2009

[Cracking with OllyDbg]

Based on OllyDbg tuts of Ricardo Narvaja (CrackLatinos Team)

www.reaonline.net

kienmanowar

20/01/2010

Mục Lục

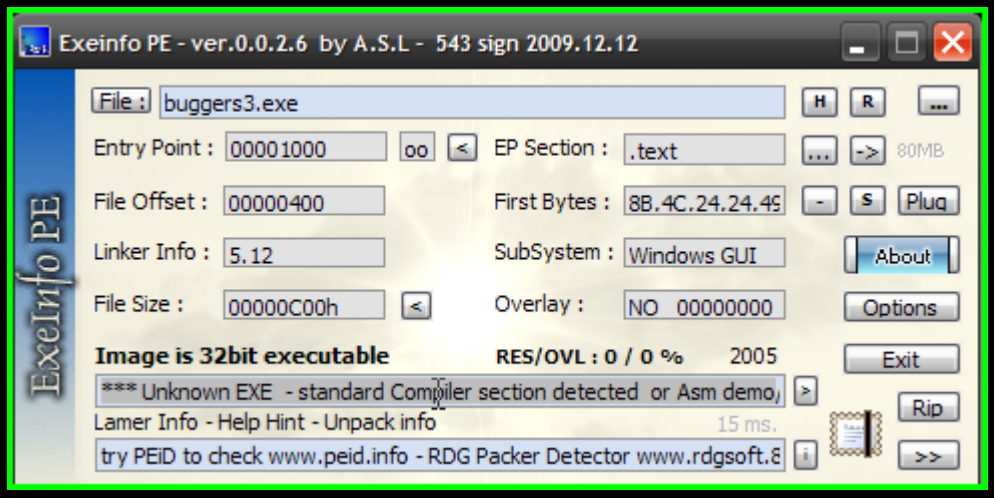
I. Giới thiệu chung

Chào các bạn, một tuần dài đã trôi qua, vợ và con về ngoại cả rồi, còn mỗi một mình tôi ngồi buồn mà chẳng biết làm gì, đành viết lách để giết thời gian vậy. Trong bài 21 này chúng ta sẽ tiếp tục nghiên cứu thêm một số kĩ thuật Anti-Debug khác. Target dùng để minh họa trong bài viết này là buggers3.exe, được chỉnh sửa bởi chính bác Ricardo. Theo như giới thiệu thì crackme này sẽ sử dụng các hàm API khác để detect process name, bao gồm việc phát hiện tên process lẫn tên class của OllyDbg. N0w let's g0……☺

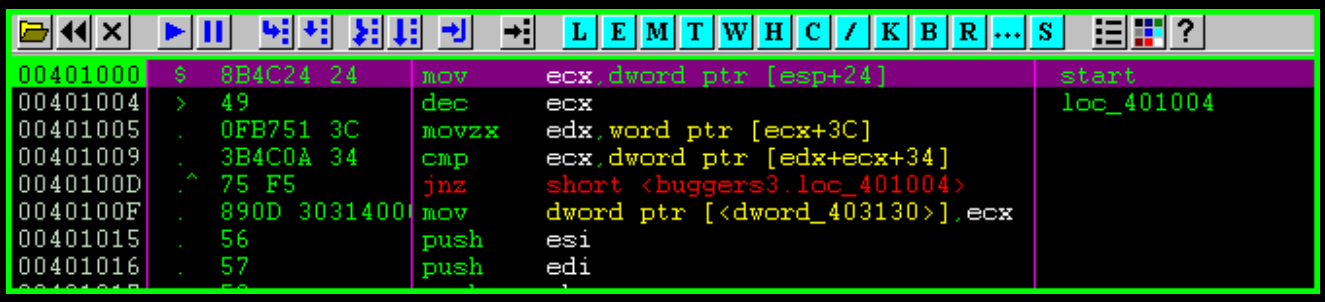
II. Phân tích và xử lý target

1. Phân tích buggers3.exe

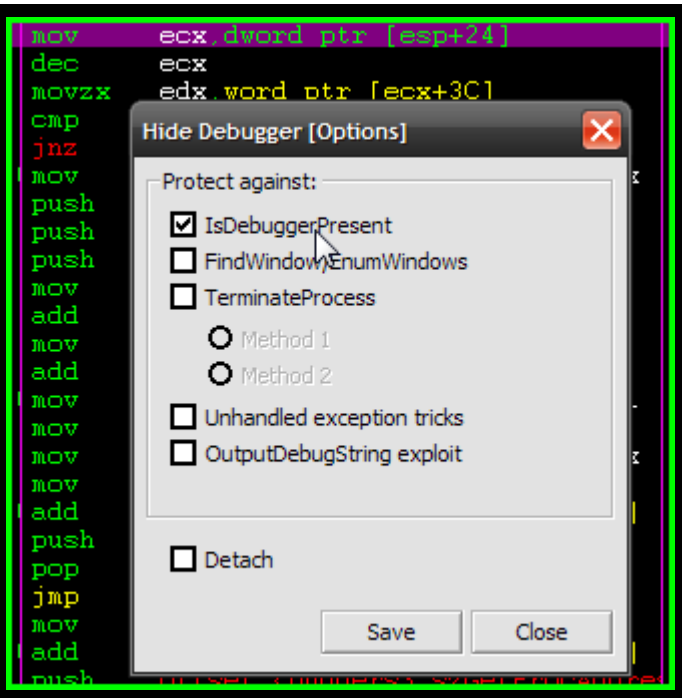
Trước tiên ta check qua info của nó :



Chà Unknown EXE ☺, có vẻ hóc búa rồi đây! Ta load vào Olly cái đã, ở đây tôi vẫn sử dụng OllyDbg nguyên bản nhé, không đổi tên gì hết. Có như vậy chúng ta mới phân tích để xem cơ chế Anti-Debug của target này như thế nào. Khi load vào OllyDbg, chúng ta dừng lại tại EP như sau :

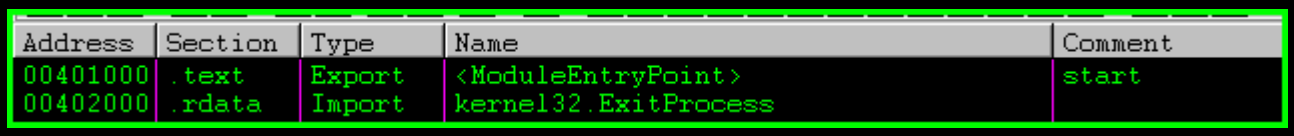


Ta sử dụng Plugin HideDebugger để bypass cơ chế Anti bằng `IsDebuggerPresent`. Ở đây tôi không quan tâm là target có sử dụng cơ chế này hay không, mục đích đơn giản là ta đã hiểu biết cơ chế này rồi, nếu như target này có sử dụng thì ta dùng plugin để bypass luôn cho đỡ mất thời gian :



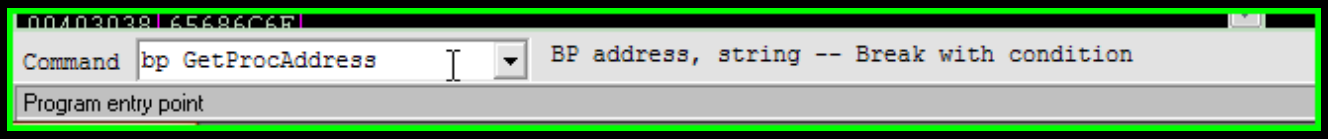
Ta Save lại và sau đó restart Olly để thiết lập trên có hiệu lực. Tiếp theo, sử dụng Task Manager hay Process Explorer để xem danh sách các process đang chạy (trong đó có OLLYDBG.exe của chúng ta) :

OK .. như các bạn thấy trên hình là tên nguyên gốc của Olly, tôi không có thay đổi gì hết. Ta quay trở lại màn hình chính của Olly và tìm kiếm danh sách các hàm API được target sử dụng. Nhấn phím tắt là **Ctrl + N** :

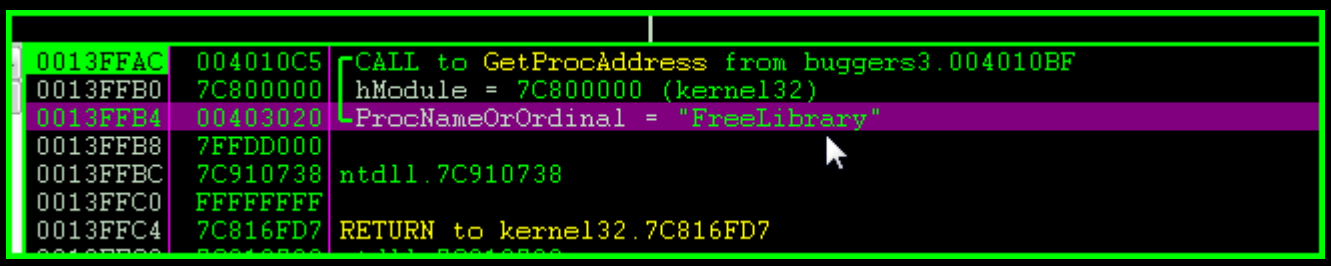


Address	Section	Type	Name	Comment
00401000	.text	Export	<ModuleEntryPoint>	start
00402000	.rdata	Import	kernel32.ExitProcess	

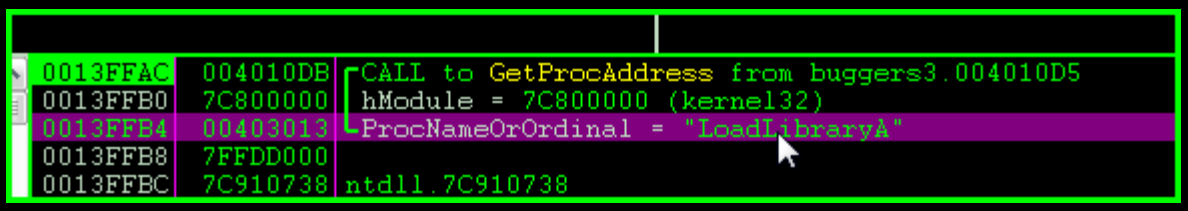
Kết quả hơi ngạc nhiên, có mỗi hàm `ExitProcess`. Ở phần 20 trước, ta còn thấy crackme sử dụng hàm `GetProcAddress` để mà mò theo, chứ còn ở target này thì không thấy được liệt kê trong danh sách ☺. Giờ không có thì làm thế nào đây, thôi thì cứ đặt BP tại hàm `GetProcAddress` thử xem thế nào :



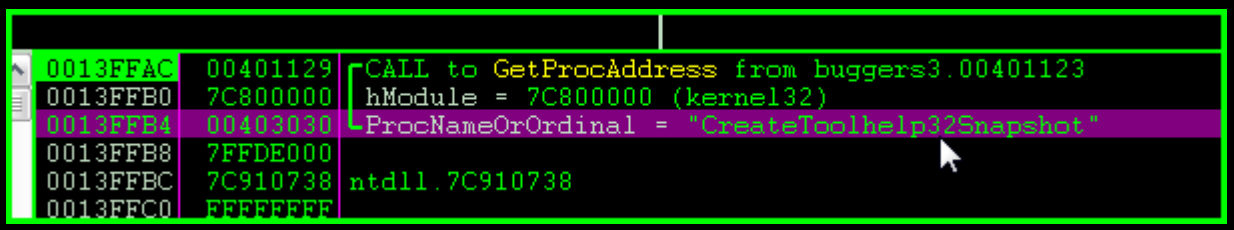
Sau khi đặt xong BP, nhấn **F9** để thực thi target, Olly sẽ break tương tự như hình :



