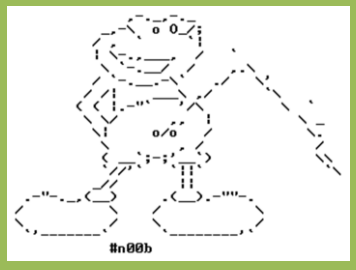
2016

[Cracking with OllyDbg]

Based on OllyDbg tuts of Ricardo Narvaja (CrackLatinos Team)

www.reaonline.net

kienmanowar



17/07/2016

Mục Lục

I. Giới thiệu chung.....2

II. Phân tích và xử lý target.....2

III. Kết luận 23

I. Giới thiệu chung

19/05/2015, tôi hoàn thành OllyDbg_tut27, 17/07/2016 đặt tay lên bàn phím và hoàn thành OllyDbg_Tut28. Hơn 1 năm tôi mới có thời gian để lại viết tiếp tục những thứ còn đang dang dở, kiểu như người ta viết thơ tình mà đang nghĩ thì tụt mịa nó cảm xúc ... đành phải đợi lúc nào đó cảm xúc nó hồi sinh. Mà cứ đợi như thế, rồi đợi mãi, rồi tôi đọc Blog của Yêuchimsẻ (http://blog.yeuchimse.com) có đoạn ***"Thời gian chẳng bao giờ dừng lại, mới đó mà cũng đã được một năm."*** Cuộc sống, công việc nhiều khi cứ cuốn tôi rời xa khỏi đam mê, đôi khi không phải vì thời gian eo hẹp, mà vì viết lách là một nghệ thuật, nhiều khi mở máy lên định viết nhưng rồi nghĩ một hồi tôi đếch nghĩ ra nổi phải mở đầu như thế nào chứ chưa nói tới viết về phần kỹ thuật, thế nên mỗi lần định viết lại là một lần tôi thở dài, đóng máy đi làm việc khác. Có thể sẽ có nhiều bạn đọc đọc các bài viết của tôi, rồi tự đặt câu hỏi **"Sao ông này cứ viết mãi những thứ đơn giản, cơ bản mà ai cũng biết?"**. Trong thế giới quan của tôi, những thứ cơ bản mà chưa làm được thì nói gì đến những điều cao siêu, ngày trước tôi đã không đi từ những thứ cơ bản, nên bây giờ tôi coi mỗi lần viết là một lần tôi học lại.

Lâu lắm rồi tôi mới lại mở đầu bài viết một cách lâm ly, bi đát như thế này, quay trở lại nội dung chính, phần 28 này sẽ là tiếp tục của phần 27 cách đây hơn 1 năm có lẻ. Có lẽ, tôi có thể khẳng định với các bạn rằng, đây là siêu phẩm về unpack UPX đỉnh cao nhất trong suốt sự nghiệp viết lách của tôi, chưa bao giờ các bạn tưởng tượng lại có một bài viết nào về unpack UPX dài đến thế.... , viết xong thực sự tôi cũng cạn lời!!!

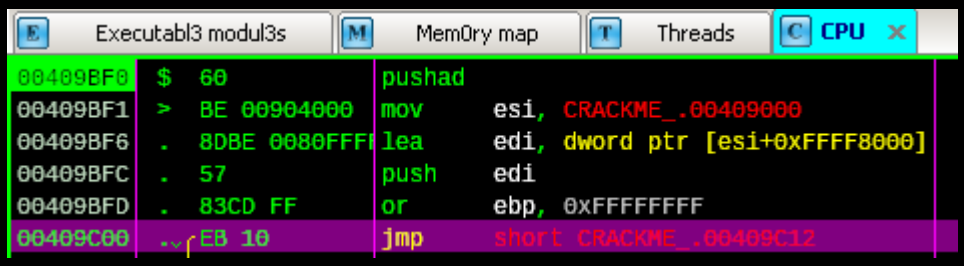
II. Phân tích và xử lý target

Trong phần trước, chúng ta đã tìm hiểu và nắm được cách hoạt động của **IT (Import Table)** và **IAT (Import Address Table)** đối với một PE file. Cũng trong phần trước, tôi chưa đề cập đến cách thực hiện sửa IAT (fix IAT) như thế nào. Tôi biết rằng, sẽ có những bạn đã biết cách làm thế nào để sửa được hoặc có đọc đâu đó các tài liệu nói về quá trình sửa IAT, và có thể trong số các bạn sẽ nghĩ rằng không cần thiết phải tìm hiểu quá trình sửa IAT làm gì, bởi đã có nhiều công cụ hỗ trợ cho phép sửa IAT gần như tự động.

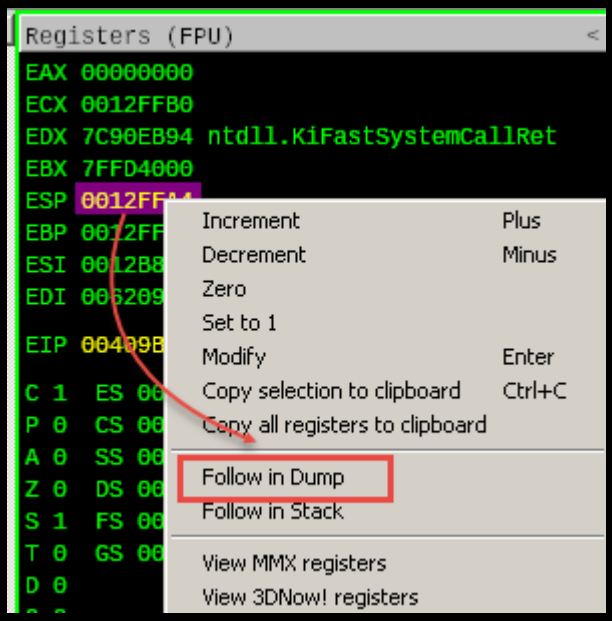
Tuy nhiên, theo quan điểm cá nhân của mình, tôi nghĩ chúng ta nên tìm hiểu phương pháp thực hiện để có thể áp dụng các cách khác nhau tùy theo từng thời điểm, do hiện

nay có rất nhiều trình packers sử dụng các cơ chế nhằm đánh lừa các công cụ và làm cho việc sửa IAT bằng công cụ thất bại. Vì vậy, khi có được kiến thức cơ bản, trong một số trường hợp sẽ giúp chúng ta có thể thực hiện tự chỉnh sửa bằng "cơm" (manual), hoặc có thể tìm được cách khác để sửa nếu như công cụ hiện có không đáp ứng được nhu cầu.

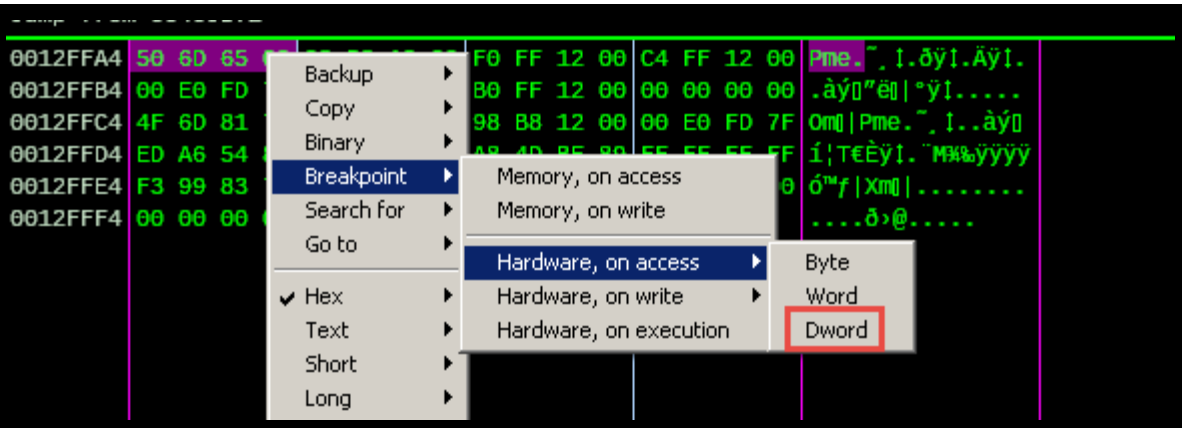
Trong bài viết này, chúng ta sẽ vẫn lại thực hành với Cruehead Crackme được packed bằng UPX, đi từng bước từ quá trình giải nén (unpack) cho tới bước sửa chữa lại bảng IAT (fix IAT). Bước đầu tiên của quá trình unpack như đã biết là đi tìm OEP, load crackme vào OllyDbg, dừng lại tại đây:



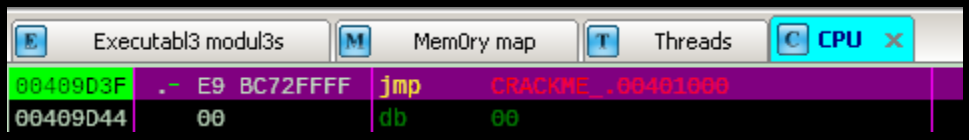
Áp dụng phương pháp **PUSHAD** để tìm tới OEP, nhấn **F7** trace qua lệnh **PUSHAD**. Tại cửa sổ Registers, lựa chọn thanh ghi **ESP**, nhấn chuột phải và chọn **Follow in Dump**.



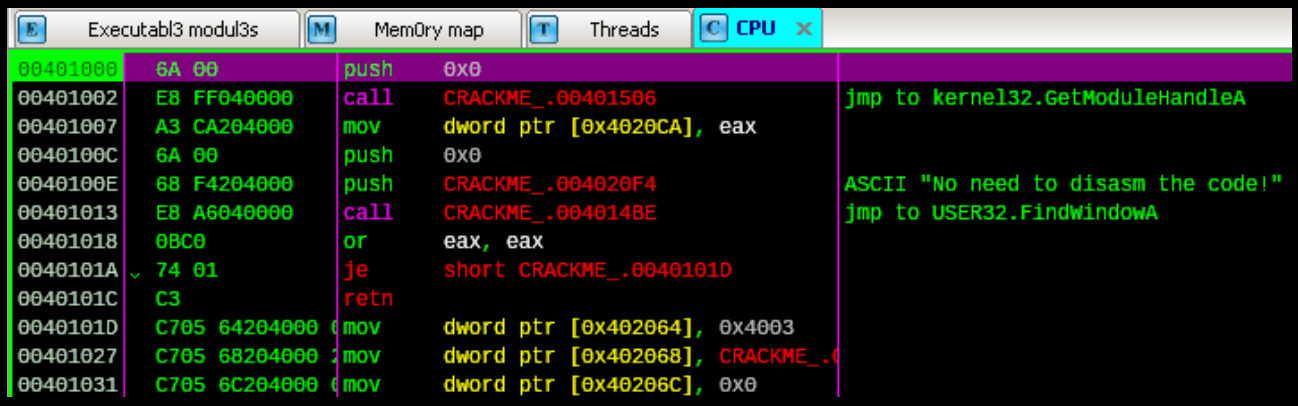
Quan sát tại cửa sổ Dump, đánh dấu 4 bytes đầu tiên và lựa chọn đặt BP như sau:



Sau khi đặt BP xong, nhấn **F9** để Run, ta sẽ dừng lại tại lệnh nhảy tới OEP:

