

## 教程三：011yDbg 的使用（上）

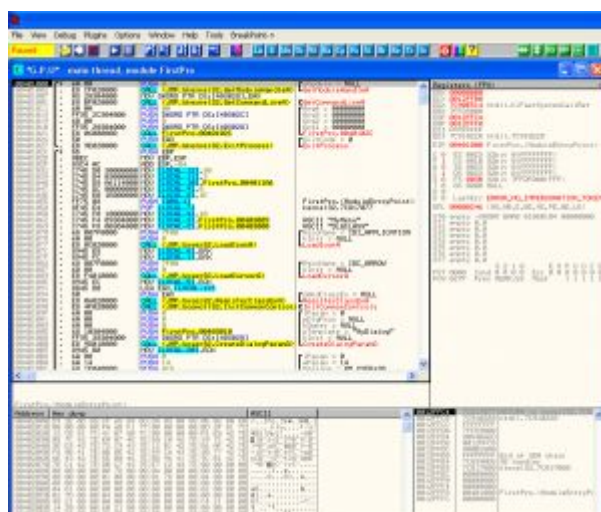
本章中我将会介绍 011yDbg 的使用。011y 有许多的功能，唯一学好它们的方式是实践和练习。也就是说，本教程也只是给你一个简单的概述。此教程不会涉及额外的内容，后面会进行重点讨论。到最后，你应该会比较好的掌握 011y。

本章包含了一些文件。你能够下载那些文件，以及可以在[这里](#)下载到次教程的 PDF 版本。

它们包括一个我们将在 011y 中用到的二进制文件、一个 011y 备忘单、我使用的外观上有些不同的 011y 以及一个新的 ini 文件。你可以用这个 ini 文件替换掉 011y 默认的 ini，可以给新人提供一些帮助(感谢伟大的 Lena151 做的这些)。你可以从[这里](#)直接下载或者从教程页面下载。如果你更愿意用原版的 011y，你可以从[这里](#)下载。

### 一、载入应用

第一步是将目的二进制文件载入 011y。你可以将二进制文件拖放到 011y 的反汇编窗口，或者点击顶部工具栏中的载入图标选择目的文件。我们这里载入 “FirstProgram.exe”，可以从本网站下载。011y 会进行分析（011y 的底部状态栏会显示分析进程）然后停在程序的入口点（EP）：



需要注意的第一件事是 EP 的地址是 401000，就是图片中的第一列。这是可执行文件的一个相当标准的起点（该可执行文件至少没有加过壳或混淆过）。如果你的看起来不太一样，并且 Olly 没有停在 401000，你可以尝试点击 Appearance 菜单，然后选择 debugging options，点击 ‘Events’ 标签，并且确保 ‘WinMain(if location is known)’ 被勾选上。然后重启应用。

让我们给 “FirstProgram.exe” 的内存空间占用情况来张快照。点击 “Me” 图标（如果你使用的是不同版本的 Olly 的话应该是 “M”）：

Memory map							
Address	Size	Owner	Section	Contains	Type	Access	Initial access
00010000	00001000				Priv 00021004	RW	RW
00020000	00001000				Priv 00021004	RW	RW
0012C000	00001000				Priv 00021104	RW	RW
0012D000	00003000			stack of main thread	Priv 00021104	RW	RW
00130000	00003000				Map 00041002	R	R
00140000	00005000				Priv 00021004	RW	RW
00240000	00006000				Priv 00021004	RW	RW
00250000	00003000				Map 00041004	RW	RW
00260000	00016000				Map 00041002	R	R
00280000	00041000				Map 00041002	R	R
002D0000	00041000				Map 00041002	R	R
00320000	00006000				Map 00041002	R	R
00330000	00004000				Map 00041020	R E	R E
003F0000	00002000				Map 00041020	R E	R E
00400000	00001000	FirstPro		PE header	Imag 01001002	R	RWE
00401000	00001000	FirstPro	.text	SFX,code	Imag 01001002	R	RWE
00402000	00001000	FirstPro	.rdata	data, imports	Imag 01001002	R	RWE
00403000	00001000	FirstPro	.data		Imag 01001002	R	RWE
00404000	00001000	FirstPro	.rsrc	resources	Imag 01001002	R	RWE
00410000	00103000				Map 00041002	R	R
00520000	00001000				Priv 00021004	RW	RW
00530000	00076000				Map 00041020	R E	R E
00830000	00001000				Priv 00021004	RW	RW
00840000	00004000				Priv 00021004	RW	RW
00850000	00003000				Map 00041002	R	R
00860000	00001000				Priv 00021040	RWE	RWE
00900000	00002000				Map 00041002	R	R
009EF000	00021000				Priv 00021104	RW	RW
50090000	00001000	comct132		PE header	Imag 01001002	R	RWE
50091000	00071000	comct132	.text	SFX,code, imports, exports	Imag 01001002	R	RWE
50102000	00003000	comct132	.data		Imag 01001002	R	RWE
50105000	00020000	comct132	.rsrc	resources	Imag 01001002	R	RWE
50125000	00005000	comct132	.reloc		Imag 01001002	R	RWE
76390000	00001000	imm32		PE header	Imag 01001002	R	RWE
76391000	00015000	imm32	.text	SFX,code, imports, exports	Imag 01001002	R	RWE
763A6000	00001000	imm32	.data		Imag 01001002	R	RWE
763A7000	00005000	imm32	.rsrc	resources	Imag 01001002	R	RWE
763AC000	00001000	imm32	.reloc		Imag 01001002	R	RWE
77DD0000	00001000	advapi32		PE header	Imag 01001002	R	RWE
77DD1000	00075000	advapi32	.text	SFX,code, imports, exports	Imag 01001002	R	RWE
77E46000	00005000	advapi32	.data		Imag 01001002	R	RWE
77E4B000	0001B000	advapi32	.rsrc	resources	Imag 01001002	R	RWE
77E66000	00005000	advapi32	.reloc		Imag 01001002	R	RWE
77F70000	00001000	rpcrt4		PE header	Imag 01001002	R	RWE
77F71000	00084000	rpcrt4	.text	SFX,code, imports, exports	Imag 01001002	R	RWE
77EF5000	00007000	rpcrt4	.orpc	code	Imag 01001002	R	RWE
77EFC000	00001000	rpcrt4	.data		Imag 01001002	R	RWE
77EFD000	00001000	rpcrt4	.rsrc	resources	Imag 01001002	R	RWE
77EFE000	00005000	rpcrt4	.reloc		Imag 01001002	R	RWE
77F10000	00001000	gdi32		PE header	Imag 01001002	R	RWE
77F11000	00043000	gdi32	.text	SFX,code, imports, exports	Imag 01001002	R	RWE
77F54000	00002000	gdi32	.data		Imag 01001002	R	RWE
77F56000	00001000	gdi32	.rsrc	resources	Imag 01001002	R	RWE
77F57000	00002000	gdi32	.reloc		Imag 01001002	R	RWE
77FE0000	00001000	secur32		PE header	Imag 01001002	R	RWE
77FE1000	0000D000	secur32	.text	SFX,code, imports, exports	Imag 01001002	R	RWE
77FEE000	00001000	secur32	.data		Imag 01001002	R	RWE
77FEF000	00001000	secur32	.rsrc	resources	Imag 01001002	R	RWE
77FF0000	00001000	secur32	.reloc		Imag 01001002	R	RWE
7C800000	00001000	kernel32		PE header	Imag 01001002	R	RWE
7C801000	00084000	kernel32	.text	SFX,code, imports, exports	Imag 01001002	R	RWE
7C885000	00005000	kernel32	.data		Imag 01001002	R	RWE

