

این قسمت درواقع ادامه ی قسمت قبل است و ما با توجه به آموخته های قبلی سعی خواهیم کرد آن را گسترش دهیم

DLLs

فایل های DLL یک فرمت استاندارد برای ویندوز مایکروسافت بوده و مخفف عبارت Dynamic-Link Library است. فایل های lib و oex و drv هم تقریباً مشابه DLL هستند. یک فایل DLL شامل توابع، کدها، منابع (تصویر، آیکون و ...) و داده هایی است که به توسعه دهندگان و برنامه نویسان این امکان را فراهم می کند که به آن ها لینک نموده و از توابعشان در برنامه های خود استفاده کنند. یکی از فواید مهم فایل های کتابخانه ای DLL این است که در یک زمان چندین برنامه می توانند از آن ها استفاده کنند در حالی که کدهایشان در یک مکان ثابت قرار داشته و نیازی به گرفتن فضای بیشتر برای هر برنامه نیست. از طرفی فرض کنید در حال نوشتن برنامه ای بزرگ هستیم و مدام باید یک کار تکراری مثلاً تبدیل رشته را انجام دهیم، بدیهی است که این کار خسته کننده است، اما با به کار گیری Dll فقط آن تابع را فراخوانی کنیم. ویندوز کالکشنی عظیم از توابع را دارا است که هر کدام کارهای بخصوصی انجام میدهند. به طور کلی اگر برنامه ی شما از توابع استفاده خواهد کرد شما دو انتخاب خواهید داشت :

- ۱- شما کد مربوط به هر عملیات را باید شفافاً بنویسید و مدام در حال Copy/Past در طول کد نویسی باشید؛ این واقعاً هم خسته کننده است و هم حجم کلی برنامه را افزایش میدهد و مستند سازی آن هم مشکل خواهد شد
- ۲- شما میتوانید توابع را درون یک Dll قرار داده و موقع نیاز آنها را به سادگی فراخوانی کنید (در هر زبان برنامه نویسی)

چرا این فایل ها را کتابخانه نامیده اند؟

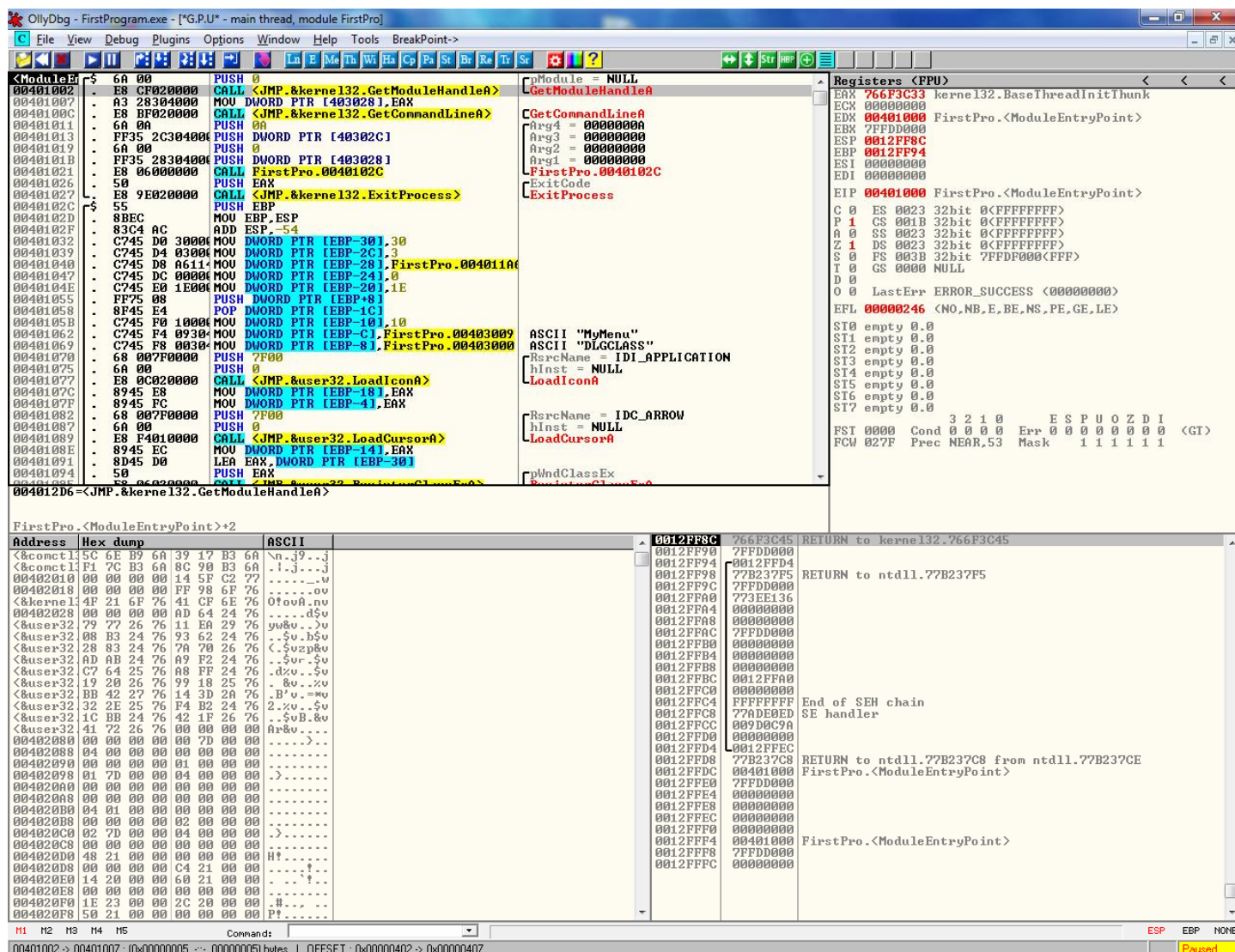
آشپزی و در هر یک از این کتاب های آشپزی یک دستور پخت بگذارید مثالی بزنم: یک کتابخانه عمومی را فرض کنید که پر از کتاب های نیز این چنین هستند؛ یعنی پر از توابع DLL افراد به کتابخانه سر زده و از آن کتاب ها استفاده می کنند. فایل های نوشته شده است و می دهند و برنامه ها می توانند از آن ها استفاده کنند مختلف که هر یک کار خاصی را انجام