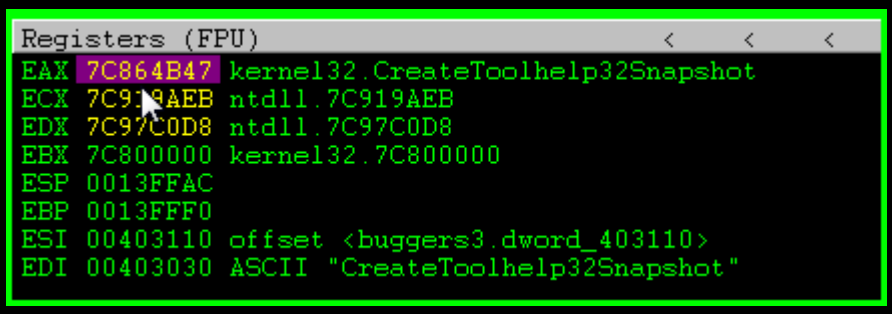
Mục đích của GetProcAddress được sử dụng làm gì thì các bạn cũng đã hiểu rồi, ở đây ta thấy nó đang chuẩn bị lấy địa chỉ của hàm FreeLibrary. Kiểm tra thông tin về hàm này thấy không có gì quan trọng, ta bỏ qua và tiếp tục nhấn **F9** :



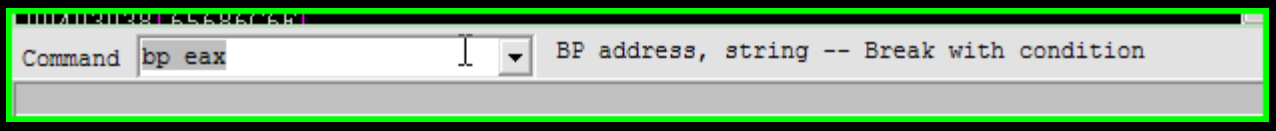
Tiếp tục **F9** cho tới khi ta nhận được thông tin sau :



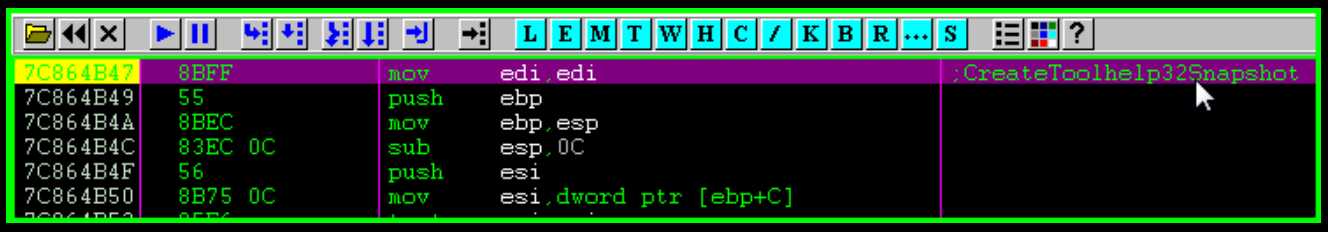
Hơi nghi ngờ hàm này! Tại sao tôi lại nghi ngờ nó, đơn giản tôi thấy có từ *Snapshot* trong tên hàm, thường thì như các bạn hay thấy là từ snapshot liên quan tới việc chụp ảnh, nhưng trong ngữ nghĩa của hệ thống thì có thể là nó được dùng để *"chụp"* các thông tin gì đó về hệ thống mà ta chưa biết rõ ngay lúc này. Do đó, ta cứ đặt BP tại hàm này cho chắc. Nhấn **Ctrl + F9 (execute till return),** quan sát thanh ghi EAX ở cửa sổ Registers :



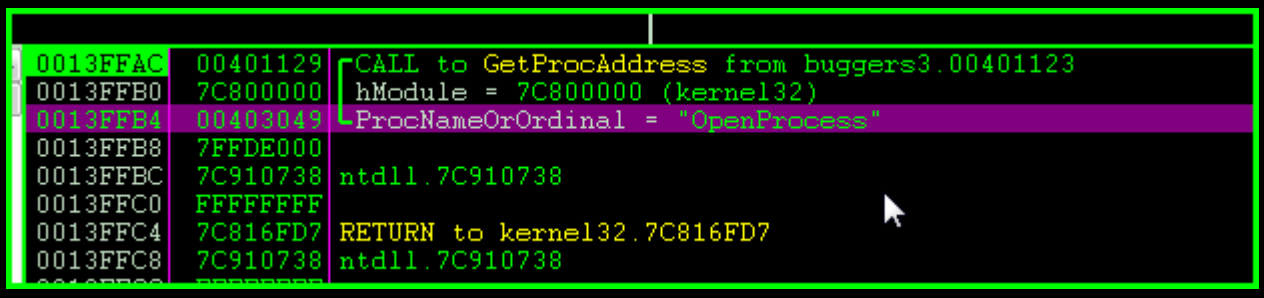
Ta thấy, EAX đang lưu địa chỉ của hàm CreateToolhelp32Snapshot . Đặt BP tại hàm này bằng cách :



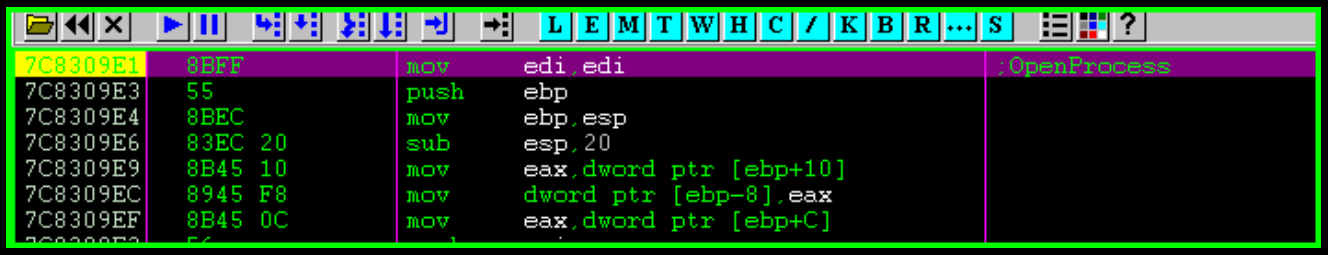
Sau đó ta thêm comment và label để ghi nhớ hàm này. Mục đích để phân biệt các địa chỉ ta đặt BP :



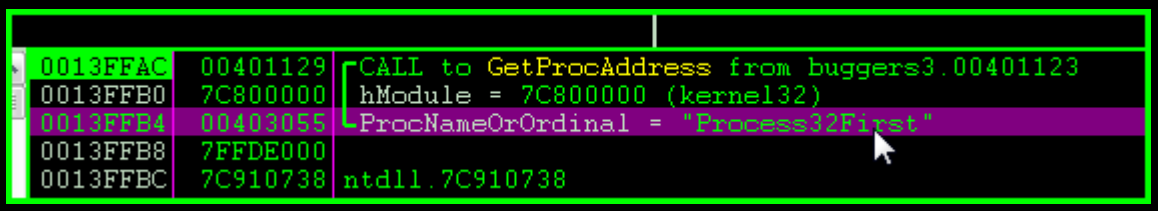
Tiếp tục nhấn **F9** và quan sát cửa sổ Stack :



Ta thấy có `OpenProcess`, chức năng hàm này thế nào thì bài trước tôi giới thiệu rồi. Ta cũng tiến hành đặt BP ở hàm này bằng cách tương tự như trên :

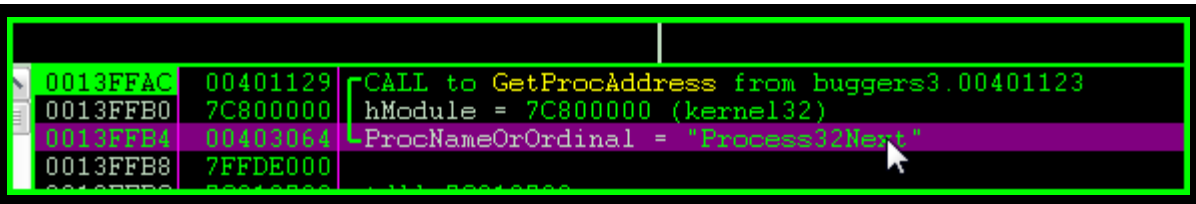


Tiếp tục nhấn **F9** :

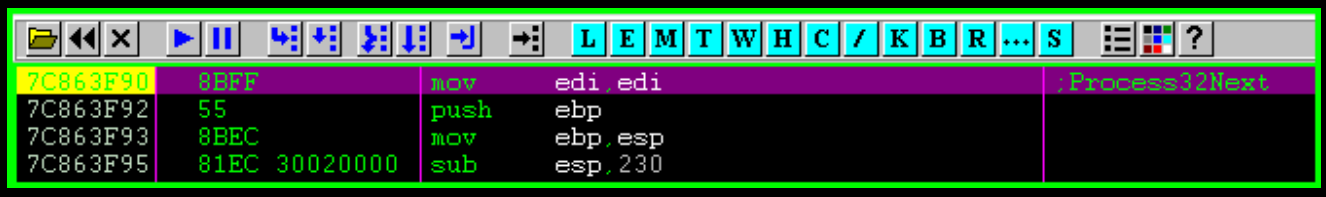


Hàm tiếp theo `Process32First`, ta cũng đặt BP tại hàm này :

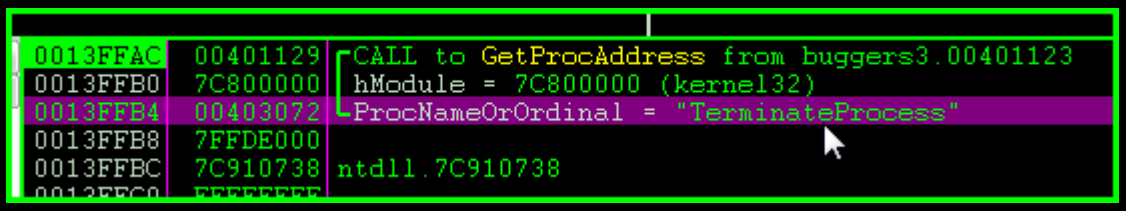
Khai thác tiếp thông tin :



