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.

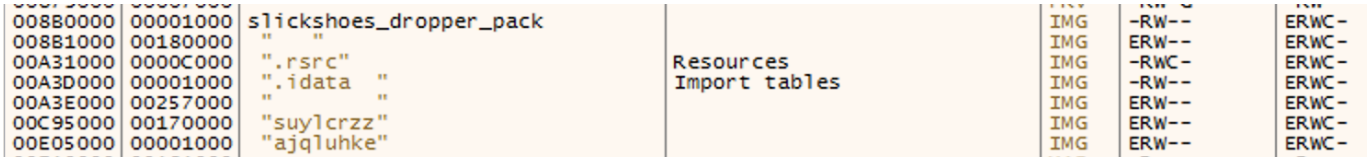
2. 2.2. Giải mã các section về đúng các byte gốc

HIện tại, sample mà chúng ta phân tích đang dừng ở Entry Point. Vấn đề cần đặt ra hiện tại là làm sao thể 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.

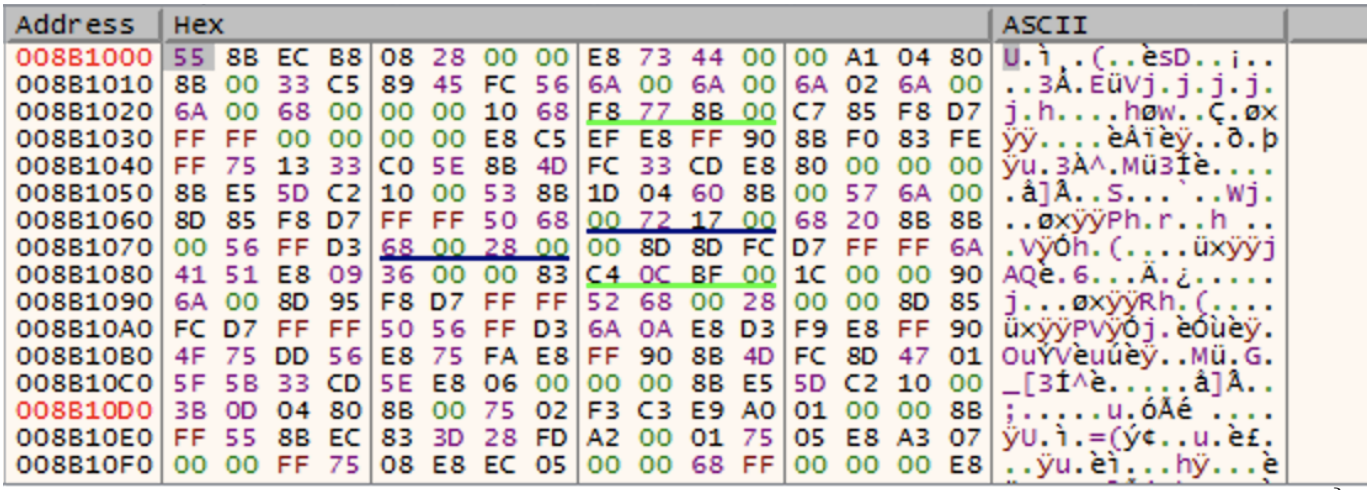
Theo lí thuyết, anti-debug đã có ScyllaHide giải quyết, và khi chúng ta bấm F9 (Run) thì mã độc sẽ nhảy vào vòng lặp kết nối tới C&C để chờ lệnh. Nhưng cuộc đời đâu phải lúc nào cũng suôn sẻ như vậy được, nếu config C&C là một file riêng biệt không nằm trong bản thân mã độc thì khả năng cao khi ta bấm 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 GetCurrentProccessID. 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

