2009

[Cracking with OllyDbg]

Based on OllyDbg tuts of Ricardo Narvaja (CrackLatinos Team)

www.reaonline.net

kienmanowar

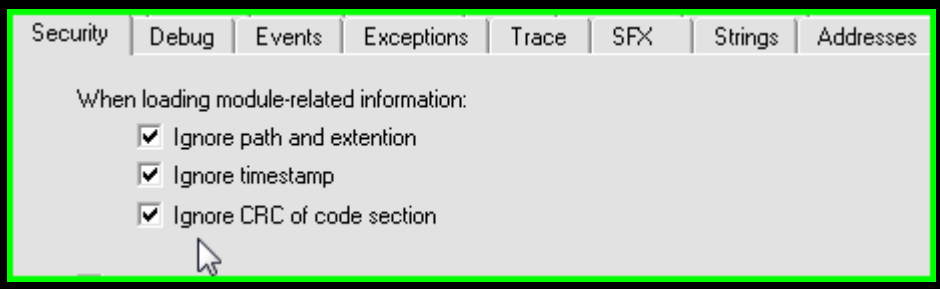
13/01/2010

Mục Lục

I. Giới thiệu chung

Chào các bạn, để tiếp tục với chủ đề Anti-OllyDbg, ở phần 20 này tôi sẽ tập trung vào giới thiệu cách các target phát hiện ra Olly thông qua việc kiểm tra tên của process. Trong tuần vừa qua, chắc các bạn đã đọc xong phần 19 - trình bày về cách Anti-Olly bằng API `IsDebuggerPresent` và những phương pháp để vượt qua cơ chế này. Cũng có thể tới thời điểm này, có nhiều bạn chưa giải được Crackme ở phần 18 để unlock phần 19, rất mong các bạn cố gắng tự lực đừng nên phụ thuộc vào lời giải hoặc các đáp án mà một số bạn đã public.

Trước khi đi vào phần chính của bài viết, chúng ta cần cấu hình lại Olly như sau (Ở đây tôi vẫn sử dụng Olly nguyên bản nhé) :



Theo như bác Ricardo giải thích thì mục đích là để giữ lại các BP với hàm API mà ta đã thiết lập khi chúng ta đóng hoặc restart lại Olly, đỡ phải mỗi lần load xong target là lại phải thiết lập lại BP. Bác nói thêm là bác cũng không hiểu rõ được cơ chế hoạt động bên trong của Olly, nhưng bác đã kiểm nghiệm nhiều lần rồi nên mới rút ra được cách này. Tôi cũng đã kiểm tra thử và thấy đúng như thế thật ☺. Sau khi thiết lập như trên xong ta đi vào phần chính của bài 20, N0w let's g0……☺

II. Phân tích và xử lý target

1. Phân tích DaXXoR - Decryptme

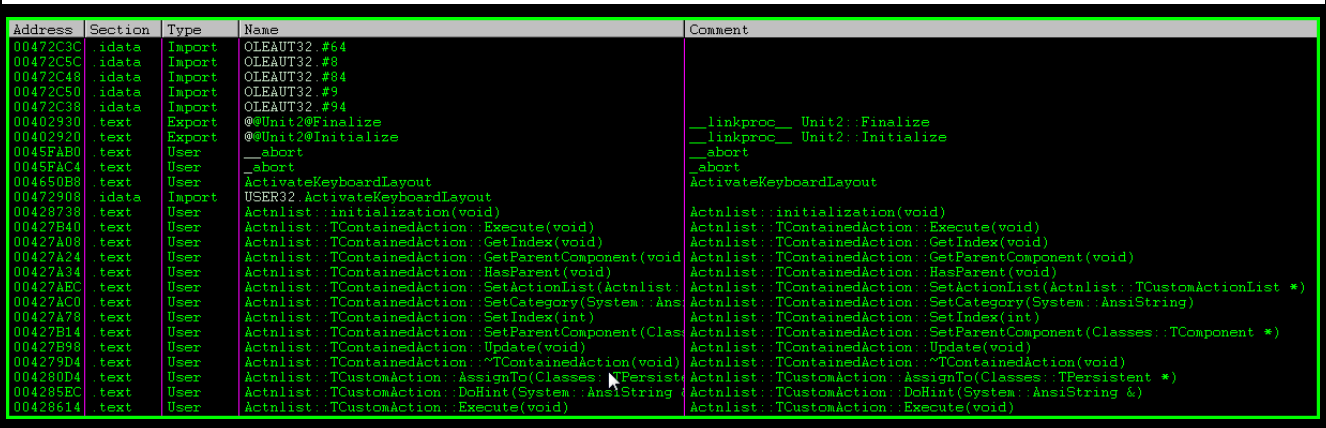
Ta đi tổng quan một chút, bình thường các bạn hay sử dụng Task Manager hay một chương trình nào đó để xem các Process đang chạy trên hệ thống của mình (Trên máy tôi sử dụng Process Explorer). Khi chúng ta đang chạy OllyDbg và xem danh sách các Process, ta sẽ thấy như sau :

Như các bạn thấy trên hình minh họa, danh sách các Process hiện ra rất rõ ràng. Điều này đồng nghĩa với việc sẽ có một cơ chế Anti-OllyDbg sử dụng cách thức nào đó liệt kê ra danh sách các Process đang hoạt động, sau đó so sánh tên của Process với OLLYDBG, nếu như phát hiện ra có Olly đang chạy thì bùm…coi như Olly đã hi sinh, tương tự như việc ta chọn Process và Kill nó ☺.

Trong bài viết này ta sẽ phân tích DaXXoR – Decryptme, để xem cơ chế Anti-Olly của nó như thế nào. Sau đó tìm cách vượt qua các cơ chế này. Để kiểm nghiệm lý thuyết vừa nói ở trên, ta chạy Olly trước sau đó chạy Crackme DaXXoR . Ngay khi ta chạy crackme thì Olly lập tức cũng bị terminate liền. Công nhận là ác thật, Olly bay luôn mà không kịp trăng trối điều gì ☹.

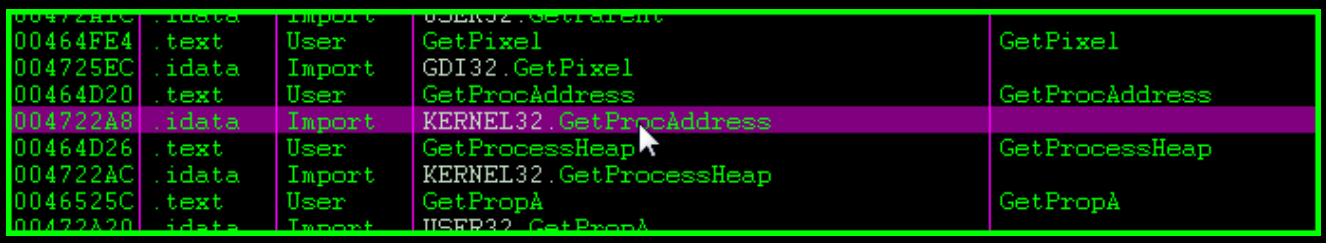
Ok, bắt đầu nghiên cứu … ta load target vào Olly :

Ta đang dừng lại ở EP của target, tiến hành tìm kiếm danh sách các hàm API :



Address	Section	Type	Name	Comment
00472C3C	.idata	Import	OLEAUT32.#64	
00472C5C	.idata	Import	OLEAUT32.#8	
00472C48	.idata	Import	OLEAUT32.#84	
00472C50	.idata	Import	OLEAUT32.#9	
00472C38	.idata	Import	OLEAUT32.#94	
00402930	.text	Export	@@Unit2@Finalize	__linkproc__ Unit2::Finalize
00402920	.text	Export	@@Unit2@Initialize	__linkproc__ Unit2::Initialize
0045FAB0	.text	User	__abort	__abort
0045FAC4	.text	User	_abort	_abort
004650B8	.text	User	ActivateKeyboardLayout	ActivateKeyboardLayout
00472908	.idata	Import	USER32.ActivateKeyboardLayout	
00428738	.text	User	Actnlist::initialization(void)	Actnlist::initialization(void)
00427B40	.text	User	Actnlist::TContainedAction::Execute(void)	Actnlist::TContainedAction::Execute(void)
00427A08	.text	User	Actnlist::TContainedAction::GetIndex(void)	Actnlist::TContainedAction::GetIndex(void)
00427A24	.text	User	Actnlist::TContainedAction::GetParentComponent(void	Actnlist::TContainedAction::GetParentComponent(void)
00427A34	.text	User	Actnlist::TContainedAction::HasParent(void)	Actnlist::TContainedAction::HasParent(void)
00427AEC	.text	User	Actnlist::TContainedAction::SetActionList(Actnlist:	Actnlist::TContainedAction::SetActionList(Actnlist::TCustomActionList *)
00427AC0	.text	User	Actnlist::TContainedAction::SetCategory(System::Ans	Actnlist::TContainedAction::SetCategory(System::AnsiString)
00427A78	.text	User	Actnlist::TContainedAction::SetIndex(int)	Actnlist::TContainedAction::SetIndex(int)
00427B14	.text	User	Actnlist::TContainedAction::SetParentComponent(Clas	Actnlist::TContainedAction::SetParentComponent(Classes::TComponent *)
00427B98	.text	User	Actnlist::TContainedAction::Update(void)	Actnlist::TContainedAction::Update(void)
004279D4	.text	User	Actnlist::TContainedAction::~TContainedAction(void)	Actnlist::TContainedAction::~TContainedAction(void)
004280D4	.text	User	Actnlist::TCustomAction::AssignTo(Classes::TPersist	Actnlist::TCustomAction::AssignTo(Classes::TPersistent *)
004285EC	.text	User	Actnlist::TCustomAction::DoHint(System::AnsiString &	Actnlist::TCustomAction::DoHint(System::AnsiString &)
00428614	.text	User	Actnlist::TCustomAction::Execute(void)	Actnlist::TCustomAction::Execute(void)

Chà một danh sách khá dài, làm sao biết hàm nào là hàm cần tìm, hàm nào được chương trình sử dụng trong việc Anti-Olly đây ☺. Như các bạn thấy, khi ta chạy crackme thì nó mới thực hiện cơ chế Anti-Debug của nó, như vậy có nghĩa là hàm API được sử dụng sẽ không được load từ đầu và đưa vào danh sách các hàm API như trên. *Tự hỏi, hàm không được nạp vào trong danh sách API thì làm sao mò ra nó ?* Rất may mắn là Windows cung cấp cho chúng ta một thủ thuật để tìm kiếm các APIs được load khi chúng ta thực thi chương trình, từ đó tìm ra hàm API quan trọng. Nếu các bạn là dân coder chuyên nghiệp hay là những người am tường về hệ thống Windows thì chắc rằng các bạn đã biết tôi nhắc tới API nào. Đó chính là hàm GetProcAddress! Giờ ta tìm thử trong Crackme này có sử dụng hàm này không nhé :



00464FE4	.text	User	GetPixel	GetPixel
004725EC	.idata	Import	GDI32.GetPixel	
00464D20	.text	User	GetProcAddress	GetProcAddress
004722A8	.idata	Import	KERNEL32.GetProcAddress	
00464D26	.text	User	GetProcessHeap	GetProcessHeap
004722AC	.idata	Import	KERNEL32.GetProcessHeap	
0046525C	.text	User	GetPropA	GetPropA

Ồ thật may mắn, crackme có sử dụng hàm này. Thông tin chi tiết về hàm này như sau :

The GetProcAddress function returns the address of the specified exported dynamic-link library (DLL) function.

```
FARPROC GetProcAddress(

    HMODULE hModule,  // handle to DLL module
    LPCSTR lpProcName      // name of function
   );
```

Parameters

hModule: Identifies the DLL module that contains the function. The LoadLibrary or GetModuleHandle function returns this handle.

lpProcName: Points to a null-terminated string containing the function name, or specifies the function's ordinal value. If this parameter is an ordinal value, it must be in the low-order word; the high-order word must be zero.