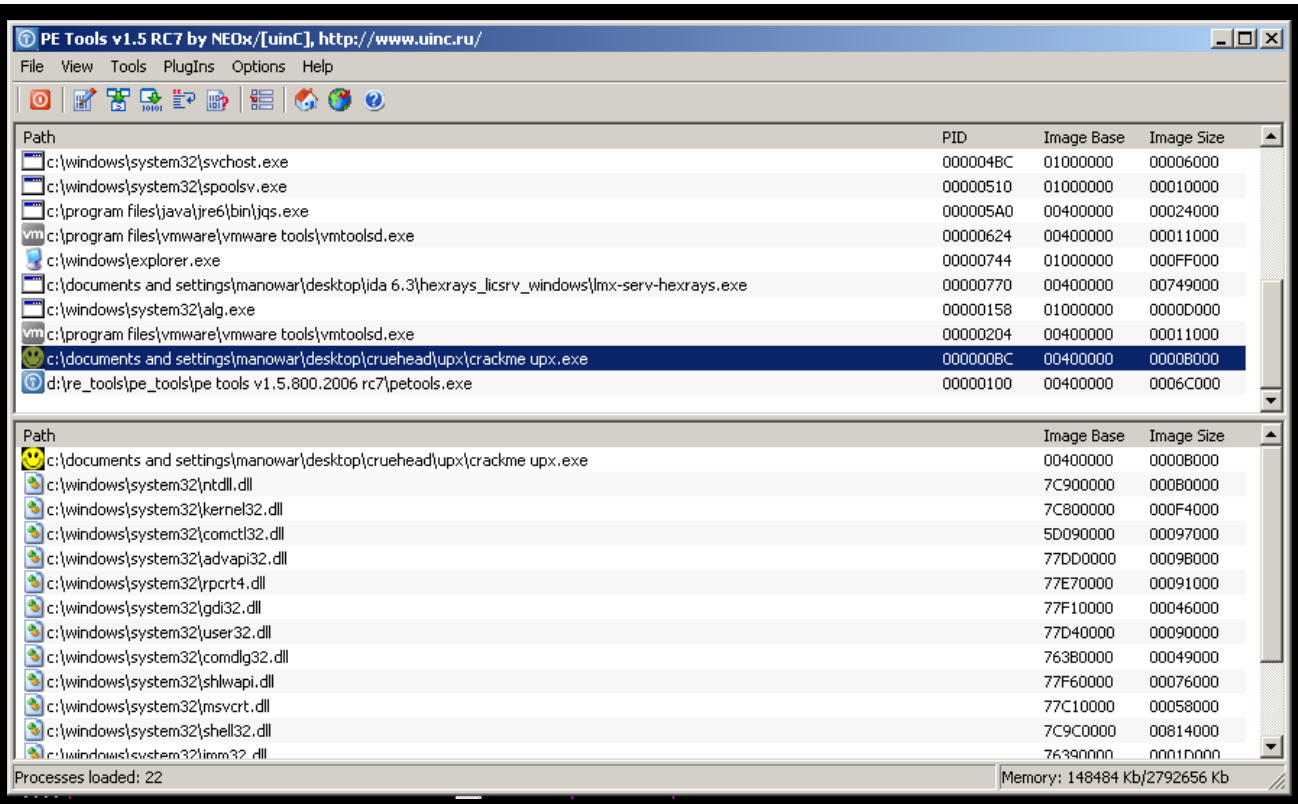


Nhấn **F7** sẽ tới được OEP gốc:

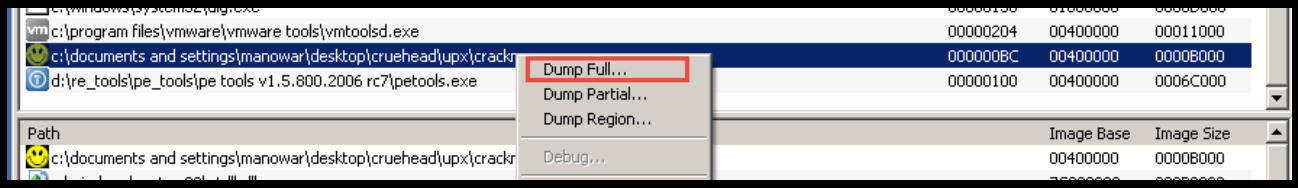


OK, vậy là toàn bộ bước tìm OEP đã xong. Chương trình đã được unpack trong bộ nhớ. Bước tiếp theo ta sẽ thực hiện dump file. Như đã biết, có rất nhiều công cụ có thể được sử dụng để dump file, ngay cả OllyDbg cũng có riêng một plugin OllyDump được dùng để dump file rất tốt, nhưng trong bài viết này tôi sẽ dùng công cụ nổi tiếng là **PE Tools**, các bạn có thể download tại địa chỉ sau: http://www.uinc.ru/files/neox/PE_Tools.shtml.

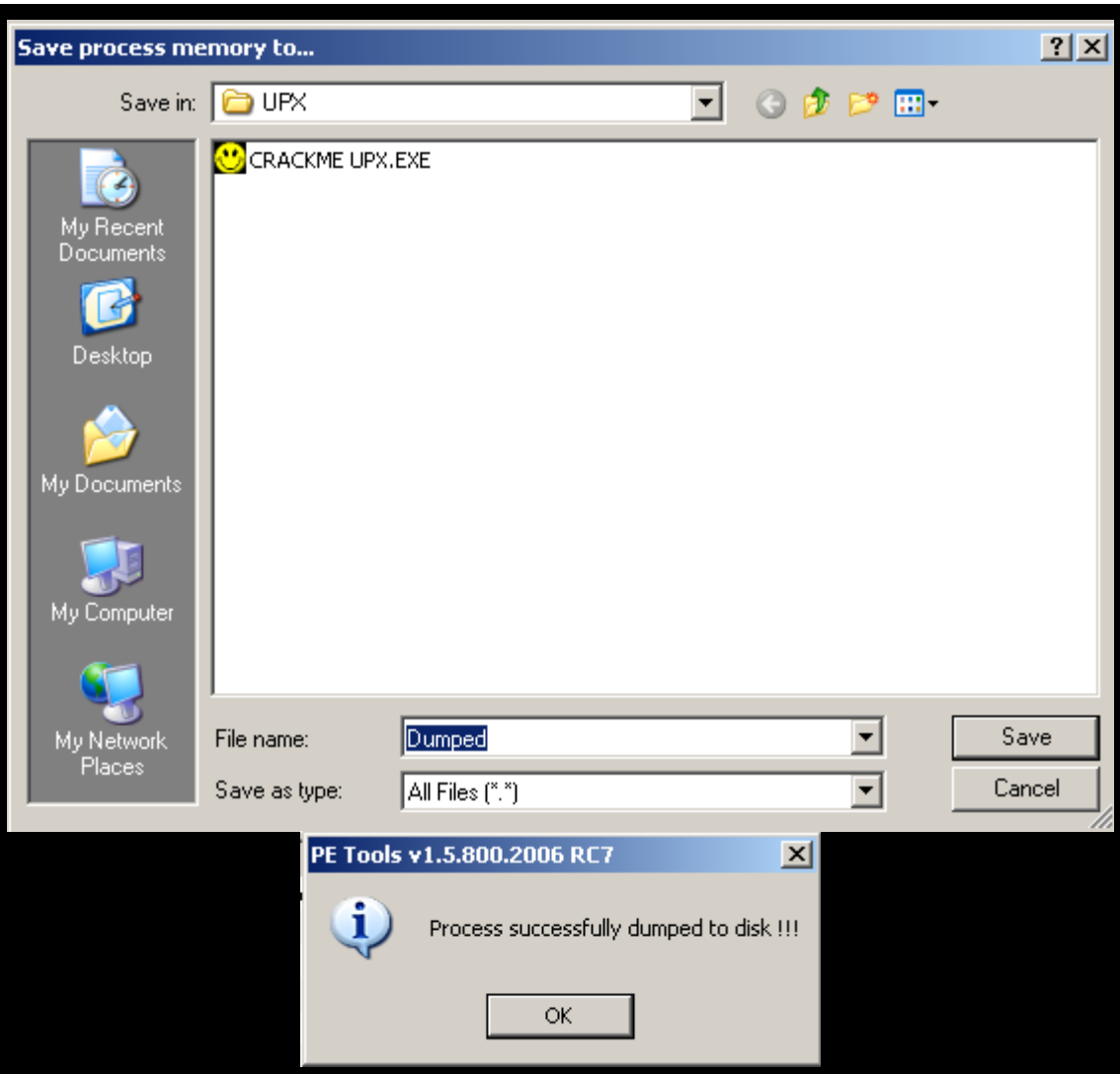
Tôi dùng bản **PE Tools v1.5.800.2006 RC7** có sẵn trong bộ Tools của tôi. Giữ nguyên màn hình OllyDbg hiện đang dừng lại tại OEP, mở công cụ PE Tools:



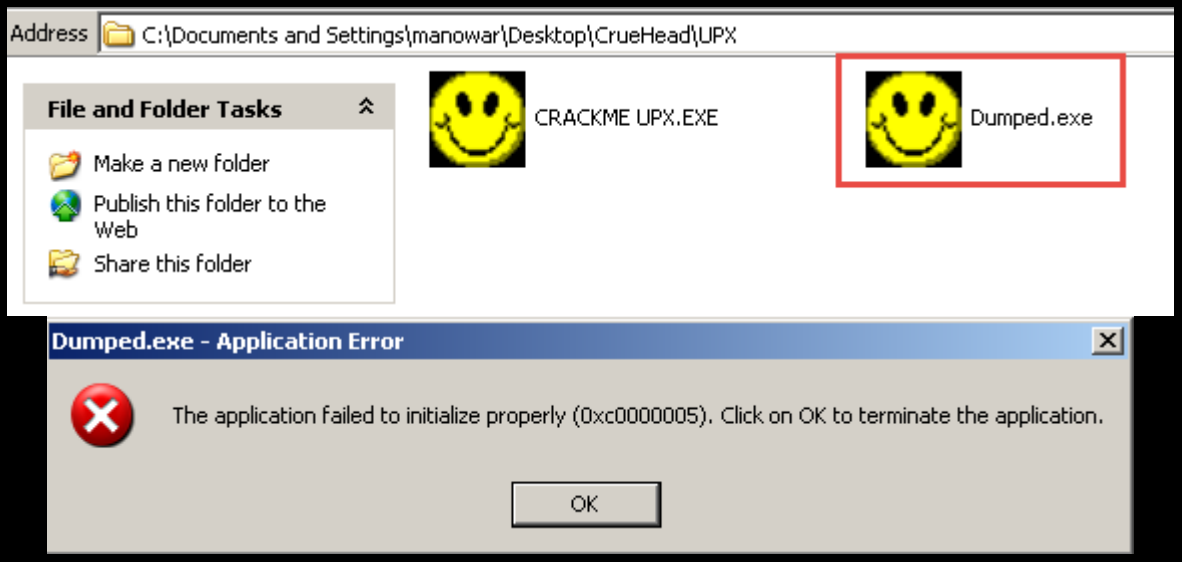
Quan sát tại màn hình của PE Tools ta sẽ thấy có tiến trình **crackme upx.exe**. Chọn tiến trình đó, nhấn chuột phải và chọn **Dump Full**:



Lưu lại tại cùng thư mục của Crackme với tên file là **Dumped.exe**, sau đó đóng công cụ PE Tools. Bước tiếp theo chúng ta sẽ tiến hành sửa lại IAT.

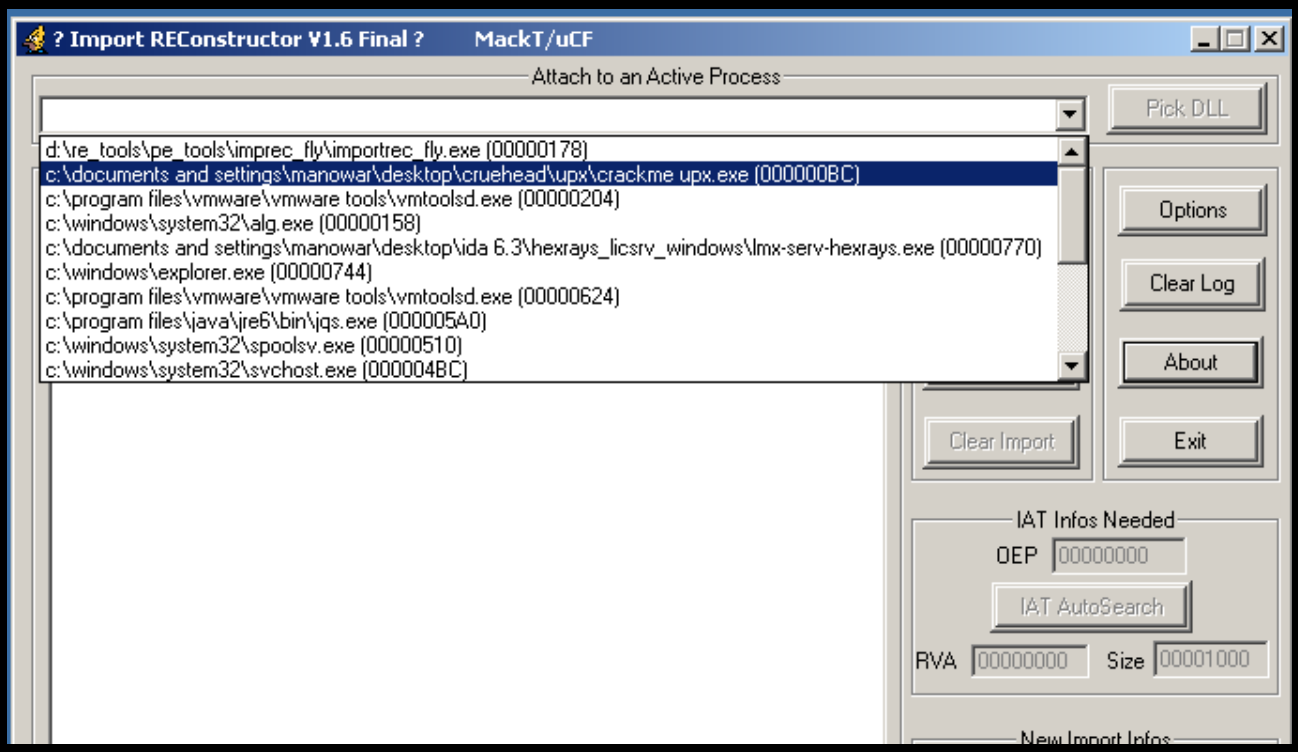


Trước khi tới quá trình sửa, ta chạy thử file **Dumped.exe** lúc chưa sửa lại IAT để xem điều gì sẽ xảy ra, ta nhận được thông báo sau:



Như vậy, rõ ràng file Dumped là file chưa hợp lệ, cần phải sửa lại IAT để file Dumped thực thi được bình thường trên máy của chúng ta. Việc sửa IAT cũng nhằm đảm bảo file Dumped có thể chạy được trên cả các máy khác nữa chứ không phải trên mỗi máy của chúng ta. Để sửa được IAT ta sẽ sử dụng công cụ nổi tiếng là **Import REConstructor (ImpREC)** và tất nhiên vẫn giữ nguyên màn hình OllyDbg hiện đang dừng lại tại OEP của Crackme, bởi ImpREC sẽ căn cứ theo đó để lấy các thông tin. Công cụ này được tôi gửi kèm theo bài viết.

Mở ImpREC, tìm và chọn tiến trình **crackme upx.exe**, tương tự như trong hình:



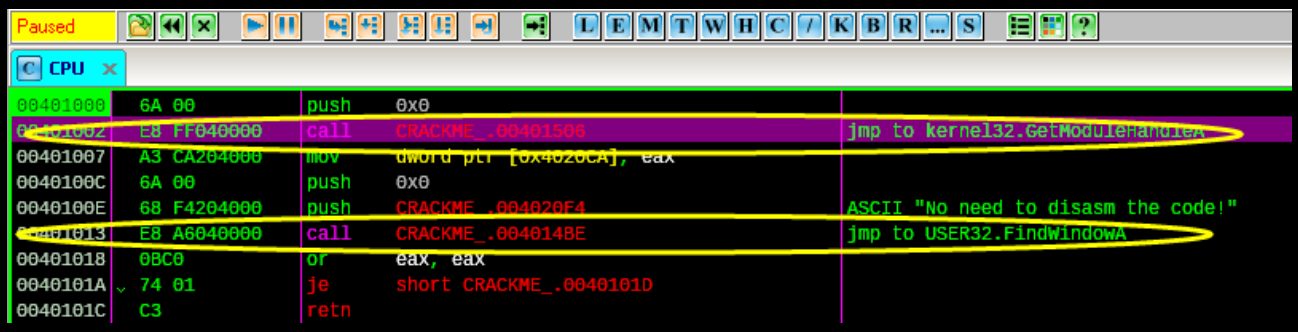
Theo phần 27 cũng như những gì ta đã làm từ đầu đến giờ, chúng ta hiện tại đã tới được OEP, tuy nhiên ta thấy rằng packer đã hủy thông tin về IT. Bây giờ một vấn để hơi khó đối với nhiều bạn cũng như tôi  là làm thế nào để tìm được thông tin địa chỉ bắt đầu và kết thúc của IAT. Như đã viết trong phần 27 về IT, một cấu trúc IID sẽ có DWORD thứ 4 và thứ 5 là cần phải quan tâm: Tại DWORD thứ 4 là một RVA trỏ tới tên của DLL và tại DWORD thứ 5 là một RVA trỏ tới một mảng của cấu trúc **IMAGE_THUNK_DATAs**, đây là những RVAs, mỗi cấu trúc tương ứng với một imported function. Đó chính là thông tin IAT mà chúng ta còn thiếu, vì vậy bằng cách nào đó ta phải đi tìm nó.

Như đã biết, khi một ứng dụng được biên dịch, nó được thiết kế để đảm bảo rằng tất cả các lời gọi hàm API không sử dụng địa chỉ hardcoded mà thay vào đó sẽ làm việc thông qua một con trỏ hàm. Bảng con trỏ này có thể được truy cập bằng nhiều cách.

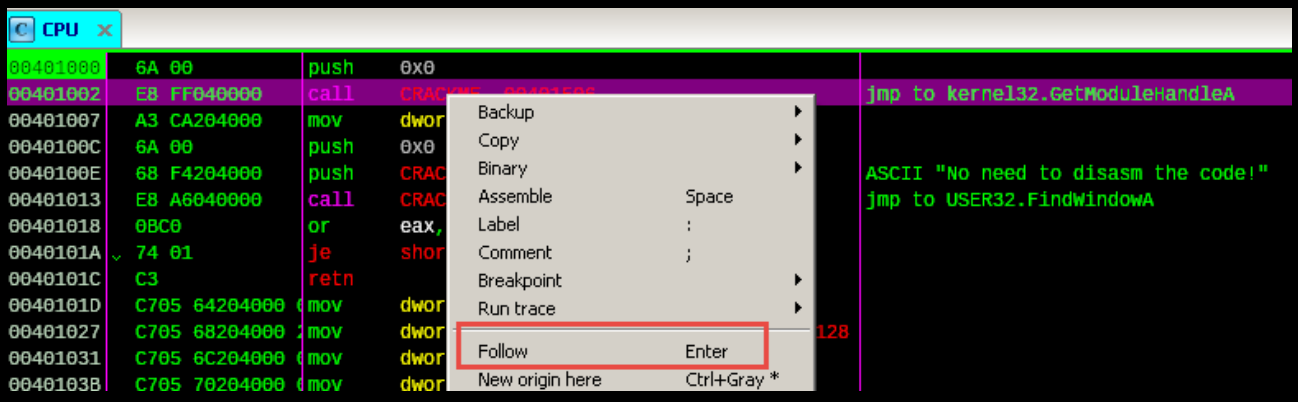
Hoặc trực tiếp bởi một lệnh **call [pointer address]** hoặc thông qua một **jmp thunk table**.

CALL [xxxxxx] hoặc JMP [xxxxxx]

Quay trở lại màn hình OllyDbg, quan sát bên dưới địa chỉ OEP ta sẽ thấy có một số lệnh CALL với chú thích bên cạnh là ***"JMP to ..."***, hàm ý có nghĩa là đây là những lệnh CALL sẽ nhảy tới các hàm APIs mà crackme sử dụng.



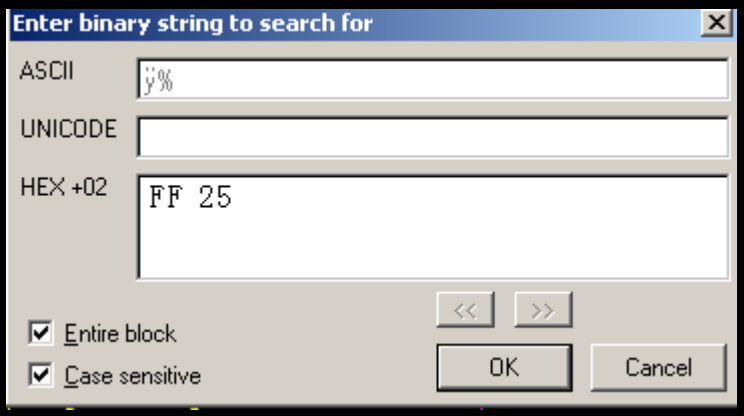
Lựa chọn một lệnh CALL bất kỳ, ví dụ ta chọn lệnh CALL tại địa chỉ **0x00401002**, nhấn chuột phải tại lệnh này và chọn **Follow**:



Ta sẽ tới địa chỉ **0x00401506**, nơi có lệnh nhảy tới hàm API là **GetModuleHandleA**:

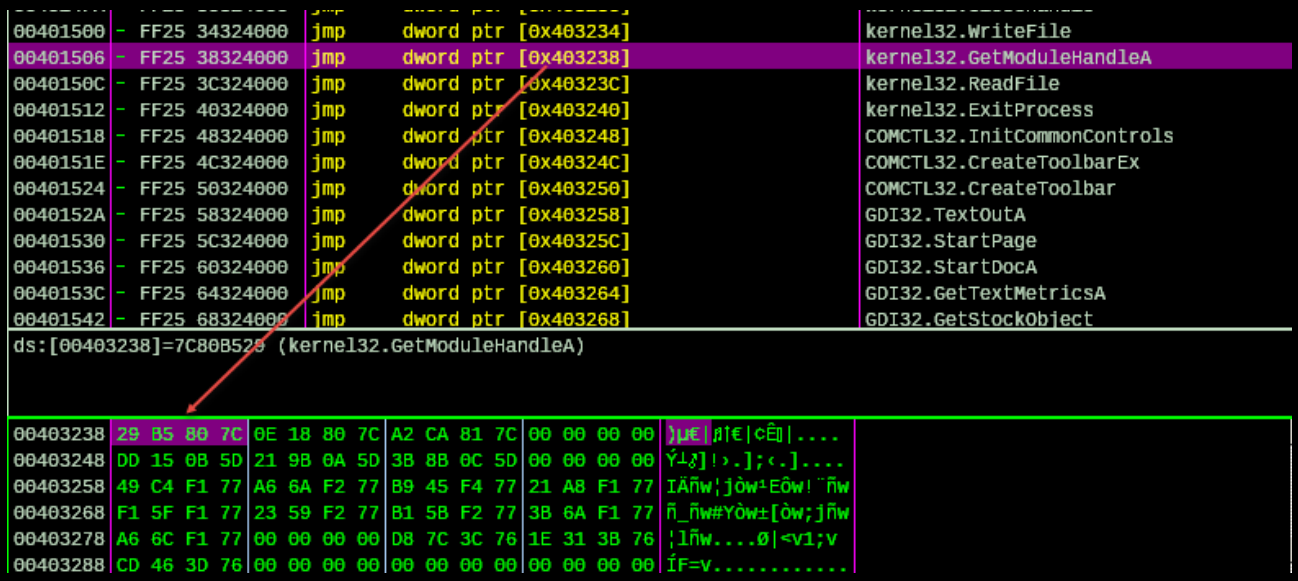
```
CPU
004014D6 - FF25 14324000  jmp     dword ptr [0x403214]   USER32.GetMessageA
004014DC - FF25 1C324000  jmp     dword ptr [0x40321C]   kernel32.GetLocalTime
004014E2 - FF25 20324000  jmp     dword ptr [0x403220]   kernel32.OpenFile
004014E8 - FF25 24324000  jmp     dword ptr [0x403224]   kernel32.GlobalFree
004014EE - FF25 28324000  jmp     dword ptr [0x403228]   kernel32.GlobalAlloc
004014F4 - FF25 2C324000  jmp     dword ptr [0x40322C]   kernel32.lstrlenA
004014FA - FF25 30324000  jmp     dword ptr [0x403230]   kernel32.CloseHandle
00401500 - FF25 34324000  jmp     dword ptr [0x403234]   kernel32.WriteFile
00401506 - FF25 38324000  jmp     dword ptr [0x403238]   kernel32.GetModuleHandleA
0040150C - FF25 3C324000  jmp     dword ptr [0x40323C]   kernel32.ReadFile
00401512 - FF25 40324000  jmp     dword ptr [0x403240]   kernel32.ExitProcess
00401518 - FF25 48324000  jmp     dword ptr [0x403248]   COMCTL32.InitCommonControls
0040151E - FF25 4C324000  jmp     dword ptr [0x40324C]   COMCTL32.CreateToolbarEx
00401524 - FF25 50324000  jmp     dword ptr [0x403250]   COMCTL32.CreateToolbar
0040152A - FF25 58324000  jmp     dword ptr [0x403258]   GDI32.TextOutA
00401530 - FF25 5C324000  jmp     dword ptr [0x40325C]   GDI32.StartPage
00401536 - FF25 60324000  jmp     dword ptr [0x403260]   GDI32.StartDocA
0040153C - FF25 64324000  jmp     dword ptr [0x403264]   GDI32.GetTextMetricsA
00401542 - FF25 68324000  jmp     dword ptr [0x403268]   GDI32.GetStockObject
00401548 - FF25 6C324000  jmp     dword ptr [0x40326C]   GDI32.EndPage
0040154E - FF25 70324000  jmp     dword ptr [0x403270]   GDI32.EndDoc
00401554 - FF25 74324000  jmp     dword ptr [0x403274]   GDI32.DeleteObject
0040155A - FF25 78324000  jmp     dword ptr [0x403278]   GDI32.DeleteDC
00401560 - FF25 80324000  jmp     dword ptr [0x403280]   COMDLG32.GetSaveFileNameA
00401566 - FF25 84324000  jmp     dword ptr [0x403284]   COMDLG32.GetOpenFileNameA
0040156C - FF25 88324000  jmp     dword ptr [0x403288]   COMDLG32.PrintDlgA
```

