BÁO CÁO THỰC HÀNH

**Môn học: Cơ chế hoạt động mã độc**

**Tên chủ đề: Infect PE file**

*GVHD: Phan Thế Duy*

*Ngày thực hiện: 26/06/2020 – 7/7/2020*

1. **THÔNG TIN CHUNG:**

Lớp: NT330.K21.ANTN.1

|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **Họ và tên** | **MSSV** | **Email** |
| 1 | Lê Ngọc Huy | 17520074 | 17520074@gm.uit.edu.vn |
| 2 | An Văn Hiếu | 17520467 | 17520467@gm.uit.edu.vn |

1. **NỘI DUNG THỰC HIỆN:[[1]](#footnote-1)**

|  |  |  |
| --- | --- | --- |
| **STT** | **Công việc** | **Kết quả tự đánh giá** |
| 1 | Tiêm shellcode | Tốt |
| 2 | Anti VM | Tốt |
| 3 | Anti Debug | Tốt |
| 4 | Packer | Tốt |

BÁO CÁO CHI TIẾT

# Yêu cầu.

Viết 01 chương trình virus đơn giản cho phép lây nhiễm các tập tin thực thi PE bất kỳ, có chức năng:

1. Chèn shellcode hiển thị thông điệp trong Cửa số cảnh báo trên Windows (message Box): “Infected by MSSV1-MSSV2-MSSV3”, trong đó MSSV là thông tin về thành viên nhóm.
2. Có chức năng phát hiện khi chạy trong máy ảo (ví dụ: VMware/VirtualBox): nếu chạy trên môi trường máy ảo thì không hiển thị thông điệp trên.
3. Có chức năng phát hiện trình gỡ lỗi (Debugger) dùng trong phân tích mã độc (ví dụ: OllyDbg, IDA Pro,...) nếu chương trình được mở ở chế độ debug.
4. Áp dụng Packer/Decryptor (có sẵn/hoặc tự viết) cho việc chèn shellcode khi file PE bị lây nhiễm.

# Ý tưởng và thiết kế

## Ý tưởng

### Shellcode

Đoạn code tấn công được viết trong 1 hàm duy nhất trên ngôn ngữ C và sử dụng toàn bộ nội dung của hàm sau khi được biên dịch làm shellcode.

Sau khi hoàn thành đoạn code, ta biên dịch và sử dụng các trình debug (ở đây nhóm sử dụng gdb) để lấy thông tin các vị trí quan trọng cần phải sửa đổi, lưu lại và sửa đổi nó giá trị tại các vị trí đó trước mỗi lần chèn vào PEFile.

Đoạn code sử dụng các hàm trong các DLL có sẵn trong file PE (chẳng hạn MessageBox), do đó cần phải đọc được toàn bộ các hàm được load trong file.

### Packer và Unpacker

Unpacker cũng được xem như là một đoạn code tấn công và quá trình sửa đổi, chèn nó vào file PE cũng giống như shellcode.

Packer sử dụng mã hóa xor để mã hóa đoạn shellcode cần tấn công và chèn nó vào file như một loại dữ liệu.

Để có thể chạy đoạn shellcode đã mã hóa, unpacker được chèn vào shellcode sẽ cấp phát một vùng nhớ, sau đó đoạn code đã giải mã sẽ được đặt trên vùng nhớ này. Vùng nhớ cũng sẽ được thiết lập để có thể thực thi code và một lời gọi hàm sẽ gọi tới vùng nhớ này.

Trước khi chạy đoạn code, unpacker cần phải tự động sửa đổi một số giá trị (chẳng hạn địa chỉ của hàm).

## Thiết kế tổng quan

Chương trình cần có các chức năng cơ bản sau:

* Đọc và phân tích các header của PE File bao gồm DOS header, NT header (File header và Optional header) và các Section header.
* Tạo thêm section, mở rộng section và tìm codecave còn trống cho mục đích thêm shellcode vào những vị trí đó.

