**BẢN BÁO CÁO CỦA NHÓM: 9**

Giáo viên bộ môn: Phan Thế Duy.

**Thành viên của nhóm:**

* Trần Quang Kha 18520875.
* Châu Kim Lộc 18521003.
* Huỳnh Minh Hoàng 18520875.

1. **Phát triển tính năng trình gỡ lỗi (debugger) và phát hiện máy ảo (VMWare/VirtualBox/…) cho mã độc (Virus) đơn giản đã thực hiện (hiển thị thông điệp MSSV) (mỗi loại áp dụng 2 phương thức khác nhau).**

* 2 kỹ thuật Anti-VM và Anti-Debug được ứng dụng để nhằm chống lại sự phát hiện và điều tra của các malware analysis. Nó đi đôi với nhau và đồng thời thực hiện theo tuần tự: Anti-Debug -> Anti-VM.

Ví dụ:

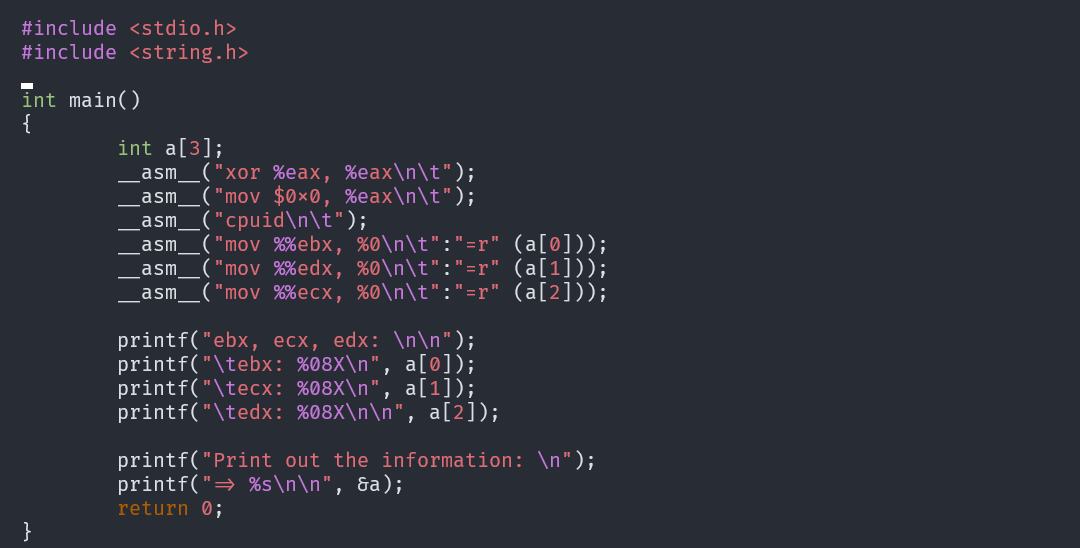
* + 1 Malware được Anti-Debug bằng cách được Hacker packed lại thông qua Packer (dùng Tool hoặc tự dùng 1 algorithm như XOR,…) để mà che giấu đi PE Malware thực sự đồng thời sẽ có những Chunks of garbage data để obfusticate (dữ liệu thừa có thể là mov %eax, random) mục đích là để gây rối cho người phân tích. Sau khi Malware được chạy thì Code của Malware thật sự sẽ được Unpacked ra và khi chạy tới dòng Code kiểm tra Virtual Machine sẽ tự động thực hiện tắt chương trình, hoặc đánh văng cả Virtual Machine.

**ĐỐI VỚI KỸ THUẬT ANTI-VM HIỆN ĐẠI:**

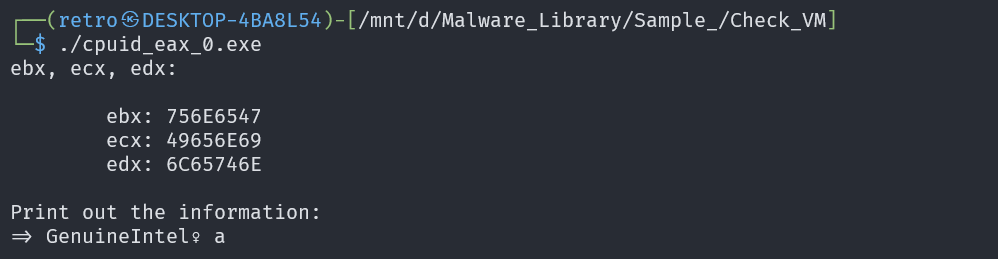
* Đối với kỹ thuật hiện đại để chống lại ANTI-VM. Các Malware Analysis thường dùng instruction CPUID để khai thác thông tin CPU của nhà sản xuất.
* Bằng cách khai báo thanh ghi %eax cho một số nhất định như là 0, CPUID sẽ dựa theo %eax để khai thác thông tin và truyền cho %ebx, %ecx và %edx.