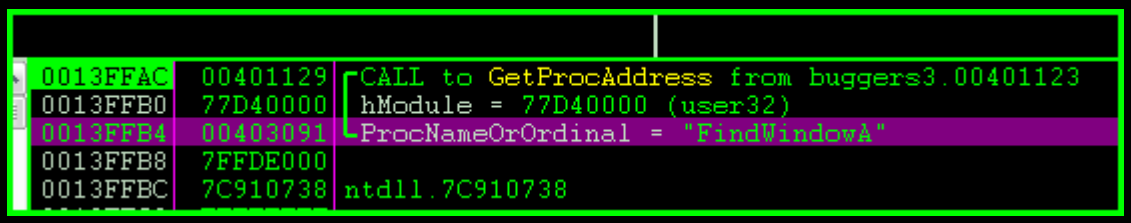
Kết quả ta có là Process32Next, target này dùng nhiều API mới lạ quá ☺. Ta đặt BP tại hàm này :



Không biết còn bao nhiêu hàm nữa đây ☹, **F9** :



Ái chà, tìm địa chỉ của hàm TerminateProcess kìa lolz. Hàm này ở bài 20 các bạn đã biết nó dùng để làm gì rồi, do đó khỏi cần đặt BP tại hàm này. Ta tiếp tục nhấn **F9** :



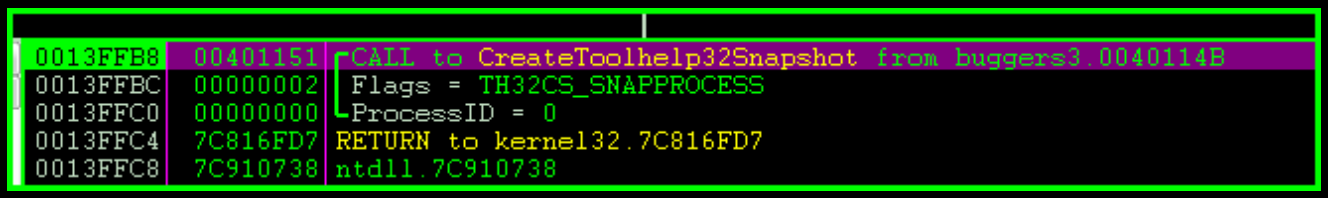
Hàm tiếp theo như ta thấy là FindWindowA, chắc là nó dùng để tìm kiếm thông tin gì đó. Ta đặt BP tại hàm này :

Tổng kết toàn bộ từ đầu đến giờ chúng ta đã thiết lập những BP sau :

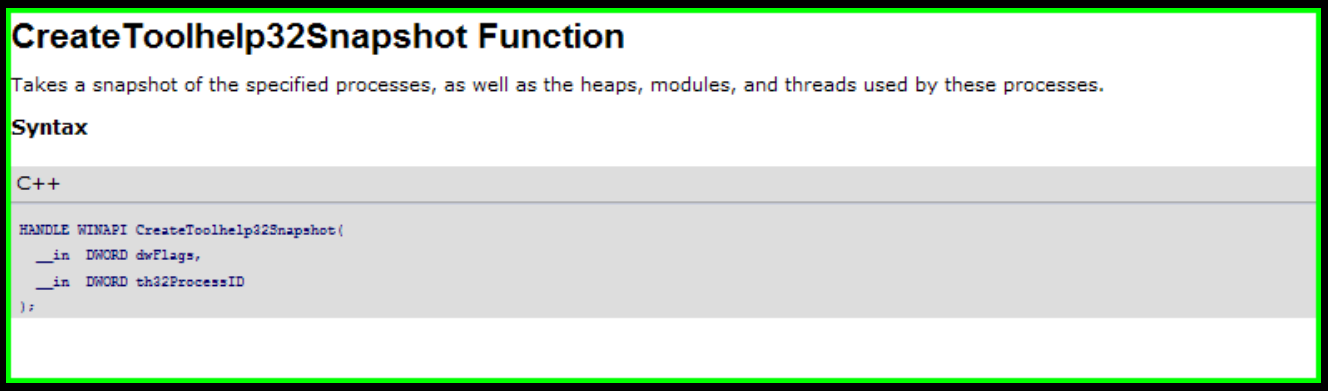
Address	Module	Active	Disassembly	Comment
77D54482	user32	Always	mov edi,edi	;FindWindowA
7C80ADA0	kernel32	Always	mov edi,edi	
7C8309E1	kernel32	Always	mov edi,edi	;OpenProcess
7C863E1D	kernel32	Always	mov edi,edi	;Process32First
7C863F90	kernel32	Always	mov edi,edi	;Process32Next
7C864B47	kernel32	Always	mov edi,edi	;CreateToolhelp32Snapshot

F9 thêm một lần nữa, lần này ta sẽ break tại hàm `CreateToolhelp32Snapshot` :

Thông tin trên cửa sổ Stack :



Tìm kiếm thông tin về hàm này :



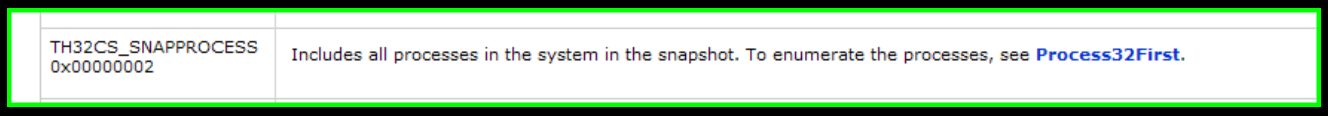
CreateToolhelp32Snapshot Function

Takes a snapshot of the specified processes, as well as the heaps, modules, and threads used by these processes.

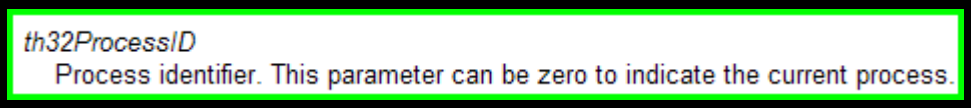
Syntax

C++

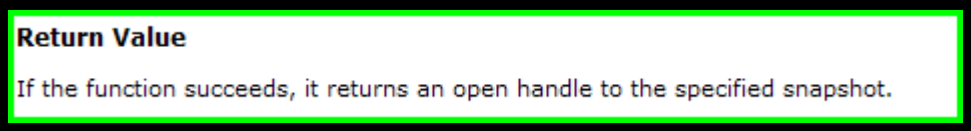
```
HANDLE WINAPI CreateToolhelp32Snapshot(
  __in  DWORD dwFlags,
  __in  DWORD th32ProcessID
);
```



TH32CS_SNAPPROCESS 0x00000002	Includes all processes in the system in the snapshot. To enumerate the processes, see **Process32First**.



th32ProcessID
Process identifier. This parameter can be zero to indicate the current process.



Return Value

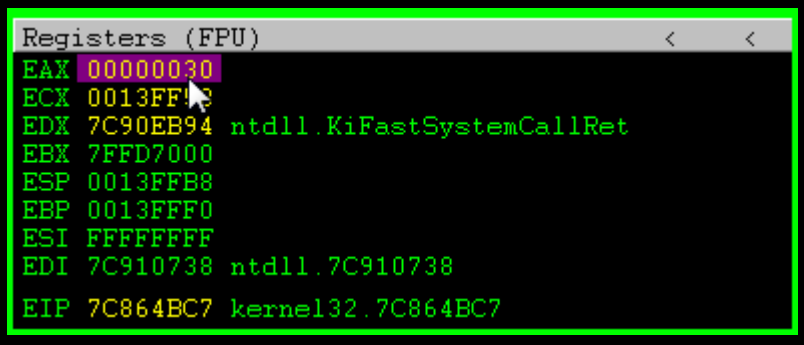
If the function succeeds, it returns an open handle to the specified snapshot.

Theo thông tin có được ở trên, ta nắm được mục đích sử dụng của hàm này như sau : Hàm này được sử dụng để chụp nhanh thông tin của các Process, nó nhận hai tham số truyền vào là `dwFlags` và `th32ProcessID`. Sau khi thực hiện, hàm sẽ trả về một handle (gọi là handle của snapshot), mà handle này sẽ được sử dụng bởi các hàm khác. Cụ thể với target này ta có :

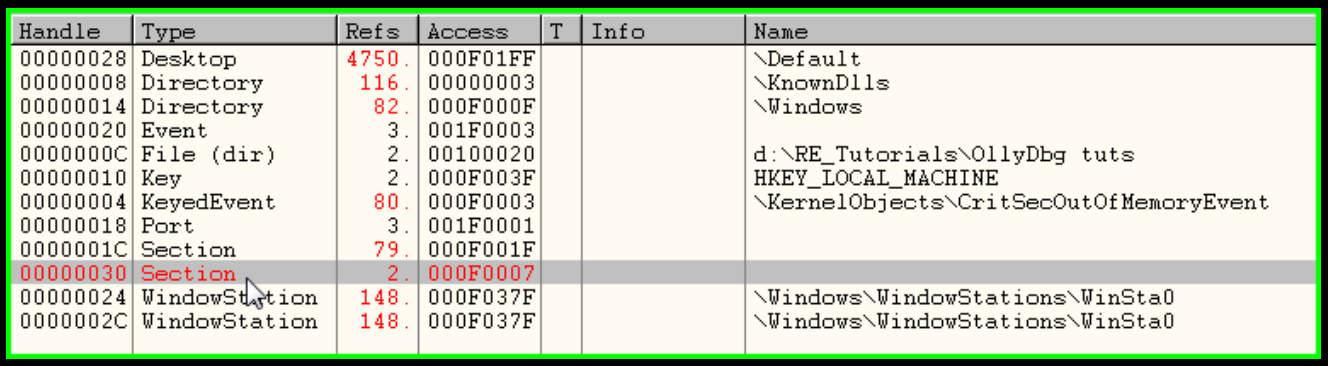
a. `dwFlags = TH32CS_SNAPPROCESS` ← có nghĩa là tất cả các process đang chạy trên hệ thống. Sau đó sẽ sử dụng hàm `Process32First` để lấy tiếp thông tin cụ thể.
b. `th32ProcessID = 0` ← có nghĩa là nó muốn lấy thông tin của Process hiện tại.

Để đơn giản hơn ta hiểu như kiểu ta chụp một bức ảnh, trong bức ảnh đó có rất nhiều người - đại diện cho các process trên hệ thống. Sau khi chụp xong, ta đánh một mã hay một con số cho bức ảnh ta chụp được để sau này ta cần sử dụng lại bức ảnh thì sẽ dễ dàng hơn, con số này tương ứng với handle.

