

2009

[Cracking with OllyDbg]

Based on OllyDbg tuts of Ricardo Narvaja (CrackLatinos Team)



www.reaonline.net

kienmanowar



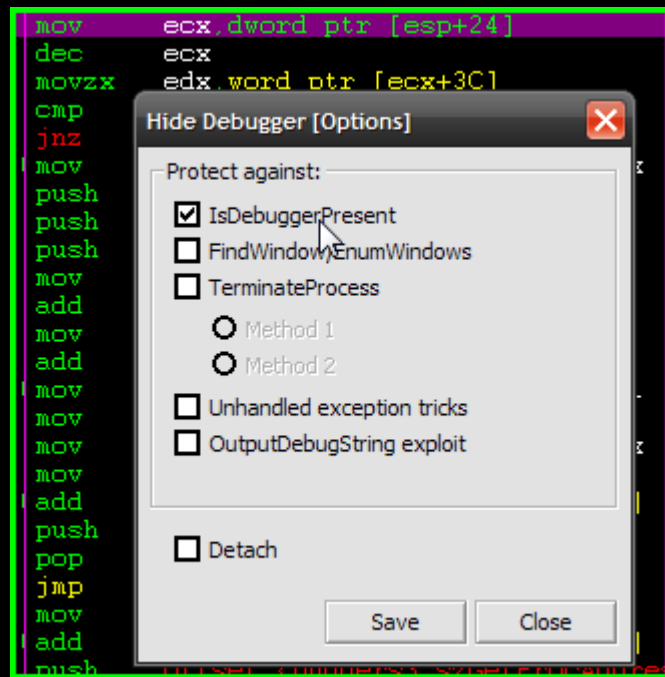
20/01/2010

Mục Lục

I. Giới thiệu chung	2
II. Phân tích và xử lý target.....	3
1. Phân tích buggers3.exe.....	3
III. Kết luận.....	22

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. Now let's go.....☺



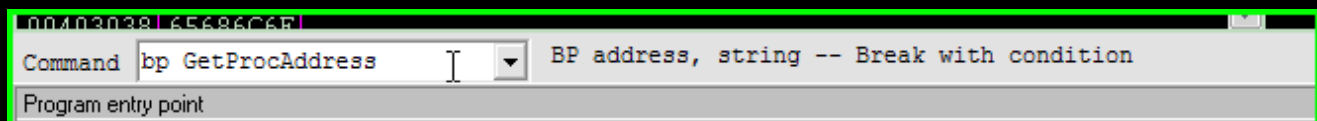
Ta Save lại và sau đó restart Olly để thiết lập trên có hiệu lực. Tiếp theo, sử dụng Task Manager hay Process Explorer để xem danh sách các process đang chạy (trong đó có **OLLYDBG.exe** của chúng ta) :

OLLYDBG.EXE buggers3.exe	1888 OllyDbg, 32-bit analysin 3740
-----------------------------	--

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 :

0013FFAC	004010C5	CALL to GetProcAddress from buggers3.004010BF
0013FFB0	7C800000	hModule = 7C800000 (kernel32)
0013FFB4	00403020	ProcNameOrOrdinal = "FreeLibrary"
0013FFB8	7FFDD000	
0013FFBC	7C910738	ntdll.7C910738
0013FFC0	FFFFFFFF	
0013FFC4	7C816FD7	RETURN to kernel32.7C816FD7

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** :

0013FFAC	004010DB	CALL to GetProcAddress from buggers3.004010D5
0013FFB0	7C800000	hModule = 7C800000 (kernel32)
0013FFB4	00403013	ProcNameOrOrdinal = "LoadLibraryA"
0013FFB8	7FFDD000	
0013FFBC	7C910738	ntdll.7C910738

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

0013FFAC	00401129	CALL to GetProcAddress from buggers3.00401123
0013FFB0	7C800000	hModule = 7C800000 (kernel32)
0013FFB4	00403030	ProcNameOrOrdinal = "CreateToolhelp32Snapshot"
0013FFB8	7FFDE000	
0013FFBC	7C910738	ntdll.7C910738
0013FFC0	FFFFFFFF	

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 :

Registers (FPU)		
EAX	7C864B47	kernel32.CreateToolhelp32Snapshot
ECX	7C919AEB	ntdll.7C919AEB
EDX	7C97C0D8	ntdll.7C97C0D8
EBX	7C800000	kernel32.7C800000
ESP	0013FFAC	
EBP	0013FFF0	
ESI	00403110	offset <buggers3.dword_403110>
EDI	00403030	ASCII "CreateToolhelp32Snapshot"