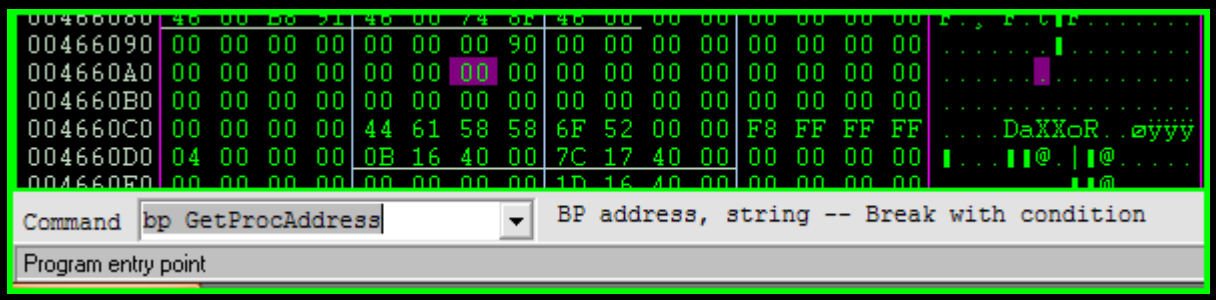
Return Values

If the function succeeds, the return value is the address of the DLL's exported function.If the function fails, the return value is NULL. To get extended error information, call GetLastError.

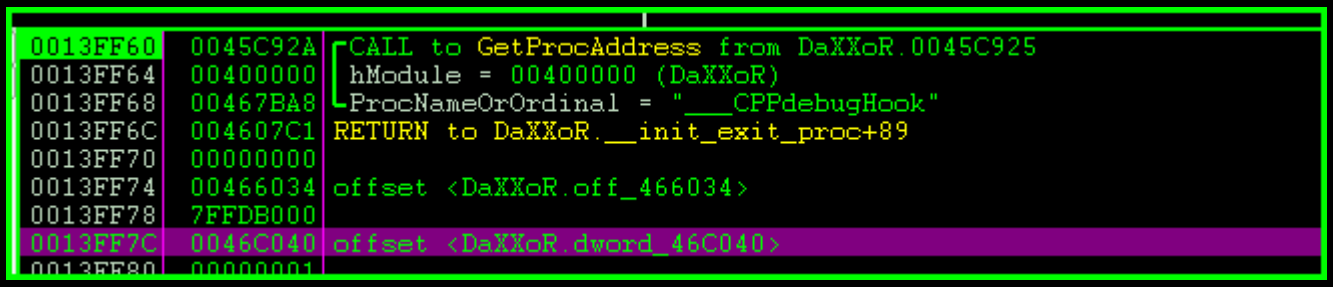
Remarks

The GetProcAddress function is used to retrieve addresses of exported functions in DLLs.

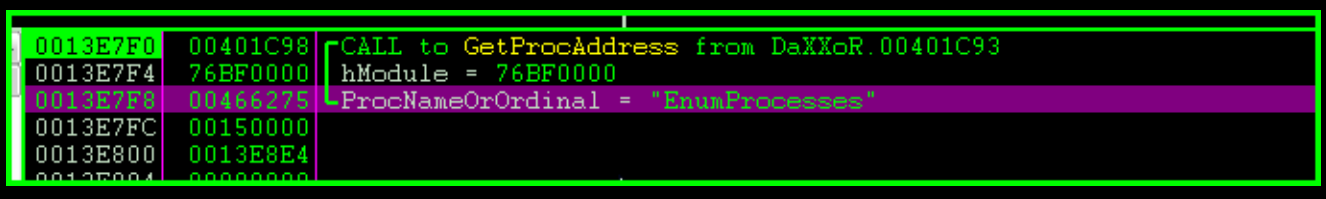
Tóm lại, nhiệm vụ của hàm GetProcAddress là tìm ra địa chỉ của một hàm trong file Dll. Thường được sử dụng khi chương trình cần load một hàm API mới mà không được liệt kê trong danh sách. Ta đặt BP tại hàm này :



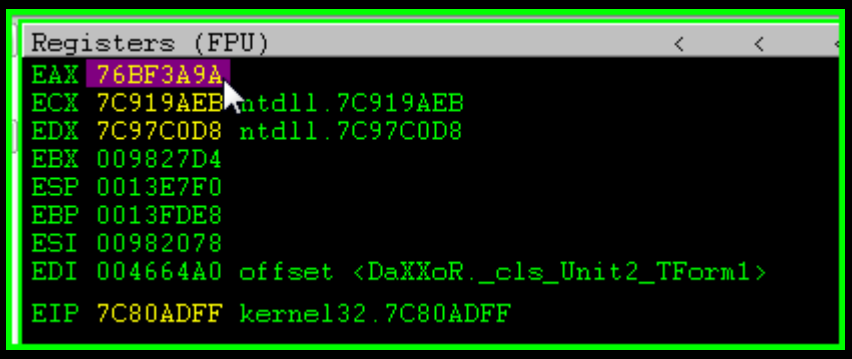
Sau khi thực hiện đặt BP xong, nhấn F9 để thực thi chương trình … Olly sẽ dừng lại :



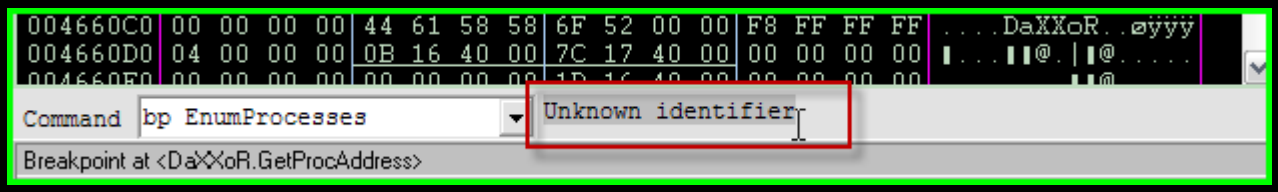
Quan sát trên cửa sổ Stack ta có được thông tin như trên. 0013FF68 00467BA8 \ProcNameOrOrdinal = "___CPPdebugHook" là tên của hàm truyền vào để GetProcAddress tìm ra địa chỉ của hàm đó. Sau khi thực hiện GetProcAddress xong thì kết quả trả về là địa chỉ hàm nằm ở thanh ghi EAX. Do đây chưa phải là hàm mà ta quan tâm nên tiếp tục nhấn F9, cho tới khi Olly dừng lại tại hàm sau trên cửa sổ Stack :



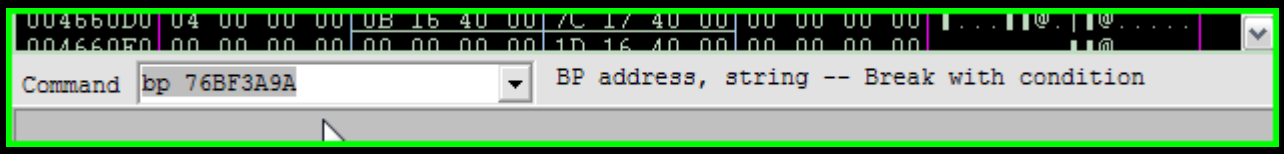
Tại sao ta lại quan tâm tới hàm này, đơn giản là vì tôi thấy tên của nó có dính tới Proccess nên đặt nghi ngờ. Nhấn **Ctrl + F9 (Execute till Return)** và quan sát giá trị của EAX ở cửa sổ Registers :

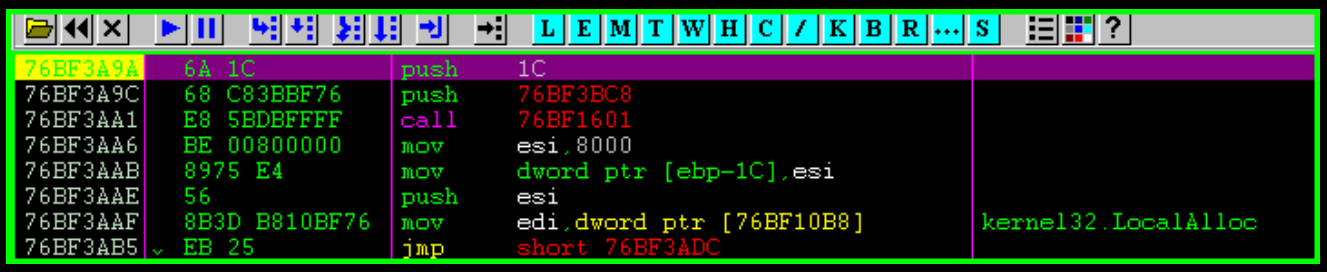


Thanh ghi EAX của tôi đang lưu địa chỉ của hàm EnumProcesses, cụ thể là 0x76BF3A9A. Giá trị này có thể khác trên máy của các bạn. Như đã nói ở trên, do hàm này không được liệt kê trong danh sách các hàm APIs, cho nên nếu ta thử đặt BP tại hàm này thì sẽ nhận được thông báo sau :

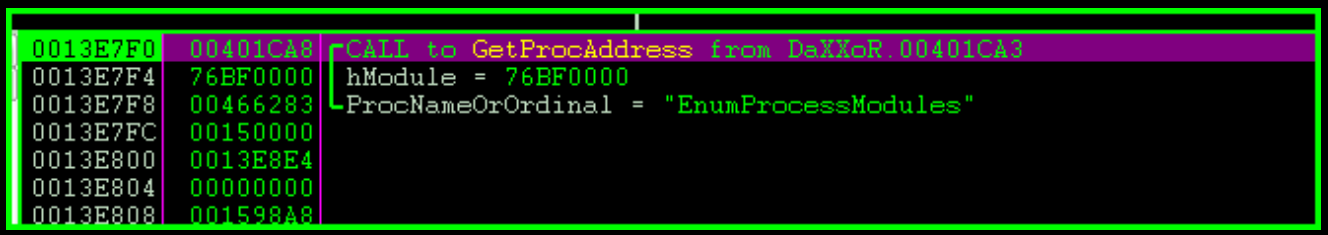


Nhưng giờ ta đã biết được địa chỉ của hàm này nên ta có thể đặt BP như sau :

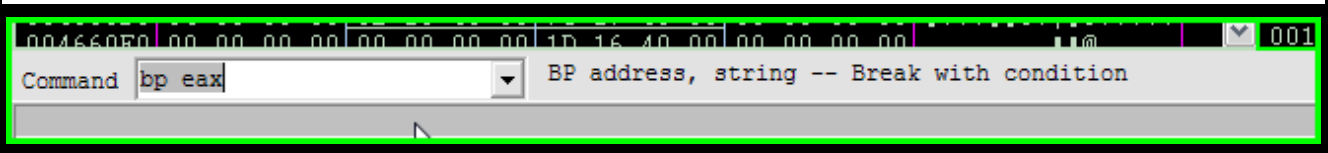


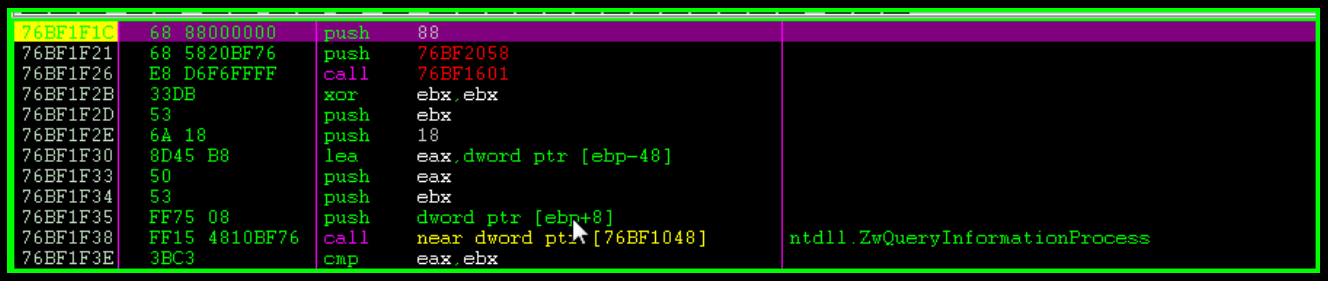


Đặt BP xong tiếp tục nhấn F9 để tìm kiếm thông tin về các hàm APIs khác :