如果你看地址那一列，你会看到 401000 那行包含有大小 1000、名称 “FirstPro”（FirstProgram 的简写形式）、区块名 “.text”、包含里是 “SFX,code”。随着学习进度的展开，我们就会知道 exe 文件中有不同的区块，包含不同的数据类型。该区块中是程序的 “代码”。它有 1000 字节大，从内存的 401000 开始。

在这个的下面你会看到 FirstProgram 的其他区块。其中 .rdata 区包含着数据，其导入地址是 402000，地址 403000 的 .data 区中什么都没有。最后的那个 .rsrc 区中存有资源（比如对话框、图片、文本等）。要注意的是这些区可以叫任何名字，这个完全依赖于程序员。

你可能会问为什么 .data 区是空的。好吧，它事实上就是那样。它一般包含全局变量和随机数据。Olly 只是选择了不显示，因为它确实不知道那里存储了哪种数据。

区段的顶部是一个叫做 “Pe Header” 的区块。这是一个非常重要的区，一个我们会在将来文章中深入探讨的区。不过对于目前来说，我们只需要知道它对于 Windows 就像一本指令手册一样，用来按步将文件载入内存，程序运行需要多少空间，还有其他某些事情等。它在大约所有 exe 的头部（DLL 也是一样）。

如果你继续往下看，你可以看到不只是 FirstProgram 程序，还有其他的文件。我们看到有 comctl32, imm32, gdi32, kernel32 等。这些 DLL 是我们程序运行所需要的。DLL 是函数的集合，我们的程序能够调用那些 Windows 已经提供的（或者其他程序员提供的）函数。比如打开对话框、比较字符串、创建窗口以及类似的功能。统称为 Windows API。程序使用这些函数的原因是，假如我们写每一个用到的函数，仅仅显示一个消息框就需要数千行

的代码。然而，Windows 已经提供了像 `CreateWindow` 这样的函数来为我们做这些工作。对于程序员来说这使得编程要简单的多。

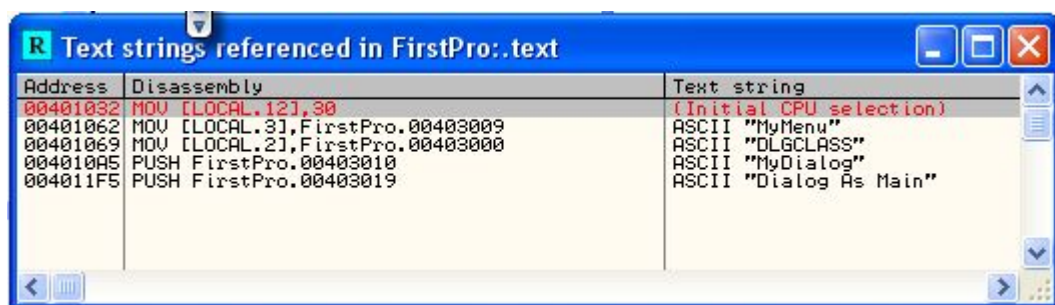
你或许会问这些 DLL 是如何进入我们的地址空间的，windows 是怎么知道哪一个是我们需要的。好吧，这些信息是存储在上列出的 PE Header 中的。当 Windows 将我们的 exe 载入内存时，它会检查头并找出 DLL 的名字，以及每个 DLL 中我们程序需要的函数，然后将这些函数载入我们的程序内存空间，以便于我们的程序调用它们。每个程序被载入内存时，它所需要的 DLL 也会被载入它的内存空间。可以想象得到，当有好几个程序当前都需要被载入内存并且都需要某个特定的 DLL 时，那么有些 DLL 就有可能被载入内存好几次。如果你需要准确的知道我们的程序调用了哪些函数，你可以右键点击 Olly 的反汇编窗口，选择 “Search for” ——> “All Intermodular Calls”。会显示如下图：



Found intermodular calls		
Address	Disassembly	Destination
00401000	PUSH 0	(Initial CPU selection)
00401002	CALL <JMP.&kernel32.GetModuleHandleA>	kernel32.GetModuleHandleA
0040100C	CALL <JMP.&kernel32.GetCommandLineA>	kernel32.GetCommandLineA
00401027	CALL <JMP.&kernel32.ExitProcess>	kernel32.ExitProcess
00401077	CALL <JMP.&user32.LoadIconA>	user32.LoadIconA
00401089	CALL <JMP.&user32.LoadCursorA>	user32.LoadCursorA
00401095	CALL <JMP.&user32.RegisterClassExA>	user32.RegisterClassExA
0040109A	CALL <JMP.&comctl32.InitCommonControls>	comctl32.InitCommonControls
004010B0	CALL <JMP.&user32.CreateDialogParamA>	user32.CreateDialogParamA
004010C9	CALL <JMP.&user32.SendDlgItemMessageA>	user32.SendDlgItemMessageA
004010E2	CALL <JMP.&user32.SendDlgItemMessageA>	user32.SendDlgItemMessageA
004010F1	CALL <JMP.&comctl32.ImageList_Create>	comctl32.ImageList_Create
0040110C	CALL <JMP.&user32.LoadImageA>	user32.LoadImageA
00401118	CALL <JMP.&comctl32.ImageList_Add>	comctl32.ImageList_Add
0040111F	CALL <JMP.&gdi32.DeleteObject>	gdi32.DeleteObject
00401136	CALL <JMP.&user32.SendDlgItemMessageA>	user32.SendDlgItemMessageA
00401143	CALL <JMP.&user32.GetDlgItem>	user32.GetDlgItem
00401149	CALL <JMP.&user32.SetFocus>	user32.SetFocus
00401153	CALL <JMP.&user32.ShowWindow>	user32.ShowWindow
0040115B	CALL <JMP.&user32.UpdateWindow>	user32.UpdateWindow
0040116A	CALL <JMP.&user32.GetMessageA>	user32.GetMessageA
0040117A	CALL <JMP.&user32.IsDialogMessageA>	user32.IsDialogMessageA
00401187	CALL <JMP.&user32.TranslateMessage>	user32.TranslateMessage
00401190	CALL <JMP.&user32.DispatchMessageA>	user32.DispatchMessageA
0040119A	CALL <JMP.&comctl32.ImageList_Destroy>	comctl32.ImageList_Destroy
004011B1	CALL <JMP.&user32.PostQuitMessage>	user32.PostQuitMessage
004011EE	CALL <JMP.&user32.GetDlgItemTextA>	user32.GetDlgItemTextA
00401202	CALL <JMP.&user32.MessageBoxA>	user32.MessageBoxA
00401219	CALL <JMP.&user32.SetDlgItemTextA>	user32.SetDlgItemTextA
00401229	CALL <JMP.&user32.DestroyWindow>	user32.DestroyWindow
0040123C	CALL <JMP.&user32.DefWindowProcA>	user32.DefWindowProcA

