**ĐẠI HỌC QUỐC GIA HÀ NỘI**

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

BÁO CÁO THỰC TẬP

**DỊCH NGƯỢC và PHÂN TÍCH MÃ ĐỘC**

Sinh viên : Đinh Văn Kiệt

MSSV : 14020247

Lớp : K59H

SĐT : 0974659271

Công ty thực tập : Trung tâm an ninh mạng Viettel

Giáo viên hướng dẫn : TS. Ngô Thị Duyên

**MÔ TẢ**

Khóa thực tập ở Trung Tâm An Ninh Mạng Viettel (TTANM), em được đào tạo về 2 mảng là Mã độc và Machine Learning

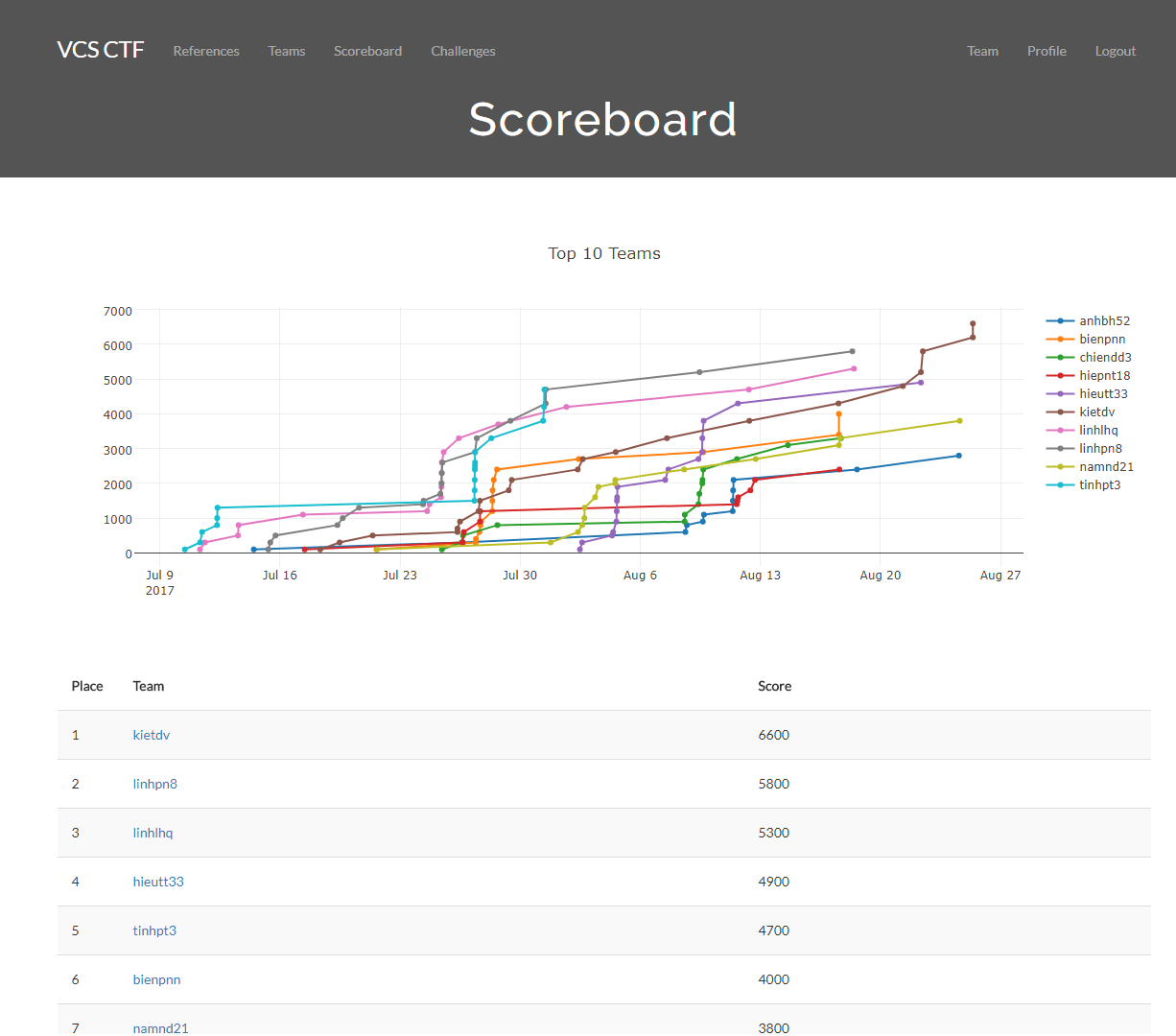
Giai đoạn đâu là học về Mã độc: Assembly, Win32, Pack & Unpack, Malware.

Giai đoạn sau là học về áp dụng Machine Learning vào phát hiện bất thường.

Các bài tập dưới đây thuộc phần Dịch ngược và Mã độc, nằm trong tổ hợp các bài Capture The Flag của TTANM (có chấm điểm)

**BẢNG ĐIỂM**

(cập nhật 25/8/2017):



**MỤC LỤC**

1. Lý thuyết
   1. PE file
   2. Pack và Unpack
   3. Malware
2. Bài tập
   1. Bài toán 1 – UPX [Dễ]
   2. Bài toán 2 - Aosuke [Dễ]
   3. Bài toán 3 – Synchronous [Bình thường]
   4. Bài toán 4 – ASPack, MPRESS, Petite [Dễ]
   5. Bài toán 5 – WinCrypt [Khó]
   6. Bài toán 6 – HiddenTreasure [Bình thường]
   7. Bài toán 7 – Findme [Bình thường]
   8. Bài toán 8 – Crackme [Khó]
   9. Bài toán 9 – The Numbers [Dễ]
   10. Bài toán 10 – Domain Generation Algorithm [Bình thường]
   11. Bài toán 11 – Ransomware [Bình thường]

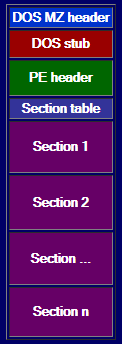
**TIMELINE**

|  |  |  |  |
| --- | --- | --- | --- |
| STT | Ngày | Nội dung | Ghi chú |
| 1 | 3/8/2017 | Nộp báo cáo lần 1 | I.a, I.b, II.a, II.b |
| 2 | 16/8/2017 | Nộp báo cáo lần 2 | II.c, II.d, II.e |
| 3 | 26/8/2017 | Nộp báo cáo lần 3 | II.f, II.g, II. h |
|  |  |  |  |

**I. LÝ THUYẾT**

**A. PE FILE**

PE là viết tắt của cụm từ Protable Executable. Nó là định dạng file thuần túy của Win32. Ý nghĩa của cụm từ “Protable Executable” là định dạng file tiêu chuẩn, rộng rãi trên khắp các nền tảng Win32: PE Loader của mỗi nền tảng Win32 nhận dạng và sử dụng định dạng file này khi hệ điều hành Windows chạy trên các nền tảng chip khác Intel. Điều đó không có nghĩa là file PE của bạn không cần phải điều chỉnh gì để chạy trên các nền tảng khác. Tất cả các file thực thi của win32 (trừ VxDs và 16-bit Dlls) sử dụng định dạng fie PE. Kể cả các nhân driver cũng sử dụng định dạng PE file.



Trên đây là khung cả một PE file. Tất cả các file PE đều phải bắt đầu với trường DOS MZ header. Trong trường hợp file PE được khởi chạy trên môi trường DOS, DOS có thể nhận diện được file này có đúng là file thực thi không nhờ giá trị lưu trong trường DOS MZ header. DOS stub chứa một chuỗi để hiển thị thông báo “This program requires Windows”.

Phía dưới DOS stub là PE header. Chứa struct IMAGE\_NT\_HEADERS. Struct này chứa các trường dữ liệu được sử dụng bởi PE loader. Khi chương trình khởi chạy và hệ điều hành có thể hiểu được PE file thì PE loader sẽ tìm đến offset bắt đầu của PE header, phía sau DOS MZ header.

Nội dung chính của PE file được chia vào trong các block tên là sections. Mỗi section chứa các thuộc tính và thông tin của dữ liệu. Ví như ta coi định dạng PE file như là một ổ đĩa, thì PE header chính là phần boot và các sections chính là các file. Ngay phái sau PE header chính là bảng sections chứa một mảng struct. Mỗi struct bao gồm thông tin về từng sections đó của PE file như thuộc tính, file offset, va,…

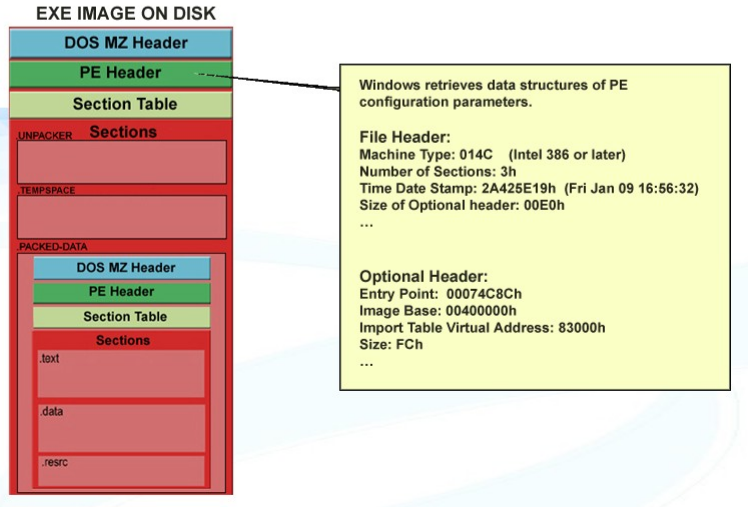
**B. LÝ THUYÊT VỀ PACK VÀ UNPACK**

PE-Packer là gì?

Là 1 file thực thi nằm bên trong một file thực thi khác. Và cũng có thể nằm trong 1 file thực thi khác nữa.

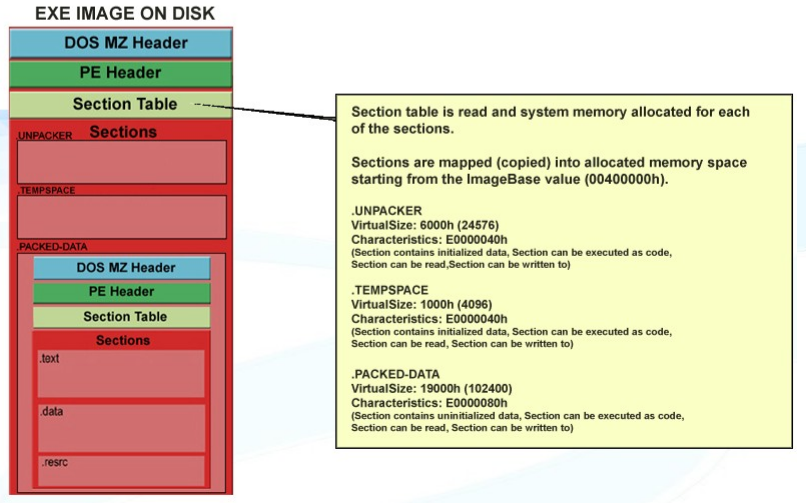
File thực thi (exe) bị “packed” bên trong một file thực thi khác

Khi được thực thi, file exe bao bên ngoài sẽ tiến hành giải nén nội dung của file thực thi nằm bên trong. Sau đó chương trình mới thực sự được thực thi.