Tại đây, ta thấy một jump table nhận các giá trị của IAT, các lệnh nhảy này sẽ đưa ta tới từng hàm API. Để ý một chút ta thấy những lệnh nhảy đều bắt đầu bằng opcodes là **FF 25**, vì vậy khi các bạn đọc các bài viết về unpack sẽ thấy các tác giả thực hiện tìm kiếm Binary với nội dung tìm kiếm là **FF 25**, kết quả sẽ tới vùng jump này nhanh hơn:

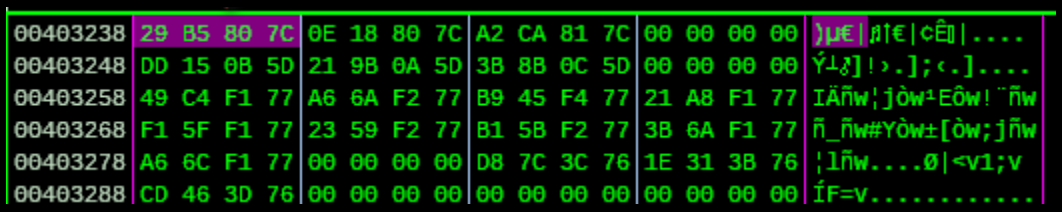


Vấn đề ở chỗ, nhiều lúc không phải chương trình nào cũng dùng các lệnh nhảy gián tiếp để tới các hàm API, vì thế đôi khi phương pháp này bị thất bại, nhưng tốt nhất và để không bao giờ thất bại là phải tìm được một lệnh CALL đến một hàm API, và quan sát nơi nó nhận giá trị được lưu trữ để đưa ta đến API, và giá trị đó phải được lưu trữ trong IAT.

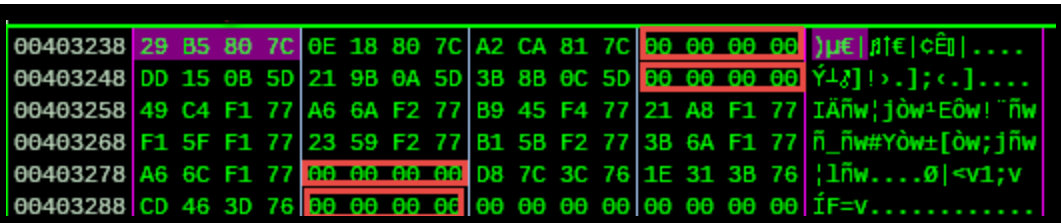
Quay trở lại vấn đề, ta đang đứng tại lệnh nhảy **jmp dword ptr [0x403238]**. Tại cửa sổ DUMP, ta thấy rõ ràng tại **0x403238** là một phần của bảng IAT, bởi đây là nơi lưu địa chỉ của hàm API **GetModuleHandleA**, những gì ta cần bây giờ là tìm kiếm địa chỉ bắt đầu và kết thúc của bảng IAT.



Các thô sơ nhất là xem tất cả các lệnh nhảy gián tiếp JMPs và ghi nhớ địa chỉ nhỏ nhất và địa chỉ lớn nhất, nhưng cách này thường rất chậm, đặc biệt với những chương trình sử dụng nhiều APIs là oải luôn  . Cách tốt nhất là chuyển tới cửa số DUMP và cuộn chuột lên xuống để quan sát, và như chúng ta biết mỗi địa chỉ này sẽ chứa một địa chỉ của một hàm API, ví dụ trong trường hợp này là **7C80B529**, lưu theo kiểu little endian là **29 B5 80 7C**, giá trị thay đổi tại hai cột **Address** và **Hex Dump** sẽ cho ta cái nhìn rõ ràng hơn. Tuy nhiên, còn có một cách nhanh hơn nữa, tôi sẽ chỉ cho các bạn sau  .



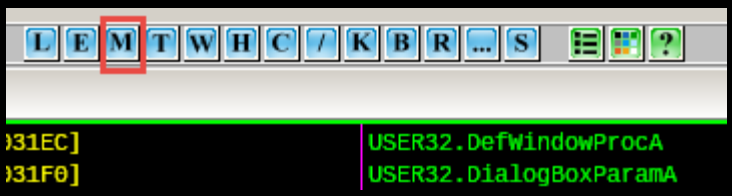
Tại cửa số Dump ta sẽ quan sát được việc tổ chức của IAT, ta đã thấy từ địa chỉ **0x00403238** đang chứa các giá trị địa chỉ API của cùng một DLL (**7Cxxxxxx**) và sau đó được phân tách bằng 4 bytes **00** để bắt đầu với các địa chỉ APIs của các DLL tiếp theo (**5Dxxxxxx**, **77xxxxxx**, **76xxxxxx** ...). Tôi đánh dấu các bytes **00** này lại để dễ quan sát:



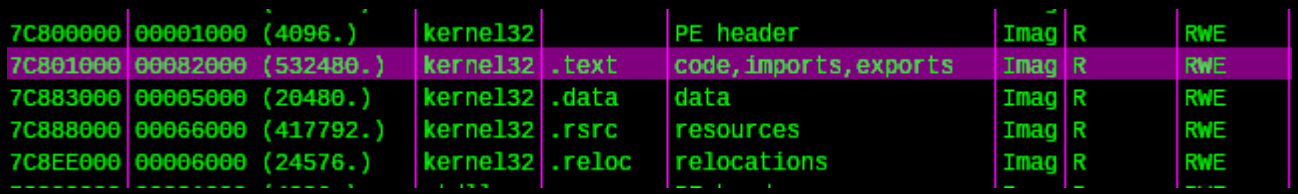
Tuy nhiên, có một số trình packer cao cấp hơn sẽ ghi đè lên các bytes **00** bằng các giá trị rác làm cho việc việc xây dựng lại bảng IAT trở nên khó khăn, mặc dù về bản chất toàn bộ các bytes **00** này không được sử dụng, cũng như không cần thiết trong quá trình thực thi chương trình.

```
00403238 29 B5 80 7C 0E 18 80 7C A2 CA 81 7C 00 00 00 00 )µ€|€|¢Ê□|....
```

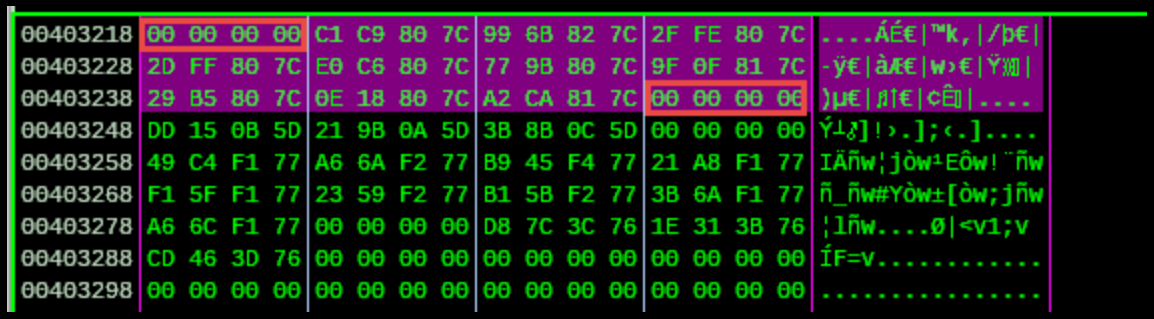
Như đã thấy có 03 địa chỉ APIs bắt đầu bằng **7Cxxxxxx** và kết thúc bằng các bytes **00** để phân tách với địa chỉ của DLL tiếp theo. Nhấn **M** để mở cửa sổ **Memory Map**, sau đó tìm kiếm vùng địa chỉ của DLL tương ứng với **7Cxxxxxx**:



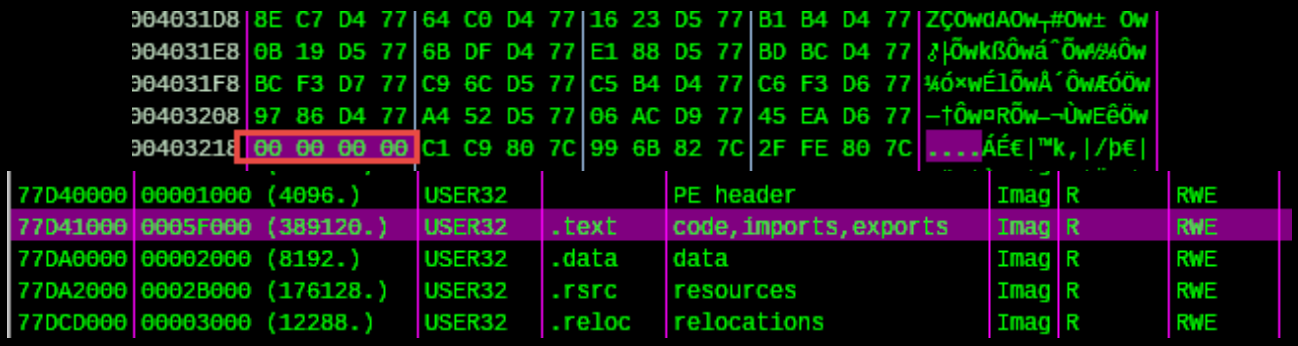
Kết quả, ta thấy rằng tất cả những địa chỉ trên đều nằm trong trong phần code của **kernel32.dll**:



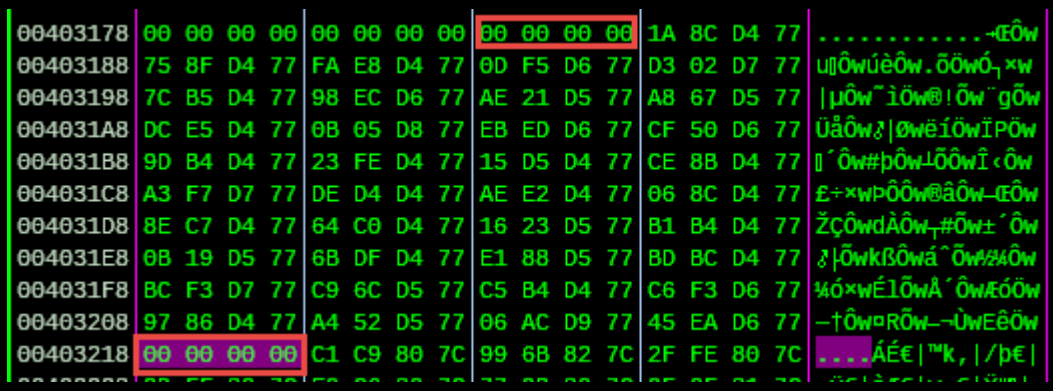
Tại cửa sổ **Memory Map**, các DLLs trong máy của các bạn có thể được bố trí ở vị trí khác với máy tôi, nhưng tóm lại máy tôi hay của bạn thì những địa chỉ này đều tương ứng thuộc vào CODE Section của **kernel32.dll**, vậy từ điều này ta suy luận các địa chỉ khác cũng sẽ nằm trong các DLL tương ứng. Cuối cùng, chúng ta có được thông tin toàn bộ các địa chỉ API thuộc về **kernel32.dll** như sau:



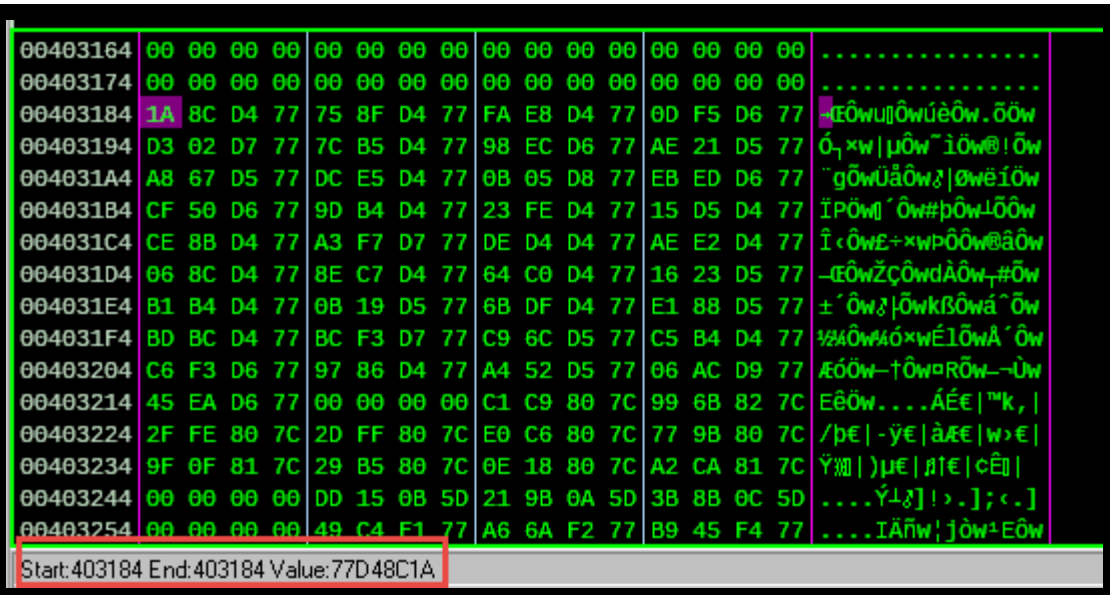
Cũng tại cửa sổ DUMP, cuộn chuột lên một chút ta thấy các bytes phân tách `00` được đánh dấu màu đỏ và ở trên có các địa chỉ thuộc về một DLL khác, các địa chỉ này bắt đầu bằng `77Dxxxxx`. Tìm kiếm tại màn hình **Memory Map**, chúng ta sẽ tìm thấy được DLL tương ứng:



Như vậy, các địa chỉ trên tương ứng với CODE section của `user32.dll`, tiếp tục thực hiện tìm kiếm tiếp lên trên cho tới khi thấy các bytes `00`:



Như vậy, ta thấy tất cả các địa chỉ hàm APIs trên đều thuộc `user32.dll` và bên trên toàn các bytes `00`, do đó có thể khẳng định rằng địa chỉ bắt đầu của bảng IAT là `0x403184`:

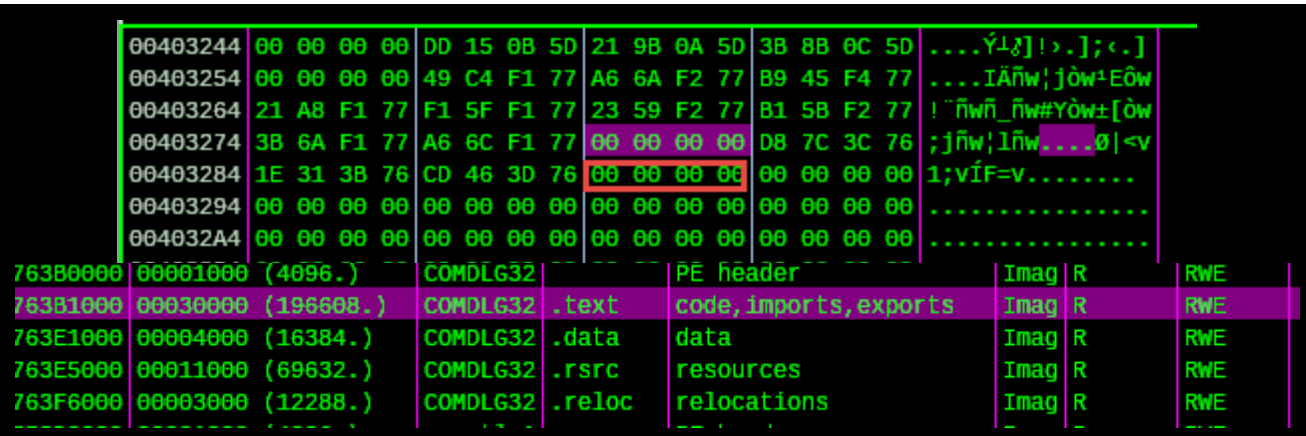


Rõ ràng bên trên không có địa chỉ nào thuộc DLL khác và trong ví dụ này ta thấy toàn các bytes **00**, đó là lý do giúp ta suy luận được địa chỉ bắt đầu của IAT. Tuy nhiên, một số trình packer cao cấp hơn thường chèn đầy bytes rác trước và sau IAT, vì vậy dẫn đến khó khăn trong việc làm sao để xác định địa chỉ bắt đầu của IAT, nhưng nếu ta biết rằng các giá trị của bảng IAT luôn luôn dẫn tới phần code của một dll, thì ta sẽ nhận ra những gì trị nào là rác, bởi địa chỉ rác sẽ không dẫn đến bất cứ phần code của bất kỳ DLL nào.

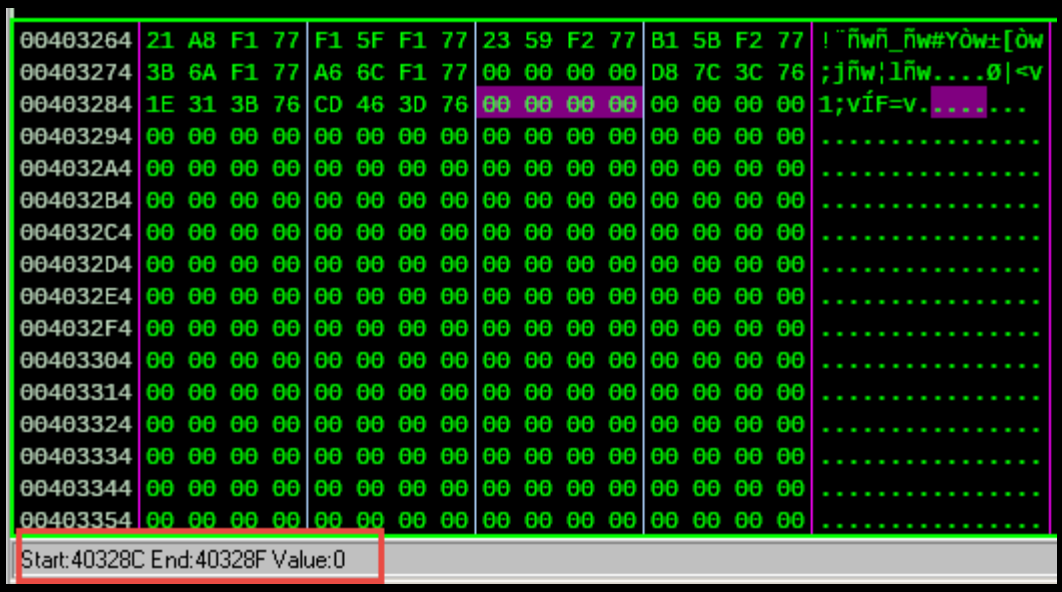
Trong hình trên, ta đã có được thông tin địa chỉ bắt đầu của IAT là **0x403184**, tiếp theo ta sẽ lần xuống dưới để tìm địa chỉ kết thúc của bảng IAT bằng cách áp dụng cùng một phương pháp như đã mô tả.

Tuy nhiên, sau này ta sẽ gặp các trình packers có khả năng thay đổi các giá trị trong IAT và chuyển hướng tới các đoạn code riêng, để từ các đoạn code này sẽ nhảy tới các APIs. Đương nhiên, với bài viết này thì trường hợp đó chưa xảy ra, ta sẽ nghiên cứu nó trong các bài viết tới đây (hi vọng thế ). Nhưng bây giờ, những gì chúng ta quan sát tại các giá trị của bảng IAT là địa chỉ các hàm APIs, và các địa chỉ này sẽ phải thuộc vào đoạn code của một DLL nào đó mà chương trình import để sử dụng.

Ok, sau một vài cú lăn chuột và quan sát, ta đã thấy được giá trị cuối cùng của bảng IAT bắt đầu bằng địa chỉ **76xxxxxx**, quan sát tại màn hình **Memory Map** ta thấy vùng code của DLL tương ứng với các địa chỉ này:



Trong trường hợp này sẽ tương ứng với vùng code của **COMDLG32.dll**, các giá trị tiếp sau đó lại là các bytes **00**, như vậy **0x40328C** chính là địa chỉ kết thúc của bảng IAT:



Tổng kết lại ta có được thông tin về địa chỉ bắt đầu và kết thúc của bảng IAT như sau:

IAT Start: 0x403184

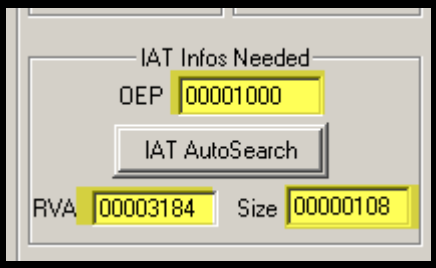
IAT End: 0x40328C

Như quan sát phần **"IAT Infos Needed"** tại màn hình của ImpREC, ta thấy ImpREC yêu cầu cung cấp ba thông tin:

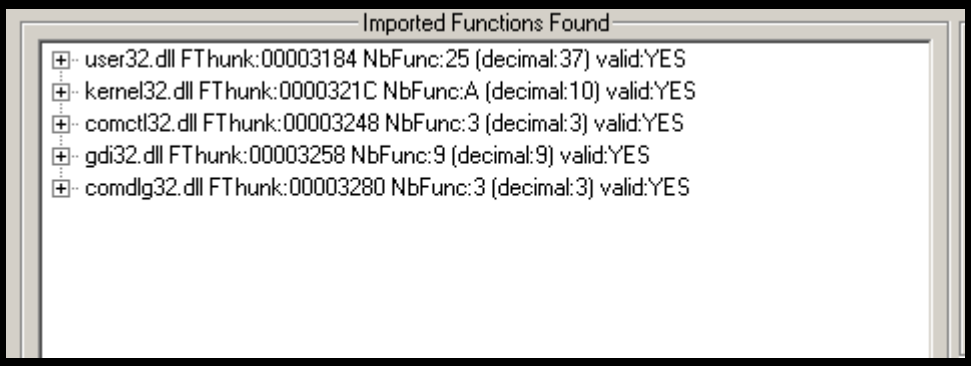
1. Thông tin thứ nhất là địa chỉ OEP, địa chỉ OEP ta đã tìm được ở trên, lấy giá trị này trừ đi địa chỉ **ImageBase**: **401000-400000 = 1000.**
2. Thông tin thứ hai là địa chỉ bắt đầu của IAT, giá trị cần cung cấp là một RVA, do vậy chúng ta phải trừ đi địa chỉ **ImageBase** của chương trình, ở đây là **400000**. Như vậy, ta có **403184 – 400000 = 3184**.