Đối với yêu cầu a, b và c ta cần thực hiện:

* Tạo shellcode.
* Điều chỉnh shellcode để phù hợp với từng file

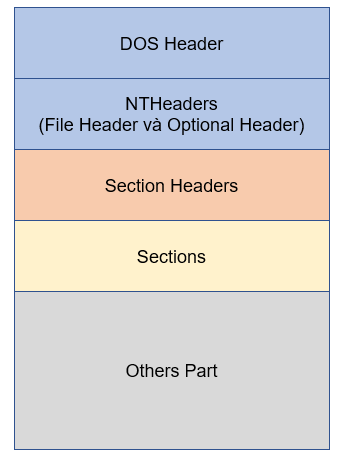
Đối với yêu cầu d, ta cần thực hiện:

* Thiết kế hàm mã hóa và giải mã.
* Giải mã code đã mã hóa và đặt nó lên vùng nhớ có thể thực thi để chạy chương trình.

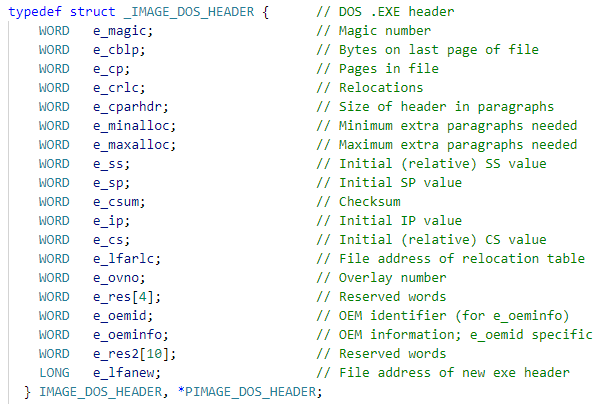
## Ý tưởng chi tiết

### Đọc và phân tích header file PE

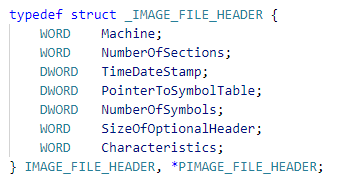
Cấu tạo File PE gồm các Header, các Section và các thành phần khác của file. Trong đó Header và Section là các thành phần được load lên RAM để thực thi chương trình.



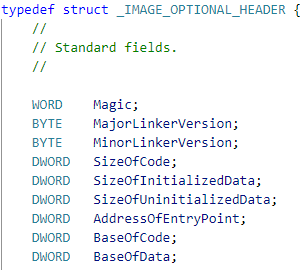
1. Cấu trúc file PE



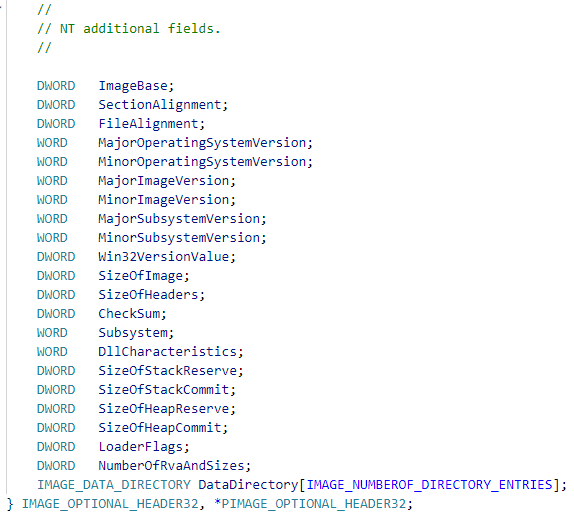
1. Cấu trúc của DOS Header.



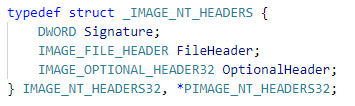
1. Cấu trúc File Header.



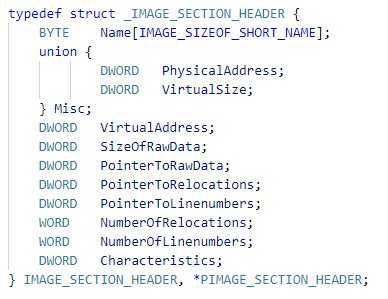
1. Cấu trúc phần đầu của Optional Header (32-bit PE file).



