

说 明

《加密与解密》（第四版）中附录 C 有对 Visual Basic 程序逆向进行阐述，但由于篇幅限制，对 VB 的 P-code 未进行深入探讨。《软件加密技术内幕》一书中周文雄撰写的“第 8 章 Visual Basic 6 逆向工程”对 VB 逆向进行了深入的探讨。由于该书已停止印刷，为了方便大家研究 VB 程序，特将电子版放上来，供大家参考。

(c) 看雪学院 www.kanxue.com 2000-2018

目 录

第 8 章 Visual Basic 6 逆向工程.....	3
8.1 P-code 传奇.....	3
8.2 VB 编译奥秘	4
8.3 VB 与 COM	6
8.4 VB 可执行程序结构研究.....	10
8.5 VB 程序事件解读.....	20
8.6 VB 程序图形界面解读.....	23
8.7 VB 执行代码研究.....	28
8.7.1 VB 函数的解读.....	28
8.7.2 VB 函数调用约定.....	30
8.7.3 执行代码中对控件属性的操作	31
8.8 P-code 代码.....	36
8.8.1 理解 P-code 代码指令	37
8.8.2 P-code 程序调用约定	38
8.8.3 调试时中断 P-code 程序的几种方法	38
8.8.4 WKT VB Debugger 实现原理	39
8.8.5 VB6 P-code Crackme 分析实例	45
8.9 VB 程序保护篇	50
8.9.1 Anti-Loader 技术	50
8.9.2 VB“自锁”功能实现	53
8.10 相关工具点评	53

第 8 章 Visual Basic 6 逆向工程^①

自从微软公司推出 Visual Basic 5/6 版本以来，摆脱了以前只是单纯调用 `vbrun*.dll` 的方式。在继续保留 P-code 编译的同时也引入了 Native 编译方式，使得生成本地二进制代码成为可能。

本文将介绍 VB6 的反编译研究情况，也将演示如何保护 VB6 程序。关于 VB.net，也就是所谓的“VB7”，其生成的 EXE 程序完全可用 .NET 反编译工具完成，此处不再介绍。

8.1 P-code 传奇

编译器的编译技术可以分为 Native-Compile（自然编译）与 Pcode-Compile（伪编译）两种。

自然编译是编译器将高级语言转换为汇编代码，并经链接生成 EXE 程序的过程。

伪编译是编译器将高级语言转换为某种编码后，将能解释、执行此编码的一段程序一同链接，生成 EXE 程序的过程。

伪代码 P-code，最早应该叫做 Pascal-Code，其名称起源于一个 Pascal 编译器使用的“中间代码”编译技术。现在则一般解释为 Pseudo-Code（伪代码），或 Packed-Code（压缩代码）。此项编译技术的出现，最终导致了“虚拟机”的出现。而微软在其编程工具 Basic，C 中都使用了类似的编译技术。

伪代码的基本工作原理是编译器先把执行程序编译成比 80x86 机器码紧凑得多的中间代码形式，然后在链接时把一个小工作引擎嵌入执行程序中，最后在运行时由此工作引擎把 P-code 解释为本地机器码实际执行，所以叫做 Packed-Code。同时又由于此代码并不是最终的机器码形式，实际上是“变形的源代码”，所以也被称为 Pseudo-Code。依靠 P-code 编译技术，使得编程语言不依赖于机器或系统平台成为可能。

^① 作者：周文雄，网名小楼。学习软件加密与解密，研究 Visual Basic 6 反编译 2 年余，皆业余爱好。现在喜欢编程，探索编译器的奥秘。或问：先生何至于此？曰：无他，惟专一耳。《阴符》有云：“绝利一源，用师十倍”。吾能达于此。

目前实现伪代码编码方式的最流行方法是在特定硬件系统（比如 x86，Mac）上，用基于“堆栈（Stack）”的字节码编码实现。

虽然伪代码编译形式大多见于 VB，但 Java，PowerBuilder 的编译实质也是一样的。最新的 Microsoft Visual Studio .NET 中，只是将名称换成“中间语言（Microsoft Intermediate Language）”，笔者只能将 IL 语言理解为是 P-code + Native 编译方式（参考 http://windows.oreilly.com/news/hejlsberg_0800.html 《Deep Inside C#: An Interview with Microsoft Chief Architect Anders Hejlsberg》）。

因为 VB3 和 VB4 都使用伪代码编译形式，伪代码编译的代码实际只是“变形的源代码”，所以只要能理解其对应机制，就能做出反编译器来。例如 Dodi's VB 3/4 Disassembler。同样，Java 也有反编译器，对付最新.NET 的反编译程序也可以被找到。

VB 5/6 伪代码反编译器现在已经有了几个“半成品”，就是 Exdec 及建立在其基础上的 Wkt vb P-code debugger，还有 VBDE；2001 年也有人写出一个 VB6 add-in 来实现 VB6 源代码到 P-code 的转换。

8.2 VB 编译奥秘

采用伪代码编译时，每个 VB 源文件（包括.frm，.bas，.cls 文件）经 VB IDE 编译后各自生成相应的.obj 文件，由链接程序 Link.exe 生成伪编译的可执行文件（EXE，DLL，OCX 等），如图 8.1 所示。

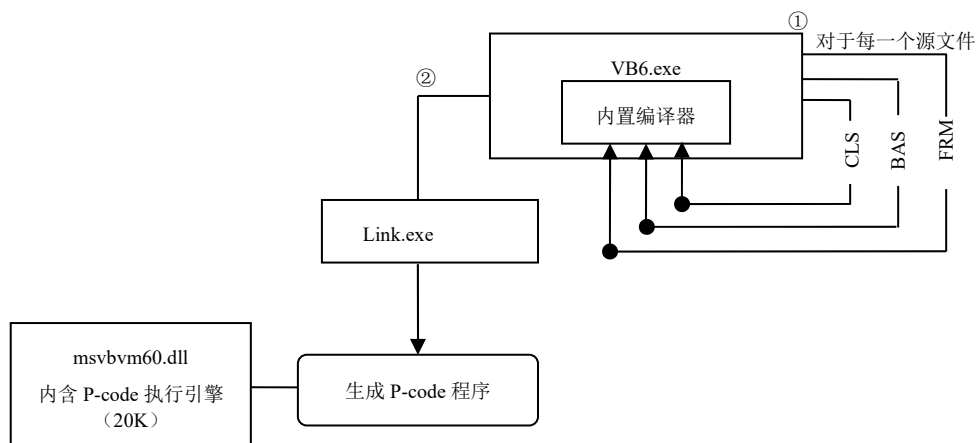


图 8.1 VB6 伪编译过程

而自然编译时，每个 VB 源文件由 C2.exe 编译生成汇编代码，生成相应的 obj 文件，再由 Link.exe 链接成为完整的可执行文件，如图 8.2 所示。

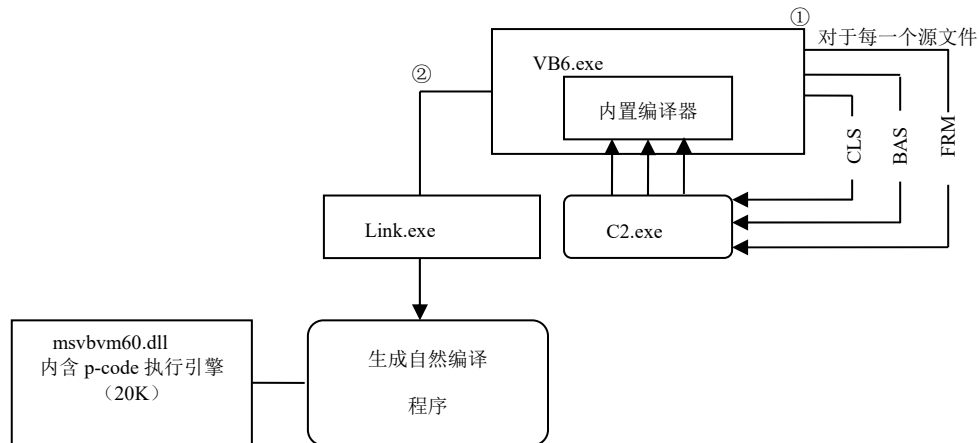


图 8.2 VB6 自然编译过程

熟悉 VC6 的读者可以发现，在 VC6 安装目录下也存在 C2.dll 及 Link.exe 文件，查看 C2.dll 与 C2.exe 属性，都表明是微软的 32 位编译程序。比较可以发现 VB 与 VC 版本的编译、链接程序具有一致性，如表 8-1 所示。

表 8-1 VB 与 VC 的编译、链接程序

文 件	版 本
C2.EXE (Visual Basic 5)	5.00.0.7182
C2.EXE (Visual Basic 6 SP5)	6.00.8783
C2.DLL (Visual C++ 6)	6.00.8168
Link.exe (Visual Basic 6 SP5)	6.00.8447
Link.exe (Visual C++ 6)	6.00.8168

读者可以用 VC.net 所带的 Link.exe 替换 VB6 中的同名程序，这样链接生成的可执行文件就不能被 VBDE 所分析了。

8.3 VB 与 COM

VB5/6 与 COM（组件对象模型，Components Object Model）是紧密相连的，可以认为 VB 程序是 COM 的客户端，而 msvbvm60.dll 就是服务器端。

在谈 COM 之前，有必要先说说 C++ 语言中的虚函数表（Virtual Function Table, VFT）。简单地说，虚函数就是用 **virtual** 关键字声明的函数。实现虚函数是通过函数指针进行的。编译时，编译器为包含虚函数的类建立虚函数表，依次按照函数声明次序放置类的特定虚函数的地址；每个这样的地址叫做虚函数指针 VPTR。

以下是用 C++ 程序示例：

```
// 源程序见光盘:\chap08\example1\example1.cpp

#include <iostream>

using namespace std;

class Base {
public:
    Base() {
        cout << "In Base" << endl;

        cout << "Virtual Pointer = "
            << (int*)this << endl;

        cout << "Address of Vtable = "
            << (int*)((int*)this) << endl;

        cout << "Value at Vtable 1st entry = "
            << (int*)((int*)((int*)this+0)) << endl;

        cout << "Value at Vtable 2nd entry = "
            << (int*)((int*)((int*)this+1)) << endl;

        cout << "Value at Vtable 3rd entry = "
            << (int*)((int*)((int*)this+2)) << endl;

        cout << endl;
    }

    virtual void fl() { cout << "Base::fl" << endl; }
```

```
virtual void f2() { cout << "Base::f2" << endl; }

virtual void f3() { cout << "Base::f3" << endl; }

};

int main() {

    Base d;

    system("PAUSE");

    return 0;

}
```

经用 Dev C++ v4.98 编译后，运行结果如图 8.3 所示。

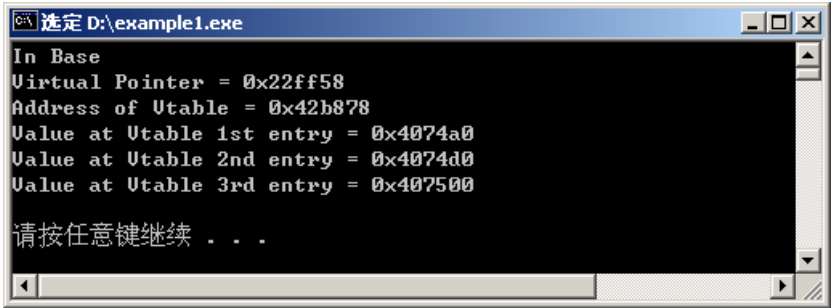


图 8.3 实例的运行结果

从 0x42B878 到 0x42B884 这 12 个 byte 数据段就是 VFT，而 0x4074a0, 0x4074d0, 0x407500 就是 VPtr。

理解了 VFT 的二进制存储结构，再理解 COM 的二进制存储结构就容易了。简单地说，COM 结构也是一个虚函数，只是这个虚函数的最前面三位必定依次是函数指针 QueryInterFace()、AddRef()、Release()，所以其二进制存储结构也就是一张有固定开头的 VFT，如图 8.4 所示。

