

1. Tổng quan

Hiện nay, các hướng dẫn về unpack Themida tương đối nhiều. Tuy nhiên, đa số chỉ dừng lại ở việc dùng trick hay script có sẵn. Vì thế, thông qua bài viết này, tôi sẽ giúp các bạn hiểu được bản chất hoạt động của Scylla và cách các WinAPI được gọi trong chương trình như thế nào. Tôi nói trước, bài viết này không giúp các bạn có thể unpack toàn bộ phiên bản Themida mà sẽ giúp các bạn tư duy vấn đề nếu phải gặp các mẫu tương tự. Ngoài ra, địa chỉ chương trình mà tôi đề cập trong bài viết có thể sẽ không giống với các bạn nên đừng thắc mắc vì sao nhé.

2. Phân tích

Tôi sẽ sử dụng sample Slickshoes của Lazarus làm ví dụ cho bài viết. Đường dẫn download sample nằm ở cuối bài viết. Mong các anh Lazarus tha thứ cho tôi, chỉ vì Google hiện kết quả của các anh ở trang đầu tiên thôi. Để có thể unpack cũng như khôi phục lại bảng IAT, chúng ta thường sẽ thực hiện các bước sau:

- Giải mã các section về đúng các byte gốc
- Tìm OEP
- Khôi phục IAT

1. 2.1. Bypass anti-debug

Themida sử dụng nhiều kĩ thuật anti-debug khác nhau. Thay vì chúng ta phải tự giải quyết vấn đề này, chúng ta sẽ sử dụng plugin ScyllaHide kết hợp với x64dbg. Cách cài đặt như thế nào thì các bạn tự tìm hiểu nhé. Ngay khi load sample vào x64dbg, chúng ta sẽ lựa chọn profile là "Themida x86/x64" và bấm Run (F9) để đi đến Entry Point của sample.

2. 2.2. Giải mã các section về đúng các byte gốc

Hiện tại, sample mà chúng ta phân tích đang dừng ở Entry Point. Vấn đề cần đặt ra hiện tại là làm sao để tìm ra OEP sau khi unpack thành công. Đối với tất cả các chương trình bị pack, để một chương trình có thể thực thi được như lúc chưa pack thì rõ ràng là trước khi chương trình gốc thực thi thì packer phải unpack hoàn toàn chương trình trên bộ nhớ sau đó nhảy tới OEP của chương trình đó để thực thi.

Theo lí thuyết, anti-debug đã có ScyllaHide giải quyết, và khi chúng ta bấm F9 (Run) thì mã độc sẽ nhảy vào vòng lặp kết nối tới C&C để chờ lệnh. Nhưng cuộc đời đâu phải lúc nào cũng suôn sẻ như vậy được, nếu config C&C là một file riêng biệt không nằm trong bản thân mã độc thì khả năng cao khi ta bấm hàm Run mã độc sẽ tự chấm dứt và vô số trường hợp khác nữa. Vì vậy, trước khi mã độc tự chấm dứt, chúng ta cần ngăn chặn hành động này. Lúc này coi như mã độc đã giải mã hoàn toàn trên bộ nhớ. Tôi thường áp dụng mẹo sau để giải quyết vấn đề này.

Các bạn đặt breakpoint tại WinAPI mà trước khi mã độc chấm dứt sẽ chạy qua đó, tôi sẽ chọn GetCurrentProcessID. Sau đó, các bạn chuyển qua tab Memory của x64dbg, tìm đến phân vùng được cấp phát cho mã độc. Các bạn chú ý đến các section không có tên. Lí do là vì các byte trong

section này không giống các byte có trong một chương trình bình thường, chẳng hạn như "55 8B EC" hay "E8 ? ? ? ?". Khả năng cao 01 trong 02 section này là sẽ là section .text của chương trình ban đầu.

008B0000	00001000	slickshoes_dropper_pack		IMG	-RW--	ERWC-
008B1000	00180000	"		IMG	ERW--	ERWC-
00A31000	0000C000	".rsrc"	Resources	IMG	-RW-	ERWC-
00A3D000	00001000	".idata "	Import tables	IMG	-RW--	ERWC-
00A3E000	00257000	"suy1crzz"		IMG	ERW--	ERWC-
00C95000	00170000	"ajqluhke"		IMG	ERW--	ERWC-
00E05000	00001000			IMG	ERW--	ERWC-

Các bạn nhớ chọn "follow in dump" đối với 02 section đó để quan sát nhé và sau đó chúng ta bấm F9 vài lần thì sẽ thấy section không tên đầu tiên đã giải mã ra thành công. Chúng ta nhận thấy 03 byte quen thuộc của prologue là "55 8B EC" tại đầu section.