1. Cấu trúc phần sau của Optional Header (32-bit PE file).



1. Cấu trúc NT Headers



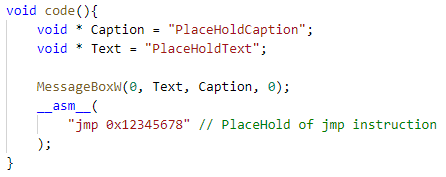
1. Cấu trúc của MỘT section header

Lưu ý:

* Vị trí bắt đầu của NT Headers trong file được chỉ định bởi trường e\_lfanew trong DOS Header.
* Vị trí bắt đầu của các Section được chỉ định bởi trường SizeOfHeader trong Optional Header.
* Nội dung trong bài tập trung vào các file PE 32-bit.

### Viết shellcode

Để thu được shellcode, ta đơn giản là viết 1 hàm trong C. Sau đó sử dụng memcpy để lấy vùng nhớ của hàm đó, đó chính là shellcode mà ta cần. Lưu ý shellcode chỉ nên viết trong 1 hàm duy nhất.



1. Shellcode đơn giản

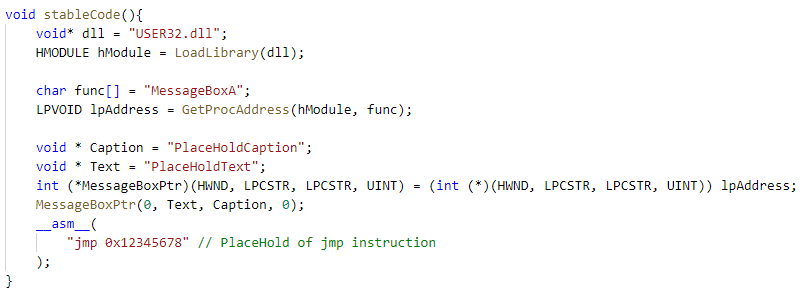
Shellcode trên sẽ được tự động chỉnh sửa ở 4 vị trí:

* Giá trị của Caption.
* Giá trị của Text.
* Địa chỉ MessageBox.
* Lệnh jmp.

Vì MessageBox có 2 phiên bản là A (sử dụng đầu vào là chuỗi 1 byte) và W (sử dụng đầu vào là chuỗi 2 bytes) nên ta không thể gán trực tiếp chuỗi cho Caption và Text mà phải thay thế bằng địa chỉ trỏ đến chuỗi, sau đó tùy vào hàm MessageBox loại nào mà ta thiết lập chuỗi loại ấy.

Địa chỉ hàm MessageBox trong mỗi file PE là khác nhau. Do đó, để dễ dàng trong việc tính toán, ta thay thế hàm call đến địa chỉ tương đối thành call đến địa chỉ tuyệt đối.

Trong trường hợp file PE không load file USER32.dll thì ta không thể tìm được địa chỉ hàm MessageBox. Do đó, một shellcode phiên bản cải tiến là sử dụng 2 hàm LoadLibrary và GetProcAddress để load file dll và lấy địa chỉ của hàm cần tìm. Các hàm này nằm trong file KERNEL32.dll và thường được load vào các file PE hơn là hàm MessageBox.



1. Phiên bản code cải tiến, chạy được trong bất kỳ chương trình nào

Giống MessageBox, LoadLibrary cũng có nhiều phiên bản, trong đó có A và W, do đó ta không gán trực tiếp chuỗi mà thay vào đó sẽ tùy cơ ứng biến với mỗi file PE.

Hàm GetProcAddress chỉ có một phiên bản duy nhất nên ta có thể gán trực tiếp tên hàm cần tìm vào biến func.

Lưu ý:

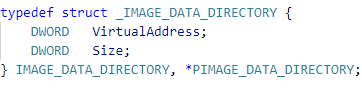
* Đối với các biến dll, Caption và Text, các biến này được khởi tạo bằng cách tạo ra một chuỗi nằm đâu đó trong file PE. Sau khi tạo ra chuỗi, địa chỉ của nó được gán cho biến. Do đó trong trường hợp muốn thay đổi chuỗi một cách dễ dàng, ta thực hiện cách khai báo như các biến này, sau đó đặt các chuỗi vào cùng với shellcode và trỏ giá trị các biến tới các địa chỉ này.
* Đối với biến func, các khởi tạo này sẽ tạo thành một mảng nhiều phần tử char và đặt ngay trên stack của chương trình, vì stack được xác định bởi thanh ghi %ebp nên ta không cần quan tâm đến việc bị sai địa chỉ khi tiêm shellcode vào file PE khác.
* Mặc dù có nhiều phiên bản MessageBox, nhưng để đơn giản, trong shellcode này ta sẽ xác định luôn sẽ sử dụng MessgeBox phiên bản A. Ta cũng có thể chạy lại chương trình bằng cách thay A bằng W nếu chèn code không thành công (chưa xảy ra trong quá trình thí nghiệm). Tôi cũng đã viết một tool đơn giản để chèn tự động, trong đó cho phép thay đổi phiên bản MessageBox (được giới thiệu trong phần 4 - thực hiện).

