

# 教程四：OlllyDbg 的使用（下）

## 一、简介

此次教程我们继续学习 Ollly 的使用。我们将继续使用上一章的程序（我也会将它包含在下载里）。

你可以在 [tutorials](#) 中下载文件和 PDF 版的教程。

## 二、DLLS

就像我前面说的，当你启动程序时，DLL 被系统载入器载入。这回我会细致的讲解。DLL (Dynamic Link Libraries) 是函数的集合，通常由 Windows 提供（当任何人都可以提供），其中含有很多 Windows 程序要用的函数。这些函数可以让程序员更容易的完成一些乏味的重复性的任务。

例如，将字符串全部转换成大写是许多程序要实现的功能。如果你的程序要多次使用该功能的话，你有三个选择：一是在你的程序中自己编码实现；问题是，你不知道你的下一个程序是不是也会用到该功能很多次。你可能需要在你使用到的程序里复制粘贴很多次相同的代码。二是创建一个自己的库，这样任何程序都可以调用。这种情况下，你可以创建一个 DLL，然后包含在程序中。该 DLL 可能有像 `convertToUpper` 这样的通用函数以便于程序调用，因此你只需要写一次代码就行了。这样做的另一个好处是，你可以说你为字符串转大写想到了一个很好的优化方案。第一个例子中，你需要将代码拷贝到所有要用到该代码的程序中，但是在那个通用 DLL 例子中，你只需要修

改 DLL 的代码，然后所有使用该 DLL 的程序都可以以最快的速度获益。爽吧！这就是 DLL 产生的真正原因。

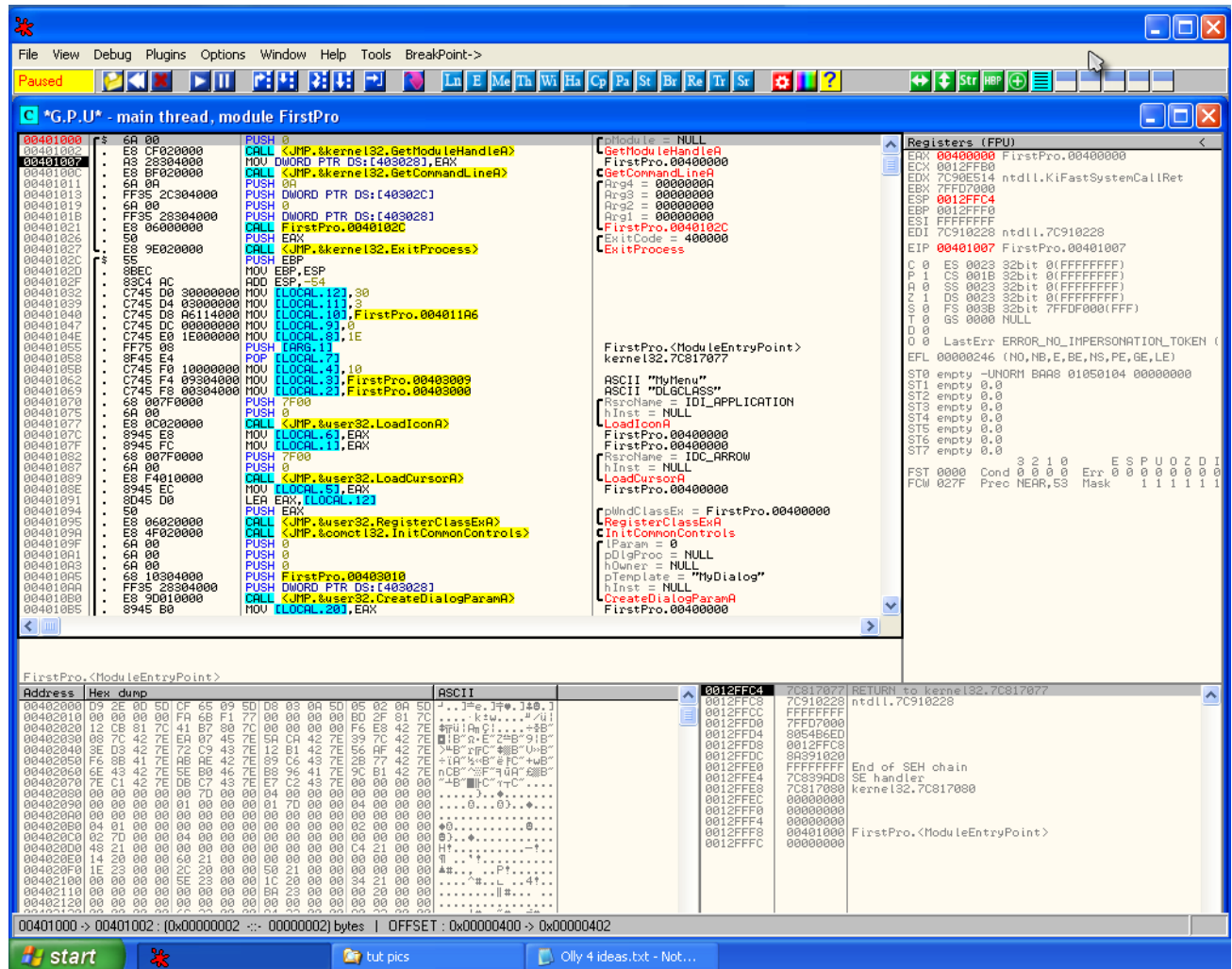
最后一个选择是，使用 Windows 提供的一堆 DLL 中包含的数千个函数中的一个。这样做有很多好处。第一个是，Microsoft 的程序员已经花了多年时间来优化他们的函数，他们在很大程度上要比你牛逼。第二，你不需要将你的 DLL 包含在应用中，因为 Windows 操作系统已经内建了这些 DLL。最后，如果 Windows 决定修改他们的操作系统，你自己的 DLL 有可能和新系统不兼容。同时，如果你使用 Windows 的 DLL，它们肯定是兼容的。

### 三、如何使用 DLL

现在你已经知道了什么是 DLL，那就谈谈如何使用它们。DLL 基本上就是一个你的程序可以调用的函数库。在你第一次载入应用程序时，Windows 载入器就会检查 PE 头（还记不记得 PE 头？）的特定区段，看看你的程序调用了哪些函数，以及这些函数都在哪些 DLL 中。在将你的程序载入内存后，载入器就迭代这些 DLL，将它们载入到你的应用程序的内存空间。然后它再仔细检查你的程序的代码，并将你的程序调用的 DLL 函数注入到正确的地址。例如，如果你的程序调用 Kernel32 DLL 中（只是一个例子啊）的 StrToUpper 函数来将一个缓冲区里的字母转换成大写，载入器要找到 Kernel32 DLL，找到 StrToUpper 函数的地址，并将地址注入到你的程序中调用该函数的那行代码处。你的程序就会通过调用进入到 Kernel32 DLL 的内存空间，执行 StrToUpper 函数，最后再返回到程序中。

