运行并输入 9876543287654321, 触发断点, 进入跟踪。
2. 跟踪最上面两个 `call` 没有发现, 但很快在 `00401625 cmp dword ptr [ebp-0x4], 0x0;` 处第一次发现了 `[ebp-0x4]` 为字符串“9876543287654321”, 这证明了这段代码与注册码有关。
3. 继续跟踪, 发现下面有若干个可疑函数, 按 F7 进入跟踪。虽然也在一些地方出现了若干次注册码字符串“9876543287654321”, 但并没有明显的处理的代码, 跟踪过程这里略去。
4. 一直跟着, 直到进入 `0040167C call 004E05B4`, 再进入 `004E05EA call 004C8BF8`。在这个函数内进入 `004C8C02 call 004D11D4`。执行完后发现 `eax` 从注册码字符串变为了 0x10, 恰好是字符串的长度 16, 但还并不能确定。
5. 继续跟踪, 如图 10 所示, 依次进入两个循环。第一个循环从 1 到 16, 每一次顺序取注册码字符串的一个字符, 和 0x20 比较, 小于则跳转。第二个循环从 16 到 1, 每一次倒序取对应的注册码字符串的字符, 和 0x20 比较, 小于则跳转。这两个条件都没有满足, 判断这一段代码的作用应当是判断是否含有非法字符。

004C8C0E	._ EB 01	jmp	short 004C8C11
004C8C10	> 43	inc	ebx
004C8C11	> 3BF3	cmp	esi, ebx
004C8C13	._ 7C 07	j1	short 004C8C1C
004C8C15	. 807C1F FF 20	cmp	byte ptr [edi+ebx-0x1], 0x20
004C8C1A	. ^ 76 F4	jbe	short 004C8C10
004C8C1C	> 3BF3	cmp	esi, ebx
004C8C1E	._ 7D 0A	jge	short 004C8C2A
004C8C20	. 8BC5	mov	eax, ebp
004C8C22	. E8 2D830000	call	004D0F54
004C8C27	._ EB 17	jmp	short 004C8C40
004C8C29	> 4E	dec	esi
004C8C2A	> 807C37 FF 20	cmp	byte ptr [edi+esi-0x1], 0x20
004C8C2F	. ^ 76 F8	jbe	short 004C8C29

图 10: 两个循环

6. 继续跟踪，进入 004C8C3B call 004D13DC。经过一段时间跟踪，发现一段可疑代码，如图 11 所示。其中，004CFE5F lea esi, [esi+ecx-0x4] 取到了注册码的后四位，004CFE67 sar ecx, 0x2 将注册码长度右移两位（变为 4），然后 004CFE6D rep movs dword ptr es:[edi], dword ptr [esi] 将注册码从后向前每隔 4 位转化为十进制储存在堆栈中。

004CFE5F	> 8D740E FC	lea	esi, [esi+ecx-0x4]	ecx=length,last 4
004CFE63	. 8D7C0F FC	lea	edi, [edi+ecx-0x4]	
004CFE67	. C1F9 02	sar	ecx, 0x2	
004CFE6A	._ 78 11	js	short 004CFE7D	
004CFE6C	. FD	std		
004CFE6D	. F3:A5	rep	movs dword ptr es:[edi], dword ptr [esi]	to decimal
004CFE6F	. 89C1	mov	ecx, eax	ecx=length
004CFE71	. 83E1 03	and	ecx, 0x3	
004CFE74	. 83C6 03	add	esi, 0x3	
004CFE77	. 83C7 03	add	edi, 0x3	
004CFE7A	. F3:A4	rep	movs byte ptr es:[edi], byte ptr [esi]	to decimal
004CFE7C	. FC	cld		
004CFE7D	> 5F	pop	edi	
004CFE7E	. 5E	pop	esi	edi=esi=reg code
004CFE7F	. C3	ret		