OK, sau khi phân tích về hàm xong ta nhấn **Ctrl + F9** và kiểm tra kết quả trả về ở thanh ghi EAX :

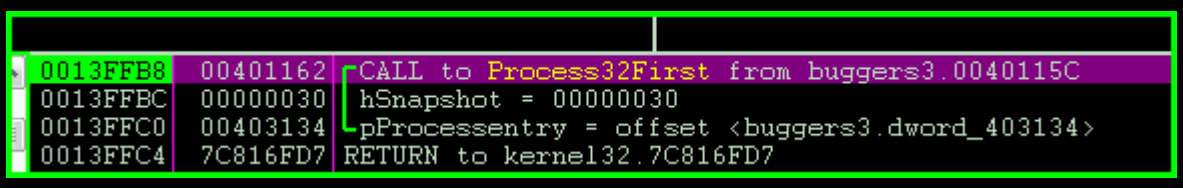


EAX có giá trị là `0x30`, đó chính là handle của snapshot. Ta kiểm tra cửa sổ Handles của Olly xem có thông tin gì về nó không :



Handle	Type	Refs	Access	T	Info	Name
00000028	Desktop	4750.	000F01FF			\Default
00000008	Directory	116.	00000003			\KnownDlls
00000014	Directory	82.	000F000F			\Windows
00000020	Event	3.	001F0003			
0000000C	File (dir)	2.	00100020			d:\RE_Tutorials\OllyDbg tuts
00000010	Key	2.	000F003F			HKEY_LOCAL_MACHINE
00000004	KeyedEvent	80.	000F0003			\KernelObjects\CritSecOutOfMemoryEvent
00000018	Port	3.	001F0001			
0000001C	Section	79.	000F001F			
00000030	Section	2.	000F0007			
00000024	WindowStation	148.	000F037F			\Windows\WindowStations\WinSta0
0000002C	WindowStation	148.	000F037F			\Windows\WindowStations\WinSta0

Như các bạn thấy trên hình, thông tin mang lại cho chúng ta không nhiều và hơi trừu tượng. Nhưng tóm lại ta hiểu là chúng ta đã có được handle của snapshot - mà snapshot này bao gồm danh sách các process. Ta nhấn **F9** để thực thi target, Olly sẽ break tại :



Như thông tin ta đọc về hàm CreateToolhelp32Snapshot thì cờ khi dwFlags = TH32CS_SNAPPROCESS, để liệt kê các process ta cần sử dụng hàm Process32First. Ta tìm hiểu về hàm này :

Process32First

Retrieves information about the first process encountered in a system snapshot.

```
BOOL WINAPI Process32First(HANDLE hSnapshot, LPPROCESSENTRY32 lppe);
```

Parameters

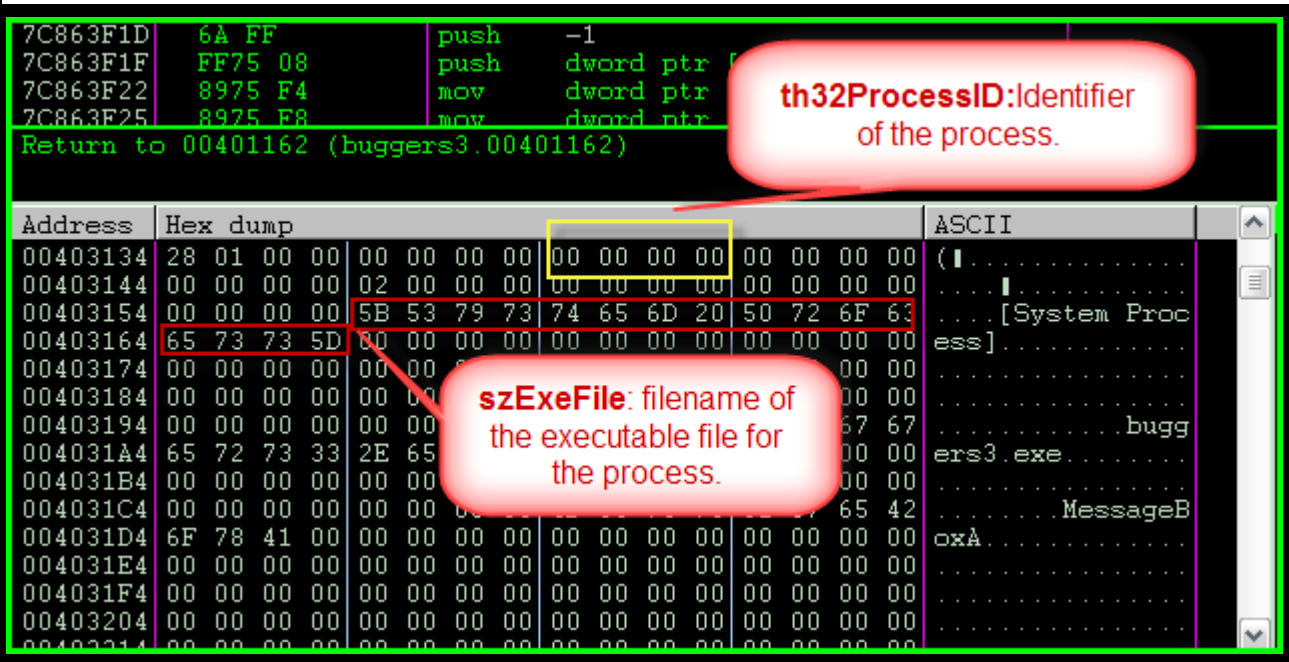
hSnapshot
Handle of the snapshot returned from a previous call to the CreateToolhelp32Snapshot function.

lppe
Address of a PROCESSENTRY32 structure.

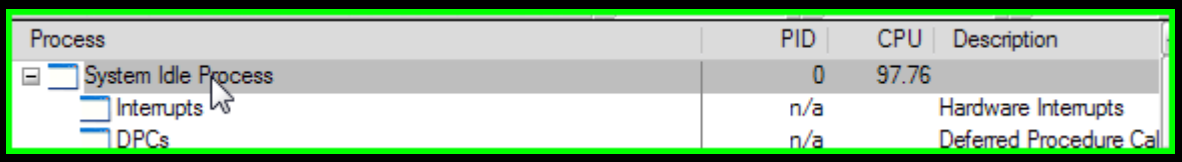
Ta thấy hàm này dùng để lấy ra thông tin của Process đầu tiên dựa vào handle của snapshot có được thông qua hàm CreateToolhelp32Snapshot, sau đó thông tin của process sẽ được lưu vào một cấu trúc có tên gọi là PROCESSENTRY32.Căn cứ vào thông tin từ cửa sổ Stack ta biết được hSnapshot = 00000030 và vùng nhớ để lưu thông tin về Process là pProcessentry = offset <buggers3.dword_403134>.Tìm hiểu thông tin về cấu trúc dùng để lưu thông tin về Process :

```
typedef struct tagPROCESSENTRY32 {
    DWORD dwSize;
    DWORD cntUsage;
    DWORD th32ProcessID;
    DWORD th32DefaultHeapID;
    DWORD th32ModuleID;
    DWORD cntThreads;
    DWORD th32ParentProcessID;
    LONG  pcPriClassBase;
    DWORD dwFlags;
    char szExeFile[MAX_PATH];
} PROCESSENTRY32;
typedef PROCESSENTRY32 *  PPROCESSENTRY32;
typedef PROCESSENTRY32 *  LPPROCESSENTRY32;
```

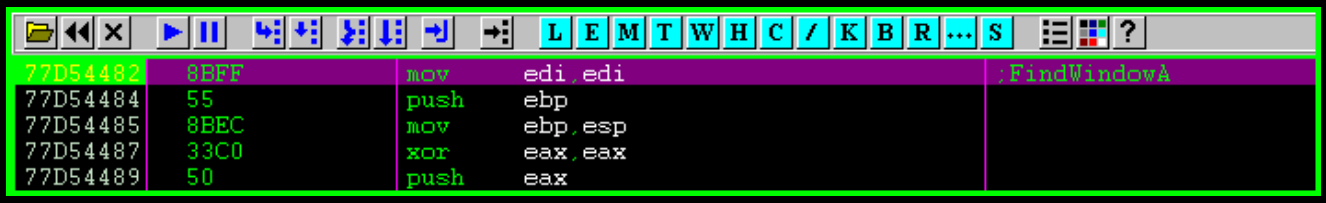
Follow in Dump tại vùng nhớ 0x00403134, nơi dùng để chứa thông tin. Nhấn **Ctrl+F9** và quan sát kết quả tại vùng nhớ này :

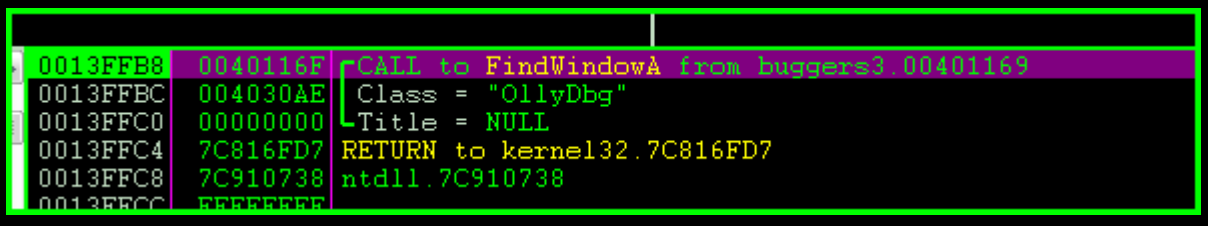


Ta quan tâm tới hai thành phần của cấu trúc này là `szExeFile` và `th32ProcessID`. Như các bạn thấy trên hình, process đầu tiên của chúng ta có tên là System Process, và PID của nó là `0x0`. Ta dùng ProcessExplorer để kiểm tra xem nó trùng với process nào :



Kết quả rất rõ ràng và chính xác. Ta tiếp tục nhấn **F9**, Olly sẽ break tại đây :



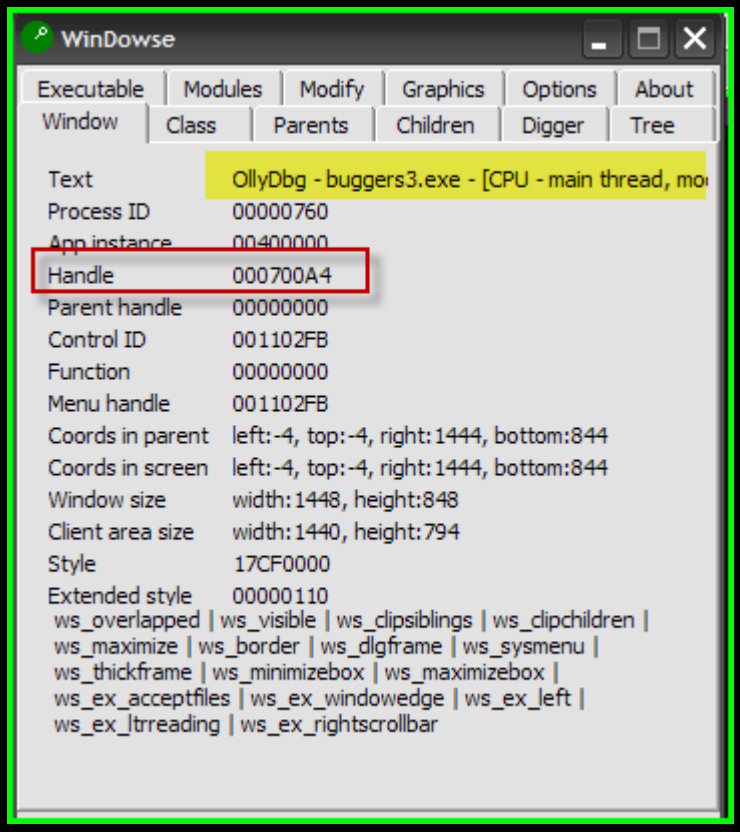


Khà khà, mấu chốt vấn đề bắt đầu tại đây. Ta thấy target sử dụng hàm API `FindWindowA` để tìm kiếm thông tin. Thông tin cụ thể về hàm này như sau :