这有点惊奇，不过这个列表非常的小。通常，对于一个商业产品来说，需要数百或数千函数。不过因为我们的程序太简单了，它需要的不是很多。你想想我们的程序干了什么，看起来好像是那么多的函数只完成了如此简单的功能！欢迎来到 Windows。该窗口首先显示了 DLL 的名字，紧跟着的是函数的名字。比如，User32.LoadIconA 是在 DLL User32 中，函数名字是 LoadIconA。该函数通常用来载入窗口左上角的图标。

下一步，我们搜索下程序中的所有字符串。右键点击反汇编窗口，选择“SearchFor” -> “All Referenced Text Strings”：



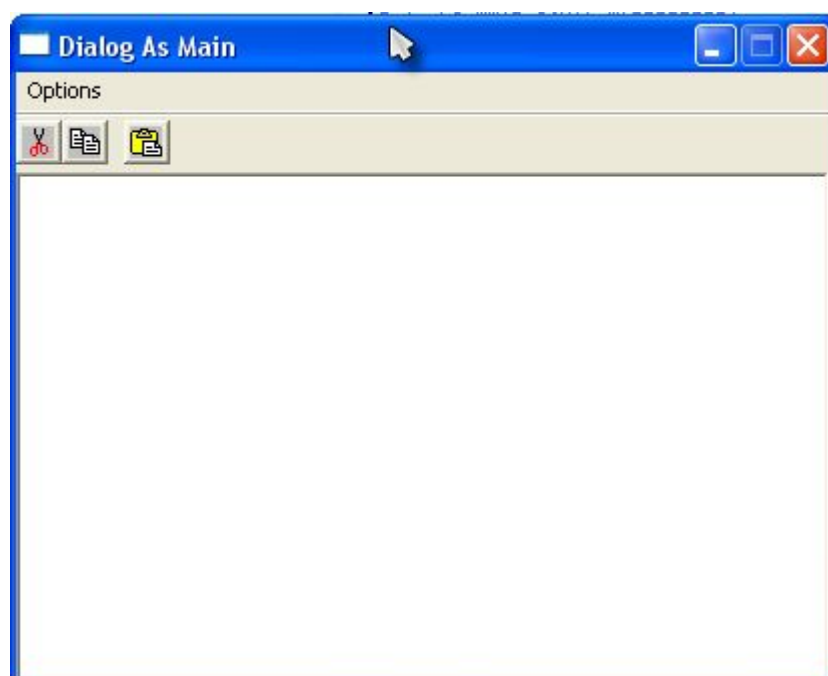
Address	Disassembly	Text string
00401032	MOV [LOCAL.12],30	(Initial CPU selection)
00401062	MOV [LOCAL.3],FirstPro.00403009	ASCII "MyMenu"
00401069	MOV [LOCAL.2],FirstPro.00403000	ASCII "DLGCLASS"
004010A5	PUSH FirstPro.00403010	ASCII "MyDialog"
004011F5	PUSH FirstPro.00403019	ASCII "Dialog As Main"

该窗口显示了我们程序中所有能找到的字符串。因为程序非常简单，所以这里只有一点。大多数的程序如果没有加壳或混淆的话，都有多得多的字符串（有时能达到十万）。这种情况下，你有可能一个也看不到！加壳工具这样做的原因是逆向工程师（至少新人是这样）严重依赖字符串来查找重要的函数。而删除了字符串后就会难的多。想象一下，如果你搜索字符串然后看到了“Congratulations! You entered the correct serial（恭喜！你输入了正确的序列号）”会怎么样？嗯，这对于逆向来说是巨大的帮助（我们会一次又一次的看到这个）。另外，双击其中的字符串，你会来到反汇编窗口中使用该字符串的指令那。这是一个很好的特性，你能够正确的跳转到使用字符串的代码。

## 二、运行程序

如果你看 011y 的左上角的话，会看到一个黄色背景的小区块，里面写着“暂停（Pause）”。意思是程序已经暂停了（本例中是在开始的时候），等着你进行其他操作。所以，咱们开始干一票吧！按一下 F9（或者从“Debug”菜单中选择“Run”）。一

会儿后，我们的程序会弹出一个对话框（它有可能显示在 011y 的后面，所以最小化 011y 以确保能看见窗口）。



刚才显示“Pause”的地方现在应该显示的是“Runing”。意思是程序正在运行，不过是在 011y 中运行的。你可能会与我们的程序进行一些交互，看看它是如何工作的以及它干了些什么。如果你不小心关了它的话，返回到 011y 并且按下 Ctrl+F2（或选择 Debug->Restart）以重新载入程序，然后你可以点击 F9 让程序再一次运行起来。

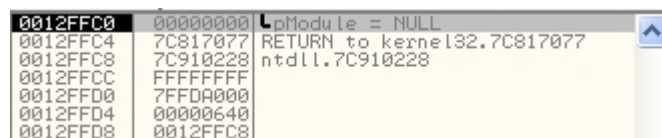
现在照着做：程序运行的时候，点击回到 011y 中，然后点击暂停图标（或点击 F12，也可以点击 Debug->Pause 菜单）。即使我们的程序正在运行，该操作会让程序暂停在内存中的任何地方。如果这时候你想看看程序，你会发现挺有意思的（程序一点