Dll ها چگونه استفاده میشوند

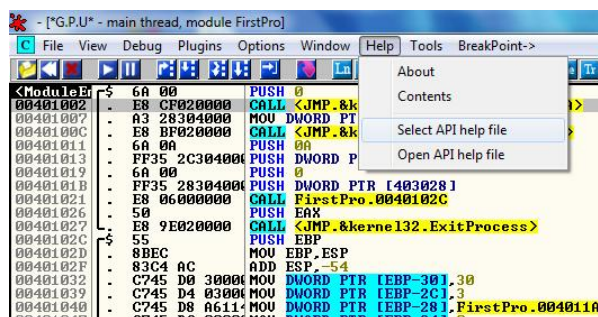
حال که با Dll آشنا شدیم، بگذارید ببینیم چگونه مورد استفاده قرار میگیرد، وقتی که شما برنامه ی تان را اجرا میکنید، ویندوز لودر سکشن خاصی در PE هدر را چک میکند (یادتان هست؟) چه چیزی را؟ در واقع Dll ها و توابع مورد استفاده در آنها و همچنین آدرس آنها را، وقتی که برنامه در آدرس حافظه لود شد. توابع لازم و ضروری را در فضای حافظه ی مختص برنامه ی شما با آدرس صحیح، هرکجا که برنامه از توابع استفاده کرده Inject میکند.

مثلاً تابع StrToUpper واقع در Kernel32.dll را در نظر بگیرید، کار این تابع نمایش رشته ها با حروف بزرگ است، حال اگر برنامه شما از این تابع استفاده کرده باشد به این شکل StrToUpper Call، ویندوز لودر ابتدا Kernel32.dll را وبعد آدرس تابع StrToUpper را پیدا کرده و آدرس مورد نظر را در فایل ما Inject میکند. و سپس به برنامه باز میگردد.

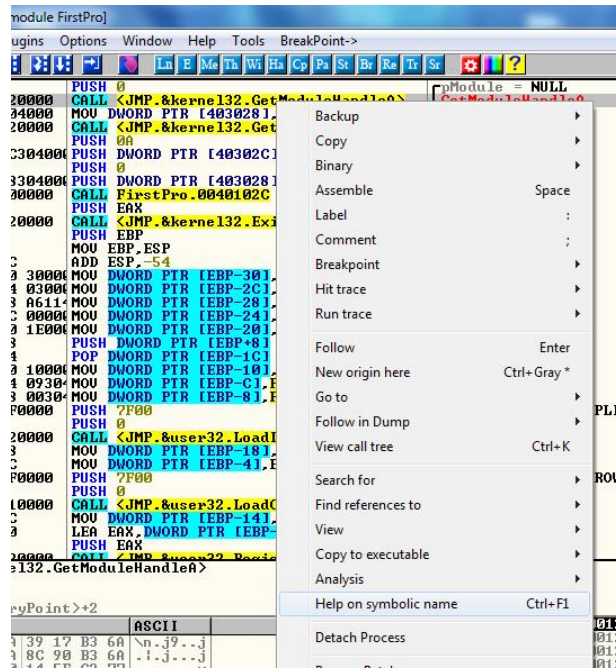
باید تمام توضیحات بالا را به صورت عملی ببینیم در ضمیمه ی این آموزش فایل با نام "FirstProgram.exe" موجود است، آن را در OllyDBG بارگذاری کنید. OllyDBG در خط اول متوقف میشود که به آن Entry Point یا به اختصار EP گفته میشود. EP مملى است که ویندوز لودر بعد از اتمام کار خود به آن قسمت پرش میکند و کنترل به دست برنامه جهت اجرا می افتد. اگر به خط دوم نگاه کنید میبینید که تابعی با نام kernel32.GetModuleHandleA فراخوانی شده:



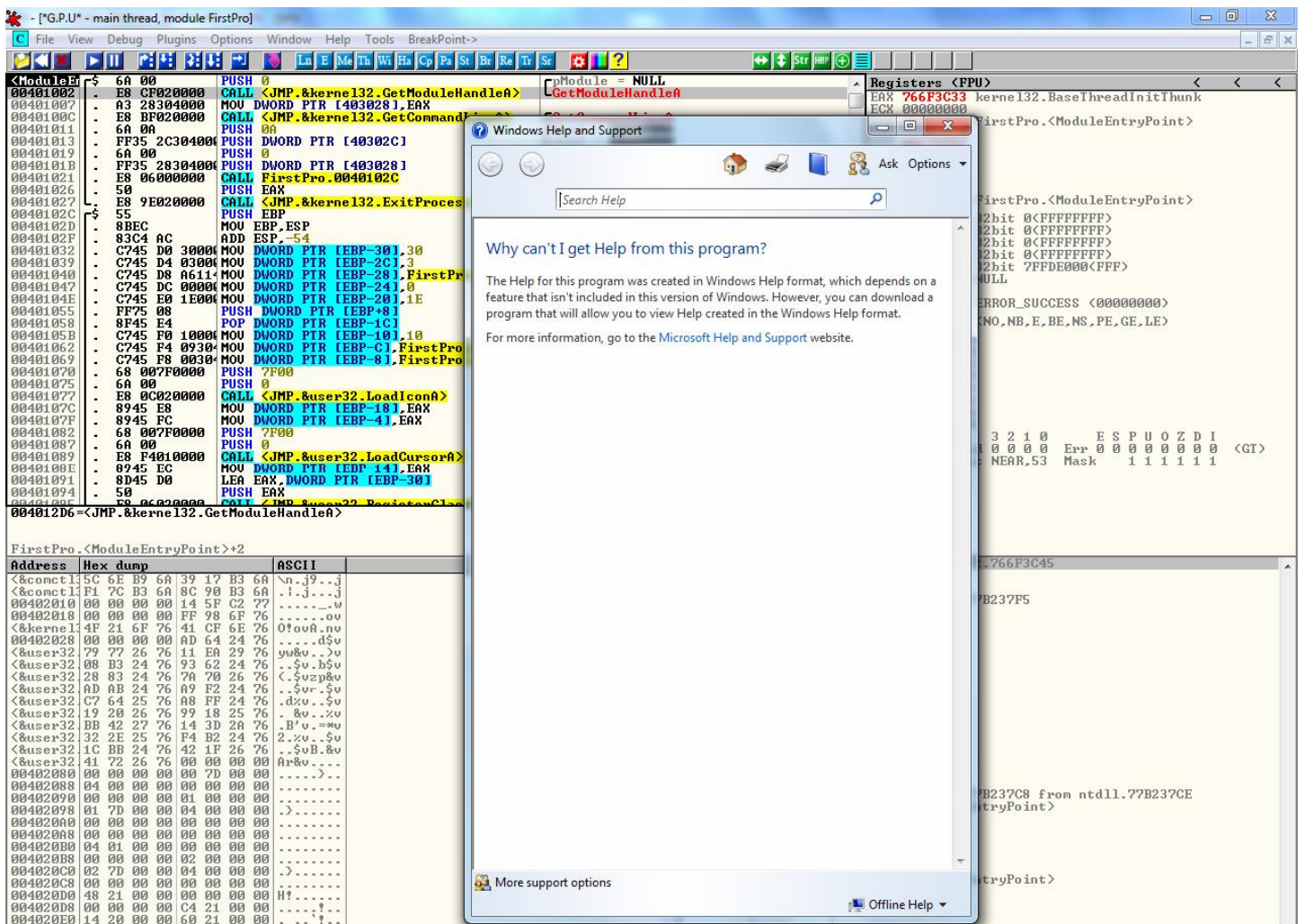
اول از همه بگذارید ببینیم این تابع چکاری انجام میدهد، در این آموزش فایل WIN32.HLP ضمیمه شده، این فایل قابلیت نمایش نمونه ی عملکرد تمامی توابع ویندوز را که تا کنون از طرف مایکروسافت معرفی و سند دار هستند را نشان میدهد، برای اینکار مانند تصویر زیر عمل کنید و فایل WIN32.HLP را انتخاب کنید:



بعد از انتخاب اگر بروی هر تابع راست کلیک کنید (آیتم Help on symbolic name به منو اضافه شده)



با انتخاب آن اطلاعات مربوط به تابع نمایش داده میشود، اگر از ویندوز ۷ به بالا استفاده میکنید ممکن است با صفحه ی :



رو برو شوید برای حل این مشکل به آدرس اینترنتی زیر بروید :

<https://support.microsoft.com/en-us/kb/917607>

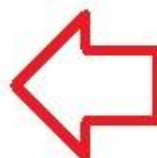
با توجه به نسخه ی سیستم عامل مورد استفاده فایل مذکور را دانلود کرده و نصب کنید:

When this problem occurs, you receive one of the following error messages.

 **Expand to see the full error message**

To resolve this problem, download and install the Windows Help program (WinHlp32.exe) for your version of Windows.

-  [WinHlp32.exe for Windows 8.1](#)
-  [WinHlp32.exe for Windows 8](#)
-  [WinHlp32.exe for Windows 7](#)
-  [WinHlp32.exe for Windows Server 2008](#)
-  [WinHlp32.exe for Windows Server 2008 R2](#)
-  [WinHlp32.exe for Windows Vista](#)



Validation Required

For more information about the validation process [click here](#)

File Name:	Size:
Windows6.1-KB917607-x64.msu	702 KB
Windows6.1-KB917607-x86.msu	688 KB

Continue



Windows Help program (WinHlp32.exe) for Windows 7

Select Language:

English



Download

808x187 25kb JPEG

نکته: لازم به ذکر است تمامی فایل های HLP از جمله Help مربوط به Olly یا هر HLP دیگر اجرا خواهند شد، بعد از نصب ollydbg را ریستارت کنید.

The screenshot shows a debugger window with assembly code on the left and a Win32 Programmer's Reference window on the right. The assembly code is for a module named "FirstPro" and shows instructions like PUSH, CALL, MOV, and JMP. The Win32 Programmer's Reference window displays the documentation for the **GetModuleHandle** function, including its parameters, return values, remarks, and see also section.

GetModuleHandle Quick Info Overview Group

The **GetModuleHandle** function returns a module handle for the specified module if the file has been mapped into the address space of the calling process.

HMODULE GetModuleHandle(
LPCSTR // address of module name to return handle
lpModuleName for
);

Parameters
lpModuleName
 Points to a null-terminated string that names a Win32 module (either a .DLL or .EXE file). If the filename extension is omitted, the default library extension .DLL is appended. The filename string can include a trailing point character (.) to indicate that the module name has no extension. The string does not have to specify a path. The name is compared (case independently) to the names of modules currently mapped into the address space of the calling process.
 If this parameter is NULL, **GetModuleHandle** returns a handle of the file used to create the calling process.

Return Values
 If the function succeeds, the return value is a handle to the specified module.
 If the function fails, the return value is NULL. To get extended error information, call [GetLastError](#).

Remarks
 The returned handle is not global, inheritable, or duplicative, and it cannot be used by another process.
 The handles returned by **GetModuleHandle** and **LoadLibrary** can be used in the same functions — for example, [GetProcAddress](#), [FreeLibrary](#), or [LoadResource](#). The difference between the two functions involves the reference count. **LoadLibrary** maps the module into the address space of the calling process, if necessary, and increments the module's reference count, if it is already mapped. **GetModuleHandle**, however, returns the handle of a mapped module without incrementing its reference count.
 Note that the reference count is used in **FreeLibrary** to determine whether to unmap the function from the address space of the process. For this reason, use care when using a handle returned by **GetModuleHandle** in a call to **FreeLibrary** because doing so can cause a dynamic-link library (DLL) module to be unmapped prematurely.
 This function must also be used carefully in a multithreaded application. There is no guarantee that the module handle remains valid between the time this function returns the handle and the time it is used by another function. For example, a thread might retrieve a module handle by calling **GetModuleHandle**. Before the thread uses the handle in another function, a second thread could free the module and the system could load another module, giving it the same handle as the module that was recently freed. The first thread would then be left with a module handle that refers to a module different than the one intended.