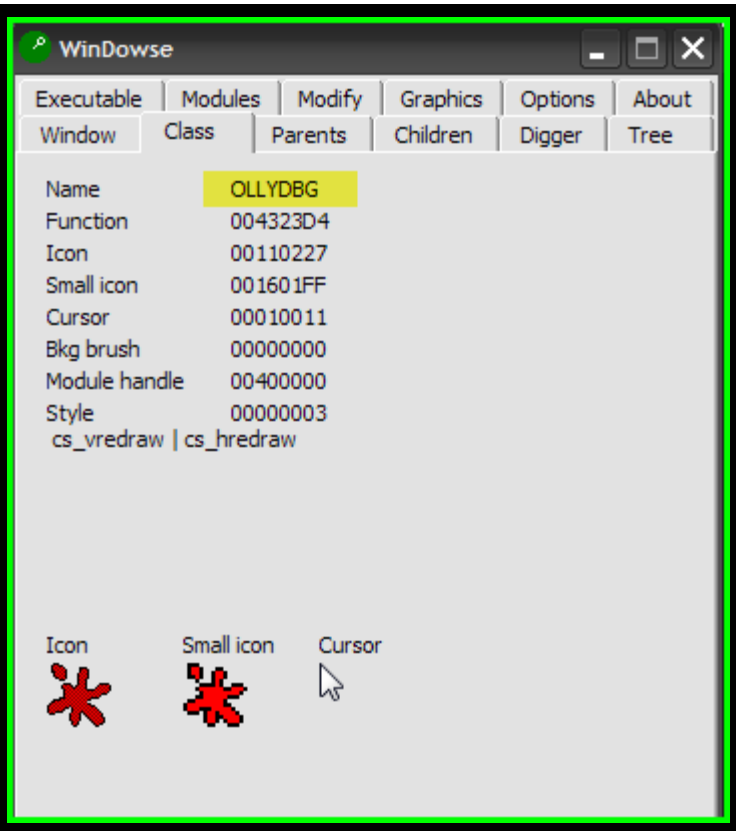
The **FindWindow** function retrieves the handle to the top-level window whose class name and window name match the specified strings. This function does not search child windows.

```
HWND FindWindow(
  LPCTSTR lpClassName,        // pointer to class name
  LPCTSTR lpWindowName        // pointer to window name
);
```

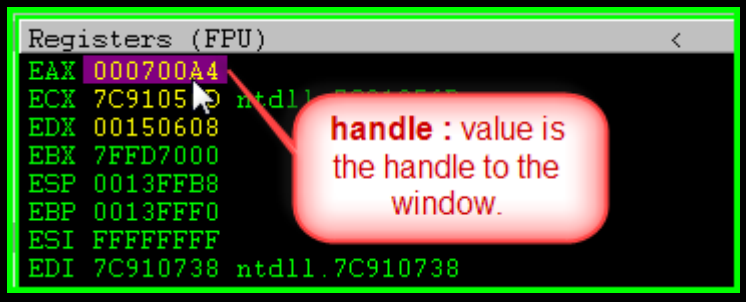
Như vậy, hàm này dùng để tìm ra handle của cửa sổ (top-level) mà có tên cửa sổ hoặc tên class trùng với thông tin mà nó chỉ định. Cụ thể với target của chúng ta, nó mượn hàm này để tìm kiếm thông tin về Class có tên là OllyDbg (`0013FFBC  004030AE |Class = "OllyDbg"`). Trước khi đi tiếp chúng ta dừng lại một chút để sử dụng một chương trình tìm kiếm thông tin về các window, đó là chương trình Greatis WinDowse - Advanced Windows Analyser. Tôi có kèm theo trong bài viết này. Ta cài đặt và chạy chương trình này, sau đó tìm kiếm thông tin liên quan tới Olly :



Chuyển qua tab Class :



Kết quả cho ta thấy Class name là OLLYDBG. Giờ ta quay lại Olly và nhấn **Ctrl+F9** để thực hiện hàm `FindWindowA`, đồng thời quan sát giá trị của thanh ghi EAX tại cửa sổ Registers :



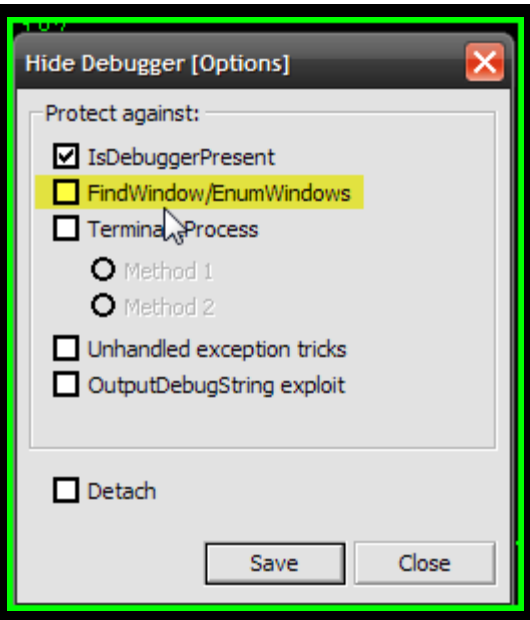
Kết quả này trùng với giá trị mà chương trình WinDowse thu thập được. Ta nhấn **F8** trace qua lệnh `RETN` để trở về code chính của target, quan sát xem nó sẽ làm gì với giá trị handle có được :

```
00401162   .   6A 00          push    0
00401164   .   68 AE304000    push    offset <buggers3.szOllyDbg>      ASCII "OllyDbg"
00401169   .   FF15 2831400(  call    near dword ptr [<dword_403128>]  <user32.FindWindowA>
0040116F   .   83F8 00        cmp     eax,0
00401172   .   0BC0           or      eax,eax
00401174   .v  75 04          jnz     short <buggers3.loc_40117A>
00401176   .v  7C 27          jl      short <buggers3.loc_40119F>
00401178   .v  EB 25          jmp     short <buggers3.loc_40119F>
0040117A   >   50             push    eax                              loc_40117A
0040117B   .   56             push    esi
0040117C   .   57             push    edi
0040117D   .   BF 01000000    mov     edi,1
00401182   .   BE 2C314000    mov     esi,offset <buggers3.dword_4031
00401187   >   FF36           push    dword ptr [esi]                  loc_401187
00401189   .   FF15 0C31400(  call    near dword ptr [<dword_40310C>]  kernel32.FreeLibrary
0040118F   .   83C6 04        add     esi,4
00401192   .   4F             dec     edi
00401193   .^  75 F2          jnz     short <buggers3.loc_401187>
00401195   >   5F             pop     edi                              loc_401195
00401196   .   5E             pop     esi
00401197   .   58             pop     eax
00401198   .   6A 00          push    0                                ExitCode = 0
0040119A   .   E8 57000000    call    <buggers3.ExitProcess>           ExitProcess
0040119F   >   68 B6304000    push    offset <buggers3.szOLLYDBG_EXE>  loc_40119F
```

Code thể hiện rất rõ ràng, EAX lưu giá trị handle có được sau quá trình tìm kiếm, nó được đem so sánh với 0. Nếu như kết quả tìm kiếm trả về cho thanh ghi EAX giá trị là 0 thì CMP sẽ kiểm tra và bật cờ ZF thành 1. Lệnh OR sau đó sẽ kiểm tra xem EAX có bằng 0 hay không, nếu bằng thì vẫn giữ nguyên cờ ZF và lệnh nhảy JNZ bên dưới sẽ không thực hiện. Nhưng lúc này EAX đang giữ giá trị handle <> 0, cho nên lệnh nhảy JNZ sẽ thực hiện :

```
00401174   .v  75 04          jnz     short <buggers3.loc_40117A>
00401176   .v  7C 27          jl      short <buggers3.loc_40119F>
00401178   .v  EB 25          jmp     short <buggers3.loc_40119F>
0040117A   >   50             push    eax                              loc_40117A
0040117B   .   56             push    esi
0040117C   .   57             push    edi
0040117D   .   BF 01000000    mov     edi,1
00401182   .   BE 2C314000    mov     esi,offset <buggers3.dword_4031
00401187   >   FF36           push    dword ptr [esi]                  loc_401187
00401189   .   FF15 0C31400(  call    near dword ptr [<dword_40310C>]  kernel32.FreeLibrary
0040118F   .   83C6 04        add     esi,4
00401192   .   4F             dec     edi
00401193   .^  75 F2          jnz     short <buggers3.loc_401187>
00401195   >   5F             pop     edi                              loc_401195
00401196   .   5E             pop     esi
00401197   .   58             pop     eax
00401198   .   6A 00          push    0                                ExitCode = 0
0040119A   .   E8 57000000    call    <buggers3.ExitProcess>           ExitProcess
0040119F   >   68 B6304000    push    offset <buggers3.szOLLYDBG_EXE>  loc_40119F
```

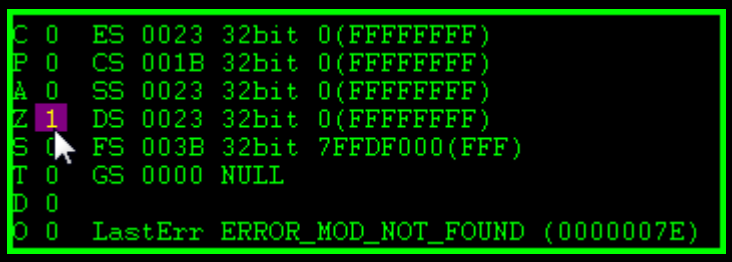
Nhìn vào đoạn code này, điều mà ta mong muốn là lệnh nhảy sẽ không thực hiện vì nếu nó thực hiện nó sẽ tới đoạn code gọi tới hàm API `ExitProcess` như bạn đang thấy trên hình. Mà để nó không thực hiện thì kết quả trả về của hàm `FindWindowA` phải là thanh ghi EAX có giá trị 0x0. Như vậy trong trường hợp này ta phải tìm cách nào đó để thay đổi Class Name của Olly window. Nếu để ý ở plugin HideDebugger các bạn sẽ thấy có một option cho phép ta vượt qua kiểu Anti-Debug này :



Tuy nhiên, ta sẽ không chọn option này vào thời điểm này. Vì nếu ta chọn thì phải restart lại OllyDbg mới có hiệu lực. Do đó ta bypass bằng tay, để không cho lệnh nhảy thực hiện thì có hai cách :

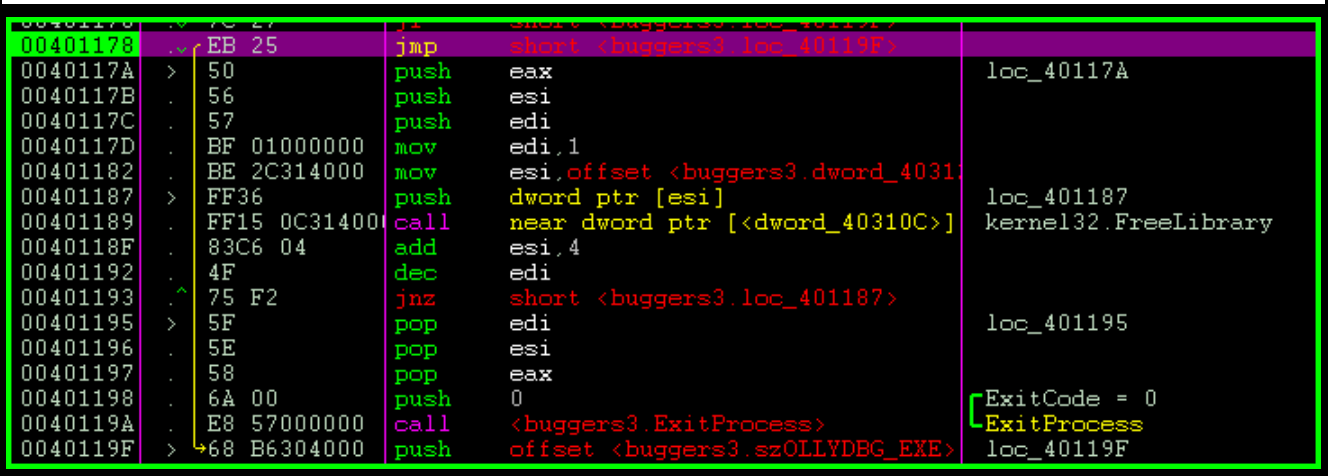
1. Fix cứng bằng cách patch lệnh JNZ thành JMP.
2. Fix tạm thời bằng cách thay đổi giá trị cờ ZF.