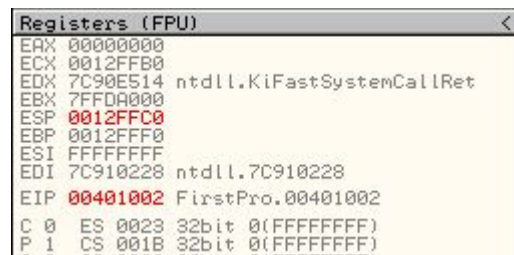
也不会显示出来)。这是因为当程序暂停的时候，Windows 不会更新视图。现在再一次点击 F9，你会发现你又可以和程序进行交互了。如果有什么问题的话，只需要点击那个双左尖括号图标或 Debug-restart (或者 ctrl-F2)，程序就会重新载入并暂停在入口处。如果你需要的话，你可以再一次运行它。

### 三、单步运行程序

运行一个程序确实挺爽，不过你却得不到有关于程序运行的太多信息。让我们试试单步运行。重新载入应用程序（重新载入按钮、Ctrl+F2 或 Debug->restart），然后我们会暂停在程序的开始处。按一下 F8，你就会发现当前的行选择器下移了一行。01ly 运行了一行指令，然后又暂停了下来。如果你够激灵的话，就会发现堆栈区下滚了一行，并且在顶部有了一个新的入口。



这是因为我们执行了一条指令，“PUSH 0”往堆栈里“压”了一个 0。在堆栈中的显示是“pModule=NULL”。NULL 是 0 的另一个名字。你有可能也注意到了那个寄存器区，ESP 和 EIP 寄存器变红了。



```
Registers (FPU)
EAX 00000000
ECX 0012FFB0
EDX 7C90E514 ntdll.KiFastSystemCallRet
EBX 7FFDA000
ESP 0012FFC0
EBP 0012FFF0
ESI FFFFFFFF
EDI 7C910228 ntdll.7C910228
EIP 00401002 FirstPro.00401002
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
```

当一个寄存器变红的时候，这就意味着最后执行的指令修改了该寄存器。本例中，ESP 寄存器（用来存放指向栈顶的地址）增加了 1，因为我们向栈中压了一个新值。EIP 寄存器增加了 2，其中存放了将要运行的指令的地址。因为我们已经不在地址 401000 了，而是在 401002。因为上一个运行的指令是两个字节长。我们现在暂停在下一个指令处。这个指令是在 401002，这正是当前 EIP 的值。

011y 现在暂停的指令是一个 CALL。CALL 指令意味着我们要临时暂停在我们当前所在的函数中，然后去运行另一个函数。这类似于高级语言中的方法调用，举个例子：

```
int main()
{
    int x = 1;
    call doSomething();
    x = x + 1;
}
```

这段代码中，我们首先让 x 等于 1，然后呢我们要在逻辑上暂停这行代码，转而去调用 doSomething()。当 doSomething() 执行完毕后，我们会返回我们原来的逻辑，然后将 x 加 1。

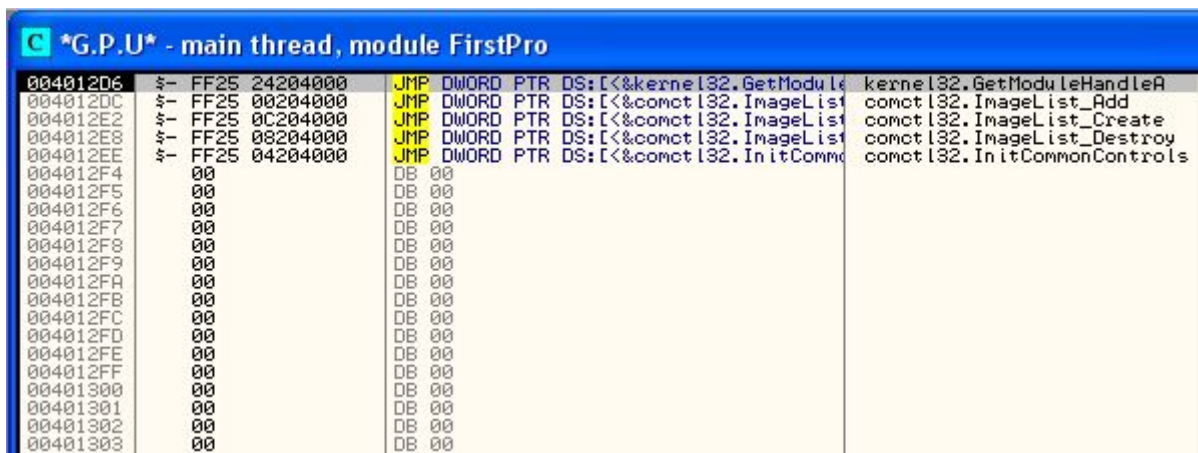
当然，在汇编语言里也是一样。我们首先往栈中压了一个 0，现在呢我们又想调用一个函数，例子中调用了 Kernel32.dll 中的 GetModuleHandleA()：



```
*G.P.U* - main thread, module FirstPro
00401000 6A 00 PUSH 0
00401002 E8 CF020000 CALL <JMP.<kernel32.GetModuleHandleA>
00401007 A3 28304000 MOV DWORD PTR DS:[403028],EAX
0040100C E8 BF020000 CALL <JMP.<kernel32.GetCommandLineA>
00401011 6A 00 PUSH 0
Comments: pModule = NULL, GetModuleHandleA, GetCommandLineA, Proc4 = 00000000
```

好，再按一次 F8。当前的行指示器会下移一行，而 EIP 仍然会保持红色并且加了 5 (因为刚刚运行的指令是 5 字节大小)，堆栈也回到了它原来的地方。刚刚发生的这些是从我们按下 F8 开始的，F8 的意思是 “Step-Over (单步步过)”，CALL 中的代码被调用，然后 Olly 暂停在了 CALL 的下一行。也就是 CALL 中的程序执行了也做了某些事，但是我们跳过去了。

好了，现在我们看看其他的选项。重启程序 (Ctrl+F2)，按下 F8 步过第一条指令，不过在 CALL 指令上我们这次按 F7。你会注意到整个窗体都变得不一样了：