图 11: 一段可疑代码

7. 回到最外层，发现一段关键代码，如图 12 所示。

0040167C	. E8 33EF0D00	call	004E05B4	!!!!
00401681	. 837D F4 00	cmp	dword ptr [ebp-0xC], 0x0	cmp reg code
00401685	74 05	je	short 0040168C	
00401687	. 8B45 F4	mov	eax, [ebp-0xC]	eax=reg code
0040168A	EB 05	jmp	short 00401691	
0040168C	> B8 4D254E00	mov	eax, 004E254D	
00401691	> 8BF8	mov	edi, eax	
00401693	. 33C0	xor	eax, eax	edi=reg code, eax=0
00401695	. 83C9 FF	or	ecx, 0xFFFFFFFF	
00401698	. 8D95 60FFFFFF	lea	edx, [ebp-0xA0]	
0040169E	. F2:AE	repne	scas byte ptr es:[edi]	calculate length of reg code
004016A0	. F7D1	not	ecx	ecx=length
004016A2	. 2BF9	sub	edi, ecx	edi=reg code
004016A4	. 8BF2	mov	esi, edx	
004016A6	. 87FE	xchg	esi, edi	esi=reg code
004016A8	. 8BD1	mov	edx, ecx	edx=length
004016AA	. 8BC7	mov	eax, edi	
004016AC	. C1E9 02	shr	ecx, 0x2	length/4
004016AF	. 8D45 F4	lea	eax, [ebp-0xC]	
004016B2	. F3:A5	rep	movs dword ptr es:[edi], dword ptr [esi]	copy reg code to stack
004016B4	. 8BCA	mov	ecx, edx	ecx=length
004016B6	. BA 02000000	mov	edx, 0x2	
004016BB	. 83E1 03	and	ecx, 0x3	length&3
004016BE	. F3:A4	rep	movs byte ptr es:[edi], byte ptr [esi]	
004016C0	. FF4D E4	dec	dword ptr [ebp-0x1C]	
004016C3	. E8 A8EE0D00	call	004E0570	

图 12: 一段关键代码

这段代码首先判断注册码是否为空,然后出现了汇编语言中经典的计算字符串长度的语句 0040169E repne scas byte ptr es:[edi]。取反后这时 ecx 为长度 +1, 即 0x11。之后, 将长度赋给 edx, 并右移两位 (除以 4 后变为 4), 每 4 格为单位将注册码存进堆栈。

- 再向下跟踪, 发现一个带参数的函数, 如图 13 所示。这个参数的第 1 个参数为注册码字符串, 非常可疑。进入跟踪后发现一个五参数的函数 004D5F95 call 004D5474, 按 F7 进入跟踪, 如图 14 所示。

004016DB	. 8D85 60FFFFFF	lea	eax, [ebp-0xA0]	eax=reg code
004016E1	. 51	push	ecx	Arg3
004016E2	. 68 4E254E00	push	004E254E	Arg2 = 004E254E ASCII "%X"
004016E7	. 50	push	eax	Arg1 = reg code
004016E8	. E8 8F480D00	call	004D5F7C	Reverse2.004D5F7C

图 13: 带参数的函数

004D5F7F	. 8D45 10	lea	eax, [ebp+0x10]	<div>Arg5</div> <div>Arg4</div> <div>Arg3 = 0012FB0C</div> <div>Arg2 = 004D5F70</div> <div>Arg1 = 004D5F50</div> <div>!!!Reverse2.004D5474</div>
004D5F82	. 50	push	eax	
004D5F83	. 8B55 0C	mov	edx, [ebp+0xC]	
004D5F86	. 52	push	edx	
004D5F87	. 8D4D 08	lea	ecx, [ebp+0x8]	
004D5F8A	. 51	push	ecx	
004D5F8B	. 68 705F4D00	push	004D5F70	
004D5F90	. 68 505F4D00	push	004D5F50	
004D5F95	. E8 DAF4FFFF	call	004D5474	

图 14: 五参数的函数

9. 进入函数后发现很多 switch，经过多次跳转后在 default 部分找到一个关键函数：004D5771 call 004D5CAC。进入跟踪后进入一个很长的循环（局部如图 15 所示）。

