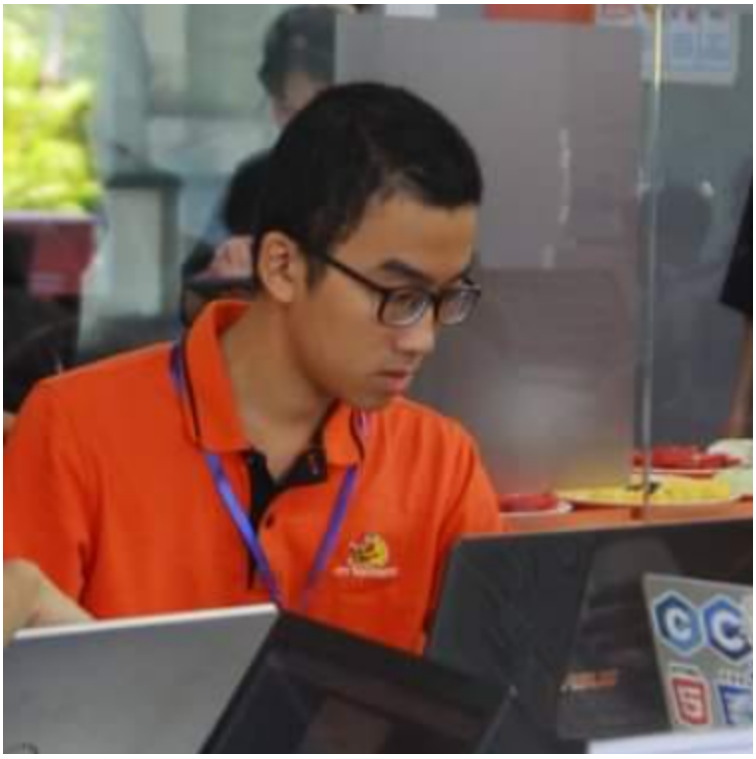
Phương pháp phân tích và unpack các file PE (Portable Executable) bị nén



Phương pháp phân tích và unpack các file PE (Portable Executable) bị nén ###### Author: `HoangNCH` ## :rocket: I. Packed PE file và Packer là gì? **Packed PE (Portable Executable) file** là một file mà mã thực thi gốc của nó sẽ bị ẩn đi khi ta load nó vào các trình disassembler để phân tích và lưu lại bằng cách áp dụng các kỹ thuật như nén, mã hóa, tự động bổ sung một hoặc nhiều section và đoạn stub, cuối cùng là chèn thêm đoạn **Packed Data**. Trong **Packed Data** này có chứa mã thực thi gốc, do đó khi thực thi, **Entry Point** (EP) của chương trình sẽ được chuyển hướng vào vùng code này, gọi là **Original Entry Point** (OEP). OEP là mấu chốt để chúng ta unpack và lấy ra được packed data từ file PE đã bị packed. Các trình **packer** chính là công cụ để tạo ra các file PE bị packed từ các file PE bình thường nhằm mục đích khiến cracker/reverser gặp khó khăn và tốn thời gian hơn trong việc bẻ khóa hoặc đảo ngược phần mềm của họ. Nó cũng được sử dụng bởi những kẻ phát tán malware nhằm "tăng sức đề kháng" cho malware khi gặp phải sự "truy quét" của các phần mềm anti-virus được cài đặt trên hệ thống, bởi khi quét một file PE bị packed các phần mềm anti-virus thông thường sẽ quét theo tuyến tính, từ DOS MZ Header, PE Header rồi các section trong section table. Mà section table của một file PE bị packed như đã nói ở trên sẽ được chèn thêm rất nhiều section "rác", và các section "rác" này với các phần mềm anti-virus thì nó vô hại. Trong khi section thực sự "có hại" nằm ở **Packed Data**, dữ liệu được nén trong **Packed Data** chỉ

được giải nén khi chạy chương trình (runtime). Các bạn có thể tìm hiểu sâu hơn về cấu trúc file PE tại [đây](https://securitydaily.net/tim-hieu-ve-cau-truc-pe-file/), nguyên lý hoạt động của Packer hay nguyên lý unpack tại [đây](https://securitydaily.net/tong-quan-ve-pack-va-unpack/). Mình thì không thích nói lý thuyết suông nên trong bài viết này sẽ dùng các bài lab mình cho là tiêu biểu để nói về kỹ thuật Pack và Unpack. Pack mà mình sử dụng trong tất cả các bài lab này là **UPX** vì nó khá là đơn giản và thân thiện với newbie khi nghiên cứu về unpacking. ## :rocket: II. Phân tích tĩnh (Static analysis) > UPX là một packer hoàn toàn miễn phí và open-source, các bạn có thể tải nó về tại [đây] (https://github.com/upx/upx/releases/tag/v3.96). Ở đây ta sẽ dùng bản **upx-3.96-win64** để pack một phần mềm khá là quen thuộc đó là [Putty] (https://samsclass.info/127/proj/putty.exe). Sau đó sử dụng một tool phân tích tĩnh là [Pestudio](https://www.winitor.com/features) (vì nó khá đa năng, cung cấp cho ta nhiều thông tin về một file thực thi) để so sánh 2 file **putty gốc** và file **putty bị pakced** xem có gì khác biệt. Chạy command prompt với quyền admin, gõ lệnh `upx -o putty_packed.exe putty.exe` để pack file `putty.exe` và cho ra output là file `putty_packed.exe`. ![] (https://i.imgur.com/Ttgyc0m.png) Cho file `putty_packed.exe` và `putty.exe` vào [Pestudio] (https://www.winitor.com/features) để kiểm tra các [artifact] (https://nasbench.medium.com/windows-forensics-analysis-windows-artifacts-part-i-c7ad81ada16c) có trong file PE này. Có thể thấy tất cả các giá trị hash của 2 file này đều khác nhau, nhưng khi click vào để chạy thì cả 2 file này đều hoạt động như nhau không có gì khác biệt. Điều đó chứng tỏ sau khi packed thì nội dung của file `putty.exe` đã bị thay đổi. Đồng thời ở property **signature** của `putty.exe` có giá trị là `Microsoft Visual C++ 7.0 MFC`cho chúng ta biết `putty.exe` được viết bằng C++, trong khi **signature** của `putty_packed.exe` chỉ cho biết packer mà nó sử dụng. Trả lời được câu hỏi "nó được viết bằng ngôn ngữ lập trình gì?" là bước đầu tiên và quan trọng trong quá trình dịch ngược, và đây rõ ràng là khó khăn đầu tiên khi phân tích một file bị packed. ![] (https://i.imgur.com/3YbAB8s.png) Không những vậy, khi kiểm tra mục **sections** của cả 2 file trên ta cũng thấy sự khác biệt rõ rệt: Có thể tóm tắt những điểm khác nhau đó như sau: - `putty.exe`: \+ Có [4 section] (https://keystrokes2016.wordpress.com/2016/06/03/pe-file-structure-sections/) đặc trưng của một file PE là **Executable Code Section** (`.text`), 2 **Data Section** gồm`.rdata` và `.data`, cuối cùng là **Resources Section** (`.rsrc`). \+ Khi so sánh **raw-size** (kích thước của file PE khi được lưu trên ổ đĩa) và **virtual-size** (kích thước của file PE khi được load từ ổ đĩa vào bộ nhớ) ở từng section (ngoại trừ section `.data`), có thể thấy sự chênh lệch **raw size - virtual size** là không đáng kể, **raw size** nhỉnh hơn một chút so với **virtual size**, tuân theo đúng cấu trúc của một file PE thông thường (xem giải thích tại [đây] (http://www.asmcommunity.net//forums/topic/?id=29605)). \+ Về **phân quyền**, có thể thấy file `putty.exe` đã tuân thủ đúng quy tắc bảo mật cơ bản của một file PE: **không cho phép** **quyền ghi** (writable) **và** **quyền thực thi** (executable) **nằm trên cùng một section** . Chỉ có section `.text` có **quyền thực thi** vì đây là nơi chứa toàn bộ các opcode và system call, trong khi đó chỉ section `.data` mới có **quyền ghi** vì nó là nơi chứa dữ liệu đầu vào và đầu ra của chương trình. \+ **Entry point** - nơi chương trình bắt đầu chạy nằm