See Also
[FreeLibrary](#), [GetModuleFileName](#), [GetProcAddress](#), [LoadLibrary](#), [LoadResource](#)

همانطور که میبینید این تابع هندل (شناسه) را از ویندوز میگیرد. هر آبجکت یک شناسه ی دسترسی دارد و برای ویندوز بسیار مهم

هست ،اساسا با همین شناسه است که ویندوز میفهمد شما به کدام آبجکت اشاره میکنید

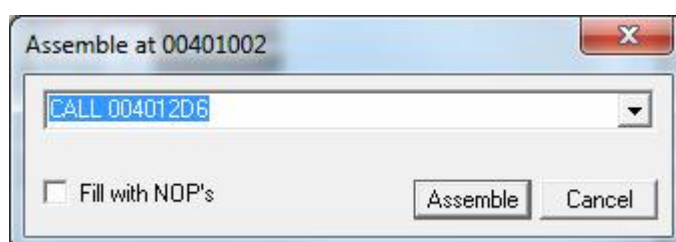
پنجره ی HLP را ببندید، و به فراخوانی (خط دوم) نگاه کنید که به کدام آدرس اشاره دارد، که به چند روش قابل فهمیدن است :

۱- بروی خط مذکور Enter کرده

۲- از قسمت Info قابل مشاهده است

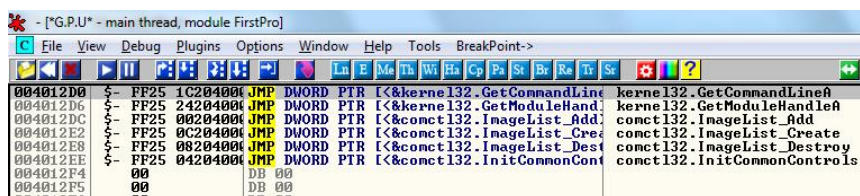
۳- با فشردن دکمه ی Space

۴- با فشردن کلید ترکیبی Ctrl+G و وارد کردن آدرس



نکته: کلید Space دوکاره است ،هم برای ویرایش کد و هم برای دریافت آدرس فراخوانی ها

بسیار خوب ، با فشردن کلید Enter بروی خط دوم “CALL GetModuleHandleA” به آدرس فراخوانی میرویم :



جدول آدرس پرش ها

همانطور که در تصویر بالا میبینید این یک پرش به سمت تابع GetModuleHandleA واقع در kernel32.dll است. با فشردن Enter بروی آن به ابتدای این تابع ارجاع داده خواهیم شد. چیزی که درباره Dll ها باید بدانید این است که همیشه در همان آدرس مشخص شده که در ImageBase هست ، لود نمیشوند!! ویندوز لودر مسئول بارگذاری و همچنین فراهم کردن تمام توابع و dll های مورد نیاز برنامه است، همچنین میتواند آدرس لود dll ها و متی فوده برنامه را تخمین دهد!! فرض کنید که dll ای داریم که میخواهد در آدرس 80000000 بارگذاری شود و از قضا dll دیگری داریم که میخواهد در همان آدرس لود شود!! تکلیف چیست؟ از آنجایی که دو dll نمیتوانند در یک آدرس مشابه بارگذاری شوند. ویندوز لودر باید یکی از آنها را به آدرس دیگری ببرد. که در اکثر مواقع اتفاق می افتد که به آن relocation گفته میشود.

وقتی ما برنامه ای مینویسیم و به فرض تابع GetModuleHandleA را درون آن فراخوانی میکنیم ، کمپایلر آدرس دستیابی به آن را نمیداند ، مال وقتی برنامه اجرا شود ، و اگر ویندوز لودر آن در آدرس 80000E300 بارگذاری کند چه اتفاقی می افتد؟ مشخص است خطا رخ میدهد ، فایل های اجرایی این مشکل را با به کارگیری جدول پرش حل کرده اند، این یعنی هنگام کمپایل برنامه هرگونه ارجاع مثلا به تابع GetModuleHandleA علامت گذاری میشود و به ممض فراخوانی ، آدرس مورد نظر جایگزین میشود و در نهایت به آدرس مناسب تبدیل میشوند. در واقع همه ی dll ها از این متد استفاده میکنند .

[*G.P.U* - main thread, module FirstPro]			
File View Debug Plugins Options Window Help Tools BreakPoint->			
Ln E Me Th Wi Ha Cp Pa St Br Re Tr Sr			
00401247	C9	LEAVE	
00401248	C2 1000	RET 10	
0040124B	CC	INT3	
0040124C	FF25 14204000	JMP DWORD PTR	[<&gdi32.DeleteObject>]
00401252	FF25 74204000	JMP DWORD PTR	[<&user32.CreateDialogParamA>]
00401258	FF25 70204000	JMP DWORD PTR	[<&user32.DefWindowProcA>]
0040125E	FF25 6C204000	JMP DWORD PTR	[<&user32.DestroyWindow>]
00401264	FF25 68204000	JMP DWORD PTR	[<&user32.DispatchMessageA>]
0040126A	FF25 60204000	JMP DWORD PTR	[<&user32.GetDlgItem>]
00401270	FF25 64204000	JMP DWORD PTR	[<&user32.GetDlgItemTextA>]
00401276	FF25 5C204000	JMP DWORD PTR	[<&user32.GetMessageA>]
0040127C	FF25 58204000	JMP DWORD PTR	[<&user32.IsDialogMessageA>]
00401282	FF25 40204000	JMP DWORD PTR	[<&user32.LoadCursorA>]
00401288	FF25 2C204000	JMP DWORD PTR	[<&user32.LoadIconA>]
0040128E	FF25 30204000	JMP DWORD PTR	[<&user32.LoadImageA>]
00401294	FF25 34204000	JMP DWORD PTR	[<&user32.MessageBoxA>]
0040129A	FF25 38204000	JMP DWORD PTR	[<&user32.PostQuitMessage>]
004012A0	FF25 3C204000	JMP DWORD PTR	[<&user32.RegisterClassExA>]
004012A6	FF25 78204000	JMP DWORD PTR	[<&user32.SendDlgItemMessageA>]
004012AC	FF25 44204000	JMP DWORD PTR	[<&user32.SetDlgItemTextA>]
004012B2	FF25 48204000	JMP DWORD PTR	[<&user32.SetFocus>]
004012B8	FF25 4C204000	JMP DWORD PTR	[<&user32.ShowWindow>]
004012BE	FF25 50204000	JMP DWORD PTR	[<&user32.TranslateMessage>]
004012C4	FF25 54204000	JMP DWORD PTR	[<&user32.UpdateWindow>]
004012CA	FF25 20204000	JMP DWORD PTR	[<&kernel32.ExitProcess>]
004012D0	FF25 1C204000	JMP DWORD PTR	[<&kernel32.GetCommandLineA>]
004012D6	FF25 24204000	JMP DWORD PTR	[<&kernel32.GetModuleHandleA>]
004012DC	FF25 00204000	JMP DWORD PTR	[<&comctl32.ImageList_Add>]
004012E2	FF25 0C204000	JMP DWORD PTR	[<&comctl32.ImageList_Create>]
004012E8	FF25 08204000	JMP DWORD PTR	[<&comctl32.ImageList_Destroy>]
004012EE	FF25 04204000	JMP DWORD PTR	[<&comctl32.InitCommonControls>]
004012F4	00	DB 00	