004D5CDD	> FF45 F8	inc	dword ptr [ebp-0x8]	start processing!!!
004D5CE0	. 8B45 10	mov	eax, [ebp+0x10]	get the first element now
004D5CE3	. 50	push	eax	
004D5CE4	. FF55 08	call	[ebp+0x8]	
004D5CE7	. 59	pop	ecx	
004D5CE8	. 8B08	mov	ebx, eax	
004D5CEA	. 85FF	test	edi, edi	
004D5CEC	~ 75 1C	jnz	short 004D5D0A	
004D5CEE	. 85DB	test	ebx, ebx	
004D5CF0	~ 7C 13	jl	short 004D5D05	
004D5CF2	. 81FB 80000000	cmp	ebx, 0x80	
004D5CF8	~ 7D 0B	jge	short 004D5D05	Arg1 = the first element Reverse2.004D51B8
004D5CFA	. 53	push	ebx	
004D5CFB	. E8 B8540000	call	004D51B8	

图 15: 循环局部图

10. 该循环从第一行 004D5CDD 开始，下面有一行 004D5CE4 call [ebp+0x8]，进入代码如图 16 所示。

004D5F50	. 55	push	ebp	edx=reg code the first element
004D5F51	. 8BEC	mov	ebp, esp	
004D5F53	. 8B45 08	mov	eax, [ebp+0x8]	
004D5F56	. 8B10	mov	edx, [eax]	
004D5F58	. FF00	inc	dword ptr [eax]	
004D5F5A	. 8A02	mov	al, [edx]	
004D5F5C	. 84C0	test	al, al	
004D5F5E	~ 75 05	jnz	short 004D5F65	
004D5F60	. 83C9 FF	or	ecx, 0xFFFFFFFF	
004D5F63	~ EB 04	jmp	short 004D5F69	
004D5F65	> 33C9	xor	ecx, ecx	cl=the first element
004D5F67	. 8AC8	mov	cl, al	
004D5F69	> 8BC1	mov	eax, ecx	
004D5F6B	. 5D	pop	ebp	
004D5F6C	. C3	ret		

图 16: 提取注册码第一位的函数

经过模拟不难发现该代码的逻辑为：先将注册码字符串尚未处理的部分赋给 edx，然后利用 al 提取 edx 的第一个元素，再赋给 cl，此时 ecx 寄存器中的就是剩余字符串的第一个元素。这里是第一次运行，得到的 ecx 为 00000039（ASCII'9'）。

- 回到上一层的循环，经过一段复杂的代码后来到了串很长的判断出，如图 17 所示。这时 ebx 存储的是剩余字符串的第一个元素，发现这里将它与 0x30 和 0x39 相比较，然后减去 0x30，不难判断这里是在处理字符串，即和字符'0'与'9'比较，然后减去'0'转化为整数，因此可以放心地跳过。

004D5D95	~v 0F8C E6000000	j1	004D5E81	
004D5D98	. 83FE 24	cmp	esi, 0x24	
004D5D9E	~v 0F8F DD000000	jg	004D5E81	
004D5DA4	> 83FF 03	cmp	edi, 0x3	
004D5DA7	^ 0F85 30FFFFFF	jnz	004D5CDD	
004D5DAD	. 83FB 30	cmp	ebx, 0x30	
004D5DB0	~v 7C 0D	j1	short 004D5DBF	
004D5DB2	. 83FB 39	cmp	ebx, 0x39	>='0'&&<='9'
004D5DB5	~v 7F 08	jg	short 004D5DBF	
004D5DB7	. 8D43 D0	lea	eax, [ebx-0x30]	-'0'
004D5DBA	. 8945 E4	mov	[ebp-0x1C], eax	
004D5DBD	~v EB 2A	jmp	short 004D5DE9	