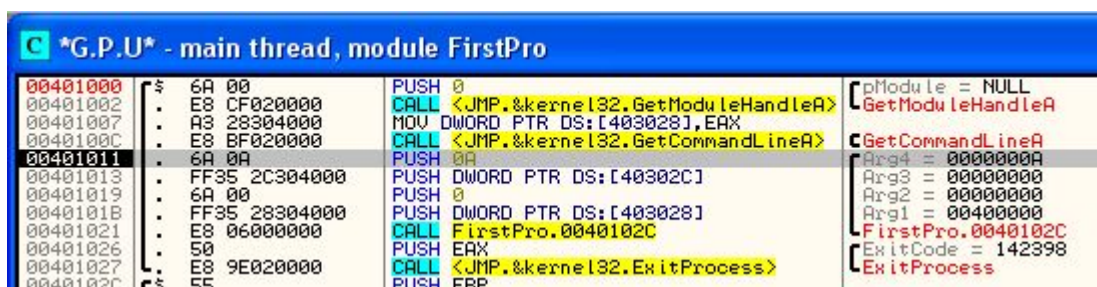


```
*G.P.U* - main thread, module FirstPro
00401206 FF25 24204000 JMP DWORD PTR DS:[<&kernel32.GetModuleHandleA>] kernel32.GetModuleHandleA
0040120C FF25 00204000 JMP DWORD PTR DS:[<&comctl32.ImageList_Add>] comctl32.ImageList_Add
004012E2 FF25 0C204000 JMP DWORD PTR DS:[<&comctl32.ImageList_Create>] 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 00 DB 00
004012F5 00 00 DB 00
004012F6 00 00 DB 00
004012F7 00 00 DB 00
004012F8 00 00 DB 00
004012F9 00 00 DB 00
004012FA 00 00 DB 00
004012FB 00 00 DB 00
004012FC 00 00 DB 00
004012FD 00 00 DB 00
004012FE 00 00 DB 00
004012FF 00 00 DB 00
00401300 00 00 DB 00
00401301 00 00 DB 00
00401302 00 00 DB 00
00401303 00 00 DB 00
```

这是因为 F7 “Step-In (单步步入)” 那个 CALL，意思是 Olly

做了这个调用并暂停在了新函数的第一行。这种情况下，CALL 跳转到了一个新的内存区域（EIP=4012d6）。理论上，如果我们按行通过这个新函数的话，我们最终还是会回到将我们带进来的那个 CALL 后面的语句。当然，有快捷键可以完成同样的功能，不过目前来说，咱们还是重启程序从头来吧。因为我怕教的太多容易忘。

现在我们暂停在了程序的开始，按下 F8（单步步过）4 次，我们会停在如下图的语句处：



The screenshot shows a debugger window titled "\*G.P.U\* - main thread, module FirstPro". It displays assembly instructions on the left and stack variables on the right. The assembly instructions are as follows:

Address	Disassembly
00401000	PUSH 0
00401002	CALL <JMP.&kernel32.GetModuleHandleA>
00401007	MOV DWORD PTR DS:[403028],EAX
0040100C	CALL <JMP.&kernel32.GetCommandLineA>
00401011	PUSH 0
00401013	PUSH DWORD PTR DS:[40302C]
00401019	PUSH 0
0040101B	PUSH DWORD PTR DS:[403028]
00401021	CALL FirstPro.0040102C
00401026	PUSH EAX
00401027	CALL <JMP.&kernel32.ExitProcess>
0040102C	FINISH FRP

The stack variables on the right are:

Variable	Value
pModule	NULL
GetModuleHandleA	
GetCommandLineA	
Arg4	00000000
Arg3	00000000
Arg2	00000000
Arg1	00400000
FirstPro.0040102C	
ExitCode	142398
ExitProcess	

你会看到在一块的四个 PUSH 语句。这回当你四次按下 F8 的时候，注意观察堆栈区，会看到栈的增长（确实是向下增长，还记得那个盘子的例子？）。我觉得我们开始理解什么是压栈了.....

你可能会问我们为什么要将这些乱七八糟的数字往栈里压。本例中这四个数字是作为参数传递给函数的（那个函数是在地址 401021 处）。我们将前面的那个高级语言程序做一点点修改就会比较清楚了：

```
int main()
```

```
{  
    int x = 1;  
    int y = 0;  
    call doSomething( x, y );  
    x = x + 1;  
}
```

这里我们声明了两个变量 `x` 和 `y`，并且将它们传递给了 `doSomething()` 函数。`doSomething` 函数将会（可能）对这些变量做些什么，然后将控制权还给调用该函数的程序。通过堆栈是将变量传递给函数的主要方法之一：每个变量被压进堆栈，然后调用函数。然后在函数中，这些变量被访问到。通常 `PUSH` 指令的逆操作是 `POP`。

堆栈并不是做这件事的唯一方法，它只是最常用的。这些变量也可以被放到寄存器中，然后在被调用的函数内部访问寄存器。不过本例中，我们程序的编译器选择将变量放到堆栈中。在你学了汇编语言后，这些东西都会变得清晰（你正在学习汇编语言，不是吗？）。后面我们还会复习几次的。

现在，如果我们再按一次 `F8`，你会注意到 `011y` 的工具栏中会显示 “`Runing`”，我们程序的对话框就会显示。这是因为我们单步步过了那个 `CALL`，说明那个 `CALL` 中存在程序的大部分。这个调用的代码是等待用户进行一些操作的循环，所以我们永远也不会将控制权交给 `CALL` 的下一行。那么，让我们修复它.....。



点击回到我们的程序，点那个关闭按钮来结束应用。011y 会立即暂停在那个 CALL 的下一行：

00401018	FF35 28304000	PUSH DWORD PTR DS:[403028]	Arg1 = 00400000
00401021	E8 06000000	CALL FirstPro.0040102C	FirstPro.0040102C
00401026	50	PUSH EAX	ExitCode = 0
00401027	E8 9E020000	CALL <JMP.&kernel32.ExitProcess>	ExitProcess
0040102C	55	PUSH EBP	

你会注意到我们的程序也消失了。那是因为，在那个 CALL 的某个地方，对话框窗口被关闭了。如果你看下一行，你会发现我们正准备调用 kernel32.dll -> ExitProcess。这是一个停止应用程序的 Windows API。所以，基本上 011y 在窗口被关闭了之后就暂停了，不过是在程序确实被终止之前。如果你这时按 F9，程序就会终止，011y 的活动栏就会显示 “Terminated (已终止)”，我们就再也不能调试任何东西了。

#### 四、断点

我们试试别的东西，重新载入应用 (Ctrl+F12)，然后在地 址 401011 处的第二列上双击（也就是双击那个 “6A 0A” 操作码）。然后地址 401011 就会变红：

00401000	6A 00	PUSH 0	pModule = NULL
00401002	E8 CF020000	CALL <JMP.&kernel32.GetModuleHandleA>	GetModuleHandleA
00401007	A3 28304000	MOV DWORD PTR DS:[403028],EAX	
0040100C	E8 BF020000	CALL <JMP.&kernel32.GetCommandLineA>	GetCommandLineA
00401011	6A 0A	PUSH 0A	Arg4 = 00000000
00401013	FF35 2C304000	PUSH DWORD PTR DS:[40302C]	Arg3 = 00000000
00401019	6A 00	PUSH 0	Arg2 = 00000000
0040101B	FF35 28304000	PUSH DWORD PTR DS:[403028]	Arg1 = 00000000
00401021	E8 06000000	CALL Tutorial.0040102C	Tutorial.0040102C
00401026	50	PUSH EAX	ExitCode = 0
00401027	E8 9E020000	CALL <JMP.&kernel32.ExitProcess>	ExitProcess
0040102C	55	PUSH EBP	
0040102D	8BEC	MOV EBP,ESP	
0040102F	83C4 AC	ADD ESP,-54	
00401032	C745 D0 30000000	MOV [LOCAL:123],30	