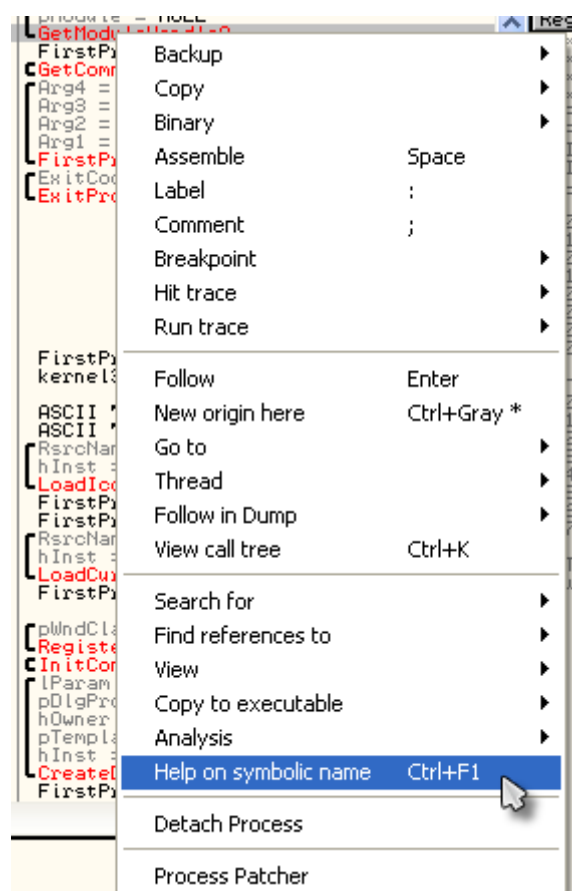
让我们实际看看这个过程。Olly 载入本教程包含的

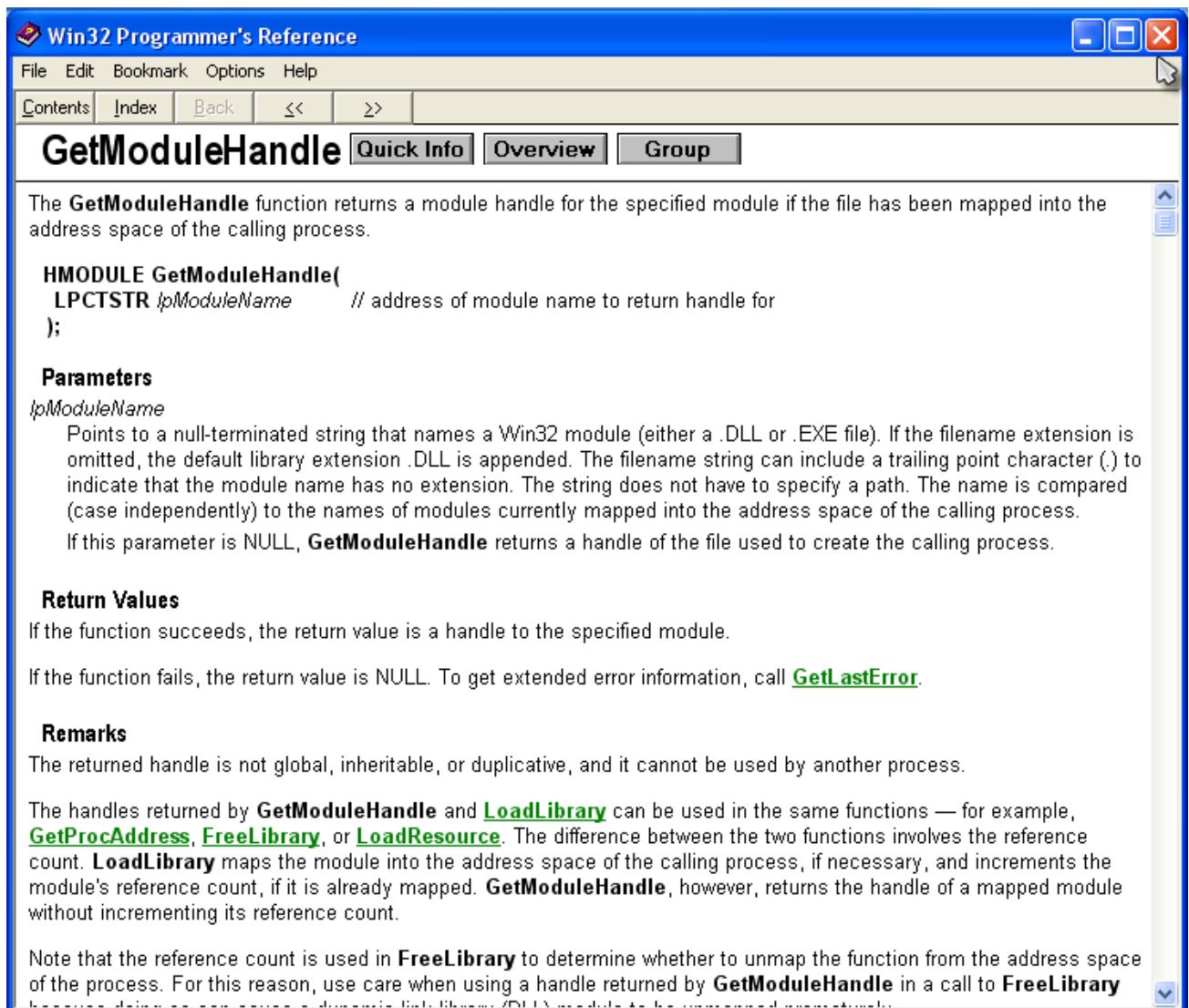
FirstProgram.exe。Ollly 断在了第一行代码（从现在起我们就叫它入口点（Entry Point）——这很重要，因为这是我们详细讨论 PE 头的时候 PE 头中的叫法）。



如果你看第二行代码的话，你会看到一个对函数 kernel32.GetModuleHandleA 的调用。第一步，我们看看这个函数是干嘛的。我已经将 WIN32.HLP 文件以及一个教你怎样将它安装到你的 Ollly 中的文本文档包含在了本课的下载里，就是为了防止你上一课没有拿到它。安装该文件后，你在你不熟悉的 Windows API 上右键，会显示一个该 API 是干什么的菜单。在你拷贝过去后，你需要重启下 Ollly。现在，在 GetModuleHandleA 上右键，选择“Help on Symbolic

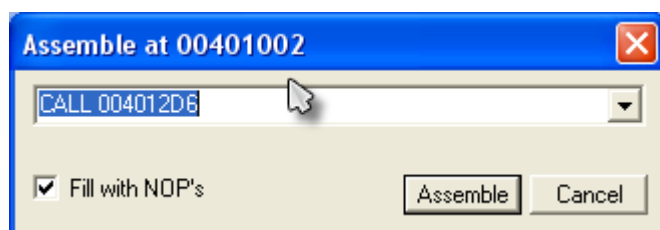
Name”。然后 Ollly 会显示一个该函数的备忘单：





那么，基本上这个函数就是为了获取我们程序内存空间的句柄。在 Windows 中，如果你想对一个窗口（或者是相当一部分的其他对象）做任何事情，你必须取得它的句柄。这基本上是 Windows 知道你正在操作的对象的最唯一标识符。GetModuleHandle 其实比这个稍微复杂点，不过当我们经历了更多知道了更多知识以后再回过头来讨论这个。

关闭帮助窗口，我们看看这个 CALL 去了哪里。Olly 已经试着帮助我们，它用函数名替换掉了 GetModuleHandleA 的真实地址。让我们看看它驻留的地址是什么。点一下调用 GetModuleHandleA 的那行代码，再按一下空格键，就会打开一个汇编窗口：



该窗口有两个目的：第一它向你显示了正在被计算的（以防 011y 帮助性的替换了地址）真实的汇编语言指令，第二它允许我们编辑汇编语言指令。在下一课前我们不会做任何编辑，这次我们只是看看地址：4012D6。有两种方法可以跳转到该地址看看那儿有什么（而不用真的运行程序）。选中“Call GetModuleHandleA”然后按下“Enter”，你也可以按 Ctrl+G 手动输入地址。我们试试第一种方法，选中 401002 那行（第三列有相关指令）然后按回车键，你就会来到该 CALL 要调用的地方：

