

2009

[Cracking with OllyDbg]

Based on OllyDbg tuts of Ricardo Narvaja (CrackLatinos Team)



www.reasonline.net

kienmanowar



13/01/2010

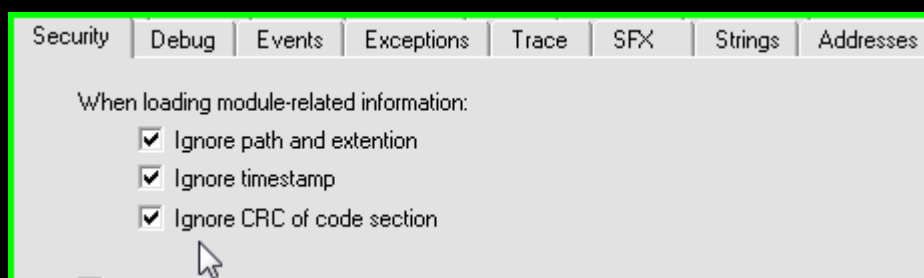
Mục Lục

I. Giới thiệu chung	2
II. Phân tích và xử lý target	3
1. Phân tích DaXXoR - Decryptme	3
III. Kết luận.....	21

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, Now let's go.....☺

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 :

Foxit Reader.exe	2736	Foxit Reader 3.1. Best Reader for Everyday Use!	Foxit Software
Snagit32.exe	4384	Snagit	TechSmith Corporation
OLLYDBG.EXE	3504	OllyDbg, 32-bit analysing debugger	

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 trắng trời điều gì ☹.

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

004012FC	EB 10	jmp	short <DaXXoR loc_40130E>	start
004012FE	66	db	66	CHAR 'f'
004012FF	62	db	62	CHAR 'b'
00401300	3A	db	3A	CHAR ':'
00401301	43	db	43	CHAR 'C'
00401302	2B	db	2B	CHAR '+'
00401303	2B	db	2B	CHAR '+'
00401304	48	db	48	CHAR 'H'
00401305	4F	db	4F	CHAR 'O'
00401306	4F	db	4F	CHAR 'O'
00401307	4B	db	4B	CHAR 'K'
00401308	90	nop		
00401309	E9	db	E9	
0040130A	98604600	dd	offset DaXXoR.___CPPdebugHook	
0040130E	A1 8B604600	mov	eax,dword ptr [<dwTlsIndex>]	loc_40130E

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::TContainedActionList *)	Actnlist::TContainedAction::SetActionList(Actnlist::TContainedActionList *)
00427AC0	text	User	Actnlist::TContainedAction::SetCategory(System::AnsiString)	Actnlist::TContainedAction::SetCategory(System::AnsiString)
00427A78	text	User	Actnlist::TContainedAction::SetIndex(int)	Actnlist::TContainedAction::SetIndex(int)
00427B14	text	User	Actnlist::TContainedAction::SetParentComponent(Classes::TComponent *)	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::TPersistent *)	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 :

00472A1C	.idata	Import	USER32.SetParentComponent	
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
00472A20	.idata	Import	USER32.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.

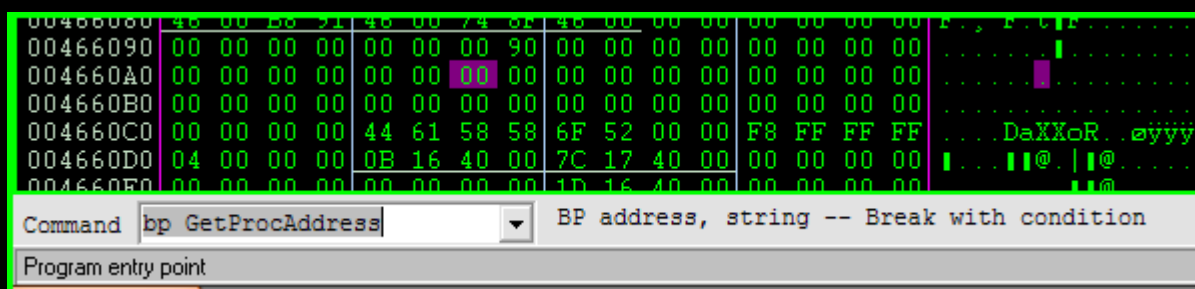
Return Values

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.