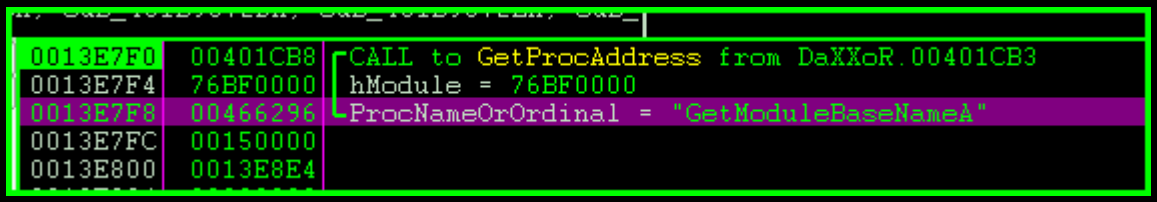


Chà lại một hàm nữa, ta thực hiện tương tự như trên và đặt BP :

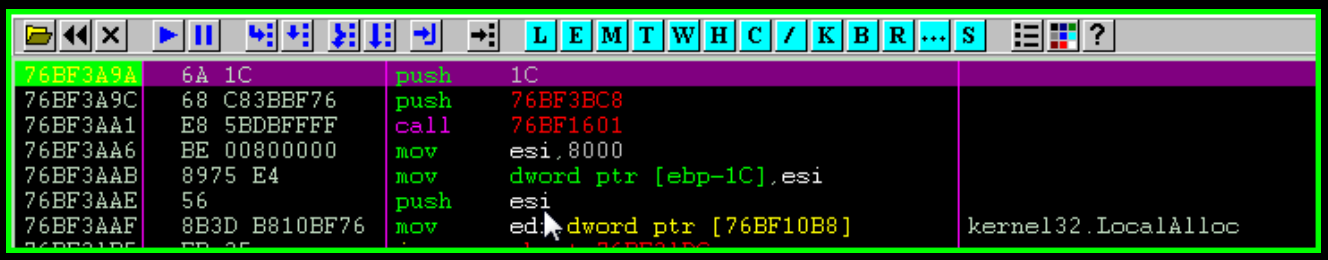




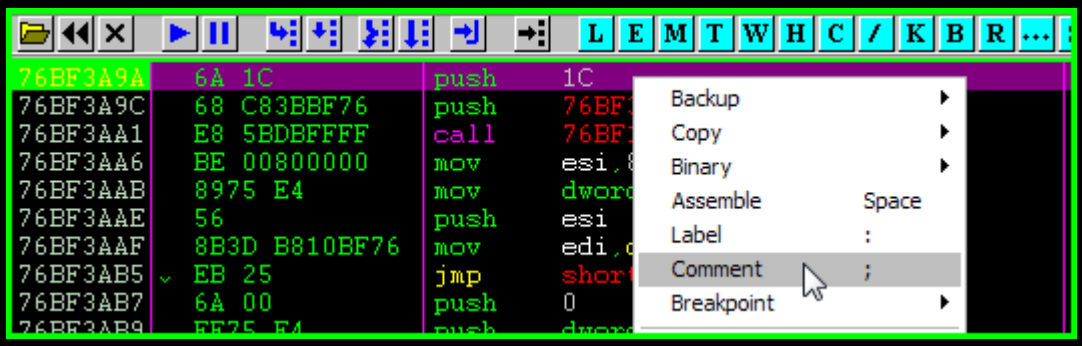
Tiếp tục nhấn F9 và Olly lại break :



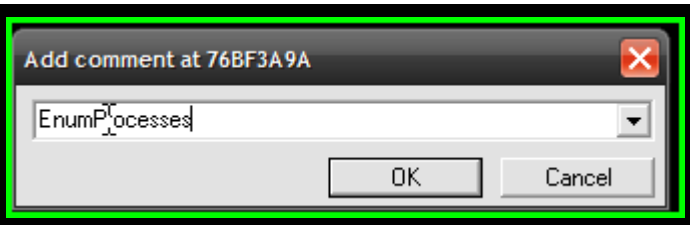
Đặt BP tương tư như những gì đã thực hiện. Sau đó nhấn F9, lúc này Olly sẽ break tại hàm EnumProcesses mà ta đã đặt BP :



Có quá nhiều BP mà ta đã đặt, cho nên để phân biệt là ta đang dừng lại tại BP nào chúng ta sẽ ghi chú tại BP đó như sau, chuột phải tại nơi cần comment và chọn :



Nhập thông tin và sau đó nhấn OK, ta sẽ có được kết quả như sau :



Thử tìm kiếm thông tin về hàm `EnumProcesses` trong file **win32.hlp**, ta nhận được là không có kết quả nào cho hàm này. Không lẽ hàm này thuộc dạng *Private*. Cách tốt nhất là sử dụng Google để tìm kiếm thông tin. Sau khi tìm kiếm tôi có được thông tin như sau :

EnumProcesses Function

Retrieves the process identifier for each process object in the system.

Syntax

C++

```
BOOL WINAPI EnumProcesses(
  __out  DWORD *pProcessIds,
  __in   DWORD cb,
  __out  DWORD *pBytesReturned
);
```

Parameters

pProcessIds [out]
A pointer to an array that receives the list of process identifiers.

cb [in]
The size of the *pProcessIds* array, in bytes.

pBytesReturned [out]
The number of bytes returned in the *pProcessIds* array.

Return Value

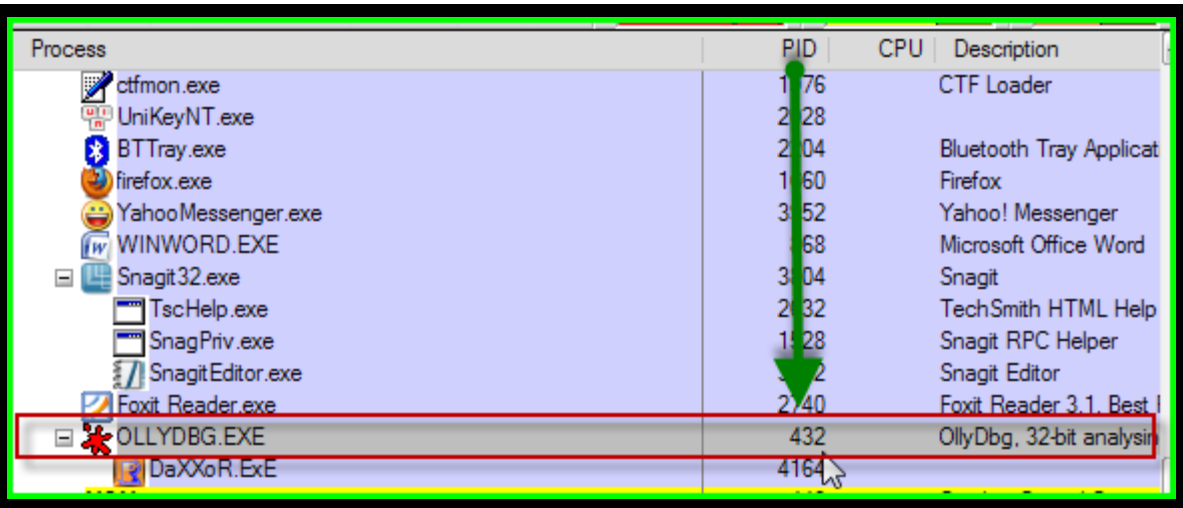
If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

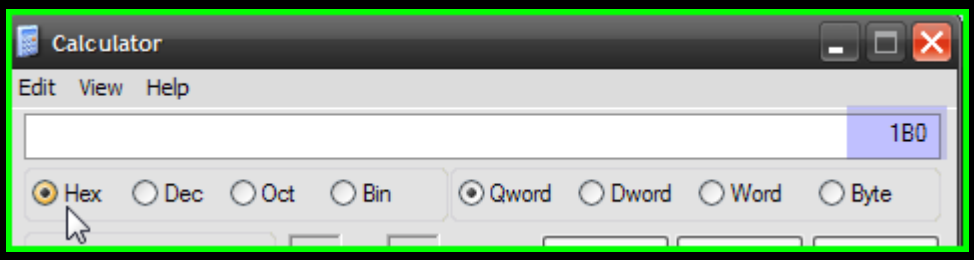
Remarks

It is a good idea to use a large array, because it is hard to predict how many processes there will be at the time you call **EnumProcesses**.

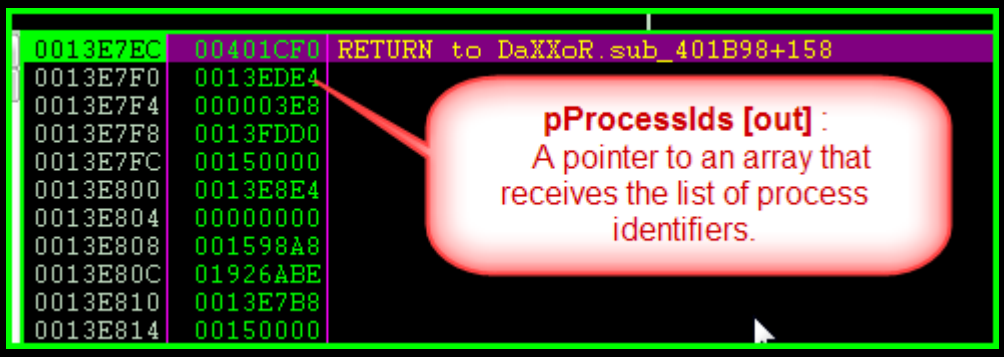
Theo kết quả tìm kiếm về hàm này cho thấy nó được sử dụng để lấy thông tin về *Process Identifier (PID)* của các chương trình đang chạy trên hệ thống. Mỗi chương trình khi thực thi sẽ được nhận biết bởi một con số, con số này sẽ thay đổi khác nhau mỗi khi ta thực thi chương trình đó. Quan sát danh sách các Process đang chạy ta có được như sau :



Như các bạn thấy PID lúc này của Olly đang là 432 ở dạng decimal, để tìm kiếm thông tin về PID trong Olly chúng ta cần đổi về dạng hexa. Dùng chương trình calculator của Windows để chuyển về dạng hexa :



Như vậy, tôi có được PID ở dạng hexa là 0x1B0. Xin nhắc lại một lần nữa rằng con số này ứng với thời điểm hiện tại mà tôi đang làm việc, nếu như tôi đóng Olly và run lại thì con số này sẽ thay đổi không còn như trên hình minh họa nữa. Ta đã có được PID của Olly, giờ bước tiếp theo là làm sao tìm được giá trị này trong quá trình phân tích target. Ta quay trở lại màn hình Olly và quan sát cửa sổ Stack :



Tại sao tôi lại khẳng định được giá trị 0x0013EDE4 tương ứng với tham số pProcessIds. Đơn giản là vì hàm EnumProcesses nhận 3 tham số truyền vào, mà cơ chế push tham số lên Stack thì các bạn chắc cũng đã hiểu rồi :