3. Thông tin cuối cùng là kích thước của bảng IAT, rất đơn giản ta chỉ việc lấy địa chỉ kết thúc của IAT trừ đi địa chỉ bắt đầu của IAT là ra được kích thước. Kết quả có được như sau: **Size = 40328C-403184 = 108.**

Sau khi có được 3 thông tin này, điền các giá trị thu được vào phần "**IAT Infos needed"** tại màn hình của ImpREC:

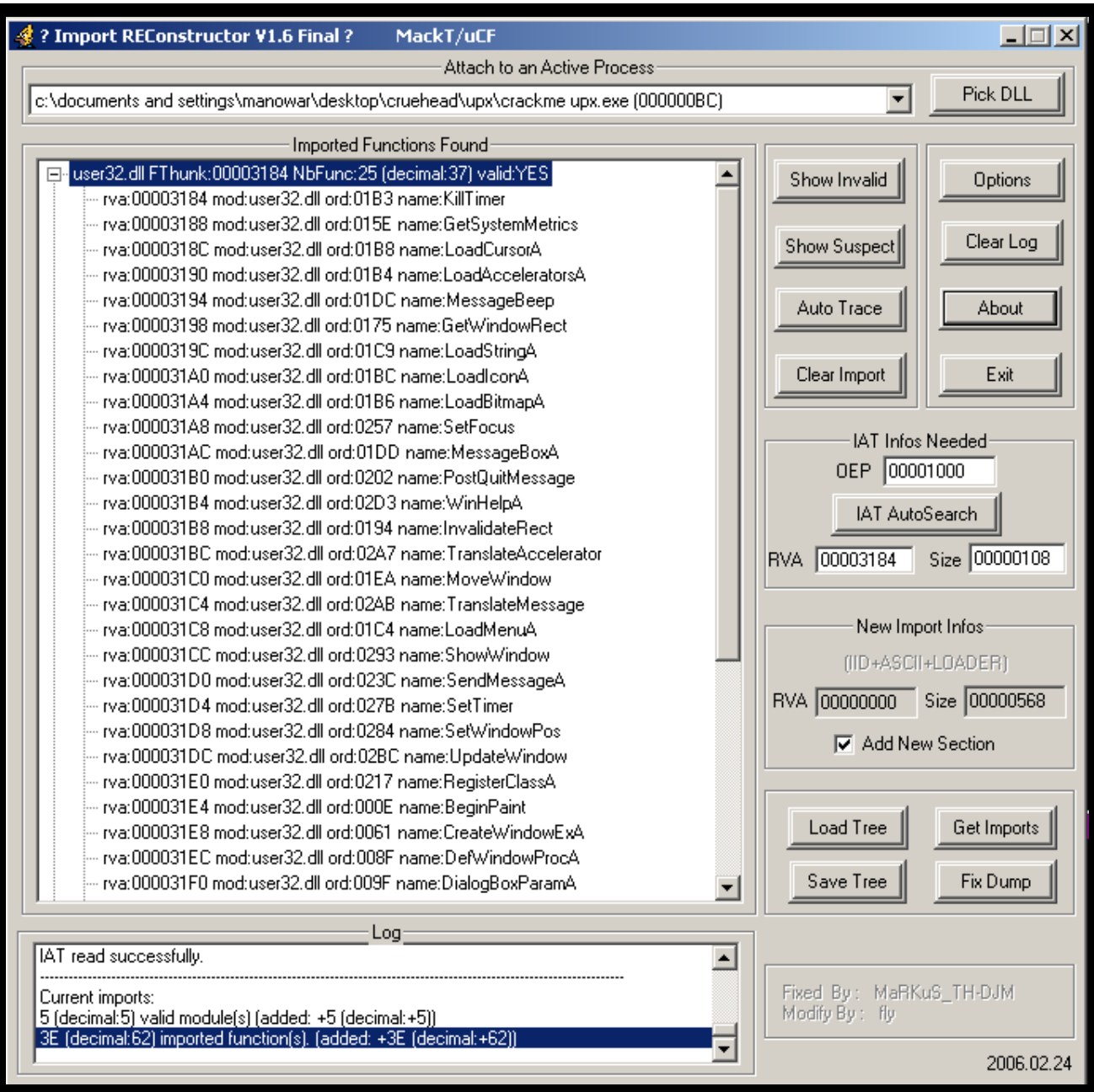


Sau khi điền xong nhấn **Get Imports**, kết quả có được như sau:

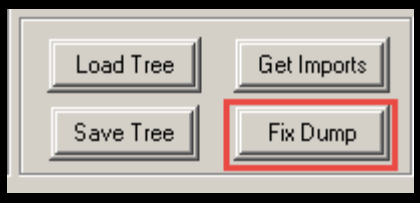


Như vậy, ImpREC thực hiện tìm kiếm các hàm APIs thuộc từng DLL trong IAT, số lượng hàm tìm thấy được mô tả tại thông số **NbFunc**, tính hợp lệ của kết quả tìm được sẽ được ghi rõ là **YES**, trong trường hợp một giá trị không tới trực tiếp một API thì ImpREC sẽ hiển thị thông tin là **NO** và trong trường hợp này, ta sẽ phải tìm ra giá trị đó có thực sự thuộc về một hàm API nào đó không, sau đó cấu hình lại để ImpREC nhận diện chính xác, sau đó chúng ta có thể tiến hành bước tiếp theo là sửa chữa lại file đã được dumped.

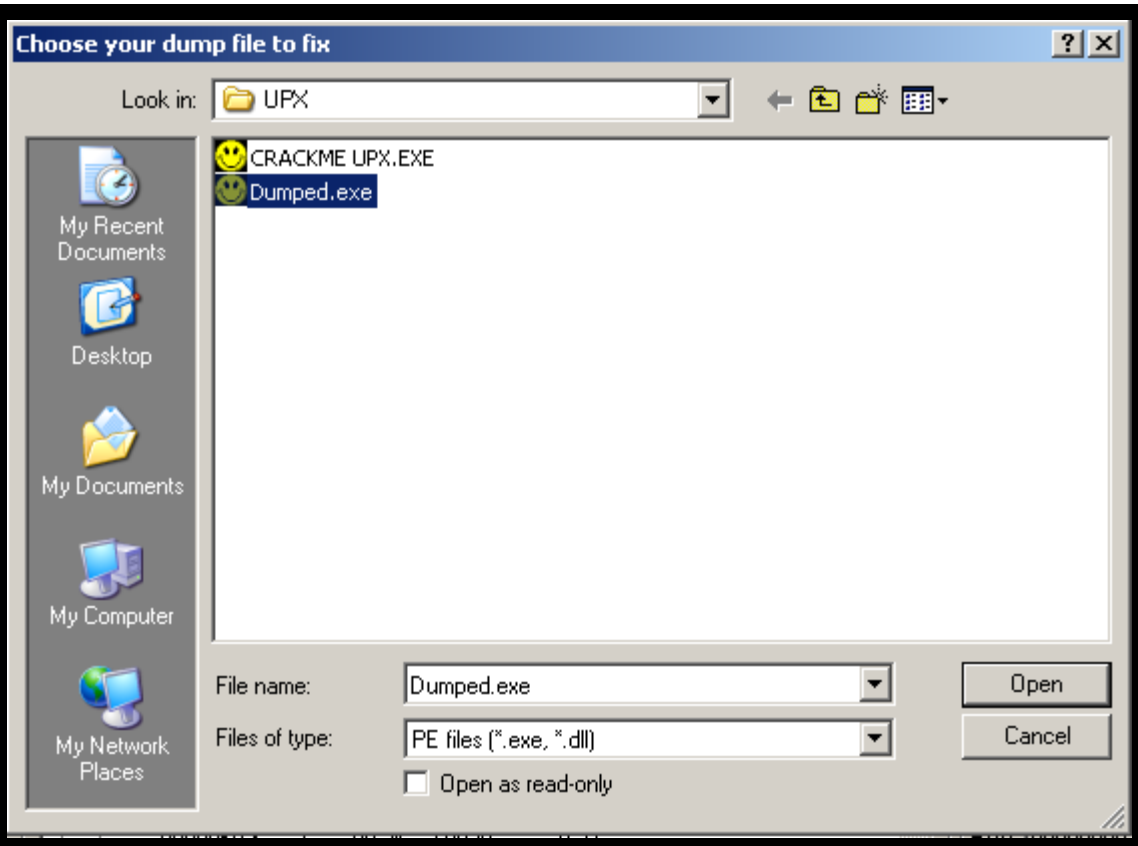
Để xem thông tin về các hàm APIs mà ImpREC tìm được của từng DLL, ta nhấn vào dấu **"+"** tại từng DLLs để xổ ra danh sách các hàm:



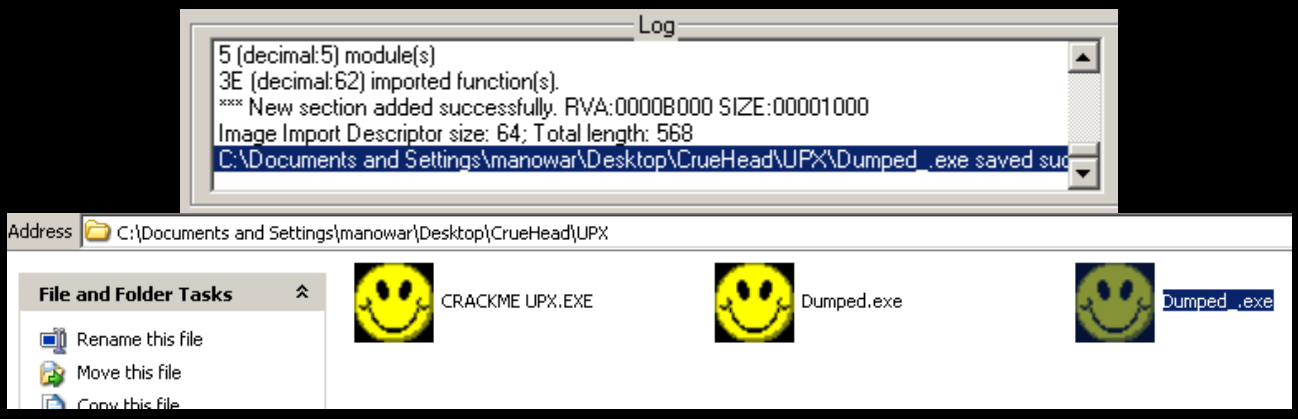
Theo kết quả hiện thị trên màn hình của ImpREC, ta thấy toàn bộ thông tin mà ImpREC tìm được hoàn toàn hợp lệ và chính xác, để sửa chữa file đã dump ta nhấn vào nút **Fix Dump**.



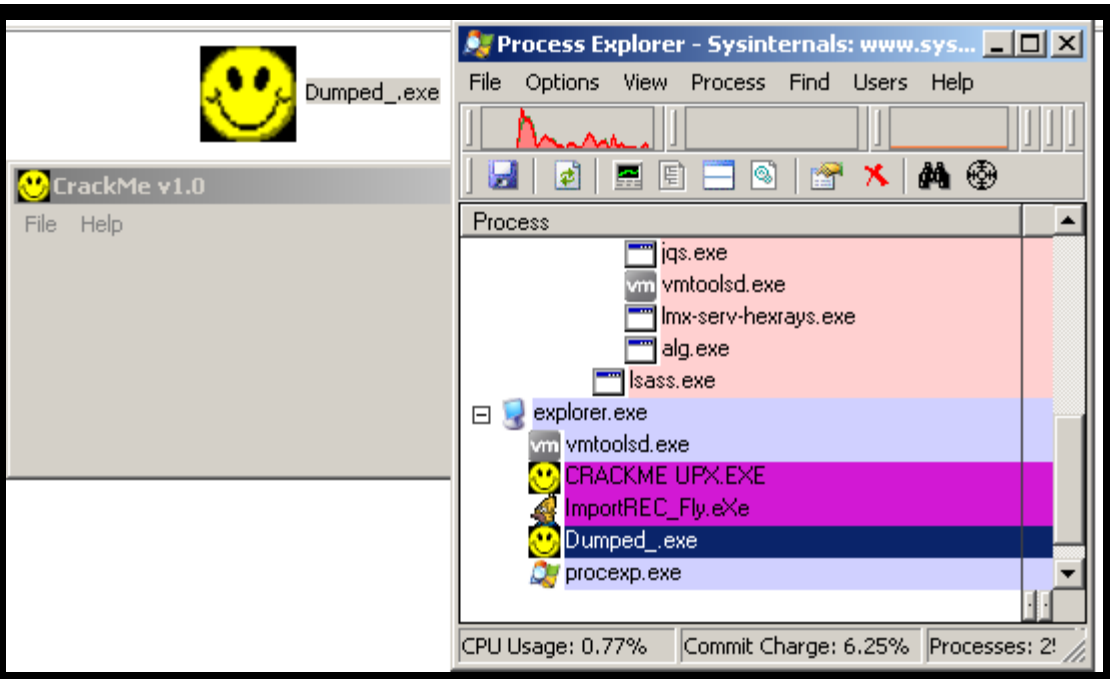
Tìm đến đường dẫn lưu file đã dumped và lựa chọn file này, sau đó nhấn **Open**:



ImpREC sẽ tiến hành sửa chưa lại file Dumped nhưng không lưu trực tiếp thông tin sửa chữa vào file này mà sẽ tạo một file mới với tên là **DUMPED_.exe** và lưu luôn tại cùng thư mục với file dumped.