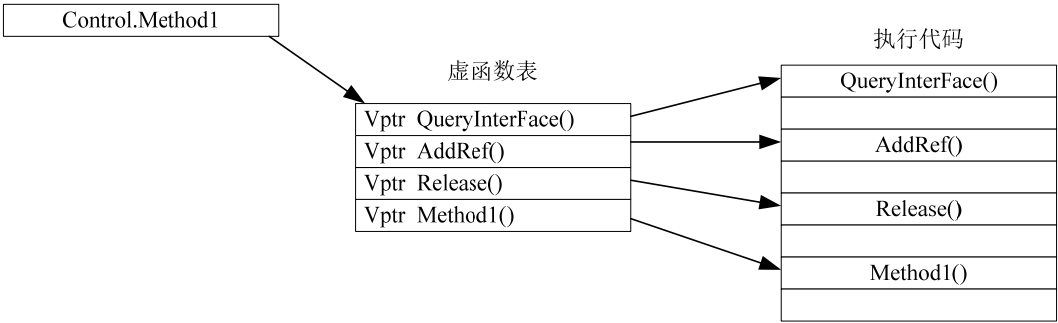


图 8.4 COM 的二进制存储结构

对于 VB6 编译的 EXE 程序来说，不论是伪编译还是自然编译，都至少需要调用 msvbvm60.dll 中 4 个函数：

EVENT_SINK_QueryInterface

EVENT_SINK_AddRef

EVENT_SINK_Release

__vbaExceptionHandler

前三个函数可认为是对 COM 结构的模拟。函数 __vbaExceptionHandler 是处理“异常 (Exception)”的代码段，编译器会自动生成。

编程时每个 VB 控件就是一个组件，编译后都会生成各自独立的虚函数表，存储其事件的执行代码地址。

Example2-P.exe 是个 VB6 P-code 程序。用 W32dasm 反汇编后跳至入口点代码处。显示汇编代码如下：

```
//程序见光盘: \chap08\example2\example2-p.exe
* Reference To: MSVBVM60.EVENT_SINK_AddRef, Ord:0000h
:00401032 FF2508104000          Jmp dword ptr [00401008]

* Reference To: MSVBVM60.EVENT_SINK_Release, Ord:0000h
:00401038 FF250C104000          Jmp dword ptr [0040100C]

* Reference To: MSVBVM60.MethCallEngine, Ord:0000h
:0040103E FF2500104000          Jmp dword ptr [00401000]

* Reference To: MSVBVM60.ThunRTMain, Ord:0064h
:00401044 FF2518104000          Jmp dword ptr [00401018]
:0040104A 0000                    add byte ptr [eax], al

//***** Program Entry Point *****
:0040104C 68BC114000          push 004011BC

* Reference To: MSVBVM60.ThunRTMain, Ord:0064h
:00401051 E8EEFFFFFF          Call 00401044
```

注意到转入 EVENT_SINK_Release 那行的地址 0x401038，用 W32dasm 搜索此地

址，可以找到两处：

(1)			
:004014FC	00000000	DWORD 00000000	
:00401500	A8144000	DWORD 004014A8	;指针指向的结构说明是 Form 控件
:00401504	1C144000	DWORD 0040141C	;与(2)此处内容相同，表明它们来源于同一 frm ; (或 bas、cls) 文件
:00401508	2C104000	DWORD 0040102C	;Ptr EVENT_SINK_QueryInterface
:0040150C	32104000	DWORD 00401032	;Ptr EVENT_SINK_AddRef
:00401510	38104000	DWORD 00401038	;Ptr EVENT_SINK_Release
:00401514	00000000	DWORD 00000000	
.....			
.....			
:0040158C	00000000	DWORD 00000000	
(2)			
:00401590	00000000	DWORD 00000000	
:00401594	D0144000	DWORD 004014D0	;说明是 CommandButton 控件
:00401598	1C144000	DWORD 0040141C	;与(1)此处内容相同，表明它们来源于同一 frm ; (或 bas、cls) 文件
:0040159C	2C104000	DWORD 0040102C	;Ptr EVENT_SINK_QueryInterface
:004015A0	32104000	DWORD 00401032	;Ptr EVENT_SINK_AddRef
:004015A4	38104000	DWORD 00401038	;Ptr EVENT_SINK_Release
:004015A8	BC154000	DWORD 004015EC	;Ptr Command1_Click()
:004015AC	00000000	DWORD 00000000	
.....			
.....			
:004015E8	00000000	DWORD 00000000	

上述显示了 example2-P.exe 事件中的两个控件 Form1 及 CommandButton1 处理事件的虚函数表。位于 Ptr EVENT_SINK_Release 后的即是指向控件的特定事件的执行代码的指针列表。若此事件存在，则有具体地址指针填入，否则则是空指针（00000000）。

不论是自然编译还是伪编译，上述结构都是相同的。

8.4 VB 可执行程序结构研究

对于 VB5/6 编译生成的程序，不管是自然编译还是伪编译，其程序入口点处的结构都是一样的。用 W32dasm 反汇编 example2-P.exe，来到 OEP 处：

```

//***** Program Entry Point *****

:0040104C 68BC114000          push 004011BC

* Reference To: MSVBVM60.ThunRTMain, Ord:0064h
|

:00401051 E8EEFFFFFF          Call 00401044
```

指针 0x4011BC 指向的结构就是 VB 程序的 VBHeader 结构，或者有些人将之称为 VBExeInitTable 结构。由此伸展开去，整个 VB 程序的框架就尽在此中了。VBHeader 结构对应的是 vbp 文件。

虽然对 VB 程序框架的完全定义还存在很多争论和未知，但已经初步定义了以下结构（以下资料主要来源于 <http://www.vb-decompiler.com>，并根据笔者研究进行了校正）。

1. VB 程序框架结构

可能的 VB 程序框架如图 8.5 所示。

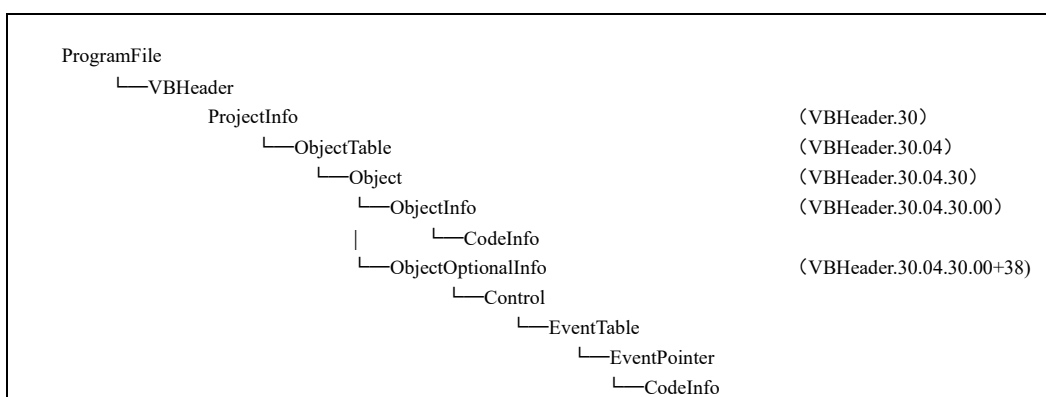


图 8.5 可能的 VB 程序框架

2. TVBHeader 结构定义

Const

VBHEAD_SIGNATURE = \$21354256 ; {VB5!}

Type

PVBHEAD = ^tVBHEAD;

tVBHEAD = packed record

Signature	array [1..4] of char;	// 00H, 签名, 必须为'VB5!'(VBHead)
Flag1	WORD;	// 04H,未知标志
LanguageDll	array [1..14] of char;	// 06H,语言链接库名。通常为"*" 或者 vb6chs.dll
BackupLanguageDll	array [1..14] of char;	// 14H,后备语言链接库名
RuntimeDLLVersion	WORD;	// 22H,运行库版本
LanguageId	DWORD;	// 24H,语言标识
BackupLanguageID	DWORD;	// 28H,未知标志
aSubMain	DWORD;	// 2CH,不为 00000000 则指向 Sub Main()指针 ^①
aProjectInfo	DWORD;	// 30H,指针, 指向 tProjectInfo 结构
Flag2	WORD;	// 34H,未知标志
Flag3	WORD;	// 36H,未知标志
Flag4	DWORD;	// 38H,未知标志
ThreadSpace	DWORD;	// 3CH,线程空间
Flag5	DWORD;	// 40H,未知标志
ResCount	WORD;	// 44H,数量, 表示 form 与 cls 文件个数 ^②
ImportCount	WORD;	// 46H,数量, 引用的 ocx、dll 文件个数 ^③
Flag6	BYTE;	// 48H,未知, 可能代表运行时程序所占内存大小
Flag7	BYTE;	// 49H,未知, 可能与程序启动时花费时间有关
Flag8	WORD;	// 4AH,未知
AGUITable	DWORD;	// 4CH,指针, 指向 Form GUI 描述表
AExtCompTable	DWORD;	// 50H,指针, 指向“引入的 ocx、dll 文件”描述表 ^④

AProjectDescription	DWORD;	// 54H,指针, 指向 tProjectDescriptionTable 的指针
OProjectExename	DWORD;	// 58H,偏移, Offset ProjectExename
OProjectTitle	DWORD;	// 5CH,偏移, Offset ProjectTitle
OHelpFile	DWORD;	// 60H,偏移, Offset HelpFile
OProjectName	DWORD;	// 64H,偏移, Offset ProjectName
End;		// Sizeof(VBHeader)=68H

其中各标注含义：

- ① 此处若为有效指针，即代表以 Sub_MainI（）启动，该指针指向 Sub_MainI（）执行代码起始处。
- ② 此处数值为此工程中包括的 frm，cls 文件数和，但不包括 bas 文件数。
- ③ 此处数值为工程引用的 ocx，activex dll 文件数量，比如 winsock.ocx。
- ④ 指针，指向对引入的 ocx，activex dll 的描述表。

3. TGUITable 结构定义

Type		
PGUITable	=^tGuiTable;	
tGuiTable	= packed record	
Signature	DWORD;	//00H,必须是 ‘50 00 00 00’
FormID	TGUID;	//04H,可能是以 GUID 方式命名的 form ID ^⑤
Flag1	array [1..52] of char;	//14H,意义不明
AGUIDescriptionTable	DWORD	//48H,指针指向以“FF CC ** **” ^⑥ 开始的 FormGUI 表
Flag2	Dword	//4CH,意义不明
End;		//Sizeof(aGUITable)=50H
		//有多少 form 结构，就有多少连续排列的 aGUITable

其中各标注含义：

- ⑤ 此处明显是一个 GUID 结构，可能代表此 Form。
- ⑥ *处可以根据 VBHeader 结构+44H 处得到的 Form 与 cls 合计数量值，减去 aGUITable 的数量，得到 cls 文件数量。

以 example2-N.exe 程序说明（程序见光盘：\chap08\example2\example2-N.exe），用

十六进制工具打开，如图 8.6 所示。

```
000012b0h: 00 00 00 00 9C 11 40 00 4C 00 00 00 56 42 35 21 ; .... @.L...VB5!  
000012c0h: 1C 23 2A 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .#*.....  
000012d0h: 7E 00 00 00 00 00 00 00 00 00 00 00 00 00 0A 00 ; ~.....  
000012e0h: 09 04 00 00 00 00 00 00 00 00 00 00 14 17 40 00 ; .....@.  
000012f0h: 10 F0 30 00 00 FF FF FF 08 00 00 00 01 00 00 00 ; . .yyy.....  
00001300h: 01 00 00 00 E9 00 00 00 6C 12 40 00 6C 12 40 00 ; .... .l.0.l.0.  
00001310h: 58 11 40 00 78 00 00 00 83 00 00 00 97 00 00 00 ; X.0.x... ..  
00001320h: A6 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....  
00001330h: 00 00 00 00 65 78 61 6D 70 6C 65 32 2D 4E 00 54 ; ....example2-N.T  
00001340h: 68 69 73 49 73 45 78 61 6D 70 6C 65 32 54 69 74 ; hisIsExample2Tit  
00001350h: 6C 65 00 54 68 69 73 49 73 48 65 6C 70 46 69 6C ; le.ThisIsHelpFil  
00001360h: 65 00 50 72 6F 6A 65 63 74 31 00 00 00 00 00 00 ; e.Project1.....
```