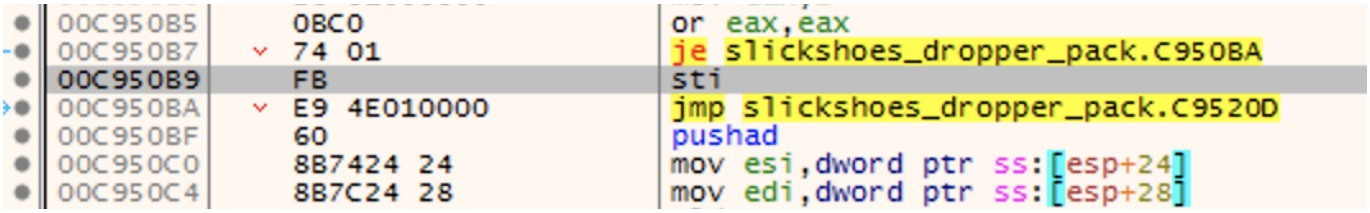
section này không giống các byte có trong một chương trình bình thường, chẳng hạn như "55 8B EC" hay "E8 ? ? ? ?". Khả năng cao 01 trong 02 section này là sẽ là section .text của chương trình ban đầu.



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.



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 lí thì sẽ ra sai kết quả cuối, vì thế ta sẽ chuyển cho chương trình xử lí bằng cách bấm Shift + F9.



3. 2.3. Tìm OEP

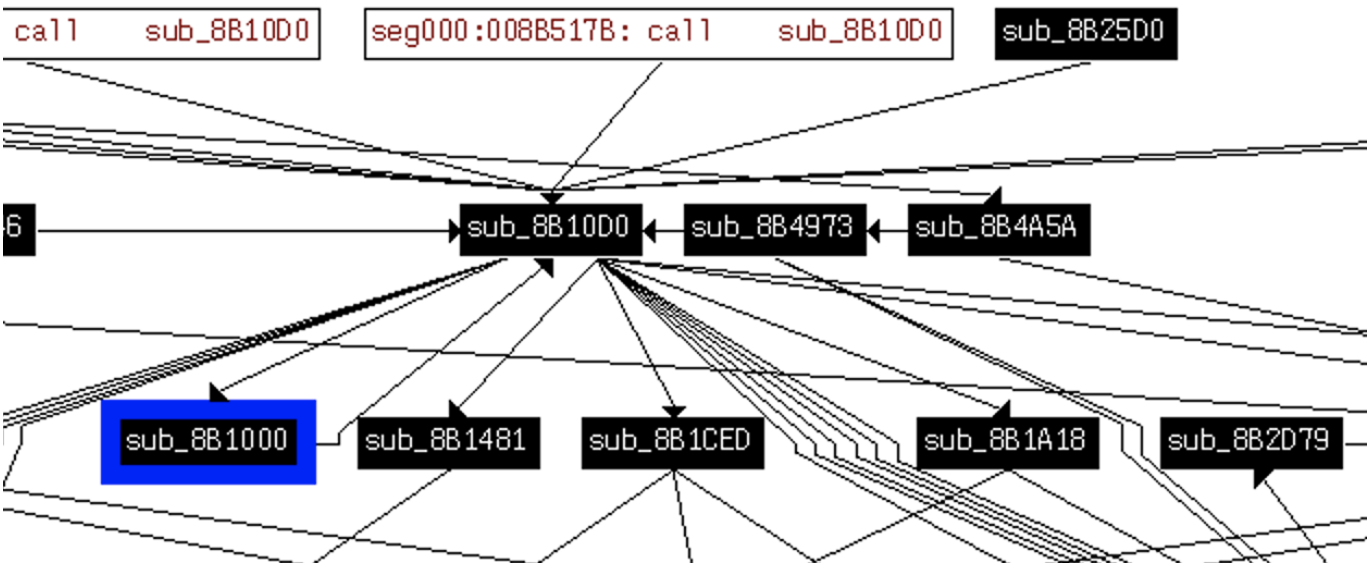
Chúng ta đã giải mã thành công trên bộ nhớ mà mã độc chưa tự chấm dứt bản thân, việc tiếp theo cần làm là tìm OEP. Nếu các bạn hay để ý khi RE sẽ thấy OEP của một chương trình EXE sẽ là lời gọi đến hàm ___security_init_cookie. Hàm này có nhiệm vụ khởi tạo cookie để dùng cho việc chống stack overflow.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