图 17: 处理注册码

- 沿着循环走，发现如图 18 的代码。进入 004D5E0C call 004D920C 跟踪，函数部分如图 19 所示。

004D5E04	. 52	push	edx	
004D5E05	. 50	push	eax	
004D5E06	. 8B45 E8	mov	eax, [ebp-0x18]	
004D5E09	. 8B55 EC	mov	edx, [ebp-0x14]	
004D5E0C	. E8 FB330000	call	004D920C	
004D5E11	. 52	push	edx	
004D5E12	. 50	push	eax	
004D5E13	. 8B45 E4	mov	eax, [ebp-0x1C]	
004D5E16	. 99	cdq		
004D5E17	. 030424	add	eax, [esp]	last * 0x10 + this
004D5E1A	. 135424 04	adc	edx, [esp+0x4]	edx is flow out

图 18: 字符串分解（外部）

004D920C	\$ 52	push	edx	
004D920D	. 50	push	eax	
004D920E	. 8B4424 10	mov	eax, [esp+0x10]	
004D9212	. F72424	mul	dword ptr [esp]	
004D9215	. 8BC8	mov	ecx, eax	
004D9217	. 8B4424 04	mov	eax, [esp+0x4]	
004D921B	. F76424 0C	mul	dword ptr [esp+0xC]	
004D921F	. 03C8	add	ecx, eax	
004D9221	. 8B0424	mov	eax, [esp]	
004D9224	. F76424 0C	mul	dword ptr [esp+0xC]	last element * 0x10
004D9228	. 03D1	add	edx, ecx	
004D922A	. 59	pop	ecx	
004D922B	. 59	pop	ecx	
004D922C	. C2 0800	retn	0x8	

图 19: 字符串分解（内部）

经过多次模拟（沿循环执行多次），以第 4 次为例，图 19 中 004D9221 mov eax, [esp] 执行完后，eax 的值为 0x987，下一行 004D9224 mul dword ptr [esp+0xC] 中被乘数的值为 0x10，因此执行后 eax 变为 0x9870。回到函数外，图 18 中 004D5E17 add eax, [esp] 的作用是将 eax 加上 [esp]，此时后者的值就是 0x9870，前者的值为 0x6，执行完后为 0x9876。当位数超过 8 位以后，发现 eax 中只储存最后 8 位，溢出部分储存在 edx 中（即第 004D5E1A 行执行的结果）。

13. 继续跟踪，进入 004D5E41 call 004D92A7，如图 20 所示。

004D92BB	. 0BDB	or	ebx, ebx	
004D92BD	~ 74 2B	je	short 004D92EA	
004D92BF	> 8BE9	mov	ebp, ecx	
004D92C1	. B9 40000000	mov	ecx, 0x40	
004D92C6	. 33FF	xor	edi, edi	
004D92C8	. 33F6	xor	esi, esi	
004D92CA	> D1E0	shl	eax, 1	
004D92CC	. D1D2	rcl	edx, 1	
004D92CE	. D1D6	rcl	esi, 1	
004D92D0	. D1D7	rcl	edi, 1	
004D92D2	. 3BFD	cmp	edi, ebp	
004D92D4	~ 72 0B	jb	short 004D92E1	
004D92D6	~ 77 04	ja	short 004D92DC	
004D92D8	. 3BF3	cmp	esi, ebx	
004D92DA	~ 72 05	jb	short 004D92E1	
004D92DC	> 2BF3	sub	esi, ebx	
004D92DE	. 1BFD	sbb	edi, ebp	
004D92E0	. 40	inc	eax	
004D92E1	> E2 E7	loopd	short 004D92CA	take the nearest 8 elements
004D92E3	> 5F	pop	edi	
004D92E4	. 5E	pop	esi	
004D92E5	. 5B	pop	ebx	
004D92E6	. 5D	pop	ebp	
004D92E7	. C2 0800	retn	0x8	

图 20: 字符串分解（后续）

这是一个长达 64 次的循环，快速执行完并后发现 eax 中保存的是离当前处理的数位最近的 8 个数位组成的数，edx 储存溢出数位组成的数。整个循环执行结束后，eax 为 87654321，edx 为 98765432，与预期结果相符。