Ở đây tôi chọn cách 2, nhấn đúp chuột vào cờ ZF để thay đổi giá trị cờ này :

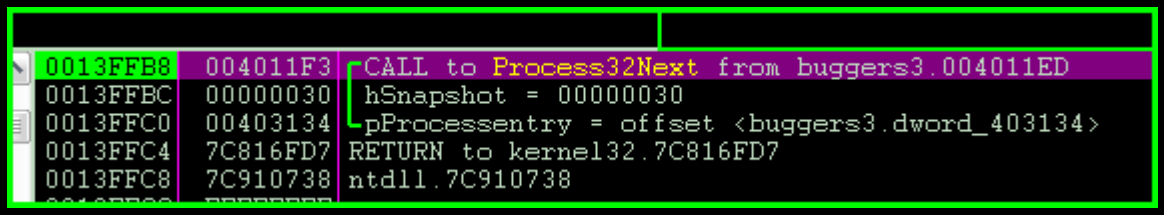


Quan sát cửa sổ code, ta thấy lệnh nhảy sẽ không được thực hiện :

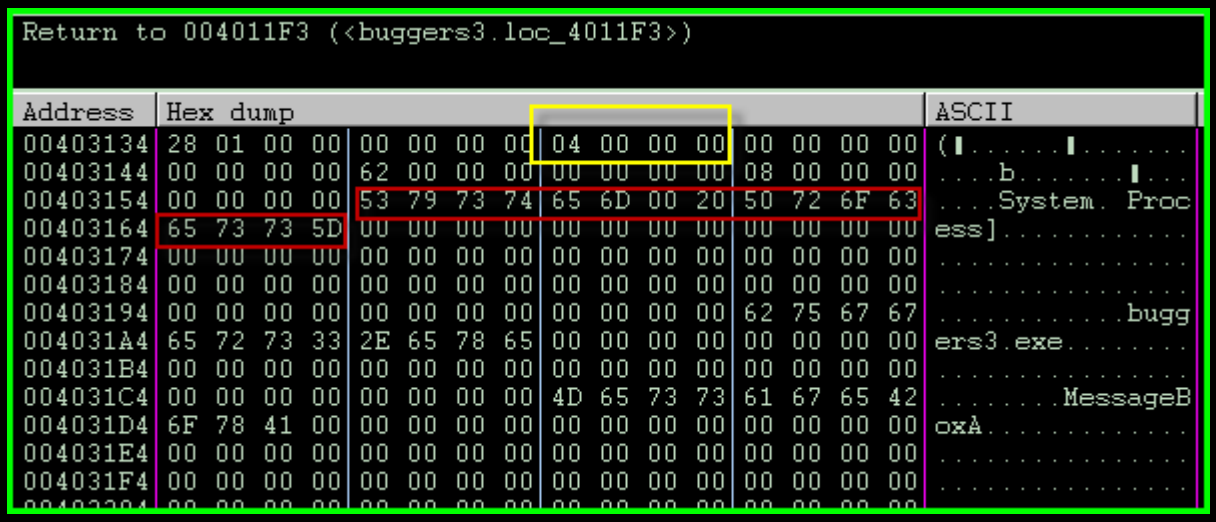
Nhấn **F8** để trace tới lệnh JMP, lệnh này sẽ nhảy qua lời gọi hàm `ExitProcess` :



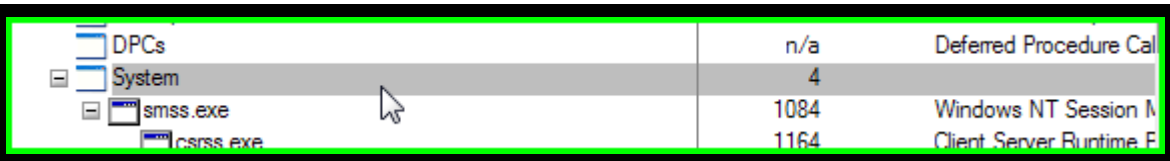
OK … vậy là tôi và các bạn vừa bypass qua cơ chế Anti-Debug sử dụng hàm FindWindowA. Không lẽ chỉ có mỗi cơ chế này, mấy cái hàm liên quan tới Process phía trên chỉ để làm cảnh thôi sao? Nghi ngờ quá, ta tiếp tục nhấn **F9** để thực thi chương trình :



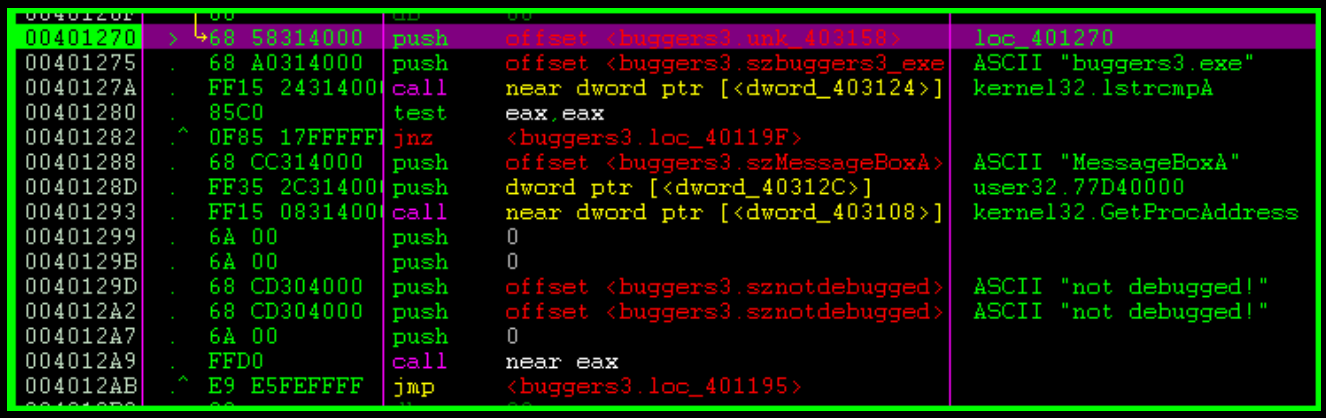
Olly break và trên cửa sổ Stack ta có thông tin như trên. Target sử dụng tiếp hàm API Process32Next để tìm kiếm thông tin về các process tiếp theo. Thông tin về process tiếp tục được lưu vào vùng buffer 0x00403134. Nhấn **Ctrl + F9** để thực hiện hàm này, quan sát kết quả có được tại vùng buffer :

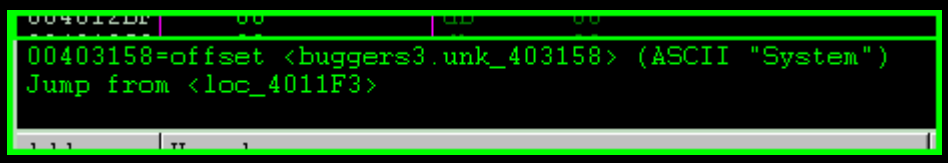


Kết quả này tương ứng với process sau :

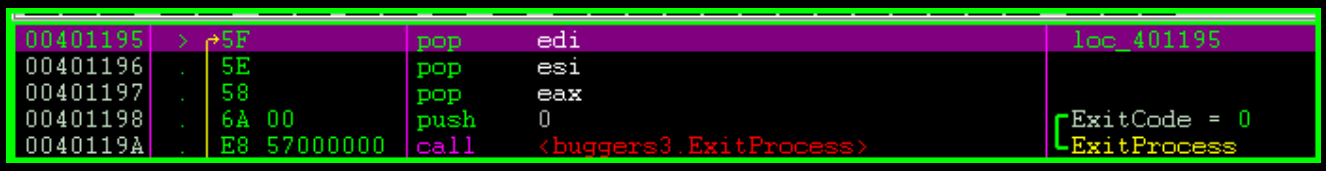


Nhấn F8 trace qua lệnh RETN để trở về code chính, ta dừng lại tại đây :

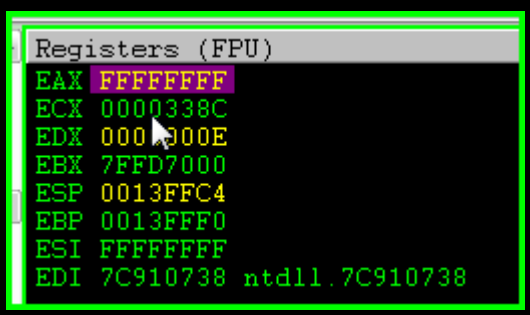




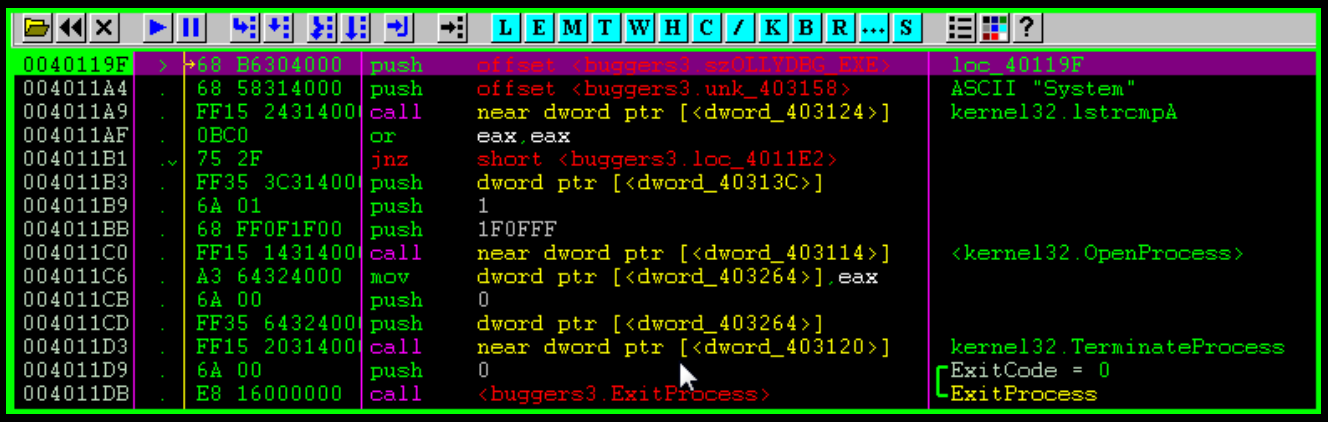
Ta có thông tìn gì từ đoạn code này nào? Nó sử dụng hàm `lstrcmpA` để so sánh 2 chuỗi, chuỗi đầu tiên là tên của target (ASCII "buggers3.exe"), chuỗi thứ hai là tên của Process mà ta có được thông qua hàm `Process32Next`. Nếu hai chuỗi này là giống nhau thì ta sẽ nhận được thông báo là ***not debugged!*** và sau đó thực hiện lệnh nhảy tới địa chỉ `004012AB .^\E9 E5FEFFFF jmp <buggers3.loc_401195>`. Nơi sẽ gọi hàm `ExitProcess` để thoát khỏi chương trình :