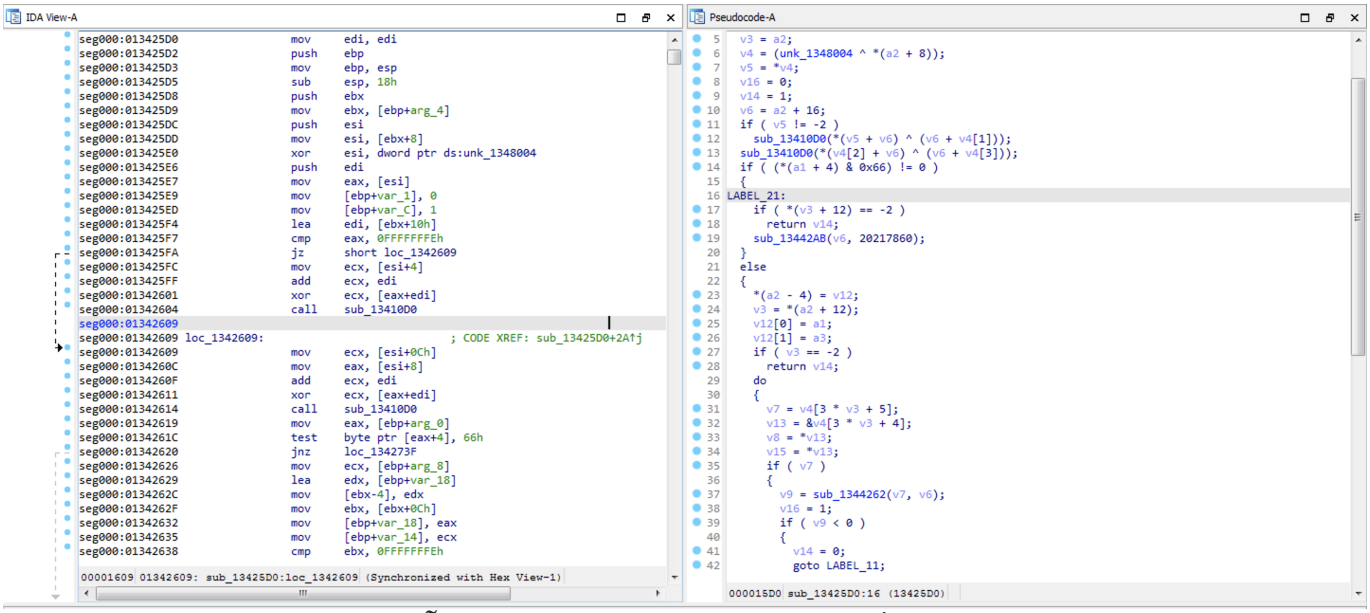
```
        and     [ebp+SystemTimeAsFileTime.dwLowDateTime], 0
        and     [ebp+SystemTimeAsFileTime.dwHighDateTime], 0
        push    ebx
        push    edi
        mov     edi, 0BB40E64Eh
        mov     ebx, 0FFFF0000h
        cmp     eax, edi
        jz      short loc_1392791
        test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

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
```

```
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 }
```