004012D6	FF25 24204000	JMP	DWORD PTR DS:[<&kernel32.GetModuleHandleA>]	kernel32.GetModuleHandleA
004012DC	FF25 00204000	JMP	DWORD PTR DS:[<&comctl32.ImageList_Add>]	comctl32.ImageList_Add
004012E2	FF25 0C204000	JMP	DWORD PTR DS:[<&comctl32.ImageList_Create>]	comctl32.ImageList_Create
004012E8	FF25 08204000	JMP	DWORD PTR DS:[<&comctl32.ImageList_Destroy>]	comctl32.ImageList_Destroy
004012EE	FF25 04204000	JMP	DWORD PTR DS:[<&comctl32.InitCommonControls>]	comctl32.InitCommonControls
004012F4	00	DB	00	
004012F5	00	DB	00	
004012F6	00	DB	00	
004012F7	00	DB	00	
004012F8	00	DB	00	
004012F9	00	DB	00	
004012FA	00	DB	00	
004012FB	00	DB	00	
004012FC	00	DB	00	

现在这里比较有趣：它看起来确实不像执行 GetModuleHandleA 的代码。更像是一些跳转。对此有一个很好的原因说明，不过不幸的是，需要解释一下。

## 四、地址跳转表

有件事你需要知道，DLL 并不是总是一次性全部载入到内存。Windows 载入器负责载入你的程序和所有需要的 DLL，它可以修改被载入的 DLL 在内存中的位置（坦白的说，它甚至能够修改你的程序被载入的位置，这些我们后面再说）。原因是这样的，现在有一个 Windows DLL 属于最先载入的那种，被映射到地址 80000000。好吧，恰好你自

己的程序也带有一个 DLL 且需要载入到地址 80000000。两个 DLL 当然不能被载入到同一个地址，载入器必须将其中一个移到另一个地址。这种情况时常发生，还被叫做重定位。

这里有个问题：在你首次编写一个程序并写了一个调用 `GetModuleHandleA` 的指令，编译器会准确的知道正确的 DLL 在哪，然后它会放一个地址在指令里，有些类似于“`Call 80000000`”。现在，当你的程序被载入内存时，它仍然会让这个 CALL 调用 80000000（我这里说的有点过于简单了）。不过，如果载入器将这个 DLL 移到 80000E300 会怎么样？你的 CALL 会调用错误的函数！

PE 文件和此后的 Windows 文件围绕这个问题提出的解决方法是建立一个跳转表。意思是你的代码在首次编译时，每一个对 `GetModuleHandleA` 的调用都指向你的程序的一个地点，然后这个地点就会立即跳转到一个随意的地址（这是最后的正确的地址）。事实上，所有对 DLL 函数的调用都采用了同样的技术。它们每一个调用特定的地址，然后立即跳转到一个随意的地址。当载入器载入所有的 DLL 时，它会遍历“跳转表”，然后在内存中用真实的函数地址替换掉所有的随意地址。下面是所有真实地址被填充后的跳转表的样子：



0040124C	FF25	14204000	JMP	DWORD PTR DS:[<&gdi32.DeleteObject>]	gdi32.DeleteObject
00401252	FF25	74204000	JMP	DWORD PTR DS:[<&user32.CreateDialogParamA>]	user32.CreateDialogParamA
00401258	FF25	70204000	JMP	DWORD PTR DS:[<&user32.DefWindowProcA>]	user32.DefWindowProcA
0040125E	FF25	6C204000	JMP	DWORD PTR DS:[<&user32.DestroyWindow>]	user32.DestroyWindow
00401264	FF25	68204000	JMP	DWORD PTR DS:[<&user32.DispatchMessageA>]	user32.DispatchMessageA
0040126A	FF25	60204000	JMP	DWORD PTR DS:[<&user32.GetDlgItem>]	user32.GetDlgItem
00401270	FF25	64204000	JMP	DWORD PTR DS:[<&user32.GetDlgItemTextA>]	user32.GetDlgItemTextA
00401276	FF25	5C204000	JMP	DWORD PTR DS:[<&user32.GetMessageA>]	user32.GetMessageA
0040127C	FF25	58204000	JMP	DWORD PTR DS:[<&user32.IsDialogMessageA>]	user32.IsDialogMessageA
00401282	FF25	40204000	JMP	DWORD PTR DS:[<&user32.LoadCursorA>]	user32.LoadCursorA
00401288	FF25	2C204000	JMP	DWORD PTR DS:[<&user32.LoadIconA>]	user32.LoadIconA
0040128E	FF25	30204000	JMP	DWORD PTR DS:[<&user32.LoadImageA>]	user32.LoadImageA
00401294	FF25	34204000	JMP	DWORD PTR DS:[<&user32.MessageBoxA>]	user32.MessageBoxA
0040129A	FF25	38204000	JMP	DWORD PTR DS:[<&user32.PostQuitMessage>]	user32.PostQuitMessage
004012A0	FF25	3C204000	JMP	DWORD PTR DS:[<&user32.RegisterClassExA>]	user32.RegisterClassExA
004012A6	FF25	78204000	JMP	DWORD PTR DS:[<&user32.SendDlgItemMessageA>]	user32.SendDlgItemMessageA
004012AC	FF25	44204000	JMP	DWORD PTR DS:[<&user32.SetDlgItemTextA>]	user32.SetDlgItemTextA
004012B2	FF25	48204000	JMP	DWORD PTR DS:[<&user32.SetFocus>]	user32.SetFocus
004012B8	FF25	4C204000	JMP	DWORD PTR DS:[<&user32.ShowWindow>]	user32.ShowWindow
004012BE	FF25	50204000	JMP	DWORD PTR DS:[<&user32.TranslateMessage>]	user32.TranslateMessage
004012C4	FF25	54204000	JMP	DWORD PTR DS:[<&user32.UpdateWindow>]	user32.UpdateWindow
004012CA	FF25	20204000	JMP	DWORD PTR DS:[<&kernel32.ExitProcess>]	kernel32.ExitProcess
004012D0	FF25	1C204000	JMP	DWORD PTR DS:[<&kernel32.GetCommandLineA>]	kernel32.GetCommandLineA
004012D6	FF25	24204000	JMP	DWORD PTR DS:[<&kernel32.GetModuleHandleA>]	kernel32.GetModuleHandleA
004012DC	FF25	00204000	JMP	DWORD PTR DS:[<&comctl32.ImageList_Add>]	comctl32.ImageList_Add
004012E2	FF25	0C204000	JMP	DWORD PTR DS:[<&comctl32.ImageList_Create>]	comctl32.ImageList_Create
004012E8	FF25	08204000	JMP	DWORD PTR DS:[<&comctl32.ImageList_Destroy>]	comctl32.ImageList_Destroy
004012EE	FF25	04204000	JMP	DWORD PTR DS:[<&comctl32.InitCommonControls>]	comctl32.InitCommonControls
004012F4	00		DB	00	
004012F5	00		DB	00	
004012F6	00		DB	00	
004012F7	00		DB	00	

这个有点复杂，下面我举个例子。我会写一个短程序，使用完全随意的信息（只是为了证明我们的观点）来调用一个 Kernel32 DLL 中的函数 ShowMarioBrosPicture。下面是我的程序（没有特指哪种语言）：

```
main()
{
    call ShowMarioBrosPicture();
    call ShowDoYouLikeDialog()
    exit();
}

ShowDoYouLikeDialog()
{
    If ( user clicks yes )
    {
        call ShowMarioBrosPicture();
        Call ShowMessage( "Yes, it's our favorite too!")
    }
    else
    {
        call showMessage( "You obviously never played Super Mario
Bros. ");
    }
}
```