ở địa chỉ `0x000550F0`, thuộc section `.text` (đúng chuẩn). Đây được gọi là OEP (Original Entry Point). \+ **Entropy**: không có section nào có entropy vượt quá **6.7**, do đó file `putty.exe` không có section nào chứa packed data. 2 section có entropy cao nhất là `.text` và `.rdata`, do `.text` là nơi tập trung mã thực thi và `.rdata` dữ liệu chính có thể đọc được dùng để nạp vào khi chương trình chạy. 2 section có entropy thấp nhất là `.data` và `.rsc` vì chứa các dữ liệu không được sử dụng, được chủ yếu là các bit 0. - `putty_packed.exe:` \+ Chỉ có **3 section** là **UPX0**, **UPX1** và `.rsrc`, với **UPX0**, **UPX1** được sinh ra khi UPX nén file. \+ Khi so sánh **raw-size** và **virtual-size** ở từng section thì ta đã thấy có sự chênh lệch bất thường ở section **UPX0** khi raw size là 0 bytes còn virtual size thì rất lớn, lên đến 294912 bytes. Từ đó kết luận section UPX0 không tồn tại trên ổ đĩa, và sẽ được tự động lấp đầy khi chương trình chạy. \+ Về **phân quyền**, có tới 2 section vừa có quyền ghi vừa có quyền thực thi là UPX0 và UPX1, không tuân theo quy tắc bảo mật. Quyền ghi được cấp cho 2 section này có lẽ sẽ sử dụng cho việc khôi phục lại mã chương trình gốc ban đầu ở các section trống như UPX0. \+ **Entry point** nằm ở địa chỉ `0x000860E0` thuộc section UPX1. Đây chỉ là Entry Point của một đoạn stub trong UPX1. \+ **Entropy**: raw size của UPX0 là 0 bytes, nó không chứa một cái gì cả, do đó entropy của nó cũng không tồn tại.Section `.rsrc` thì có entropy vẫn ở mức bình thường, nhưng entropy của UPX1 lại lên đến `7.932`, lớn hơn **6.7**. **--> Giả thuyết đặt ra: UPX1 là nơi chứa packed data, còn UPX0 là không gian trống mà file `putty_packed.exe` dành ra nhằm phục vụ cho việc khôi phục lại mã chương trình gốc ban đầu. (1)** ## :rocket: III. Kiểm chứng giả thuyết, phân tích động và tách packed data ra khỏi file bị packed bằng công cụ OllyDbg > Ta sẽ "đào sâu" và kiểm chứng giả thuyết đặt ra ở mục II, từ đó tìm kiếm thêm các thông tin có ích cho việc phân tích động (bằng cách debug) và cuối cùng là tách packed data ra khỏi file bị packed. OllyDbg là một debugger sử dụng Assembly 32bit dành riêng cho Windows. Mặc dù hiện nay đã có rất nhiều tool hiện đại hơn như IDA (ta sẽ đề cập đến nó ở phần sau) hay x64dbg nhưng OllyDbg vẫn là một trong những debugger ra đời sớm nhất, không gây tình trạng bị "ngợp" vì quá nhiều tính năng như IDA. OllyDbg cũng hoàn toàn miễn phí và bạn có thể tải nó tại [đây](https://www.ollydbg.de/). ### 1. Kiểm chứng giả thuyết: Lưu ý luôn chạy OllyDbg với quyền admin. Sau đó ta sẽ mở file `putty.exe`, cùng lúc đó bật thêm một cửa sổ OllyDbg nữa rồi mở file `putty_packed.exe`. Ở cửa sổ mở file `putty_packed.exe`, nếu có pop-up **Compressed code?** hiện lên thì click **No** để giữ nguyên trạng file bị packed. Maximum của sổ "CPU" lên, ta thấy các tham số thu được trong quá trình phân tích tĩnh hoàn toàn trùng khớp với những gì thể hiện trên trình disassembler của OllyDbg: Trong đó, file `putty.exe` bắt đầu với lệnh `PUSH 60` ở địa chỉ `004550F0` còn với file `putty_packed.exe` thì nó là `PUSHAD` ở địa chỉ `004860E0`. Lệnh `PUSHAD` sẽ đẩy giá trị của tất cả các thanh ghi (general registers) vào stack. Điều đó thể hiện khi tại `PUSHAD` ta ấn **F7** (step into) để chạy lệnh này: Trên cửa sổ OllyDbg, click **View -> Memory** để kiểm tra các vùng nhớ trên ổ đĩa mà 2 file PE này chiếm dụng khi thực thi. Cả 2 chương trình khi chạy đều được nạp vào ổ đĩa tại địa chỉ gốc (base address) `400000` giống như mọi chương trình Windows 32 bit khác. Ngoài ra có thể thấy các section quen thuộc `.text`, `.rdata`, `.data`, `.rsrc` của file `putty.exe` và `UPX0`, `UPX1`, `.rsrc` của file

`putty_packed.exe` ở đây: Sử dụng **Pestudio** mới chỉ cho ta biết kích thước của từng section khi phân bố trên ổ đĩa như thế nào (raw size). OllyDbg còn cho ta biết bên trong từng section đó có cái gì. Click chuột phải vào section `.text` của `putty.exe` và `UPX0` của `putty_packed.exe` rồi chọn **Dump** để xem dữ liệu trong các section này. Trong khi `.text` chứa mã assembly của chương trình thì `UPX0` chỉ toàn là các bit `0`. Đồng thời khi sử dụng **Pestudio** để xem xét **entropy** của từng section, ta có đặt giả thuyết "section UPX1 là nơi chứa packed data" do entropy của section này lớn hơn **6.7**. Khi dump dữ liệu của section UPX1 ra như trên thì đúng là như vậy: Dữ liệu trong section UPX1 của `putty_packed.exe` bao gồm một đoạn stub có nội dung giống đến khoảng 60% mã assembly trong section `.text` của `putty.exe`, nhưng đoạn stub này hoặc là đã bị obfuscate, hoặc là bị chèn thêm một số đoạn dữ liệu "trông có vẻ như" là các lệnh assembly nhưng lại không phải vì OllyDbg không thể biên dịch được chúng, được đánh dấu là "Unknown command". Từ giá trị entropy thu được bằng **Pestudio** ta biết được đó chỉ là những dữ liệu được sinh ra ngẫu nhiên bởi packer. **--> Giả thuyết (1) đặt ra là đúng.** Không những vậy, tại dữ liệu được dump ra từ section `UPX1`, khi kéo chuột xuống dưới để tìm điểm kết thúc của section này thì thấy một loạt lệnh `ADD` như `ADD BYTE PTR DS:[EAX],AL`. Nhưng điều đáng quan tâm là trước loạt lệnh `ADD` này có lệnh `JMP putty_pa.004550F0`. Lệnh `JMP` sẽ bắt chương trình phải nhảy đến câu lệnh tại địa chỉ `004550F0`. Rõ ràng, ở đây sau khi thực thi toàn bộ mã lệnh tại Stub và tái tạo lại mã gốc ban đầu của chương trình, chương trình sẽ nhảy tới OEP tại địa chỉ `0x004550F0`. Địa chỉ này có một chút khác biệt so với OEP mà ta thấy khi dùng **Pestudio** để kiểm tra ở mục II, đó là do Entry Point `0x000550F0` hiển thị trong **Pestudio** là địa chỉ trong bộ nhớ ảo (virtual address), khi mã được nạp vào ổ đĩa tại địa chỉ gốc (base address) `0x00400000` thì địa chỉ thật (raw address) của nó sẽ là: `base address + virtual address = 0x00400000 + 0x000550F0 = 0x004550F0`. Suy cho cùng, tuy hai nhưng vẫn là một. **--> Vậy OEP của file `putty_packed.exe` là chính là `0x004550F0`** ### 2. Phân tích động (dynammic analysis): Như đã đề cập ở mục I, OEP chính là mấu chốt để lấy ra được packed data từ file PE đã bị packed. OEP thì đã có, để xem mã của chương