### Đọc Import Table và tìm địa chỉ các hàm trong DLL

Vì mỗi hàm trong shellcode (chẳng hạn LoadLibrary, MessageBox, GetProcAddress) có địa chỉ khác nhau đối với các file PE khác nhau, nên ta cần xác định được địa chỉ các hàm này trong các file PE để có thể thay đổi shellcode trước khi lây nhiễm.

Địa chỉ của các hàm trong dll được đặt trong Import table (một trong các Data Directory). Header của Import table là header thứ 2 trong 16 header Data Directory, nằm ở cuối Optional Header.

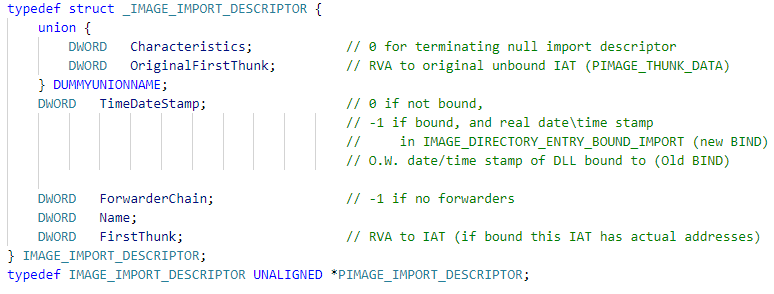
Header của Import table nói riêng và các Data Directory khác nói chung đều có 8 bytes, chứa 2 thông tin là Virtual Address và Size của Table.



1. Cấu trúc header của các Data Directory

Dựa vào 2 thông tin này, ta có thể tìm được nơi lưu trữ Import Table.

Import table có dạng là một mảng các cấu trúc Import Descriptor, mỗi cấu trúc là 20 bytes, kết thúc bằng một cấu trúc có tất cả các byte đều bằng 0.

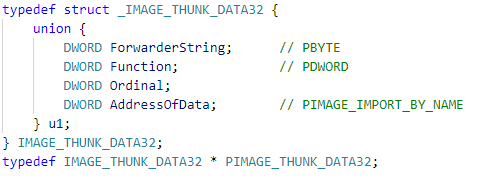


1. Cấu trúc của Import Descriptor

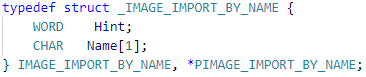
Các Import Descriptor mô tả thông tin một file dll được load vào PE và có 3 trường quan trọng là OriginalFirstThunk (còn có tên là Characteristics), First Thunk và Name:

* Name: Lưu trữ địa chỉ chứa tên của file dll.
* OriginalFirstThunk: lưu trữ địa chỉ của một mảng các Thunk.
* FirstThunk: Nếu OriginalFirstThunk có giá trị bằng 0, trường này sẽ lưu địa chỉ của một mảng các Thunk, đồng thời trường này còn là địa chỉ của hàm đầu tiên.