这些代码被编译之后，对函数的调用将会被真实的地址替换，就像下面这样（再次声明，这里没有特指某种语言）：



```

401000    call 402000    // Call ChowMarioBrosPicture
401002    call 401006    // Call showDoYouLikeDialog
401004    call ExitProcess
401006    Code for "Do You like It" dialog
.
.
.
40109A    if (user clicks yes)
40109C    call 402000    // call showMarioBrosPicture
40109E    call 4010FE    // call show message
4010a1    call ExitProcess
4010a3    if (user clicks no)
4010a5    call 4010FE    // call show message
4010a7    call ExitProcess

4010FE    code for show message
...
40110A    retn

```

这些代码的后面就有可能是我们的跳转表（本例中，跳转表中只有 ShowMarioBrosPicture）。

**402000 JMP XXXXXXXX**

我们的程序（译者注：这里作者应该是将 our program 写成了 out program，所以我给翻译成我们的，小伙伴们可以自己查阅）并不知道 ShowMarioBrosPicture 在哪（或者说不知道 Kerner32 DLL 在哪），我们程序的编译器只是用实际的调用地址填充 X（并不是正真的地址，你知道那么意思就行）。

当 Windows 载入器载入我们的程序时，它首先将二进制文件载入内存，完成跳转表的构建，不过跳转表里没有任何真实的地址。然后开始载入 DLL 到我们的内存空间，最后开始找出所有函数驻留的地方。一旦它找到了 showMarioBrosPicture 的地址，它就准备进入跳转表并

用函数的真实地址替换掉 X。假定 showMarioBrosPicture 的地址是 77CE550A。我们的跳转表代码就会被替换成如下：

```
402000 JMP 77CE550A
```

因为 Olly 能够发现该地址指向的是 showMarioBrosPicture，所以它会帮助性的进入跳转表并将跳转表显示如下：

```
402000 JMP DWORD PTR DS:[<&kernel32.showMarioBrosPicture>]
```

现在，让我们回到 FirstProgram 看看跳转表：

0040124C	FF25	14204000	JMP	DWORD PTR DS:[<&gdi32.DeleteObject>]	gdi32.DeleteObject
00401252	FF25	74204000	JMP	DWORD PTR DS:[<&user32.CreateDialogParamA>]	user32.CreateDialogParamA
00401258	FF25	70204000	JMP	DWORD PTR DS:[<&user32.DefWindowProcA>]	user32.DefWindowProcA
0040125E	FF25	6C204000	JMP	DWORD PTR DS:[<&user32.DestroyWindow>]	user32.DestroyWindow
00401264	FF25	68204000	JMP	DWORD PTR DS:[<&user32.DispatchMessageA>]	user32.DispatchMessageA
0040126A	FF25	60204000	JMP	DWORD PTR DS:[<&user32.GetDlgItem>]	user32.GetDlgItem
00401270	FF25	64204000	JMP	DWORD PTR DS:[<&user32.GetDlgItemTextA>]	user32.GetDlgItemTextA
00401276	FF25	5C204000	JMP	DWORD PTR DS:[<&user32.GetMessageA>]	user32.GetMessageA
0040127C	FF25	58204000	JMP	DWORD PTR DS:[<&user32.IsDialogMessageA>]	user32.IsDialogMessageA
00401282	FF25	40204000	JMP	DWORD PTR DS:[<&user32.LoadCursorA>]	user32.LoadCursorA
00401288	FF25	2C204000	JMP	DWORD PTR DS:[<&user32.LoadIconA>]	user32.LoadIconA
0040128E	FF25	30204000	JMP	DWORD PTR DS:[<&user32.LoadImageA>]	user32.LoadImageA
00401294	FF25	34204000	JMP	DWORD PTR DS:[<&user32.MessageBoxA>]	user32.MessageBoxA
0040129A	FF25	38204000	JMP	DWORD PTR DS:[<&user32.PostQuitMessage>]	user32.PostQuitMessage
004012A0	FF25	3C204000	JMP	DWORD PTR DS:[<&user32.RegisterClassExA>]	user32.RegisterClassExA
004012A6	FF25	78204000	JMP	DWORD PTR DS:[<&user32.SendDlgItemMessageA>]	user32.SendDlgItemMessageA
004012AC	FF25	44204000	JMP	DWORD PTR DS:[<&user32.SetDlgItemTextA>]	user32.SetDlgItemTextA
004012B2	FF25	48204000	JMP	DWORD PTR DS:[<&user32.SetFocus>]	user32.SetFocus
004012B8	FF25	4C204000	JMP	DWORD PTR DS:[<&user32.ShowWindow>]	user32.ShowWindow
004012BE	FF25	50204000	JMP	DWORD PTR DS:[<&user32.TranslateMessage>]	user32.TranslateMessage
004012C4	FF25	54204000	JMP	DWORD PTR DS:[<&user32.UpdateWindow>]	user32.UpdateWindow
004012CA	FF25	20204000	JMP	DWORD PTR DS:[<&kernel32.ExitProcess>]	kernel32.ExitProcess
004012D0	FF25	1C204000	JMP	DWORD PTR DS:[<&kernel32.GetCommandLineA>]	kernel32.GetCommandLineA
004012D6	FF25	24204000	JMP	DWORD PTR DS:[<&kernel32.GetModuleHandleA>]	kernel32.GetModuleHandleA
004012DC	FF25	00204000	JMP	DWORD PTR DS:[<&comctl32.ImageList_Add>]	comctl32.ImageList_Add
004012E2	FF25	0C204000	JMP	DWORD PTR DS:[<&comctl32.ImageList_Create>]	comctl32.ImageList_Create
004012E8	FF25	08204000	JMP	DWORD PTR DS:[<&comctl32.ImageList_Destroy>]	comctl32.ImageList_Destroy
004012EE	FF25	04204000	JMP	DWORD PTR DS:[<&comctl32.InitCommonControls>]	comctl32.InitCommonControls
004012F4	00		DB	00	
004012F5	00		DB	00	
004012F6	00		DB	00	
004012F7	00		DB	00	

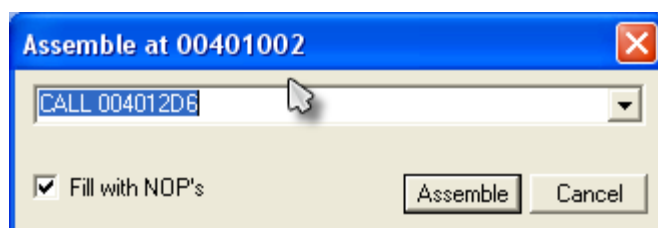
在首次编写这个程序时，各种 DLL 中的函数被调用，但是编译器不知道我们的程序在运行的时候这些函数是在内存的什么地方，所以它要创建一些像下面这样的东西（不是很准确这里）：

```
40124C JMP XXXXX // gdi32.DeleteObject
401252 JMP XXXXX // user32.CreateDialogParamA
401258 JMP XXXXX // user32.DefWindowProcA
40125E JMP XXXXX // user32.DestroyWindow
...
```

在载入器将我们的程序载入之后，再载入所有的 DLL 并查找所有的函数的地址。然后它会遍历每一个函数，用这些函数当前驻留的真实地址替换，就行前面图片中的那样。如果仔细想想的话，这确实是

相当巧妙的处理方法。如果不这样做的话，那么载入器就得遍历整个程序，并对每个 DLL 中的每个函数的调用都用真实的地址进行替换。那个工作量就大了。使用这种方法，载入器对于每个函数的调用只需要替换一个地方，就是跳转表那样的。