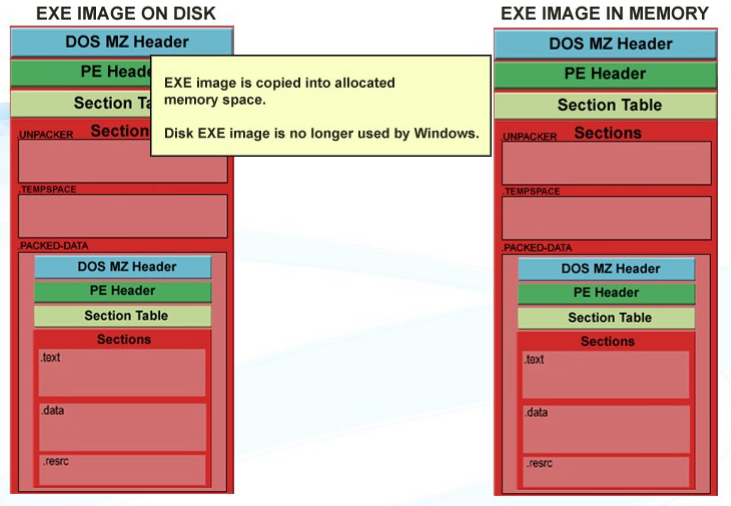


*Chương trình được khởi chạy*

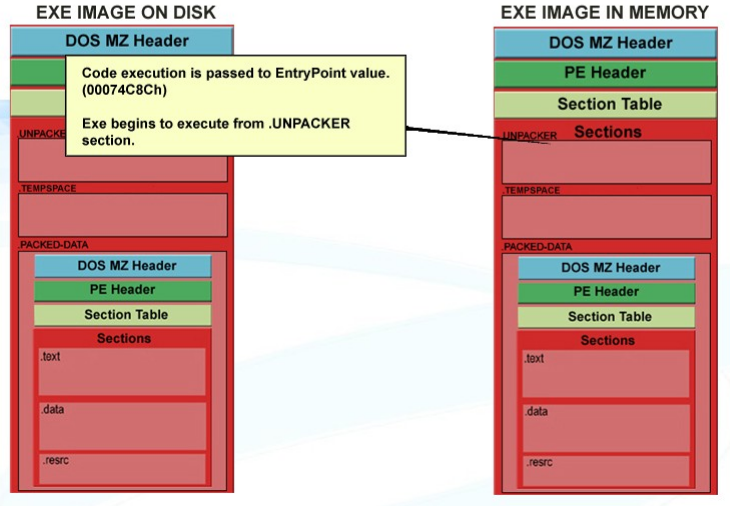
*Trường PE Header chứa các dữ liệu tổng quan về file cùng trường Optional Header*



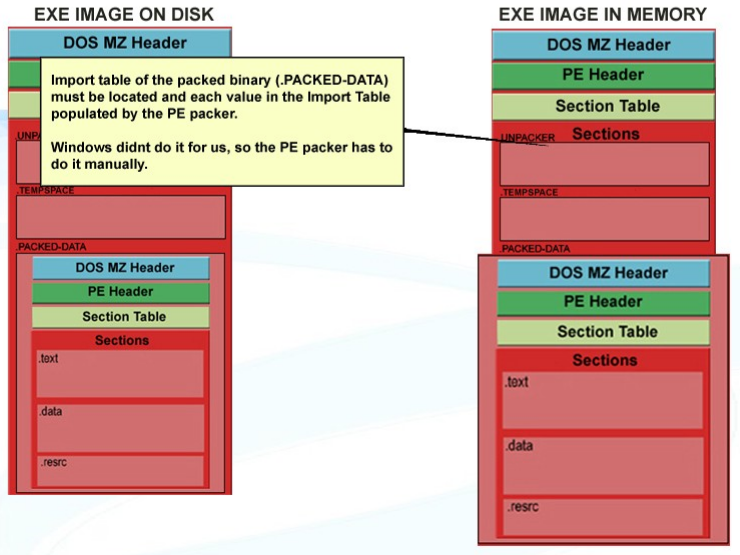
*Section Table chính là nơi chứa các thông tin chi tiết về các section tương ứng phía dưới. Các section đầu tiên chính là của packer*



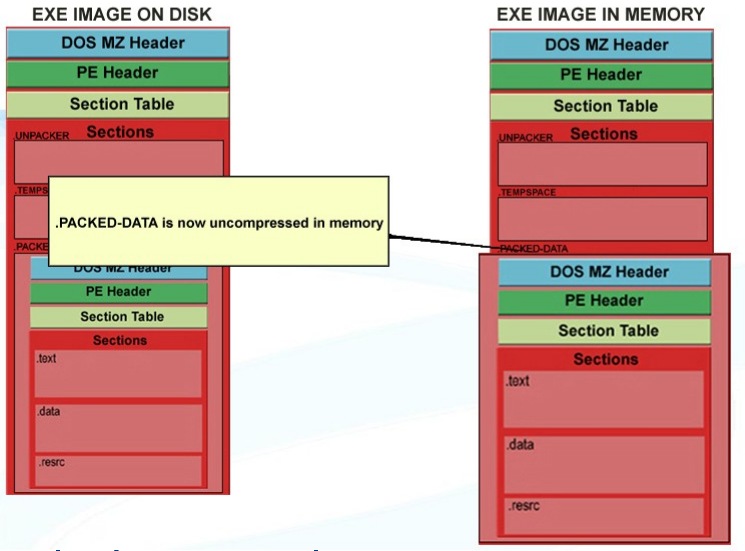
*File được load lên trên memory để thực thi*



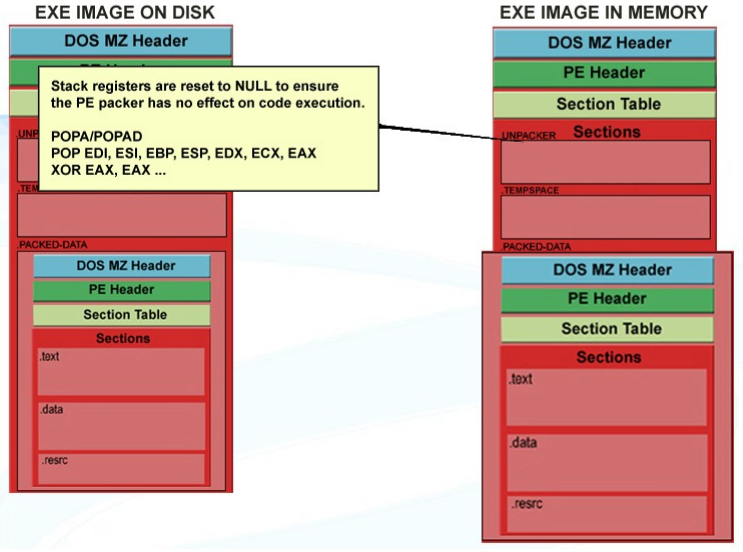
*Tại Optional Header, chương trình tìm đến EntryPoint chính là điểm bắt đầu chạy code. Với file đã bị pack, thì đây là điểm thực thi code giải nén của packer*



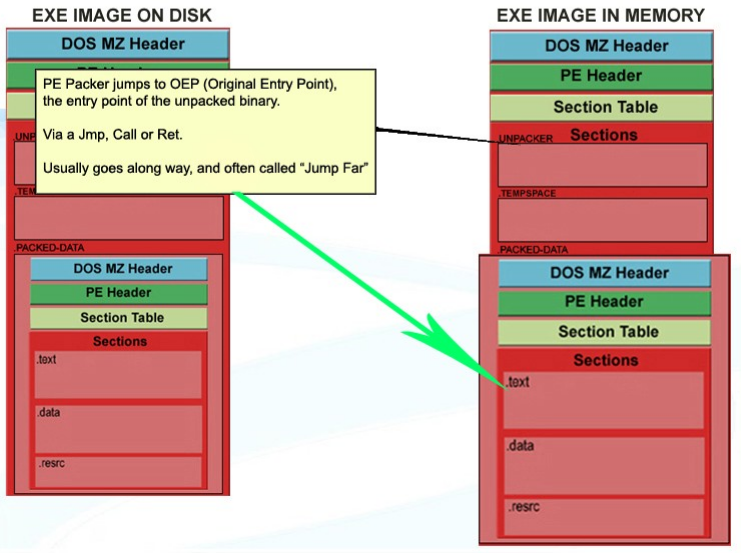
*Bảng Import Table (chứa các hàm thư viện ngoài) được packer tự build lại*



*Sau đó, packer sẽ khởi chạy chương trình chính. Sau khi được giải nén, file exe sẽ phình ra hơn so với trên disk*



*Mặc định khi chương trình được thực thi, các thanh ghi mang giá trị quy định của hệ điều hành. Packer cần lưu lại tất cả các giá trị này, để thực hiện việc giải nén mà ko để ảnh hưởng đến chương trình chung => phải push tất cả các thanh ghi vào Stack. Sau khi hoàn thành giải nén, packer khôi phục lại trạng thái ban đầu của các thanh ghi.*



*Sau khi hoàn thành tất cả nhiệm vụ, Packer sẽ jump đến địa chỉ entrypoint của chương trình gốc. Đây chính là điểm ta cần tìm khi unpack*

**C. MALWARE**

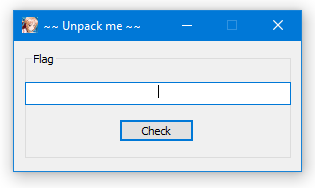
**II. BÀI TẬP**

**A. BÀI TOÁN 1 - UPX**

*[MỨC ĐỘ DỄ]*

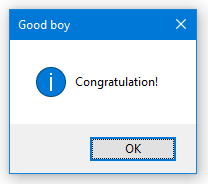
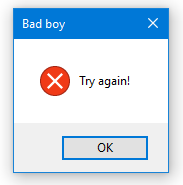
ĐỀ BÀI

Cho một phần mềm A máy tính có giao diện như sau:



Input: một chuỗi nhập từ bàn phím vào khung text

Output: Hiển thị thành công nếu chuỗi nhập vào khớp, hiển thị lỗi nếu không khớp



Nhiệm vụ là ta sẽ đọc code assembly và dịch ngược chương trình xem cách mà chương trình so sánh chuỗi nhập với key vào để hiển thị thông báo. Từ đó ta sẽ tìm được key.

Trước khi thực hiện nhiệm vụ trên, ta cần phải unpack phần mềm này sau đó mới dễ dàng thực hiện dịch ngược.