图 8.6 TGUITable 结构

将图 8.6 中的数据列表，结果如表 8-2 所示。

表 8-2 TGUITable 结构的数据列表

offset 000012BC	+ 00 H	'VB5!'
	+ 04 H	
	+ 06 H	语言链接库名
	+ 14 H	
	+ 22 H	0A 代表 msvbvm60.dll 版本 6.0.8964
	+ 24 H	语言标识
	+ 28 H	
	+ 2C H	00000000 表示非由 sub_main 启动
	+ 30 H	PTR 00404714, 指向 ProjectInfo 结构
	+ 34 H	
	+ 36 H	
	+ 38 H	
	+ 3C H	
	+ 40 H	
	+ 44 H	只有 1 个 form
	+ 46 H	引用的 ocx, dll 文件个数为 0
	+ 48 H	
	+ 49 H	
	+ 4A H	
	+ 4C H	Ptr 0040126C
	+ 50 H	若没有“引入的 ocx, dll 文件”，此指针无效
	+ 54 H	在指针 00401158+44 处开始描述 form 的属性
	+ 58 H	000012BC+78, ProjectExename ="example2-N"
	+ 5C H	000012BC+83, ProjectTitle="ThisIsExample2Title"
	+ 60 H	000012BC+97, HelpFile="ThisIsHelpFile"
	+ 64 H	000012BC+A6, ProjectName="project1"

4. TProjectInfo 结构定义

TProjectInfo 结构定义如下：

Type		
PProjectInfo	= ^tProjectInfo;	
tProjectInfo	= packed record;	// VBHead+30H
Signature	WORD;	// 00H,标志"F4"
NonFormFlag	WORD;	// 02H,0000 表示包含 bas 或 cls, 0001 表示仅有 Frm
AObjectTable	DWORD;	// 04H,指针指向 tObjectTable 结构
Reserved1	DWORD;	// 08H,总是"00 00 00 00"
AStartOfCode	DWORD;	// 0CH,未知作用指针
aEndOfCode	DWORD;	// 10H,未知作用指针
DataSize	DWORD;	// 14H,数据大小, 表示".data"区段大小
ADataSection	DWORD;	// 18H,指针, 指向".data"区段起始地址
AExcepHandler	DWORD;	// 1CH,MSVBVM60.__vbaExceptionHandler 指针
aNativeCode	DWORD;	// 20H,00000000 表示伪编译, 非 0 表示自然编译
VbpName	array [1..\$210] of char;	// 24H,字符串, {保存项目完整路径名称}
aExternalTable	DWORD;	// 234H,指针, 指向“引入信息表”结构
ExternalCount	DWORD;	// 238H,数量, 表示“引入信息表”总数
End;		// sizeof(tProjectInfo) = 23CH

5. TObjectTable 结构定义

TObjectTable 结构定义如下：

Type		
PobjectTable	= ^aObjectTable;	
AObjectTable	=packed record;	// tProjectInfo+04H
Null1	DWORD;	// 00H, '00 00 00 00' null
Address1	DWORD;	// 04H,
Address2	DWORD;	// 08H,
Const1	DWORD;	// 0CH, "FF FF FF FF"

```
Null2          DWORD;          // 10H, null
Address3       DWORD;          // 14H,
UUID           GUID;           // 18H, UUID
Const2         WORD;           // 28H,
ObjectCount1   WORD;           // 2AH, frm、bas、cls 等文件合计值
ObjectCount2   WORD;           // 2CH, 意义不明
ObjectCount3   WORD;           // 2EH, 意义不明
aObject        DWORD;          // 30H, 指针, 指向 tObject 表
Null3          DWORD;          // 34H, null
Null4          DWORD;          // 38H, null
Null5          DWORD;          // 3CH, null
aProjectName   DWORD;          // 40H, 指针指向区域为 project 组成名单
LangID1        DWORD;          // 44H,
LangID2        DWORD;          // 48H,
Null6          DWORD;          // 4CH, null
Const3         DWORD;          // 50H,
End;            // Sizeof(aObjectTable)=54H
```

6. TObject 结构定义

TObject 结构定义:

```
Type
Pobject=^aObject;
tObject=packed record;          // tObjectTable+30H
AobjectInfo          DWORD;     // 00H, 指针, 指向 tObjectInfo 表
Const1               DWORD;     // 04H, "FF FF FF FF"
Address1             DWORD;     // 08H, 指针, 指向未知数据
Null1               DWORD;     // 0CH, null
Address2             DWORD;     // 10H,
Null2               DWORD;     // 14H, null
```

AobjectName	DWORD;	// 18H, 指针指向对象 (form, bas, cls) 名
ProcCount	DWORD;	// 1CH, frm (bas, cls) 中 function()、sub()总数
AprocNamesArray	DWORD;	// 20H,
Const2	DWORD;	// 24H,
ObjectType	Byte;	// 28H, "83"=frm, "80"=bas, "01"=cls
Const3	array [1..3] of char;	// 29H, "80 01 00"
Null3	DWORD;	// 2CH, null
End;		// Sizeoff(tObject)=30H

7. TObjectInfo 结构定义

TObjectInfo 结构定义如下：

Type		
PobjectInfo=^aObjectInfo;		
TObjectInfo=packed record;		
Flag1	WORD;	// 00,标志, 通常为"1"
ObjectIndex	WORD;	// 02,序号, 表示该控件对象位于组件中的索引
AobjectTable	DWORD;	// 04,反身指针, 指向拥有该控件对象的 TObjectTable 结构
Reserved1	DWORD;	// 08,保留, 通常为"0"
TinyComHeaderPtr1	DWORD;	// 0C,微表指针, 指向类似 TObjectTable 结构; 但有时为 // "FFFFFFFF", 代表 bas
Flag2	DWORD;	// 10,标志, 通常为"FFFFFFFF"
Reserved2	DWORD;	// 14,保留, 通常为"0"
AObject	DWORD;	// 18,反身指针, 指向拥有该控件对象的 TObject 结构
VfDataPtr1	DWORD;	// 1C,地址, 指向程序运行时的数据地址
NumberOfProc	DWORD;	// 20,数量, 表示保存在 ProcTable 中的元素个数
AProcTable	DWORD;	// 24,地址, 指向 ProcTable 结构
ConsCount	WORD;	// 28,数量, 表示"常量"阵列的元素个数
Flag3	WORD;	// 2A,标志, 通常为"0"或者是"20"
Reserved3	DWORD;	// 2C,保留, 通常为"0"

UnkData1	DWORD;	// 30,未知数据, 比如为"0"以及其他
ConstPool	DWORD;	// 34,地址, 指向"常量"阵列
		//其后是 TOptionalObjectInfo 结构
End;		

8. TOptionalObjectInfo 结构定义

TOptionalObjectInfo 结构定义如下:

Type		
PoptionalObjectInfo	=^optionalObjectInfo;	
TOptionalObjectInfo	=packed record;	
Flag1	DWORD;	// 00,标志, 通常为"01 00 00 00", 如果组件类型为"Module", //那么此成员以后的定义无意义
UUIDPtr	DWORD;	// 04,UUID 指针, 指向 UUID
Reserved1	DWORD;	// 08,保留, 通常为"0"
UUIDPtrPtr1	DWORD;	// 0C,UUID 指针的指针, 指向 UUID 指针的指针
Flag2;	DWORD;	// 10,标志, 通常为"1"
ObjTablePtr0	DWORD;	// 14,地址, 通常与 ObjTablePtr 成员值相同
Reserved3	DWORD;	// 18,保留, 通常为"0"
UUIDPtrPtr2	DWORD;	// 1C,UUID 指针的指针, 指向 UUID 指针的指针
AControlCount	DWORD;	// 20,数量, 表示一个窗体中包含控件个数
AControl	DWORD;	// 24,地址, 指向 Control 结构
VCodeCount	WORD;	// 28,数量, 表示"事件代码阵列"元素总数
Flag4	WORD;	// 2C,标志, 通常为"1B7"
Flag5	WORD;	// 30,标志, 通常为"68"
Flag6	WORD;	// 34,标志, 通常为"6C"
VCodeArrayPtr	DWORD;	// 38,地址, 指向"事件代码阵列"
VfDataPtr2	DWORD;	// 3C,地址, 指向程序运行时的数据地址
Reserved2	DWORD;	// 40,保留, 通常为"0"

```
Flag6          DWORD;          // 44,标志, 比如为"6A3518"或者"5F8FD0"等
End;
```

9. TControl 结构定义

TControl 结构定义如下:

```
Type
Pcontrol=^TControl;
Tcontrol=packed record;

  Const1          WORD;          // 00
  EventCount       WORD;          // 02,控件的事件数
  Flag2           DWORD;          // 04
  AGUID           DWORD;          // 08,指向控件 GUID 指针
  Index           DWORD;          // 0C,控件在 Form 中的排列顺序
  Const1          DWORD;          // 0E,常数
  Null1           DWORD;          // 10,null
  Null2           DWORD;          // 14,null
  AeventTable     DWORD;          // 18,指向 EventTable 结构
  Flag3           Byte;          // 1C
  Const2          Byte;          // 1D
  Const3          Word;          // 1E
  Aname           Dword;          // 20,指向控件名称
  Index2          Word;          // 24,值同 index1
  Const1Copy      Word;          // 26
End;
```

10. 其他

利用 Control.aGuid, 可以得到 GUID 值, 并据此明确此控件的类型, 如表 8-3 所示。

表 8-3 控件类型

GUID 值	控 件 类 型
'33AD4ED2-6699-11CF-B70C-00AA0060D393'	'PictureBox'
'33AD4EDA-6699-11CF-B70C-00AA0060D393'	'Label'
'33AD4EE2-6699-11CF-B70C-00AA0060D393'	'TextBox'
'33AD4EEA-6699-11CF-B70C-00AA0060D393'	'Frame'
'33AD4EF2-6699-11CF-B70C-00AA0060D393'	'CommandButton'
'33AD4EFA-6699-11CF-B70C-00AA0060D393'	'CheckBox'
'33AD4F02-6699-11CF-B70C-00AA0060D393'	'OptionButton'
'33AD4F0A-6699-11CF-B70C-00AA0060D393'	'ComboBox'
'33AD4F12-6699-11CF-B70C-00AA0060D393'	'ListBox'
'33AD4F1A-6699-11CF-B70C-00AA0060D393'	'HScrollBar'
'33AD4F22-6699-11CF-B70C-00AA0060D393'	'VScrollBar'
'33AD4F2A-6699-11CF-B70C-00AA0060D393'	'Timer'
'33AD4F32-6699-11CF-B70C-00AA0060D393'	'Printer'
'33AD4F3A-6699-11CF-B70C-00AA0060D393'	'Form'
'33AD4F42-6699-11CF-B70C-00AA0060D393'	'Screen'
'33AD4F4A-6699-11CF-B70C-00AA0060D393'	'Clipboard'
'33AD4F52-6699-11CF-B70C-00AA0060D393'	'DriveListBox'
'33AD4F5A-6699-11CF-B70C-00AA0060D393'	'DirListBox'
'33AD4F62-6699-11CF-B70C-00AA0060D393'	'FileListBox'
'33AD4F6A-6699-11CF-B70C-00AA0060D393'	'Menu'
'33AD4F72-6699-11CF-B70C-00AA0060D393'	'MDIForm'
'33AD4F7A-6699-11CF-B70C-00AA0060D393'	'App'
'33AD4F82-6699-11CF-B70C-00AA0060D393'	'Shape'
'33AD4F8A-6699-11CF-B70C-00AA0060D393'	'Line'
'33AD4F92-6699-11CF-B70C-00AA0060D393'	'Image'
'33AD4FFA-6699-11CF-B70C-00AA0060D393'	'Data'
'33AD5002-6699-11CF-B70C-00AA0060D393'	'Ole'
'33AD5012-6699-11CF-B70C-00AA0060D393'	'UserControl'
'33AD501A-6699-11CF-B70C-00AA0060D393'	'PropertyPage'
'33AD5022-6699-11CF-B70C-00AA0060D393'	'UserDocument'
'164CBDD1-7321-11D1-A1E8-00A0C90F2731'	'VBControlExtender'
'FCFB3D21-A0FA-1068-A738-08002B3371B5'	'ModalClass'

还有一些其他结构的定义，由于存在相当多的争论，故不在此列出。读者可结合

wkt vb debugger 或访问 <http://www.vb-decompiler.com> 自行研究。在“VB 程序事件解读”一节中会继续讨论 EventTable 结构。