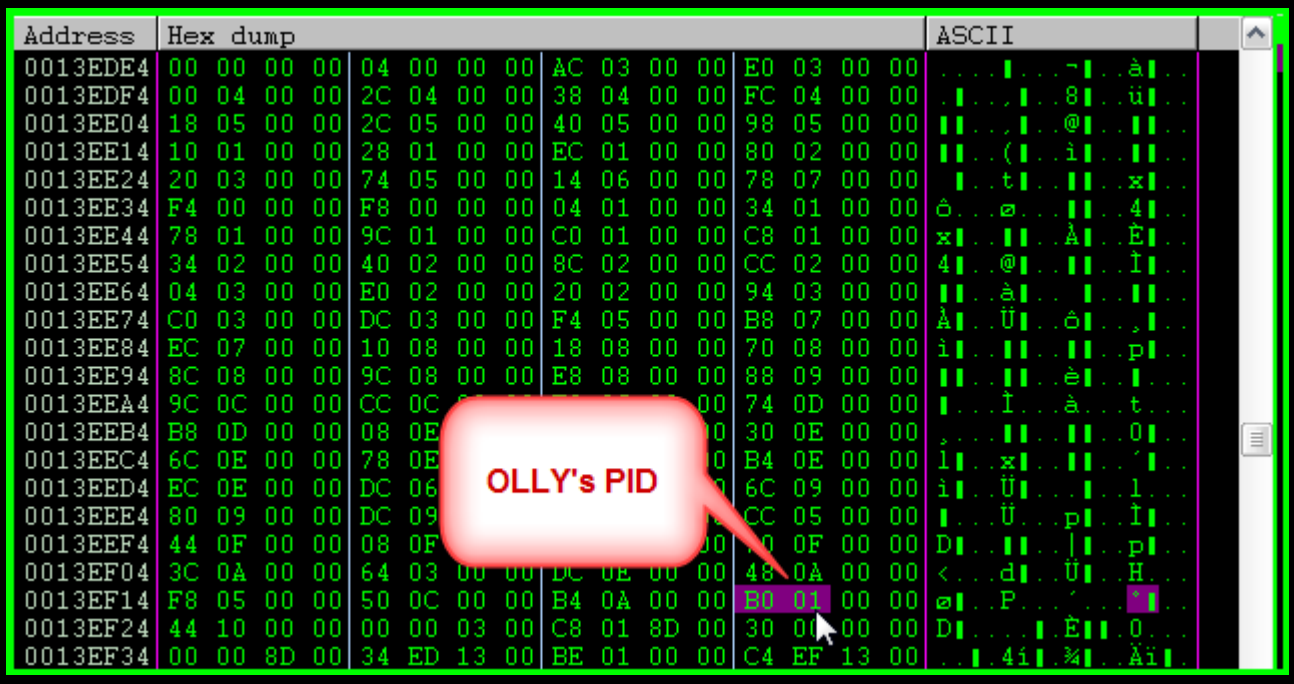
Parameters

pProcessIds [out]
A pointer to an array that receives the list of process identifiers.

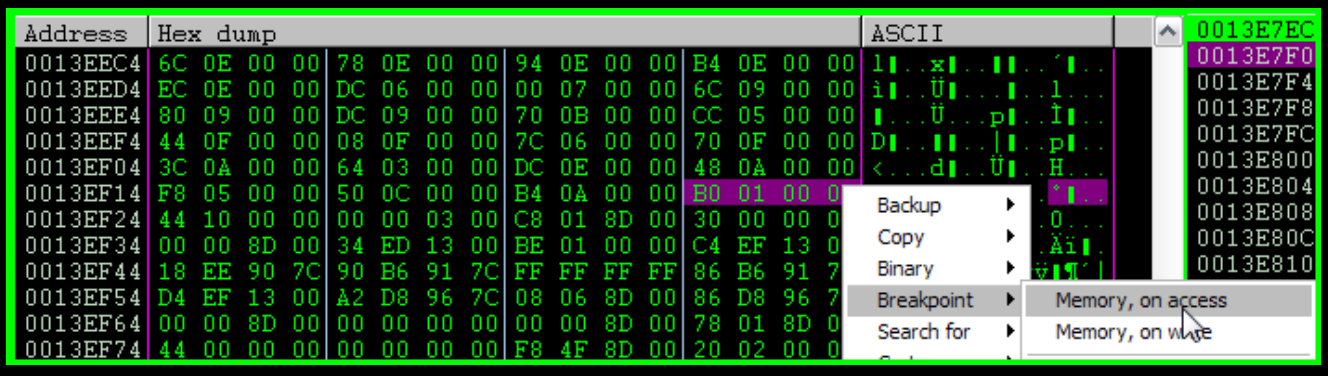
cb [in]
The size of the *pProcessIds* array, in bytes.

pBytesReturned [out]
The number of bytes returned in the *pProcessIds* array.

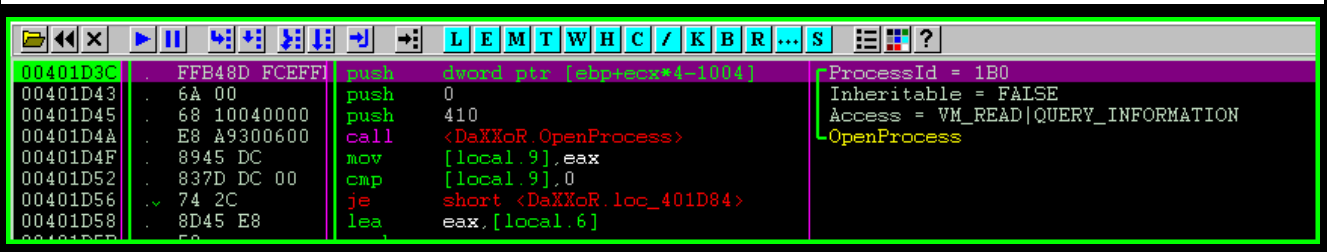
Thứ tự đưa các biến lên Stack như hình tôi mình họa ở trên. Do đó : `0x0013EDE4` là một *"pointer to an array that receives the list of process identifiers"*. Giờ ta nhấn Ctrl+F9 để thực hiện hàm `EnumProcesses` và **Follow in Dump** tại giá trị `0x0013EDE4` để tìm PID của Olly :



Ta đặt một BP lên giá trị vừa tìm được như sau :



Nhấn F9 để thực thi target, Olly sẽ dừng lại tại đoạn code có truy xuất tới giá trị PID của Olly :



Như các bạn thấy trên hình mình họa, target của chúng ta sử dụng hàm API là OpenProcess để kiểm tra xem Process của chúng ta có đang chạy hay không, và nếu có thì sẽ trả về handle của Process đó.

The **OpenProcess** function returns a handle of an existing process object.

```
HANDLE OpenProcess(
  DWORD dwDesiredAccess,      // access flag
  BOOL bInheritHandle,        // handle inheritance flag
  DWORD dwProcessId           // process identifier
);
```

Return Values

If the function succeeds, the return value is an open handle of the specified process.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

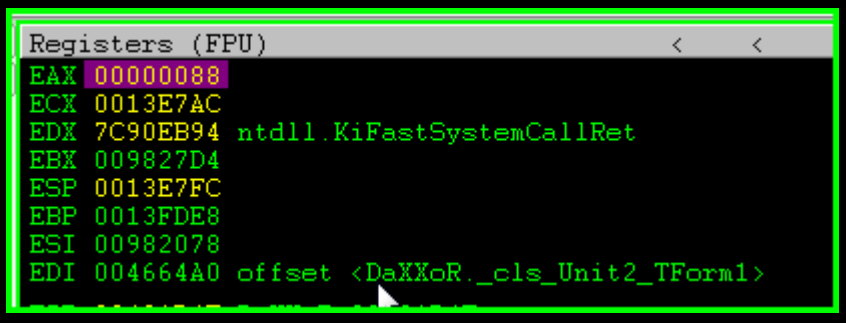
Remarks

The handle returned by the **OpenProcess** function can be used in any function that requires a handle to a process, such as the wait functions, provided the appropriate access rights were requested.

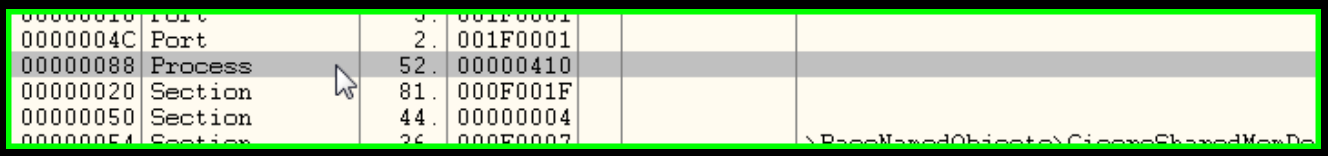
When you are finished with the handle, be sure to close it using the **CloseHandle** function.

Chà giờ đây ta có thêm một giá trị nữa là handle. Vậy PID và handle khác nhau thế nào? Trong phạm vi hiểu biết của tôi, tôi chỉ có thể giải thích như sau : PID là một con số định danh chung cho một Process đang chạy, con số này sẽ thay đổi mỗi khi bạn chạy lại chương trình. Còn handle, nó cũng là một con số liên quan tới Process mà hệ thống trả về cho chương trình của bạn, để từ đó chương trình của bạn có thể điều khiển và kiểm soát được Process đó. Với các Process khác nhau thì giá trị handle cũng khác nhau. Handle được sử dụng mỗi khi ta muốn kiểm soát một ứng dụng.

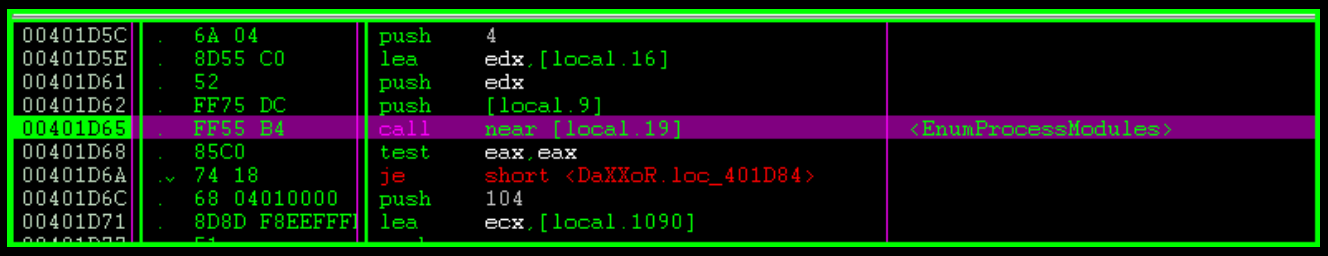
Ta tiếp tục, nhấn F8 để trace qua hàm OpenProcess. Quan sát cửa sổ Registers ta sẽ thấy EAX đang giữ một con số, đó chính là handle của Process OllyDbg :