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

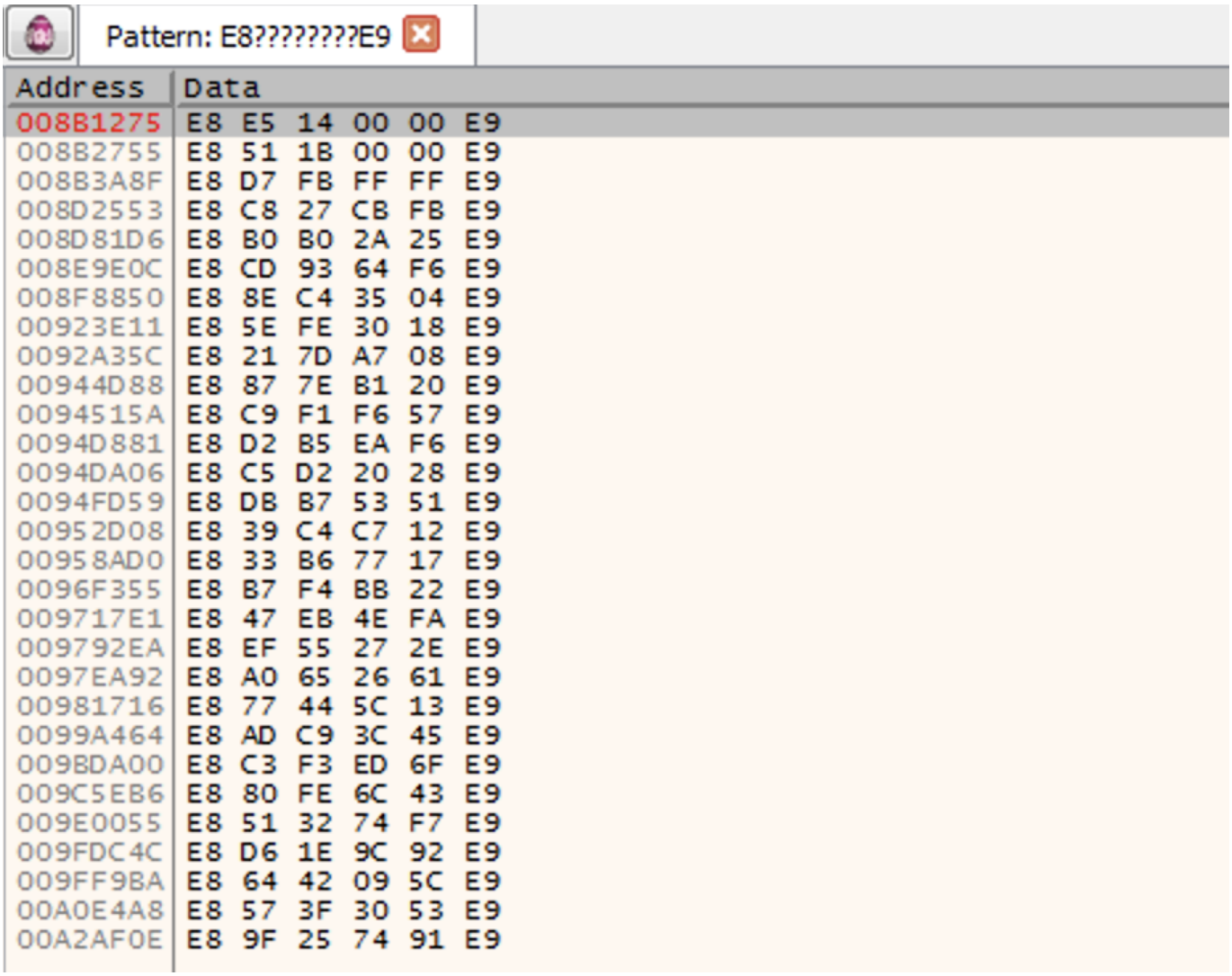
1. Chúng ta sẽ dump section không tên mới được giải mã xong và sử dụng tính năng "Xrefs graph to" có trong IDA để tìm hàm gốc. Các bạn load file dump vào IDA và phân tích như file shellcode bình thường. Để thuận tiện cho việc chuyển đổi raw address sang virtual address, tôi sẽ thay đổi base address giống 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_0 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 kì 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.