8.5 VB 程序事件解读

编程时程序员主要是利用 VB 提供的常用几个标准控件,如 CommandButton, Label 等。在编译生成的 VB6 程序中,每个控件的所有事件也组成一张虚函数表。

定位这张事件表的方法可以参考“VB 与 COM”一节中介绍的搜索 EVENT_SINK_Release 方法或根据 VB 程序框架结构来定义来定位 EventTable 结构。

TEventTable 结构定义:

```
Type
PEventTable=^EventTable;

tEventTable=packed record;

    Null1          DWORD;           // 00, 常数"00 00 00 00"

    aControl        DWORD;           // 04, 指向 Tcontrol 结构

    aObjectInfo     DWORD;           // 08, 指向 ObjectInfo 结构

    aQueryInterface DWORD;           // 0C, 指向 MSVBVM60.EVENT_SINK_QueryInterFace

    aAddRef          DWORD;           // 10, 指向 MSVBVM60.EVENT_SINK_AddRef

    aRelease         DWORD;           // 14, 指向 MSVBVM60.EVENT_SINK_Release

    aEventPointer    array [1..Control.EventCount] of Dword; // 18, 依次排列具体控件的事件指针, 若指针为 0 则无此
                                                                // 事件, 否则指针指向的地址为事件代码的起始

End;
```

事件表的起始总是如表 8-4 所示。

表 8-4 事件表的起始

+ 00H	DWORD	00000000
+ 04H	DWORD	ControlNamePointer
+ 08H	DWORD	OwnerFormPointer
+ 0cH	DWORD	MSVBVM60.EVENT_SINK_QueryInterFace Pointer

(续表)

+ 10H	DWORD	MSVBVM60.EVENT_SINK_AddRef Pointer
+ 14H	DWORD	MSVBVM60.EVENT_SINK_Release Pointer
+ 18H	DWORD	Event1Pointer
+ 1cH	DWORD	Event2Pointer
...

控件 `CommandButton` 的事件表如表 8-5 所示。

表 8-5 控件 `CommandButton` 的事件

+ 00H	DWORD	00000000
+ 04H	DWORD	ControlNamePointer
+ 08H	DWORD	OwnerFormPointer
+ 0cH	DWORD	MSVBVM60.EVENT_SINK_QueryInterFace Pointer
+ 10H	DWORD	MSVBVM60.EVENT_SINK_AddRef Pointer
+ 14H	DWORD	MSVBVM60.EVENT_SINK_Release Pointer
+ 18H	DWORD	CommandButton_Click
+ 1cH	DWORD	CommandButton_DragDrop
+ 20H	DWORD	CommandButton_DragOver
+ 24H	DWORD	CommandButton_GotFocus
+ 28H	DWORD	CommandButton_KeyDown
+ 2cH	DWORD	CommandButton_KeyPress
+ 30H	DWORD	CommandButton_KeyUp
+ 34H	DWORD	CommandButton_LostFocus
+ 38H	DWORD	CommandButton_MouseDown
+ 3cH	DWORD	CommandButton_MouseMove
+ 40H	DWORD	CommandButton_MouseUp
+ 44H	DWORD	CommandButton_OLEDragOver
+ 48H	DWORD	CommandButton_OLEDragDrop
+ 4cH	DWORD	CommandButton_OLEGiveFeedback

(续表)

+ 50H	DWORD	CommandButton_OLEStartDrag
+ 54H	DWORD	CommandButton_OLESetData
+ 58H	DWORD	CommandButton_OLECompleteDrag

下面演示如何更改程序结构来改变 main Form 显示的例子。如光盘：
\\chap08\\example4\\example4.exe, 其运行后 main form 是 form1, 在此将 form2 改为 main form。

(1) 先用十六进制工具打开 example4.exe, 定位到特征字符串“VB5!”, 如图 8.7 所示。

```
00001260h: 56 42 35 21 1C 23 2A 00 00 00 00 00 00 00 00 00 ; VB5!.*.....
00001270h: 00 00 00 00 7E 00 00 00 00 00 00 00 00 00 00 00 ; .....~.....
00001280h: 00 00 0A 00 09 04 00 00 00 00 00 00 00 00 00 00 ; .....
00001290h: A4 16 40 00 10 F0 30 00 00 FF FF FF 08 00 00 00 ; 0.. ..ÿÿÿ....
```

图 8.7 定位特征字符串“VB5!”

(2) 定位到 ProjectInfo 结构 (VBHead.30): Offset 000016A4h, 如图 8.8 所示。

```
000016a0h: 00 00 00 00 F4 01 00 00 E0 18 40 00 00 00 00 00 ; .... 0.....
```

图 8.8 定位到 ProjectInfo 结构

(3) 定位到 ObjectTable 结构 (ProjectInfo.04): offset 000018E0h, 如图 8.9 所示。

```
000018e0h: 00 00 00 00 40 30 40 00 04 1C 40 00 FF FF FF FF ; ...000...0.ÿÿÿÿ
000018f0h: 00 00 00 00 30 30 40 00 62 5C 59 4D F1 E5 D7 11 ; ...000.b\YM
00001900h: A7 E7 44 45 53 54 77 77 0A 00 02 00 04 00 02 00 ; xDESTw.....
00001910h: 34 19 40 00 00 00 00 00 00 00 00 00 00 00 00 00 ; 4.0.....
```

图 8.9 定位到 ObjectTable 结构

(4) 据 offset 0000190A 值“02”知道程序共有 2 个 Frm (或 cls, bas 文件), 位置始于偏移 00001934 处, 每块结构长度为 0x30, 共 2 块, 都属于 Object 结构。作为 main Form, 其 Object 结构是位于第一位的, 所以如果将后面的 Object 块前移至第一, 就能改变 main form 的定义, 如图 8.10 所示。Example4-patch.exe 是修改后的程序。

```
00001930 02 00 00 00 D0 14 40 00 FF FF FF FF 94 1A 40 00 .....@.....@.
00001940 00 00 00 00 00 00 00 00 00 00 00 00 00 A8 19 40 00 .....@.....@.
00001950 01 00 00 00 94 19 40 00 FF FF 00 00 83 80 01 00 .....@.....@.
00001960 00 00 00 00 94 13 40 00 FF FF FF FF E0 1A 40 00 .....@.....@.
00001970 00 00 00 00 00 00 00 00 00 00 00 00 00 00 19 40 00 .....@.....@.
00001980 00 00 00 00 D4 CC 60 00 FF FF 00 00 83 80 01 00 .....x.....
00001990 00 00 00 00 00 00 00 00 00 00 00 00 00 50 72 6F 6A .....Proj
```

图 8.10 对调 Object 块

8.6 VB 程序图形界面解读

前面说过，tVBHeader 结构偏移 4CH 处指针 aGUITable 指向 TGUITable 结构，TGUITable 结构偏移 48H 处指针 aGUIDescriptionTable 指向 form 等界面描述数据段。

结合 example2-N.exe 看，由 PE 结构，知此程序基地址为 00400000H：

aGUITable: 0040126CH =126CH

aGUIDescriptionTable: 0040119CH =119CH

顺着指针导引，来到位于以偏移 119C 开始的 form 界面描述表，如图 8.11 所示。

```
001190h: 00 85 40 00 58 85 40 00 00 00 00 00 FF CC 31 00 ; .X.XX....y .
0011a0h: 01 03 6B 47 17 0D 87 D7 11 A7 E3 44 45 53 54 77 ; ..kG..c DESTw
0011b0h: 77 04 6B 47 17 0D 87 D7 11 A7 E3 44 45 53 54 77 ; w.kG..c DESTw
0011c0h: 77 3A 4F AD 33 99 66 CF 11 B7 0C 00 AA 00 60 D3 ; w:O 程 .
0011d0h: 93 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
0011e0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
0011f0h: 00 00 00 00 00 76 00 00 00 3E 00 00 00 00 05 00 ; ....v...>....
001200h: 46 6F 72 6D 31 00 0D 01 05 00 46 6F 72 6D 31 00 ; Form1....Form1.
001210h: 19 01 00 42 00 23 FF FF FF FF 24 05 00 46 6F 72 ; ...B.#yyyy$.For
001220h: 6D 31 00 35 3C 00 00 00 59 01 00 00 48 12 00 00 ; m1.5<...Y...H...
001230h: 7B 0C 00 00 46 03 FF 01 31 00 00 00 01 08 00 43 ; {...F.y.1.....C
001240h: 6F 6D 6D 61 6E 64 31 00 04 01 0F 00 54 68 69 73 ; ommand1.....This
001250h: 20 49 73 20 43 61 70 74 69 6F 6E 00 04 D0 02 48 ; Is Caption.. H
001260h: 03 03 0C 0B 04 11 00 00 FF 02 04 00 50 00 00 00 ; .....y...P...
```

图 8.11 form 界面描述表

将图 8.11 数据以表的形式表示，结果如表 8-6 所示。

表 8-6 form 界面描述表

偏 移	注 释
119C + 00H	Frm 或 cls、UserControl 等结构起始标志，绝大部分是"FF CC 31 00"，但也可以是"FF CC 2C 00"或"FF CC 80 00"
119C + 04H	除去 form 控件不算，其具有的控件总数为 1
119C + 05H	GUID {17476B03-870D-11D7-A7E3-444553547777}
119C + 15H	GUID {17476B04-870D-11D7-A7E3-444553547777}
119C + 25H	GUID {33AD4F3A-6699-11CF-B70C-00AA0060D393},是 FORM_EVENT GUID
119C + 35H	NULL
119C + 59H	Frm 结构在 119C+59+00000076=126B 处结束
119C + 5DH	Form 属性定义在 119C+5D+0000003E=1237 处结束
119C + 61H	"00"表示是 frm 上第一个控件
119C + 62H	控件 name
119C + 6AH	"0D"意味着控件为 form
119C + 6BH	Form_caption="Form1"

(续表)

偏 移	注 释
119C + 74H	意义不明
119C + 79H	Form Icon=none
119C + 7EH	Form linktopic="form1"
119C + 87H	Form_size Left = 0000003CH Top = 00000159H Width = 00001248H Height = 00000C7BH
119C + 98H	"46 03" 意义不明
119C + 9AH	'FF 01'表示第一个控件描述结束（前面说过 form 属性描述结束于+9B 处）
119C + 9CH	"31 00 00 00"表明控件 CommandButton GUI 描述结束于 119C+0031=11CD 处
119C + A0H	"01"控件 ID 为 01
119C + A1H	控件 name
119C + ACH	"04"意味着控件为 CommanButton
119C + ADH	CommandButton_caption="This Is Caption"
119C + COH	Command_Size Left =02d0H Top =0348H Width =0c03H Height =040bH
119C + C9H	CommandButton_Tabindex=0000
119C + CCH	"FF 02": 第二个控件描述结束
119C + CEH	"04 00"意义不明
	至此整个 frm 图形界面描述结束

表 8-7 列出了 VB6 标准控件的 ID 值。

表 8-7 VB6 标准控件的 ID 值

ID	控 件	ID	控 件
00	PictureBox	16	Shape
01	Label	17	Line
02	TextBox	18	Image
03	Frame	19	--
04	CommandButton	1A	--
05	CheckBox	1B	--

(续表)

(续表)

ID	控 件		ID	控 件
06	OptionBox		1C	--
07	ComboBox		1D	--
08	ListBox		1E	--
09	Scrollbar		1F	--
0A	VscrollBar		20	--
0B	Timer		21	--
0C	Printer		22	--
0D	Form		23	--
0E	Screen		24	--
0F	Clipboard		25	Data
10	DriveListBox		26	OLE
11	DirListBox		27	--
12	FileListBox		28	UserControl
13	Menu		29	PropertyPage
14	MDIForm		2A	UserDocument
15	App			

表 8-8 列出了 form 控件的各个属性 ID 值。

表 8-8 form 控件的各个属性 ID 值

ID	控 件		ID	控 件
00	Name		26	MaxButton
01	Caption		27	MinButton
02	HWnd		28	ControlBox
03	BackColor		29	Image
04	ForeColor		2A	HasDC
05	Left		2B	--
06	Top		2C	--

(续表)