你刚才做的就是地址 401011 处设置断点。当 011y 到达该处时，断点就会强制 011y 暂停。有好几种不同的断点会因为不

同的事件而阻止程序运行。

### 1、软件断点 (Software Breakpoints)

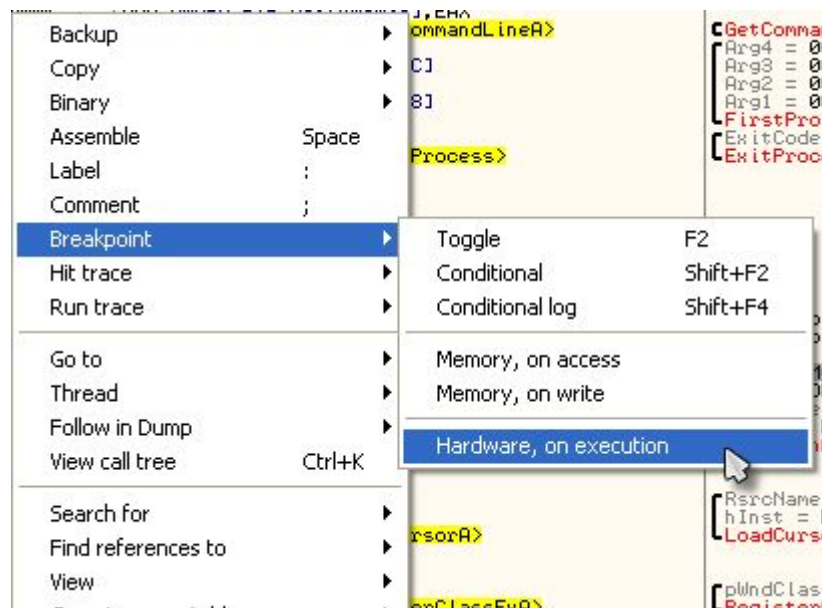
软件断点就是将断点所在地址处的字节用 0xCC 操作码替换掉，也就是 int 3 指令。这是一个特殊的中断，用以告知操作系统调试器希望在这里暂停，并且在执行该指令之前将控制权交给调试器。你不会看到指令被修改成 0xCC，因为 011y 在背后做了这个。当 011y 遇到异常时它会设一个陷阱，让用户做他们希望做的事。如果你选择让程序继续运行（通过运行它或单步运行），0xCC 操作码就会被原来的操作码替换回来。为了设置一个操作码，你可以双击操作码那一列，也可以先选中你想设置断点的那一行，然后右键点击它，选择 Breakpoints->Toggle（或按下 F2）。要删除断点你可以双击同一行，或右键点击选择 Breakpoints->Remove Software Breakpoint（或再次按下 F2）。

现在我们在 401011 处设置了一个 BP（Breakpoints），让程序暂停在第一行指令处。按下 F9，程序将运行并在我们设置的断点处暂停。

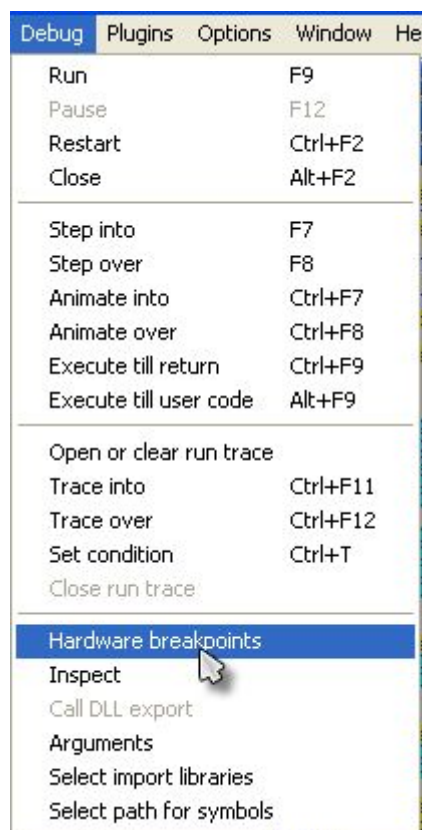
这里我告诉大家一些有用的东西。点击工具栏上的“Br”图标或选择菜单中的 View->Breakpoints。你会看到一个断点窗口，里面显示了我们设置的断点。



更可靠，尤其是在加壳或被保护的软件中。通过右键点击相关行可以设置硬件断点，选择 Breakpoints，然后选择 Hardware, on Execution。



唯一查看你已经设置的内存断点（译者注：这里应该是硬件断点）是打开“Debug”菜单，选择“Hardware Breakpoints”。有个插件可以提供方便，不过我们后面再讨论。



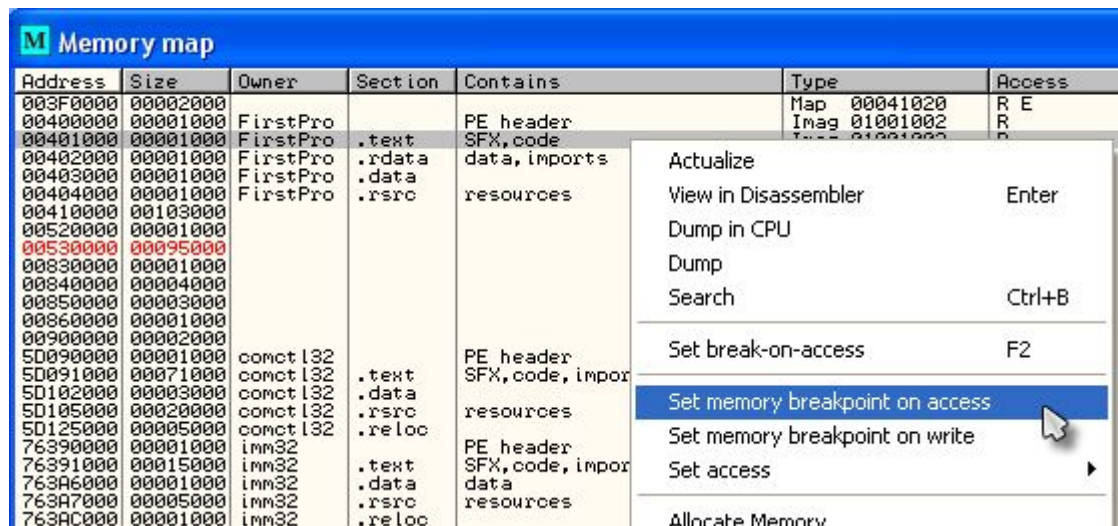
### 3、内存断点 (Memory Breakpoints)