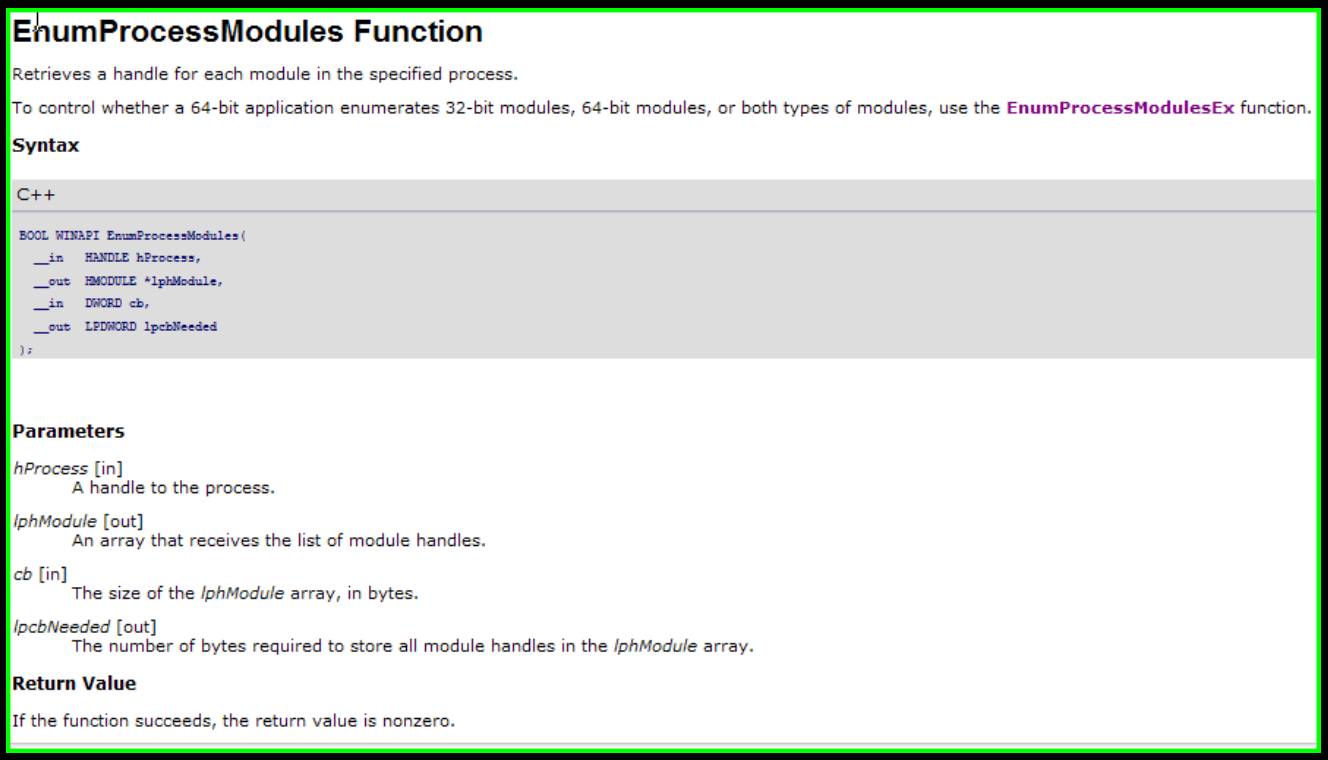
Trên máy tôi nhận được giá trị là 0x88 (giá trị này có thể khác ở máy các bạn). Để kiểm tra giá trị này có chính xác không ta mở cửa sổ **Handles (View > Handles)** :



Ta thấy có giá trị 0x88 tương ứng với kiểu Type là Process, như vậy khẳng định đây là handle của OllyDbg. Trở lại cửa sổ code, nhấn F8 và trace tới đây :



Ở đây có thể các bạn sẽ hỏi tại sao máy tôi lại hiện thông tin về lệnh call là EnumProcessModules, đơn giản là vì tôi đã đặt comment và lable cho BP này. Bước làm thế nào tôi đã nói ở trên ☺. Tra cứu và tìm kiếm thông tin về hàm này xem nó hoạt động thế nào :



EnumProcessModules Function

Retrieves a handle for each module in the specified process.

To control whether a 64-bit application enumerates 32-bit modules, 64-bit modules, or both types of modules, use the **EnumProcessModulesEx** function.

Syntax

C++

```
BOOL WINAPI EnumProcessModules(
  __in   HANDLE hProcess,
  __out  HMODULE *lphModule,
  __in   DWORD cb,
  __out  LPDWORD lpcbNeeded
);
```

Parameters

hProcess [in]
A handle to the process.

lphModule [out]
An array that receives the list of module handles.

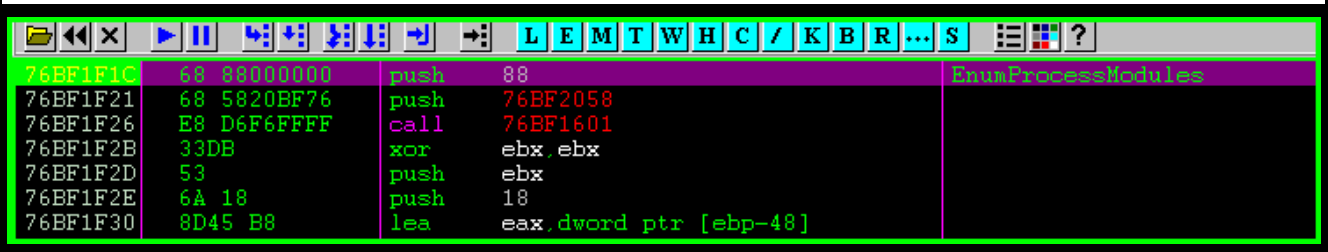
cb [in]
The size of the *lphModule* array, in bytes.

lpcbNeeded [out]
The number of bytes required to store all module handles in the *lphModule* array.

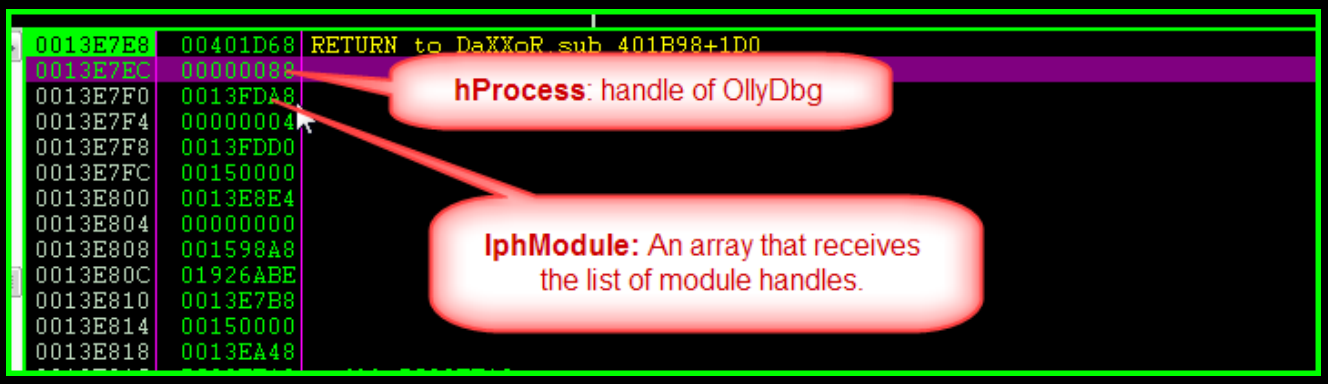
Return Value

If the function succeeds, the return value is nonzero.

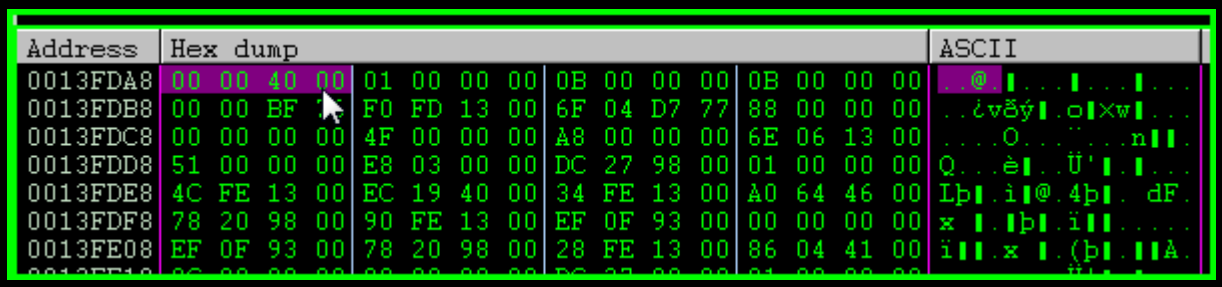
Hàm này có nhiệm vụ tìm kiếm và trả về handle của mỗi module trong Process được chỉ định. Ta thấy nó có 4 tham số được truyền vào, tham số mà ta quan tâm ở đây là *hProcess [in]: A handle to the process* và *lphModule [out]: An array that receives the list of module handles*. Nhấn F7 để trace vào lời gọi hàm,ta dừng lại tại đây :



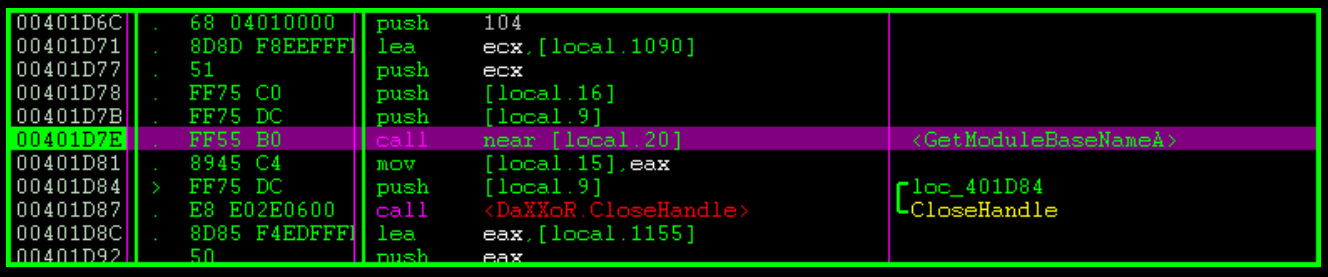
Quan sát các tham số trên cửa sổ Stack :



Nhấn Ctrl+F9 và **Follow in Dump** tại giá trị `lphModule`, ta có được như sau :



Giá trị này có nghĩa là khi ta yêu cầu handle của module thì hệ thống sẽ trả về cho ta giá trị base address. Base address là địa chỉ vùng nhờ mà process của chúng ta bắt đầu. Trong hình trên đó là giá trị `0x00400000`, tức là nơi OllyDbg bắt đầu tại đó. Tiếp tục trace tiếp cho tới khi xuất hiện hàm API thứ 3 :



Đó chính là hàm `GetModuleBaseNameA`, một lần nữa ta đi tìm thông tin về hàm này :

GetModuleBaseName Function

Retrieves the base name of the specified module.

Syntax

C++

```
DWORD WINAPI GetModuleBaseName(
  __in      HANDLE hProcess,
  __in_opt  HMODULE hModule,
  __out     LPTSTR lpBaseName,
  __in      DWORD nSize
);
```

Parameters

hProcess [in]
A handle to the process that contains the module.

The handle must have the PROCESS_QUERY_INFORMATION and PROCESS_VM_READ access rights. For more information, see Process Security and Access Rights.

hModule [in, optional]
A handle to the module. If this parameter is NULL, this function returns the name of the file used to create the calling process.

lpBaseName [out]
A pointer to the buffer that receives the base name of the module. If the base name is longer than maximum number of characters specified by the *nSize* parameter, the base name is truncated.

nSize [in]
The size of the *lpBaseName* buffer, in characters.

Return Value

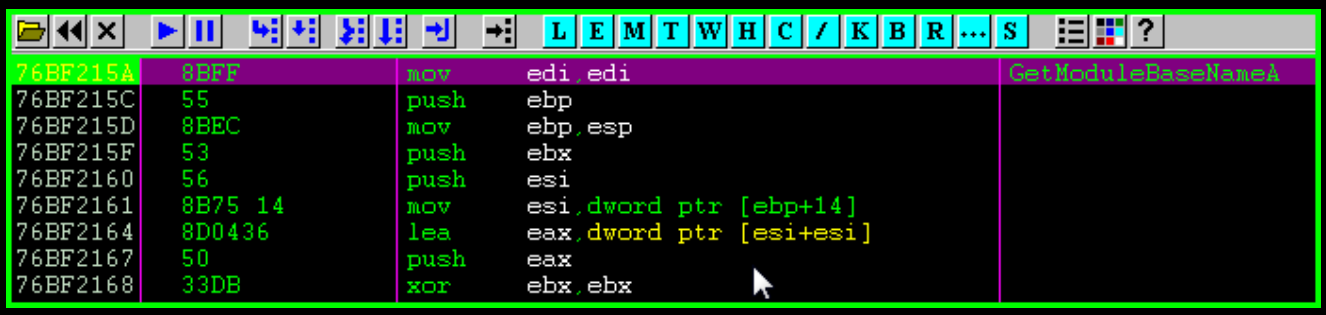
If the function succeeds, the return value specifies the length of the string copied to the buffer, in characters.

Mục đích của nó quá rõ ràng, đó là lấy ra tên của module đang được chỉ định. Trong đó :

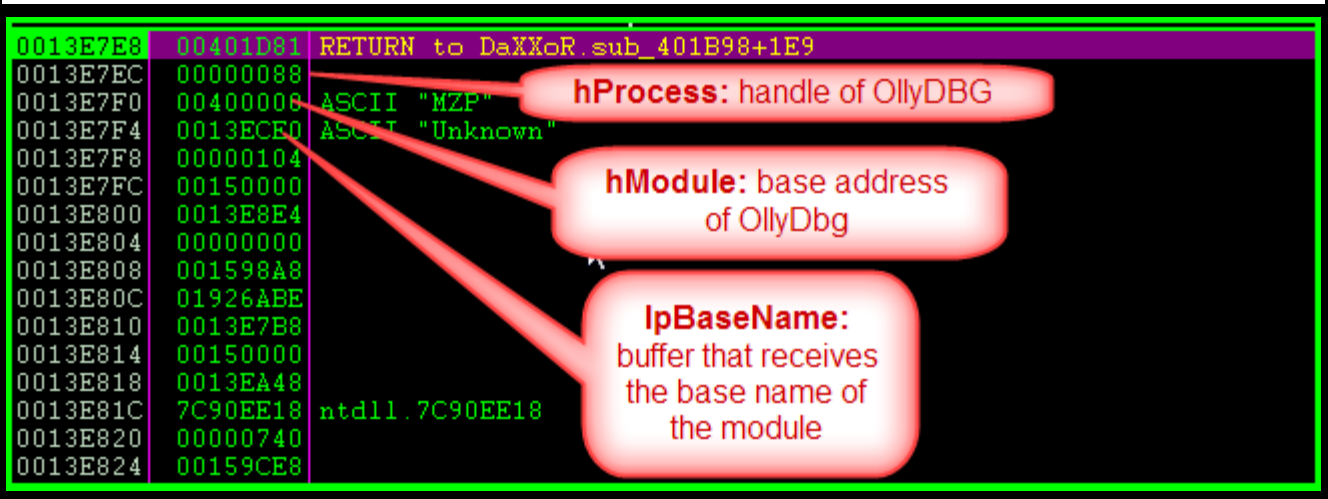
```
lpBaseName [out]

    A pointer to the buffer that receives the base name of the module. If
the base name is longer than maximum number of characters specified by the
nSize parameter, the base name is truncated.
```

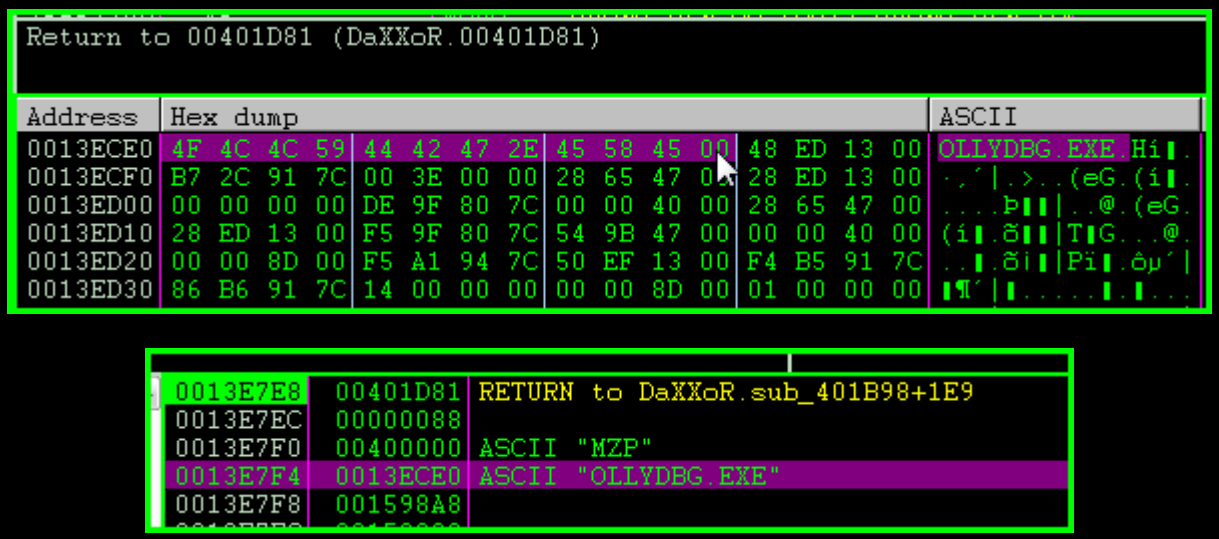
Tham số này sẽ nhận kết quả trả về của hàm là tên của module. Nhấn F7 để trace vào trong lời gọi hàm, chúng ta dừng lại tại đây :



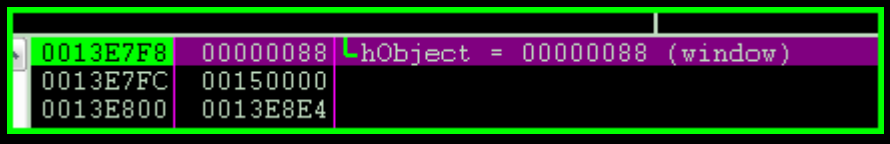
Quan sát cửa sổ Stack :



Như trên hình 0x88 là handle của Olly, `0x400000` là base address của module và `0x0013E7F4` là vùng buffer dùng để chứa tên của module. Đây là 3 tham số mà ta cần quan tâm. Ta Follow in Dump tại `lpBaseName` và nhấn Ctrll + F9, quan sát cửa sổ Dump để xem kết quả có được :



Ồ cái tên gì mà đẹp thế kia lolz ☺! Có phải là OLLYDBG.EXE không nhỉ ? Như ta thấy có được tên rồi, giờ nếu đem so sánh chuỗi với cái Name vừa tìm được, kết quả mà giống nhau thì là toi Olly. Tiếp tục nhấn F8 để trace, ta tới đây :



Khi ta đọc thông tin về hàm `OpenProcess` thì thấy có đoạn sau :

When you are finished with the handle, be sure to close it using the CloseHandle function.

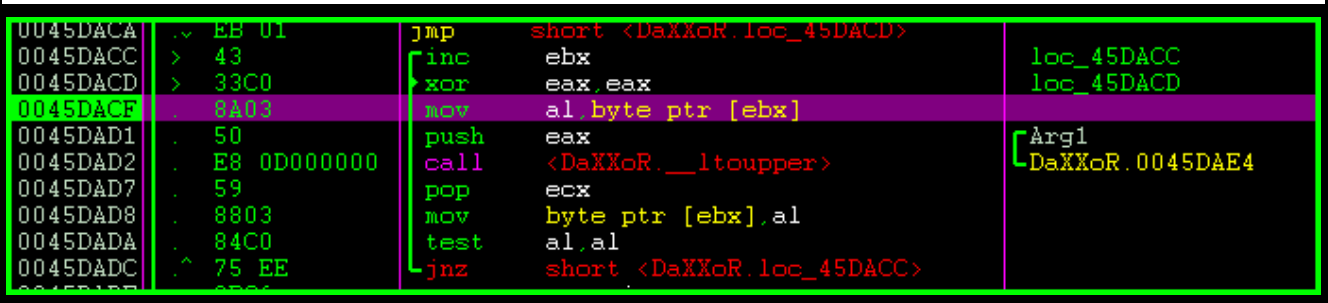
Nhấn F8 để thực hiện hàm này và quan sát cửa sổ Handle, ta thấy handle của Olly không còn trong danh sách nữa.

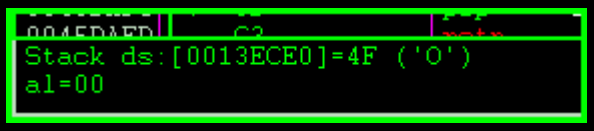
Handle	Type	Refs	Access	T	Info	Name
00000028	Desktop	5260.	000F01FF			\Default
00000008	Directory	118.	00000003			\KnownDlls
00000014	Directory	84.	000F000F			\Windows
00000030	Directory	769.	0002000F			\BaseNamedObjects
00000010	Event	3.	001F0003			
0000003C	Event	2.	001F0003			
00000040	Event	2.	001F0003			
00000044	Event	2.	001F0003			
00000048	Event	2.	001F0003			
00000034	File (dev)	2.	00100001			\Device\KsecDD
0000000C	File (dir)	2.	00100020			d:\RE_Tutorials\OllyDbg tuts
0000001C	Key	2.	000F003F			HKEY_LOCAL_MACHINE
00000038	Key	2.	000F003F			HKEY_CURRENT_USER
00000004	KeyedEvent	82.	000F0003			\KernelObjects\CritSecOutOfMemoryEve
00000058	Mutant	36.	001F0001			\BaseNamedObjects\CTF.LBES.MutexDefa
0000005C	Mutant	36.	001F0001			\BaseNamedObjects\CTF.Compart.MutexD
00000060	Mutant	36.	001F0001			\BaseNamedObjects\CTF.Asm.MutexDefau
00000064	Mutant	36.	001F0001			\BaseNamedObjects\CTF.Layouts.MutexD
00000068	Mutant	36.	001F0001			\BaseNamedObjects\CTF.TMD.MutexDefau
0000006C	Mutant	25.	001F0001			\BaseNamedObjects\CTF.TimListCache.F
00000074	Mutant	32.	001F0001			\BaseNamedObjects\SynTPFcsMutex
0000007C	Mutant	44.	00120001			\BaseNamedObjects\ShimCacheMutex
00000018	Port	3.	001F0001			
0000004C	Port	2.	001F0001			
00000020	Section	81.	000F001F			
00000050	Section	44.	00000004			
00000054	Section	36.	000F0007			\BaseNamedObjects\CiceroSharedMemDef
00000070	Section	25.	000F001F			\BaseNamedObjects\CTF.TimListCache.F
00000078	Section	32.	000F0007			\BaseNamedObjects\SynTPFcsMemMap
00000080	Section	44.	00000002			\BaseNamedObjects\ShimSharedMemory
00000024	WindowStation	160.	000F037F			\Windows\WindowStations\WinSta0
0000002C	WindowStation	160.	000F037F			\Windows\WindowStations\WinSta0

Tiếp tục trace, ta tới đoạn code sau :

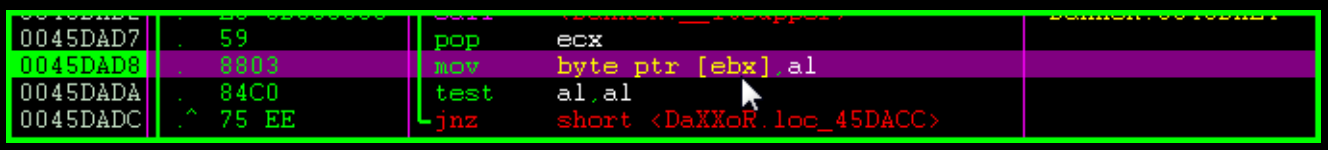
```
00401D81   .  8945 C4       mov     [local.15],eax
00401D84   >  FF75 DC       push    [local.9]                ; loc_401D84
00401D87   .  E8 E02E0600   call    <DaXXoR.CloseHandle>     ; CloseHandle
00401D8C   .  8D85 F4EDFFFF lea     eax,[local.1155]
00401D92   .  50            push    eax
00401D93   .  8D95 F8EEFFFF lea     edx,[local.1090]
00401D99   .  52            push    edx                      ; Arg1 = 0013ECE0 ASCII "OLLYDBG.EXE"
00401D9A   .  E8 21BD0500   call    <DaXXoR.__lstrupr>       ; DaXXoR.0045DAC0
00401D9F   .  59            pop     ecx
00401DA0   .  50            push    eax
00401DA1   .  E8 F29C0500   call    <DaXXoR._strcmp>
```