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

Command	bp eax	BP address, string -- Break with condition
---------	--------	--

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

Address	Disassembly	Comment
7C864B47	8BFF mov edi edi	:CreateToolhelp32Snapshot
7C864B49	55 push ebp	
7C864B4A	8BEC mov ebp, esp	
7C864B4C	83EC 0C sub esp, 0C	
7C864B4F	56 push esi	
7C864B50	8B75 0C mov esi, dword ptr [ebp+C]	

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

Address	Disassembly	Comment
0013FFAC	00401129 CALL to GetProcAddress from buggers3.00401123	hModule = 7C800000 (kernel32)
0013FFB0	7C800000	
0013FFB4	00403049 ProcNameOrOrdinal = "OpenProcess"	
0013FFB8	7FFDE000	
0013FFBC	7C910738 ntdll.7C910738	
0013FFC0	FFFFFFFF	
0013FFC4	7C816FD7 RETURN to kernel32.7C816FD7	
0013FFC8	7C910738 ntdll.7C910738	

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 :

Address	Disassembly	Comment
7C8309E1	8BFF mov edi edi	:OpenProcess
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]	

Tiếp tục nhấn **F9** :

Address	Disassembly	Comment
0013FFAC	00401129 CALL to GetProcAddress from buggers3.00401123	hModule = 7C800000 (kernel32)
0013FFB0	7C800000	
0013FFB4	00403055 ProcNameOrOrdinal = "Process32First"	
0013FFB8	7FFDE000	
0013FFBC	7C910738 ntdll.7C910738	

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

Address	Disassembly	Comment
7C863E1D	8BFF mov edi edi	:Process32First
7C863E1F	55 push ebp	
7C863E20	8BEC mov ebp, esp	
7C863E22	81EC 30020000 sub esp, 230	

Khai thác tiếp thông tin :

0013FFAC	00401129	CALL to GetProcAddress from buggers3.00401123
0013FFB0	7C800000	hModule = 7C800000 (kernel32)
0013FFB4	00403064	ProcNameOrOrdinal = "Process32Next"
0013FFB8	7FFDE000	

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

7C863F90	8BFF	mov	edi,edi	Process32Next
7C863F92	55	push	ebp	
7C863F93	8BEC	mov	ebp,esp	
7C863F95	81EC 30020000	sub	esp,230	

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

0013FFAC	00401129	CALL to GetProcAddress from buggers3.00401123
0013FFB0	7C800000	hModule = 7C800000 (kernel32)
0013FFB4	00403072	ProcNameOrOrdinal = "TerminateProcess"
0013FFB8	7FFDE000	
0013FFBC	7C910738	ntdll.7C910738

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** :

0013FFAC	00401129	CALL to GetProcAddress from buggers3.00401123
0013FFB0	77D40000	hModule = 77D40000 (user32)
0013FFB4	00403091	ProcNameOrOrdinal = "FindWindowA"
0013FFB8	7FFDE000	
0013FFBC	7C910738	ntdll.7C910738

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 :

77D54482	8BFF	mov	edi,edi	FindWindowA
77D54484	55	push	ebp	
77D54485	8BEC	mov	ebp,esp	

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` :

7C864B47	8BFF	mov	edi, edi	CreateToolhelp32Snapshot
7C864B49	55	push	ebp	
7C864B4A	8BEC	mov	ebp, esp	
7C864B4C	83EC 0C	sub	esp, 0C	
7C864B4F	56	push	esi	

Thông tin trên cửa sổ Stack :

0013FFB8	00401151	CALL to CreateToolhelp32Snapshot from buggers3.0040114B
0013FFBC	00000002	Flags = TH32CS_SNAPPROCESS
0013FFC0	00000000	ProcessID = 0
0013FFC4	7C816FD7	RETURN to kernel32.7C816FD7
0013FFC8	7C910738	ntdll.7C910738

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 :

- `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.
- `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 :

```
Registers (FPU)
EAX 00000030
ECX 0013FF43
EDX 7C90EB94 ntdll.KiFastSystemCallRet
EBX 7FFD7000
ESP 0013FFB8
EBP 0013FFF0
ESI FFFFFFFF
EDI 7C910738 ntdll.7C910738
EIP 7C864BC7 kernel32.7C864BC7
```

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 :

```
7C863E1D 8BFF mov edi,edi ;Process32First
7C863E1F 55 push ebp
7C863E20 8BEC mov ebp,esp
7C863E23 81EC 00000000
```

```
0013FFB8 00401162 CALL to Process32First from buggers3.0040115C
0013FFBC 00000030 hSnapshot = 00000030
0013FFC0 00403134 pProcessentry = offset <buggers3.dword_403134>
0013FFC4 7C816FD7 RETURN to kernel32.7C816FD7
```

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 :

Assembly code snippet:

```

7C863F1D  6A FF      push     -1
7C863F1F  FF75 08    push     dword ptr [7C863F1F]
7C863F22  8975 F4    mov      dword ptr [7C863F22], eax
7C863F25  8975 F8    mov      dword ptr [7C863F25], eax
Return to 00401162 (buggers3.00401162)