还是看看我们自己的程序吧。重载应用并按下 F7。点击选中 401002 那行指令（和前面做的一样），再按下空格（和前面做的一样）：




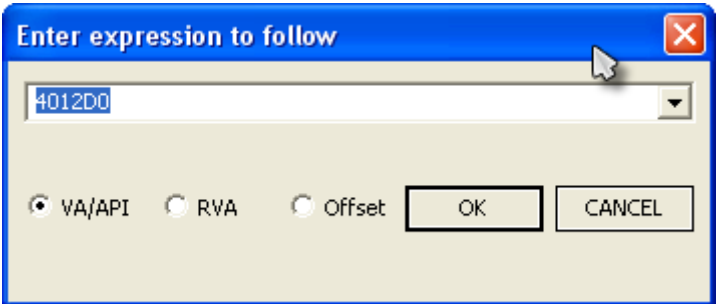
再一次提醒你注意那个地址，4012D6。现在按 F7 步入那个 CALL，注意我们来到了 4012D6。如果你向上翻，你会注意到我们来到了跳转表的中间：

0040124C	FF25 14204000	JMP DWORD PTR DS:[<&gdi32.DeleteObject>]	gdi32.DeleteObject
00401252	FF25 74204000	JMP DWORD PTR DS:[<&user32.CreateDialogParamA>]	user32.CreateDialogParamA
00401258	FF25 70204000	JMP DWORD PTR DS:[<&user32.DefWindowProcA>]	user32.DefWindowProcA
0040125E	FF25 6C204000	JMP DWORD PTR DS:[<&user32.DestroyWindow>]	user32.DestroyWindow
00401264	FF25 68204000	JMP DWORD PTR DS:[<&user32.DispatchMessageA>]	user32.DispatchMessageA
0040126A	FF25 60204000	JMP DWORD PTR DS:[<&user32.GetDlgItem>]	user32.GetDlgItem
00401270	FF25 64204000	JMP DWORD PTR DS:[<&user32.GetDlgItemTextA>]	user32.GetDlgItemTextA
00401276	FF25 5C204000	JMP DWORD PTR DS:[<&user32.GetMessageA>]	user32.GetMessageA
0040127C	FF25 58204000	JMP DWORD PTR DS:[<&user32.IsDialogMessageA>]	user32.IsDialogMessageA
00401282	FF25 40204000	JMP DWORD PTR DS:[<&user32.LoadCursorA>]	user32.LoadCursorA
00401288	FF25 2C204000	JMP DWORD PTR DS:[<&user32.LoadIconA>]	user32.LoadIconA
0040128E	FF25 30204000	JMP DWORD PTR DS:[<&user32.LoadImageA>]	user32.LoadImageA
00401294	FF25 34204000	JMP DWORD PTR DS:[<&user32.MessageBoxA>]	user32.MessageBoxA
0040129A	FF25 38204000	JMP DWORD PTR DS:[<&user32.PostQuitMessage>]	user32.PostQuitMessage
004012A0	FF25 3C204000	JMP DWORD PTR DS:[<&user32.RegisterClassExA>]	user32.RegisterClassExA
004012A6	FF25 78204000	JMP DWORD PTR DS:[<&user32.SendDlgItemMessageA>]	user32.SendDlgItemMessageA
004012AC	FF25 44204000	JMP DWORD PTR DS:[<&user32.SetDlgItemTextA>]	user32.SetDlgItemTextA
004012B2	FF25 48204000	JMP DWORD PTR DS:[<&user32.SetFocus>]	user32.SetFocus
004012B8	FF25 4C204000	JMP DWORD PTR DS:[<&user32.ShowWindow>]	user32.ShowWindow
004012BE	FF25 50204000	JMP DWORD PTR DS:[<&user32.TranslateMessage>]	user32.TranslateMessage
004012C4	FF25 54204000	JMP DWORD PTR DS:[<&user32.UpdateWindow>]	user32.UpdateWindow
004012CA	FF25 20204000	JMP DWORD PTR DS:[<&kernel32.ExitProcess>]	kernel32.ExitProcess
004012D0	FF25 1C204000	JMP DWORD PTR DS:[<&kernel32.GetCommandLineA>]	kernel32.GetCommandLineA
004012D6	FF25 24204000	JMP DWORD PTR DS:[<&kernel32.GetModuleHandleA>]	kernel32.GetModuleHandleA
004012DC	FF25 00204000	JMP DWORD PTR DS:[<&comctl32.ImageList_Add>]	comctl32.ImageList_Add
004012E2	FF25 0C204000	JMP DWORD PTR DS:[<&comctl32.ImageList_Create>]	comctl32.ImageList_Create
004012E8	FF25 08204000	JMP DWORD PTR DS:[<&comctl32.ImageList_Destroy>]	comctl32.ImageList_Destroy
004012EE	FF25 04204000	JMP DWORD PTR DS:[<&comctl32.InitCommonControls>]	comctl32.InitCommonControls
004012F4	00	DB 00	
004012F5	00	DB 00	
004012F6	00	DB 00	
004012F7	00	DB 00	

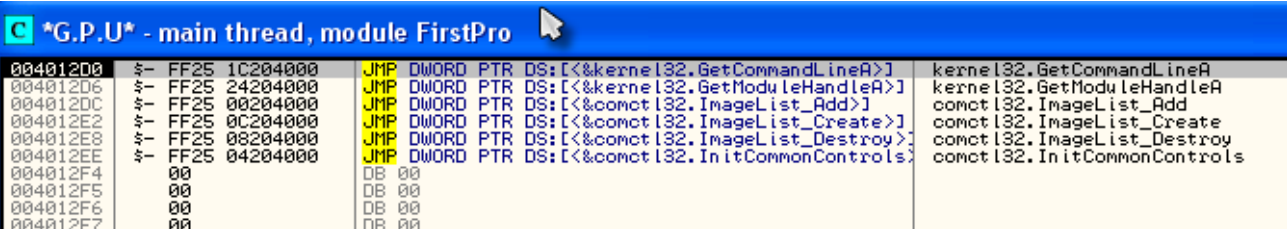
现在，再点一下 F7，我们就会来到 GetModuleHandleA 的真正的地址 7780B741。有两种方法可以知道我们现在正在模块 kernel32 中，两者在不同的场合你都可能用到。第一个是 Olly 的 CPU 窗口标题：



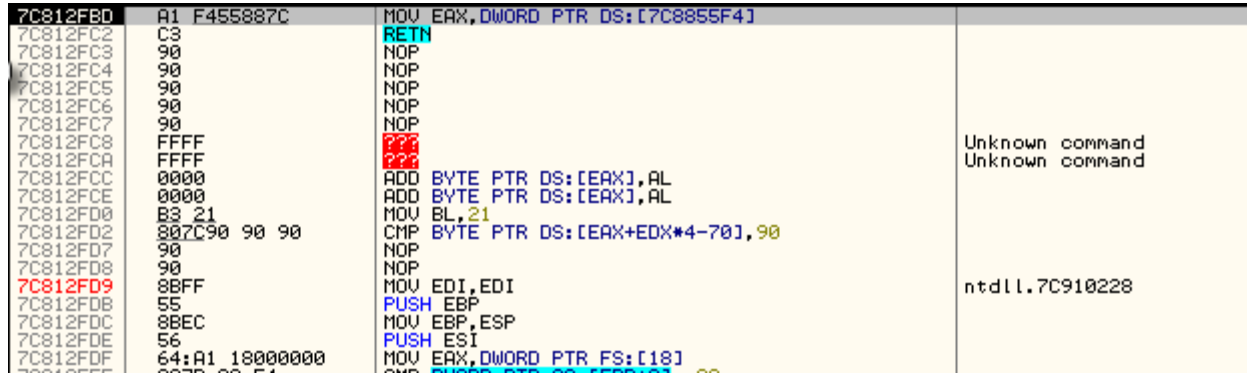
手动定位该地址，你会经常用到这种方法的。按 Ctrl+G 或点那个转到图标，输入我们要转到的地址：