**KHI %EAX = 0:**

* CPUID sẽ trả lại thông tin về nhà sản xuất CPU. Thông qua đọan Code sau:

****

* Sau khi biên dịch qua và chạy chương trình sẽ ouput như sau:

****

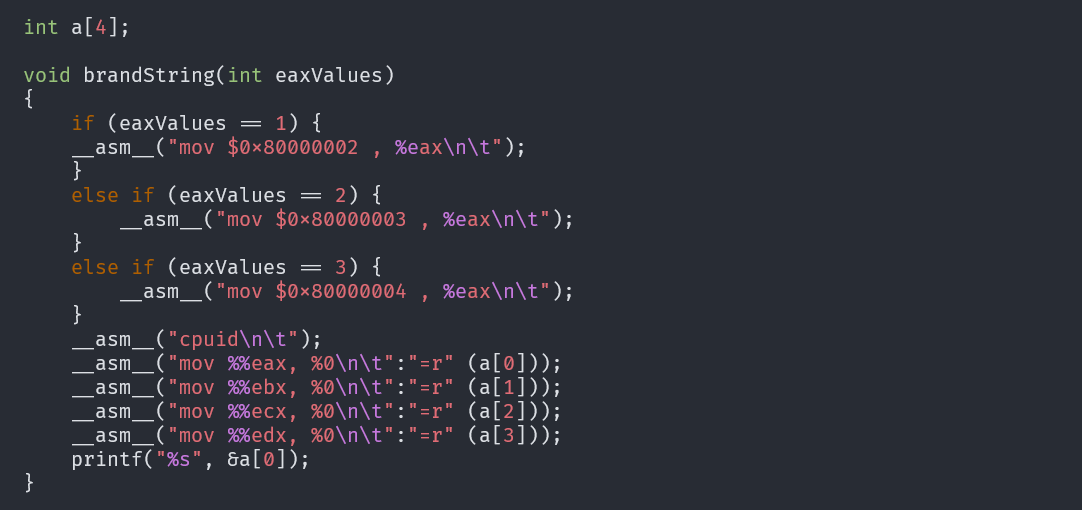
* Kết quả cho thấy hiện tại CPU của em được tạo ra bởi nhà sản xuất Intel.

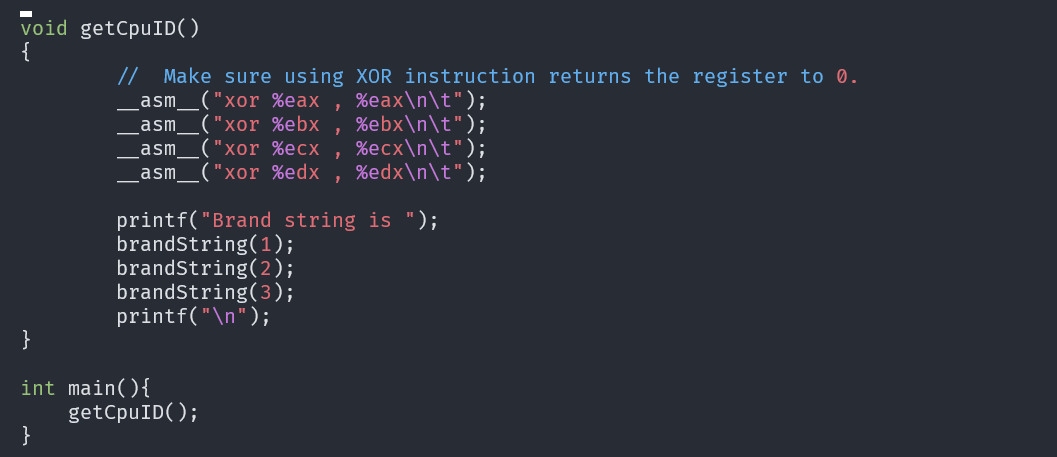
**KHI %EAX = 1:**

* Sử dụng CPUID với %eax=1 xuất ra các thông tin về mẫu và tính năng của Bộ Vi Xử Lý.

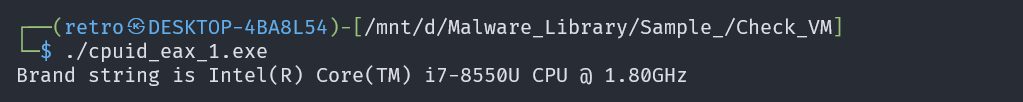
**LẤY THÔNG TIN CỦA CPU:**

* %eax lần lượt gán với giá trị 0x80000002,3,4 để lần theo xem CPU có đúng với giá trị ứng %eax đó không.