```

Hex dump snippet:

Address	Hex dump	ASCII
00403134	28 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00	(.....)
00403144	00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 00
00403154	00 00 00 00 00 5B 53 79 73 74 65 6D 20 50 72 6F[System Proc
00403164	65 73 73 5D 00 00 00 00 00 00 00 00 00 00 00 00	ess].....
00403174	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403184	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403194	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00bugg
004031A4	65 72 73 33 2E 65 00 00 00 00 00 00 00 00 00 00	ers3.exe.....
004031B4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004031C4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00MessageB
004031D4	6F 78 41 00 00 00 00 00 00 00 00 00 00 00 00 00	oxA.....
004031E4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004031F4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403204	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Annotations:

- th32ProcessID:** Identifier of the process.
- szExeFile:** filename of the executable file for the process.

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 :

Process	PID	CPU	Description
System Idle Process	0	97.76	
Interrupts	n/a		Hardware Interrupts
DPCs	n/a		Deferred Procedure Call

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

Address	Disassembly	Comment
77D54482	8BFF	mov edi, edi
77D54484	55	push ebp
77D54485	8BEC	mov ebp, esp
77D54487	33C0	xor eax, eax
77D54489	50	push eax

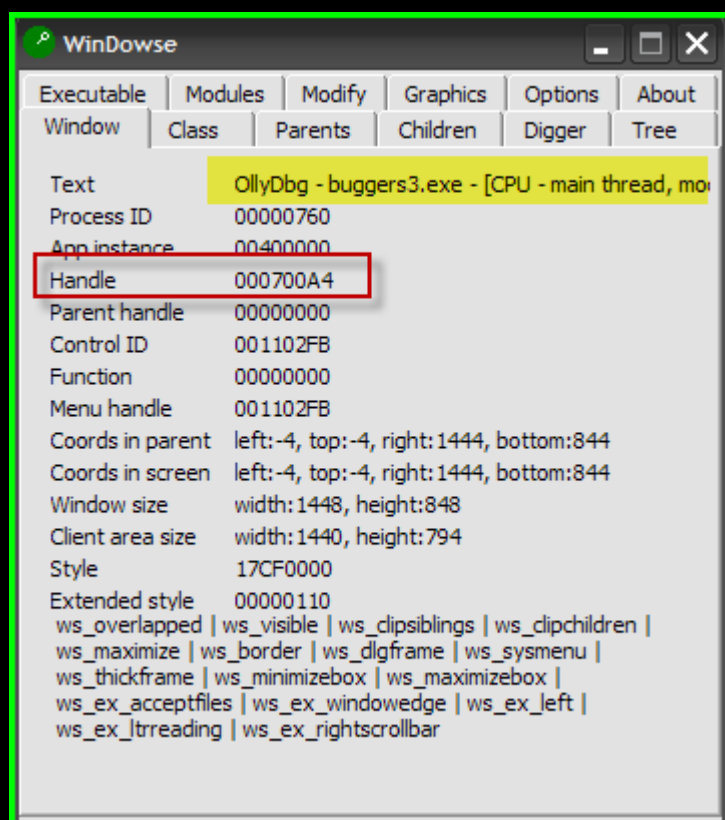
Address	Disassembly	Comment
0013FFB8	CALL to FindWindowA from buggers3.00401169	
0013FFBC	Class = "OllyDbg"	
0013FFC0	Title = NULL	
0013FFC4	RETURN to kernel32.7C816FD7	
0013FFC8	ntdll.7C910738	
0013FFCC	FFFFFFFF	

Khà khà, mẫu 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 :