14. 回到函数外，将 eax（87654321）和 edx（98765432）分别存进内存，如图 21 所示。

004D5771	. E8 36050000	call	004D5CAC	???
004D5776	. 83C4 1C	add	esp, 0x1C	
004D5779	. 8945 D8	mov	[ebp-0x28], eax	87654321
004D577C	. 8955 DC	mov	[ebp-0x24], edx	98765432

图 21: 字符串分解完成

15. 至此，我们可以得出结论，图 22 所示的五参数函数是本程序的第一个核心函数，004016E8 call 004D5F7C 的作用是将注册码字符串分解，前 8 位储存在 edx 中，后 8 位储存在 eax 中。

004016D8	. 8D85 60FFFFFF	lea	eax, [ebp-0xA0]	eax=reg code
004016E1	. 51	push	ecx	Arg3
004016E2	. 68 4E254E00	push	004E254E	Arg2 = 004E254E ASCII "%X"
004016E7	. 50	push	eax	Arg1 = reg code
004016E8	. E8 8F480D00	call	004D5F7C	Reverse2.004D5F7C

图 22: 第一个核心函数

1.2.3 注册码核心运算

1. 继续向下跟踪，发现如图 23 所示的可疑代码。该函数有两个参量，一个固定为 0x2，另一个发现是注册码的最后两位，这里为'21'。于是进入跟踪。

004016ED	. 83C4 0C	add	esp, 0xC	
004016F0	. 8D5D C4	lea	ebx, [ebp-0x3C]	
004016F3	. 6A 02	push	0x2	Arg2 = 00000002
004016F5	. 8A03	mov	al, [ebx]	last 2 elements
004016F7	. 50	push	eax	eax=last 2 elements
004016F8	. E8 CFFEFFFF	call	004015CC	left shift 2

图 23: 可疑代码

2. 函数部分如图 24 所示。很容易发现 al 和 cl 分别是注册码最后两位和固定值 0x2，即函数的两个参数。下面一句代码 004015D7 rol al, cl 很明显是将注册码最后两位左移 2 位。

004015CC	\$ 55	push	ebp
004015CD	. 8BEC	mov	ebp, esp
004015CF	. 33C0	xor	eax, eax
004015D1	. 8A45 08	mov	al, [ebp+0x8]
004015D4	. 8A4D 0C	mov	cl, [ebp+0xC]
004015D7	. D2C0	rol	al, cl
004015D9	. 5D	pop	ebp
004015DA	. C3	retn	

图 24: 左移函数

3. 回到函数外, 00401700 mov [ebx], al 将堆栈内注册码后两位改为左移后的值。至此, 我们可以得出结论: 图 25 所示的代码段是本程序的第二个核心函数, 即将注册码最后两位左移两位。

004016F3	. 6A 02	push	0x2	Arg2 = 00000002 last 2 elements eax=last 2 elements left shift 2
004016F5	. 8A03	mov	al, [ebx]	
004016F7	. 50	push	eax	
004016F8	. E8 CFFEFFFF	call	004015CC	
004016FD	. 83C4 08	add	esp, 0x8	
00401700	. 8803	mov	[ebx], al	

图 25: 第二个核心函数

4. 继续向下跟踪, 发现如图 26 所示的可疑代码。该函数有两个参量, 一个固定为 0x3, 另一个发现是注册码的倒数第 3、4 位, 这里为'43'。于是进入跟踪。

00401702	. 6A 03	push	0x3	Arg2 = 00000003 last 3&4 Arg1 right shift 3
00401704	. 8A53 01	mov	dl, [ebx+0x1]	
00401707	. 52	push	edx	
00401708	. E8 CFFEFFFF	call	004015DC	
0040170D	. 83C4 08	add	esp, 0x8	
00401710	. 8843 01	mov	[ebx+0x1], al	

图 26: 可疑代码

5. 函数部分如图 27 所示。很容易发现 al 和 cl 分别是注册码倒数第 3、4 位和固定值 0x3, 即函数的两个参数。下面一句代码 004015E7 ror al, cl 很明显是将注册码倒数第 3、4 位右移 3 位。