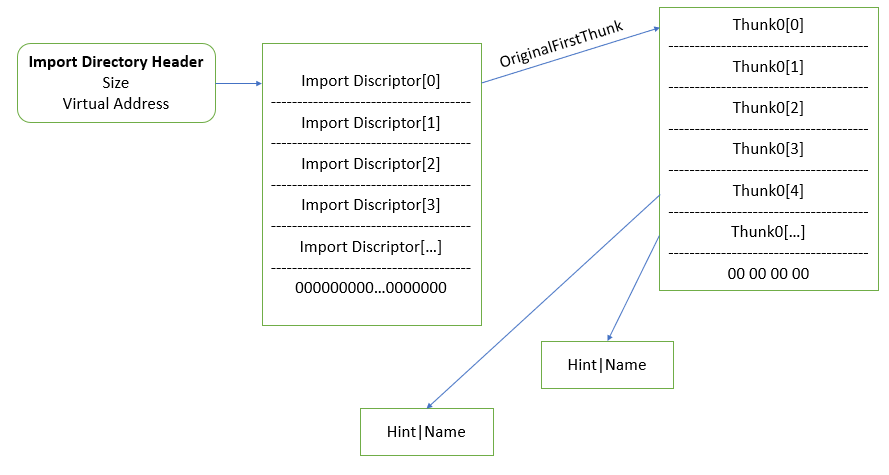
Thunk là một cấu trúc có 4 bytes chứa thông tin để xác định một hàm trong file dll. Nếu bytes cao nhất của cấu trúc này là 0x80 (tức giá trị cấu trúc này có dạng 0x80zzzzzz) thì thông tin xác định của hàm này là Ordinal (4 bytes thấp nhất chứa số định danh của hàm), nếu không, hàm này sẽ được xác định bằng tên, và địa chỉ của chuỗi chứa tên hàm chính là giá trị của trường này cộng thêm 2.



1. Cấu trúc của một Thunk. Một biến 4 bytes được chia sẻ bởi 4 tên khác nhau.



1. Cấu trúc được trỏ đến bởi Thunk nếu hàm được xác định bằng tên. Trường Hint có giá trị 2 bytes ngay trước trường Name là lý do ta phải cộng 2 như được nêu ở trên.



1. Cấu trúc tổng quát của Import Table nếu hàm xác định bởi Name

Đối với trường FirstThunk, trường này là địa chỉ của hàm được xác định bởi Thunk đầu tiên. Và các địa chỉ các hàm kế tiếp được xác định bằng cách cộng thêm với 4.

Ví dụ:

* FirstThunk có giá trị là 0x124A.
* Địa chỉ của hàm được xác định bởi Thunk đầu tiên là 0x124A.
* Địa chỉ của hàm được xác định bởi Thunk thứ hai là 0x124E.
* Địa chỉ của hàm được xác định bởi Thunk thứ ba là 0x1252.
* …

Dựa trên cấu trúc này, ta có thể tìm được tên các file dll được load vào file PE và địa chỉ của các hàm trong file dll tương ứng, từ đó xác định được địa chỉ các hàm cần tìm.

### Tìm vị trí đặt shellcode

Để có thể có nơi đặt được shellcode, ta có 3 cách:

* Thêm section: Tạo một vùng section mới ngay đằng sau section cuối cùng, thay đổi các thông tin như NumberOfSections (trong File Header), SizeOfImage (trong Optional Header), …. Ta cũng cần thêm một section header, có 5 thông tin quan trong của một section header là VirtualAddress, VirtualSize, RawAddress, RawSize và Chracteristics. Để một section có thể chạy được chương trình, Characteristics cần được bật cờ cho phép MEM\_EXCUTE, READ, WRITE, CODE và một số cờ khác.
* Mở rộng section cuối: Ta cũng có thể mở rộng section cuối cùng để thêm shellcode, các thông tin cần thay đổi cũng gần giống với thêm section, ngoại trừ NumberOfSections.
* Tìm vùng trống (codecave) có sẵn: Các phương pháp trên đều khiến shellcode bị tăng kích thước, nếu shellcode đủ nhỏ, ta có thể tìm một số vùng trống có sẵn trong file để chèn shellcode. Hơn nữa, trong một số trường hợp cuỗi PE file có một lượng dữ liệu của file (chẳng hạn ở các file cài đặt …) thì việc thêm hay mở rộng section có thể dẫn đến lỗi khi chạy file.

### Unpacker

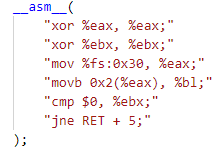
Một điểm lưu ý để viết unpacker là để giải mã và thực thi được vùng code đã bị mã hóa, ta cần yêu cầu hệ điều hành cấp phát một vùng nhớ trong heap CÓ THỂ THỰC THI. Để làm được việc này, hai hàm VirtualAlloc (cấp phát vùng nhớ) và VirtualProctect (thay đổi chức năng của một vùng nhớ) sẽ được sử dụng.

Lưu ý khác là vì vùng nhớ được cấp phát có thể ở bất kỳ địa chỉ nào nên unpacker cũng phải thực hiện nhiệm vụ thay đổi các giá trị đó cho phù hợp trước khi chạy.