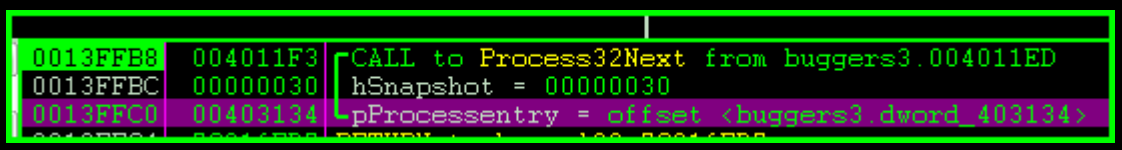
Tuy nhiên, do hai chuỗi của ta là không giống nhau cho nên kết quả không được đẹp như mô tả ở trên. Kết quả của hàm `lstrcmpA` trả về cho thanh ghi EAX giá trị như sau :



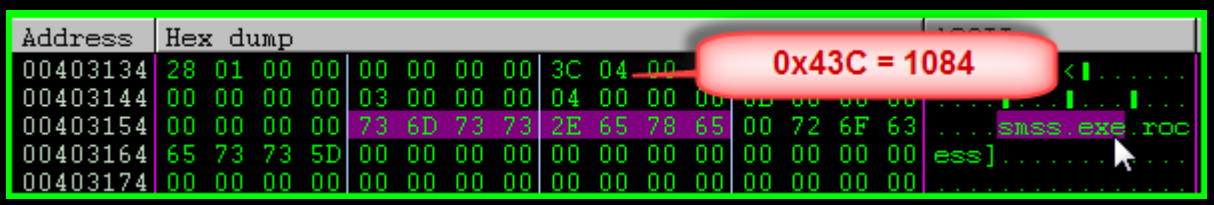
Do EAX <> 0x0 cho nên kết quả lệnh TEST bên dưới sẽ không tác động lên cờ ZF, do đó lệnh nhảy sẽ thực hiện. Lệnh nhảy này đưa ta tới đoạn code sau :

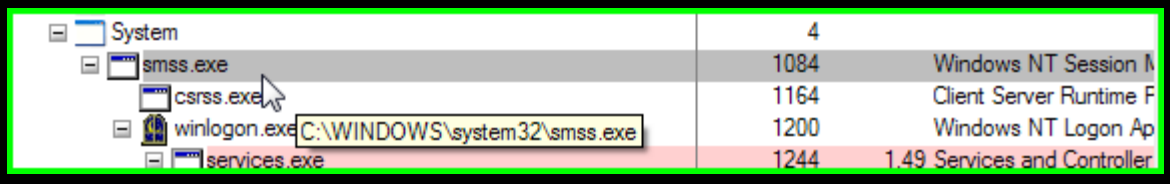


Chà đoạn này hay đây!! So sánh tên của Process tìm được với chuỗi (ASCII "OLLYDBG.EXE"). Nếu giống nhau thì EAX sẽ bằng không và lệnh nhảy sẽ không thực hiện. Đoạn code ở dưới sẽ gọi tới hàm `OpenProcess` để lấy handle của OllyDbg, sau đó truyền handle này cho hàm `TerminateProcess` để thực hiện kill Olly của chúng ta ☺. Do hiện tại tên Process đưa vào để so sánh chưa trùng cho nên target tiếp tục thực hiện hàm `Process32Next` để tìm kiếm tiếp :



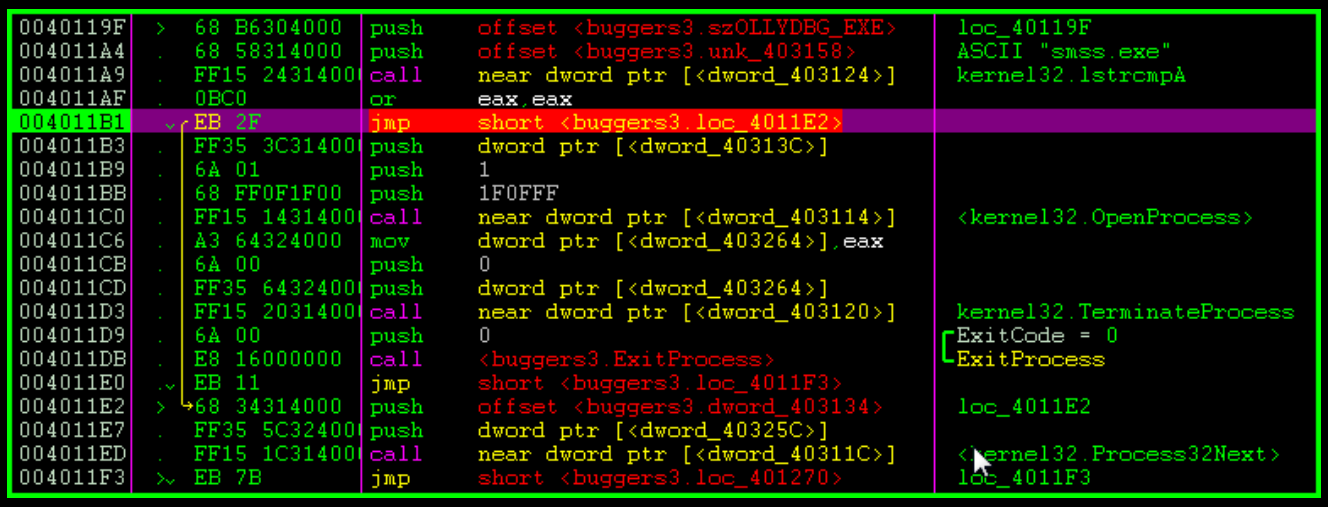
Để xem Process tiếp theo là gì nào, nhấn **Ctrl+F9** thực thi hàm. Quan sát kết quả :



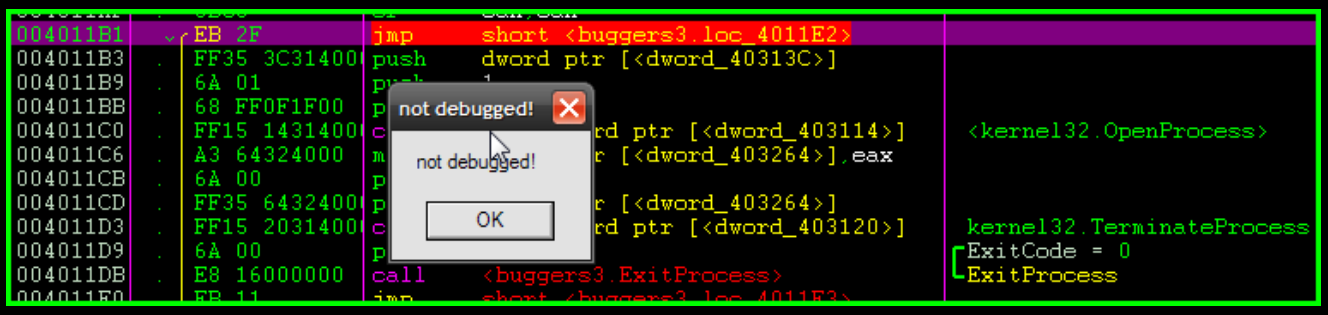


Process tiếp theo là smss.exe, nó là child process của System process. Như vậy, tuần tự nó cứ gọi hàm để lấy thông tin về Process, sau đó lấy tên process có được so sánh với chuỗi OLLYDBG.EXE. Cứ đà đó không sớm thì muộn, kiểu gì cũng tìm thấy OllyDbg process của chúng ta :

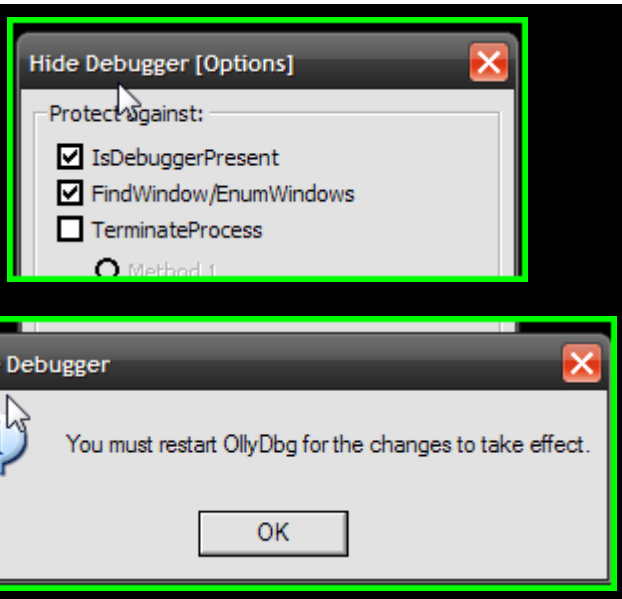
Tới đây thì ta đã hiểu được cơ chế Anti-Debug thứ hai được sử dụng bởi target này rồi. Để không phải mất thời gian thêm nữa, ta patch lệnh nhảy tại địa chỉ 0x004011B1 như sau :