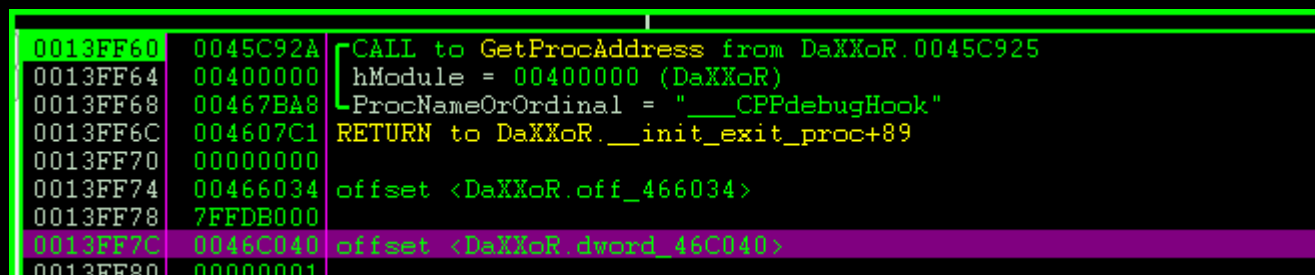
Remarks

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

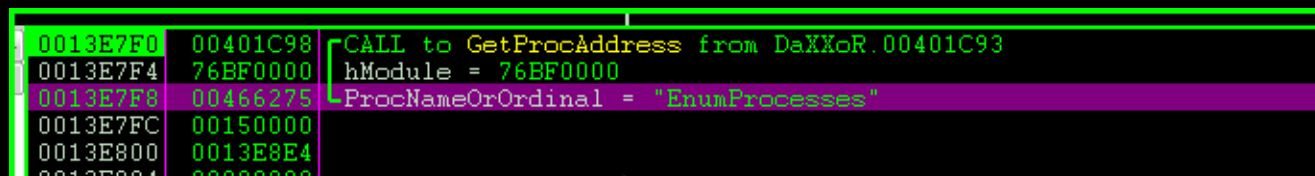
Tóm lại, nhiệm vụ của hàm `GetProcAddress` là tìm ra địa chỉ của một hàm trong file DLL. Thường được sử dụng khi chương trình cần load một hàm API mới mà không được liệt kê trong danh sách. Ta đặt BP tại hàm này :



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 Process nên đặt nghi ngờ. Nhấn **Ctrl + F9 (Execute till Return)** và quan sát giá trị của EAX ở cửa sổ Registers :

```

Registers (FPU)
EAX 76BF3A9A
ECX 7C919AEB ntdll.7C919AEB
EDX 7C97C0D8 ntdll.7C97C0D8
EBX 009827D4
ESP 0013E7F0
EBP 0013FDE8
ESI 00982078
EDI 004664A0 offset <DaXXoR._cls_Unit2_TForm1>
EIP 7C80ADFF kernel32.7C80ADFF

```

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 :

```

004660C0 00 00 00 00 44 61 58 58 6F 52 00 00 F8 FF FF FF ...DaXXoR..øyyy
004660D0 04 00 00 00 0B 16 40 00 7C 17 40 00 00 00 00 00 |...||@.||@.....
004660E0 00 00 00 00 00 00 00 00 1D 16 40 00 00 00 00 00 |...||@.....

Command bp EnumProcesses Unknown identifier
Breakpoint at <DaXXoR.GetProcAddress>

```

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

```

004660D0 04 00 00 00 0B 16 40 00 7C 17 40 00 00 00 00 00 |...||@.||@.....
004660E0 00 00 00 00 00 00 00 00 1D 16 40 00 00 00 00 00 |...||@.....

Command bp 76BF3A9A BP address, string -- Break with condition

```

```

76BF3A9A 6A 1C push 1C
76BF3A9C 68 C83BBF76 push 76BF3BC8
76BF3AA1 E8 5DBEFF7F call 76BF1601
76BF3AA6 BE 00800000 mov esi,8000
76BF3AAB 8975 E4 mov dword ptr [ebp-1C],esi
76BF3AAE 56 push esi
76BF3AAF 8B3D B810BF76 mov edi,dword ptr [76BF10B8]
76BF3AB5 EB 25 jmp short 76BF3ADC
kernel32.LocalAlloc

```

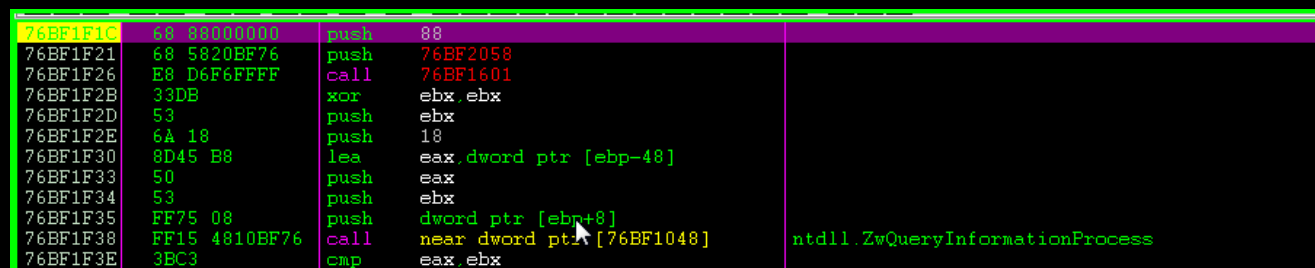
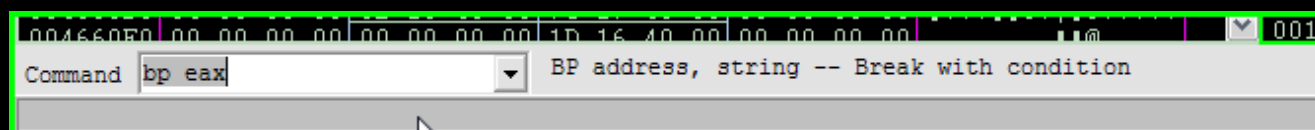
Đặt BP xong tiếp tục nhấn F9 để tìm kiếm thông tin về các hàm APIs khác :

```

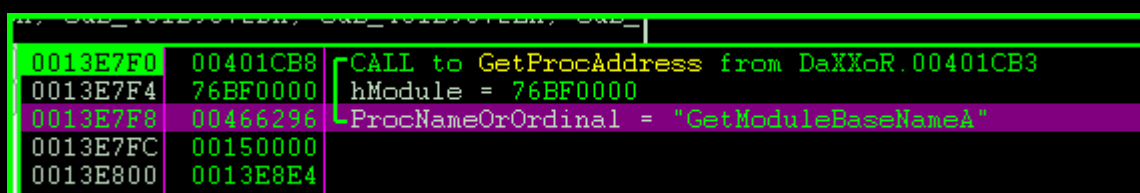
0013E7F0 00401CA8 CALL to GetProcAddress from DaXXoR.00401CA3
0013E7F4 76BF0000 hModule = 76BF0000
0013E7F8 00466283 ProcNameOrOrdinal = "EnumProcessModules"
0013E7FC 00150000
0013E800 0013E8E4
0013E804 00000000
0013E808 001598A8