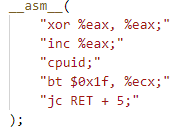
### Packer

Packer có nhiệm vụ chèn cả shellcode và unpacker vào file PE. Nó cũng phải thay đổi các thông tin trên Unpacker để phù hợp trước khi chèn.

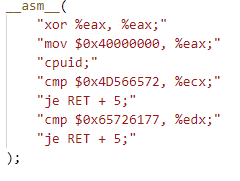
### AntiVM và AntiDebug



1. Kỹ thuật AntiDebug sử dụng PEB



1. Kỹ thuật Anti VM sử dụng CPUID



1. Kỹ thuật Anti VM bằng cách kiểm tra branh

### Một số lưu ý khác

Base relocation là một table được sử dụng để yêu cầu hệ điều hành cấp một vùng nhớ có địa chỉ ngẫu nhiên (thành vì Virtual Address) và giúp chương trình chỉ định các giá trị trong PE file cần được thay đổi lại khi load.

Nếu một file PE có table này, ta không thể chạy được shellcode do tất cả các địa chỉ (bao gồm cả địa chỉ hàm, chuỗi,…) đều đã bị thay đổi.

Theo lý thuyết, để chương trình hoạt động bình thường khi có Base Relocation Table, ta cần thêm các giá trị cần chỉ định được thay đổi vào vùng nhớ này. Tuy nhiên trong thực nghiệm, nhóm không thể làm được.

Tuy nhiên, một giải pháp khác được đặt ra là phá hỏng (xóa bỏ) table này để chương trình không được cấp một vùng nhớ ngẫu nhiên, từ đó giữ nguyên được toàn bộ các giá trị, địa chỉ nguyên dạng trong chương trình. Ngoài ra, việc xóa bỏ table này cũng giúp ta tạo thêm một khoảng trống lớn để chèn shellcode.

# Thực hiện

## Cấu trúc tệp tin

|  |  |
| --- | --- |
| Nhiệm vụ | File |
| Đọc, trích xuất thông tin, thay đổi, sửa đổi và các tác vụ khác trong file PE | PEFile.h/PEFile.c |
| Shellcode phiên bản đơn giản và phiên bản ổn định | code.h/code.c |
| Chương trình/công cụ chèn shellcode phiên bản đơn giản | infectMessageBox.c |
| Chương trình/công cụ chèn shellcode phiên bản ổn định | stableInfectMessageBox.c |
| Chương trình/công cụ packer | packer.c |
| Một số hàm hỗ trợ tiện ích | utils.h/utils.c |

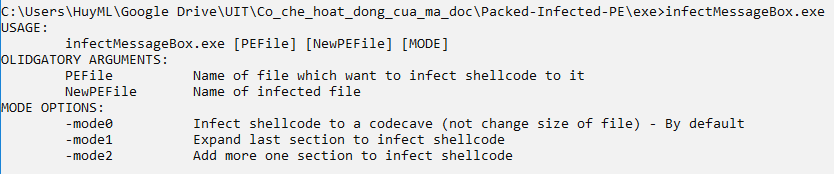
## Hướng dẫn biên dịch

|  |  |
| --- | --- |
| Chương trình | Lệnh |
| infectMessageBox | $ gcc infectMessageBox.c code.c utils.c PEFile.c -o infectMessageBox.exe |
| stableInfectMessageBox | $ gcc stableInfectMessageBox.c code.c utils.c PEFile.c -o stableInfectMessageBox.exe |
| packer | $ gcc packer.c code.c utils.c PEFile.c -o packer.exe |

Lưu ý: Nên sử dụng MinGW (mặc dù một số MinGW cũng biên dịch bị lỗi, đa phần lỗi do chuyển từ code C sang assembly không đồng nhất giữa các phiên bản).

## Hướng dẫn chạy

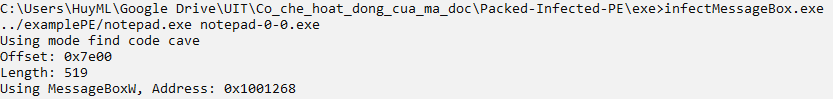
Chạy chương trình mà không sử dụng tham số để xem hướng dẫn. Lưu ý, các chương trình có hướng dẫn đôi chút khác nhau.