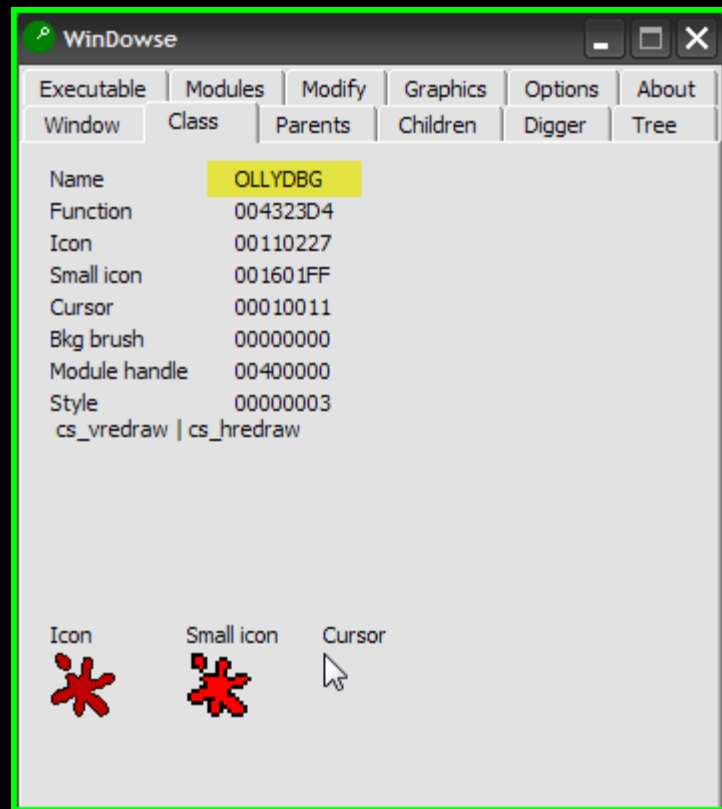
The **FindWindow** function retrieves the handle to the top-level window whose class name and window name match the specified strings. This function does not search child windows.

```
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 :



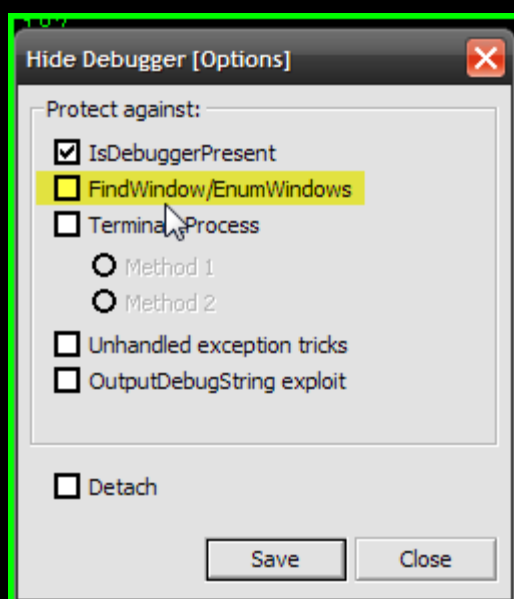
Kết quả này trùng với giá trị mà chương trình WinDowse thu thập được. Ta nhấn **F8** trace qua lệnh `RETN` để trở về code chính của target, quan sát xem nó sẽ làm gì với giá trị handle có được :

00401162	6A 00	push	0	
00401164	68 AE304000	push	offset <buggers3.szOlllyDbg>	ASCII "OlllyDbg"
00401169	FF15 28314000	call	near dword ptr [<dword_403128>]	<user32.FindWindowA>
0040116F	83F8 00	cmp	eax, 0	
00401172	0BC0	or	eax, eax	
00401174	75 04	jnz	short <buggers3.loc_40117A>	
00401176	7C 27	jl	short <buggers3.loc_40119F>	
00401178	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_403128>	
00401187	FF36	push	dword ptr [esi]	loc_401187
00401189	FF15 0C314000	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	75 04	jnz	short <buggers3.loc_40117A>	
00401176	7C 27	jl	short <buggers3.loc_40119F>	
00401178	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_403128>	
00401187	FF36	push	dword ptr [esi]	loc_401187
00401189	FF15 0C314000	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 :