你的“GOTO（转到）”窗口可能不太一样，这个问题待会解决。现在点击 OK，我们会跳到跳转表中 GetCommandLineA 的位置：



按下 F7 我们就来到了 kernel32 中的 GetCommandLineA 的开始处。这个函数从 7C812FBD 开始：



## 五、跳入及跳出 DLL

当我们围着一个程序转的时候，你不知道什么时候就在 DLL 中结束了。如果你正在尝试攻克一些保护方案时，通常你是不愿意在 DLL 中转的，因为 Windows DLL 中真的没什么东西。关于这方面的一个告诫，如果你正试着逆向的程序本身就带有 DLL 并且你就是想将它们也进行逆向工程（或者是保护机制确实在 DLL 中）。这里有几种从 DLL 回



到我们的程序的方法。一个方法是单步通过所有的 DLL 函数代码直到最后你返回到程序，当然这可能得一会时间（有些情况下像 VB 程序，就是永远）。第二个选择是，点开“Debug”菜单并选择“Execute till user code（执行直到用户代码）”或者按 Alt+F9。意思是执行 DLL 中的代码直到我们返回到我们自己的程序代码。要注意的是，有时候这不一定好使，因为如果 DLL 访问了一个在我们的程序空间中的 buffer 或者变量的话，Ollly 就会停在那儿，所以你最终可能会按 Alt+F9 好几次才能回来。

我们来试试这个方法。我们当前应该暂停在 7C812FBD，也就是 GetCommandLineA 的开始处。好，按下 Alt+F9。我们会回到程序中对 kernel32 调用的指令的后面那条指令（往上一行就是那个 CALL）。

现在我们来试试另外一个回到我们的代码的方法。重启程序，单步步过（F8）直到对 GetCommandLineA 调用的那个 CALL（40100C）。单步步入（F7）那个 CALL，并且单步步入那个 jmp 进入跳转表。现在，我们回到了 GetCommandLineA 的开始处：

7C812FBD	A1 F455887C	MOV EAX,DWORD PTR DS:[7C8855F4]	
7C812FC2	C3	RETN	
7C812FC3	90	NOP	
7C812FC4	90	NOP	
7C812FC5	90	NOP	
7C812FC6	90	NOP	
7C812FC7	90	NOP	
7C812FC8	FFFF	???	Unknown command
7C812FCA	FFFF	???	Unknown command
7C812FCC	0000	ADD BYTE PTR DS:[EAX],AL	
7C812FCE	0000	ADD BYTE PTR DS:[EAX],AL	
7C812FD0	B3 21	MOV BL,21	
7C812FD2	807C90 90 90	CMP BYTE PTR DS:[EAX+EDX*4-70],90	
7C812FD7	90	NOP	
7C812FD8	90	NOP	
7C812FD9	8BFF	MOV EDI,EDI	ntdll.7C910228
7C812FDB	55	PUSH EBP	
7C812FDC	8BEC	MOV EBP,ESP	
7C812FDE	56	PUSH ESI	
7C812FDF	64:A1 18000000	MOV EAX,DWORD PTR FS:[18]	
7C812FE0	5D	POP EBP	

现在打开内存映射窗口，滚动到我们的程序的代码段那块（起始地址是 400000，写着 PE Header）：

Address	Size	Owner	Section	Contains	Type	Access
00240000	00006000				Priv 00021004	RW
00250000	00003000				Map 00041004	RW
00260000	00016000				Map 00041002	R
00280000	00041000				Map 00041002	R
002D0000	00041000				Map 00041002	R
00320000	00006000				Map 00041002	R
00330000	00005000				Map 00041020	R E
003F0000	00002000				Map 00041020	R E
00400000	00001000	FirstPro		PE header	Imag 01001002	R
00401000	00001000	FirstPro	.text	SFX,code	Imag 01001002	R
00402000	00001000	FirstPro	.rdata	data,imports	Imag 01001002	R
00403000	00001000	FirstPro	.data		Imag 01001002	R
00404000	00001000	FirstPro	.rsrc	resources	Imag 01001002	R
00410000	00103000				Map 00041002	R
00520000	00001000				Priv 00021004	RW
00530000	000C2000				Map 00041020	R E
00830000	00001000				Priv 00021004	RW
00840000	00004000				Priv 00021004	RW
00850000	00003000				Map 00041002	R
00860000	00001000				Priv 00021040	RWE
00900000	00002000				Map 00041002	R
5D090000	00001000	comctl32		PE header	Imag 01001002	R
5D091000	00071000	comctl32	.text	SFX,code,imports,exports	Imag 01001002	R
5D102000	00003000	comctl32	.data		Imag 01001002	R
5D105000	00020000	comctl32	.rsrc	resources	Imag 01001002	R
5D125000	00005000	comctl32	.reloc		Imag 01001002	R

现在，点击选中 401000 那行，我们的 .text 区段在那行。按下 F2 设一个内存访问断点（或右键选择 Breakpoint on access）：

Address	Size	Owner	Section	Contains	Type	Access
00240000	00006000				Priv 00021004	RW
00250000	00003000				Map 00041004	RW
00260000	00016000				Map 00041002	R
00280000	00041000				Map 00041002	R
002D0000	00041000				Map 00041002	R
00320000	00006000				Map 00041002	R
00330000	00005000				Map 00041020	R E
003F0000	00002000				Map 00041020	R E
00400000	00001000	FirstPro		PE header	Imag 01001002	R
00401000	00001000	FirstPro	.text	SFX,code	Imag 01001002	R
00402000	00001000	FirstPro	.rdata	data,imports	Imag 01001002	R
00403000	00001000	FirstPro	.data		Imag 01001002	R
00404000	00001000	FirstPro	.rsrc	resources	Imag 01001002	R
00410000	00103000				Map 00041002	R
00520000	00001000				Priv 00021004	RW
00530000	000C2000				Map 00041020	R E
00830000	00001000				Priv 00021004	RW
00840000	00004000				Priv 00021004	RW
00850000	00003000				Map 00041002	R
00860000	00001000				Priv 00021040	RWE
00900000	00002000				Map 00041002	R
5D090000	00001000	comctl32		PE header	Imag 01001002	R
5D091000	00071000	comctl32	.text	SFX,code,imports,exports	Imag 01001002	R
5D102000	00003000	comctl32	.data		Imag 01001002	R
5D105000	00020000	comctl32	.rsrc	resources	Imag 01001002	R
5D125000	00005000	comctl32	.reloc		Imag 01001002	R

现在，运行程序。Ollly 会断在和上面相同的那行，就是 401011 处，也就是我们对 DLL 调用 CALL 之后的那行!!! 好，现在删除内存断点，否则你会纳闷，为什么每次你运行程序的时候它都会断在下一行

## 六、再议堆栈



堆栈是逆向工程中的非常重要的一部分，如果对它理解的不够深入的话，你永远也不会成为一个伟大的逆向工程师。下面我们针对它做几个实验：

首先，看看寄存器窗口（在重启应用之后），看那个 ESP 寄存器。该寄存器中的地址指向栈顶。本例中，ESP 的值是 12FFC4。现在看看下面的堆栈窗口，列表中的顶部地址和 ESP 中的地址是一样的。

0012FFC4	7C817077	RETURN to kernel32.7C817077
0012FFC8	7C910228	ntdll.7C910228
0012FFCC	FFFFFFFF	
0012FFD0	7FFD6000	
0012FFD4	8054B6ED	
0012FFD8	0012FFC8	
0012FFDC	870CB4D0	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C839AD8	SE handler
0012FFE8	7C817080	kernel32.7C817080
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00401000	FirstPro.<ModuleEntryPoint>
0012FFFC	00000000	