Address	Hex	ASCII
008B1000	55 8B EC B8 08 28 00 00 E8 73 44 00 00 A1 04 80	U.ì.(..èsD..i..
008B1010	8B 00 33 C5 89 45 FC 56 6A 00 6A 00 6A 02 6A 00	..3Á.Eüvj.j.j.j.
008B1020	6A 00 68 00 00 00 10 68 F8 77 8B 00 C7 85 F8 D7	j.h....høw..Ç.øx
008B1030	FF FF 00 00 00 00 E8 C5 EF E8 FF 90 8B F0 83 FE	ÿÿ....èÁièÿ..ð.p
008B1040	FF 75 13 33 C0 5E 8B 4D FC 33 CD E8 80 00 00 00	ÿÿ.3Á^..Mü3iè....
008B1050	8B E5 5D C2 10 00 53 8B 1D 04 60 8B 00 57 6A 00	..à]Á..S...`.wj.
008B1060	8D 85 F8 D7 FF FF 50 68 00 72 17 00 68 20 8B 8B	..øÿÿPh.r..h..
008B1070	00 56 FF D3 68 00 28 00 00 8D 8D FC D7 FF FF 6A	.vÿöh.(...üxÿÿj
008B1080	41 51 E8 09 36 00 00 83 C4 0C BF 00 1C 00 00 90	AQè.6...Ä.¿.....
008B1090	6A 00 8D 95 F8 D7 FF FF 52 68 00 28 00 00 8D 85	j...øÿÿRh.(....
008B10A0	FC D7 FF FF 50 56 FF D3 6A 0A E8 D3 F9 E8 FF 90	üxÿÿPVÿÿÿj.èöüèÿ.
008B10B0	4F 75 DD 56 E8 75 FA E8 FF 90 8B 4D FC 8D 47 01	Ouÿÿvèuüèÿ..Mü.G.
008B10C0	5F 5B 33 CD 5E E8 06 00 00 00 8B E5 5D C2 10 00	_[3iÀè.....à]Á..
008B10D0	3B 0D 04 80 8B 00 75 02 F3 C3 E9 A0 01 00 00 8B	;.u.óÁé....
008B10E0	FF 55 8B EC 83 3D 28 FD A2 00 01 75 05 E8 A3 07	ÿÿ.ì.=(ÿc..u.èf.
008B10F0	00 00 FF 75 08 E8 EC 05 00 00 68 FF 00 00 00 E8	..ÿÿ.èì...hÿ...è

Trong quá trình Run, mã độc sẽ dừng tại các instruction sti. Đây là exception được chèn vào để thay đổi luồng thực thi của chương trình. Nếu chúng ta debug mà chuyển quyền xử lý exception về cho debugger xử lý thì sẽ ra sai kết quả cuối, vì thế ta sẽ chuyển cho chương trình xử lý bằng cách bấm Shift + F9.

00C950B5	0BC0	or eax,eax
00C950B7	74 01	je slickshoes_dropper_pack.C950BA
00C950B9	FB	sti
00C950BA	E9 4E010000	jmp slickshoes_dropper_pack.C9520D
00C950BF	60	pushad
00C950C0	8B7424 24	mov esi,dword ptr ss:[esp+24]
00C950C4	8B7C24 28	mov edi,dword ptr ss:[esp+28]

3. 2.3. Tìm OEP

Chúng ta đã giải mã thành công trên bộ nhớ mà mã độc chưa tự chấm dứt bản thân, việc tiếp theo cần làm là tìm OEP. Nếu các bạn hay để ý khi RE sẽ thấy OEP của một chương trình EXE sẽ là lời gọi đến hàm ____security_init_cookie. Hàm này có nhiệm vụ khởi tạo cookie để dùng cho việc chống stack overflow. Nội dung của hàm sẽ tương tự nhau ở các chương trình.

```

and [ebp+SystemTimeAsFileTime.dwLowDateTime], 0
and [ebp+SystemTimeAsFileTime.dwHighDateTime], 0
push ebx
push edi
mov edi, 0BB40E64Eh
mov ebx, 0FFFF0000h
cmp eax, edi
jz short loc_1392791
test ebx, eax
jz short loc_1392791
not eax
mov dword_1398008, eax
jmp short loc_13927F6

```

```

1 void __cdecl __security_init_cookie()
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5     SystemTimeAsFileTime.dwLowDateTime = 0;
6     SystemTimeAsFileTime.dwHighDateTime = 0;
7     if ( dword_1398004 == -1153374642 || (dword_1398004 & 0xFFFF0000) == 0 )
8     {
9         GetSystemTimeAsFileTime(&SystemTimeAsFileTime);
10        v0 = SystemTimeAsFileTime.dwLowDateTime ^ SystemTimeAsFileTime.dwHighDateTime;
11        v1 = GetCurrentProcessId() ^ v0;
12        v2 = GetCurrentThreadId() ^ v1;
13        v3 = GetTickCount() ^ v2;
14        RtlQueryPerformanceCounter(v5);
15        v4 = v5[0] ^ v5[1] ^ v3;
16        if ( v4 == -1153374642 )
17        {
18            v4 = -1153374641;
19        }
20        else if ( (v4 & 0xFFFF0000) == 0 )
21        {
22            v4 |= (v4 | 0x4711) << 16;
23        }
24        dword_1398004 = v4;
25        dword_1398008 = ~v4;
26    }
27    else
28    {
29        dword_1398008 = ~dword_1398004;
30    }
31 }