شاید هنوز متوجه نشده باشید، بگذارید مثالی بزنم، در این مثال ما از kernel32.dll تابع فرضی ShowMarioBrosPicture را فراخوانی میکنیم :

```
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.");
    }
}
```

بعد از کمپایل تکه کد بالا فراخوانی تابع ها با آدرس واقعی جایگزین میشوند :

```
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 retm
```

همینطور Jump Table ما برای تابع ShowMarioBrosPicture اینگونه خواهد بود :

```
402000 JMP XXXXXXXX
```

وقتی ویندوز لودر برنامه را بارگذاری میکند به سراغ Jump Table رفته و آن را پر میکند، اما هنوز Jump Table هیچ آدرس واقعی ندارد!! و بعد Dll را در فضای برنامه ما لود میکند، وقتی لود کامل شد، سپس آدرس تابع ما را (مثلاً: ShowMarioBrosPicture) پیدا میکند و به Jump Table رجوع میکند و آدرس را بجای "XXXXXXXX" مینویسد، فرض کنیدکه آدرس 77CE550A را نوشته: 402000 JMP 77CE550A.

و در نهایت که Jump Table اینگونه خواهد بود:

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

بیاید دوباره به برنامه خودمان برگردیم:

```

00401247  C9          LEAVE
00401248  C2 1000     RET 10
0040124B  CC          INT3
0040124C  $- FF25 14204000 JMP DWORD PTR [&gdi32.DeleteObject] gdi32.DeleteObject
00401252  $- FF25 74204000 JMP DWORD PTR [&user32.CreateDialogParamA] user32.CreateDialogParamA
00401258  $- FF25 70204000 JMP DWORD PTR [&user32.DefWindowProcA] user32.DefWindowProcA
0040125E  $- FF25 6C204000 JMP DWORD PTR [&user32.DestroyWindow] user32.DestroyWindow
00401264  $- FF25 68204000 JMP DWORD PTR [&user32.DispatchMessageA] user32.DispatchMessageA
0040126A  $- FF25 60204000 JMP DWORD PTR [&user32.GetDlgItem] user32.GetDlgItem
00401270  $- FF25 64204000 JMP DWORD PTR [&user32.GetDlgItemTextA] user32.GetDlgItemTextA
00401276  $- FF25 5C204000 JMP DWORD PTR [&user32.GetMessageA] user32.GetMessageA
0040127C  $- FF25 58204000 JMP DWORD PTR [&user32.IsDialogMessageA] user32.IsDialogMessageA
00401282  $- FF25 40204000 JMP DWORD PTR [&user32.LoadCursorA] user32.LoadCursorA
00401288  $- FF25 2C204000 JMP DWORD PTR [&user32.LoadIconA] user32.LoadIconA
0040128E  $- FF25 30204000 JMP DWORD PTR [&user32.LoadImageA] user32.LoadImageA
00401294  $- FF25 34204000 JMP DWORD PTR [&user32.MessageBoxA] user32.MessageBoxA
0040129A  $- FF25 38204000 JMP DWORD PTR [&user32.PostQuitMessage] user32.PostQuitMessage
004012A0  $- FF25 3C204000 JMP DWORD PTR [&user32.RegisterClassExA] user32.RegisterClassExA
004012A6  $- FF25 78204000 JMP DWORD PTR [&user32.SendDlgItemMessageA] user32.SendDlgItemMessageA
004012AC  $- FF25 44204000 JMP DWORD PTR [&user32.SetDlgItemTextA] user32.SetDlgItemTextA
004012B2  $- FF25 48204000 JMP DWORD PTR [&user32.SetFocus] user32.SetFocus
004012B8  $- FF25 4C204000 JMP DWORD PTR [&user32.ShowWindow] user32.ShowWindow
004012BE  $- FF25 50204000 JMP DWORD PTR [&user32.TranslateMessage] user32.TranslateMessage
004012C4  $- FF25 54204000 JMP DWORD PTR [&user32.UpdateWindow] user32.UpdateWindow
004012CA  $- FF25 20204000 JMP DWORD PTR [&kernel32.ExitProcess] kernel32.ExitProcess
004012D0  $- FF25 1C204000 JMP DWORD PTR [&kernel32.GetCommandLineA] kernel32.GetCommandLineA
004012D6  $- FF25 24204000 JMP DWORD PTR [&kernel32.GetModuleHandleA] kernel32.GetModuleHandleA
004012DC  $- FF25 00204000 JMP DWORD PTR [&comctl32.ImageList_Add] comctl32.ImageList_Add
004012E2  $- FF25 0C204000 JMP DWORD PTR [&comctl32.ImageList_Create] comctl32.ImageList_Create
004012E8  $- FF25 08204000 JMP DWORD PTR [&comctl32.ImageList_Destroy] comctl32.ImageList_Destroy
004012EE  $- FF25 04204000 JMP DWORD PTR [&comctl32.InitCommonControls] comctl32.InitCommonControls
004012F4  00         DB 00
  
```

همانطور که گفتیم، وقتی برنامه ای را مینویسیم (هنوز آن را اجرا نکرده ایم) قاعدتاً کمپایلر نمیداند که توابع به کار رفته درون این برنامه در چه آدرس هستند، حال برنامه را درون ollydbg باز کنید و وقتی به آدرس 401002 رسیدید کلید F7 را بفشارید تا به درون این call برویم :