****

****

* Biên dịch ra và xuất ra ngoài màn hình thông tin như sau:

****

**KHI %EAX=2:**

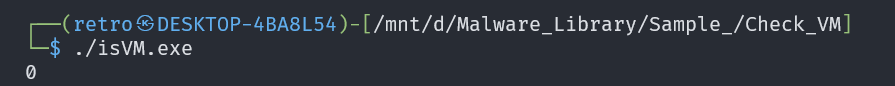
* Khi %eax bằng 2 sẽ xuất ra Cache của CPU.
* **Dựa vào đó ta có thể kiểm tra được PE file chạy trên VM hay không.**

**DÙNG CPUID ĐỂ CHECK VM.**

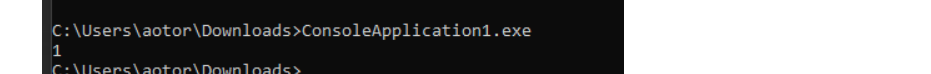
* Thực hiện dòng Code sau để kiểm tra Virtual Machine.

****

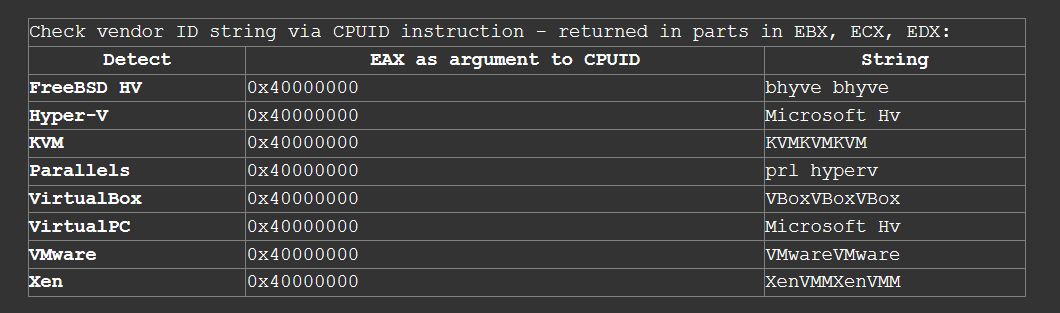
* **Giải thích:**
  + Khai báo biến vb với giá trị false.
  + Xor %eax để thanh ghi trở về 0.
  + Gán %eax với 0x40000000.
  + Thực hiện gọi lệnh CPUID.
  + So sánh nó với 0x4D566572 với %ecx và 0x65726177 với %edx. Nếu không bằng thì không chạy trên VM. Nếu có thì sẽ gán với 1.
* Em thực hiện biên dịch và chạy nó trên máy thật.

****

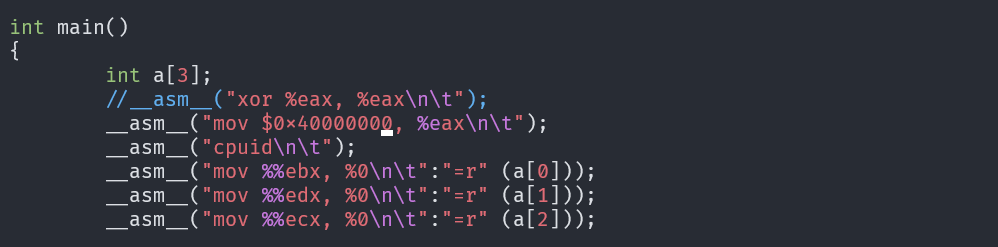
* Không phải là VM.
* Thực hiện chạy trên VM (yêu cầu VM phải có cài Visual Studio).



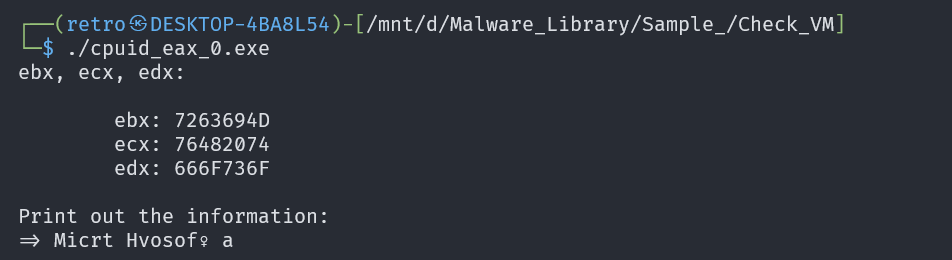
* Là VM.
* Để giải thích lý do vì sao %eax phải được gán với 0x40000000. %eax được gán với 0x40000000 sẽ thực hiện Check Hypervisor của từng hệ điều hành.