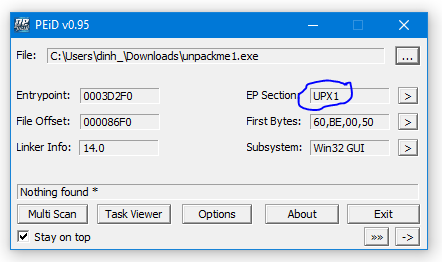
LỜI GIẢI

*CÔNG CỤ SỬ DỤNG:*

* PEID dùng để phát hiện Packer
* OllyDbg 2.01 dùng để debug
* Scylla v0.9.7c (ImpREC 1.7e) dùng để sửa lại Import Table

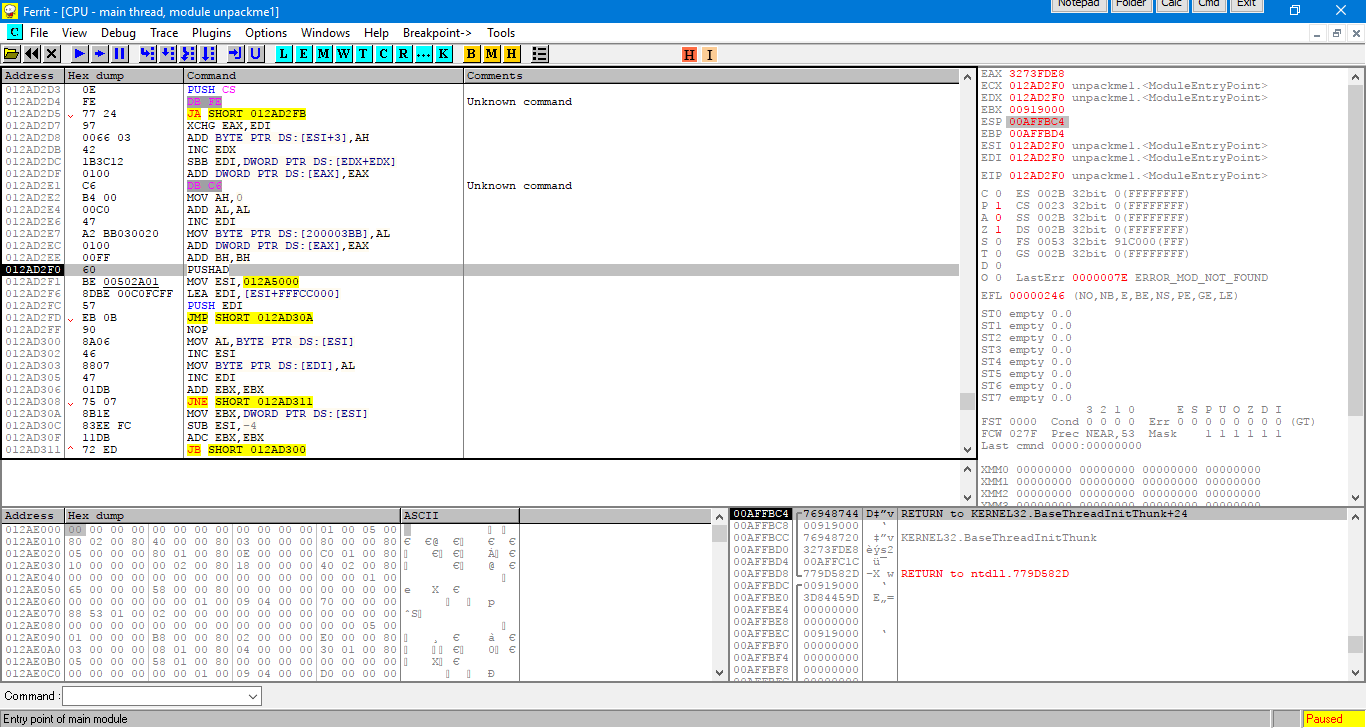
*CÁC BƯỚC:*

Bước 1. Do phần mềm đã được bảo vệ bởi packer, ta tiến hành unpack. Sử dụng PEID xác định packer



Tại trường EP Section, PEID cho ra thông tin UPX là packer

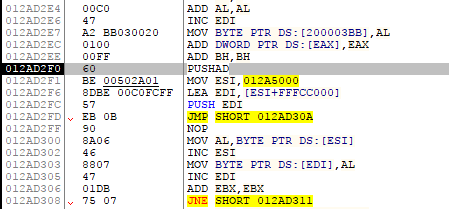
Bước 2. Mở phần mềm A bằng OllyDbg



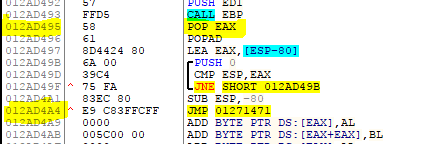
Bước 3. Ta tìm hiểu cơ chế hoạt động của UPX Packer, ta xác định cách thức để unpack như sau

* Do phần mềm A được đóng gói trong 1 file exe của packer, nên khi khởi chạy, packer sẽ tiến hành giải nén phần mềm A
* Do quy định của hệ điều hành, thông số các thanh ghi khi phần mềm khởi chạy phải đc chỉ rõ, nên các packer sẽ phải lưu lại các giá trị của thang ghi bằng lệnh PUSHAD
* Code giải nén của packer được thực hiện
* Sau khi thực hiện xong code giải nén, packer cần phải khôi phục các giá trị của thanh ghi. Nên sẽ sử dụng lệnh POPAD
* Packer hoành thành nhiệm vụ: giải nén và khôi phục thanh ghi. Nên tiếp theo packer sẽ nhảy đến vị trí OEP (Origin Entry Point) của phần mềm A. Đó chính là điểm ta cần tìm
* Packer nhảy đến OEP bằng lệnh JMP (jump không cần điều kiện)
* Nếu có nhiều lệnh JMP thì lệnh JMP nào nhảy đi xa, xa khỏi section hiện tại sẽ là OEP.

Bước 4. Đầu tiên ta tìm kiếm lệnh POPAD trong danh sách các câu lệnh



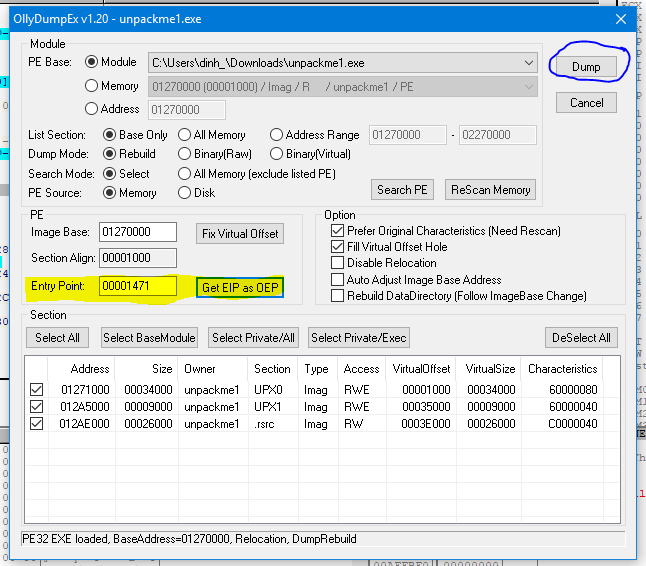
*EP của Packer ngay tại PUSHAD*



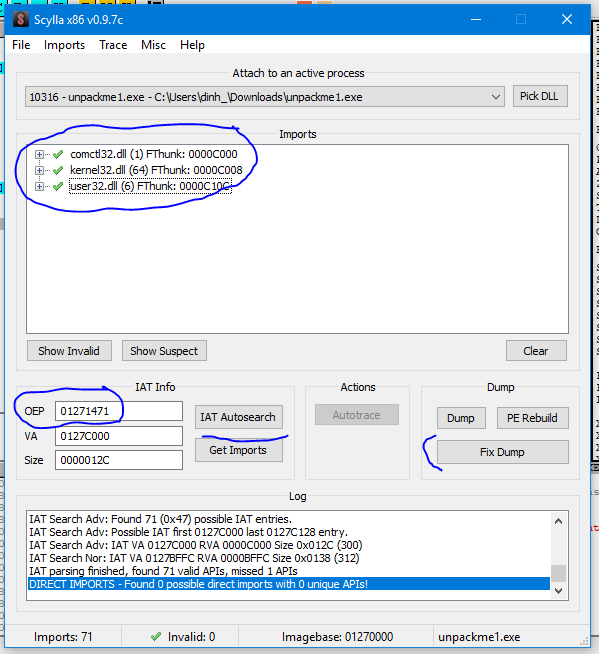
*Tìm kiếm lệnh POPAD*

Vì sau khi khôi phục thanh ghi, packer sẽ nhảy đến OEP thật nên ta chỉ cần tìm lệnh JMP XXX gần nhất sau POPAD. XXX chính là địa chỉ OEP thật. Với bài này OEP cần tìm là **01271471**

Bước 5. Tiến hành jump đến đía chỉ OEP tìm được. Tại đây ta sẽ dump file phần mềm này với OEP vừa tìm được



Bước 6. Sau khi dump, ta cần sửa lại bảng Import Table. Tiến hành mở Scylla, nhập OEP tìm được, Get Import và Fix Dump



Bước 7. Ta thu được phần mềm A đã unpack có kích thước lớp hơn file đã được pack ban đầu (do đã được nén lại)

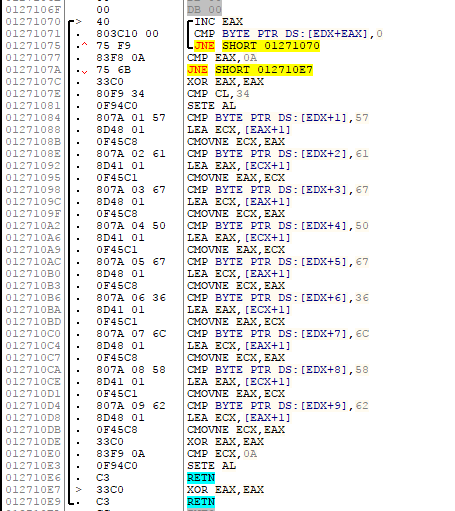
Bước 8. Tiến hành load phần mềm A đã unpack vào trong OllyDbg để tìm key

Bước 9. Dự đoán phần mềm ở dạng Win32 Application, ta sẽ tìm các hàm mà chương trình nào cũng sử dụng (thông tin các hàm có thể tra trên msdn của Microsoft)

* Để tạo cửa sổ, hàm sử dụng DialogBoxParam(…)
* Để hiển thị Popup thường là dùng hàm MessageBox(…)
* Lấy dữ liệu từ Edit Text bằng GetDlgItemText(…)

Đặt các breakpoint tại đây ta sẽ theo dõi được luồng chạy của phần mềm.

Bước 10. Ta theo dõi dữ liệu đi qua hàm GetDlgItemText(…) nhận thấy đoạn code so sánh text ta nhập vào



Tại đây ta nhận thấy phần mềm A so sánh từng kí tự mà ta nhập vào với mã hex của các kí tự khóa, ta ghi lại các mã hex này rồi chuyển qua ASCII sẽ thu được key

34 -> 4

57 -> W

61 -> a

…

Cứ như vậy ta sẽ thu được chuỗi key như sau **4WagPg6lXb**

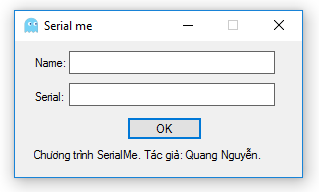
Bước 11. Thử key này vào và thành công

**B. BÀI TOÁN 2 - AOSUKE**

*[MỨC ĐỘ DỄ]*

ĐỀ BÀI

Cho phần mềm B có giao diện như sau:



Input:

* Nhập Name
* Nhập Serial

Ouput: chương trình sẽ tạo ra chuỗi serial từ Name, sau đó kiểm tra xem có đúng với serial hay không?