ID	控 件		ID	控 件
07	Width		2D	--
08	Height		2E	Visible
09	Enabled		2F	Tag
0A	WindowState		30	MDIChild
0B	MousePointer		31	KeyPreview
0C	FontName		32	ClipControls
0D	FontSize		33	HelpContextID
0E	FontBold		34	ActiveControl
0F	FontItalic		35	ClientLeft
10	FontStrikethru		36	ClientTop
11	FontUnderline		37	ClientWidth
12	HDC		38	ClientHeight
13	CurrentX		39	Count
14	CurrentY		3A	Controls
15	ScaleLeft		3B	MouseIcon
16	ScaleTop		3C	--
17	ScaleWidth		3D	LockControls
18	ScaleHeight		3E	NegotiateMenus
19	ScaleMode		3F	--
1A	FontTransparent		40	Font
1B	DrawStyle		41	Appearance
1C	DrawWidth		42	WhatsThisButton
1D	FillStyle		43	WhatsThisHelp
1E	FillColor		44	ShowInTaskbar
1F	DrawMode		45	RightToLeft
20	AutoRedraw		46	StartPosition
21	Picture		47	OLEDropMode

(续表)

ID	控 件		ID	控 件
22	BorderStyle		48	Palette
23	Icon		49	PaletteMode
24	LinkTopic		4A	Moveable
25	LinkMode			

表 8-9 列出了 CommandButton 的属性 ID 对照表。

表 8-9 CommandButton 的属性 ID 对照表

ID	控 件		ID	控 件
00	Name		16	DragMode
01	Caption		17	DragIcon
02	Index		18	TabStop
03	BackColor		19	Tag
04	Left		1A	HWnd
05	Top		1B	HelpContextID
06	Width		1C	MouseIcon
07	Height		1D	Font
08	Enabled		1E	WhatsThisHelpID
09	Visible		1F	Appearance
0A	MousePointer		20	Container
0B	FontName		21	RightToLeft
0C	FontSize		22	Picture
0D	FontBold		23	DisabledPicture
0E	FontItalic		24	DownPicture
0F	FontStrikethru		25	ToolTipText
10	FontUnderline		26	OLEDropMode
11	TabIndex		27	MaskColor
12	Value		28	UseMaskClor

(续表)

ID	控 件		ID	控 件
13	Default		29	Style
14	Cancel		2A	CausesValidation
15	Parent			

8.7 VB 执行代码研究

自然编译是 VB 源代码生成汇编代码并链接的过程。因为是生成汇编代码，所以对其解读完全遵照一般的反汇编常规。

伪编译是生成伪代码并链接的过程。运行时依赖解释引擎将伪代码翻译为汇编代码再执行。伪代码运行完全依赖堆栈。

8.7.1 VB 函数的解读

VB 实际上的数据计算和比较是在 msbvm60.dll 及 oleaut32.dll 中完成的。msbvm60 的全称是：MicroSoft Visial Basic Virtual Machine 6.0。

如果要深入研究 VB，不妨用 W32dasm 反汇编 msbvm60.dll 及 oleaut32.dll，理解其函数的意义。通过研究 OLEAUT.H, OAIDL.h 文件可以了解 VB 函数中的各类变量，并进而根据函数名称猜出大部分函数的用途，见表 8-10。

表 8-10 部分 VB 函数变量

名 称	注 释
Empty	Nothing
Null	SQL style Null
I1	Signed char
I2	2 byte signed integer
I4	4 byte signed integer
I8	64-bit integer
UI1	unsigned char

(续表)

名 称	注 释
UI2	unsigned short integer
UI4	unsigned short integer
UI8	unsigned 64-bit integer
R4	4 byte real
R8	4 byte real
Int	signed machine integer
UInt	unsigned machine integer
Error	Scode
Date	Date
Bstr	OLE Automation string
Bool	Boolean, True=-1, False=0
Str	string
Dispatch	Idispatch
safearray	ARRAY
var	Variant
cy	currency
decimal	16 byte fixed point
record	user defined type
void	C style void
HRESULT	Standard return type
ptr	pointer type
carray	C style array
lpstr	null terminated string
lpwstr	wide null terminated string
fp	floating point
userdefined	user defined type

(续表)

名 称	注 释
filetime	File time
blob	Length prefixed bytes
stream	Name of the stream follows
storage	Name of the storage follows
cf	Clipboard format
clsid	Class ID
vector	simple counted array
byref	void* for local use

msvbvm60.dll 中包含的函数名总是以__vba 或 rtc 开头,而 oleaut32.dll 函数名以 var 开头。对于函数的作用,一般可以按照函数名从右往左理解。例如:

varBstrFromI2 整数到字符串
__vbaI2Str 字符串转换为整数
Pcode 指令也可以同样理解:
CR8I4 Convert I4 to R8
AddI2 add I2

8.7.2 VB 函数调用约定

Windows 平台上,对 VB 函数的调用还是遵循 stdcall 原则,即函数入口参数按从右到左的顺序入栈,并由被调用的函数在返回前清理传送参数的内存栈。

比如函数 MsgBox,对应 msvbvm60.dll 中 rtcmsgbox 函数。其语法为:

MsgBox(prompt[, buttons] [, title] [, helpfile, context])

汇编代码形式为:

push context
push helpfile
push title
push buttons

```
push    prompt
call    rtcmsgbox
```

8.7.3 执行代码中对控件属性的操作

例如下面一段代码：

```
Private Sub Command1_Click()
Command1.Caption = "ad"
End Sub
```

反汇编如下：

```
:00401A2B  68A4164000    push 004016A4          // string "ad"
:00401A30  56            push esi
:00401A31  8B0E          mov ecx, dword ptr [esi]
:00401A33  FF5154        call [ecx+54]
:00401A36  3BC7          cmp eax, edi
:00401A38  DBE2          Fclex
:00401A3A  7D0F          jge 00401A4B
:00401A3C  6A54          push 00000054
:00401A3E  68AC164000    push 004016AC
:00401A43  56            push esi
:00401A44  50            push eax
:00401A45  FF1510104000  Call __vbaHresultCheckObj
```

以上是一段典型的控件属性操作代码，其中：

```
:00401A31  8B0E  mov ecx, dword ptr [esi]
```

ECX 表示指向 Command1 虚函数表的指针。

```
:00401A33  FF5154  call [ecx+54]
```

执行虚函数表中位于 54H 处指针指向的操作，也就是 put_CommandButton_Caption。其后总会有一个对象检测 Call __vbaHresultCheckObj。每个控件都有一个这样的虚函

数表。

CommandButton 控件属性操作表如表 8-11 所示。

表 8-11 CommandButton 控件属性操作表

虚函数指针名称	位 置
QueryInterface	call DWORD PTR [eax+0000]
AddRef	call DWORD PTR [eax+0004]
Release	call DWORD PTR [eax+0008]
GetTypeInfoCount	call DWORD PTR [eax+000C]
GetTypeInfo	call DWORD PTR [eax+0010]
GetIDsOfNames	call DWORD PTR [eax+0014]
Invoke	call DWORD PTR [eax+0018]
HctlDemandLoad	call DWORD PTR [eax+001C]
ChkProp	call DWORD PTR [eax+0020]
SetPropA	call DWORD PTR [eax+0024]
GetPropA	call DWORD PTR [eax+0028]
GetPropHsz	call DWORD PTR [eax+002C]
LoadProp	call DWORD PTR [eax+0030]
SaveProp	call DWORD PTR [eax+0034]
GetPalette	call DWORD PTR [eax+0038]
Reset	call DWORD PTR [eax+003C]
Get_Default() As Boolean	call DWORD PTR [eax+0040]
Let_Default(RHS As Boolean)	call DWORD PTR [eax+0044]
Get Name() As String	call DWORD PTR [eax+0048]
Get Caption() As String	call DWORD PTR [eax+0050]
Let Caption(RHS As String)	call DWORD PTR [eax+0054]
Get Index() As Integer	call DWORD PTR [eax+0058]
	call DWORD PTR [eax+005C]

(续表)

虚函数指针名称	位 置
Get BackColor() As Long	call DWORD PTR [eax+0060]
Let BackColor(RHS As Long)	call DWORD PTR [eax+0064]
Get Left() As Single	call DWORD PTR [eax+0068]
Let Left(RHS As Single)	call DWORD PTR [eax+006C]
Get Top() As Single	call DWORD PTR [eax+0070]
Let Top(RHS As Single)	call DWORD PTR [eax+0074]
Get Width() As Single	call DWORD PTR [eax+0078]
Let Width(RHS As Single)	call DWORD PTR [eax+007C]
Get Height() As Single	call DWORD PTR [eax+0080]
Let Height(RHS As Single)	call DWORD PTR [eax+0084]
Get Enabled() As Boolean	call DWORD PTR [eax+0088]
Let Enabled(RHS As Boolean)	call DWORD PTR [eax+008C]
Get Visible() As Boolean	call DWORD PTR [eax+0090]
Let Visible(RHS As Boolean)	call DWORD PTR [eax+0094]
Get MousePointer() As Integer	call DWORD PTR [eax+0098]
Let MousePointer(RHS As Integer)	call DWORD PTR [eax+009C]
Get FontName() As String	call DWORD PTR [eax+00A0]
Let FontName(RHS As String)	call DWORD PTR [eax+00A4]
Get FontSize() As Single	call DWORD PTR [eax+00A8]
Let FontSize(RHS As Single)	call DWORD PTR [eax+00AC]
Get FontBold() As Boolean	call DWORD PTR [eax+00B0]
Let FontBold(RHS As Boolean)	call DWORD PTR [eax+00B4]
Get FontItalic() As Boolean	call DWORD PTR [eax+00B8]
Let FontItalic(RHS As Boolean)	call DWORD PTR [eax+00BC]
Get FontStrikethru() As Boolean	call DWORD PTR [eax+00C0]
Let FontStrikethru(RHS As Boolean)	call DWORD PTR [eax+00C4]
Get FontUnderline() As Boolean	call DWORD PTR [eax+00C8]

(续表)


虚函数指针名称	位 置
Let FontUnderline(RHS As Boolean)	Call DWORD PTR [eax+00C8]
Get TabIndex() As Integer	call DWORD PTR [eax+00D0]
Let TabIndex(RHS As Integer)	call DWORD PTR [eax+00D4]
Get Value() As Boolean	call DWORD PTR [eax+00D8]
Let Value(RHS As Boolean)	call DWORD PTR [eax+00DC]
Get Default() As Boolean	call DWORD PTR [eax+00E0]
Let Default(RHS As Boolean)	call DWORD PTR [eax+00E4]
Get Cancel() As Boolean	call DWORD PTR [eax+00E8]
Let Cancel(RHS As Boolean)	call DWORD PTR [eax+00EC]
Get Parent() As Form	call DWORD PTR [eax+00F0]
Get DragMode() As Integer	call DWORD PTR [eax+00F8]
Let DragMode(RHS As Integer)	call DWORD PTR [eax+00FC]
Get DragIcon() As Picture	call DWORD PTR [eax+0100]
Let DragIcon(RHS As Picture)	call DWORD PTR [eax+0104]
Get TabStop() As Boolean	call DWORD PTR [eax+0108]
Let TabStop(RHS As Boolean)	call DWORD PTR [eax+010C]
Get Tag() As String	call DWORD PTR [eax+0110]
Let Tag(RHS As String)	call DWORD PTR [eax+0114]
Get hWnd() As Long	call DWORD PTR [eax+0118]
--	call DWORD PTR [eax+011C]
Get HelpContextID() As Long	call DWORD PTR [eax+0120]
Let HelpContextID(RHS As Long)	call DWORD PTR [eax+0124]
Get MouseIcon() As Picture	call DWORD PTR [eax+0128]
Let MouseIcon(RHS As Picture)	call DWORD PTR [eax+012C]
Get Font() As Font	call DWORD PTR [eax+0130]
Set Font() As Font	call DWORD PTR [eax+0134]
Get WhatsThisHelpID() As Long	call DWORD PTR [eax+0138]

(续表)