现在按 F8（或者 F7）一次，将 0 压入堆栈，再看看堆栈窗口：

0012FFC0	00000000	Module = NULL
0012FFC4	7C817077	RETURN to kernel32.7C817077
0012FFC8	7C910228	ntdll.7C910228
0012FFCC	FFFFFFFF	
0012FFD0	7FFD6000	
0012FFD4	8054B6ED	
0012FFD8	0012FFC8	
0012FFDC	870CB4D0	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C839AD8	SE handler
0012FFE8	7C817080	kernel32.7C817080
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00401000	FirstPro.<ModuleEntryPoint>
0012FFFC	00000000	

就像我们上次课提到的那样，该操作将 0（null）压入堆栈。现在看看 ESP 寄存器：

Registers (FPU)	
EAX	00000000
ECX	0012FFB0
EDX	7C90E514 ntdll.KiFastSystemCallRet
ESI	7FFD6000
ESP	0012FFC0
EBP	0012FFFF
ESI	FFFFFFFF
EDI	7C910228 ntdll.7C910228
EIP	00401002 FirstPro.00401002
CS	FS:0023 32bit 0(FFFFFFFF)

已经变成了 12FFC0。因为，在向堆栈中压入一个字节后，该字节就变成了新的栈顶。按 F8 一次，单步步过对 GetModuleHandleA 的调

用，再看看堆栈窗口：



0012FFC4	7C817077	RETURN to kernel32.7C817077
0012FFC8	7C910228	ntdll.7C910228
0012FFCC	FFFFFFFF	
0012FFD0	7FFD6000	
0012FFD4	8054B6ED	
0012FFD8	0012FFC8	
0012FFDC	870CB4D0	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C839A08	SE handler
0012FFE8	7C817080	kernel32.7C817080
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00401000	FirstProc.<ModuleEntryPoint>
0012FFFC	00000000	

注意我们的堆栈已经向下回退了一位(ESP 寄存器也回到了原来的值)。这是因为 GetModuleHandleA 函数使用了这个被压入堆栈的 0，并把它作为参数。然后把它“POP（弹）”出了堆栈，因为这个 0 已经没用了。就行上一课提到的，这是向函数传递参数的一种方法：将参数压栈，被调用的函数将它们弹出栈，使用它们，然后返回，通常我们需要的信息都在寄存器里（后面会看到）。

接着继续...。如果你按 F8 两次单步步过对 GetCommandLineA 的调用，会发现堆栈并没有改变。因为，我们没有向堆栈中压入任何信息以供函数使用。接下来，是一个 PUSH 0A 的指令。这是准备传递给下一个被调用函数的第一个参数。单步步过，然后你会发现 0A 出现在了栈顶，ESP 寄存器下移了 4（当你向堆栈压入一个值时，ESP 寄存器会向下移，因为堆栈在内存中是向下“增长”的。译者注：堆栈是从高址向低址增长。）现在再按一次 F8，ESP 寄存器会再次下移 4。因为我们向堆栈中压入了一个 4 字节的值。如果你看堆栈的顶部，就会发现我们向堆栈中压入了 00000000。为什么呢？

我们看看做这个压入操作的那行代码，在 401013 处：

```
PUSH DWORD PTR DS:[40302c]
```

这行代码的意思（我保证你知道什么意思，因为你已经学了汇编

语言:p) 是取地址 40302C 开始的 4 字节内容，然后将它们压入堆栈。那么在 40302C 的是什么呢？好吧，当然是 00000000！（开个玩笑）我们来自己看看。右键 401013 处的指令，选择“Follow in Dump（数据窗口跟随）”->“Memory Address（内存地址）”。然后会在内存数据窗口中显示以 40302C 开始的内存中的内容：

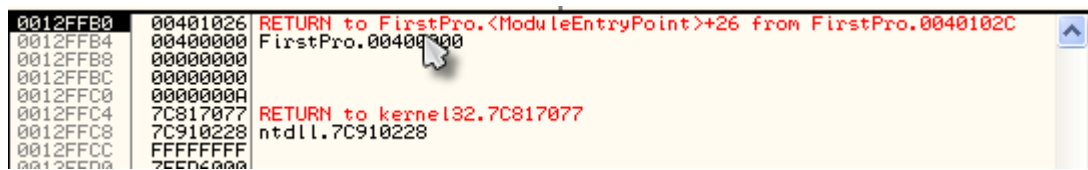
Address	Hex dump	ASCII
0040302C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0040303C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0040304C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0040305C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0040306C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0040307C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0040308C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0040309C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
004030AC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
004030BC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
004030CC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
004030DC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
004030EC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
004030FC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0040310C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0040311C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0040312C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0040313C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0040314C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

显然，哪里可没有那么多内容！不过你至少知道 0 是从哪儿来的。如果你想知道更多的细节比如这块内存是干什么的，这块内存空间被用来存储变量，并且最终会被这些变量填充。不过对于目前来说，所有的变量都被初始化为 0。

现在按一下 F8，我们遇到了另一个 PUSH 指令，不过这次是从 403028 开始。如果你在数据窗口中向上翻，会看到该地址处也是 0（在我们上一次课修改的字符串的后面）。这一块正在做的是将内存指针压栈，当前被设置为 0，我们的代码将会以变量的形式使用。单步步过一个 PUSH 然后单步步入对地址 40101C 的调用。你应该注意的第一件事是有什么东西被压入堆栈里了：我们的 CALL 的返回地址，401026。

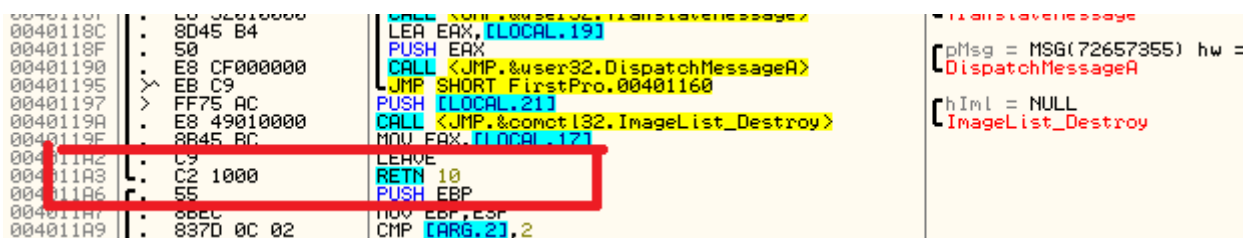
任何代码在使用 CALL 指令时，在我们还没有执行这个调用前，下一条将要被执行的指令（译者注：非 CALL 内部的指令）的地址会被自动的压入堆栈。原因是，我们调用的函数执行完后，它需要知道返回

到什么地方。被自动压入堆栈的地址就是返回地址。看那个堆栈窗口的顶部：



可以看到 Olly 已经指出了它是一个返回地址，并且它指回到我们的程序 (FirstPro)，需要被返回的地址是 40102C (CALL 的下一条指令)。

现在，在函数的结尾，一个 RETN 指令将会被执行 (你肯定知道它是“return”的意思，因为它出现在你的汇编语言书的开头处)。这个返回指令真正的意思是“弹出栈顶的地址，将正在运行的代码指向这个地址” (它主要是用弹出的值替换 EIP 寄存器——存储当前正在运行的行的地址)。那么现在，被调用的函数在执行完后准确的知道了要返回到哪！事实上，如果你向下滚动一点，就会发现 4011A3 处的 RETN 语句会从堆栈中弹出这个地址，然后从该地址开始运行：



(RETN 语句后面的那个 10，意思是给我返回地址，然后再从堆栈中删除 10h 字节的空间，因为我再也不需要它们了。看看你汇编语言书籍的下一页吧)