Nhiệm vụ của ta là dịch ngược chương trình, tìm hiểu cách tạo mã, viết code tạo mã và tiến hành test code đó qua 500 test case.



*Code connect tới địa chỉ server, server gửi lần lượt 500 Name, với mỗi Name ta gửi lại serial tương ứng! Nếu pass 500 test case đó, ta nhận được flag và hoàn thành bài toán*

LỜI GIẢI

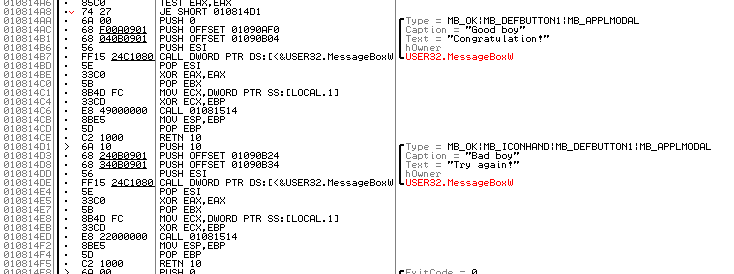
*CÔNG CỤ SỬ DỤNG:*

* OllyDbg 2.01 dùng để debug
* Python dùng để viết code khai thác và làm việc với server

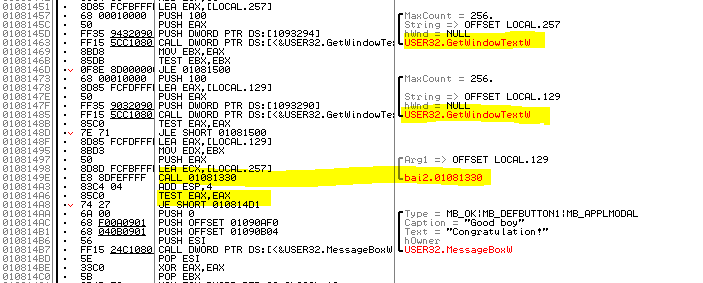
*CÁC BƯỚC:*

Bước 1. Mở chương trình B bằng OllyDbg. Do chương trình không bị pack nên sẽ khởi chạy ngay tại EntryPoint

Bước 2. Biết rằng chương trình sẽ hiển thị hộp thoại thành công hay thất bại, nên ta tiến hành tìm đến vị trí chứa các chuỗi thông báo.



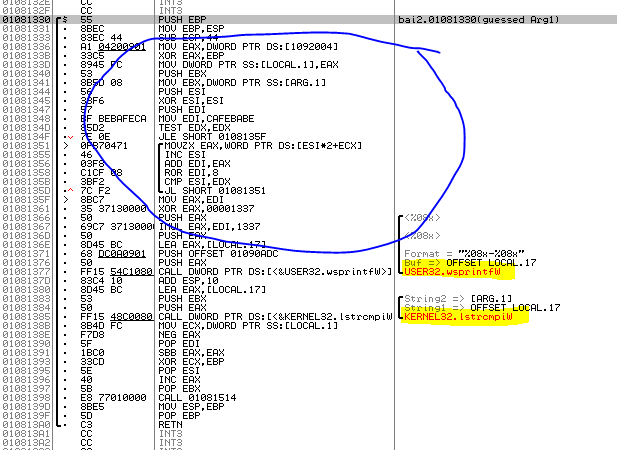
Bước 3. Nhận thấy sẽ có lúc chương trình phải kiểm tra chuỗi được mã hóa từ Name có chuẩn với chuỗi serial ta nhập vào hay không?



Ta thấy có 2 hàm GetWindowTextW(….), khi trace qua 2 hàm này ta thấy được đây là 2 hàm lấy dữ liệu từ edit text Name và Serial

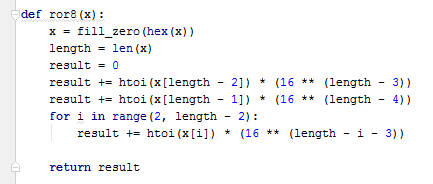
Sau khi lấy xong Name và Serial ta nhập vào, chương trình sẽ gọi hàm **CALL 01081330** với tham số được PUSH ngay phía trên nhằm mục đích tạo ra chuỗi mã hóa từ Name. Kết quả so sánh sẽ được trả về ở EAX, nếu EAX = 0 tức là serial sai, chương trình sẽ jump đến chuỗi thông báo sai.

Bước 4. Tiến hành đi sâu vào hàm tạo mã và so sánh.

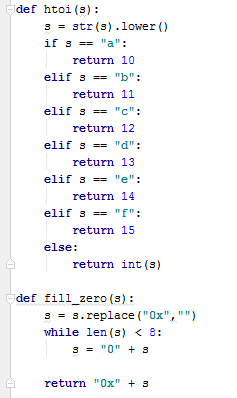


Bước 5. Tiến hành viết code Python làm việc với Server. Ta cần viết lại hàm ROR 8, có nhiệm vụ chuyển từ 0xABCDEFGH thành 0xGHABCDEF. Có nhiều cách để viết hàm này , chuyển qua decimal rồi tính lại chuỗi hex, hoặc chuyển sang binary rồi rotate.

Bài này ta sẽ làm theo cách 1

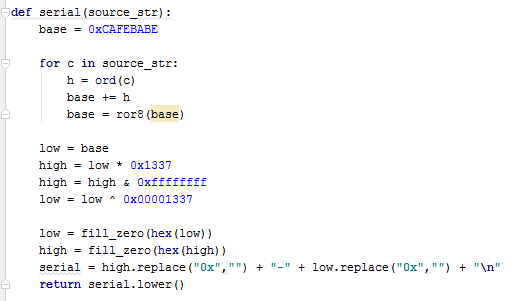


*Hàm ror xxx, 8*

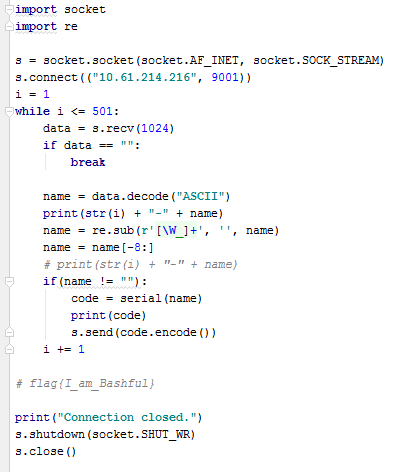


*Hàm htoi() chuyển hex “a,b,c,d,e,f” to int*

*Hàm fill\_zero() để chuyển 0x1234567 thành 0x01234567*



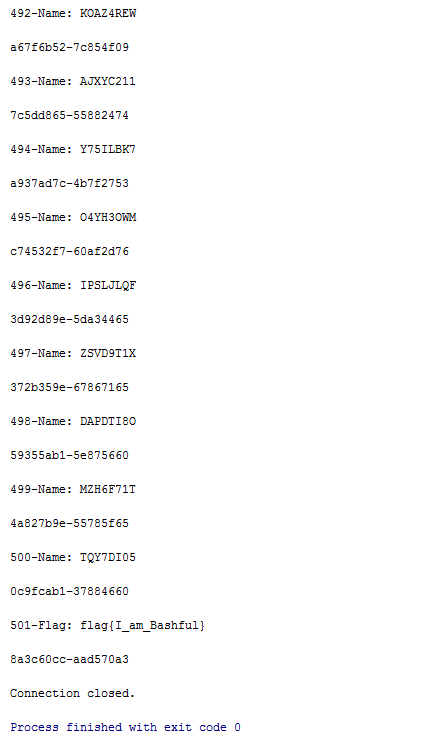
*Hàm tính serial*



*Kết nối và trao đổi dữ liệu vói server*

*Bước 6. Source code tại* [*https://gist.github.com/dinhvankiet124/b9a9b534dbbcd39ec6fe1db29d945e1a*](https://gist.github.com/dinhvankiet124/b9a9b534dbbcd39ec6fe1db29d945e1a)

*Thực hiện chạy code ta thu được kết quả như sau:*



*Ta lấy được flag là “flag{I\_am\_Bashful}”*

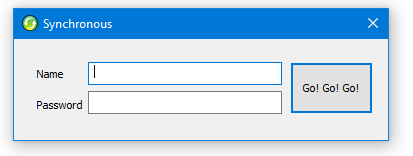
Bước 7. Submit flag và nhận kết quả thành công

**C. BÀI TOÁN 3 - SYNCHRONOUS**

*[MỨC ĐỘ BÌNH THƯỜNG]*

ĐỀ BÀI

Cho phần mềm C có giao diện như sau:



Input:

* Nhập Name
* Nhập Password

Ouput: chương trình sẽ tạo ra chuỗi Password từ Name, sau đó kiểm tra xem có đúng với Password hay không?

Nhiệm vụ của ta là dịch ngược chương trình, tìm hiểu cách tạo mã, viết code tạo mã và tiến hành test code đó qua 500 test case.

LỜI GIẢI

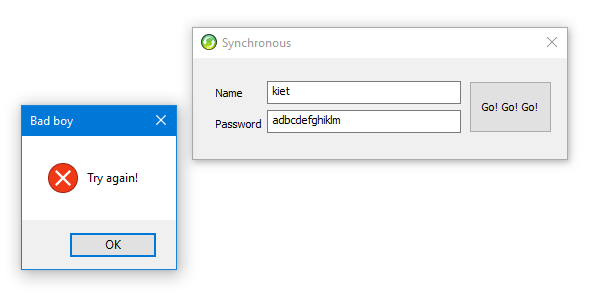
*CÔNG CỤ SỬ DỤNG:*

* OllyDbg 2.01 dùng để debug
* Python dùng để viết code khai thác và làm việc với server

*CÁC BƯỚC:*

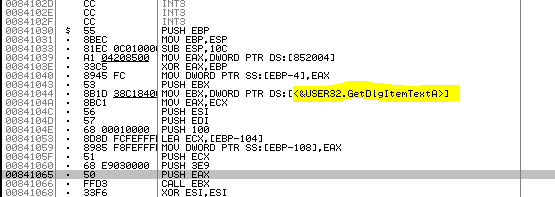
Bước 1. Mở chương trình C bằng OllyDbg. Do chương trình không bị pack nên sẽ khởi chạy ngay tại EntryPoint

Bước 2. Ta sẽ nhập Name và Password xem chương trình sẽ thông báo gì. Ta tìm vị trí dòng thông báo đó trong Olly