```

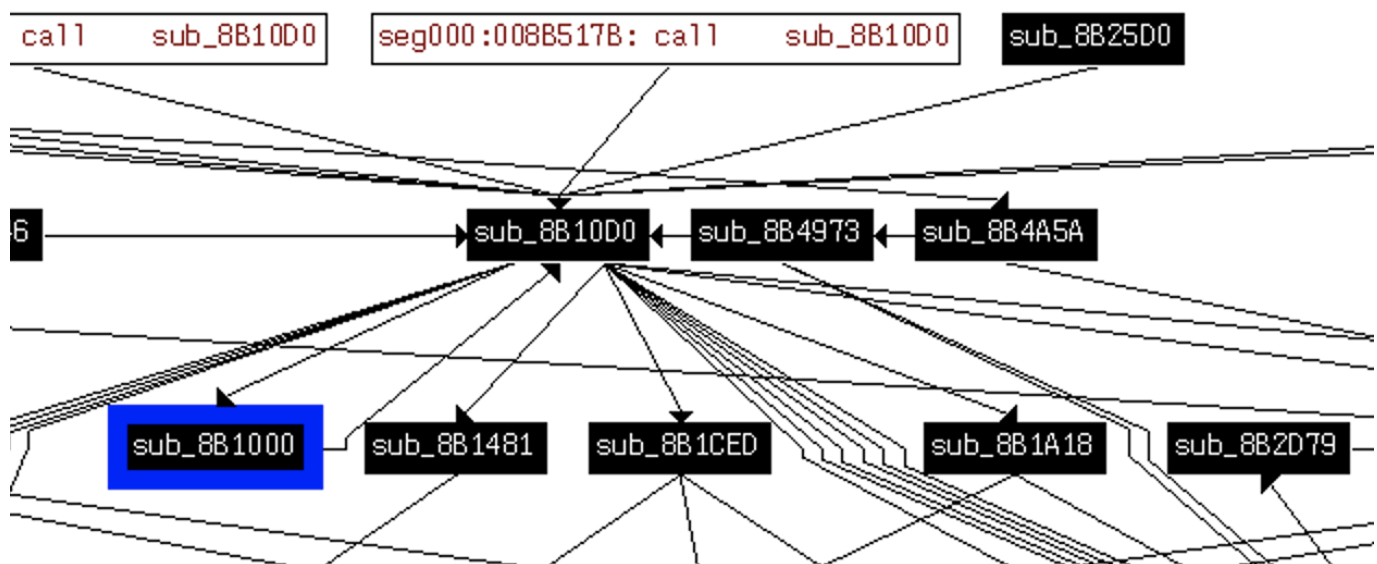
```

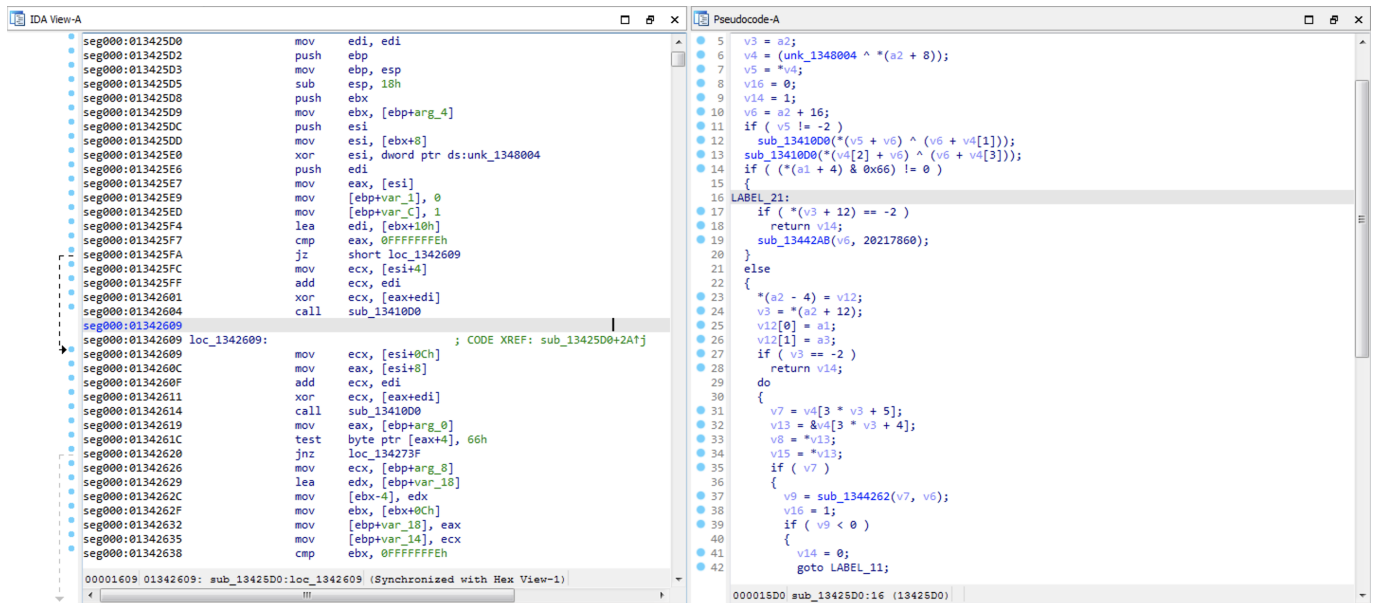
91:          ; CODE XREF: __security_init_cookie+23↑j
          ; __security_init_cookie+27↑j
push esi
lea eax, [ebp+SystemTimeAsFileTime]
push eax
; lpSystemTimeAsFileTime
call GetSystemTimeAsFileTime
mov esi, [ebp+SystemTimeAsFileTime.dwHighDateTime]
xor esi, [ebp+SystemTimeAsFileTime.dwLowDateTime]
call GetCurrentProcessId
xor esi, eax
call GetCurrentThreadId
xor esi, eax
call GetTickCount
xor esi, eax
lea eax, [ebp+var_10]
push eax
call RtlQueryPerformanceCounter