虚函数指针名称	位 置
Let WhatsThisHelpID(RHS As Long)	call DWORD PTR [eax+013C]
Get Appearance() As Integer	call DWORD PTR [eax+0140]
Let Appearance(RHS As Integer)	call DWORD PTR [eax+0144]
Get Container() As Object	call DWORD PTR [eax+0148]
Set Container() As Object	call DWORD PTR [eax+014C]
Get RightToLeft() As Boolean	call DWORD PTR [eax+0150]
Let RightToLeft(RHS As Boolean)	call DWORD PTR [eax+0154]
Get Picture() As Picture	call DWORD PTR [eax+0158]
Let Picture(RHS As Picture)	call DWORD PTR [eax+015C]
Get DisabledPicture() As Picture	call DWORD PTR [eax+0160]
Let DisabledPicture(RHS As Picture)	call DWORD PTR [eax+0164]
Get DownPicture() As Picture	call DWORD PTR [eax+0168]
Let DownPicture(RHS As Picture)	call DWORD PTR [eax+016C]
Get ToolTipText() As String	call DWORD PTR [eax+0170]
Let ToolTipText(RHS As String)	call DWORD PTR [eax+0174]
Get OLEDropMode() As Integer	call DWORD PTR [eax+0178]
Let OLEDropMode(RHS As Integer)	call DWORD PTR [eax+017C]
Get MaskColor() As Long	call DWORD PTR [eax+0180]
Let MaskColor(RHS As Long)	call DWORD PTR [eax+0184]
Get UseMaskColor() As Boolean	call DWORD PTR [eax+0188]
Let UseMaskColor(RHS As Boolean)	call DWORD PTR [eax+018C]
Get Style() As Integer	call DWORD PTR [eax+0190]
--	call DWORD PTR [eax+0194]
--	call DWORD PTR [eax+0198]
--	call DWORD PTR [eax+019C]
--	call DWORD PTR [eax+01A0]
--	call DWORD PTR [eax+01A4]

(续表)

虚函数指针名称	位 置
--	call DWORD PTR [eax+01A8]
--	call DWORD PTR [eax+01AC]
--	call DWORD PTR [eax+01B0]
--	call DWORD PTR [eax+01B4]
--	call DWORD PTR [eax+01B8]
--	call DWORD PTR [eax+01BC]
Get CausesValidation() As Boolean	call DWORD PTR [eax+01C0]
Let CausesValidation(RHS As Boolean)	call DWORD PTR [eax+01C4]

 注意：虚函数表中有部分指针名称缺失，原因不明。此表虽用[`eax+****`]表示，但实际中 `eax` 处可以是 `ebx`, `ecx`, `edx`。

8.8 P-code 代码

P-code 不是微软发明的一个新生词汇，它仅仅是边执行边解释的代码。P-code 可以理解为 CPU 不能直接解释的代码，需要先被翻译成可执行的机器码。与编译后的 Java 类似，即用虚拟机来执行 Java 编写的应用程序，虚拟机是指将 Java 代码翻译成 CPU 能理解的机器代码的解释器。

使用 P-code 的优势很明显。如果定义一系列的指令而不公布规范，代码将很难被理解。另一个好处是可以减小代码的体积，通过将一段操作码定义为一个字符，可以让这一指令执行一系列操作，而汇编代码则需要多得多的指令来完成相同的操作。微软的 Visual Basic P-code 正是这样一个将 P-code 翻译成处理器能理解的机器码的虚拟机。虚拟机位于程序运行时装载的 DLL（动态链接库）中。正如大多数人推测的那样，这些动态链接库的名称是：MSVBVM50.DLL, MSVBVM60.DLL。文件名是由 Microsoft Visual Basic Virtual Machine（MSVBVM）的首字母缩写加上版本号构成。这两个版本之间的区别很小：版本 6 中使用了一些新指令，而且指令的名称比版本 5 更直观。换句话说，版本 6 中改变了指令名称但并没有改变实质。

虚拟机不仅解释 VB 中的 P-code 指令，也被自然编译的可执行程序使用。这是因

为动态链接库（DLL）中包括了所有 VB 应用程序需使用的应用编程接口（API）。例如 `rtcMsgBox` 和 Windows 标准应用编程接口 `MessageBox` 是等同的。

用 SoftICE 跟踪 P-code 是很痛苦的，因为结果只是跟踪虚拟机的代码。确切地说，SoftICE 只能理解 CPU 的机器码，而不能理解 P-code。实际上，如果试图跟踪 P-code，所见到的结果将是 P-code 指令翻译后的机器码。

8.8.1 理解 P-code 代码指令

同汇编指令一样，P-code 代码指令结构也分成操作码和地址码两部分。
比如汇编指令：

```
00402220 EB10 jmp 00402230
```

EB 就是操作码，而 10 为地址码。

P-code 中基指令一共是 $256 \times 6 - 5 = 1531$ 条。包括单字节指令，即 00~FA 共 251 条；双字节指令 1280 条，分别由 FB, FC, FD, FE, FF 后面跟 00~FF 构成。

虽然 P-code 指令有如此之多，但有相当多的指令并无意义，可能是为今后新添加的函数预留的空位。这些无意义的指令多见于双字节基指令中。有统计表明，P-code 程序中单字节基指令要占到所有指令的 70% 以上。

由于这不是一篇讨论具体软件破解的文章，所以笔者觉得对某个具体 P-code 程序的代码进行分析并无任何意义。在此请读者自行寻找此类教程学习。

需要提醒的是，P-code 代码完全基于堆栈进行。所有数据的处理都是“入栈→`msvbvm60.dll`”或“`oleaut32.dll` 函数处理→再出栈”的过程；自然编译与伪编译实际在 `msvbvm60.dll` 中运行并无明显差别；如同汇编代码以寄存器 EIP 为步进指针般，P-code 以 ESI 寄存器作为步进指针。

表 8-12 列出部分 P-code 指令及注释（由于微软并不公开 P-code 指令的意义，所以此处的注释是根据一些资料和指令实际运作情况加以描述的）。

表 8-12 部分 P-code 指令及注释

基 指 令	助 记 符	注 释
1C	BranchF	False 则跳，常与 Eql2 等合用，就是不相等则跳的意思
23	FstStrNoPop	将一宽字符字符串指针压入堆栈并保存

(续表)

基 指 令	助 记 符	注 释
2A	ConcatStr	String1+string2
32	FfreeStr	释放字符串
44	CvarI2	I2 转换为 var
64	NextI2	相当于 VB 函数 For...Next 中的 for 指令
AF	SubR4	实数减法
FB3D	NeStr	字符串与字符串比较，如果不等则...
FB51	Gel2	>=I2
FBEC	FnLenStr	相当于 len(string)

8.8.2 P-code 程序调用约定

P-code 也遵循 stdcall 原则，MsgBox ("hello", vbOKOnly, "title", 0, 100) 的 P-code 代码反编译结果是：

4018A0: 28	LitVarI2:	(local_0104) 0x64 (100)	// push context
4018A5: 28	LitVarI2:	(local_00E4) 0x0 (0)	// push helpfile
4018AA: 3a	LitVarStr:	(local_00B4) title	// push title
4018AF: 4e	FstVarCopyObj	local_00C4	
4018B2: 04	FldRfVar	local_00C4	
4018B5: f5	LitI4:	0x0 0 (....)	// push button style
4018BA: 3a	LitVarStr:	(local_0094) hello	// push prompt
4018BF: 4e	FstVarCopyObj	local_00A4	
4018C2: 04	FldRfVar	local_00A4	
4018C5: 0a	ImpAdCallFPR4:	Rtcmsgbox	// call msgbox
4018CA: 36	FfreeVar		

8.8.3 调试时中断 P-code 程序的几种方法

自然编译的 VB 程序由汇编语言构成，调试时设断与一般程序无区别。
P-code 程序设断稍有不同。如果拥有源代码，则只要在需要设断的源代码语句前

调用 Kernel32.dll 函数 DebugBreak，这样编译后运行程序在 DebugBreak 处发生 INT3 中断，并可以被调试器拦截，由此再一步步跟踪下去就在 P-code 代码中了。

调试器 Ollydbg 载入 VB 程序后，在程序入口点向上翻页，见到“JMP DWORD PTR DS:[&MSVBVM60.MethCallEngine>]”语句，在其上用 F2 设断。

用 EXDEC 反编译 VB 5/6 pcode 程序，选择设断的代码地址，以“BPM 中断地址”方式拦截。或者用 WKT VB Pcode Debugger 调试器跟踪。

8.8.4 WKT VB Debugger 实现原理

WKT VB Debugger 是由西班牙破解小组 WKT 成员编写的一个软件，可以说代表了目前研究 VB5，VB6 的 P-code 反编译工作的最高成果。使用该工具动态跟踪后显示的反编译代码能让一位受过训练的破解者“还原”出相当部分的 VB 源代码。

Wkt VB Debugger 的作者之一 Mr.Silver 曾在 Fravia 逆向工程终极论坛发表一篇 P-code 调试器编写的文章《Brief Introduction to P-code》，文章主要讲述了如下几方面的内容。

1. 调试程序读取虚拟机并获取控制权

实现调试程序必须解决的重要问题之一是找出 P-code 转换的时间和方式。一旦解决这一问题，注入的代码将获得程序流的控制权并将数据发送到调试程序。调试程序处理操作码并将控制权返还给虚拟机。构思如下：

要反汇编/解释一段代码，需要一定的条件。

(1) 获取包含 P-code 的内存缓冲 (buffer) 指针；

(2) 从内存缓冲中读取操作码，再将程序流程重新定向到原先的 P-code 解释例程中。

这一任务可以通过两种方式完成：一系列条件语句（每条操作码一条语句）或使用跳转表。第一种方法不理想，因为在 P-code 中大量使用不同的操作码将需要一个巨大的条件控制结构（速度奇慢无比）。故猜测代码的解释过程是通过跳转表来进行的。

正如编写反汇编程序一样，现在必须完成一些事情：定位 P-code 操作码所在内存缓冲的基地址，以及跳转表的基址。

用 VB 编译了如下的一个小程序：

```
Private Sub Form_Load()
```

```
MsgBox "Hello this is P-code!!!", vbInformation, "Example"

End Sub
```

在 SoftICE symbol loader 中装载(MSVBVM60.DLL)虚拟机, 并且用 bpx rtcMsgBox 设断。SoftICE 中断后按 F12 键返回调用 rtcMsgBox 的代码, 显示如下:

```
call eax           // 调用 rtcMsgBox
cmp edi,esp        // 我们在这里
jnz 66105595       // 检查堆栈指针
xor eax,eax        // 准备 eax 从缓冲中加载下一条操作码
mov al,byte ptr [esi] // 加载将要执行的 P-code 操作码, 这里是 36h
inc esi           // 增加 esi 中的指针
jmp [eax*4+660FDA58] // 跳转到解释 36h 操作码的例程
```

正如前面所推测的那样, 解释程序从缓冲 (ESI) 中读取操作码到 AL, 通过转换操作码为跳转表地址, 跳转到相应的解释例程, 本例中为 36h (这种方式高级而且灵活, 避免了成千上万的核对)。继续跟踪则可以发现用 ESI 指向对缓冲的访问的方法被不断重复, 而且, 在虚拟机内部跟踪时, ESI 是一直指向包含操作码的缓冲。所以可以通过 SoftICE 命令来找出将要执行的语句, 命令为:

```
d *esi
```

这看上去正是所需要的: ESI 包含了一个指向内存缓冲的指针, AL 保存该缓冲的下一个字节。最有意思的是无条件跳转 JMP[4*EAX+ADRESS]。可以看到: 它使用从缓冲中读取的字节作为相对位移来运作跳转表, 这样可以很容易地推算出该表的最大体积, 从 AL (256) 的最大值及 4 字节的指针的使用可以推算出长度为 $256 \times 4 = 1024$ 字节。

毫无疑问, 这正是所要找的表。根据微软的文件, P-code 有标准操作码 256 条。文件中也提到了其他的扩展操作码。这 256 个值中有一些作为前缀被保留了下来。当解释程序发现这些前缀的时候, 就会执行一条相应的扩展指令, 给出另外一套 256 条新代码, 所以使用前缀可以变换出无穷多的指令。同样每个前缀可以拥有自己的跳转表。后面将会谈到 VB P-code 中现有前缀的数目, 以及如何定位它们的跳转表。如果反汇编虚拟机并且搜索了所有跳转表中的例程, 会发现表的条目都是虚拟机内的地址, 它们都包含在 DLL 文件中的 .ENGINE 段。表指向的例程中的大多数都是同样的结构, 它们都会读取缓冲区中 ESI 所指的数据, 并执行相应的指令, 随后读取下一条操作码