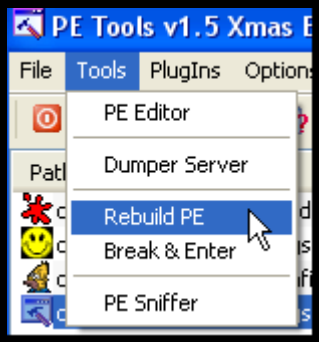


Tiến hành chạy thử file đã sửa chữa xem kết quả thế nào:

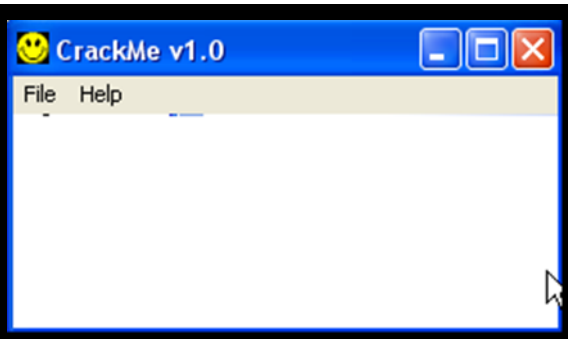


Trên máy tôi, file đã sửa chạy bình thường, tuy nhiên trong bài viết của lão Ricardo thì file này chạy bị lỗi như sau:

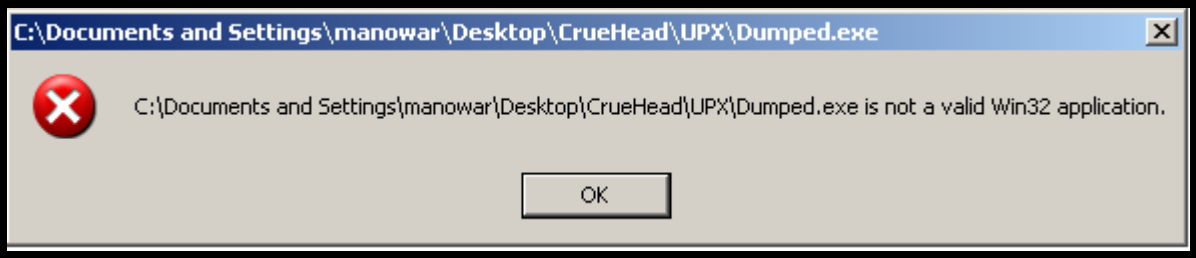
Lão Ricardo diễn giải như sau *"Lolz, dường như vẫn có vẻ bị thiếu sót một cái gì đó, nhưng đừng quá lo lắng vì điều này thường xảy ra sau khi ta sửa IAT, giải pháp là sử dụng PE Tools để Rebuild lại file đã sửa bởi ImpREC"*. Nếu bạn gặp lỗi tương tự như trên máy của lão Ricardo, mở công cụ PE Tools lên và chọn Rebuild PE:



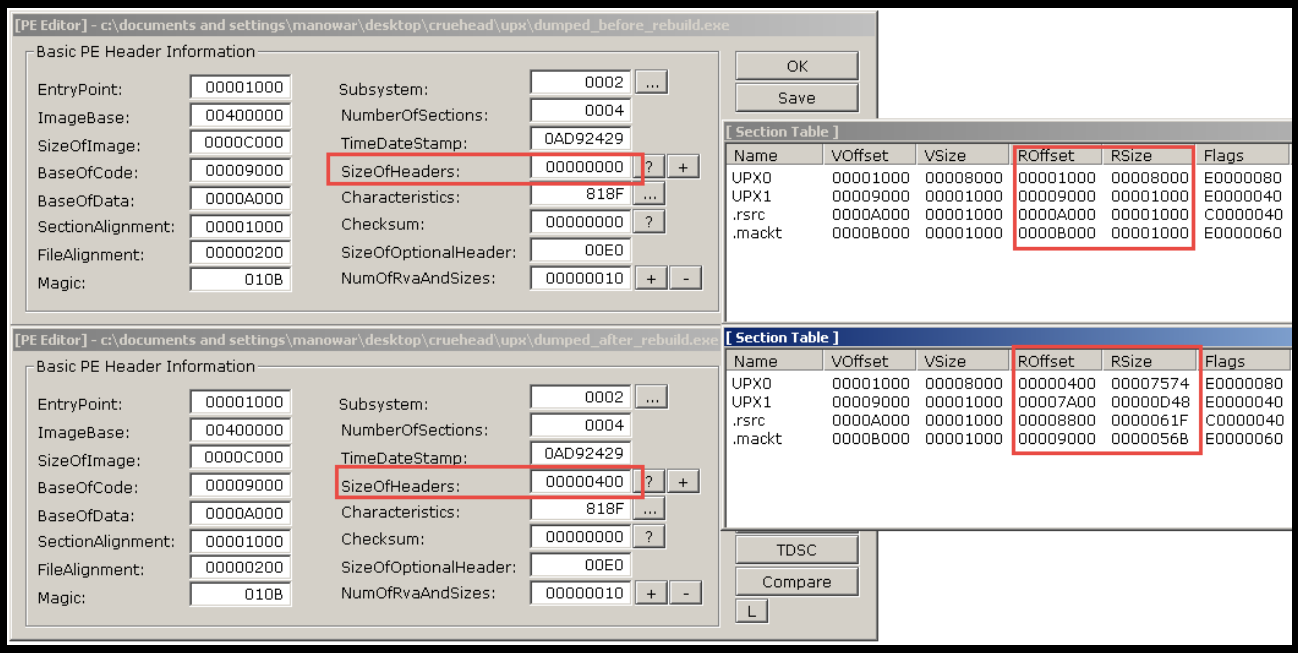
Tìm và chọn file **Dumped_.exe**, nhấn Open để PE Tools fix lại. Sau đó chạy lại **Dumped_.exe** sẽ thấy nó thực thi một cách hoàn hảo:



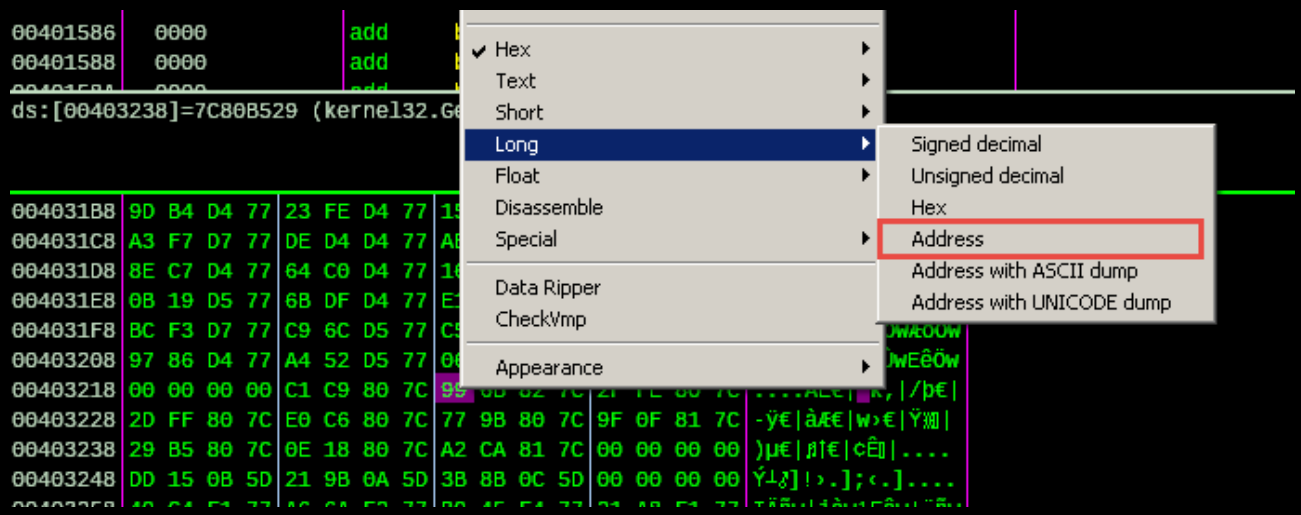
Note 1: Sau một hồi xem lại, tôi đã hiểu lý do vì sao file đã fix IAT của tôi lại chạy bình thường không cần Rebuild mà trong khi đó lão Ricardo phải Rebuild thì file mới chạy được. Đó là do tôi và lão dùng 2 phiên bản PE Tools khác nhau (phiên bản của tôi mới hơn và được fix lỗi). Do lão dùng bản cũ tải tại đường dẫn http://www.uinc.ru/files/neox/PE_Tools.shtml nên khi thực hiện Dump full, thì file dumped.exe đó sẽ bị lỗi và nếu chạy file này sẽ nhận được thông báo sau thay vì thông báo lỗi như file của tôi được dump bằng bản mới.



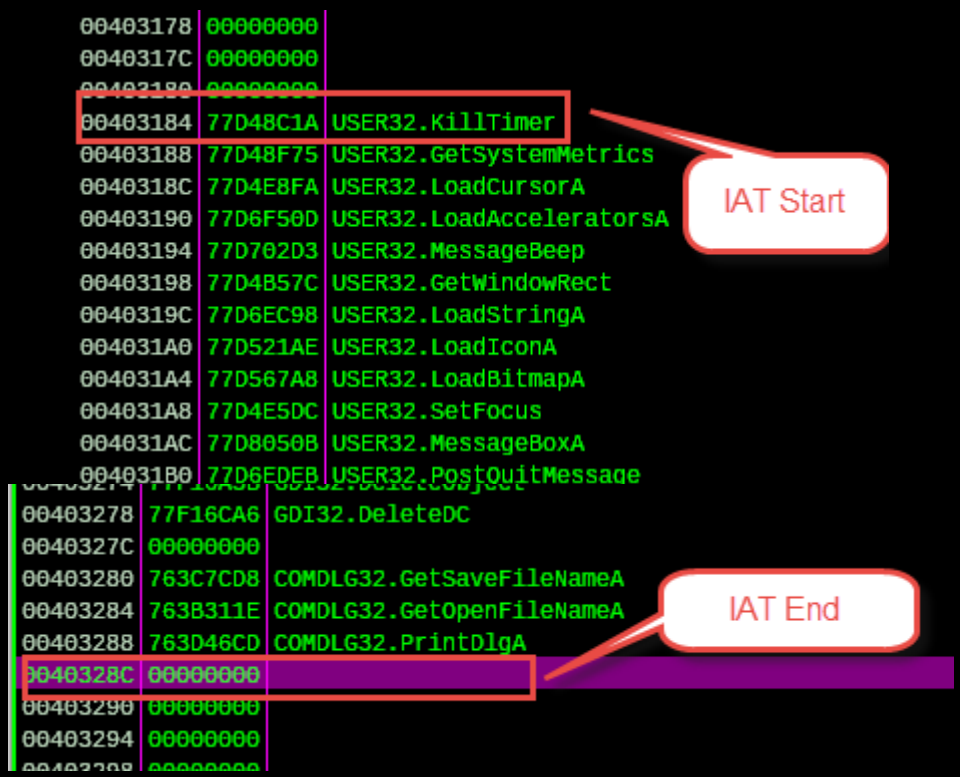
Do vậy, lúc fix lại IAT bằng ImportREC thì file sau khi fix cũng vẫn sẽ bị lỗi, nên vì thế không chạy luôn được, cần phải dùng PE Tools để Rebuild file. Tôi có compare bằng "cơm" một số chỗ khác nhau trước và sau khi rebuild file với bản PE Tools cũ:



Note 2: Trong bài viết, tôi có đề cập đến cách có thể tìm được IAT Start và IAT End nhanh hơn so với những gì tôi đã mô tả. Thực chất, việc tìm kiếm này không có gì cao siêu, chẳng qua là cách sắp sếp, bố trí lại các địa chỉ trong cửa số Dump. Tại cửa số Dump, sau khi ta Follow lời gọi hàm API và follow vùng nhớ lưu địa chỉ của các hàm APIs, ta nhấn chuột phải và chọn như sau:

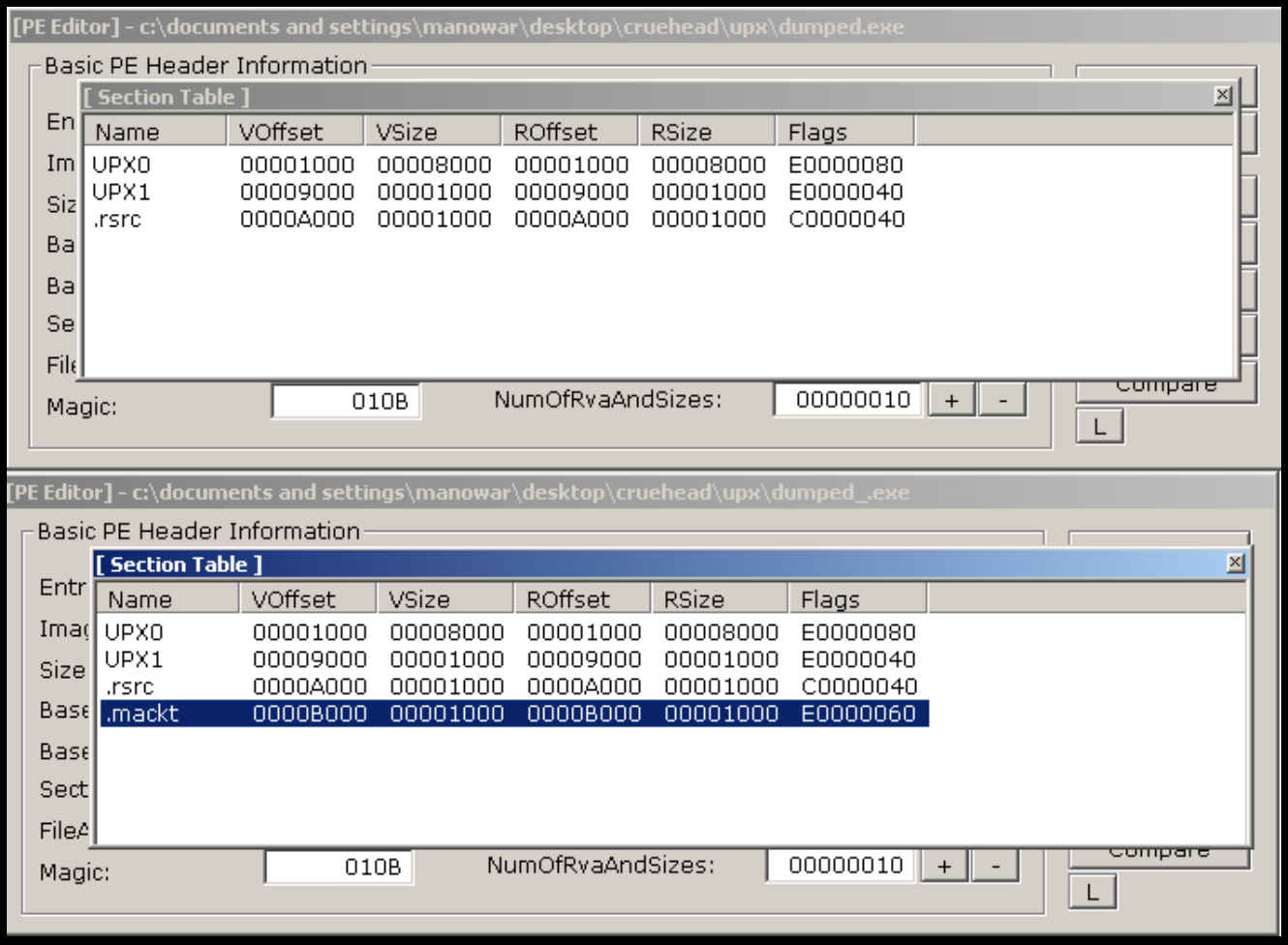


Với tết quả có được, cuộn chuột lên và xuống sẽ giúp chúng ta sẽ dàng xác định và tìm ra được IAT Start, IAT End:

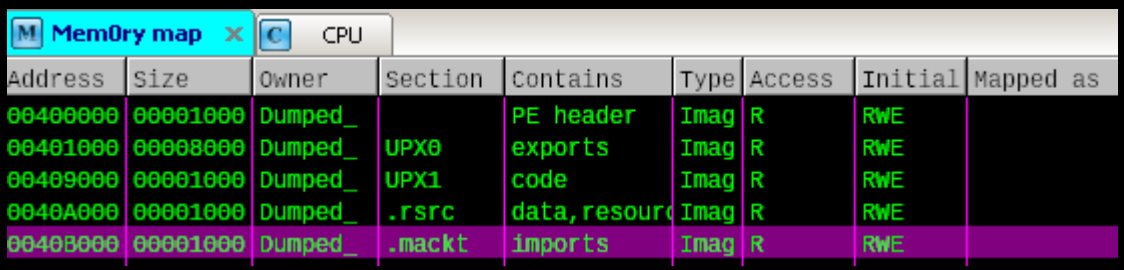


*Note 3: Để ý file được dump full (**Dumped.exe**) và file sau khi được fix IAT bởi ImpREC (**Dumped_.exe**), so sánh hai file này bằng LordPE, ta sẽ thấy ImpREC đã thêm*

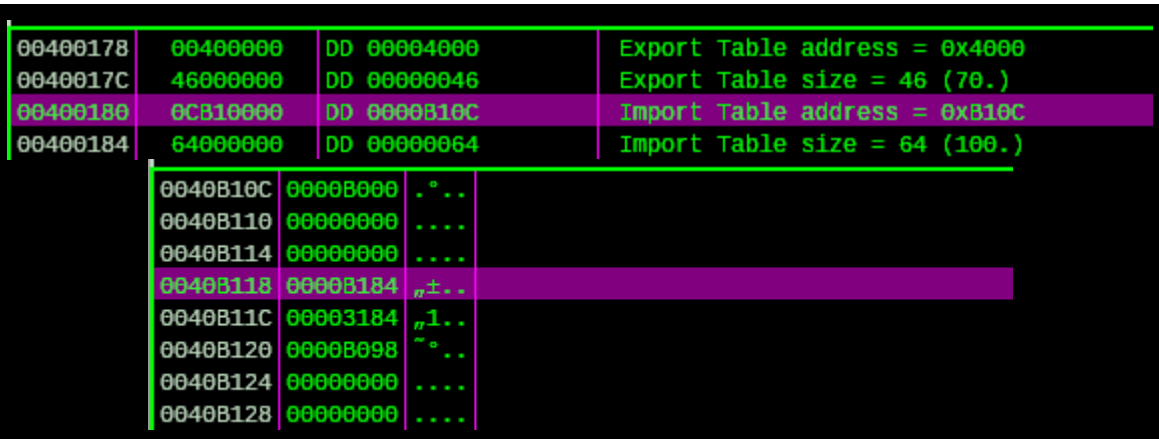
một section mới với tên là ***.mackt****, và section này là nơi lưu thông tin về Imports sau khi fix.*



Load file Dumped_.exe vào OllyDbg, mở cửa sổ Memory Map sẽ thấy thông tin mô tả về ***.mackt*** *section:*

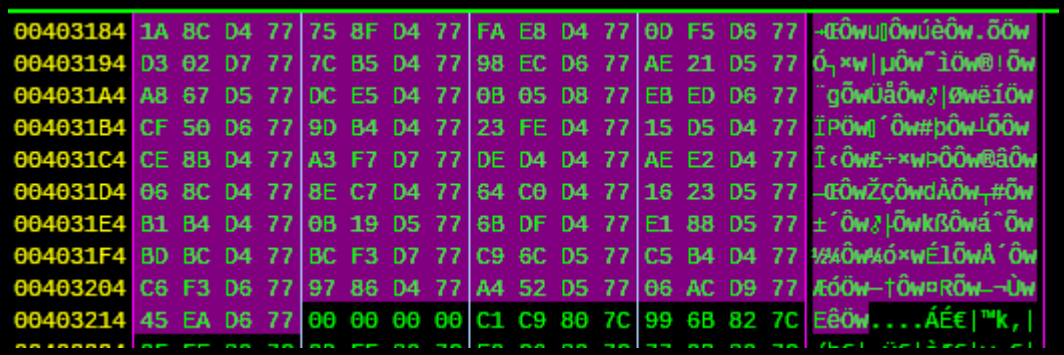


Như vậy, ImpREC sau khi thêm section .mackt để chứa toàn bộ Imports thì sẽ sửa lại thông tin về Import Table address. Quan sát tại OllyDbg thì RVA của IT là ***0xB10C****, tức là Import Directory sẽ bắt đầu từ* ***0x0040B10C****:*



Như đã biết, tại DWORD thứ 4 là một RVA trỏ tới tên của DLL. Chuyển tới địa chỉ ***0x0040B184****:*

Đó chính là ***user32.dll****, tại DWORD thứ 5 là một RVA trỏ tới một mảng của cấu trúc IMAGE_THUNK_DATAs. Chuyển tới địa chỉ* ***0x00403184*** *(đây cũng chính là địa chỉ IAT Start mà ta đã tìm được ở phần trên):*



00403184	77D48C1A	user32.KillTimer
00403188	77D48F75	user32.GetSystemMetrics
0040318C	77D4E8FA	user32.LoadCursorA
00403190	77D6F50D	user32.LoadAcceleratorsA
00403194	77D702D3	user32.MessageBeep
00403198	77D4B57C	user32.GetWindowRect
0040319C	77D6EC98	user32.LoadStringA
004031A0	77D521AE	user32.LoadIconA
004031A4	77D567A8	user32.LoadBitmapA
004031A8	77D4E5DC	user32.SetFocus
004031AC	77D8050B	user32.MessageBoxA
004031B0	77D6EDE8	user32.PostQuitMessage
004031B4	77D650CF	user32.WinHelpA
004031B8	77D4B49D	user32.InvalidateRect
004031BC	77D4FE23	user32.TranslateAcceleratorA
004031C0	77D4D515	user32.MoveWindow
004031C4	77D48BCE	user32.TranslateMessage
004031C8	77D7F7A3	user32.LoadMenuA
004031CC	77D4D4DE	user32.ShowWindow
004031D0	77D4E2AE	user32.SendMessageA
004031D4	77D48C06	user32.SetTimer
004031D8	77D4C78E	user32.SetWindowPos
004031DC	77D4C064	user32.UpdateWindow
004031E0	77D52316	user32.RegisterClassA
004031E4	77D4B4B1	user32.BeginPaint
004031E8	77D5190B	user32.CreateWindowExA
004031EC	77D4DF6B	user32.DefWindowProcA
004031F0	77D588E1	user32.DialogBoxParamA
004031F4	77D4BCBD	user32.DispatchMessageA
004031F8	77D7F3BC	user32.DrawMenuBar
004031FC	77D56CC9	user32.EndDialog
00403200	77D4B4C5	user32.EndPaint
00403204	77D6F3C6	user32.FindWindowA
00403208	77D48697	user32.GetDC
0040320C	77D552A4	user32.GetDlgItem
00403210	77D9AC06	user32.GetDlgItemTextA
00403214	77D6EA45	user32.GetMessageA

Qua đây, chúng ta thấy rằng công cụ ImpREC đã thực hiện một công việc rất tuyệt vời, phát hiện tất cả các APIs, xây dựng lại IT, sắp xếp tất cả các con trỏ, và xây dựng lại danh sách tên của từng hàm API để đảm bảo khi chương trình sau khi fix IAT thực thi bình thường trên mọi OS. Tuy nhiên, ImpREC cũng được phát triển từ lâu nên trong một số trường hợp nó cũng bị lỗi, hiện nay có một số công cụ mới hơn/port lại từ ImpREC như CHimpREC, Scylla - x64/x86 Imports Reconstruction, Universal Import Fixer.

III. Kết luận

Toàn bộ phần 28 đến đây là kết thúc, cảm ơn các bạn đã dành thời gian để theo dõi.

 .

Qua toàn bộ bài viết này, lần đầu tiên tôi và các bạn đã cùng nhau thực hiện xây dựng lại một IAT thuần túy và đơn giản nhất. Tuy đơn giản, nhưng nó là cơ sở, tiền đề cho tất cả mọi thứ sau này, bởi vì chúng ta tiếp tục tìm hiểu quá trình Manual Unpacking với các trường hợp như bảng IATs bị phá hủy, APIs bị redirect và các trường hợp khó khăn khác. Ngoài ra, như các bạn thấy UPX là một trình packer đơn giản, nó không áp dụng các cách thức bảo vệ như anti-dump, nhưng hiện nay, các trình packer đã cao cấp hơn nhiều, hi vọng trong các bài tới tôi sẽ có dịp viết về cơ chế này. Hi vọng, tôi vẫn còn đủ hứng thú để viết tiếp!! .

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:
kienbigmummy[at]gmail.com