* Thực hiện đoạn Code như sau để kiểm tra thông tin.



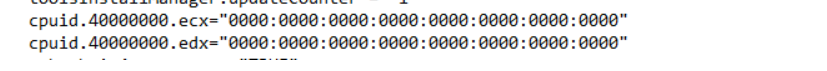
* Thực hiện biên dịch.



* **Là Microsoft Hypervisor.**

**CPUID VÀ %EAX=1:**

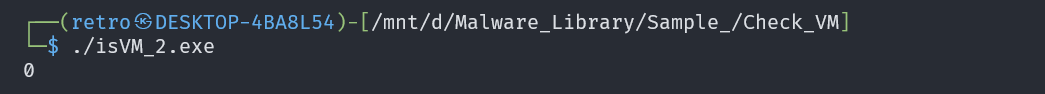
* Mục đích của %eax=1 là để lỡ như VirtualMachine Sandbox đó đã được setting và thay đổi %ecx, %edx thành như sau:

****

* %eax = 400000000 thất bại.
* Thực hiện đoạn code như sau:

****

* **Giải thích:**
  + Xor %ecx để thanh ghi trở về lại 0.
  + Inc tăng %eax lên 1, hoặc thay thế nó bằng lệnh:
    - Xor %eax, %eax.
    - Mov $0x1, %eax.
  + Gọi CPUID.
  + Lệnh bt thực hiện bit test kiểm tra bit thứ 31 được set hay là không (là kiểm tra nó là số 1 hay là 0).
  + Nếu là 1 return True, 0 return false.
* Thực hiện biên dịch và xuất ra ngoài Terminal.

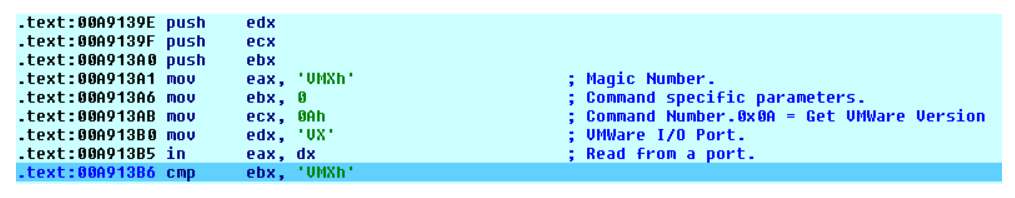
****

* **2 PHƯƠNG PHÁP DÙNG CPUID ĐỂ KIỂM TRA VM.**

**ĐỐI VỚI KỸ THUẬT ANTI-VM CỔ ĐIỂN:**

* Đối với kỹ thuật này không bị rào cản bởi Vistual Studio trên máy ảo thông thường sẽ mất đi các thư viện như VMSCP140D.dll,…
* Sử dụng được cả trên Windows XP, Windows 7, và ngay cả version Windows hiện tại.
* Hacker dùng phương pháp **VMWare Backdoor**, phương pháp thực hiện PORT I/O Stream để thực hiện lệnh và xuất ra giá trị.
* Phương pháp này sẽ sử dụng Magic Number là “5658h” được dịch ra ASCII là “VMXh”.

**DỰA THEO CẤU TRÚC SAU:**

****

* **Giải thích:**
  + **%**eax thực hiện lấy Magic Number (5658h).
  + **%**ebx, %ecx lấy phiên bản VMWare.
  + Thực hiện gọi lệnh VMXh thông qua lệnh in. Và trả lại nó %eax.
* Sau khi thực hiện %eax có kết quả như sau:

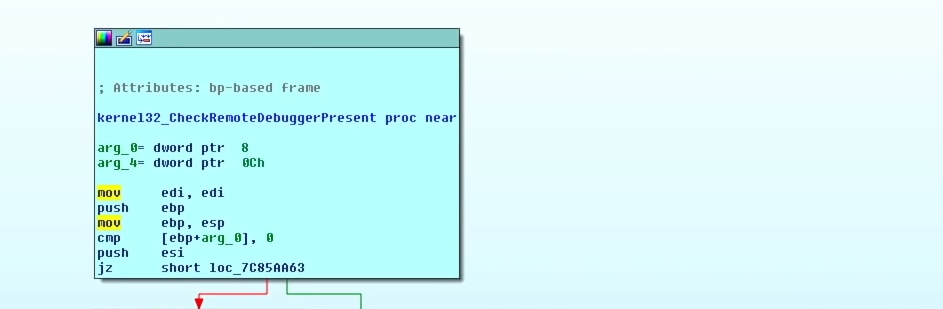
****

* **ĐÂY LÀ CÁCH PHỔ BIẾN VÀ DỄ NHẤT ĐỐI VỚI KỸ THUẬT ANTI-VM CỔ ĐIỂN.**

**KỸ THUẬT ANTI-DEBUG.**

* Đối với kỹ thuật Anti-Debug. Có 3 APIs được sử dụng phổ biến nhât.