并跳转到相应的解码例程。在这里可以找到想要找的东西：每条操作码到跳转表的地址。

下一步是注入一个地址代码进行跳转，以替换原跳转表，每条操作码的跳转地址是相同的。该地址指向的代码位于调试程序的 DLL 库中，每条操作码在执行前都会经由该地址转入我们的代码中。然后，我们的代码先保存当前各寄存器数据，运行我们自己的功能代码后再将原寄存器数据抛出，回到虚拟机的代码中去，等待下一指令……重复以上操作。

下面为调试程序例程的始末：

```
__declspec( naked ) void DebuggerProc()
{

_asm {

    mov VBDebugger.OldStack_ESP,esp    //保存当前 VB 堆栈数据
    mov VBDebugger.OldStack_EBP,ebp    //基指针和堆栈指针
    pushad                             //保存通用寄存器数据
    pushfd                             //保存标志寄存器数据

    push ebp                           //开始一个标准结构
    mov ebp, esp
    sub esp, __LOCAL_SIZE              //如果存在局部变量，将从堆栈中减去

    // 调试程序控制代码的其他部分
    // ...
    // ...

    // 更改跳转地址，因为无法静态修改
    //所以我们用了 self-modifying code 技术，在运行时自动更改跳转地址:P

_asm {

    mov ecx,offset JumpOffset
```

```

        add eax,3

        mov ebx,[VBDebugger.RedirTableAddr]

        mov [eax],ebx
    }

    mov esp,ebp                //还原初始堆栈框架
    pop ebp
    popfd
    popad                      //抛出寄存器各值

    // 最后
    // 如果操作码被内存编辑器更改，则在 AL 中也进行更改
    // 这样，使改变的操作码能被执行

    _asm {
        cmp [VBDebugger.OPCODE_CHANGE],0

        je NoChange

        mov al,VBDebugger.Opcode
NoChange:
        mov [VBDebugger.VMAddress],0
    }

    JmpOffset:

    // 返还控制权给虚拟机。注意此代码在运行是自动更改的
    jmp [eax*4+VBDebugger.RedirTableAddr]

}

```

正如读者所看到的，使用直接转向，可以按自己的需要来建立例程（并控制代码的运行）。实际上在 Windows 系统中，这种方法常常用于建立驱动程序（VXD）中，

以建立一个中间层，插入源代码与 P-code 之间。例程所做的无非是接过控制权后，再将控制权返还给源代码，却使我们能操控虚拟机执行操作码过程中的一切。

P-code 资料的缺乏是因为微软对技术的保密，只有在签定了一个叫做 NDA（保密协议）的协议之后才可以接触上述资料。而且一旦泄露将会被追究法律责任。因此，P-code 的资料极少，得到的也语焉不详。

再就是 Exdec 工具，Josephco 开发的 VB5/6 P-code 反汇编程序。Exdec 可以反汇编 P-code 程序，但操作码的显示不完全（仅显示首字节）。后面将谈到这一原因。

有了跳转表地址作为基础，可以用一个代码补丁作为虚拟机 DLL 的新组成部分，补丁程序将获取初始化数据并加载调试程序，有了这一补丁就可以将控制权重新定向到虚拟机 DLL 的 OEP。

这种方式有一个很大的缺陷是需要更改 VM.dll 文件。这时可以通过创建引导程序 Loader（以挂起模式启动 VB 应用程序的小应用程序），用 GetThreadContext 获取入口点，并加载调试程序 DLL 的代码补丁。一旦执行该补丁，它将通过使用 API 函数 SetEvent 及 WaitForSingleObject 来通知主初始化进程。完成后，补丁还原初始代码并使用 SetThreadContext 返回原始 OEP，执行过程继续进行，就好像什么事情也没有发生过一样。

调试程序得作为一个独立线程运行，检查内存地址中是否包括“VB5!”签名（用于指示加载程序）。因为原始的程序代码和部分 VB 数据已经被替换，而调试程序需要这些数据。

另一个问题就是操作码表的地址会因为虚拟机版本的不同而不同，所以必须检查所有版本的虚拟机并采取相应的措施。这实在很麻烦，一旦有新版本的虚拟机出现就得更改引导程序。由于操作码表存在于虚拟机的 ENGINE 部分，该表的一个特点就是表内包含的所有地址都以相同结构指向代码具体执行段，所以可以设计一种算法，在 ENGINE 段中，来定位那张由连续排列的代表 256 个标准前缀操作码的 DWORD 值所在的表。

2. 分析操作码

所有操作码都保存在 EXDEC 的 DLL 文件中。可以使用十六进制编辑器帮助定位，你会发现用作前缀的操作码（FF FE FD FC FB (Lead0, Lead1, Lead2, Lead3 和 Lead4) 系列中的后面五个），每一个前缀生成一个新的操作码表。总数为：

(5 个前缀+1 标准系列) × 256 操作码 = 1536 操作码

可以看出，有很多操作码没有被使用，并有一些是多余的。例如，Lead4 前缀没有使用全部的操作码表，在 msvbvm60 中只到操作码 46h（可以通过反汇编虚拟机来验证）。正如前面所说的，定位一些操作码后，认识到虚拟机调试程序文件包含了所有的符号信息，例程名称，P-code 操作码的地址与名称。用 SoftICE 将 DBG 信息转储到文本文件中就可以获取全部信息。表 8-13 列举了一部分，感兴趣的话可以看看。

表 8-13 部分 P-code 指令符号信息

RVA	大 小	Symbol 名称
0F103D8Bh	34	CCyR4
0F103DADh	19	CCyVar
0F103DC0h	9	CBoolCy
0F103DC9h	0	CBoolR8
0F103DC9h	38	CBoolR4
0F103DEFh	32	CStrVar
0F103E0Fh	18	CStrBool
0F103E21h	34	CStrR8

符号名称中显示了一些 P-code 操作码助记符名；在 RVA 部分则为虚拟机内地址。不幸的是，这些地址会因虚拟机版本的不同而不同，但调试程序可以通过搜索来定位它。

不同的版本，操作码助记符名称也不同，但它们所执行的操作是一样的。通过这些信息就可以完成对代码的简单反编译。开始仅显示执行的指令，因为还不知道每一指令的长度。这是最难的部分，必须分析所有的 P-code 指令，在每一例程的末尾检查操作码的长度。分析 1000 多条指令需要很长时间，虽然大部分的指令执行代码并不长。这些指令长度都不是固定的，一些指令有运行参数，这就使得它们的长度会发生变化，这只占很少的一部分。

WKT VB Debugger 作者们当时分配任务，每人处理一套不同的指令。完成后获得所有的指令长度后就可以进行反编译了。因为一些无法避免的错误，后来纠正了一些操作码的长度定义。但即使是现在，仍可能存在一些错误，因为有些指令不是在所有应用程序中都会使用，也就不可能测试所有的指令。最新的 WKT VB Debugger 1.3 版

中没有再发现错误的操作码。

3. 增添调试程序基本功能

这一步的编码和研究更为复杂。添加输出表不难，通过加载程序分析 PE 头可以获取跳转表地址，据此调试器程序获得控制权。获得地址后，建立一个列表，在列表里使用标志建立断点状态（ACTIVE/INACTIVE/NONE）。用这种方式可以把断点放在任何虚拟机 API 函数上。在 P-code 操作码中可以使用类似的方法设置断点。与传统调试程序不同的是：只要有指令，调试器程序就可以设断，因为调试程序拥有对虚拟机的控制权。

后来的工作是在实际代码中加上了断点，例如给操作码地址设置断点。断点保存在一个动态链接的列表里，这样做有如下优点：断点的数量没有限制而且内存的使用随着已建立断点的数量进行调整。

断点的基本功能不是很复杂。简单地说，当调试程序获得控制权时存储操作码的缓冲地址以做自用。

接着断点被显示并与其他断点进行比较。如果与列表中某一值一致，调试程序则停止运行。内存编辑器 / 查看器允许检查、编辑和转储被调试程序的内存镜像。

作者当时尽可能地优化指针使之更加可靠，但即便如此，仍然无法排除读取冲突，而且随意更改内存数据会导致被跟踪过程突然停止。如果使用一个不适合执行环境的指令来代替一个指令就有可能发生这种情况。因为调试程序会更改操作码，但最后执行操作码的不是调试程序而是虚拟机。

如果明确更改对象则不会发生这种情况。在后续版本中，调试程序将解决这个问题：它可以在错误发生前保存运行状态以使程序以正常模式继续执行。在任何情况下，就像在 SoftICE 中一样，如果汇编了错误代码，后果将一团糟。

所有过程都必须以图形形式出现在调试程序窗口中，所以 WKT VB Debugger 使用彩色代码来显示中断了的代码行。

8.8.5 VB6 P-code Crackme 分析实例

掌握了 P-code 的一些知识，就可以手工分析 P-code 程序获得源码了。一个 VB6 Pcode 分析实例（光盘：\chap08\example6\example6.exe）反汇编后 P-code 代码如下。

//程序见光盘:\chap08\example6\example6.exe

Proc: 402870 (Command1_Click)

40276C:	F5	LitI4:	0x0 0 (....)	; 7
402771:	71	FStR4	local_0098	; 1 local_0098=0
402774:	F5	LitI4:	0x0 0 (....)	; 7
402779:	71	FStR4	local_009C	; 1 local_009C=0
40277C:	04	FLdRfVar	local_00A4	; 7
40277F:	21	FLdPrThis		; 1 获取Form中ID=5控件
402780:	0F	VCallAd	(030C-02F8)/4	; 1 即text1对象指针
402783:	19	FStAdFunc	local_00A0	; 1 local_00A0=Text1Ptr
402786:	08	FLdPr	local_00A0	; 7 载入text1对象指针
402789:	0D	VCallHresult	A0	; 1 操作text1.GetText
40278E:	3E	FLdZeroAd	local_00A4	; 1 local_00A4清零
402791:	31	FStStr	local_0090	; 1 结果置入local_0090
402794:	1A	FFree1Ad	local_00A0	; 1 local_00A0清零
402797:	04	FLdRfVar	local_00A4	; 7
40279A:	21	FLdPrThis		; 1 获取Form中ID=6控件
40279B:	0F	VCallAd	(0308-2F8)/4	; 1 即text2对象指针
40279E:	19	FStAdFunc	local_00A0	; 1 local_00A0=Text2Ptr
4027A1:	08	FLdPr	local_00A0	; 7 载入text2对象指针
4027A4:	0D	VCallHresult	A0	; 1 操作text2.GetText
4027A9:	3E	FLdZeroAd	local_00A4	; 1 local_00A4清零
4027AC:	31	FStStr	local_0094	; 1 结果置入local_0094
4027AF:	1A	FFree1Ad	local_00A0	; 1 local_00A0清零
4027B2:	6C	ILdRf	local_0090	; 7
4027B5:	4A	FnLenStr		; 1 len(text1.text)
4027B6:	F5	LitI4:	0x9 9 (....)	; 1
4027BB:	D1	LtI4		; 1 len(text1.text)<9

4027BC:	6C	ILdRf	local_0090	; 7 len(text1.text)
4027BF:	4A	FnLenStr		;
4027C0:	F5	LitI4:	0xb 11 (....)	;
4027C5:	DB	GtI4		; 7 len(text1.text)>11
4027C6:	C4	AndI4		; (len(text1.text)<9)and (len(text1.text)>11)
4027C7:	6C	ILdRf	local_0094	; 7 len(text2.text)
4027CA:	4A	FnLenStr		;
4027CB:	F5	LitI4:	0x9 9 (....)	;
4027D0:	D1	LtI4		; 7 len(text2.text)>9
4027D1:	C5	OrI4	;if ((len(text1.text)<9) and(len(text1.text)>11)) ; or (len(text2.text)>9)	
4027D2:	1C	BranchF:	4027DA	;then goto 4027D5 else goto 4027DA
4027D5:	10	ThisVCallHresult	06F8	;call 40272C
4027DA:	F4	LitI2_Byte:	0x1 1 (.)	; 7
4027DC:	04	FLdRfVar	local_008A	;
4027DF:	6C	ILdRf	local_0090	;
4027E2:	4A	FnLenStr		;
4027E3:	E4	CI2I4		; for local_008A=1 to
4027E4:	FE	ad3/63 ForI2:	(when done) 402829	; 7 len(text1.text)
4027EA:	27	LitVar_Missing		; 7 length[null]
4027ED:	6B	FLdI2	local_008A	; start[local_008A]
4027F0:	E7	CI4UI1		;
4027F1:	6C	ILdRf	local_0090	; string[text1.text]
4027F4:	0B	ImpAdCallI2	rtcMidCharBstrr	; mid(text1.text, local_008A)
4027F9:	23	FStStrNoPop	local_00A4	; 7 结果置入local_00A4
4027FC:	0B	ImpAdCallI2	rtcByteValueBstr	; 7 即local_0088=AscB
402801:	E7	CI4UI1		; (text1.text, local_8A)
402802:	71	FStR4	local_0088	;
402805:	2F	FFreeIStr	local_00A4	; local_00A4清零
402808:	35	FFreeIVar	local_00C8	; 7 local_00C8清零
40280B:	6C	ILdRf	local_0098	; 7 local_0098 +