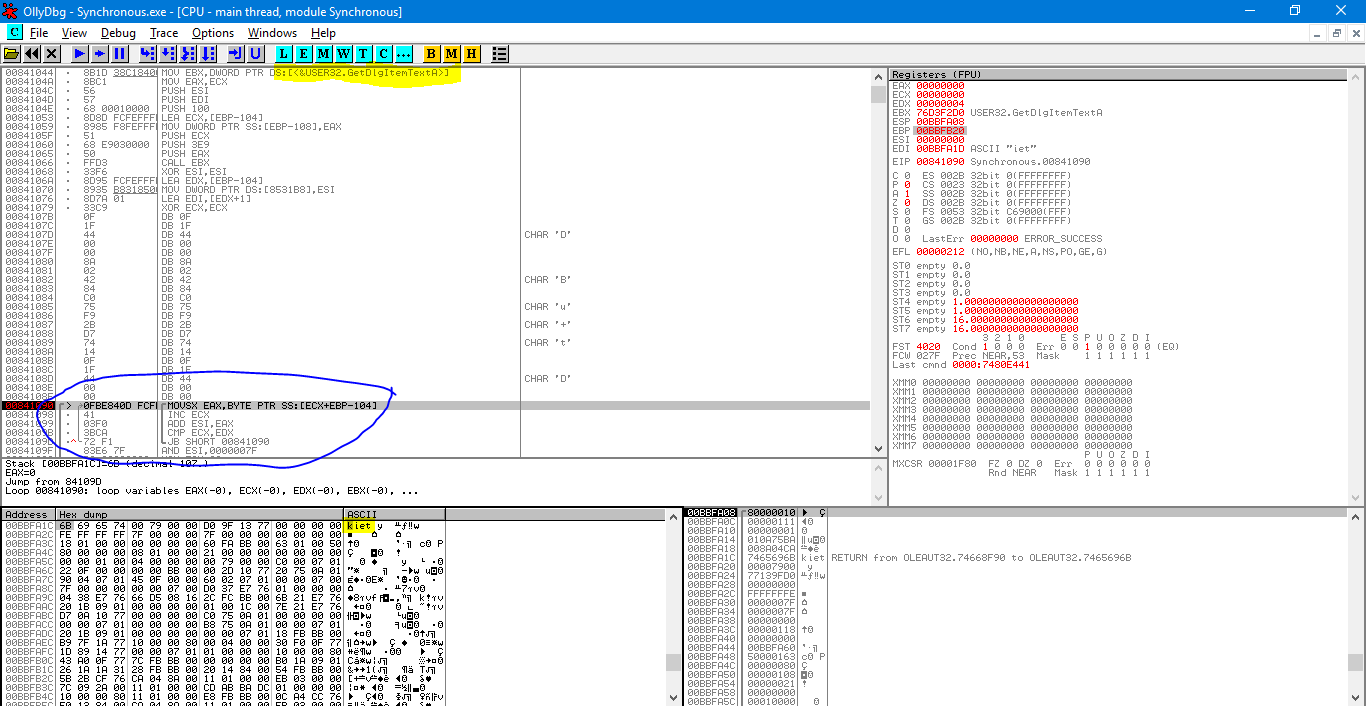
*Thông báo lỗi khi ta nhập sai*

Tiếp tục trace đến đoạn mà chương trình lấy dữ liệu từ Edit Text, thường là hàm GetDlgItemText()



*Vị trí code lấy dữ liệu*

Bước 3. Ta biết rằng sau đoạn code lấy dữ liệu sẽ là đoạn code xử lý. Thường chương trình sẽ phải duyệt Name để tạo ra Password. Ta tiến hành tìm vòng lặp nào đó ở phía dưới.



*Kéo xuống một chút ta sẽ thấy một vòng lặp ở đây*

Quan sát kĩ vòng lặp này, ta có nhận xét sau:

* EBP – 104 là vị trí Name “kiet” của ta
* ECX là biến đếm, tương đương với index

Đoạn code được viết lại như sau:

ESI = 0 (giá trị trước khi vào vòng lặp)

EDX = length(“kiet”) ( = 4 )

// vòng lặp

EAX = arr[i]

ECX++

ESI += EAX

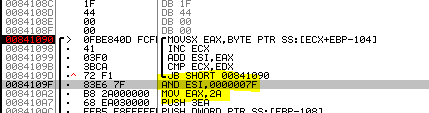
If (ECX < EDX) {

Quay lại đoạn gán EAX

}

=> Chương trình sẽ cộng tổng hex các kí tự của Name, kết thúc vòng lặp ta có tổng S

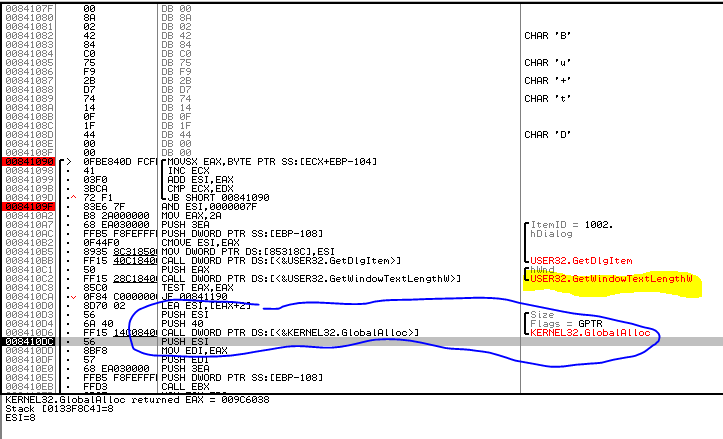
Bước 4. Tiếp tục phân tích tiếp



*2 đoạn code đáng chú ý sau vòng lặp. Ta ko xét lệnh PUSH vì nó thường để nhét các tham số cho lời gọi hàm sắp tới, nên nó ko liên quan đến đoạn code phía trên lắm*

2 dòng code tiếp thực hiện AND tổng S với 0x0000007F, nếu tổng bằng = 0 thì sẽ gán S = 0x2A. Ta chưa biết chương trình tính tổng Name để làm gì nên ta cứ note lại.

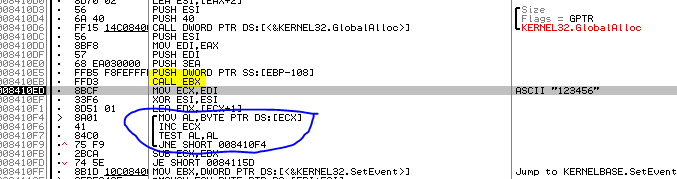
Bước 5. Xét tiếp đoạn code tiếp, ta thấy chương trình bắt đầu xử lý Password do ta nhập vào. Chương trình lấy độ dài chuỗi Password, thu được length = 6



*Đoạn code xử lý trường Password*

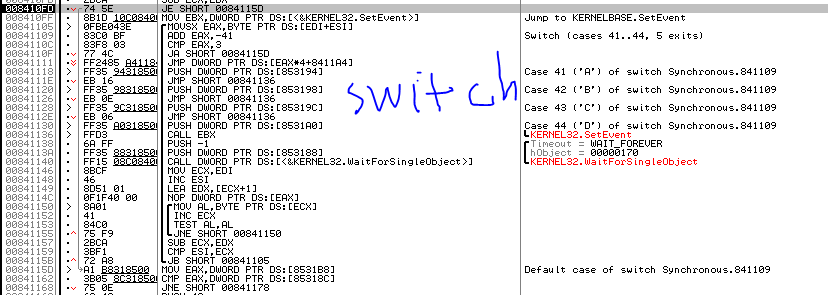
Sau khi lấy length, chương trình cấp phát một vùng nhớ với size là 6 cho đoạn Password ta nhập vừa rồi

Bước 6. Tiếp theo chương trình lấy nội dung trường Password ra và kiểm tra xem buffer vừa cấp phát có rỗng không?



*CALL EBX chính là CALL GetDlgItemText()*

Bước 7. Tiếp theo sau khi đã có Password, có Name, chương trình đi vào 1 vòng lặp qua các kí tự của Password, và thực hiện khối lệnh switch()

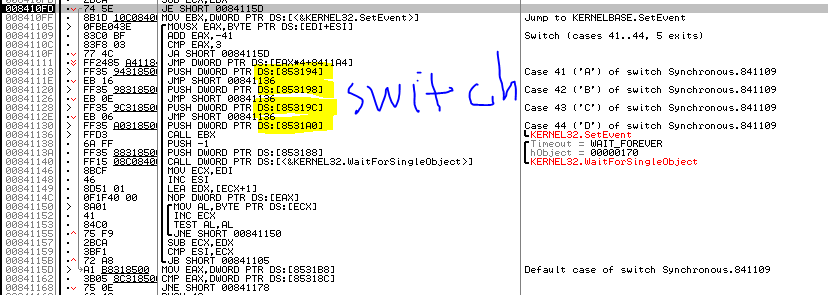


*Olly có thể giúp cho ta thấy điều này*

Lệnh switch chỉ dành cho 4 kí tự ABCD, nếu rơi vào default thì ta sẽ không thu được gì. Nên ta đoán password chỉ chứa 4 kí tự ABCD. Ta tiến hành nhập lại Password mới là ABCDABCD

Tiến hành chạy lệnh kia và quan sát, ta thấy chương trình gọi hàm lưu trong EBX và hàm SetEvent() (lưu trong EBX) và hàm WaitForSingleObject(). 2 hàm này không liên quan lắm đến việc so sánh Password ta nhập với 1 cái gì đó để biết ta nhập đúng. Vậy ta sẽ chỉ theo dõi giá trị các thanh ghi thay đổi khi đi qua hàm này

Bước 8. Thực hiên trace nhiều lần ta thu được quy luật sau với khối lệnh switch



S2 = 0

Switch(char c) {

Case “A”:

S2 += 0x1

Break

Case “B”:

S2 += 0x2

Break

Case “C”:

S2 += 0x4

Break

Case “D”:

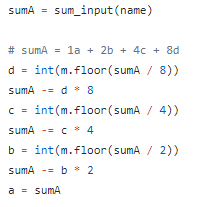
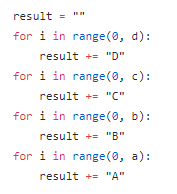
S2 += 0x8

Break

}

Sau đó chương trình sẽ so sánh tổng này với tổng của Name S1 vừa rồi. Ta đã tìm ra cách mà chương trình so sánh Password ta nhập vào với Password tạo từ Name. Ta cũng nhận ra 1 Name có thể có nhiều Password

Bước 9. Ta đề xuất cách giải quyết như sau:

* Đầu tiên ta sẽ tính tổng hex các kí tự của Name giống như chương trình đã làm
* Tiếp theo ta đặt Password có tổng được tính bởi công thức S2 = 1a + 2b + 4c + 8d
* Biết S2 = S1, công việc giờ ta sẽ tìm a,b,c,d. Trong đó a,b,c,d lần lượt là số các kí tự A, B, C, D trong chuỗi Password (không quan trọng thứ tự)
* Cách để tìm a,b,c,d rất đơn giản là
  + S2 -= [S2 / k] \* k với k = 8, 4, 2, 1
  + [S2 / k] chính là SL kí tự k tương ứng
  + Cách tìm được thể hiện bằng Python như sau: 
* Sau khi tìm được a,b,c,d, ta chỉ việc lặp từng ấy lần và tạp ra Password 

Bước 10. Source code tại:

<https://gist.github.com/dinhvankiet124/006386ecd6f1008ae883beaa3b0730bd>

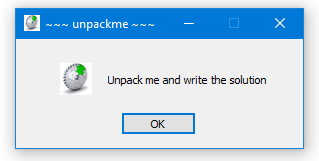
Sau khi có thể tạo được Password chuẩn, ta connect lên server và chạy qua các testcase

**D. BÀI TOÁN 4 – ASPACK, MPRESS, PETIT**

*[MỨC ĐỘ DỄ]*

ĐỀ BÀI

Cho một chương trình đã được pack bởi các packer (tương tự nhau). Công việc của ta là cần phải unpack chương trình này.