**CheckRemoteDebuggerPresent:**

****

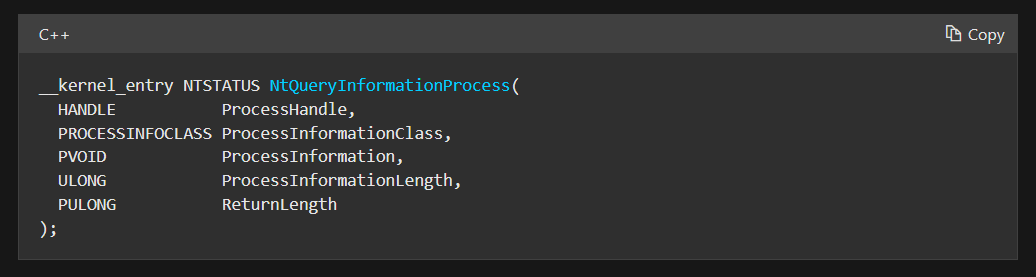
* API này sẽ kiểm tra process có được Debug hay không, nếu có trả về là 1, không sẽ trả về là 0.

**IsDebuggerPresent:**

****

* Tương tự như CheckRemoteDebuggerPresent sẽ kiểm tra process có bị Debug hay không sau đó sẽ trả lại 0 hoặc.

**NTQueryInformation:**

****

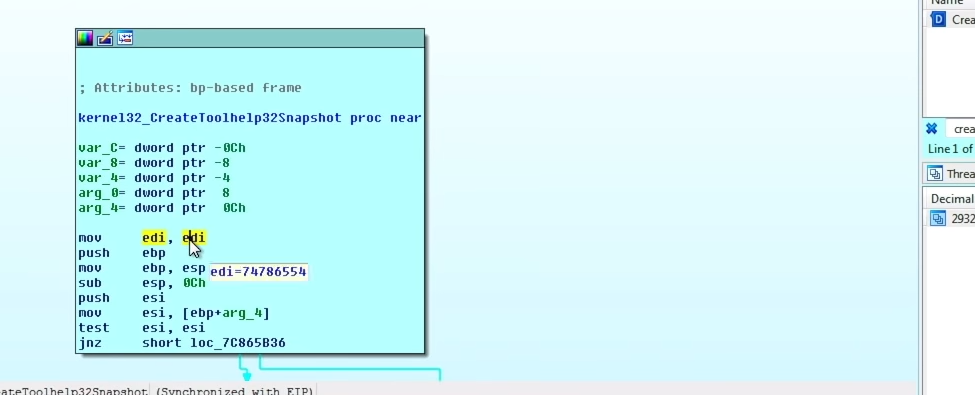
* API này sẽ kiểm tra tiến trình Debug dựa trên các Parameter cung cấp:

****

****

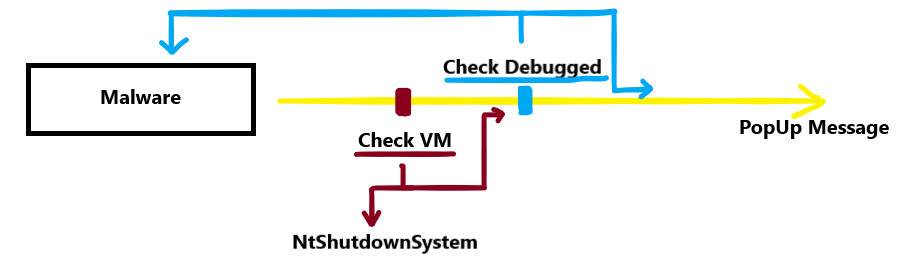
**CreateToolHelp32SnapShot:**

* Sau khi các hàm kiểm tra Debug được thực hiện, thì kiểm tra có bị Debug hay không, nếu có thì API CreateSnapShot sẽ thực hiện trở tiến trình trở về vị trí ban đầu (Snapshot) để Malware không bị Debug.

