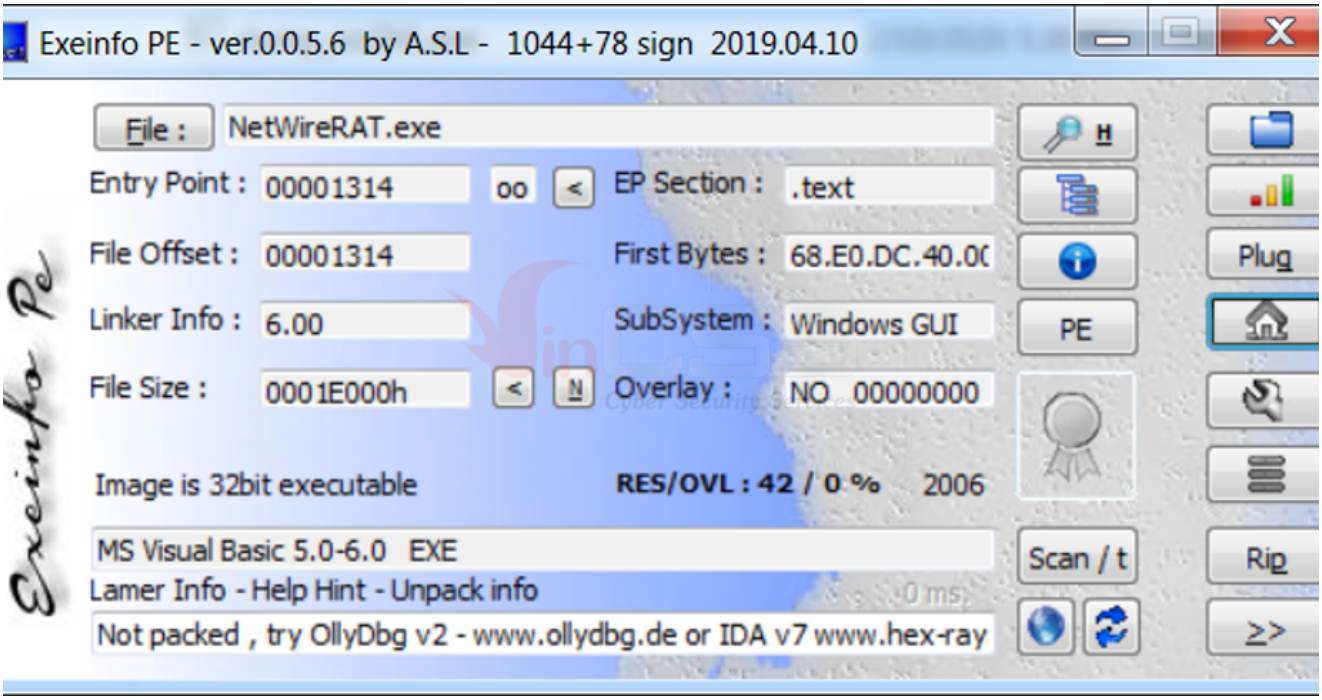
[RE011] Unpack crypter của malware Netwire bằng x64dbg

blog.vincss.net/2020/03/re011-unpack-crypter-cua-malware-netwire-bang-x64dbg.html



Gần đây, chúng tôi có tiếp cận một mẫu malware lạ. Kết quả trả về của sandbox cho thấy đây là một biến thể của dòng malware Netwire được pack bằng một loại crypter viết bằng VB6. Chúng tôi đã tiến hành phân tích mẫu này nhằm hiểu sâu về kỹ thuật của malware, đồng thời xây dựng thành tài liệu tham khảo cho những mẫu malware sử dụng kĩ thuật tương tự.