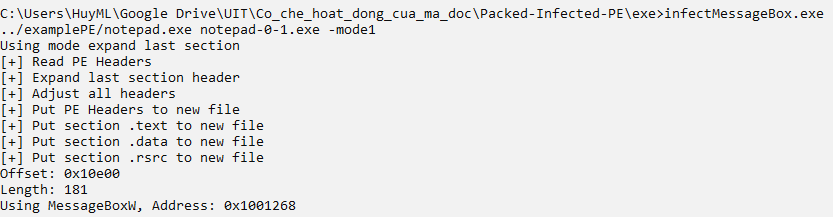
1. Ví dụ để xem hướng dẫn chạy của chương trình infectMessageBox

## Demo

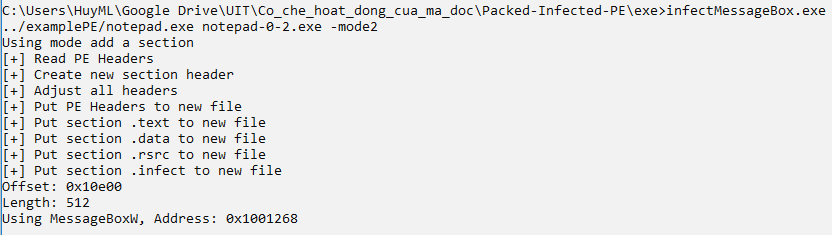
Chương trình infectMessageBox chèn shellcode đơn giản, có chức năng tự động tìm kiếm phiên bản MessageBox thích hợp có sẵn trong file PE và tạo chuỗi tương ứng. Hàm này sẽ thất bại nếu không tồn tại hàm MessageBox trong file PE (thư viện USER32.dll) không được load.



1. Chạy file infectMessageBox.exe với 2 tham số mặc định

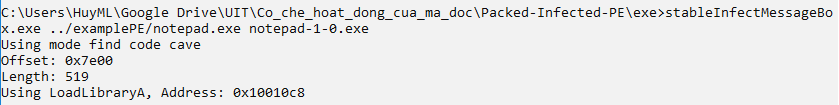


1. Chạy infectMessageBox.exe với mode 1 (mở rộng section)

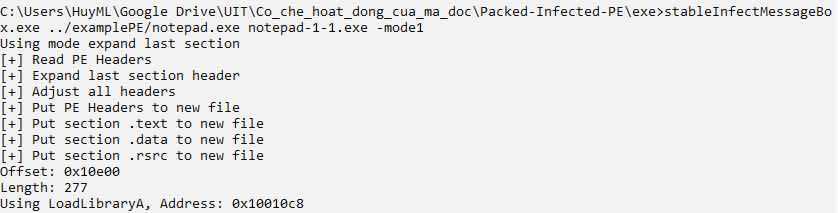


1. Chạy infectMessageBox.exe với mode 2 (thêm section)

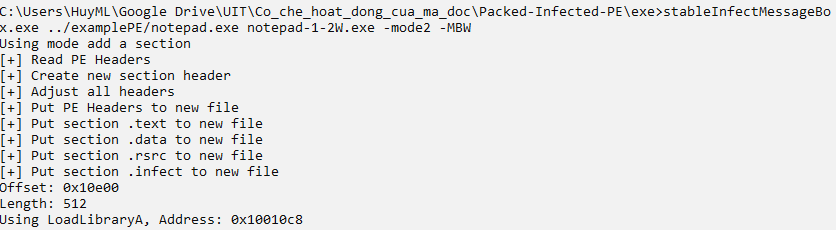
Chương trình stableInfectMessageBox chèn shellcode phiên bản ổn định, có khả năng hoạt động trên bất kỳ file nào dù không có sẵn hàm MessageBox.