```

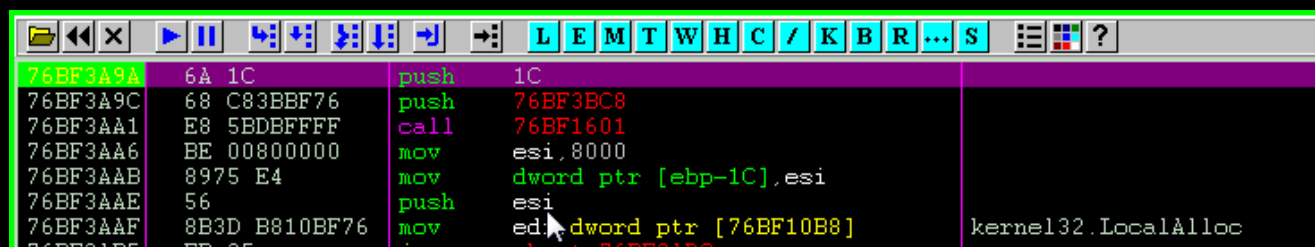
Chà lại một hàm nữa, ta thực hiện tương tự như trên và đặt BP :



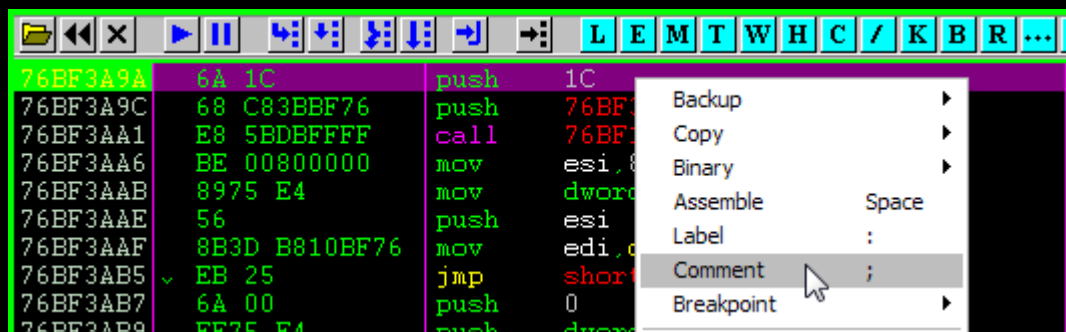
Tiếp tục nhấn F9 và Olly lại break :



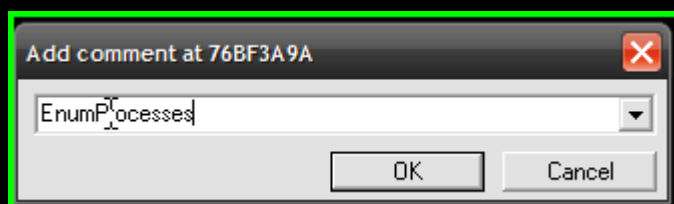
Đặt BP tương tự như những gì đã thực hiện. Sau đó nhấn F9, lúc này Olly sẽ break tại hàm **EnumProcesses** mà ta đã đặt BP :



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 :



76BF3A9A	6A 1C	push	1C	EnumProcesses
76BF3A9C	68 C83BBF76	push	76BF3BC8	
76BF3AA1	E8 5BDBFFFF	call	76BF1601	
76BF3AA6	BE 00800000	mov	esi, 8000	

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](#).

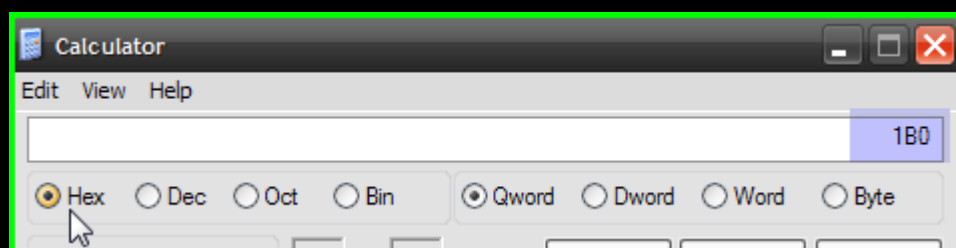
Remarks

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**.