****

1. **Mã độc sẽ thực hiện hành vi bình thường (Ví dụ: hiển thị thông điệp MSSV và thực hiện lây nhiễm) khi không thấy sự tồn tại của môi trường máo ảo và trình gỡ lỗi. Ở trường hợp ngược lại chúng thực hiện che giấu hành vi này (Không thực hiện Popup).**

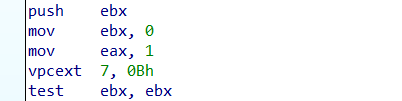
* Luồng hoạt động mà em thực hiện để viết Malware như sau:



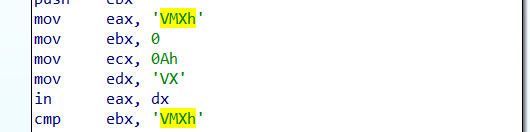
* **Giải thích:** 
  + AntiVM của em sẽ được thực hiện hoạt động trước, sau đó sẽ tới AntiDebug.
  + Sau khi AntiVM technique được thực hiện, nó sẽ kiểm tra sự tồn tại của VM và nếu như không có sẽ chuyển qua AnTiDebug, còn có thì sẽ dùng API NtShutdownSystem để tự động đánh sập VM.
  + Sau khi chuyển qua tới bước AntiDebug sẽ kiểm tra có đang bị Debug hay không nếu có sẽ Snapshot về EntryPoint của Malware, không thì PopUp Message.

**KỸ THUẬT ANTI-VM:**

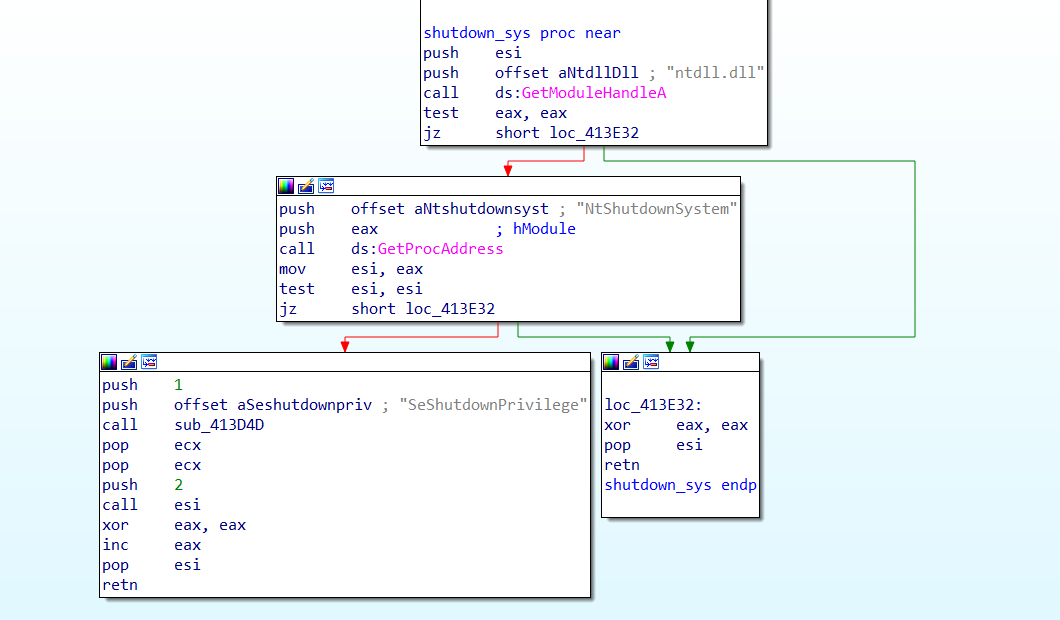
* Anti-VM em sẽ sử dụng cả 2 phương thức là VPCEXT và VMX. VPCEXT sẽ diễn tra trước rồi tới VPCEXT (phòng thủ 2 lớp).

****

* Mục đích của VPCEXT là kiểm tra cái lệnh đó “vpcext 7,0B” có đang chạy trong môi trường Virtual PC hay không, nếu có sẽ diễn ra Exception và trả về EBX = 1. Tiếp đến sẽ là:



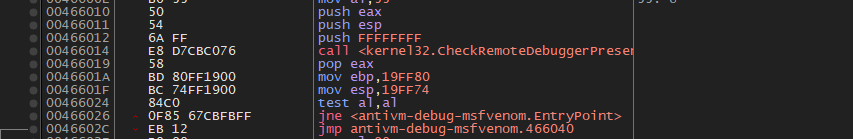
* VMX được gán vào eax, để sau này thực hiện I/O Port ở dòng “in eax, dx”, ebx được gán 0 để sau khi thực hiện VMX sẽ trả về ebx 1 giá trị mới, ecx được gán vào 0x0A mục đích là để lấy Virtual Machine Version, edx sẽ được gán với “VX” (2 bytes), dx là con của edx (1 nửa của edx là 2 bytes) kết hợp với eax để thực hiện gọi hàm kiểm tra VM.
* Sau khi thực hiện kiểm tra ebx sẽ trả về với “VMX” nếu như chạy trên môi trường máy ảo.
* **LƯU Ý: Cả 2 dạng kiểm tra này bắt buộc phải chạy ở Kernal Mode.**
* Sau khi đã kiểm tra nếu như có đang chạy trong máy ảo thì NtShutDownSystem sẽ được thực thi đánh sập nguồn của VM.