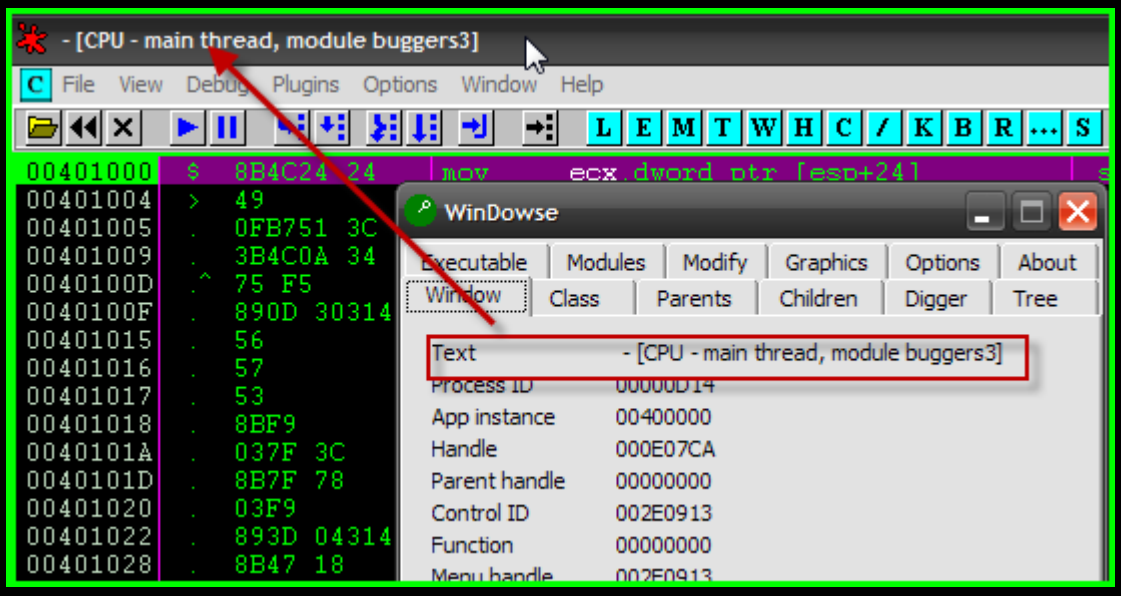
Patch xong ta disable tất cả các BP đã đặt. Sau đó nhấn **F9** để run, ta nhận được kết quả cuối cùng như sau :



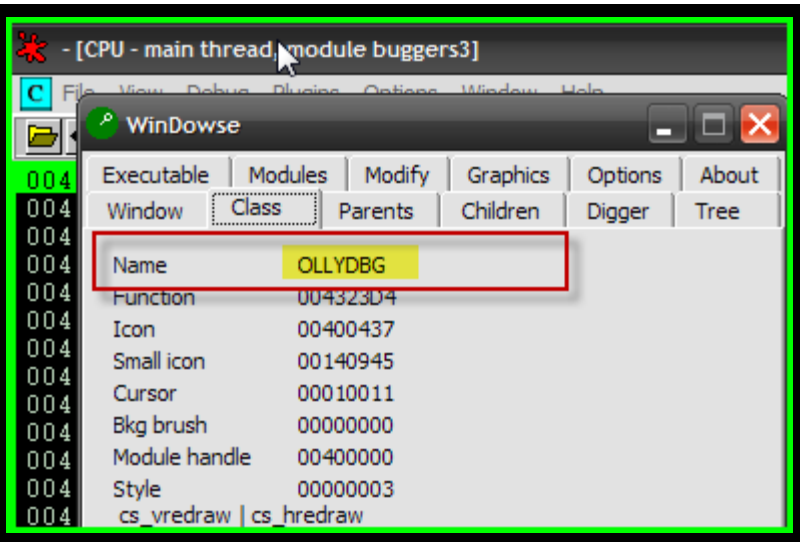
Phù toàn bộ công sức mà chúng ta bỏ ra đã được đền bù xứng đáng. Với việc phân tích target như trên chúng ta đã hoàn toàn có thể manual để bypass được 2 cơ chế Anti-Debug mới. Tuy nhiên, để đỡ phải lặp đi lặp lại việc này ta sẽ sử dụng plugin + tool để fix Olly, sao cho có thể bypass luôn hai kiểu Anti-Debug mà không phải mất công nữa. Đầu tiên, mở Olly mà ta đã đổi tên như ở bài 20 lên (của tôi là M4n0W4R.EXE), sau đó chọn tùy chọn thứ hai của HideDebugger plugin để bypass cơ chế sử dụng FindWindowA. Khi ta đổi tên Olly thì tức là ta đã pass luôn cơ chế Anti-Debug sử dụng (CreateToolhelp32Snapshot, Process32First, Process32Next).



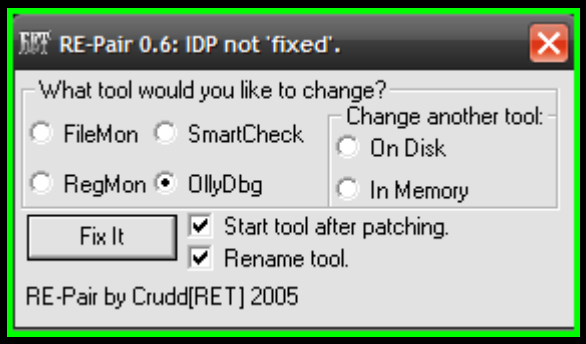
Sau đó ta load target và dừng lại tại EP. Sử dụng chương trình WinDowse để kiểm tra xem plugin đã làm gì :

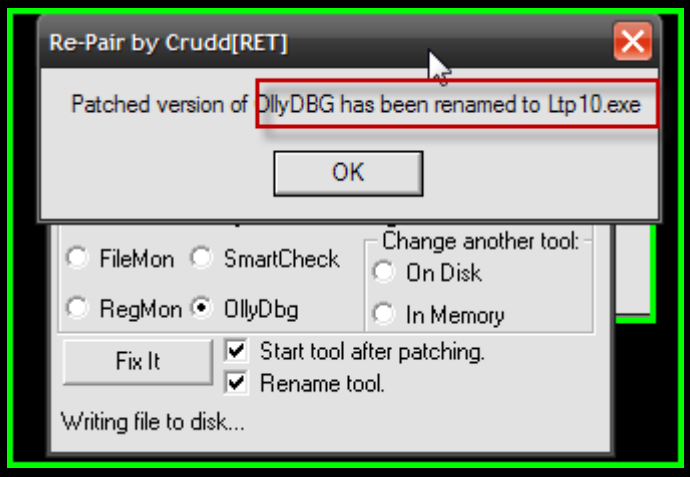


Vậy là Window name đã bị thay đổi, thế còn Class name thì sao :

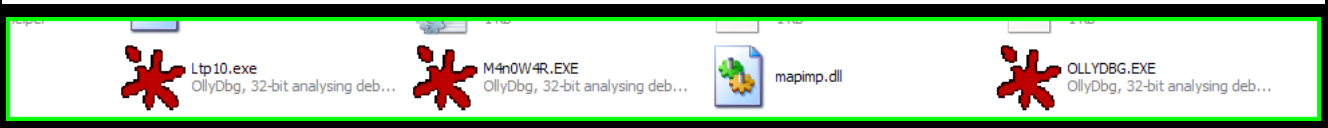


Oạch vẫn là OLLYDBG, như vậy là mới chỉ fix được window name thôi còn class name thì chưa. Mà ở target này nó lại tìm Class name, do đó nếu F9 để thực thi target là OllyDbg của chúng ta toi luôn. Rất may là *"vỏ quýt dày có móng tay nhọn"* , tác giả Crudd[RET] đã xây dựng một tool có tên là repair_v0.6 để fix Olly nhằm bypass được kiểu Anti-Debug bằng cách tìm Class name. Chạy công cụ này để patch Olly (patch cho file OLLYDBG.EXE) :

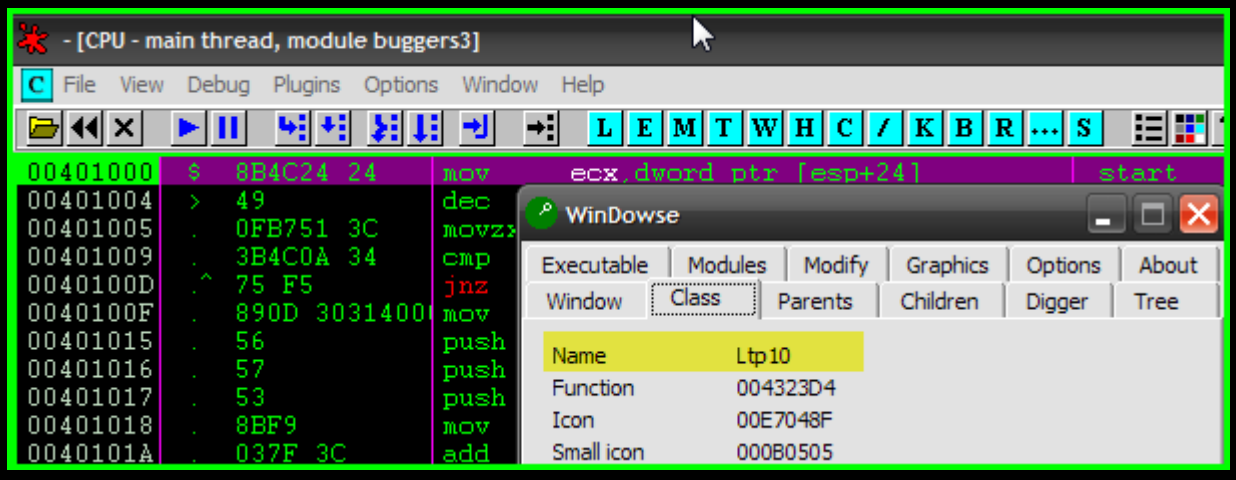




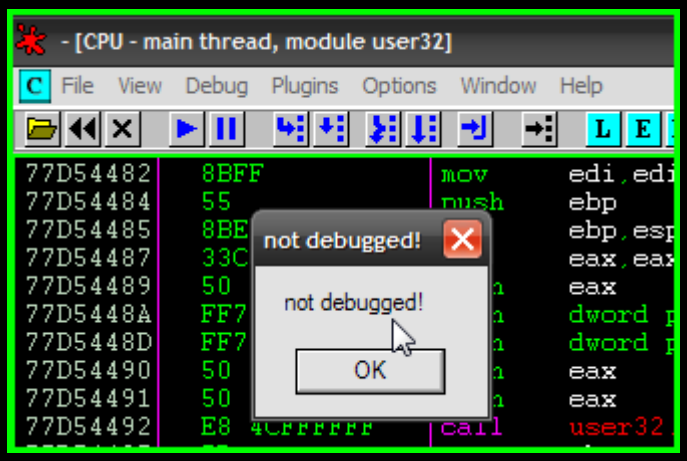
Patch xong nó đổi tên cho file đã patch thành Ltp10.exe.



Chạy file vừa được tạo ra, reload lại target và check lại Class name :



Wow, quá ngon … ta thấy class name đã bị đổi rồi.Coi như target này hết giở trò với Olly của chúng ta. Nhấn **F9** để kiểm tra, target run mượt mà lolz ☺ :



Toàn bộ bài viết xin dừng lại ở đây ☺.

III. Kết luận

OK, toàn bộ bài 21 đến đây là kết thúc. Trong bài viết này tôi đã giới thiệu tiếp tới các bạn 2 cơ chế Anti-Olly bằng cách : tìm xem trong top-level window có Class name trùng với OllyDbg hay không? Nếu trùng thì cho `ExitProcess`. Cách thứ hai là Enumerate toàn bộ các Process đang chạy trên hệ thống để lấy ra tên của từng Process, sau đó đem tên tìm được so sánh với OLLYDBG.EXE, trùng nhau là terminate Olly luôn. Qua bài viết này ta cũng nắm được các phương pháp để manual bypass cũng như áp dụng plug-in + tool để fix Olly. Hẹn gặp lại các bạn ở bài 22, 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