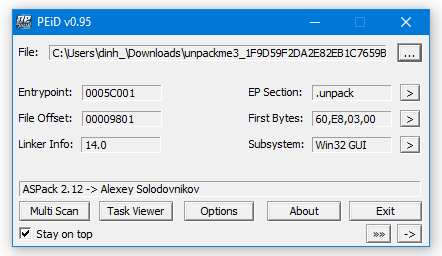
CÁCH LÀM

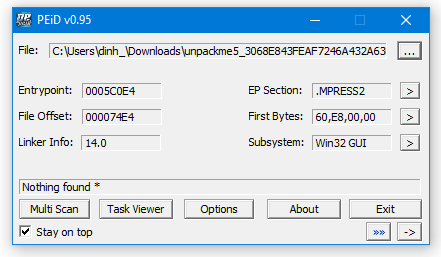
*CÔNG CỤ SỬ DỤNG*

* PEID dùng để phát hiện Packer
* OllyDbg 2.01 dùng để debug
* Scylla v0.9.7c (ImpREC 1.7e) dùng để sửa lại Import Table

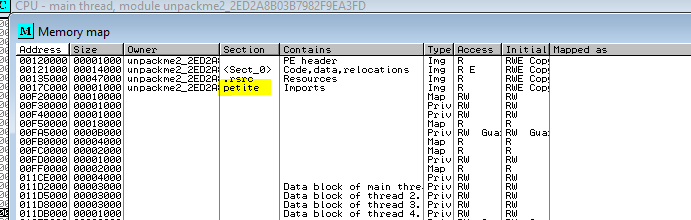
*CÁC BƯỚC*

Bước 1. Sử dụng PEID để xác định Packer



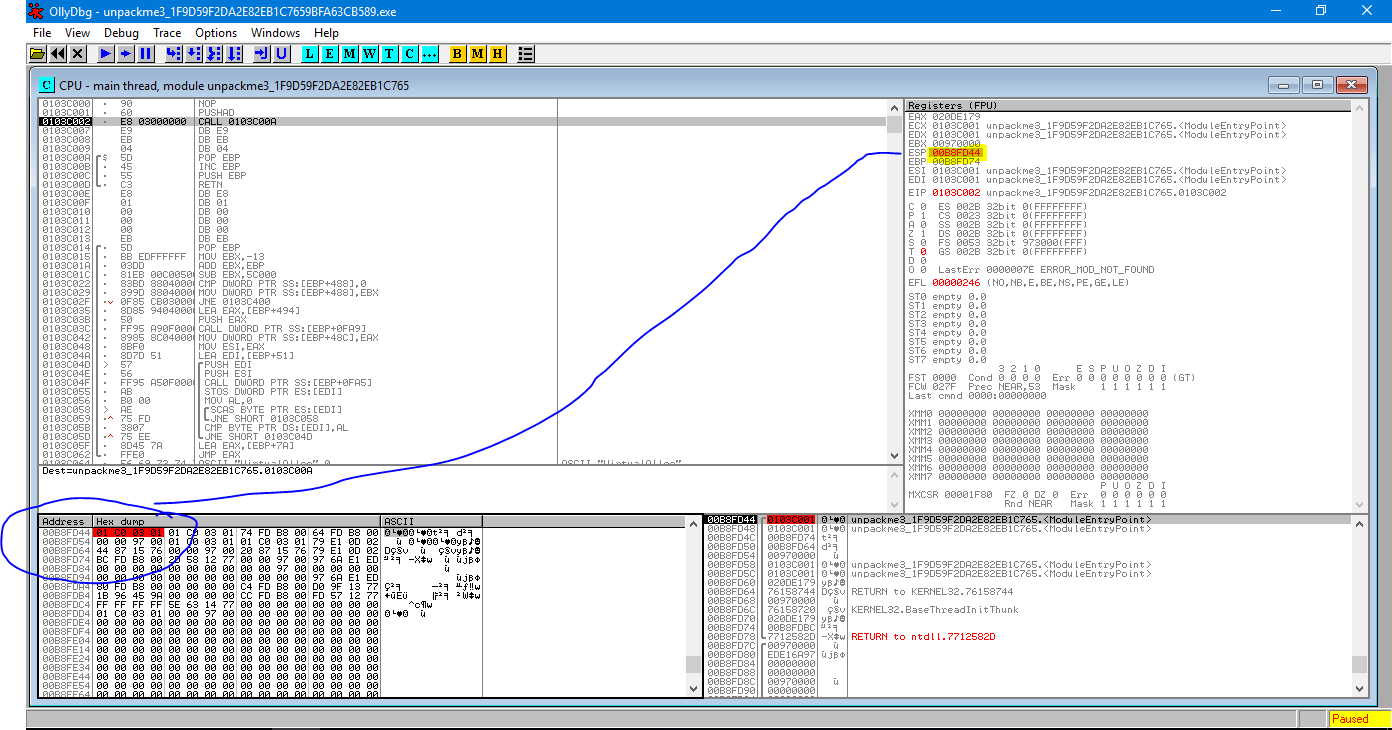


Riêng packer Petite thì PEID không nhận diện được, ta phải tìm trong các section của PE file

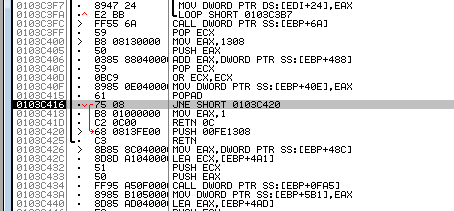


*Sử dụng Olly, load packer vào à xem các section khi được load vào memory*

Bước 2. Dựa vào tính chất của 1 số loại packer cơ bản là luôn lưu giá trị các thanh ghi bằng lênh PUSHAD. Ta cần bắt lấy sự kiện packer khôi phục các giá trị cũ các thanh ghi. Nên sau khi chạy PUSHAD, ta đặt breakpoint ở thanh ghi ESP (thanh ghi ESP luôn trỏ về đầu Stack)

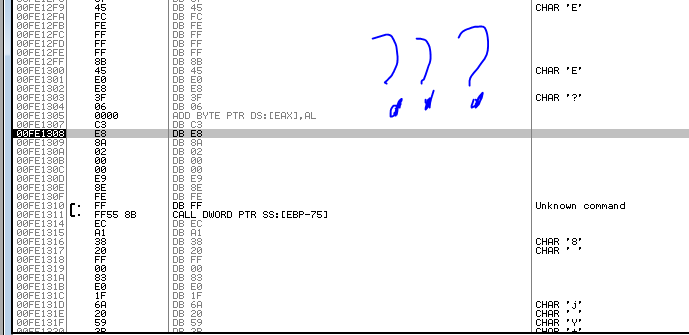


Bước 3. Ta chạy chương trình đến khi chương trình khôi phục các thanh ghi bằng POPAD.

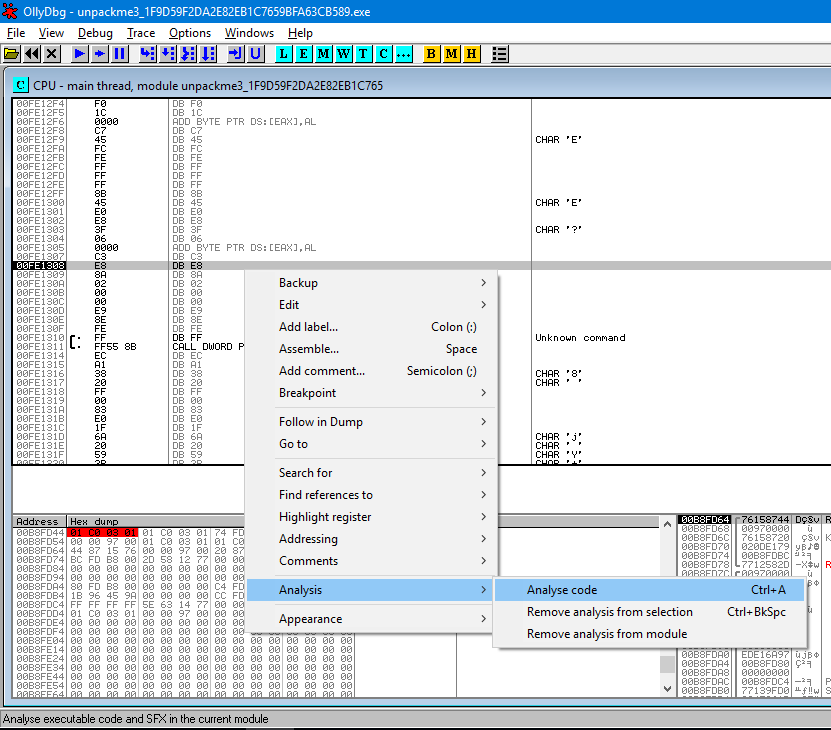


Sau lệnh POPAD không hề có lệnh JMP như UPX, nên ta không biết làm thế nào. Cách tốt nhất vẫn là chạy tiếp xem như thế nào

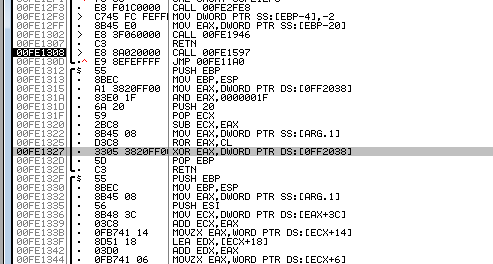
Bước 4. Tiến hành chạy tiếp ta đến lệnh RETN là kết thúc hàm, ta được đưa tới 1 section khác



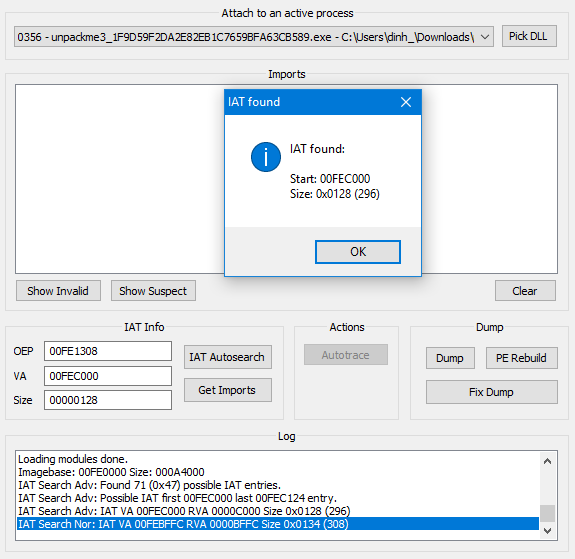
Tại đây ta không thấy câu lệnh mà toàn thấy dữ liệu. Đây là do Olly đã hiểu nhầm. Chuột phải, ta chọn Analyse code để phân tích code tại đây

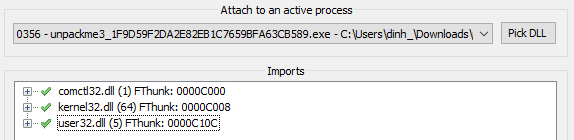


Ta thu được



Do đặc điểm của Packer là sau khi giải nén sẽ nhảy đến Entry Point của chương trình gốc và thường là nhảy rất xa, sang hẳn một section khác. Nên ta xác định đây là OEP, tiến hành dump và Fix dump giống như UPX, ta thu được file đã unpack



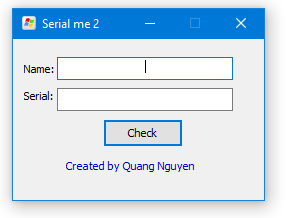


**E. BÀI TOÁN 5 – WINCRYPT**

*[MỨC ĐỘ KHÓ]*

ĐỀ BÀI

Cho một chương trình như sau:



Input:

* Nhập Name
* Nhập Serial

Ouput: chương trình sẽ tạo ra chuỗi Serial từ Name, sau đó kiểm tra xem có đúng với Serial hay không?