**KỸ THUẬT ANTI-DEBUG:**

* Trong kỹ thuật này em cũng sử dụng cả 2 phương pháp phòng thủ là CheckRemoteDebuggerPresent và IsDebuggerPresent.
* **Do cả 2 API này đều được ở trong NTDLL.DLL nên chúng ta Import nó rất dễ dàng.**

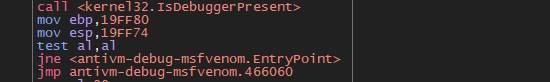
**CHECKREMOTEDEBUGGERPRESENT:**



* Trước khi gọi API CheckRemoteDebuggerPresent, ta phải đẩy thanh ghi eax, esp, -1 vào để thực hiện kiểm tra.
* Khi API được thực hiện xong sẽ pop eax ra nếu như đang bị Debug thì esp, ebp sẽ được gán lại giá trị cũ của entrypoint.

****

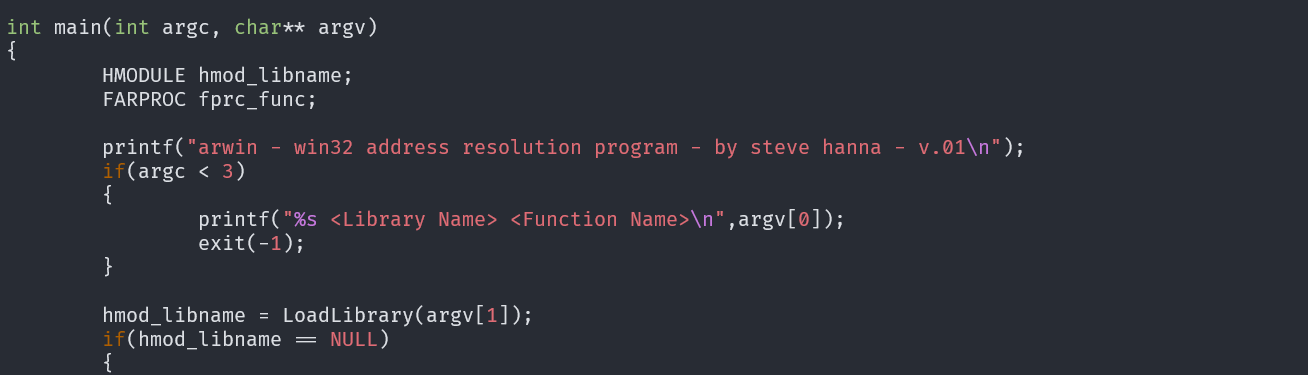
* **Và nhảy đến EntryPoint nếu không thì sẽ nhảy đến phòng thủ tiếp theo là IsDebuggerPresent.**

****

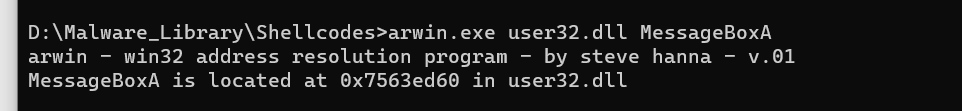
* Gọi API IsDebuggerPresent, cũng đưa ebp, esp về giá trị lúc ở EntryPoint và kiểm tra nếu có thì nhảy đến EntryPoint, không thì nhảy tới PopUpMessage.

**SHELLCODE WITH C:**

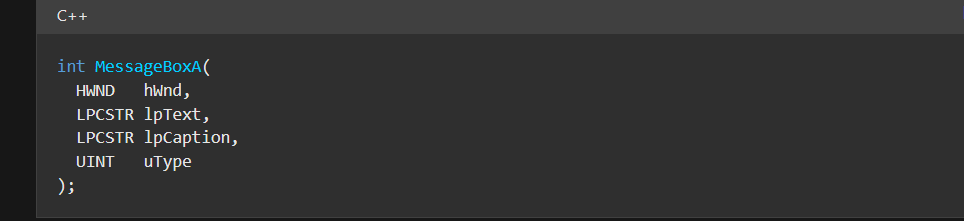
* Em sử dụng file code C sau để kiểm tra MessageBoxA đang nàm ở đâu:

****

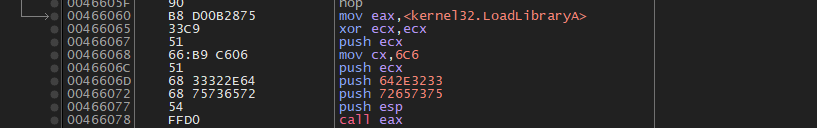
* Biên dịch và chạy chương trình theo lệnh sau:

****

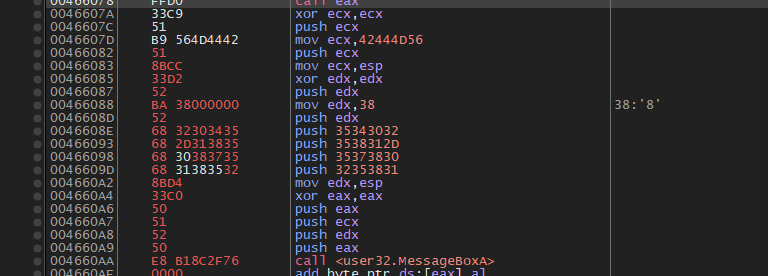
* **MessageBoxA nằm ở 0x7563ed60.**
* Từ đó, em lên trang Microsoft và xem hàm đó có 4 parameters như sau:

****

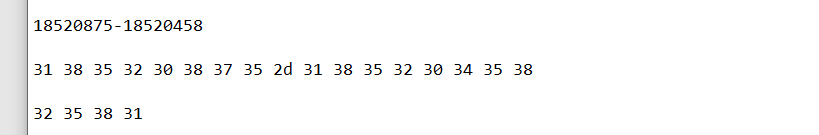
* **Trong đó:**
  + HWND và UINT có thể được gán là 0 hoặc NULL.
  + Trong kiến trúc 32 bits thì parameters cuối phải được push vào trước, rồi từ từ lên.
* Từ đó em thực hiện shellcode tự tạo như sau:

****

* **Giải thích:**
  + Sau khi Anti-Vm/Debug đã được hoàn thành thì LoadLibraryA của Kernel32.dll sẽ được thực hiện để load user32.dll lên đẩy vào Stack.
  + Đẩy esp (lúc này là địa chỉ của “user32.dll”) để gọi user32.dll.
* **Tiếp theo:**



* **Giải thích:**
  + Cần phải push 0x00 (NULL) tiếp theo để không bị
  + Đẩy tiếp Title là “VMDB” vào Stack.
  + Phần tiếp theo ta phải đẩy vào là Text Message nên, em muốn gõ vào là 18520875-18520458 thì khi chuyển sang Hex sẽ là:

****

* Đẩy 4 bytes giá trị vào với giá trị ngược lại:

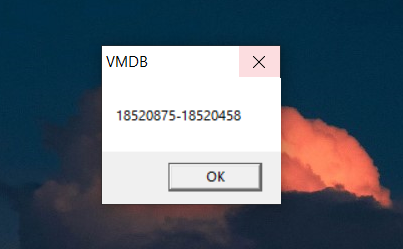
Ví dụ:

1 8 5 2 => 31 38 35 32.

* **Do cấu trúc của Little Endian sẽ chuyển ngược byte lại nên ta phải:**

mov ecx, 0x31383532 => 32 35 38 31.

* **Lúc này máy tính sẽ hiểu là 1852.**
* Phần tiếp theo là đẩy Title vào thì cũng tương tự.
* Đẩy 4 parameters theo thứ tự vào và ta sẽ gọi hàm.

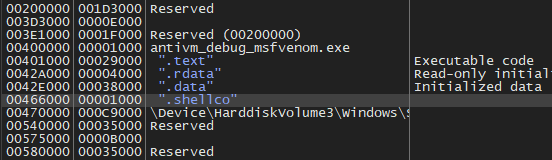


**SHELLCODE SẼ LÀ:**

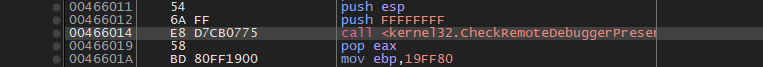
**B8 D0 0B 28 75 33 C9 51 66 B9 C6 06 51 68 33 32 2E 64 68 75 73 65 72 54 FF D0 33 C9 51 B9 56 4D 44 42 51 8B CC 33 D2 52 BA 38 00 00 00 52 68 32 30 34 35 68 2D 31 38 35 68 30 38 37 35 68 31 38 35 32 8B D4 33 C0 50 51 52 50 E8 B1 8C 2F 76**

**Sửa lỗi nếu có chạy không được File:**

* Thầy dùng các Debugger để mở file và mở đến những nơi sau:

****

* **Đến Section .Shellcode ở 0x00466000.**

****

****

* **Bấm Space và gõ lại đúng call CheckRemoteDebuggerPresent hoặc call IsDebuggerPresent nếu như trong trường hợp File chạy không được hoặc lệnh call bị biến đổi sang 1 địa chỉ nào đó.**