```

C 0 ES 0023 32bit 0(FFFFFFFF)
P 0 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_MOD_NOT_FOUND (0000007E)

```

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

00401172	0BC0	or	eax, eax	
00401174	75 04	jnz	short <buggers3.loc_40117A>	
00401176	7C 27	jl	short <buggers3.loc_40119F>	
00401178	EB 25	jmp	short <buggers3.loc_40119F>	
0040117A	> 50	push	eax	loc_40117A
Jump is NOT taken				
0040117A=<buggers3.loc_40117A>				

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

00401178	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_403140>	
00401187	FF36	push	dword ptr [esi]	loc_401187
00401189	FF15 0C314000	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

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 :

0013FFB8	004011F3	CALL to Process32Next from buggers3.004011ED
0013FFBC	00000030	hSnapshot = 00000030
0013FFC0	00403134	pProcessentry = offset <buggers3.dword_403134>
0013FFC4	7C816FD7	RETURN to kernel32.7C816FD7
0013FFC8	7C910738	ntdll.7C910738

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 :

Return to 004011F3 (<buggers3.loc_4011F3>)						
Address	Hex dump				ASCII	
00403134	28 01 00 00	00 00 00 00	04 00 00 00	00 00 00 00	(.....)	
00403144	00 00 00 00	62 00 00 00	00 00 00 00	08 00 00 00	...b.....	
00403154	00 00 00 00	53 79 73 74	65 6D 00 20	50 72 6F 63	...System. Proc	
00403164	65 73 73 5D	00 00 00 00	00 00 00 00	00 00 00 00	ess].....	
00403174	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
00403184	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
00403194	00 00 00 00	00 00 00 00	00 00 00 00	62 75 67 67bugg	
004031A4	65 72 73 33	2E 65 78 65	00 00 00 00	00 00 00 00	ers3.exe.....	
004031B4	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
004031C4	00 00 00 00	00 00 00 00	4D 65 73 73	61 67 65 42MessageB	
004031D4	6F 78 41 00	00 00 00 00	00 00 00 00	00 00 00 00	oxA.....	
004031E4	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
004031F4	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	

Kết quả này tương ứng với process sau :

DPCs	n/a	Deferred Procedure Call
System	4	
smss.exe	1084	Windows NT Session Manager
csrss.exe	1164	Client Server Runtime Process

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

00401270	> 68 50314000	push	offset <buggers3.unk_403158>	loc_401270
00401275	. 68 A0314000	push	offset <buggers3.szbuggers3_exe	ASCII "buggers3.exe"
0040127A	. FF15 24314000	call	near dword ptr [<dword_403124>]	kernel32.lstrcmpA
00401280	. 85C0	test	eax, eax	
00401282	. ^ 0F85 17FFFFFF	jnz	<buggers3.loc_40119F>	
00401288	. 68 CC314000	push	offset <buggers3.szMessageBoxA>	ASCII "MessageBoxA"
0040128D	. FF35 2C314000	push	dword ptr [<dword_40312C>]	user32.77D40000
00401293	. FF15 08314000	call	near dword ptr [<dword_403108>]	kernel32.GetProcAddress
00401299	. 6A 00	push	0	
0040129B	. 6A 00	push	0	
0040129D	. 68 CD304000	push	offset <buggers3.sznotdebugged>	ASCII "not debugged!"
004012A2	. 68 CD304000	push	offset <buggers3.sznotdebugged>	ASCII "not debugged!"
004012A7	. 6A 00	push	0	
004012A9	. FFD0	call	near eax	
004012AB	. ^ E9 E5FEFFFF	jmp	<buggers3.loc_401195>	

004012BF	. 00	db	00	
00403158	=offset <buggers3.unk_403158>	(ASCII "System")		
Jump from <loc_4011F3>				

Ta có thông tin 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 :

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

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 :

Registers (FPU)	
EAX	FFFFFFFF
ECX	0000338C
EDX	0000300E
EBX	7FFD7000
ESP	0013FFC4
EBP	0013FFF0
ESI	FFFFFFFF
EDI	7C910738 ntdll.7C910738

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 :

```

0040119F  68 B6304000  push offset <buggers3.szOLLYDBG_EXE>    loc_40119F
004011A4  68 58314000  push offset <buggers3.unk_403158>        ASCII "System"
004011A9  FF15 24314000 call near dword ptr [<dword_403124>]      kernel32.lstrcmpA
004011AF  0BC0        or     eax,eax
004011B1  75 2F        jnz    short <buggers3.loc_4011E2>
004011B3  FF35 3C314000 push dword ptr [<dword_40313C>]
004011B9  6A 01        push  1
004011BB  68 FF0F1F00  push 1F0FFF
004011C0  FF15 14314000 call near dword ptr [<dword_403114>]      <kernel32.OpenProcess>
004011C6  A3 64324000  mov     dword ptr [<dword_403264>],eax
004011CB  6A 00        push  0
004011CD  FF35 64324000 push dword ptr [<dword_403264>]
004011D3  FF15 20314000 call near dword ptr [<dword_403120>]      kernel32.TerminateProcess
004011D9  6A 00        push  0
004011DB  E8 16000000  call    <buggers3.ExitProcess>          ExitCode = 0
                                              ExitProcess

```

```

0040120C  83C0 04      add     eax,4
004030B6 offset <buggers3.szOLLYDBG_EXE> (ASCII "OLLYDBG_EXE")
Jumps from 00401176, 00401178, 00401282

```

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 :

```

7C863F90  8BFF        mov     edi,edi    Process32Next
7C863F92  55          push    ebp

```