Nhiệm vụ của ta là dịch ngược chương trình, tìm hiểu cách tạo mã, viết code tạo mã và tiến hành test code đó qua 500 test case.

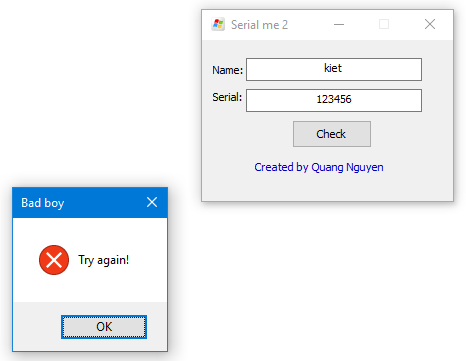
LỜI GIẢI

*CÔNG CỤ SỬ DỤNG*

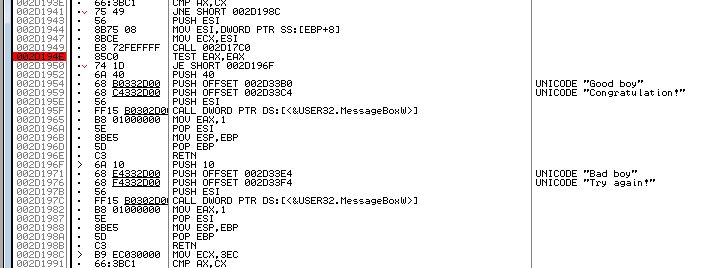
* OllyDbg 2.01 dùng để debug
* VisualStudio code C/C++

*CÁC BƯỚC*

Bước 1. Vẫn như 2 bài tập về Serial ở trên, ta nhập thử 1 bộ Name – Serial để xác định đoạn code xử lí

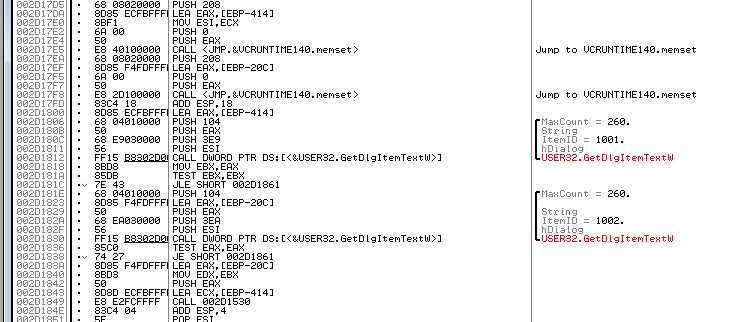


*Thông báo lỗi khi nhập sai*

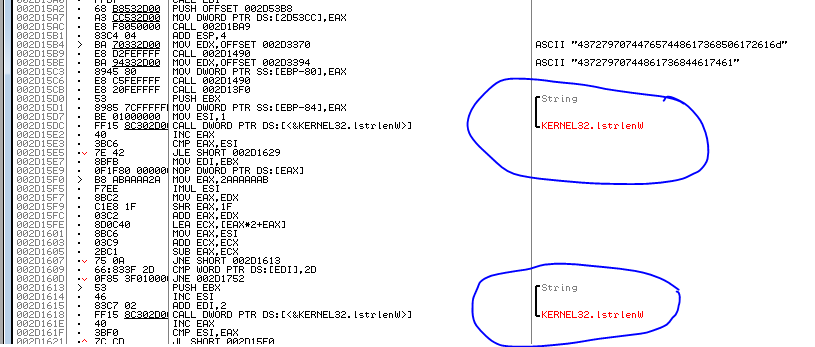


*Ví trí code thông báo lỗi*

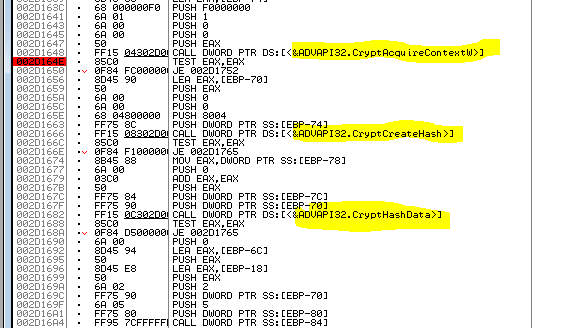
Bước 2. Xác định đoạn code lấy text từ edit text



Các hàm được gọi sau lệnh này thường là hàm chính của chương trình. Ta trace theo các hàm được gọi sau đó, đến vị trí khả nghi nhất. Các vị trí khả nghi thường là lặp duyệt, lấy length của trường ta nhập vào



Quan sát tổng quan, ta nhận thấy có các hàm mã hóa trong API của Windows

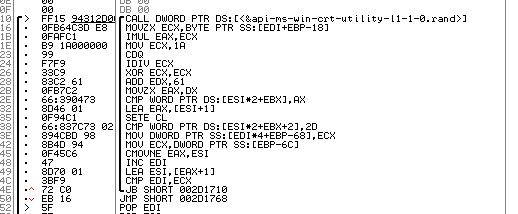


Bước 3. Tiếp theo ta cần tìm hiểu các hàm này. Google tên các hàm này ta được ví dụ sau của Microsoft (<https://msdn.microsoft.com/en-us/library/windows/desktop/aa382380(v=vs.85).aspx)>

Đọc ví dụ này ta hình dung được cách thức mã hóa và khẳng định chương trình mã hóa Name của ta bằng cách thức tương tự như trong ví dụ

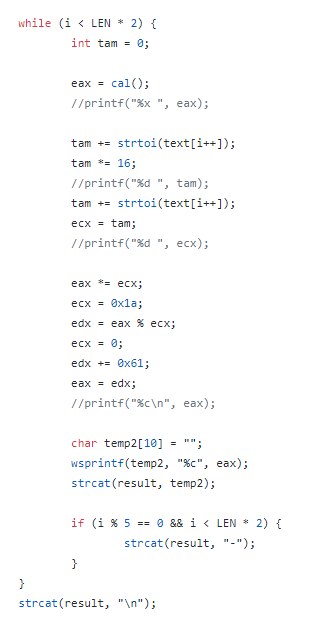
Theo dõi các hàm trên và tìm đến vị trí buffer chứa chuỗi đã mã hóa. Ta thấy sau khi được mã hóa, chuỗi đã mã hóa được chạy qua 1 hàm mã hóa khác và phân tách ra theo dạng CD key AAAAA-AAAAA-AAAAA-AAAAA-AAAAA

Bước 4. Xét tiếp đoạn code mã hóa lần 2 theo dạng CD key, cách nhận biết hàm này là thông qua vòng lặp



*Thuật toán mã hóa lần 2*

Ta cần viết 1 hàm tương tự để thực hiện công việc này.



Một điều chú ý là là length chuỗi mã hóa gấp đôi do trường Name mà chương trình đọc vào là kiểu w\_char chứ ko phải char

Bước 5. Sau khi xác định được quy trình mã hóa của chương trình, ta viết code connect lên server và thực hiên các bước như chương trình làm. Phần này khá dài

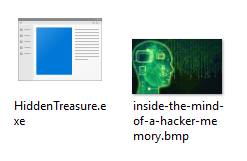
Source code: <https://gist.github.com/dinhvankiet124/65545dc85a21acc2e587e96dbf61fbbc>

**F. BÀI TOÁN 6 – HIDDENTREASURE**

*[MỨC ĐỘ BÌNH THƯỜNG]*

ĐỀ BÀI

Ta có 1 file nén, khi giải nén file này ra, ta được như sau:



Input:

* 1 file HiddenTreasure.exe
* 1 file ảnh bitmap

Ouput: chương trình kia đã dấu flag vào tấm ảnh này, ta cần tìm flag

Nhiệm vụ của ta là dịch ngược chương trình, tìm hiểu cách mà chương trình ghi nội dung vào 1 tấm ảnh. Từ đó ta đọc ngược nội dung đã được ghi vào tấm ảnh đề bài cho

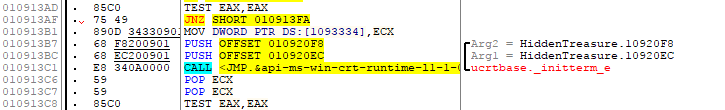
LỜI GIẢI

*CÔNG CỤ SỬ DỤNG*

* OllyDbg 2.01 dùng để debug
* Python
* CFF Explorer để xem nội dung file

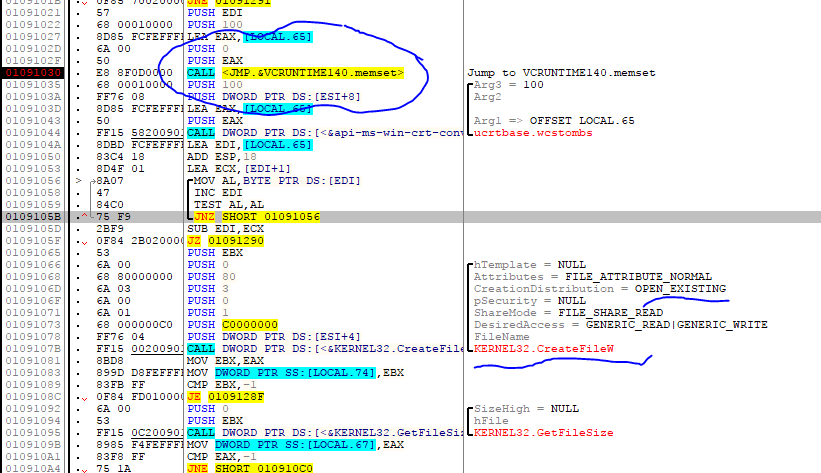
*CÁC BƯỚC*

Bước 1. Đầu tiên ta cần tìm hiểu cách mà chương trình nhận vào ảnh và nội dung. Khởi chạy file HiddenTreasure.exe, nhận thấy chương trình khởi động rồi tắt phụt! Có lẽ chương trình không được cung cấp ảnh và nội dung để ghi. Mở file exe bằng Olly, ta cũng thấy chương trình yêu cầu 2 tham số truyền vào từ CommandLine



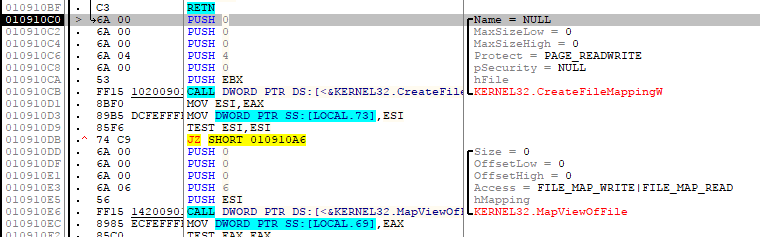
Bước 2. Ta cung cấp đối số cho chương trình trong Olly, thử truyền tên ảnh rồi đến một chuỗi hoặc chuỗi rồi đến tên ảnh. Lúc chương trình gọi hàm CreateFile để mở ảnh, nếu ảnh không tồn tại, hàm sẽ return về 0. Từ đó xác định cú pháp chạy chương trình là: HiddenTreasure.exe ABC.bmp kiet

Bước 3. Như thường lệ, để chương trình có thể ghi nội dung vào ảnh, chắc chắn chương trình sẽ cấp phát một dùng buffer, duyệt chuỗi và ghi. Ta trace chương trình đến các chỗ khả nghi như VirtualAlloc, LocalAlloc, Memset, Vòng lặp,..