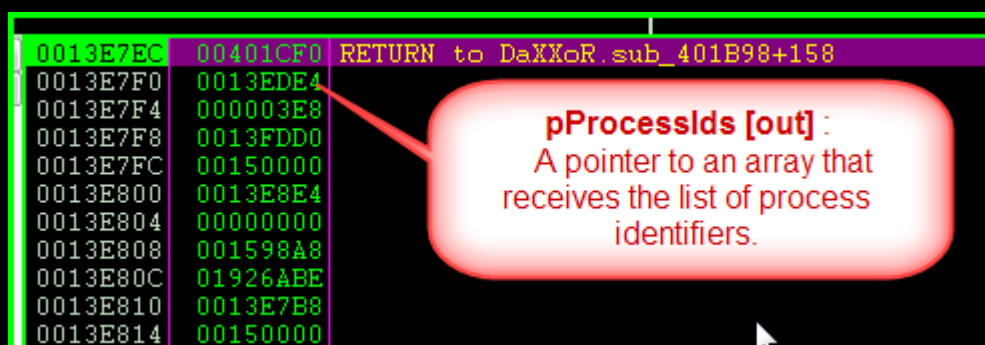
Theo kết quả tìm kiếm về hàm này cho thấy nó được sử dụng để lấy thông tin về **Process Identifier (PID)** của các chương trình đang chạy trên hệ thống. Mỗi chương trình khi thực thi sẽ được nhận biết bởi một con số, con số này sẽ thay đổi khác nhau mỗi khi ta thực thi chương trình đó. Quan sát danh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

Process	PID	CPU	Description
ctfmon.exe	176		CTF Loader
UniKeyNT.exe	228		
BtTray.exe	204		Bluetooth Tray Applicat
firefox.exe	160		Firefox
YahooMessenger.exe	352		Yahoo! Messenger
WINWORD.EXE	68		Microsoft Office Word
Snagit32.exe	304		Snagit
TscHelp.exe	232		TechSmith HTML Help
SnagitPriv.exe	128		Snagit RPC Helper
SnagitEditor.exe	22		Snagit Editor
Foxit Reader.exe	2740		Foxit Reader 3.1. Best I
OLLYDBG.EXE	432		OllyDbg. 32-bit analysi
DaXXoR.ExE	4164		

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 vị 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 minh 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 Oly :

Address	Hex dump	ASCII
0013EDE4	00 00 00 00 04 00 00 00 AC 03 00 00 E0 03 00 00-!...à!...
0013EDF4	00 04 00 00 2C 04 00 00 38 04 00 00 FC 04 00 00	...8!...ü!...
0013EE04	18 05 00 00 2C 05 00 00 40 05 00 00 98 05 00 00	...@!...!...
0013EE14	10 01 00 00 28 01 00 00 EC 01 00 00 80 02 00 00	...(!...!...
0013EE24	20 03 00 00 74 05 00 00 14 06 00 00 78 07 00 00	...t!...!x!...
0013EE34	F4 00 00 00 F8 00 00 00 04 01 00 00 34 01 00 00	ô...ø...!4!...
0013EE44	78 01 00 00 9C 01 00 00 C0 01 00 00 C8 01 00 00	x!...!À!È!...
0013EE54	34 02 00 00 40 02 00 00 8C 02 00 00 CC 02 00 00	4!...@!...!Î!...
0013EE64	04 03 00 00 E0 02 00 00 20 02 00 00 94 03 00 00	!...à!...!Î!...
0013EE74	C0 03 00 00 DC 03 00 00 F4 05 00 00 B8 07 00 00	À!...Û!...ô!...!...
0013EE84	EC 07 00 00 10 08 00 00 18 08 00 00 70 08 00 00	i!...!...!p!...
0013EE94	8C 08 00 00 9C 08 00 00 E8 08 00 00 88 09 00 00	!...!...è!...!...
0013EEA4	9C 0C 00 00 CC 0C 00 00 74 0D 00 00	!...!...à!...t!...
0013EEB4	B8 0D 00 00 08 0E 00 00 30 0E 00 00	...!...!...0!...
0013EEC4	6C 0E 00 00 78 0E 00 00 B4 0E 00 00	l!...x!...!...!...
0013EED4	EC 0E 00 00 DC 06 00 00 6C 09 00 00	i!...Û!...!...l!...
0013EEE4	80 09 00 00 DC 09 00 00 CC 05 00 00	!...Û!...p!...!Î!...
0013EEF4	44 0F 00 00 08 0F 00 00 70 0F 00 00	D!...!...!p!...
0013EF04	3C 0A 00 00 64 03 00 00 DC 0E 00 00 48 0A 00 00	<...d!...Û!...H!...
0013EF14	F8 05 00 00 50 0C 00 00 B4 0A 00 00 B0 01 00 00	ø!...P!...!...!...
0013EF24	44 10 00 00 00 00 03 00 C8 01 8D 00 30 00 00 00	D!...!...È!...0!...
0013EF34	00 00 8D 00 34 ED 13 00 BE 01 00 00 C4 EF 13 00	...4i!...!À!...