```

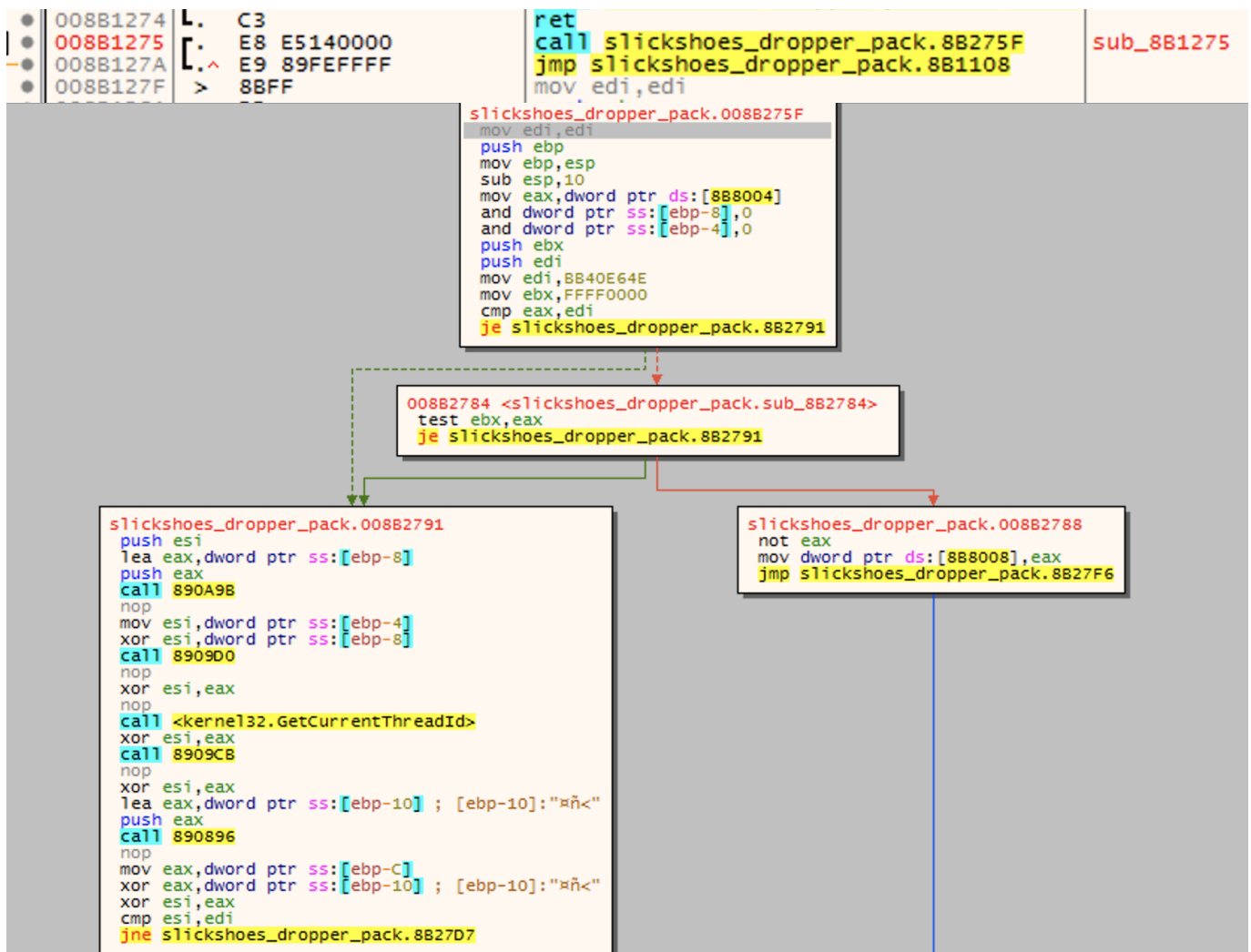
Ở đây sẽ có 02 cách để cách bạn có thể tham khảo tìm ra OEP:

- Chúng ta sẽ dump section không tên mới được giải mã xong và sử dụng tính năng "Xrefs graph to" có trong IDA để tìm hàm gốc. Các bạn load file dump vào IDA và phân tích như file shellcode bình thường. Để thuận tiện cho việc chuyển đổi raw address sang virtual address, tôi sẽ thay đổi base address giống như section đó được thể hiện bên x64dbg. Các địa chỉ virtual address dưới đây có thể khác với trên máy của bạn nhưng cách tìm thì tương tự. Ở đây, tôi sẽ chọn hàm sub_008B1000 (hàm sub_o theo địa chỉ raw address) để phân tích. Kết quả mà IDA trả về như sau: hàm sub_8B1000 => sub_8B10D0 => sub_8B25D0. Tôi đi đến hàm sub_8B25D0 để kiểm tra thì nhận thấy trong hàm này không tồn tại lời gọi hàm đến __security_init_cookie. Khả năng cao IDA phân tích thiếu vài hàm.





- Chúng ta sẽ sử dụng sự hỗ trợ của x64dbg. Chúng ta đi đến hàm sub_8B25D0 và Reanalyze lại chương trình bằng tổ hợp phím Ctrl + A. Tại vị trí 0x8B25D0, chúng ta sẽ tìm các tham chiếu đến hàm hiện tại (Xrefs) thông qua phím X và đi đến các hàm trước đó gọi đến. Nếu thành công các bạn sẽ tìm thấy địa chỉ 0x8B1275 không được gọi bởi bất kỳ hàm nào. Tại đây, tôi kiểm tra lời gọi đến hàm 0x8B275F tại vị trí 0x8B1275 và nhận thấy đây là hàm `__security_init_cookie`. Vì vậy, 0x8B1275 sẽ là OEP.



2. Đối với phương pháp này, chúng ta sẽ tìm các opcode liên quan. Ở hình trên, nếu các bạn để ý thì sẽ thấy các byte tại OEP sẽ có dạng như sau "E8 ? ? ? ? E9 ? ? ? ?". Chúng ta sẽ sử dụng tính năng "Find Pattern" có trên x64dbg để tìm các byte này và nhận thấy khá nhiều chỗ có các byte này. Ta sẽ kiểm tra từng vị trí đó và nhận thấy 0x8B1275 là vị trí mà chúng ta cần tìm.

Address	Data
008B1275	E8 E5 14 00 00 E9
008B2755	E8 51 18 00 00 E9
008B3A8F	E8 D7 FB FF FF E9
008D2553	E8 C8 27 CB FB E9
008D81D6	E8 B0 B0 2A 25 E9
008E9E0C	E8 CD 93 64 F6 E9
008F8850	E8 8E C4 35 04 E9
00923E11	E8 5E FE 30 18 E9
0092A35C	E8 21 7D A7 08 E9
00944D88	E8 87 7E B1 20 E9
0094515A	E8 C9 F1 F6 57 E9
0094D881	E8 D2 B5 EA F6 E9
0094DA06	E8 C5 D2 20 28 E9
0094FD59	E8 DB B7 53 51 E9
00952D08	E8 39 C4 C7 12 E9
00958AD0	E8 33 B6 77 17 E9
0096F355	E8 B7 F4 BB 22 E9
009717E1	E8 47 EB 4E FA E9
009792EA	E8 EF 55 27 2E E9
0097EA92	E8 A0 65 26 61 E9
00981716	E8 77 44 5C 13 E9
0099A464	E8 AD C9 3C 45 E9
009BDA00	E8 C3 F3 ED 6F E9
009C5EB6	E8 80 FE 6C 43 E9
009E0055	E8 51 32 74 F7 E9
009FDC4C	E8 D6 1E 9C 92 E9
009FF9BA	E8 64 42 09 5C E9
00A0E4A8	E8 57 3F 30 53 E9
00A2AF0E	E8 9F 25 74 91 E9

4. 2.4. Khôi phục IAT

4.1. 2.4.1. Cách Themida đánh bại Scylla

Nói một cách ngắn gọn, khi bạn sử dụng Scylla thì cách bạn phải cung cấp cho Scylla địa chỉ OEP như chúng ta tìm ở trên và địa chỉ bảng IAT. Với dữ liệu được cung cấp, Scylla sẽ bắt đầu quét từ OEP cho đến khi gặp các lời gọi hàm trực tiếp "FF 15 ? ? ? ?" và kiểm tra địa chỉ được gọi ấy có phải là địa chỉ của các hàm WinAPI hợp lệ hay không.

Chúng ta quay lại hàm `___security_init_cookie` để kiểm tra các các lời gọi call đến các WinAPI trong hàm này. Các bạn để ý tại địa chỉ 0x8B27AB, đây là lệnh call gián tiếp, địa chỉ mà lệnh này nhảy tới sẽ là $0x008B27AB + 0x753FECA + 5 = 0x75CB1450$.