trình gốc có được khôi phục sau lệnh `JMP` đến OEP như ta đã suy đoán ở cuối phần "Kiểm chứng giả thuyết" không, ta vào disassembler của OllyDbg rồi ấn **F2** đặt breakpoint tại lệnh `JMP putty_pa.004550F0`. Ấn **F9** (Run) để chương trình chạy đến lệnh này, lưu địa chỉ của lệnh `JMP putty_pa.004550F0` vào thanh ghi EIP. Tiếp tục ấn **F7** (Step into) để thực thi lệnh `JMP putty_pa.004550F0`. Chương trình sẽ nhảy tới lệnh `PUSH 60` tại OEP `0x004550F0`. Từ đây, khối mã này chứa các lệnh nằm trong section `.text` "thật", rất giống với code tại cùng địa chỉ ở file gốc `putty.exe`. **Hay nói cách khác, chúng ta đã tìm ra được vị trí của packed data.** ### 3. Tách packed data thành một file PE mới: Sau khi tìm ra packed data của file `putty_packed.exe` nằm ở đâu, ta có thể chạy nó bằng Debugger của OllyDbg. Nhưng để tách hoàn toàn nó ra và tạo thành một file PE mới, ta cần tập hợp các "mảnh binary" riêng lẻ lại với nhau đồng thời dựng lại PE header, vì UPX packer đã hủy mất import table ở trong đó. OllyDbg có một plugin cho phép chúng ta làm

được điều này đó là [OllyDump](http://www.openrce.org/downloads/details/108/OllyDump). Click **Plugins -> OllyDump -> Dump debugged process**. Pop-up **OllyDump** hiện ra, hiển thị các sections của file PE, Entry Point và phương pháp OllyDump sử dụng để xây dựng lại import table. Các thông số đều đúng với trạng thái của chương trình tại OEP nên ta để mặc định không chỉnh sửa gì, chỉ việc click vào **Dump**. Lưu file được dump ra với tên là `putty_unpacked.exe`. Kiểm tra file này bằng **Pestudio**, đối chiều với file gốc `putty.exe`: Ngoại trừ một section mới thêm là `.newIID` thì tên các section của `putty_unpacked.exe` vẫn giữ nguyên. Tuy nhiên các thông số của từng section đã thay đổi khi: - Section UPX0 không còn bị rỗng khi nạp vào ổ đĩa. - Chênh lệch giữa `raw-size` và `virtual-size` ở từng section hầu như đều bằng 0. - Không còn section nào có entropy lớn hơn 6.7. - Entry point được trả về đúng địa chỉ gốc ở `0x000550F0`. ## :rocket: IV. Kết hợp phân tích tĩnh và động sử dụng IDA Pro > OllyDbg chỉ dừng lại ở mức "một debugger tốt", nó cung cấp khả năng phân tích động mạnh mẽ nhưng khả năng phân tích tĩnh vẫn còn khá hạn chế. Reverse engineering yêu cầu phải mạnh ở cả 2 khả năng này do đó khi dùng OllyDbg ta phải bổ trợ thêm các công cụ phân tích tĩnh như Pestudio. Liệu có công cụ nào phân tích tĩnh hay động đều mạnh hay không? Đó chính là IDA Pro. IDA Pro là một công cụ reversing hoàn chỉnh, sử dụng được trên cả hai nền tảng 32-bit và 64-bit, vừa là một trình disassembler đồng thời cũng hỗ trợ debug như một trình debugger, hoạt động được trên các môi trường hệ điều hành khác nhau như Windows, Linux hoặc Mac OS X và thậm chí là remote (từ xa). Trong phần **IV** của bài viết, mình sẽ giải quyết bài toán đi tìm hàm main của một mẫu file khác cũng bị