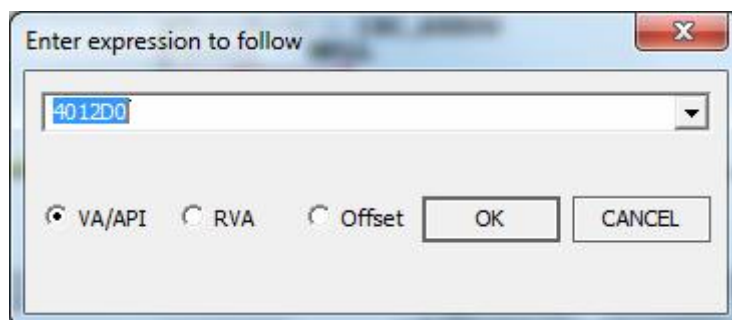
حالا یک بار دیگر کلید F7 را بفشارید تا به آدرس واقعی تابع (GetModuleHandleA) در آدرس (7780B741) منتقل شویم :

خب اگر به تایتل پنجره ی CPU نگاه کنید میبینید که وارد ماژول Kernel32.dll شده ایم :

دوباره برنامه را ری استارت کنید و اینبار از روی تابع (GetModuleHandleA) با کلید F8 عبور کنید و بر روی فضا توقف کنید

Space را بفشارید ما آدرس 4012D0 را میبینیم

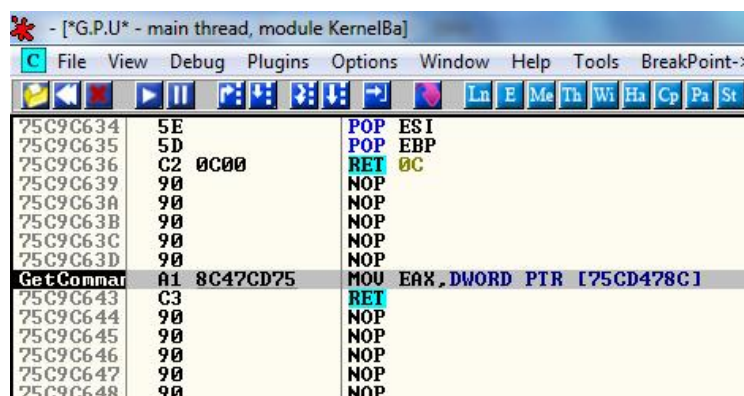
به صورت دستی می‌فرواهیم وارد آدرس بالا شویم (برای آشنایی بهتر با OllyDBG)، کلید های Ctrl+G را بفشارید یا آیکن  کلیک کنید. و آدرس را وارد و کلید Ok را بفشارید:



ما بروی تابع GetCommandLineA قرار داریم :

004012CA	.-	FF25	20204000	JMP	DWORD PTR	[&kernel32.ExitProcess]	kernel32.ExitProcess
004012D0	\$.-	FF25	1C204000	JMP	DWORD PTR	[&kernel32.GetCommandLineA]	kernel32.GetCommandLineA
004012D6	\$.-	FF25	24204000	JMP	DWORD PTR	[&kernel32.GetModuleHandleA]	kernel32.GetModuleHandleA
004012DC	\$.-	FF25	00204000	JMP	DWORD PTR	[&kernel32.ImageList_Add]	kernel32.ImageList_Add

و با کلیک F7 به ابتدای تابع "GetCommandLineA" در آدرس (7C812FBD) ارجاع داده می‌شویم :



ارجاع به داخل و بیرون Dll

گاهی اوقات شما در حال شکستن حفاظت برنامه ای هستید که ناگهان درون Dll های سیستمی غوطه ور می‌شوید و نمی‌دانید چگونه به برنامه ی اصلی بازگردید برای این کار من دو روش را پیشنهاد میکنم ، برای شروع ابتدا برنامه را ری استارت کنید و با کلید F7 وارد jump Table و سپس وارد تابع "GetCommandLineA" در kernel32.dll شوید حال با کلید Ctrl+F9 یا "Execute till user code" می‌توانید به دستورالعمل بعد از **CALL <JMP.&kernel32.GetModuleHandleA>** یعنی **MOV DWORD PTR** بازگردید **EAX, [403028]**



این دستور یعنی هر جا که هستی دستورات را اجرا کن تا به کد برنامه ی ما برسی و آنجا توقف کن. با اینکار میتوان روند را ادامه داد، روش دیگر استفاده از Memory Map است :

با فشردن کلید Alt+M وارد پنجره ی مذکور می‌شویم و سپس بروی سکشن کد که در اینجا آدرس 401000 است یک نقطه توقف معمولی یا حافظه ای گذاشته :

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00010000	00010000				Map	RW	RW	
00020000	00001000				Priv	RW	RW	
00120000	00001000				Priv	RW	Guar	
0012E000	00002000			stack of na	Priv	RW	Guar	
00130000	00004000				Map	R	R	
00140000	00001000				Priv	RW	RW	
00150000	00007000				Map	R	R	\Device\HarddiskVolume3\Windows\System32
001C0000	00001000				Priv	RW	RW	
001E0000	00012000				Priv	RW	RW	
002E0000	00006000				Map	R	R	
003A0000	00003000				Map	R	R	
00400000	00001000	FirstPro		PE header	Imag	R	RWE	
00401000	00001000	FirstPro	.text	SFX,code	Imag	R	RWE	
00402000	00001000	FirstPro	.rdata	data,imports	Imag	R	RWE	
00403000	00001000	FirstPro	.data		Imag	R	RWE	
00404000	00001000	FirstPro	.rsrc	resources	Imag	R	RWE	
00510000	00003000				Priv	RW	RW	
00520000	00101000				Map	R	R	
00630000	0000F000				Map	R	R	
01340000	00003000				Priv	RW	RW	
6AB30000	00001000	comctl32		PE header	Imag	R	RWE	
6AB31000	00075000	comctl32	.text	SFX,code,im	Imag	R	RWE	
6AB86000	00003000	comctl32	.data	data	Imag	R	RWE	
6AB89000	00007000	comctl32	.rsrc	resources	Imag	R	RWE	
6AB80000	00004000	comctl32	.reloc		Imag	R	RWE	
75C90000	00001000	KernelBa		PE header	Imag	R	RWE	
75C91000	000043000	KernelBa	.text	SFX,code,im	Imag	R	RWE	