有时候你想查找程序内存中的字符串或在常量,但是你又不知道程序在内存的什么地方。你可以用内存断点来告诉 011y 只要任何一条指令读或写一个内存地址 (或许多内存地址), 然后暂停就行, 在任何地方都无所谓。有三种方法设置内存断点。

- 对于一条指令,右键点击该行,然后选择 Breakpoint->Memory, On Access or Memory, On Write。
- 要在内存数据区设置断点, 在数据窗口中选中一个或多个字节, 然后右键选择和上面一样的操作。
- 你也可以对整个内存区域设置断点。打开内存映射窗口 ( “Me”



图标或 View->Memory ), 右键相关内存区域, 在弹出菜单中选择 “ Set Break On Access for either Access or Write”。



## 五、内存数据面板的使用

你可以用数据面板检查被调试进程内存空间中的内容。如果反汇编窗口的指令、寄存器或堆栈中的任何一项包含了对内存位置的引用, 你可以在该引用上右键然后选择 “Follow in Dump”, 随即数据面板就会向你显示该地址引用的内容。你也可以在数据面板的任何地方右键单击选择 “GoTo”, 然后输入要查看的地址。咱们现在试试。

确保 FirstProgram 已经载入并且停在了入口处。现在, 按下 F8 八次, 来到了 401021 地址指令处, 该处指令是 CALL FirstPro.40102c。如果你注意看这行的话, 会注意到这个 CALL 会向下跳转到 40102c 处, 在当前行下面的第三行的地方。按下 F7 我们单步步入那个跳转, 然后我们就来到了 40102c。记住这

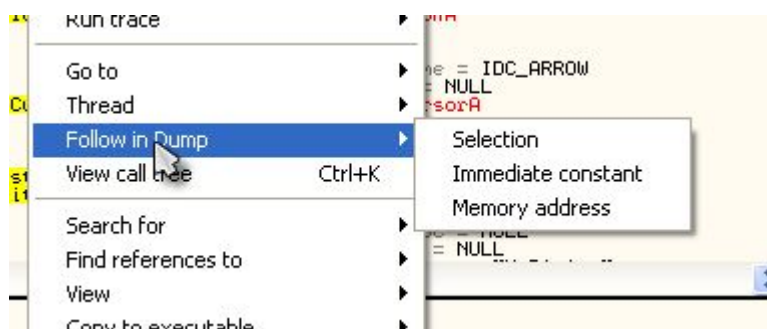
是一个 CALL 指令，所以我们最后还是会回到 401021 的（至少是该条指令后面的那条）。

00401019	6A 00	PUSH 0	Arg2 = 00000000
0040101B	FF35 28304000	PUSH DWORD PTR DS:[403028]	Arg1 = 00400000
00401021	E8 06000000	CALL FirstPro.0040102C	FirstPro.0040102C
00401026	50	PUSH EAX	ExitCode = 1423A8
00401027	E8 9E020000	CALL <JMP.&kernel32.ExitProcess>	ExitProcess
0040102C	55	PUSH EBP	
0040102D	8BEC	MOV EBP,ESP	
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	

现在，单步执行代码（F8）直到 401062。你也可以在这行设置断点，然后按 F9 运行。还记得怎么设置断点吗？双击你想设置断点的那行的操作码列。你也可以选中该行，然后按 F2 去设置或取消断点。现在我们断在了 401062：

00401055	FF75 08	PUSH [ARG.1]	FirstPro.00400000
00401058	8F45 E4	POP [LOCAL.7]	FirstPro.00400000
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"
00401069	68 007F0000	PUSH 7F00	RsrcName = IDI_APPLICATION
00401070	6A 00	PUSH 0	hInst = NULL
00401075	E8 0C020000	CALL <JMP.&user32.LoadIconA>	LoadIconA
00401077	8945 E8	MOV [LOCAL.6],EAX	
0040107C	8945 FC	MOV [LOCAL.1],EAX	

现在，我们看看断下的那行。相关指令是 MOV [LOCAL. 3]，FirstPro. 00403009。我确定你知道（因为你已经学了汇编语言:P）这条指令是将地址 00403009 中的内容移动到堆栈中（这里 011y 是用 LOCAL. 3 表示的）。你可以在注释列看到 011y 已经发现了该地址处的内容是字符串 “MyMenu”。好，下面让我们看看。在指令上右键，选择 “Follow in Dump(数据窗口跟随)”。注意这里有好几选项：



这里我们选择 “Immediate constant”。这回载入指令中的任何地址。如果你选择了 “Selection”，数据窗口会显示高亮行所在的地址，这里是 401062（也就是我们暂停的地方）。基本上我们在数据窗口中看到的就是我们在反汇编窗口中看到的。最后，如果我们选择了 “Memory address”，数据窗口会显示 LOCAL.3 的内存区域。这会显示我们正在使用的变量（在堆栈中）的内存。下面是选择了 Immediate constant 之后的数据窗口的样子：

Address	Hex dump	ASCII
00403009	4D 79 4D 65 6E 75 00 4D 79 44 69 61 6C 6F 67 00	MyMenu.MyDialog.
00403019	44 69 61 6C 6F 67 20 41 73 20 4D 61 69 6E 00 00	Dialog As Main..
00403029	00 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.e.....
00403039	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00403049	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00403059	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00403069	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00403079	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00403089	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00403099	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
004030A9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
004030B9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
004030C9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
004030D9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
004030E9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
004030F9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00403109	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

就像你看到的，数据窗口显示的内存是从 403009 开始的。正是 011y 按指令从中载入字符串的地址。在右边你可以看到字符串 “MyMenu”。左边你可以看到每个字符的十六进制数据。你可能注意到了在 “MyMenu” 后面有些其他的字符串。这些字符串会在程序的其他部分被用到。