OLLY's PID

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

Address	Hex dump	ASCII	
0013EEC4	6C 0E 00 00 78 0E 00 00 94 0E 00 00 B4 0E 00 00	l!...x!...!...!...	0013E7EC
0013EED4	EC 0E 00 00 DC 06 00 00 00 07 00 00 6C 09 00 00	i!...Û!...!...l!...	0013E7F0
0013EEE4	80 09 00 00 DC 09 00 00 70 0B 00 00 CC 05 00 00	!...Û!...p!...!Î!...	0013E7F4
0013EEF4	44 0F 00 00 08 0F 00 00 7C 06 00 00 70 0F 00 00	D!...!...!p!...	0013E7F8
0013EF04	3C 0A 00 00 64 03 00 00 DC 0E 00 00 48 0A 00 00	<...d!...Û!...H!...	0013E7FC
0013EF14	F8 05 00 00 50 0C 00 00 B4 0A 00 00 B0 01 00 00	ø!...P!...!...!...	0013E800
0013EF24	44 10 00 00 00 00 03 00 C8 01 8D 00 30 00 00 00	D!...!...È!...0!...	0013E804
0013EF34	00 00 8D 00 34 ED 13 00 BE 01 00 00 C4 EF 13 00	...4i!...!À!...	0013E808
0013EF44	18 EE 90 7C 90 B6 91 7C FF FF FF FF 86 B6 91 7C	...!...!...!...	0013E80C
0013EF54	D4 EF 13 00 A2 D8 96 7C 08 06 8D 00 86 D8 96 7C	...!...!...!...	0013E810
0013EF64	00 00 8D 00 00 00 00 00 00 00 8D 00 78 01 8D 00	...!...!...!...	
0013EF74	44 00 00 00 00 00 00 00 F8 4F 8D 00 20 02 00 00	...!...!...!...	

Backup
Copy
Binary
Breakpoint
Search for

Memory, on access
Memory, on write

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

```

00401D3C . FFB48D FCEFF push dword ptr [ebp+ecx*4-1004]
00401D43 . 6A 00 push 0
00401D45 . 68 10040000 push 410
00401D4A . E8 A9300600 call <DaXXoR.OpenProcess>
00401D4F . 8945 DC mov [local.9],eax
00401D52 . 837D DC 00 cmp [local.9],0
00401D56 . 74 2C je short <DaXXoR.loc_401D84>
00401D58 . 8D45 E8 lea eax,[local.6]

```

ProcessId = 1B0
Inheritable = FALSE
Access = VM_READ|QUERY_INFORMATION
OpenProcess

Như các bạn thấy trên hình minh 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 :

```

Registers (FPU)
EAX 00000088
ECX 0013E7AC
EDX 7C90EB94 ntdll.KiFastSystemCallRet
EBX 009827D4
ESP 0013E7FC
EBP 0013FDE8
ESI 00982078
EDI 004664A0 offset <DaXXoR._cls_Unit2_TForm1>

```

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)** :

00000010	Port	3.	001F0001		
0000004C	Port	2.	001F0001		
00000088	Process	52.	00000410		
00000020	Section	81.	000F001F		
00000050	Section	44.	00000004		
00000054	Section	36.	000F0007		

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 :

00401D5C	6A 04	push	4	
00401D5E	8D55 C0	lea	edx,[local.16]	
00401D61	52	push	edx	
00401D62	FF75 DC	push	[local.9]	
00401D65	FF55 B4	call	near [local.19]	<EnumProcessModules>
00401D68	85C0	test	eax, eax	
00401D6A	74 18	je	short <DaXXoR.loc_401D84>	
00401D6C	68 04010000	push	104	
00401D71	8D8D F8EEFF	lea	ecx,[local.1090]	

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 :

76BF1F1C	68 88000000	push	88	EnumProcessModules
76BF1F21	68 5820BF76	push	76BF2058	
76BF1F26	E8 D6F6FFFF	call	76BF1601	
76BF1F2B	33DB	xor	ebx,ebx	
76BF1F2D	53	push	ebx	
76BF1F2E	6A 18	push	18	
76BF1F30	8D45 B8	lea	eax,dword ptr [ebp-48]	

Quan sát các tham số trên cửa sổ Stack :

0013E7E8	00401D68	RETURN to DaXXoR sub 401E98+1D0
0013E7EC	00000088	
0013E7F0	0013FDA8	
0013E7F4	00000004	
0013E7F8	0013FDD0	
0013E7FC	00150000	
0013E800	0013E8E4	
0013E804	00000000	
0013E808	001598A8	
0013E80C	01926ABE	
0013E810	0013E7B8	
0013E814	00150000	
0013E818	0013EA48	

hProcess: handle of OllyDbg

lpModule: An array that receives the list of module handles.