سپس برنامه را با کلید F9 اجرا کنید، همانطور که میبینید برنامه در آدرس 401000 متوقف میشود، حال اگر نقطه ی توقف شما از نوع حافظه ایی است آن را کلا بردارید چرا که با هر بار اجرا برنامه در آن سکشن متوقف خواهد شد ☺

اطلاعات بیشتر در مورد Stack

واقعاً Stack در مهندسی معکوس بسیار مهم است، و اگر درمورد آن آگاهی نداشته باشید مطمئن باشید هیچ وقت یک کرکر واقعی نخواهید شد. سعی میکنم بسیار ساده و روان توضیح دهم

بعد از اینکه برنامه را دوباره ری استارت کردید به پنجره ی ثبات ها نگاه کنید، ثبات ESP را نگاه کنید که قرمز شده است، این ثبات به بالای استک اشاره میکند، و مقدار آن 12FF8C است (در کامپیوتر شما ممکن است آدرس فرق کند). حالا به پنجره ی پائین استک نگاه کنید، آدرس 12FF8C در ثبات ESP با آدرس پنجره ی استک یکی است:

The screenshot shows a debugger window with the 'Registers <FPU>' pane on the right. The ESP register is highlighted with a blue circle and shows the value 0012FF8C. Below the registers, the 'Memory' pane shows the stack contents starting at address 0012FF8C. The first line of memory is 766F3C45, which is the return address to kernel32.766F3C45. The second line is 7FFDF000, which is the return address to ntdll.77B237F5. The third line is 77B237F5, which is the return address to ntdll.77B237F5. The fourth line is 7FFDF000, which is the return address to ntdll.77B237F5. The fifth line is 76CC5ED0, which is the return address to ntdll.77B237F5. The sixth line is 00000000, which is the return address to ntdll.77B237F5. The seventh line is 00000000, which is the return address to ntdll.77B237F5. The eighth line is 7FFDF000, which is the return address to ntdll.77B237F5. The ninth line is 00000000, which is the return address to ntdll.77B237F5. The tenth line is 00000000, which is the return address to ntdll.77B237F5.

هالا یک بار کلید F7 یا F8 را بفشارید، و خوب به پنجره ی استک نگاه کنید که مقدار صفر روی آن ریخته میشود:

```

0012FF88 00000000 LpModule = NULL
0012FF8C 766F3C45 RETURN to kernel32.766F3C45
0012FF90 7FFDF000
0012FF94 0012FFD4
0012FF98 77B237F5 RETURN to ntdll.77B237F5
0012FF9C 7FFDF000
0012FFA0 76CC5ED0
0012FFA4 00000000
0012FFA8 00000000
0012FFAC 7FFDF000
0012FFB0 00000000
0012FFB4 00000000
0012FFB8 00000000
0012FFBC 0012FFA0
0012FFC0 00000000
0012FFC4 FFFFFFFF End of SEH chain
0012FFC8 77ADE0ED SE handler
0012FFCC 016FB37C
0012FFD0 00000000
0012FFD4 0012FFEC
0012FFD8 77B237C8 RETURN to ntdll.77B237C8 from ntdll.77B237CE
0012FFDC 00401000 FirstPro.<ModuleEntryPoint>
0012FFE0 7FFDF000
0012FFE4 00000000
0012FFE8 00000000
0012FFEC 00000000
0012FFF0 00000000
0012FFF4 00401000 FirstPro.<ModuleEntryPoint>
0012FFF8 7FFDF000
0012FFFF 00000000

```

حالا به پنجره ی ثبات ها و به ثبات ESP نگاه کنید :

The screenshot shows the Windows XP Task Manager Performance tab. The 'Processors' section is expanded, displaying a list of processors and their usage percentages. The processors are CPU0, CPU1, and CPU2, all showing 100% usage. The 'System Idle Time' is 0%.

Processor	Usage
CPU0	100%
CPU1	100%
CPU2	100%
System Idle Time	0%

12FF88-12FF8C=4

میبینیم که ۴ بایت Push شده، حال یک بار F8 را بفشارید و از تابع `GetModuleHandleA` عبور کنید و سپس به پنجره استک نگاه کنید:

0012FF8C	766F3C45	RETURN to kernel32.766F3C45
0012FF90	7FFD9000	
0012FF94	0012FFD4	
0012FF98	77B237F5	RETURN to ntdll.77B237F5
0012FF9C	7FFD9000	
0012FFA0	76D04FFC	
0012FFA4	00000000	
0012FFA8	00000000	
0012FFAC	7FFD9000	
0012FFB0	00000000	
0012FFB4	00000000	
0012FFB8	00000000	
0012FFBC	0012FFA0	
0012FFC0	00000000	
0012FFC4	FFFFFFFF	End of SEH chain
0012FFC8	77ADE0ED	SE handler
0012FFCC	0179A256	
0012FFD0	00000000	
0012FFD4	0012FFEC	
0012FFD8	77B237C8	RETURN to ntdll.77B237C8 from ntdll.77B237CE
0012FFDC	00401000	FirstPro.<ModuleEntryPoint>
0012FFE0	7FFD9000	
0012FFE4	00000000	
0012FFE8	00000000	
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00401000	FirstPro.<ModuleEntryPoint>
0012FFF8	7FFD9000	
0012FFFC	00000000	

همانطور که میبینید صفر از پشته POP شد و پشته به سمت پائین آمد ، حال با کلید F8 پائین بیایید تا به اولین Push بر سید ، میبینید تا رسیدن به اولین Push هیچ تخریری روی استک صورت نمیگیرد که دلیل آن هم این است که میزی روی آن Push نشده

اکنون دستور العمل PUSH 0A اجرا شده و 0A در بالای پشته قرار داده شده، و ثبات ESP چهار بایت کم شد (هر موقع عملیات Push صورت میگیرد این ثبات کم میشود و هرگاه POP صورت میگیرد، افزایش پیدا میکند) دوباره کلید F8 را بفشارید، اگر به بالای پشته نگاه کنید میبینید که مقدار ۴ بایتی بروی پشته قرار دارد.