004015DC	\$ 55	push ebp
004015DD	. 8BEC	mov ebp, esp
004015DF	. 33C0	xor eax, eax
004015E1	. 8A45 08	mov al, [ebp+0x8]
004015E4	. 8A4D 0C	mov cl, [ebp+0xC]
004015E7	. D2C8	ror al, cl
004015E9	. 5D	pop ebp
004015EA	. C3	ret

图 27: 右移函数

6. 回到函数外, 00401710 mov [ebx+0x1], al 将堆栈内注册码倒数第 3、4 位改为右移后的值。至此, 我们可以得出结论: 图 28 所示的代码段是本程序的第三个核心函数, 即将注册码倒数第 3、4 位右移 3 位。

00401702	. 6A 03	push 0x3	Arg2 = 00000003 last 3&4 Arg1 right shift 3
00401704	. 8A53 01	mov dl, [ebx+0x1]	
00401707	. 52	push edx	
00401708	. E8 CFFEFFFF	call 004015DC	
0040170D	. 83C4 08	add esp, 0x8	
00401710	. 8843 01	mov [ebx+0x1], al	

图 28: 第三个核心函数

7. 之后的代码就非常容易理解了, 如图 29 所示。00401713 add byte ptr [ebx+0x2],0x42 显然是将注册码倒数第 5、6 位组成的数加上 0x42 并保存到堆栈, 00401717 sub byte ptr [ebx+0x3],0x57 显然是将注册码倒数第 7、8 位组成的数减去 0x57 并保存到堆栈。这时堆栈中 [ebx] 位置储存的就是注册码计算后的结果, 这里为 30A76884。

接下来,将这个结果赋给 ecx,0040171E xor ecx,0xDEADBEEF 将 ecx 异或 0xDEADBEEF, 00401724 cmp ecx,[0x4E2540] 将 ecx 与一个数比较, 发现这个数为 5E7F5EE6, 很容易发现就是 Machine Code。

根据比较的结果不同, 对应了两种不同的跳转, 结合后面的 ASCII 码猜想这就是加密后的“成功”信息与“失败”信息, 再根据 jnz 判断出上面的是“成功”, 下面的是“失败”。

至此, 我们得出了注册码计算及比较的全部逻辑。

00401713	- 8043 02 42	add	byte ptr [ebx+0x2], 0x42	last 5&6 +0x42
00401717	- 806B 03 57	sub	byte ptr [ebx+0x3], 0x57	last 7&8 -0x57
0040171B	- 8B4D C4	mov	ecx, [ebp-0x3C]	
0040171E	- 81F1 EFBADDE	xor	ecx, 0xDEADBEEF	result xor 0xDEADBEEF
00401724	- 3B0D 40254E00	cmp	ecx, [0x4E2540]	ecx=EE0AD66B ds[]=5E7F5EE6
0040172A	- 75 0C	jnz	short 00401738	
0040172C	- C705 AC7D4E00	mov	dword ptr [0x4E7DAC], 004E28B8	ASCII "秋镁"
00401736	- EB 0A	jmp	short 00401742	
00401738	> C705 AC7D4E00	mov	dword ptr [0x4E7DAC], 004E28BD	ASCII "笋?"

图 29: 修改完成并判断

1.2.4 总结

总结上述步骤，可以得出注册机原理如下：首先，将注册码字符串拆分，只取最后 8 位，然后将第 1、2 位减去 0x57，第 3、4 位加上 0x42，第 5、6 位右移 3 位，第 7、8 位左移 2 位，然后将得到的数异或 0xDEADBEEF，再与机器码比较，若两者相同则注册成功。

同理，我们可以逆推出正确的注册码，即先将机器码异或 0xDEADBEEF，然后将第 1、2 位加上 0x57，第 3、4 位减去 0x42，第 5、6 位左移 3 位，第 7、8 位右移 2 位，即得到正确注册码。