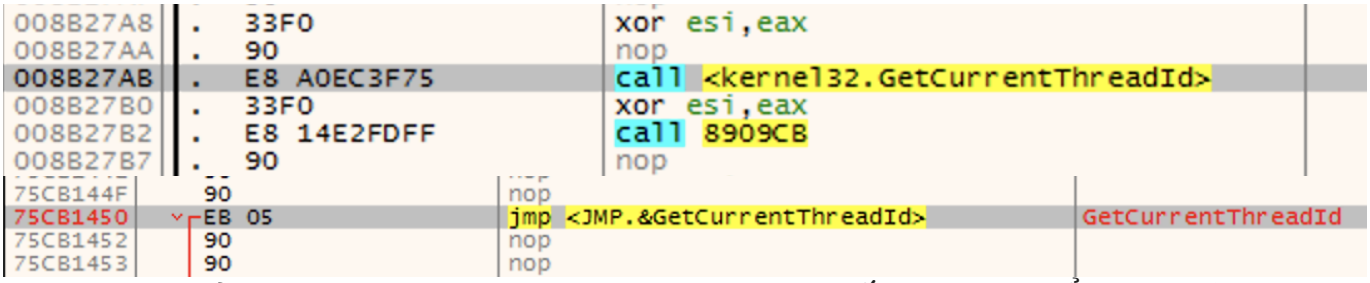


4. 2.4. Khôi phục IAT

4.1. 2.4.1. Cách Themida đánh bại Scylla

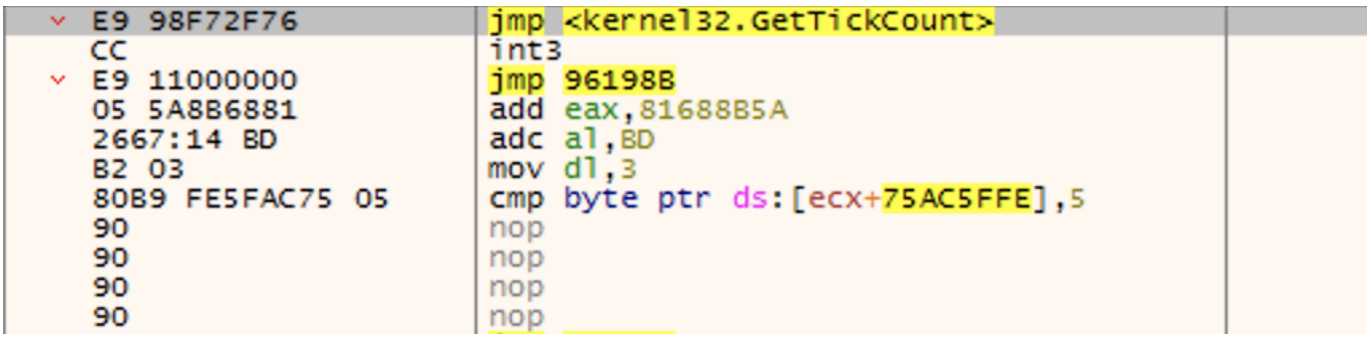
Nói một cách ngắn gọn, khi bạn sử dụng Scylla thì cách bạn phải cung cấp cho Scylla địa chỉ OEP như chúng ta tìm ở trên và địa chỉ bảng IAT. Với dữ liệu được cung cấp, Scylla sẽ bắt đầu quét từ OEP cho đến khi gặp các lời gọi hàm trực tiếp "FF 15 ? ? ? ?" và kiểm tra địa chỉ được gọi ấy có phải là địa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