به پشته نگاه کنید، مقدار 00000000 در بالای پشته قرار دارد، چرا؟ به خط زیر نگاه کنید:

00401011	FF 35 2C 30 40 00	PUSH DWORD PTR [40302C]
00401013	6A 00	PUSH 0

معنی خط بالا این است که، مقدار ۴ بایتی را از آدرس 40302C را بگیر و درون پشته قرار بده. مقدار 40302C چیست؟ 00000000، بله، البته (به بالای پنجره استک نگاه کنید) برای متوجه شدن بروی همان دستور العمل (00401013) راست کلیک کنید Follow in “Memory Address” -> “Dump” :

Address	Hex dump	ASCII
0040302C	00 00 00 00 00 00 00 00	
00403034	00 00 00 00 00 00 00 00	
0040303C	00 00 00 00 00 00 00 00	
00403044	00 00 00 00 00 00 00 00	
0040304C	00 00 00 00 00 00 00 00	
00403054	00 00 00 00 00 00 00 00	
0040305C	00 00 00 00 00 00 00 00	
00403064	00 00 00 00 00 00 00 00	
0040306C	00 00 00 00 00 00 00 00	
00403074	00 00 00 00 00 00 00 00	
0040307C	00 00 00 00 00 00 00 00	
00403084	00 00 00 00 00 00 00 00	
0040308C	00 00 00 00 00 00 00 00	
00403094	00 00 00 00 00 00 00 00	
0040309C	00 00 00 00 00 00 00 00	
004030A4	00 00 00 00 00 00 00 00	
004030AC	00 00 00 00 00 00 00 00	
004030B4	00 00 00 00 00 00 00 00	
004030BC	00 00 00 00 00 00 00 00	
004030C4	00 00 00 00 00 00 00 00	
004030CC	00 00 00 00 00 00 00 00	
004030D4	00 00 00 00 00 00 00 00	
004030DC	00 00 00 00 00 00 00 00	
004030E4	00 00 00 00 00 00 00 00	
004030EC	00 00 00 00 00 00 00 00	
004030F4	00 00 00 00 00 00 00 00	
004030FC	00 00 00 00 00 00 00 00	
00403104	00 00 00 00 00 00 00 00	
0040310C	00 00 00 00 00 00 00 00	
00403114	00 00 00 00 00 00 00 00	
0040311C	00 00 00 00 00 00 00 00	
00403124	00 00 00 00 00 00 00 00	

همانطور که میبینید مقادیر مافظه از آدرس 40302C شروع شده، (بجز چهار بایت اول push شده) بقیه، فضایی است که کامپایلر برای متغیرهای ما در نظر گرفته ولی فعلا همگی با صفر مقدار دهی اولیه شده اند، اگر میفاهید ببینید، ابتدا برنامه را کامل اجرا کنید و متنی در برنامه بنویسید و سپس به آدرس مذکور نگاه کنید:

0040302C	00 00 00 00 57 57 57 2E	...WWW.
00403034	49 52 41 4E 4C 45 44 2E	IRANLED.COM
0040303C	43 4F 4D 00 00 00 00 00	
00403044	00 00 00 00 00 00 00 00	
0040304C	00 00 00 00 00 00 00 00	
00403054	00 00 00 00 00 00 00 00	
0040305C	00 00 00 00 00 00 00 00	
00403064	00 00 00 00 00 00 00 00	
0040306C	00 00 00 00 00 00 00 00	
00403074	00 00 00 00 00 00 00 00	
0040307C	00 00 00 00 00 00 00 00	
00403084	00 00 00 00 00 00 00 00	
0040308C	00 00 00 00 00 00 00 00	

فیلی فب، مجددا کلید F8 را بفشارید تا به Push بعدی رسیم که مقدار 0 را درون استک میریزد:

00401019	6A 00	PUSH 0
----------	-------	--------

و خط بعدی:

0040101B	FF 35 28 30 40 00	PUSH DWORD PTR [403028]
0040101D	EB 0F 00 00 00 00	JMP EBX + 0000000F

میبینید که اشاره به آدرس 403028 دارد، بیایید به این آدرس برویم :

The screenshot shows a debugger window with the following components:

- Assembly List:** Displays instructions at various addresses. Address 00401021 is highlighted, showing a `CALL FirstPro.0040102C` instruction. Address 00401028 is also highlighted, showing a `PUSH EAX` instruction.
- Registers (FPU):** Shows the state of CPU registers. The `EAX` register is highlighted and contains the value `00400000`.
- Hex Dump:** Shows the memory contents at the specified address. The address `00401028` is highlighted, showing the value `00 00 40 00 00 00 00 00`.

همانطور که میبینید آدرس 403028 حاوی مقدار 400000 که با فشردن کلید F8 در پشته Push خواهد شد ، همه ی آگومانها کامل در پشته قرار گرفتند. حال کلید F7 را یکبار فشار دهید :

The screenshot shows a debugger window with the following components:

- Assembly List:** Displays instructions at various addresses. Address 00401021 is highlighted, showing a `CALL FirstPro.0040102C` instruction. Address 00401028 is also highlighted, showing a `PUSH EAX` instruction.
- Registers (FPU):** Shows the state of CPU registers. The `EAX` register is highlighted and contains the value `00400000`.
- Hex Dump:** Shows the memory contents at the specified address. The address `00401028` is highlighted, showing the value `00 00 40 00 00 00 00 00`.

درست است ، یک مقدار به پشته اضافه شد!! که در مقیقت این آدرس بازگشتی (401021) توسط دستور العمل Call درون پشته Push شده است و توسط دستور العمل RET بازایی یا POP میشود پس از فائمه ی برنامه به آدرس 401021 ارجاع داده میشود :

The screenshot shows a debugger window with the following components:

- Assembly List:** Displays instructions at various addresses. Address 0040117F is highlighted, showing a `MOV EAX, DWORD PTR [EBP-44]` instruction. Address 004011A2 is highlighted, showing a `LEAVE` instruction. Address 004011A3 is highlighted, showing a `RET 10` instruction. Address 004011A6 is highlighted, showing a `PUSH EBP` instruction.

10 مقابل دستور `ret` به معنای پاک کردن 10 بایت از بالای استک است ، برای بازگشتن به فط مورد نظر بروی ثبات EIP دبل کلیک کنید.