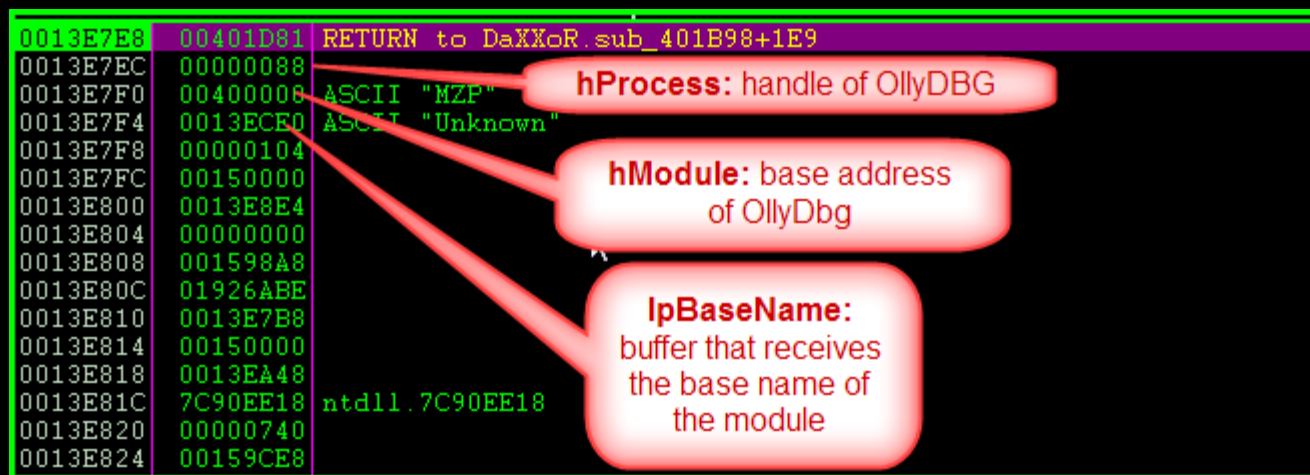
Nhấn Ctrl+F9 và **Follow in Dump** tại giá trị **lpModule**, ta có được như sau :

Address	Hex dump	ASCII
0013FDA8	00 00 40 00 01 00 00 00 0B 00 00 00 0B 00 00 00	@
0013FDB8	00 00 BF 00 F0 FD 13 00 6F 04 D7 77 88 00 00 00	...v8ý xw
0013FDC8	00 00 00 00 4F 00 00 00 A8 00 00 00 6E 06 13 00	...O... n
0013FDD8	51 00 00 00 E8 03 00 00 DC 27 98 00 01 00 00 00	Q...è...
0013FDE8	4C FE 13 00 EC 19 40 00 34 FE 13 00 A0 64 46 00	lp i @ . 4b . dF
0013FDF8	78 20 98 00 90 FE 13 00 EF 0F 93 00 00 00 00 00	x
0013FE08	EF 0F 93 00 78 20 98 00 28 FE 13 00 86 04 41 00	i x (A

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 nhớ 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 :

00401D6C	68 04010000	push	104	
00401D71	8D8D F8EEFFFF	lea	ecx,[local.1090]	
00401D77	51	push	ecx	
00401D78	FF75 C0	push	[local.16]	
00401D7B	FF75 DC	push	[local.9]	
00401D7E	FF55 B0	call	near [local.20]	<GetModuleBaseNameA>
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	

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



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 Ctrl + F9, quan sát cửa sổ Dump để xem kết quả có được :

Return to 00401D81 (DaXXoR.00401D81)									
Address	Hex dump								ASCII
0013ECE0	4F 4C 4C 59	44 42 47 2E	45 58 45 00	48 ED 13 00	OLLYDBG.EXE Hí				
0013ECF0	B7 2C 91 7C	00 3E 00 00	28 65 47 00	28 ED 13 00	...>... (eG. (i				
0013ED00	00 00 00 00	DE 9F 80 7C	00 00 40 00	28 65 47 00	...P...@. (eG.				
0013ED10	28 ED 13 00	F5 9F 80 7C	54 9B 47 00	00 00 40 00	(i.ö...TiG...@.				
0013ED20	00 00 8D 00	F5 A1 94 7C	50 EF 13 00	F4 B5 91 7C	...ö...P...ôm				
0013ED30	86 B6 91 7C	14 00 00 00	00 00 8D 00	01 00 00 00	!... ..				

0013E7E8	00401D81	RETURN to DaXXoR.sub_401B98+1E9
0013E7EC	00000088	
0013E7F0	00400000	ASCII "MZP"
0013E7F4	0013ECE0	ASCII "OLLYDBG.EXE"
0013E7F8	001598A8	

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 :

00401D81	8945 C4	mov [local.15],eax	
00401D84	FF75 DC	push [local.9]	[loc_401D84
00401D87	EB E02E0600	call <DaXXoR.CloseHandle>	CloseHandle

0013E7F8	00000088	hObject = 00000088 (window)
0013E7FC	00150000	
0013E800	0013E8E4	

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\CritSecOutOfMemoryEvent
00000058	Mutant	36.	001F0001			\BaseNamedObjects\CTF.LBES.MutexDefault
0000005C	Mutant	36.	001F0001			\BaseNamedObjects\CTF.Compart.MutexDefault
00000060	Mutant	36.	001F0001			\BaseNamedObjects\CTF.Asm.MutexDefault
00000064	Mutant	36.	001F0001			\BaseNamedObjects\CTF.Layouts.MutexDefault
00000068	Mutant	36.	001F0001			\BaseNamedObjects\CTF.TMD.MutexDefault
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 F4EDFF	lea	eax,[local.1155]	
00401D92	. 50	push	eax	
00401D93	. 8D95 F8EEFF	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 :