008B27A8	.	33F0	xor esi,eax	
008B27AA	.	90	nop	
008B27AB	.	E8 A0EC3F75	call <kernel32.GetCurrentThreadId>	
008B27B0	.	33F0	xor esi,eax	
008B27B2	.	E8 14E2FDFF	call 8909CB	
008B27B7	.	90	nop	
75CB144F	.	90	nop	
75CB1450	✓	EB 05	jmp <JMP.&GetCurrentThreadId>	GetCurrentThreadId
75CB1452	.	90	nop	
75CB1453	.	90	nop	

Rõ ràng, địa chỉ 0x75CB1450 là một lời gọi tới WinAPI hệ thống. Các bạn để ý hình trên, lệnh tại địa chỉ 0x8B27AA là lệnh nop có kích thước 1 byte, cộng với 5 byte của lệnh call bên dưới sẽ là 6 byte. 6 byte này sẽ bằng với kích thước của lệnh call trực tiếp. Như vậy, Themida đã chủ động thay đổi các lệnh call trực tiếp thành gián tiếp để đánh bại Scylla. Mục tiêu của chúng ta sẽ là biến các lệnh call gián tiếp này trở về lệnh call trực tiếp để Scylla có thể nhận diện ra được.

Một trường hợp khác, chúng ta sẽ đi đến lệnh call ở vị trí 0x8B27B2 bên dưới lệnh call WinAPI GetCurrentThreadId như ở hình trên. Thay vì gọi đến WinAPI như ở ví dụ trên thì lần này lời gọi WinAPI sẽ được nhét vô trong một hàm khác, và trong hàm này sẽ nhảy tới WinAPI của chương trình. Trong hàm con đó sẽ gọi nhiều lệnh jump để tính toán các biểu thức phức tạp rồi sẽ return về chương trình chính. Tôi sẽ gọi các hàm này là multijump.

✓ E9 98F72F76	jmp <kernel32.GetTickCount>	
CC	int3	
✓ E9 11000000	jmp 961988	
05 5A8B6881	add eax,81688B5A	
2667:14 BD	adc al,BD	
B2 03	mov dl,3	
80B9 FE5FAC75 05	cmp byte ptr ds:[ecx+75AC5FFE],5	
90	nop	
90	nop	
90	nop	
90	nop	

4.2. 2.4.2. Convert lệnh call gián tiếp qua trực tiếp

Trước khi thay đổi các lệnh call, chúng ta cần xác định vị trí bảng IAT mà Scylla sử dụng. Tuy nhiên, Themida lại sử dụng call gián tiếp để gọi đến trực tiếp các hàm WinAPI mà không dùng đến IAT. Vì vậy, chúng ta sẽ thử tìm xem địa chỉ các hàm WinAPI sẽ được lưu ở đâu trong chương trình. Chúng ta sẽ sử dụng địa chỉ WinAPI GetCurrentThreadId để tìm kiếm. Các bạn sử dụng tính năng Find Pattern và nhập địa chỉ của GetCurrentThreadId vào và có kết quả như hình dưới. Các bạn follow theo địa chỉ này và quan sát. Tại đây có cả địa chỉ hàm con chứa lệnh jump tới GetTickCount ở trên. Khả năng cao đây chính là bảng IAT mà Themida sử dụng. Như các bạn thấy ở hình dưới, bảng IAT của chúng ta sẽ bắt đầu từ 0x1346000 đến 0x13460E0.

Pattern: 5014C676					
Address		Data			
01346084		50 14 C6 76			
01345FE4	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
01345FF4	00 00 00 00	00 00 00 00	00 00 00 00	00 00 8E 00
01346004	29 06 8E 00	D2 09 8E 00	75 0A 8E 00	80 0D 8E 00)...ò...u...°...
01346014	85 0D 8E 00	99 0E 8E 00	13 0F 8E 00	CC 0F 8E 00	μ.....İ...
01346024	00 00 8F 00	00 00 90 00	08 08 90 00	10 08 90 00
01346034	07 0C 90 00	F8 79 C6 76	F0 0C 90 00	00 00 95 00	...øy&vð...
01346044	ED 00 95 00	E7 01 95 00	79 02 95 00	08 03 95 00	í...ç...y...>
01346054	EA 03 95 00	EF 03 95 00	E0 04 95 00	8E 05 95 00	è...î...â...>
01346064	00 00 96 00	80 10 96 00	8B 13 96 00	C0 13 96 00»...À...
01346074	8E 14 96 00	38 15 96 00	10 16 96 00	E0 16 96 00	...8...à...
01346084	50 14 C6 76	E6 16 96 00	EB 16 96 00	B1 17 96 00	P.&væ...ë...±...
01346094	87 18 96 00	6F 19 96 00	74 19 96 00	CC 19 96 00	...o...t...İ...
013460A4	80 22 24 77	C0 22 24 77	F4 1A 96 00	15 1C 96 00	."\$wÀ"\$wô...
013460B4	FB 1C 96 00	00 1D 96 00	05 1D 96 00	C9 14 C6 76	û.....É.&v
013460C4	AB 1D 96 00	00 00 97 00	FB 08 97 00	2E 19 C6 76	«...F&...û...&v
013460D4	D3 09 97 00	46 E0 24 77	00 00 98 00	4C 08 98 00	Ó...F&\$w...L...
013460E4	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
013460F4	F8 2E 34 01	9E 3C 34 01	7A 53 34 01	C7 13 34 01	ø.4...<4.ZS4.Ç.4.