本实验中，“5E7F5EE6”对应正确的注册码为“****D7900742”（前面的数位可任取，但至多 8 位）。实验结果如图 30 所示。

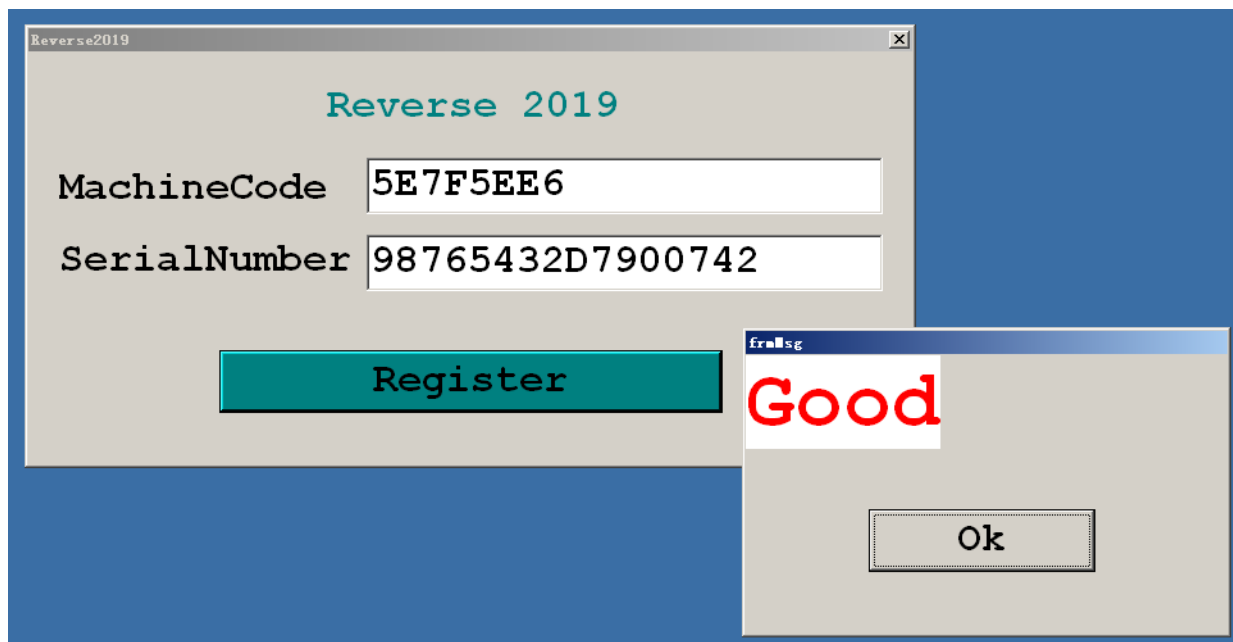


图 30: 实验成功

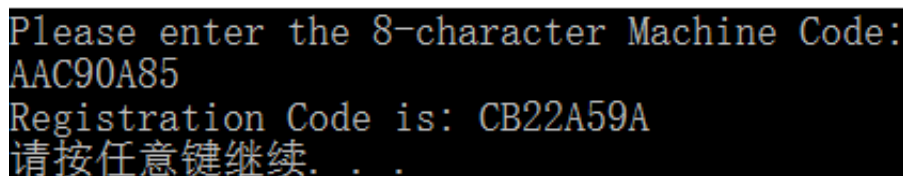
2 注册机编写

根据 1.2.4 节的原理可以编写出如下注册机代码：

```
#include "stdafx.h"
#include <stdlib.h>

unsigned int x;
int main(int argc, char* argv[])
{
    printf("Please enter the 8-character Machine Code:\n");
    scanf("%X",&x);
    x^=0xDEADBEEF;
    unsigned char y;
    y=x%256;x-=y;
    y=(y<<6)|(y>>2);x+=y;
    y=x%65536/256;x-=y*(1<<8);
    y=(y>>5)|(y<<3);x+=y*256;
    y=x/65536%256;x-=y*(1<<16);
    y-=0x42;x+=y*65536;
    y=x/65536/256;x-=y*(1<<24);
    y+=0x57;x+=y*(1<<24);
    printf("Registration Code is: %X\n",x);
    system("pause");
    return 0;
}
```