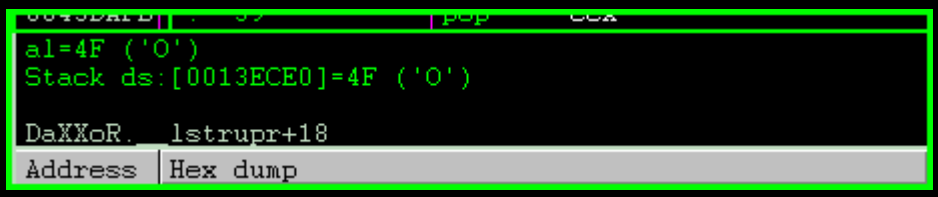
Ta thấy rằng giá trị tại vùng buffer `lpBaseName` được đẩy vào Stack, theo sau đó làm một lời gọi hàm. Nhấn F7 để trace vào hàm `00401D9A  |.  E8 21BD0500  |call <DaXXoR.__lstrupr>  ; \DaXXoR.0045DAC0`, sau đó trace tiếp đến đoạn code :



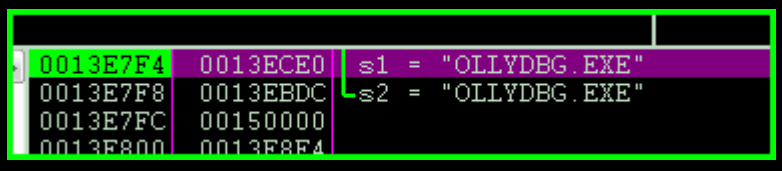


Như ta thấy, đoạn code này đọc ra kí tự đầu tiên của chuỗi OLLYDBG.EXE, đưa vào thanh ghi EAX và đẩy lên Stack. Sau đó thực hiện một lệnh CALL khác, kết quả sau đó lại lưu trở lại vùng `lpBaseName` :





Kết quả không thay đổi, vậy có nghĩa là mục đích của hàm `00401D9A |. E8 21BD0500 |call <DaXXoR.__lstrupr> ; \DaXXoR.0045DAC0` là dùng để convert từ chứ thường sang chữ hoa. Trace ra khỏi hàm này và trace tiếp tới đây :



Thông tin có được là quá rõ ràng, trên cửa sổ Stack là hai tham số `s1` và `s2` chứa hai chuỗi và hàm `00401DA1 |. E8 F29C0500 |call <DaXXoR._strcmp>` được gọi là để so sánh hai chuỗi này với nhau. Mà ta thấy hai tham số s1 và s2 cùng chứa một chuỗi là OLLYDBG.EXE ☺. Đây là toàn bộ đoạn code thực hiện công việc so sánh :

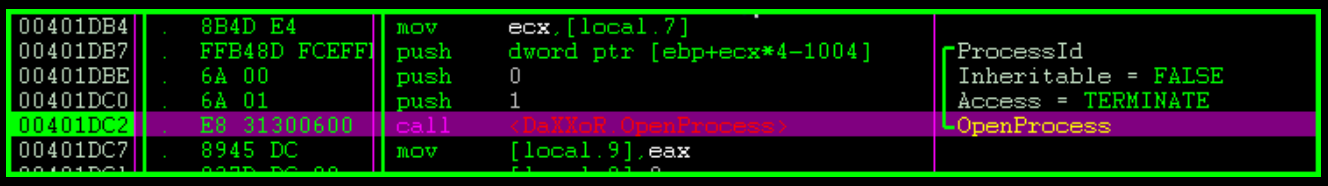
```
0045BAA5  >  8A01        mov     al,byte ptr [ecx]          loc_45BAA5
0045BAA7  .  8A1A        mov     bl,byte ptr [edx]
0045BAA9  .  2BC3        sub     eax,ebx
0045BAAB  .v 75 34       jnz     short <DaXXoR.loc_45BAE1>
0045BAAD  .  84DB        test    bl,bl
0045BAAF  .v 74 30       je      short <DaXXoR.loc_45BAE1>
0045BAB1  .  8A41 01     mov     al,byte ptr [ecx+1]
0045BAB4  .  8A5A 01     mov     bl,byte ptr [edx+1]
0045BAB7  .  2BC3        sub     eax,ebx
0045BAB9  .v 75 26       jnz     short <DaXXoR.loc_45BAE1>
0045BABB  .  84DB        test    bl,bl
0045BABD  .v 74 22       je      short <DaXXoR.loc_45BAE1>
0045BABF  .  8A41 02     mov     al,byte ptr [ecx+2]
0045BAC2  .  8A5A 02     mov     bl,byte ptr [edx+2]
0045BAC5  .  2BC3        sub     eax,ebx
0045BAC7  .v 75 18       jnz     short <DaXXoR.loc_45BAE1>
0045BAC9  .  84DB        test    bl,bl
0045BACB  .v 74 14       je      short <DaXXoR.loc_45BAE1>
0045BACD  .  8A41 03     mov     al,byte ptr [ecx+3]
0045BAD0  .  8A5A 03     mov     bl,byte ptr [edx+3]
0045BAD3  .  2BC3        sub     eax,ebx
0045BAD5  .v 75 0A       jnz     short <DaXXoR.loc_45BAE1>
0045BAD7  .  83C1 04     add     ecx,4
0045BADA  .  83C2 04     add     edx,4
0045BADD  .  84DB        test    bl,bl
0045BADF  .^ 75 C4       jnz     short <DaXXoR.loc_45BAA5>
```

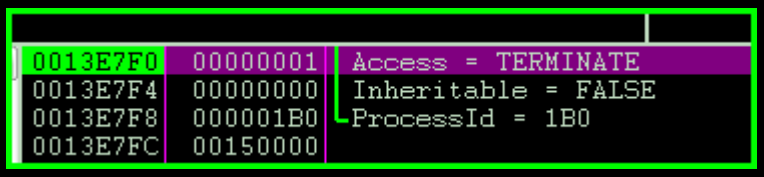
Kết quả so sánh thế nào sẽ tác động trực tiếp lên lệnh nhảy bên dưới, khác nhau thì nhảy và tiếp tục quá trình tiếp theo, còn giống nhau thì chắc các bạn đã tưởng tượng được điều gì sẽ diễn ra :

```
401DA1  .  E8 F29C0500   call    <DaXXoR._strcmp>
401DA6  .  83C4 08       add     esp,8
401DA9  .  85C0          test    eax,eax
401DAB  .v 75 74         jnz     short <DaXXoR.loc_401E21>
401DAD  .  C745 E0 0100  mov     [local.8],1
401DB4  .  8B4D E4       mov     ecx,[local.7]
401DB7  .  FFB48D FCEFFF push    dword ptr [ebp+ecx*4-1004]   ProcessId
401DBE  .  6A 00         push    0                            Inheritable = FALSE
401DC0  .  6A 01         push    1                            Access = TERMINATE
401DC2  .  E8 31300600   call    <DaXXoR.OpenProcess>         OpenProcess
401DC7  .  8945 DC       mov     [local.9],eax
401DCA  .  837D DC 00    cmp     [local.9],0
401DCE  .v 74 3F         je      short <DaXXoR.loc_401E0F>
401DD0  .  6A 00         push    0                            ExitCode = 0
401DD2  .  FF75 DC       push    [local.9]                    hProcess
401DD5  .  E8 78300600   call    <DaXXoR.TerminateProcess>    TerminateProcess
401DDA  .  85C0          test    eax,eax
401DDC  .v 74 17         je      short <DaXXoR.loc_401DF5>
401DDE  .  FF75 DC       push    [local.9]                    hObject
401DE1  .  E8 862E0600   call    <DaXXoR.CloseHandle>         CloseHandle
401DE6  .  FF75 D0       push    [local.12]                   hLibModule
401DE9  .  E8 CC2E0600   call    <DaXXoR.FreeLibrary>         FreeLibrary
```

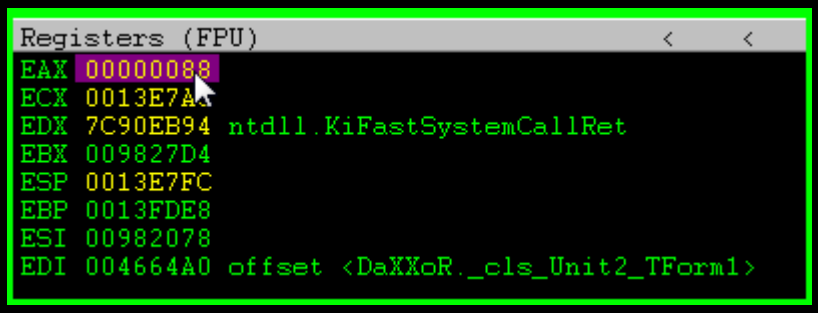
Ở đây do kết quả so sánh là giống nhau, thanh ghi EAX có giá trị là 0x0 cho nên lệnh nhảy không thực hiện. Lúc này target của chúng ta lại gọi lại hàm API là OpenProcess để lấy lại handle của Olly, đồng thời truyên thêm tham số là :

PROCESS_TERMINATE Enables using the process handle in the TerminateProcess function to terminate the process.

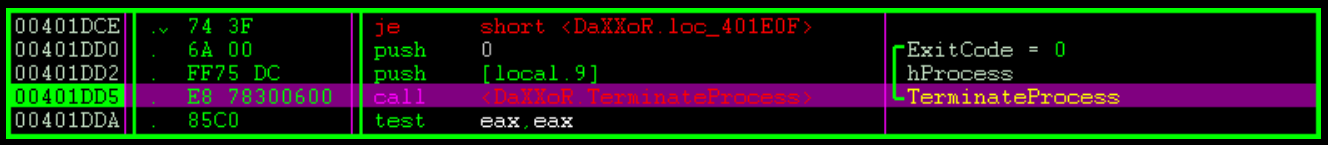


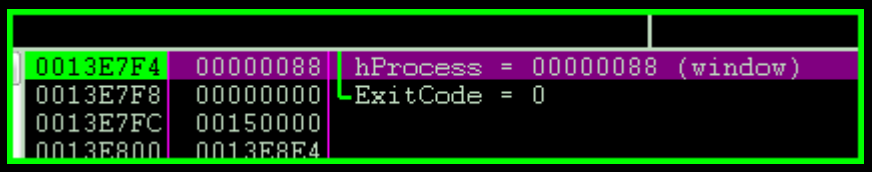


Nhấn F8 để thực hiện OpenProcess quan sát cửa sổ Registers ta thấy giá trị handle của Olly :



Ta trace tiếp tới lời gọi hàm 00401DD5 |. E8 78300600 |call <DaXXoR.TerminateProcess> ; \TerminateProcess .





Tới đây coi như Olly của chúng ta chuẩn bị bay rồi đấy, nhấn F8 một phát và bùm …, Olly văng luôn, bị terminate thẳng tay. Giờ làm cách nào để bypass cơ chế này bây giờ, có 3 cách như sau :

1. Thay đổi kết quả trả về của hàm OpenProcess.
2. Patch thẳng vào lệnh nhảy để vượt qua lời gọi hàm TerminateProcess.
3. Đổi tên của Olly thành tên khác ☺.