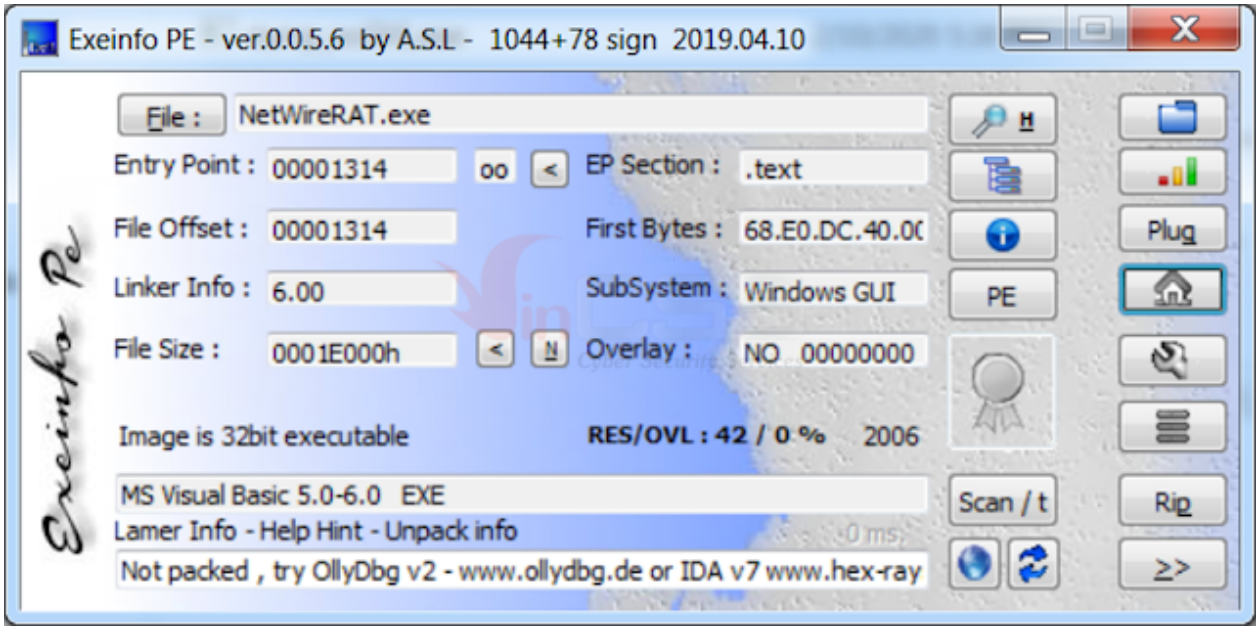
1. Công cụ sử dụng

2. Thông tin cơ bản về sample

> ***File Hash (SHA-256):*** *d381c5a5eeb46759bb5ebce67eb50cc61f91a75c204d6ec1c7750937f7f4c3f1*

> Nguồn:*Any.run*

Phân tích sơ bộ bằng các chương trình PE Scanner cho thấy mã độc được viết bằng **Visual Basic 6.0**



Hình 1. *Kết quả scan bằng Exeinfo*

3. Phân tích crypter

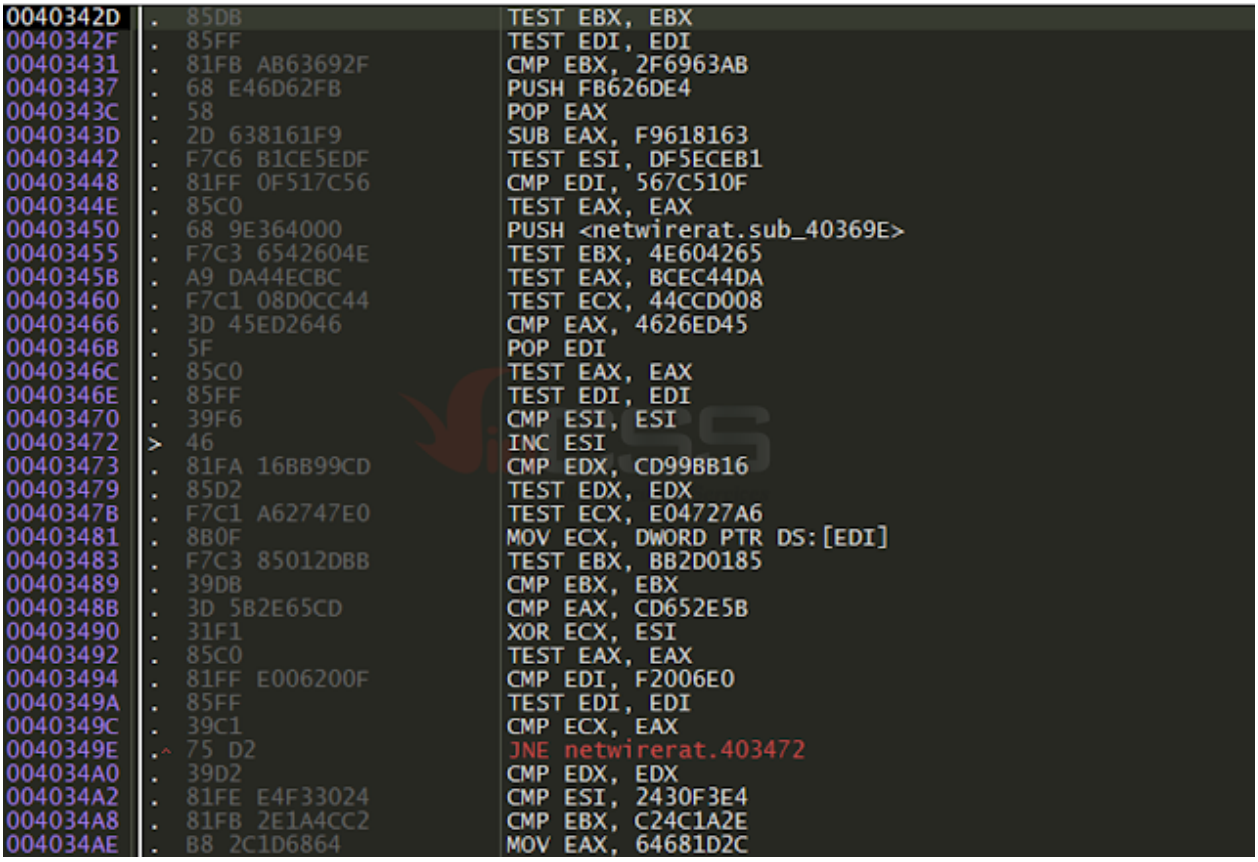
Một mẹo khi dịch ngược chương trình được biên dịch từ VB6 là sẽ có một loạt các lệnh jump nhảy đến các function được viết trong chương trình. Như trong trường hợp này, các lệnh jump nằm ở địa chỉ **0x40DF3C**. Có thể tìm các lệnh jump này bằng mẫu "**81 6C 24 04 ?? ?? ?? ?? E9**"