0045DACA	EB 01	jmp	short <DaXXoR.loc_45DACD>	
0045DACC	43	inc	ebx	loc_45DACC
0045DACD	33C0	xor	eax, eax	loc_45DACD
0045DACE	8A03	mov	al, byte ptr [ebx]	
0045DAD1	50	push	eax	Arg1
0045DAD2	E8 0D000000	call	<DaXXoR.__ltoupper>	DaXXoR.0045DAE4
0045DAD7	59	pop	ecx	
0045DAD8	8803	mov	byte ptr [ebx], al	
0045DADA	84C0	test	al, al	
0045DADC	75 EE	jnz	short <DaXXoR.loc_45DACC>	

0045DAED	C2	test	al, al	
Stack ds:[0013ECE0]=4F ('O')				
al=00				

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** :

0045DAD7	59	pop	ecx	
0045DAD8	8803	mov	byte ptr [ebx], al	
0045DADA	84C0	test	al, al	
0045DADC	75 EE	jnz	short <DaXXoR.loc_45DACC>	

0045DAED	C2	test	al, al	
al=4F ('O')				
Stack ds:[0013ECE0]=4F ('O')				
DaXXoR.lstrupr+18				
Address		Hex dump		

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 chữ thường sang chữ hoa. Trace ra khỏi hàm này và trace tiếp tới đây :

00401D9F	59	pop	ecx	
00401DA0	50	push	eax	
00401DA1	E8 F29C0500	call	<DaXXoR._strcmp>	
00401DA6	83C4 08	add	esp, 8	
00401DA9	85C0	test	eax, eax	
00401DAB	75 74	jnz	short <DaXXoR.loc_401E21>	
00401DAD	C745 F0 0100	mov	[local_81], 1	

0013E7F4	0013ECE0	s1 = "OLLYDBG.EXE"	
0013E7F8	0013EBDC	s2 = "OLLYDBG.EXE"	
0013E7FC	00150000		
0013E800	0013E8E4		

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	75 34	jnz	short <DaXXoR.loc_45BAE1>	
0045BAAD	84DB	test	bl,bl	
0045BAAF	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	75 26	jnz	short <DaXXoR.loc_45BAE1>	
0045BABB	84DB	test	bl,bl	
0045BABD	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	75 18	jnz	short <DaXXoR.loc_45BAE1>	
0045BAC9	84DB	test	bl,bl	
0045BACB	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	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	75 74	jnz	short <DaXXoR.loc_401E21>	
401DAD	C745 E0 0100	mov	[local.8],1	
401DB4	8B4D E4	mov	ecx,[local.7]	
401DB7	FFB48D FCEFF	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	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	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 truyền thêm tham số là :

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

00401DB4	8B4D E4	mov	ecx,[local.7]	
00401DB7	FFB48D FCEFF	push	dword ptr [ebp+ecx*4-1004]	
00401DBE	6A 00	push	0	
00401DC0	6A 01	push	1	
00401DC2	E8 31300600	call	<DaXXoR.OpenProcess>	ProcessId Inheritable = FALSE Access = TERMINATE OpenProcess
00401DC7	8945 DC	mov	[local.9],eax	
00401DC9	837D DC 88	jmp	short 013E7F0	

0013E7F0	00000001	Access = TERMINATE
0013E7F4	00000000	Inheritable = FALSE
0013E7F8	000001B0	ProcessId = 1B0
0013E7FC	00150000	

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

Registers (FPU)	
EAX	00000088
ECX	0013E7A5
EDX	7C90EB94 ntdll.KiFastSystemCallRet
EBX	009827D4
ESP	0013E7FC
EBP	0013FDE8
ESI	00982078
EDI	004664A0 offset <DaXXoR._cls_Unit2_TForm1>

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

00401DCE	74 3F	je	short <DaXXoR.loc_401E0F>	
00401DD0	6A 00	push	0	
00401DD2	FF75 DC	push	[local.9]	
00401DD5	E8 78300600	call	<DaXXoR.TerminateProcess>	ExitCode = 0 hProcess TerminateProcess
00401DDA	85C0	test	eax, eax	

0013E7F4	00000088	hProcess = 00000088 (window)
0013E7F8	00000000	ExitCode = 0
0013E7FC	00150000	
0013E800	0013E8E4	

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 ☺.

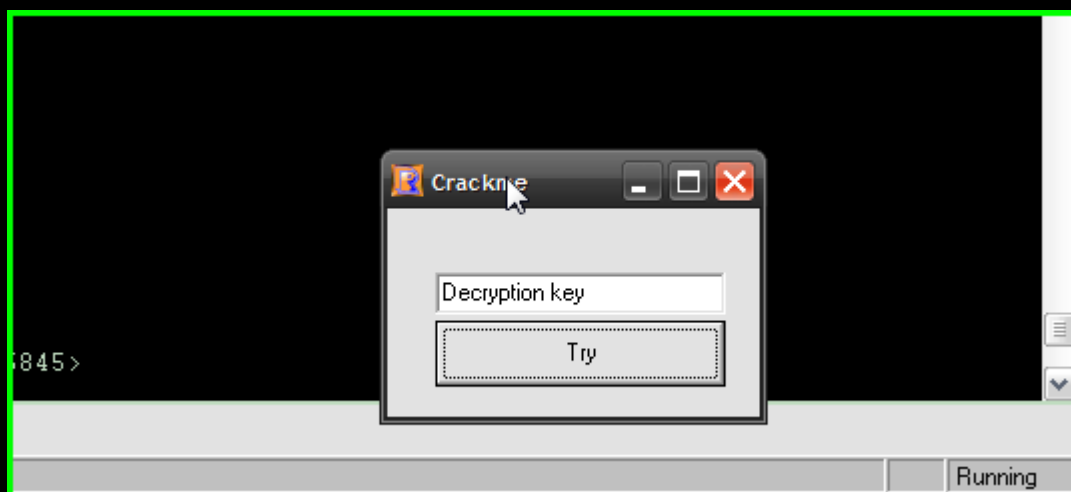
Với cách đầu tiên ta làm như sau, mở Olly lên và đặt BP tại hàm `OpenProcess` và nhấn F9 để thực thi :