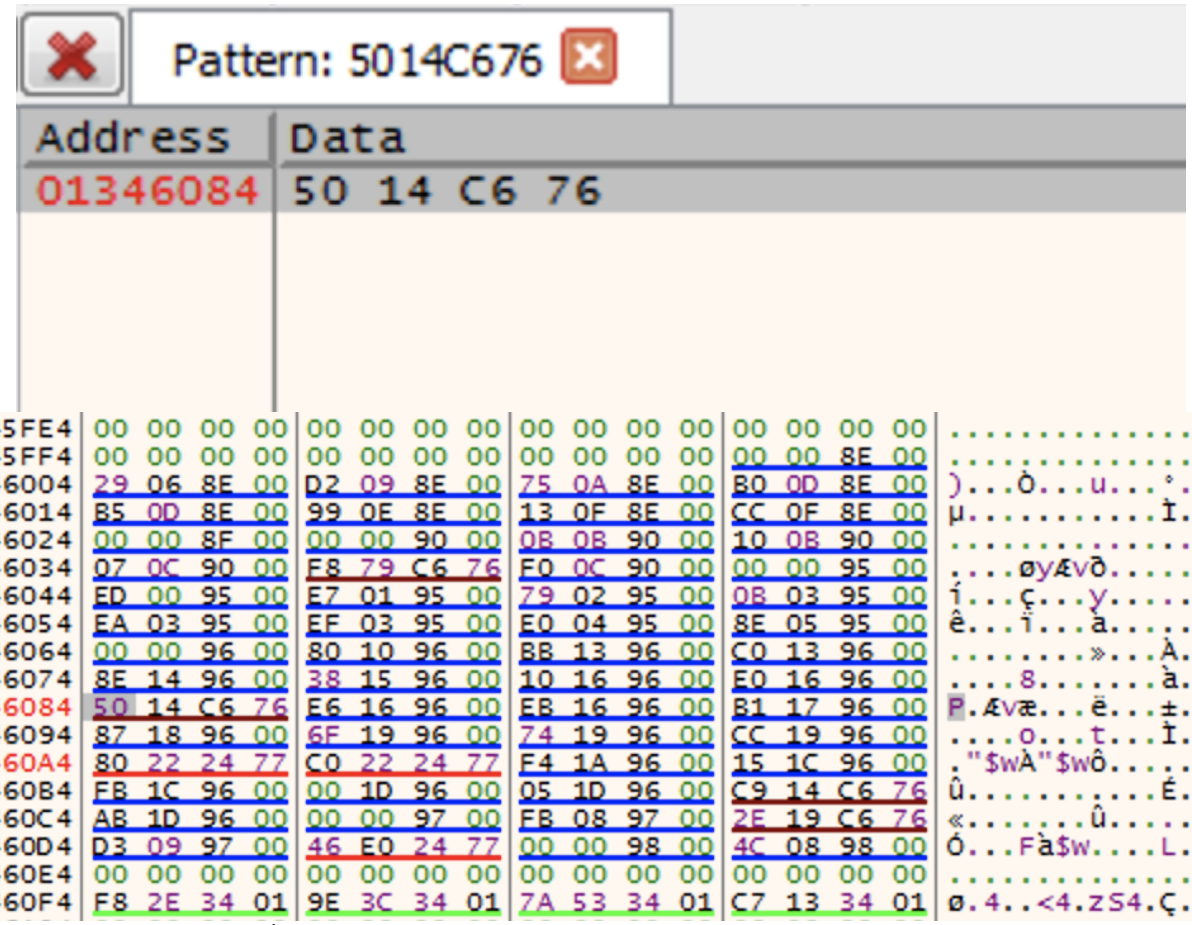
Rõ ràng, địa chỉ 0x75CB1450 là một lời gọi tới WinAPI hệ thống. Các bạn để ý hình trên, lệnh tại địa chỉ 0x8B27AA là lệnh nop có kích thước 1 byte, cộng với 5 byte của lệnh call bên dưới sẽ là 6 byte. 6 byte này sẽ bằng với kích thước của lệnh call trực tiếp. Như vậy, Themida đã chủ động thay đổi các 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.



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.



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 cụng 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 ví trị mà script này truy cập tới vị trí không hợp lệ và sửa giá trị thanh 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 gáin 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.