Như vậy chúng ta đã có đầy đủ dữ liệu, ý tưởng của chúng ta sẽ như sau:

- Tìm cách lệnh call gián tiếp theo pattern "90 E8 ?? ?? ?? ??" hoặc "E8 ?? ?? ?? ?? 90" trong chương trình. Nếu tìm thấy, ta sẽ kiểm tra hàm được gọi đến có nằm trong bảng IAT này không. Nếu có chúng ta sẽ sửa lệnh call gián tiếp thành lệnh call trực tiếp "FF 15 ?? ?? ?? ??", còn không thì bỏ qua
- Sau đó chúng ta sẽ sửa bảng IAT, chúng ta sẽ duyệt từng DWORD trong bảng IAT và kiểm tra. Nếu là địa chỉ của WinAPI luôn thì ta sẽ bỏ qua, còn nếu là hàm multijump chúng ta sẽ cho x64dbg tìm kiếm lệnh call hoặc jump đầu tiên nhảy tới WinAPI và sửa địa chỉ hàm multijump trong bảng IAT thành địa chỉ WinAPI mà chúng ta mới tìm thấy.

Đoạn script mà tôi sử dụng như sau, mặc dù chưa được tối ưu lắm nhưng cũng hoàn thành được mục tiêu mà chúng ta đã đề ra. Lưu ý, trong lúc chạy có thể sẽ phát sinh ra lỗi thì chương trình sẽ truy cập tới các địa chỉ không tồn tại, những lúc như thế, các bạn cần tạm dừng script bằng cách đặt breakpoint tại vị trí mà script này truy cập tới vị trí không hợp lệ và sửa giá trị thành ghi thành một địa chỉ bất kì mà hợp lệ và tiếp tục chạy.

Đoạn script convert call gián tiếp thành trực tiếp trường hợp "E8 ?? ?? ?? ?? 90"

```
$startAddSec = <first address of section text>
$endAddSec = <end address of section text>
$startAddIAT = <first address of IAT>
$endAddIAT = <end address of IAT>
```

MAIN:

```
find $startAddSec, "E8 ?? ?? ?? ?? 90"
$nextCall = $result
cmp $nextCall, 0
je END
$callAddRelative = dis.imm($nextCall)
$currentAddIAT = $startAddIAT
```

FINDMULTIJUMPINIAT:

```
$dwordIAT = ReadDword($currentAddIAT)
cmp $dwordIAT, $callAddRelative
je CONVERTTODIRECTCALL
$currentAddIAT = $currentAddIAT + 4
cmp $currentAddIAT, $endAddIAT
jg NEXTCALL
jmp FINDMULTIJUMPINIAT
```

CONVERTTODIRECTCALL:

```
log {a:$nextCall}
asm $nextCall, " call [0x{x:$currentAddIAT}]"
jmp NEXTCALL
```

NEXTCALL:

```
$startAddSec = $nextCall + 1
jmp MAIN
```

END:

```
log end
```

Đoạn script convert call gán tiếp thành trực tiếp trường hợp "90 E8 ?? ?? ?? ??"


```

$startAddSec = <first address of section text>
$endAddSec = <end address of section text>
$startAddIAT = <first address of IAT>
$endAddIAT = <end address of IAT>

MAIN:
    find $startAddSec, "90 E8 ?? ?? ?? ??"
    $nextCall = $result
    cmp $nextCall, 0
    je END
    $callAddRelative = dis.imm($nextCall + 1)
    $currentAddIAT = $startAddIAT

FINDMULTIJUMPINIAT:
    $dwordIAT = ReadDword($currentAddIAT)
    cmp $dwordIAT, $callAddRelative
    je CONVERTTODIRECTCALL
    $currentAddIAT = $currentAddIAT + 4
    cmp $currentAddIAT, $endAddIAT
    jg NEXTCALL
    jmp FINDMULTIJUMPINIAT

CONVERTTODIRECTCALL:
    log {a:$nextCall}
    asm $nextCall, " call [0x{x:$currentAddIAT}]"
    jmp NEXTCALL

NEXTCALL:
    $startAddSec = $nextCall + 1
    jmp MAIN

END:
    log end

```

Đoạn script sửa bảng IAT

\$endAddIAT = <first address of IAT>
\$currentAddIAT = <end address of IAT>

MAIN:

```
cmp [$currentAddIAT], 0
je NEXTDWORDIAT
jmp FINDADDRESSAPIINMULTIJUMP
```