40280E:	6C	ILdRf	local_0088	;
402811:	F5	LitI4:	0x4 4 (....)	;
402816:	B2	MulI4		; local_0088*4
402817:	AA	AddI4		;
402818:	F5	LitI4:	0x12 18 (....)	;
40281D:	AA	AddI4		; + 18
40281E:	71	FStR4	local_0098	; 结果置入local_0098
402821:	04	FLdRfVar	local_008A	
402824:	64	NextI2:	(continue) 4027EA	; 循环
402829:	6C	ILdRf	local_0098	;
40282C:	F5	LitI4:	0x817fdb0	;
			135790000 (...)	
402831:	AA	AddI4		; local_0098=
402832:	71	FStR4	local_0098	; local_0098+135790000
402835:	6C	ILdRf	local_0094	; text2.text
402838:	0A	ImpAdCallFPR4:	rtcR8ValFromBstr	; val(text2.text)
40283D:	F4	LitI2_Byte:	0x18 24 (.)	;
40283F:	EB	CR8I2		;
402840:	AB	AddR8		; val(text2.text)+24
402841:	E8	CI4R8		; 类型转换real→Integer
402842:	71	FStR4	local_009C	; local_9C=val(text2.text)+24
402845:	6C	ILdRf	local_0098	;
402848:	6C	ILdRf	local_009C	;
40284B:	C7	EqI4		; local_98= local_9C ???
40284C:	1C	BranchF:	402867	; 不等则跳至402867
40284F:	1B	LitStr:		;
402852:	21	FLdPrThis		;
402853:	0F	VCallAd	(300-2f8)/4	;
402856:	19	FStAdFunc	local_00A0	;
402859:	08	FLdPr	local_00A0	; label3.Caption=

40285C:	0D	VCallHresult	54	;
402861:	1A	FFree1Ad	local_00A0	; ↓
402864:	1E	Branch:	40286c	; goto 40286C
402867:	10	ThisVCallHresult		; goto 40272c
40286C:	13	ExitProcHresult		;
Proc: 40272c				
402700:	27	LitVar_Missing		;
402703:	27	LitVar_Missing		;
402706:	27	LitVar_Missing		;
402709:	F5	LitI4:	0x0 0 (....)	;
40270E:	3A	LitVarStr:	(local_0094)	;
402713:	4E	FStVarCopyObj	local_00A4	;
402716:	04	FLdRfVar	local_00A4	;
402719:	0A	ImpAdCallFPR4:	RtcMsgBox	; 出错提示
40271E:	36	FFreeVar		;
402729:	FC	Lead1/c8 End		;
40272B:	13	ExitProcHresult		;

至此，可以还原大部分源代码，比如这样写：

```

Private Sub Command1_Click()

Dim name As String

Dim code As String

Dim sum As Long, i As Integer

name = Text1.Text

code = Text2.Text


If (Len(name) < 9 And Len(name) > 11) Or Len(code) < 9 Then

    MsgBox "错了！重新来吧"

```

```

Else:

For i = 1 To Len(name)

sum = sum + AscB(Mid(name, i)) * 4 + 18

Next i

sum = sum + 135790000

If sum = (Val(Text2.Text) + 24) Then Label3.Caption = "高手，有空来坐坐" Else MsgBox "错了！重新来吧"

End If

End Sub

```

8.9 VB 程序保护篇

自从发现 P-code 不为破解者熟悉后，国内越来越多的软件作者倾向于用 P-code 保护他们的作品。但根据笔者的经验，若完全依赖 P-code，被破解的可能性远大于用自然编译的软件。因为 P-code 是解释执行，目前虽做不到完全反编译，但部分反编译得到的资料对破解者来说已经足够了。

采用 P-code 保护程序的好处是很多人不熟悉，此方面的资料也很少，从而限制了更多破解者的投入。但是，为了对抗那些少数的 VB 破解“专家”，有必要采用更多的“抗破解”技术；应该依赖算法的复杂性起到阻止破解的作用。

8.9.1 Anti-Loader 技术

所谓 Loader，主要指 Smartcheck，wkt vb debugger 等的加载。从目前 VB 共享软件的保护情况看，源代码层面主要还是运用如下几种方法来对抗。

1. 查找特定的窗口标题

(1) 调用 VB 函数 AppActivate 语句查找 Smartcheck 窗口并关闭

这种方法见于 exdec 附带教程中的 VB crackme anti3.exe（伪编译），其 P-code 代码如下：

```

403937: 27 LitVar_Missing
40393A: 3a LitVarStr: (local_0094) Numega SmartCheck

```

40393F: 4e FStVarCopyObj	local_00A4
403942: 04 FLdRfVar	local_00A4
403945: 0a ImpAdCallFPR4:	rtcAppActivate
40394A: 36 FFreeVar	
403951: 63 LitVar_TRUE	
403954: 1b LitStr:	%{F4}
403957: 0a ImpAdCallFPR4:	rtcSendKeys
40395C: 35 FFree1Var	local_00A4
40395F: 63 LitVar_TRUE	
403962: 1b LitStr:	%Y
40396A: 35 FFree1Var	local_00A4

程序源代码如下：

AppActivate "Numega SmartCheck"	// 尝试激活 SmartCheck 窗口
	// "Numega SmartCheck"是 smartcheck 的窗口标题
SendKeys "%{F4}", True	// 发送 ALT+F4 关闭 smartcheck
SendKeys "%Y", True	// 发送 ALT+Y 确定

（2）用 API 函数 Findwindow()查找

程序实现代码如下：

FindWindow('nmscmw50',nil);	//nmscmw50 系 smartcheck 类名
或 FindWindow(nil,'numega SmartCheck');	

2. 时间计数

通过计算某段代码运行时间判断是否被加载（类似 Smartcheck, WKT VB Debugger 等 loader 载入目标程序时需要较多时间）。

以下代码见于某个共享软件（伪编译）：

46B729: 0a ImpAdCallFPR4:	rtcgettimer	// (1)
....		
46BEF0: 0a ImpAdCallFPR4:	rtcgettimer	// (2)

46BEF5: 6e FLdFPR4		
46BEF8: af SubR4		// (2)-(1)
46BEF9: ea CR4R4		
46BEFA: f4 LitI2_Byte:	0x14 20	
46BEFC: eb CR8I2		
46BEFD: dc GtR4		// >20 秒 ?
46BEFE: 1c BranchF:	46BF05	
46BF01: 00 LargeBos		
46BF03: Lead1/c8 End		结束 ← YES
46BF05: 00 LargeBos		继续 ← NO
...		
...		
46C34F: 0a ImpAdCallFPR4:	rtegettimer	// (3)
46C354: 6e FLdFPR4		
46C357: af SubR4		// (3)-(1)
46C358: ea CR4R4		
46C359: f4 LitI2_Byte:	0x14 20	
46C35B: eb CR8I2		
46C35C: dc GtR4		// >20 秒 ?
46C35D: 1c BranchF:	46C364	
46C360: 00 LargeBos		
46C362: Lead1/c8 End		结束 ← YES
46C364: 00 LargeBos		继续 ← NO

3. 利用 SEH 反加载

SEH 技术已经广泛应用于 Anti-Debugger 技术中，但是在 VB 中这样的应用还相当少见。此处给出一个利用 SEH 构建的例子，见光盘:\ chap08\example3\example3.exe。

这里通过构建自己的“异常处理”程序，来控制代码的走向。正常情况下，按下“SEH Test”按钮，会出现提示“No tracer found!”，若用 Smartcheck、TRW2000 等跟踪，则会自动结束程序。

8.9.2 VB “自锁”功能实现

设想一下，如果我们有一个共享软件，对试用客户限制某个功能的使用，只有取得注册码的客户才能使用被限制的功能。为此考虑，由于 VB 是模块化设计，每个事件的运行都依赖事件指针。我们定位功能限制模块的事件指针，并毁坏它。取用户的注册信息，计算后得到一组数值，将此数值作为事件指针回填。这样，除非注册信息正确，否则生成的事件指针将破坏程序的正常运行。这样做的好处是只限制了功能限制模块的运行，不会影响其他模块的正常工作。

此方法对所有基于事件模块设计的语言都适用，包括 Delphi，MFC 等。限于此方法的实现比较麻烦，因此在此只是提一下。光盘中的\chap08\example7\Example7.exe 是笔者做的一个演示。

8.10 相关工具点评

在本章即将结束之际，点评一下 VB 反编译的工具。

1. W32dasm 与 IDA pro

不需要再特别介绍的反汇编工具。特别地，W32dasm 对 VB 控件的事件的虚函数表显示得比较好，不论是何种编译方式。

2. SoftICE 与 TRW2000

极棒的调试器。不要忘了，两者都可以通过载入 msvbvm60.dbg 来支持符号调试。

3. SmartCheck

能记录 VB 程序运行时的函数调用情况。是研究 VB 程序运行的最佳软件。遗憾的是，Compuware 公司没有能在此基础上进一步开发。

4. VB 程序图形界面（GUI）编辑工具

VBDE 能称得上是一个“VB 半反编译器”，但作者声明已放弃继续开发。它能显示 VB6 标准控件的基本属性，事件。缺陷是没有完全按照 PE 结构来识别，导致若 VB 程序编译时其基地址设置特别的值，VBDE 即不能识别；识别标准控件的属性也有限，比如不能识别 `text.visible=false` 等。采用新版本的 Link.exe 链接生成 VB 程序也许能

使 VBDE 失效。

VB RezQ (<http://www.vbrezq.com>), 一个商业软件, 提供演示版本, 效果略好于 VBDE。

RACE (Reverse Action Contol Extraction), 其作者与 EXDEC 的作者为同一人, 特点在于对 VB 程序中包含的图像识别能力。

VBEditor、VBReformer (<http://perso.wanadoo.fr/vbeditor>), 提供编辑控件属性能力, 但对中文操作系统支持差。

5. VB Pcode 反编译工具

EXDEC: 作者 JosephC, 这是第一个支持 VB 5/6 P-code 的反编译软件, 具有重要价值。在它的影响下, 陆续出现了其他 VB5/6 版本的反编译器。每个研究 VB P-code 程序或需要破解 P-code 程序的人必备。

WKT VB Debugger: 代表了目前研究 VB5/6 反编译工作的最高成果。虽然早期其作者受到 EXDEC 作者的点拨, 但现在看来, 该作者对 VB 反编译的掌握已经超过前者。

6. 其他

COM 结构查看工具。在 VB 安装目录下有 VB6.olb 或 VB5.olb 文件, 其中定义了标准控件属性、事件与方法。可以使用微软提供的 Microsoft OLE/COM Object Viewer 或其他工具查看。有能力的读者也可以通过调用 `tlbinf32.dll` 来编程实现类似的查看工具。

John Chamberlain 发表过两篇文章。(1) Take Control of the Compile Process, 见于 VBPI 1999 年 11 期。其中讲述了 VB 编译的详细过程; 更重要的是, 其提供一个 `addin` 能截获 VB IDE 自然编译时生成的汇编代码, 这样通过对比源代码和汇编代码, 为反编译提供了条件。(2) Microsoft's P-code Implementation, 详细描述了 VB6 虚拟机 `msvbvm60.dll` 中 P-code 解释引擎 `ProcCallEngine` 实现的堆栈框架结构。随文提供的 P-code 伪指令表是第一次、完整出现, 后来 EXDEC, WKT VB Debugger, VBParser 使用的 P-code 伪指令可能皆源于此。随文提供的 `addin` 能在 VB IDE 下有限制地反编译 P-code 程序。此后, EXDEC 出现了……