1. Chạy stableInfectMessageBox với 2 tham số mặc định

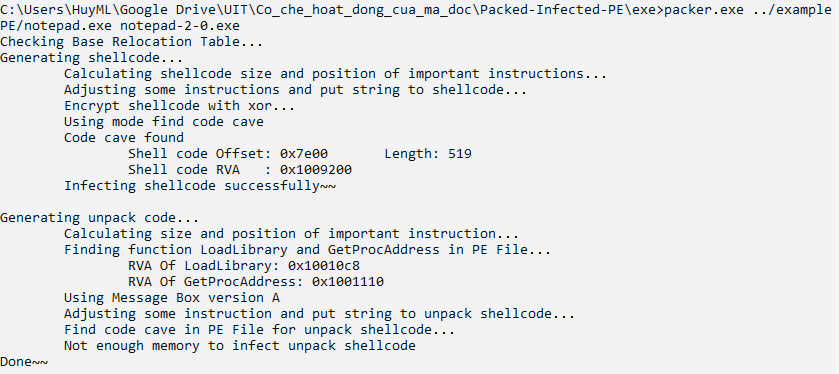


1. Chạy stableInfectMessageBox với mode 1(mở rộng section).

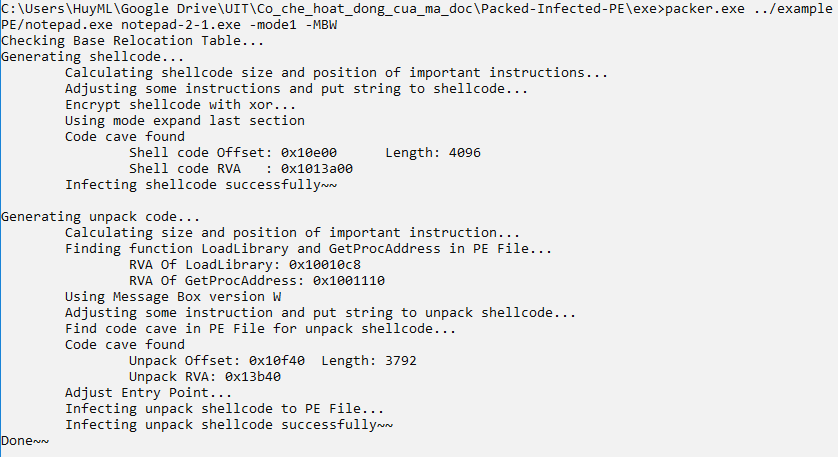


1. Chạy stableInfectMessageBox với mode 2 (thêm section) và sử dụng hàm MessageBoxW.

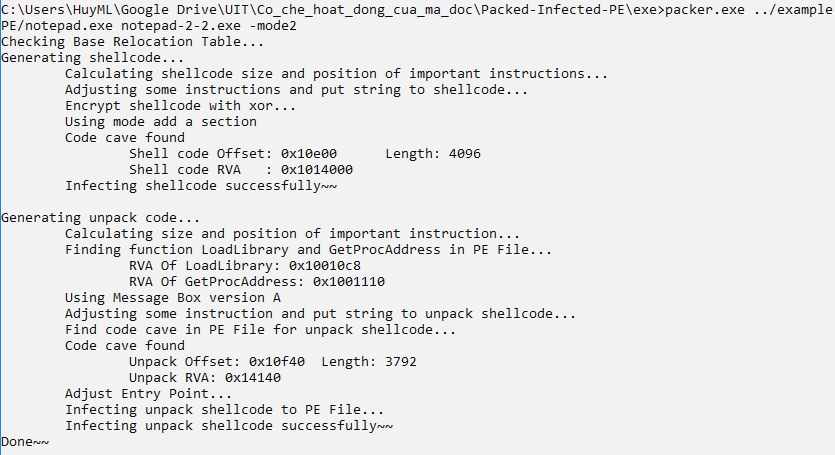
Chương trình packer sử dụng shellcode phiển bản ổn định. Có thêm chức năng anti-VM và anti-debugger, các chức năng trên được chèn trước khi đi vào packer.



1. Chạy packer với 2 tham số mặc định (thất bại – không đủ vùng nhớ để chèn unpacker).



1. Chạy packer với mode 1 (mở rộng section) và chỉ định phiên bản MessageBoxW.



1. Chạy packer với mode 2 (thêm section).

## Thành phần bài nộp

Chạy thử chương trình mà không cần biên dịch: folder ./exe

Các file PE dùng để chạy thử: folder ./examplePE

Các file PE đã chèn shellcode: folder ./exampleInfectedPE

Source code: folder ./source

Github: https://github.com/huykingsofm/Packed-Infected-PE

1. [↑](#footnote-ref-1)