---

这里我们花点时间来启动一句，我保证在逆向工程社区会火的口头禅。我喜欢叫它“Random’s Essential Truths About Reversing Data (Random 关于逆

向数据的必备真言——译者注：大体这个意思吧，就这么翻吧，反正咱们也不会喊）”，或者 R. E. T. A. R. D（首字母缩写的听起来还不错）。我正式开启下面这个即将成为传奇的戒律：

## #1. You MUST learn assembly language（#1、你必须学习汇编语言）.

如果你还没有的话，在逆向工程领域你不会取得成功。就是那么简单。

本次教程我准备最后谈论的是，Ollly 怎么处理参数和本地变量的显示。如果你双击 EIP 寄存器，我们就能跳回到代码的当前行（在 40101C 处），往下可以看到好几行蓝色标记的行，显示的有 LOCAL 字样（其中一个显示 ARG）：

0040102F	83C4 AC	ADD ESP,-54	
00401032	C745 D0 30000000	MOV [LOCAL.12],30	
00401039	C745 D4 03000000	MOV [LOCAL.11],3	
00401040	C745 D8 A6114000	MOV [LOCAL.10],FirstPro.004011A6	
00401047	C745 DC 00000000	MOV [LOCAL.9],0	
0040104E	C745 E0 1E000000	MOV [LOCAL.8],1E	
00401055	FF75 08	PUSH [ARG.1]	
00401058	8F45 E4	POP [LOCAL.7]	
0040105B	C745 F0 10000000	MOV [LOCAL.4],10	
00401062	C745 F4 09304000	MOV [LOCAL.3],FirstPro.00403009	
00401069	C745 F8 00304000	MOV [LOCAL.2],FirstPro.00403000	
00401070	68 007F0000	PUSH 7F00	
00401075	6A 00	PUSH 0	
00401077	E8 0C020000	CALL <JMP.&user32.LoadIconA>	

FirstPro.00401026

ASCII "MyMenu"

ASCII "DLGCLASS"

RsrcName = IDI\_APPLICATION

hInst = NULL

LoadIconA

如果你没有任何编程经验，你可能不太知道本地变量和参数之间有什么不同。对于参数，就像我们早些时候讨论的，是传递给函数的变量，通常通过堆栈传递。本地变量是被调用函数“创建”的用来临时性存储数据的一种变量。下面是一个例子程序，其中有两个不同的概念：

```
main()
{
    sayHello( "R4ndom");
}

sayHello( String name)
{
```

```

    int numTimes = 3;
    String hello = "Hello, ";

    for( int x = 0; x < numTimes; x++)
        print( hello + name );
}

```

程序中，字符串“R4ndom”是传递给 sayHello 函数的参数。在汇编语言中，这个字符串（至少是这个字符串的地址）会被压入堆栈，以便于 sayHello 函数引用。一旦控制权转给了 sayHello 函数，sayHello 需要设置一对本地变量（LOCAL VARIABLES），这对变量函数会使用，不过一旦函数执行完毕就不再需要它们了。例子中的本地变量是整形数据 numTimes、字符串 hello、整形 x。不幸的是，为了防止堆栈不够负责，参数和本地变量都存储在堆栈中。堆栈通过 ESP 寄存器来实现这个，不过寄存器可没有超能力。它通常指向栈顶，不过它是可以被修改的。所以，可以说我们进入了 sayHello 函数，并且堆栈中有下面的数据：

- 1、字符串“R4ndom”的地址
- 2、让我们进入函数的那个 CALL 的返回地址。

如果我们想要创建一个本地变量，我所需要做的是从 ESP 寄存器中减去一定的值，这样就会在堆栈中创建一定的空间！假如我们将 ESP 减去 4（会有 4 个字节大小，或者一个 32 位的数）。堆栈会像下面这样：

- 1、空的 32 位数
- 2、字符串“R4ndom”的地址
- 3、让我们进入函数的那个 CALL 的返回地址。

现在，我们可以在这个地址里放任何数据，比如，我们可以让它



存储 sayHello 函数中的变量 numTimes。因为我们的函数使用了三个变量(所有的都是 32 位长), 需要从 ESP 减去 12 字节(或十六进制的 0xC), 然后我们就有了三个可以使用的变量。堆栈就会像下面这样:

- 1、指向字符串“hello”的空的 32 位地址。
- 2、变量“x”的空的 32 位数
- 3、变量“numTimes”的空的 32 位数
- 4、字符串“R4ndom”的地址
- 5、让我们进入函数的那个 CALL 的返回地址。

现在, sayHello 可以填充、修改以及重用这些地址以用于我们的变量, 在第一个位置处有传递给函数的参数(就是字符串“R4ndom”)。当 sayHello 执行完毕后, 它有两种方法来删除这些变量和参数(因为函数执行完毕后不再需要它们), 然后将堆栈还原: 1) 它可以将 ESP 寄存器修改回它被修改之前; 2) 使用后面带数字的 RETN 指令。第一种方法, 为了让程序能够记住 ESP 的原始数据, 它使用了另一个寄存器——EBP, 目的是当我们第一次进入 sayHello 函数时能够追踪到堆栈指向的原始位置。当函数准备返回时, 它从 EBP 中拷贝 ESP 的原始值(开始的时候存储在 EBP 中)到 ESP 和 EIP 中。返回地址现在在堆栈的顶部, 当 RETN 指令运行时, 它用通过这个返回到我们的主程序中。

第二种方法, 你可以告诉 CPU 堆栈中有多少字节你不再需要了, 然后它就会从栈顶删除这些字节。在我们的例子里, 我们用 RETN 16 (十六进制就是 0xF), 这样就会从栈顶除去 16 字节(或 4 个 32 位数), 将返回我主程序的地址留在新的栈顶。具体的返回机制依赖于编译器, 不过你两个都会看到。



现在，我们回到我们的 FirstProgram.exe:

0040102F	. 83C4 AC	ADD ESP,-54	
00401032	. C745 D0 30000000	MOV [LOCAL.12],30	
00401039	. C745 D4 03000000	MOV [LOCAL.11],3	
00401040	. C745 D8 A6114000	MOV [LOCAL.10],FirstPro.004011A6	
00401047	. C745 DC 00000000	MOV [LOCAL.9],0	
0040104E	. C745 E0 1E000000	MOV [LOCAL.8],1E	
00401055	. FF75 08	PUSH [ARG.1]	
00401058	. 8F45 E4	POP [LOCAL.7]	FirstPro.00401026
0040105B	. C745 F0 10000000	MOV [LOCAL.4],10	
00401062	. C745 F4 09304000	MOV [LOCAL.3],FirstPro.00403009	ASCII "MyMenu"
00401069	. C745 F8 00304000	MOV [LOCAL.2],FirstPro.00403000	ASCII "DLGCLASS"
00401070	. 68 007F0000	PUSH 7F00	RsrcName = IDI_APPLICATION
00401075	. 6A 00	PUSH 0	hInst = NULL
00401077	. E8 0C020000	CALL <JMP.&user32.LoadIconA>	LoadIconA

可以看到 Olly 已经注释出了一个参数和 12 个本地变量。在我们的程序中这些本地变量是用来追踪类似于图标、我们输入的文本的缓存地址、输入的文本长度等。完成后，就会弹出这些值、将 ESP 寄存器值改回 EBP 或 RETN 一个数字（本例中，三个都有!!!）

我知道堆栈是非常复杂的设计，但是我保证在混乱一段时间以后你会掌握它的窍门。汇编语言的书也会帮很大忙的。

（最近忙着装修进度较慢，而且第三章和第四章真的好长，这一章近万字，翻译不易呀）