7C8309E1	8BFF	mov	edi,edi	<DeXXoR _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	
7C830A27	3BC6	cmp	eax,esi	

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 :

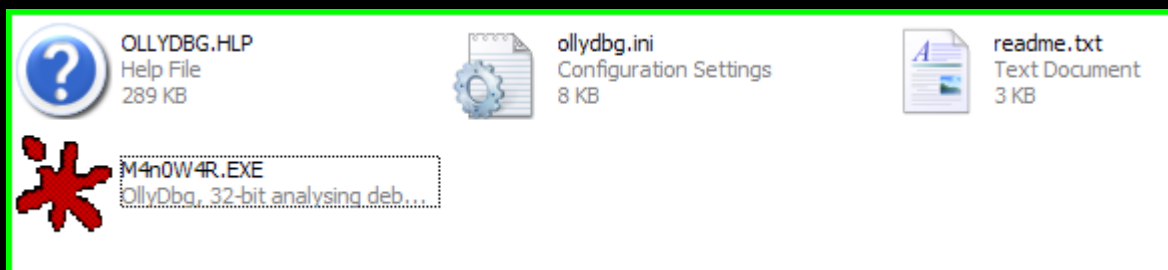


Tuy nhiên cách patch này không được khuyến khích vì nó tác động trực tiếp vào hàm API, ảnh hưởng tới quá trình sử dụng hàm sau này.

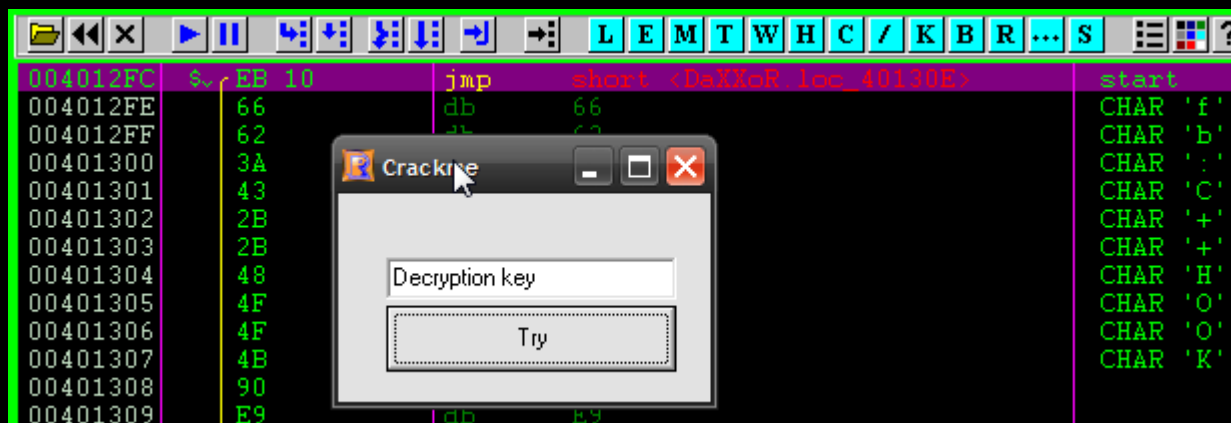
Cách thứ hai đơn giản hơn, ta patch lệnh JNZ thành JMP :

00401DA6	83C4 08	add	esp,8	
00401DA9	85C0	test	eax, eax	
00401DAB	EB 74	jmp	short <DaXXoR.loc_401E21>	
00401DAD	C745 E0 0100	mov	[local.8],1	
00401DB4	8B4D E4	mov	ecx,[local.7]	
00401DB7	FFB48D FCEFF	push	dword ptr [ebp+ecx*4-1004]	
00401DBE	6A 00	push	0	
00401DC0	6A 01	push	1	
00401DC2	E8 31300600	call	<DaXXoR.OpenProcess>	ProcessId
00401DC7	8945 DC	mov	[local.9],eax	Inheritable = FALSE
00401DCA	837D DC 00	cmp	[local.9],0	Access = TERMINATE
00401DCE	74 3F	je	short <DaXXoR.loc_401E0F>	OpenProcess
00401DD0	6A 00	push	0	
00401DD2	FF75 DC	push	[local.9]	ExitCode = 0
00401DD5	E8 78300600	call	<DaXXoR.TerminateProcess>	hProcess
				TerminateProcess

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, dqtn, 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 Rogers** 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](mailto:kienmanowar[at]reaonline.net)