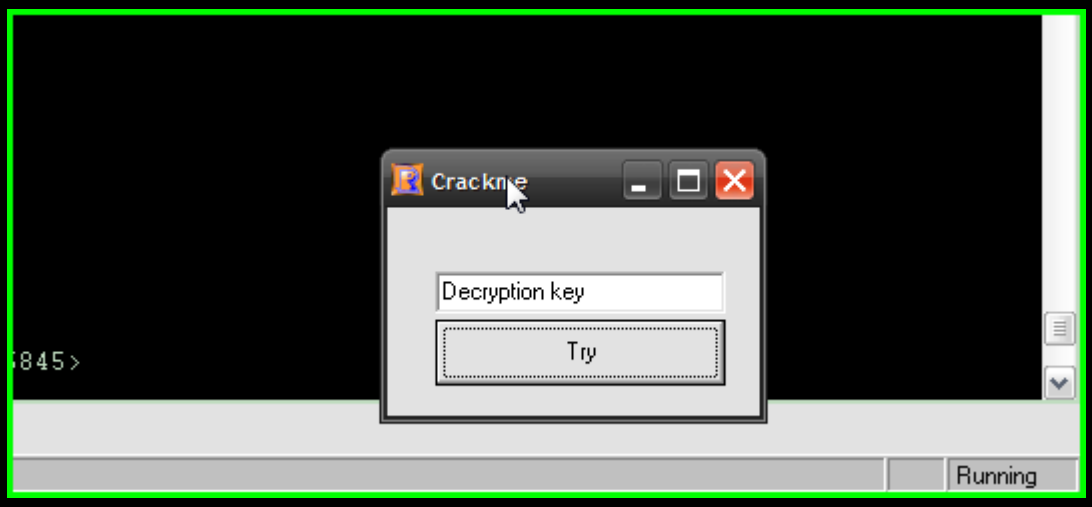
Với cách đầu tiên ta làm như sau, mở Olly lên và đặt BP tại hàm OpenProcess và nhấn F9 để thực thi :

```
7C8309E1   8BFF            mov     edi,edi                    <DaXXoR._cls_Unit2_TForm1>
7C8309E3   55              push    ebp
7C8309E4   8BEC            mov     ebp,esp
7C8309E6   83EC 20         sub     esp,20
7C8309E9   8B45 10         mov     eax,dword ptr [ebp+10]
7C8309EC   8945 F8         mov     dword ptr [ebp-8],eax
7C8309EF   8B45 0C         mov     eax,dword ptr [ebp+C]
7C8309F2   56              push    esi
7C8309F3   33F6            xor     esi,esi
7C8309F5   F7D8            neg     eax
7C8309F7   1BC0            sbb     eax,eax
7C8309F9   83E0 02         and     eax,2
7C8309FC   8945 EC         mov     dword ptr [ebp-14],eax
7C8309FF   8D45 F8         lea     eax,dword ptr [ebp-8]
7C830A02   50              push    eax
7C830A03   8D45 E0         lea     eax,dword ptr [ebp-20]
7C830A06   50              push    eax
7C830A07   FF75 08         push    dword ptr [ebp+8]
7C830A0A   8D45 10         lea     eax,dword ptr [ebp+10]
7C830A0D   50              push    eax
7C830A0E   8975 FC         mov     dword ptr [ebp-4],esi
7C830A11   C745 E0 180000  mov     dword ptr [ebp-20],18
7C830A18   8975 E4         mov     dword ptr [ebp-1C],esi
7C830A1B   8975 E8         mov     dword ptr [ebp-18],esi
7C830A1E   8975 F0         mov     dword ptr [ebp-10],esi
7C830A21   8975 F4         mov     dword ptr [ebp-C],esi
7C830A24   FF15 0C11807C   call    near dword ptr [<&ntdll.NtOpenP: ntdll.ZwOpenProcess
```

Ta fix như sau :

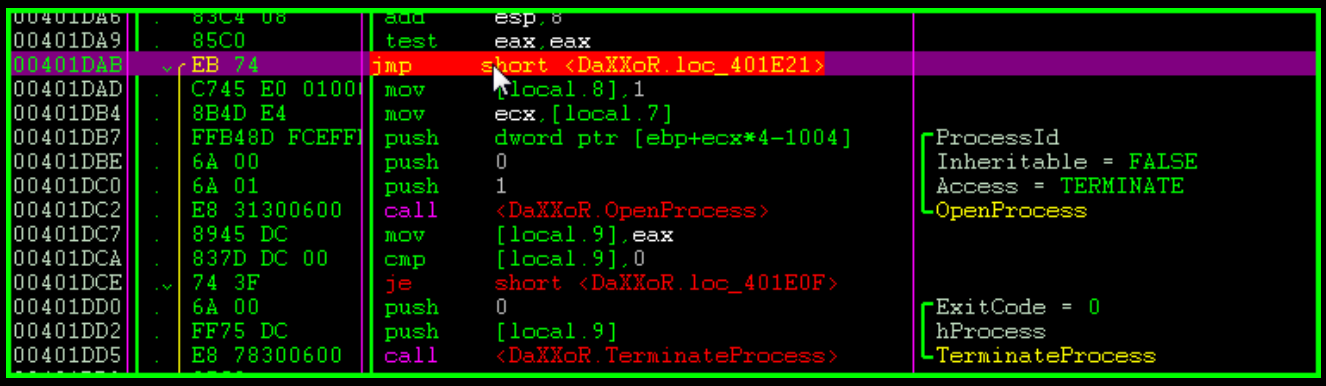
```
7C830A24   FF15 0C11807C   call    near dword ptr [<&ntdll.NtOpenP: ntdll.ZwOpenProcess
7C830A2A   3BC6            cmp     eax,esi
7C830A2C   5E              pop     esi
7C830A2D   90              nop
7C830A2E   90              nop
7C830A2F   90              nop
7C830A30   90              nop
7C830A31   90              nop
7C830A32   90              nop
7C830A33   33C0            xor     eax,eax
7C830A35   90              nop
7C830A36   C9              leave
7C830A37   C2 0C00         retn    0C
```

Patch xong ta xóa BP đã đặt đi, sau đó nhấn F9 để kiểm tra kết quả :

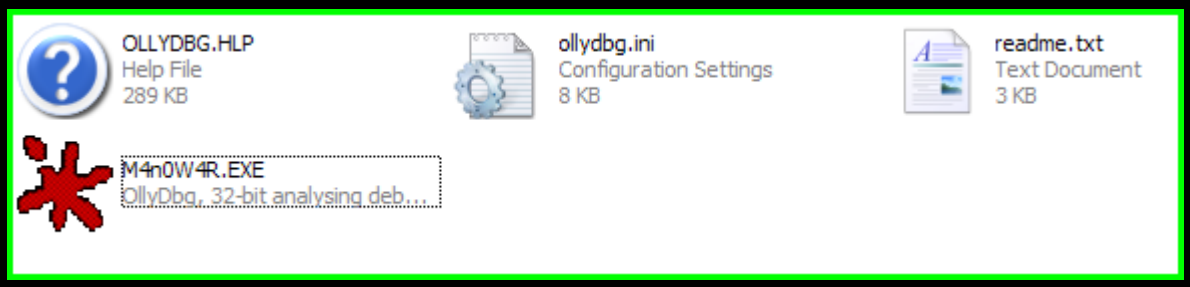


Tuy nhiên cách patch này không được khuyến khích vì nó tác động trực tiếp vào hàm API, ảnh hưởng tới quá trình sử dụng hàm sau này.

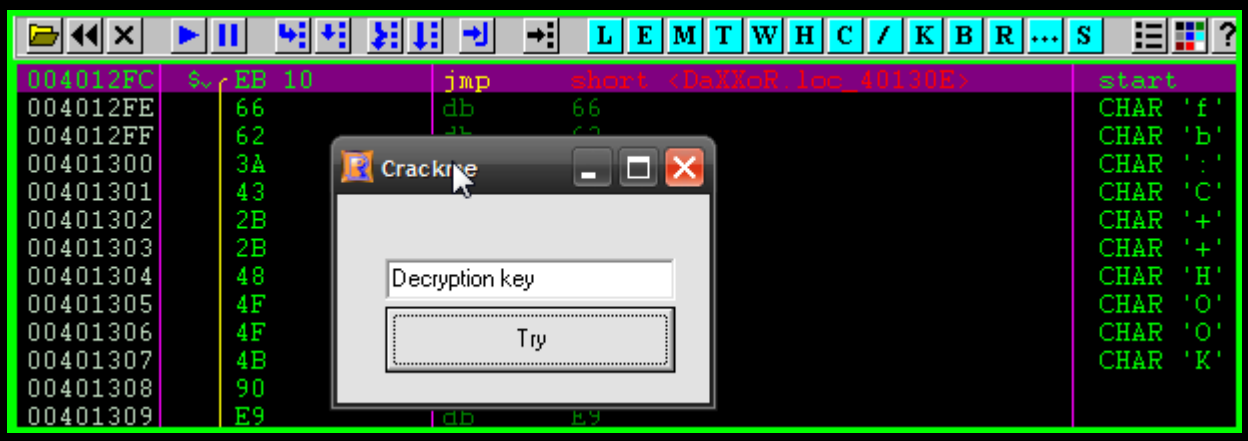
Cách thứ hai đơn giản hơn, ta patch lệnh JNZ thành JMP :



Cuối cùng là cách 3, đơn giản nhất mà lại hiệu quả cao. Ta đổi tên của OllyDbg thành tên bất kì mà ta muốn. Ví dụ :



Chạy file mới này, load target vào và nhấn F9 để run, ta thấy target chạy vù vù ☺ :



Kết thúc bài 20!

III. Kết luận

OK, toàn bộ bài 20 đến đây là kết thúc. Bài viết này giới thiệu tiếp tới các bạn cơ chế Anti-Olly bằng cách tìm xem Process của Olly có đang run hay không, nếu có thì lấy ra tên và so sánh, giống nhau thì kill process. Ngoài ra, bài viết cũng đưa ra một số giải

pháp để vượt qua cơ chế này. Hẹn gặp lại các bạn ở bài 21, hứa hẹn sẽ mang đến những kiến thức mới mẻ hơn nữa!

PS: Tài liệu này chỉ mang tính tham khảo, tác giả không chịu trách nhiệm nếu người đọc sử dụng nó vào bất kì mục đích nào.

Best Regards
[Kienmanowar]

--++--==[Greatz Thanks To]==--++--

My family, Computer_Angel, Moonbaby , Zombie_Deathman, Littleboy, Benina, QHQCrker, the_Lighthouse, Merc, Hoadongnoi, Nini ... all REA's members, TQN, HacNho, RongChauA, Deux, tlandn, light.phoenix, dqtln, ARTEAM .... all my friend, and YOU.

--++--==[Thanks To]==--++--

iamidiot, WhyNotBar, trickyboy, dzungltvn, takada, hurt_heart, haule_nth, hytkl, moth, XIANUA, nhc1987, 0xdie, Unregistered!, akira, mranglex v..v.. các bạn đã đóng góp rất nhiều cho REA. Hi vọng các bạn sẽ tiếp tục phát huy ☺

I want to thank **Teddy Roggers** for his great site, Reversing.be folks(especially **haggar**), Arteam folks(**Shub-Nigurrath, MaDMAn_H3rCuL3s**) and all folks on crackmes.de, thank to all members of **unpack.cn** (especially **fly** and **linhanshi**). Great thanks to **lena151**(I like your tutorials). And finally, thanks to **RICARDO NARVAJA** and all members on **CRACKSLATINOS.**

>>>> If you have any suggestions, comments or corrections email me:
kienmanowar[at]reaonline.net