在另一台计算机上运行结果：机器码为 AAC90A85，得到的注册码为 CB22A59A，输入后得到 Good。如图 31、32 所示。



```
Please enter the 8-character Machine Code:
AAC90A85
Registration Code is: CB22A59A
请按任意键继续...
```

图 31: 得到注册码

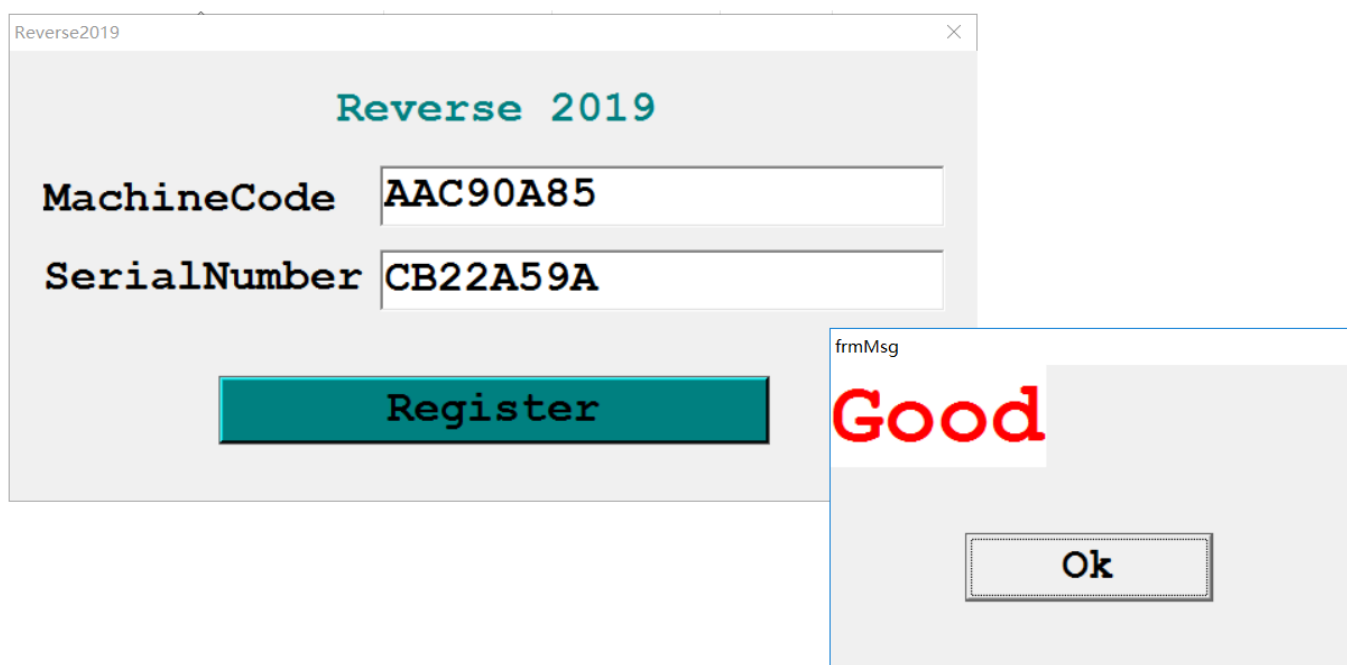


图 32: 实验成功

3 实验心得

这次实验做了我接近一整天的时间，非常好地锻炼了我的耐心以及推理能力。实验过程中遇到了很多困难，比如一开始 Ctrl+F9 快速运行一直找不到正确的代码段（正确代码段中的处理函数有一个参数为“%X”，导致我一直以为是 scanf）。后期跟踪的过程还是比较顺利的，没有遇到特别大的困难，只是比较考验耐心。最后做出来结果也是非常开心的。这次实验非常大的培养了我对这门课的兴趣，今后要继续努力学到更多的技能！