NEXTDWORDIAT:

```
$currentAddIAT = $currentAddIAT + 4
cmp $currentAddIAT, $endAddIAT
jg END
jmp MAIN
```

FINDADDRESSAPIINMULTIJUMP:

```
eip = [$currentAddIAT]
```

RUNTILLLIBRARY:

```
sti
cmp eip, 70000000
jg FINDJUMPTOADDRESSAPI
jmp RUNTILLLIBRARY
```

FINDJUMPTOADDRESSAPI:

```
$opcodeJump = ReadWord(eip)
cmp $opcodeJump, 25FF
je RUNTILLLIBRARY
$opcodeJump = ReadByte(eip)
cmp $opcodeJump, EB
je RUNTILLLIBRARY
cmp $opcodeJump, E9
je RUNTILLLIBRARY
cmp $opcodeJump, 55
je BEFOREINSTRUCTION
[$currentAddIAT] = eip
log {x:$currentAddIAT}
jmp NEXTDWORDIAT
```

BEFOREINSTRUCTION:

```
$opcodeMove = ReadWord(eip + 1)
cmp $opcodeMove, EC8B
jne RUNTILLLIBRARY
$moveEDI = ReadByte(eip - 2)
cmp $moveEDI, 8B
jne MAYBEVALIDAPI
eip = eip - 2
jmp FINDJUMPTOADDRESSAPI
```

MAYBEVALIDAPI:

```
[$currentAddIAT] = eip  
log {x:$currentAddIAT}  
jmp NEXTDWORDIAT
```

END:

```
log end
```

Nếu thành công khi dùng Scylla để dump thì các bạn sẽ có kết quả như hình dưới. Các bạn lưu ý là đối với cách unpack hiện tại mà tôi hướng dẫn sẽ không đúng hoàn toàn trong mọi trường hợp. Ví dụ như trường hợp hiện tại thì hàm CreateFileW sẽ bị nhận nhầm sang Basep8BitStringToDynamicUnicodeString. Đối với trường hợp này các bạn nên xem xét các tham số push vào và xem các hàm WinAPI liên quan tới mà sửa lại trong Scylla cho phù hợp.

```
IDA View-A  
push ebp  
mov ebp, esp  
mov eax, 2808h  
call _alloca_probe  
mov eax, dword_1398004  
xor eax, ebp  
mov [ebp+var_4], eax  
push esi  
push 0 ; hTemplateFile  
push 0 ; dwFlagsAndAttributes  
push 2 ; dwCreationDisposition  
push 0 ; lpSecurityAttributes  
push 0 ; dwShareMode  
push 10000000h ; dwDesiredAccess  
push offset FileName ; "c:\\windows\\Web\\taskenc.exe"  
mov [ebp+NumberOfBytesWritten], 0  
call CreateFileW  
mov esi, eax  
cmp esi, 0FFFFFFFFh  
jnz short loc_1391056  
  
xor eax, eax  
pop esi  
mov ecx, [ebp+var_4]  
xor ecx, ebp  
call sub_1391000  
mov esp, ebp  
pop ebp  
retn 10h  
  
loc_1391056:  
push ebx  
mov ebx, WriteFile  
push edi  
push 0 ; lpOverlapped  
lea eax, [ebp+NumberOfBytesWritten]  
push eax ; lpNumberOfBytesWritten  
push 177200h ; nNumberOfBytesToWrite  
push offset dword_1398020 ; lpBuffer  
push esi ; hFile  
call ebx ; WriteFile  
push 2800h ; Size  
lea ecx, [ebp+Buffer]  
push 41h ; 'A'  
push Val  
  
Pseudocode-A  
1 int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, ir  
2 {  
3     HANDLE FileW; // esi  
4     BOOL (__stdcall *v6)(HANDLE, LPCVOID, DWORD, LPDWORD, LPOVERLAPPED); // ebx  
5     int v7; // edi  
6     DWORD NumberOfBytesWritten; // [esp+4h] [ebp-2808h] BYREF  
7     char Buffer[10240]; // [esp+8h] [ebp-2804h] BYREF  
8  
9     NumberOfBytesWritten = 0;  
10    FileW = CreateFileW(FileName, 0x10000000u, 0, 0, 2u, 0, 0);  
11    if ( FileW == (HANDLE)-1 )  
12        return 0;  
13    v6 = WriteFile;  
14    WriteFile(FileW, dword_1398020, 0x177200u, &NumberOfBytesWritten, 0);  
15    memset(Buffer, 65, sizeof(Buffer));  
16    v7 = 7168;  
17    do  
18    {  
19        v6(FileW, Buffer, 10240, &NumberOfBytesWritten, 0);  
20        Sleep(0xAu);  
21        --v7;  
22    }  
23    while ( v7 );  
24    CloseHandle(FileW);  
25    return 1;  
26 }  
  
0000042C_WinMain@16:26 (139102C)  
Output
```