```

0013FFB8 004011F3 CALL to Process32Next from buggers3.004011ED
0013FFBC 00000030 hSnapshot = 00000030
0013FFC0 00403134 pProcessentry = offset <buggers3.dword_403134>

```

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

Address	Hex dump	
00403134	28 01 00 00 00 00 00 00 3C 04 00 00 00 00 00 00	0x43C = 1084
00403144	00 00 00 00 03 00 00 00 04 00 00 00 00 00 00 00	
00403154	00 00 00 00 73 6D 73 73 2E 65 78 65 00 72 6F 63	smss.exe.roc
00403164	65 73 73 5D 00 00 00 00 00 00 00 00 00 00 00 00	ess].....
00403174	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

System	4	
smss.exe	1084	Windows NT Session M
csrss.exe	1164	Client Server Runtime F
winlogon.exe	1200	Windows NT Logon Ap
services.exe	1244	1.49 Services and Controller

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 :

WINWORD.EXE	2024	Microsoft Word
OLLYDBG.EXE	1888	OllyDbg, 32-bit analysin
buggers3.exe	3740	
winhlp32.exe	3464	Microsoft® Help

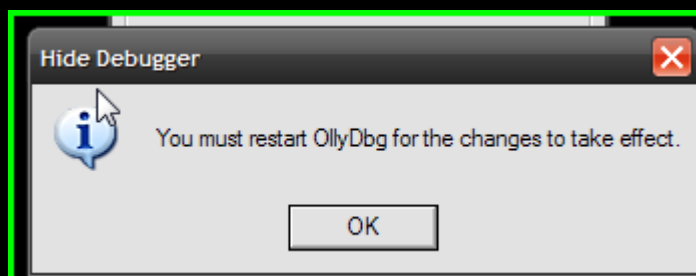
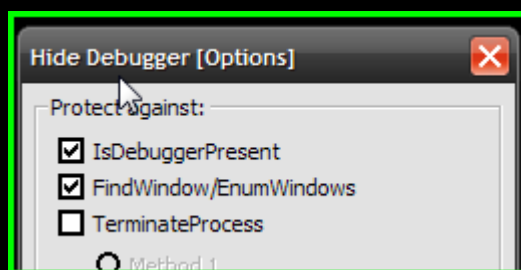
Tới đây thì ta đã hiểu được cơ chế Anti-Debug thứ hai được sử dụng bởi target này rồi. Để không phải mất thời gian thêm nữa, ta patch lệnh nhảy tại địa chỉ **0x004011B1** như sau :

0040119F	> 68 B6304000	push	offset <buggers3.szOLLYDBG_EXE>	loc_40119F
004011A4	68 58314000	push	offset <buggers3.unk_403158>	ASCII "smss.exe"
004011A9	FF15 24314000	call	near dword ptr [<dword_403124>]	kernel32.lstrcpA
004011AF	0BC0	or	eax, eax	
004011B1	EB 2F	jmp	short <buggers3.loc_4011E2>	
004011B3	FF35 3C314000	push	dword ptr [<dword_40313C>]	
004011B9	6A 01	push	1	