## 六、最后，来点有意思的！

此次教程的最后，让我们做些有意思的事情。让我们修改二进制数据来显示我们自己的信息！我们将字符串 “Dialog As Main” 改成我们自己的，然后看看效果。

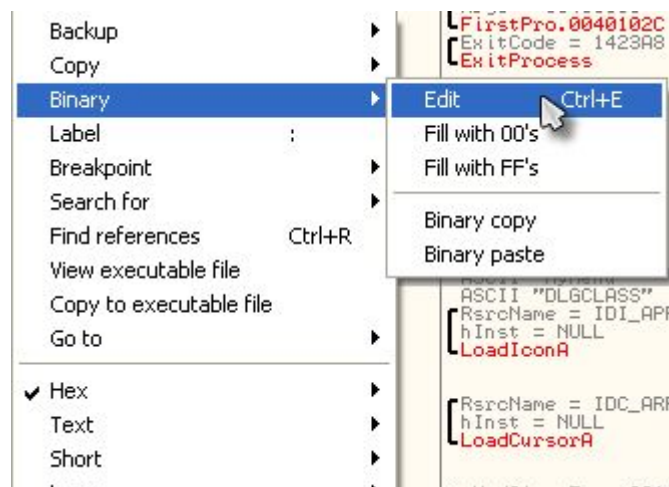
首先，在数据窗口的 ASCII 列，点那个 “Dialog As Main” 中的 “D”：

Address	Hex dump	ASCII
00403009	4D 79 4D 65 6E 75 00 4D 79 44 69 61 6C 6F 67 00	MyMenu.MyDialog.
00403019	44 69 61 6C 6F 67 20 41 73 20 4D 61 69 6E 00 00	Dialog As Main..
00403029	00 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.@.....
00403039	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

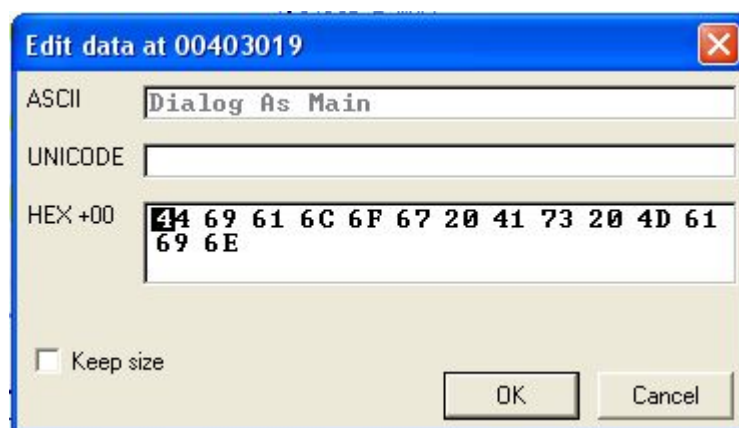
注意，左边的第一个十六进制数据也高亮了。这个数字对应字母 “D”。如果你查查 ASCII 码表的话，就会发现字母 “D” 的十六进制数正是 0x44。现在，选中整个 “Dialog As Main” 字符串：

Address	Hex dump	ASCII
00403009	4D 79 4D 65 6E 75 00 4D 79 44 69 61 6C 6F 67 00	MyMenu.MyDialog.
00403019	44 69 61 6C 6F 67 20 41 73 20 4D 61 69 6E 00 00	Dialog As Main..
00403029	00 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.@.....
00403039	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00403049	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

在选中的内容上右键，选择 “Binary” -> “Edit”。我们就可以修改我们程序在内存的内容：

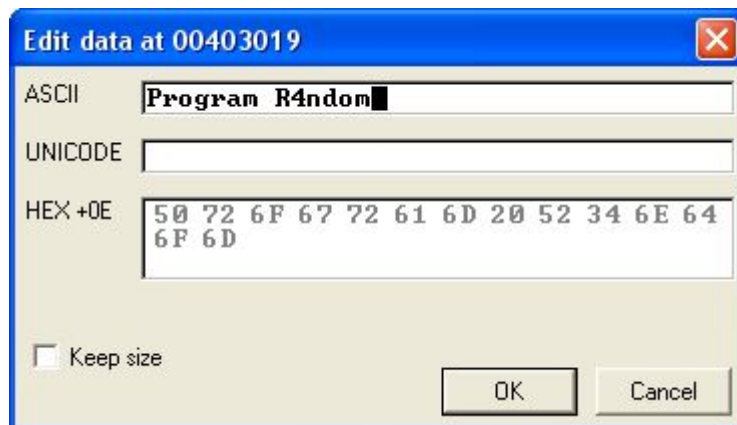


然后就会弹出一个如下的窗口：

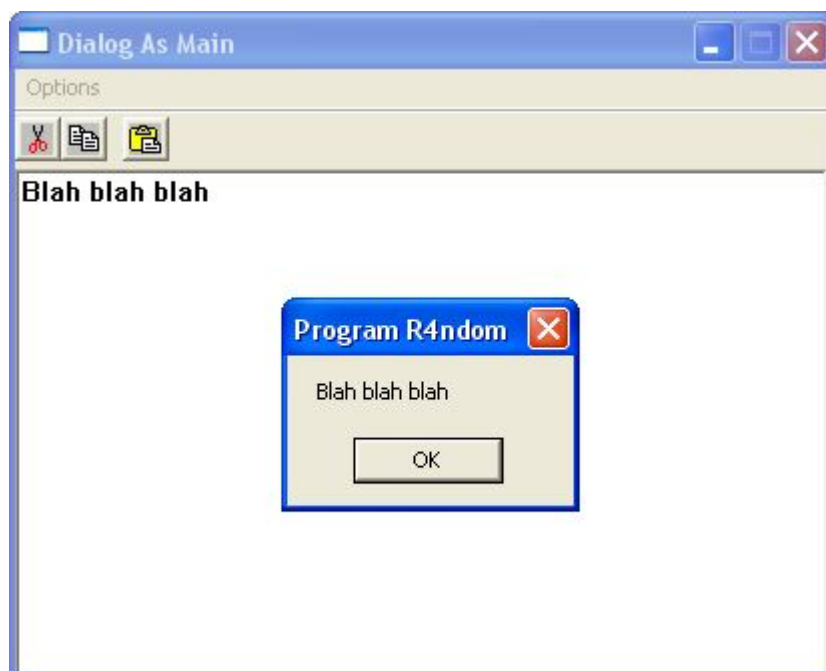


第一个文本框以 ASCII 码的形式显示字符串。第二个文本框是以 UNICODE 形式（我们的程序用不着，所以空着），最后那个文本框是相关字符串的原始数据。好，咱们改一下。点一下字符串的第一个字母（“D”），然后输入任何你想要将“Dialog As Main”覆盖掉的内容。要注意的是你输入的长度，别多了。否则你就会覆盖掉程序需要的其他字符串，或者更糟糕是覆盖掉了程序需要的代码!!! 这里呢，我输入的是“Program R4ndom”：





完了之后呢点 OK 按钮，并允许程序（点 011y 的运行按钮或按下 F9）。切换到我们的程序，然后随便输入什么都行，然后选择菜单 “Option” -> "Get Text"。现在看看我们的对话框！



注意到对话框的标题有什么不同没有 。

（这一章真 TM 长啊，翻的我累死了!!!）