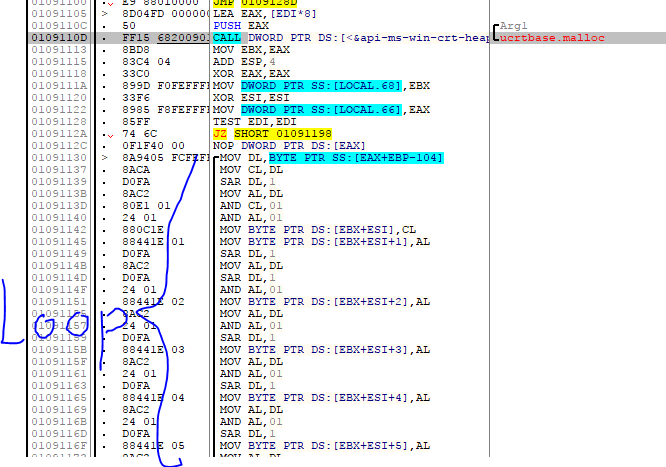


*Nhận thấy hàm memset và CreateFile*

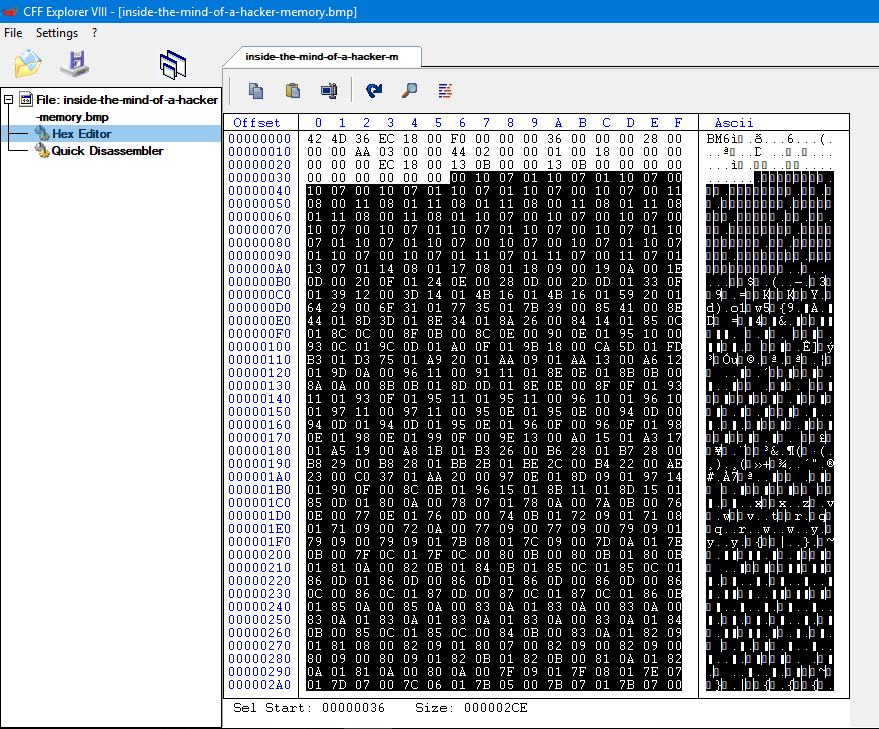
Bước 4. Sau khi mở file bitmap, chương trình map file vào bộ nhớ và tiến hành kiểm tra xem đây có đúng là file bitmap hay không nhờ chữ ký của file.



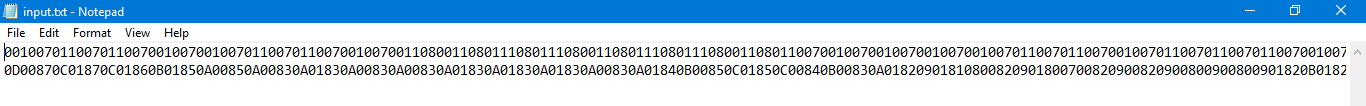
Bước 5. Ngay sau đó chương trình cấp phát bộ nhớ và tiến hành ghi chuỗi vào file



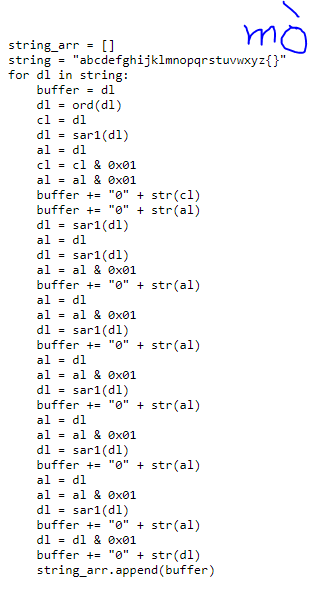
Điều quan trọng bây giờ là ta cần tìm vị trí mà chương trình ghi nội dung vào. Ta nhận thấy trong vòng lặp trên, ESI đóng vai trò là con chạy và chính EBX là con trỏ trỏ đến file ảnh bmp. Đi theo EBX, ta có được vị trí mà chương trình ghi nội dung vào file



Thuật toán chương trình sử dụng cho ouput là 00 và 01, và cách nhau 2 giá trị. Ta không biết nội dung mà chương trình đã ghi vào ảnh (nội dung giấu) dài bao nhiêu, nên ta bôi đen cho đến khi quy luật ghi vào bị phá vỡ. Paste nội dung ra notepad, bỏ các giá trị phân cách (khác 00 và 01), ta thu được



Bước 6. Ta viết chương trình dịch ngược lại thuật toán mã hóa của chương trình để tìm nội dung thật. Mặt khác nhận thấy việc viết hàm ngược khó khăn, nên ta thực hiện viết hàm brute-force



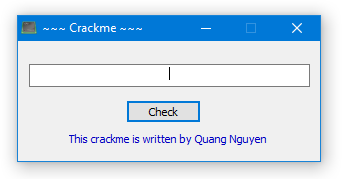
Source code: <https://gist.github.com/dinhvankiet124/bc6af2be2f6221f4512d42bfc3ccf345>

**H. BÀI TOÁN 8 – CRACKME**

*[MỨC ĐỘ KHÓ]*

ĐỀ BÀI

Cho chương trình H có giao diện như sau:



Input: nhập key

Ouput: chương trình kiểm tra key và thông báo đúng hoặc sai

Nhiệm vụ của ta là dịch ngược chương trình, tìm hiểu cách mà chương trình tạo key và so sánh key, từ đó lấy được flag

LỜI GIẢI

*CÔNG CỤ SỬ DỤNG*

* OllyDbg 2.01 dùng để debug

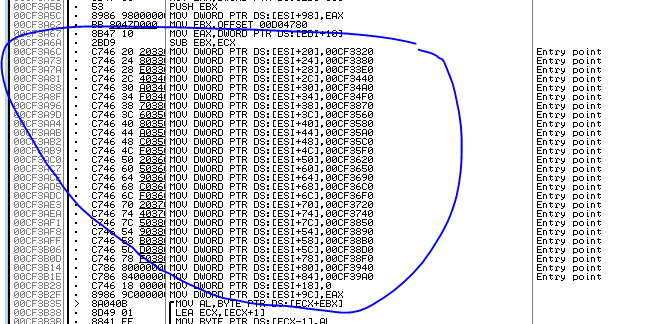
*CÁC BƯỚC*

Bước 1. Dựa trên kinh nghiệp phân tích các file crackme. Ta đi luôn vào phần chính cách thức hoạt động của chương trình. Nhập thử 1 key bất kì để tiến hành trace

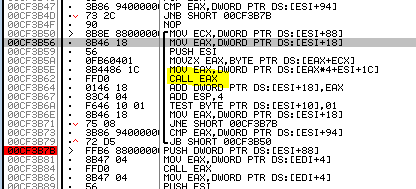
Bước 2. Xét thấy chương trình tạo ra một mạng ảo bằng 2 pipe để tự giao tiếp



Bước 3. Đi vào hàm này, ta thấy chương trình tạo ra 2 buffer, xong copy 1 dãy các hằng vào buffer A. Buffer B sẽ chứa các entry point của các hàm

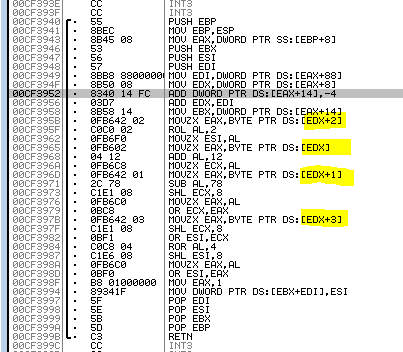


Bước 4. Trong các hàm mà được ghi địa chỉ vào buffer B, ta ko thể hiểu hết chúng, ta cần xem chúng làm gì. Ta tiến đến vòng lặp mà gọi các hàm này



Trong vòng lặp này ECX sẽ chỏ đến buffer A, ESI chỏ đến buffer B

Ta trace lần lượt và sẽ thấy đoạn sau lấy 4 kí tự đầu của chuỗi mà chúng ta nhập vào.



Hàm này được gọi 3 lần, mỗi lần 4 kí tự => key có 12 kí tự (mặc dù trước đó cũng có chỗ check số lượng kí tự ta nhập vào)

Ta gọi hàm này là hàm cal()

Trace tiếp các lần tiếp theo ta nhận được quy tắc sau:

|  |  |
| --- | --- |
| 4 kí tự đầu | 1. cal()  2. + 0xBABECAFE  3. Rol 0x6  4. 0x13371337 –  5. xor 0x13371337  = 0x2648ED87 (hằng) |
| 4 kí tự tiếp | 1. cal()  2. – 0x94C3E659 (hằng) |
| 4 kí tự cuối | 1. cal()  2. ror 0x4  3. not  4. xor 0xABBAABBA  5. + 0x89ABCDEF  6. not |

Bước 5. Từ các chuỗi hằng, ta thực hiện ngược lại các bước trên để tìm ra đủ 12 kí tự. Công việc khó khăn nhất là trace qua tất cả các entry point của các hàm đã xong.

Ta tìm được cờ **R0DE7a6iHuct**