004011BB	68 FF0F1F00	push	1F0FFF	
004011C0	FF15 14314000	call	near dword ptr [<dword_403114>]	<kernel32.OpenProcess>
004011C6	A3 64324000	mov	dword ptr [<dword_403264>], eax	
004011CB	6A 00	push	0	
004011CD	FF35 64324000	push	dword ptr [<dword_403264>]	
004011D3	FF15 20314000	call	near dword ptr [<dword_403120>]	kernel32.TerminateProcess
004011D9	6A 00	push	0	ExitCode = 0
004011DB	E8 16000000	call	<buggers3.ExitProcess>	ExitProcess
004011E0	EB 11	jmp	short <buggers3.loc_4011F3>	
004011E2	> 68 34314000	push	offset <buggers3.dword_403134>	loc_4011E2
004011E7	FF35 5C324000	push	dword ptr [<dword_40325C>]	
004011ED	FF15 1C314000	call	near dword ptr [<dword_40311C>]	<kernel32.Process32Next>
004011F3	> EB 7B	jmp	short <buggers3.loc_401270>	loc_4011F3

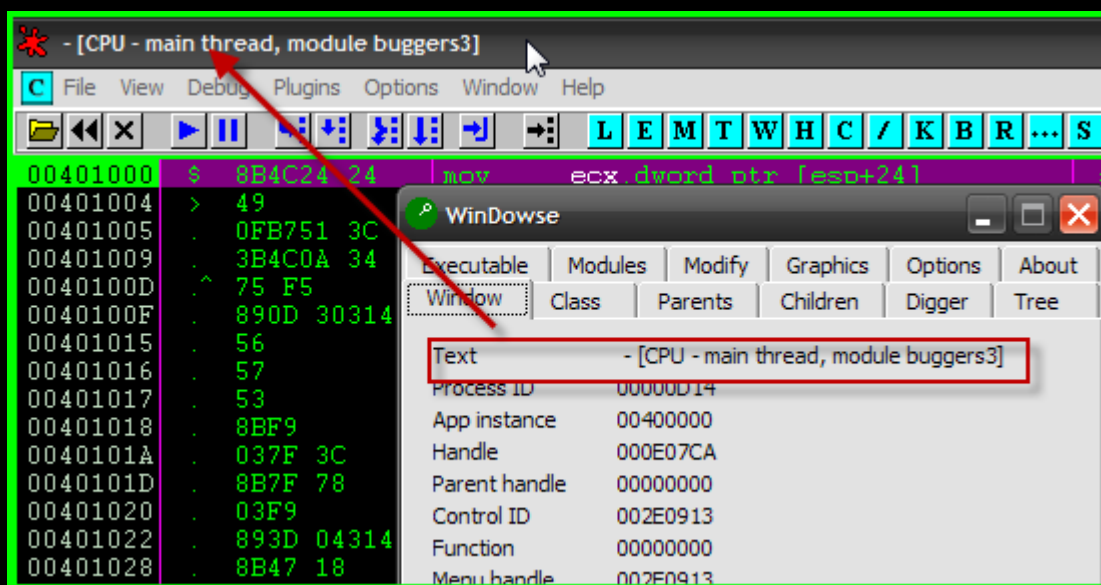
Patch xong ta disable tất cả các BP đã đặt. Sau đó nhấn **F9** để run, ta nhận được kết quả cuối cùng như sau :

004011B1	> EB 2F	jmp	short <buggers3.loc_4011E2>	
004011B3	FF35 3C314000	push	dword ptr [<dword_40313C>]	
004011B9	6A 01	push	1	
004011BB	68 FF0F1F00	push	1F0FFF	
004011C0	FF15 14314000	call	near dword ptr [<dword_403114>]	<kernel32.OpenProcess>
004011C6	A3 64324000	mov	dword ptr [<dword_403264>], eax	
004011CB	6A 00	push	0	
004011CD	FF35 64324000	push	dword ptr [<dword_403264>]	
004011D3	FF15 20314000	call	near dword ptr [<dword_403120>]	kernel32.TerminateProcess
004011D9	6A 00	push	0	ExitCode = 0
004011DB	E8 16000000	call	<buggers3.ExitProcess>	ExitProcess
004011E0	EB 11	jmp	short <buggers3.loc_4011F3>	

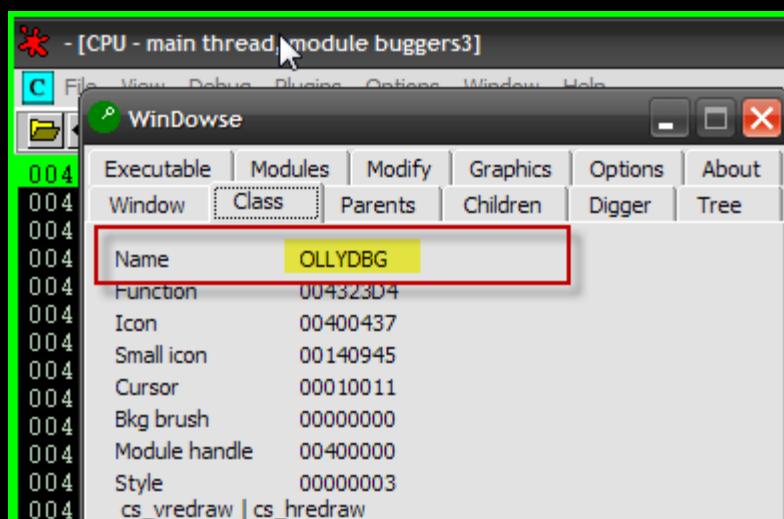