packed bởi UPX có tên là [upxflag2c1.exe](https://samsclass.info/126/proj/upxflag2c1.exe). Lần này sẽ không còn file gốc nào để đối chiếu nữa, ta sẽ không biết OEP của **upxflag2c1.exe** ở đâu và bắt buộc phải áp dụng các kĩ thuật dynamic analysis và static analysis trên IDA Pro để tìm ra hàm main của chương trình này. ### 1. Kiểm tra file thực thi là 32 bit hay 64 bit: IDA Pro có 2 bản, 1 bản để phân tích các file thực thi 32 bit và bản còn lại là 64 bit, do đó trước tiên chúng ta cần phải kiểm tra xem file `upxflag2c1.exe` này là thuộc loại nào. Ở đây ta sẽ dùng [HxD](https://mh-nexus.de/en/downloads.php?product=HxD) để kiểm tra: Các bạn có thể thấy chữ **PE..L** bên trong string của file này, do đó nó là file 32-bit. Từ thông tin đó, chúng ta biết sẽ phải mở `upxflag2c1.exe` bằng phiên bản 32-bit của IDA. Khi giao diện của IDA xuất hiện, chọn New để mở file mới, tìm đường dẫn tới `upxflag2c1.exe` để load nó vào IDA. Giữ nguyên các thiết lập mặc định và nhấn OK để IDA tiến hành phân tích file. ### 2. Phân tích tĩnh (Static analysis): Load file vào IDA và chọn **Manual load** vì tùy chọn này sẽ giúp chúng ta có thể **nạp tất các section** của file, đồng thời bỏ lựa chọn **Create imports segment** (chuyên gia Ricardo Narvaja khuyến nghị nên bỏ chọn khi làm việc với các packed files). Với các "Please enter an address" box hiện ra sau đó, chúng ta sẽ click "Ok" hết. Đây là lúc IDA nạp từng section của PE file. Click "Yes" cho tất cả các "Please confirm" box như thế này để nạp hết tất cả các section. Đây là giao diện của IDA sau khi đã load file xong: Tiếp theo ta đi tìm Entry Point của file

`upxflag2c1.exe`. Bắt đầu từ việc kiểm tra các section hay segment của file này bằng cách ấn **Shift F7** để mở ra tab **Program Segmentation**. Click vào segment UPX0 trước để xem code tại địa chỉ bắt đầu (`start=00401000`) của nó như thế nào. Để ý tại địa chỉ `0040162D` có dòng comment `CODE XREF: start+1BC↓j` (đọc thêm về XREF tại [đây] (https://reverseengineering.stackexchange.com/questions/18074/what-does-xref-mean)). Đây là một tham chiếu trong mã thực thi, chuột phải vào `start+1BC↓j` rồi chọn **Xrefs graph to ...** để xem xét câu lệnh này làm gì. Câu lệnh này đang tham chiếu đến entry point của file `upxflag2c1.exe`. ![] (https://i.imgur.com/LWJlRT0.png) Chuyển tới vùng code được tham chiếu đến và ta có lệnh `jmp near ptr byte_40162D` nằm ở entry point. Nhưng đây chỉ là **Entry point của Stub**, không phải OEP. Chúng ta cần debug chương trình bắt đầu từ lệnh `JMP` này để tìm ra hàm main. ### 3. Debug file PE bằng IDA: Ấn **F2** để đặt một breakpoint tại function `jmp near ptr byte_40162D` này. Sau khi có break point thì nhấn **F9** để debug. Lưu ý chọn Debugger là **Local Windows debugger** để chúng ta có thể debug trực tiếp trên localhost. Sau đó sẽ có pop-up **Debugger Warning** hiện lên thì ta chọn **Yes**. Thao tác debug trên IDA đồng nghĩa với việc mở và thực thi luôn file `.exe`. Nhưng khác với việc ta click vào và như thông thường file `.exe` sẽ chạy một mạch từ đầu đến điểm kết thúc của chương trình, debugger cho phép ta nhảy tới điểm nào thì file `.exe` sẽ chạy đến đó. F8 liên tục (Step Over) để trace qua tất cả các lệnh cho đến khi IDA sẽ hiển thị một thông báo như dưới, cứ nhấn **Yes** để thông báo cho IDA biết và nhận diện lại section **UPX0** ban đầu như là **CODE** (ban đầu nó được định nghĩa là *DATA*). Tiếp tục F8 liên tục cho đến khi IDA không cho chúng ta F8 nữa. Nó sẽ bị khựng lại khi nhảy đến một địa chỉ nào đó với lệnh `call sub_C1000`, đồng thời khi nhìn vào màn hình console đang chạy file `upxflag2c1.exe` lúc này thì thấy có in ra dòng chữ "Enter password:". Lệnh`call sub_C1000` sẽ gọi đến **hàm API `sub_C1000`**, rất có thể **hàm API này chính là hàm main của chương trình** nên khi debug đến đây chương trình mới in ra được "Enter password:" như thế. ![] (https://i.imgur.com/yIDPNoT.png) Đi vào hàm `sub_C1000` này để xem thì quả đúng là như vậy. IDA cho phép ta đọc mã giả của hàm main này bằng cách ấn phím **Tab**. Mã giả do IDA Pro biên dịch có syntax cực kì giống C, chỉ khác ở chỗ các tên hàm chưa được IDA biên dịch sẽ được để theo format là `sub_XXXXX` và nhiệm vụ của chúng ta là debug và đọc assembly để xem các hàm `sub_XXXXX` nó là hàm gì trong C. Đây là code C do mình biên dịch lại, đi kèm theo comment giải thích từng dòng code ở bên cạnh. ```c= int main() { char v1; // contains flag int i; // _BYTE v3[100]; // byte pointer char v4[12]; // contains inputted password char v5[8]; // this variable depends on v3 strcpy(v3, aW000111222asss); // copy value of aW000111222asss (a string) to pointer v3 for (i = 0; i < 5; ++i) // run a for loop from i=0 to 5 v5[i] = v3[10 * i]; // to append each element in the index "10*1" of the string aW000111222asss pointed by v3. v5[5] = 0; // assign the final element of string v5 with '\x00' to terminate it. printf('Enter Password:', 5); scanf(&unk_3581B4, (char)v4); // input password to v4 if(!strcmp(v4, v5)) // compare v4 and v5 printf("You didn't say the magic word! No flag

for you!", v1); else printf('Congratulations! The flag is', v1); // v4 and v5 is the same, print v1 containning. return 0; } ``` Tóm lại, password mà chúng ta đã cần tìm đang nằm ở biến `v5`. Biến này được tạo ra bằng cách iterate và append từng phần từ của string `v3` với index có công thức là `10*i`. Mà `v3` này là pointer trỏ đến constant `aW000111222asss`. Và `aW000111222asss` nó nằm ở đây (click thẳng vào nó để xem): Chúng ta chỉ cần demo lại thuật toán để tạo ra `v5` rồi chạy là lấy được password. Ở đây mình sử dụng python: Password là `WALDO`. Giờ chỉ cần bật file `upxflag2c1` lên rồi nhập vào là lấy được flag. **Flag:** `CORRUPTED_FILE_CONFUSES_UPX`