Phù toàn bộ công sức mà chúng ta bỏ ra đã được đền bù xứng đáng. Với việc phân tích target như trên chúng ta đã hoàn toàn có thể manual để bypass được 2 cơ chế Anti-Debug mới. Tuy nhiên, để đỡ phải lặp đi lặp lại việc này ta sẽ sử dụng plugin + tool để fix Olly, sao cho có thể bypass luôn hai kiểu Anti-Debug 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**).



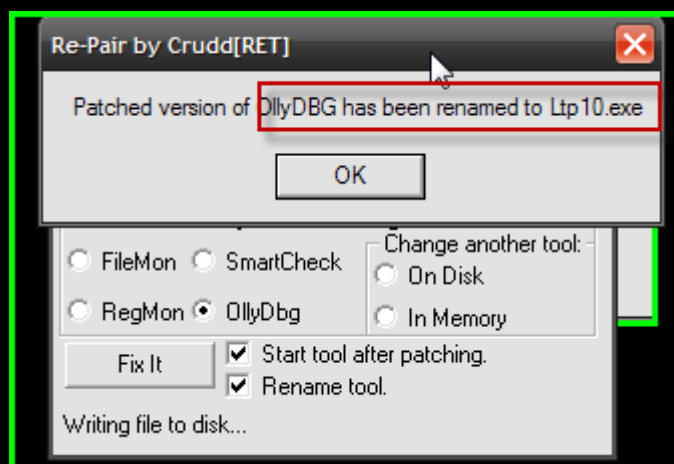
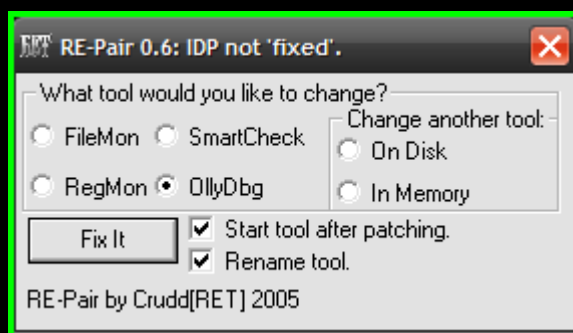
Sau đó ta load target và dừng lại tại EP. Sử dụng chương trình WinDowse để kiểm tra xem plugin đã làm gì :



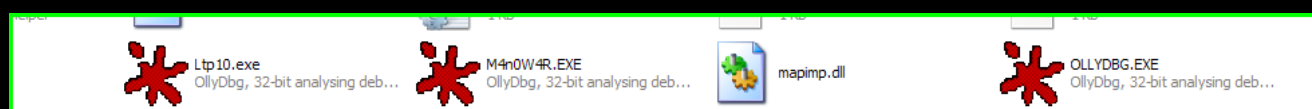
Vậy là Window name đã bị thay đổi, thế còn Class name thì sao :



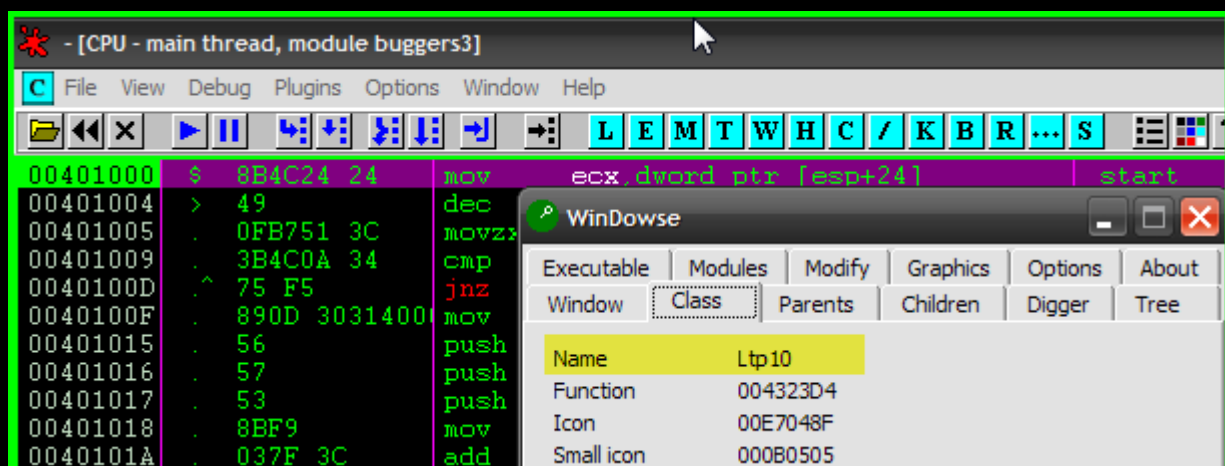
Oạch vẫn là **OLLYDBG**, như vậy là mới chỉ fix được window name thôi còn class name thì chưa. Mà ở target này nó lại tìm Class name, do đó nếu F9 để thực thi target là OllyDbg của chúng ta toi luôn. Rất may là "*vỏ quýt dày có móng tay nhọn*", tác giả Crudd[RET] đã xây dựng một tool có tên là repair_v0.6 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) :



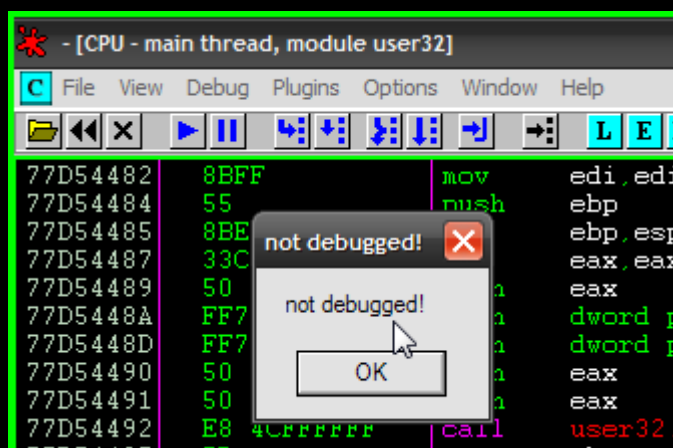
Patch xong nó đổi tên cho file đã patch thành **Ltp10.exe**.



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 giờ 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]

Kien

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

My family, Computer_Angel, Moonbaby , Zombie_Deathman, Littleboy, Benina, QHQCkcr, 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, dzungltn, 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)