Hình 2. *Các lệnh jump đến function của VB6*

Có thể thấy mã độc chỉ có hai function chính. Chọn function tại địa chỉ **0x40E9DE** để phân tích vì function này có kích thước lớn nhất. Tiếp tục trace dần sẽ tới một hàm call như sau:

Hình 3. *Lệnh call đến shellcode*

Với những ai đã quen với việc dịch ngược các chương trình được viết bằng VB6 sẽ nhận thấy điểm bất thường khi trong code của chương trình VB6 gọi thẳng đến một thanh ghi. Tiếp tục debug vào lệnh call sẽ tới đoạn shellcode:

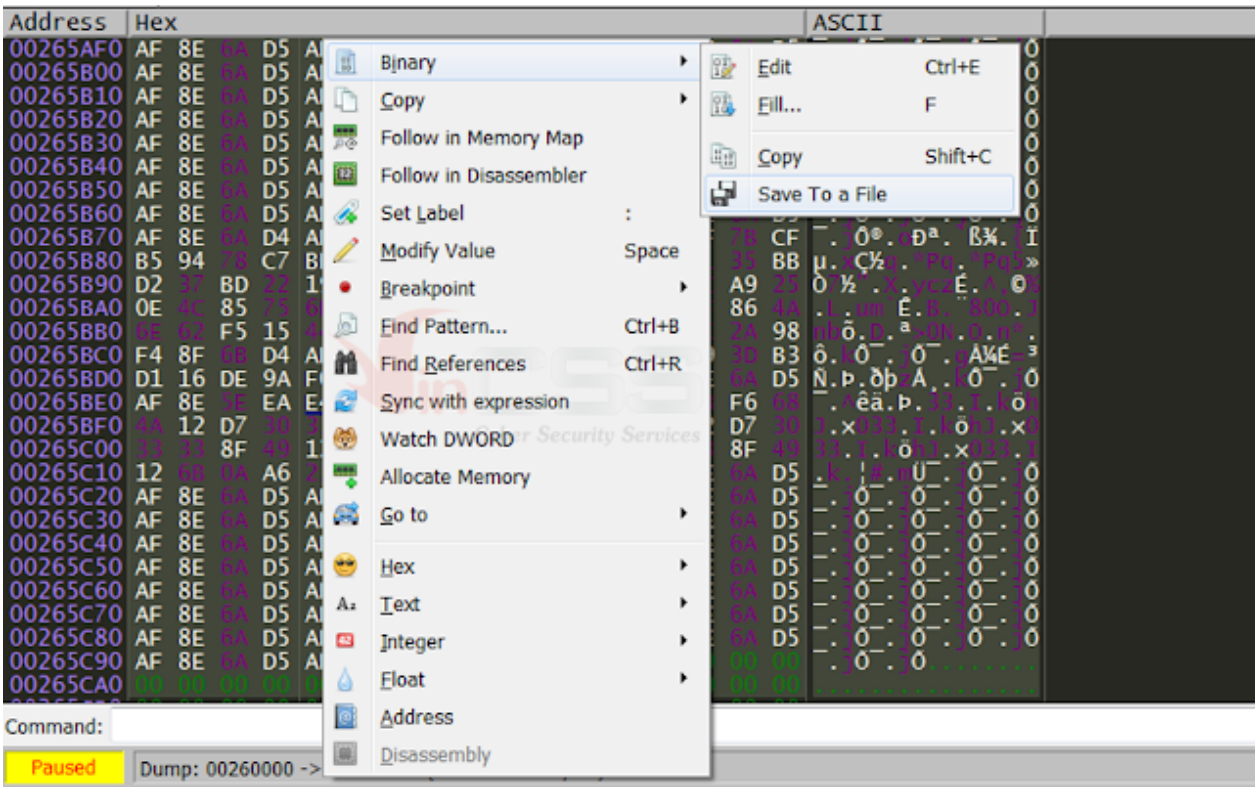


Hình 4. *Đoạn code cấp phát memory và chạy shellcode*

Toàn bộ đoạn shellcode này đã được obfuscated. Tuy nhiên, kích thước của nó cũng khá nhỏ nên nếu trace dần từng bước và tổng hợp thông tin, chúng ta sẽ thấy shellcode thực hiện những bước sau:

- Gọi hàm **VirtualAlloc** để cấp phát bộ nhớ.
- Giải mã shellcode và copy vào vùng bộ nhớ vừa được cấp phát.
- Tiến hành gọi shellcode mới.

Tới đây, có thể dùng x64dbg để dump toàn bộ shellcode mới và tiến hành phân tích. Để dump shellcode, lựa chọn đoạn cần dump trong cửa sổ Hex, chuột phải và chọn **Binary > Save To a File**:

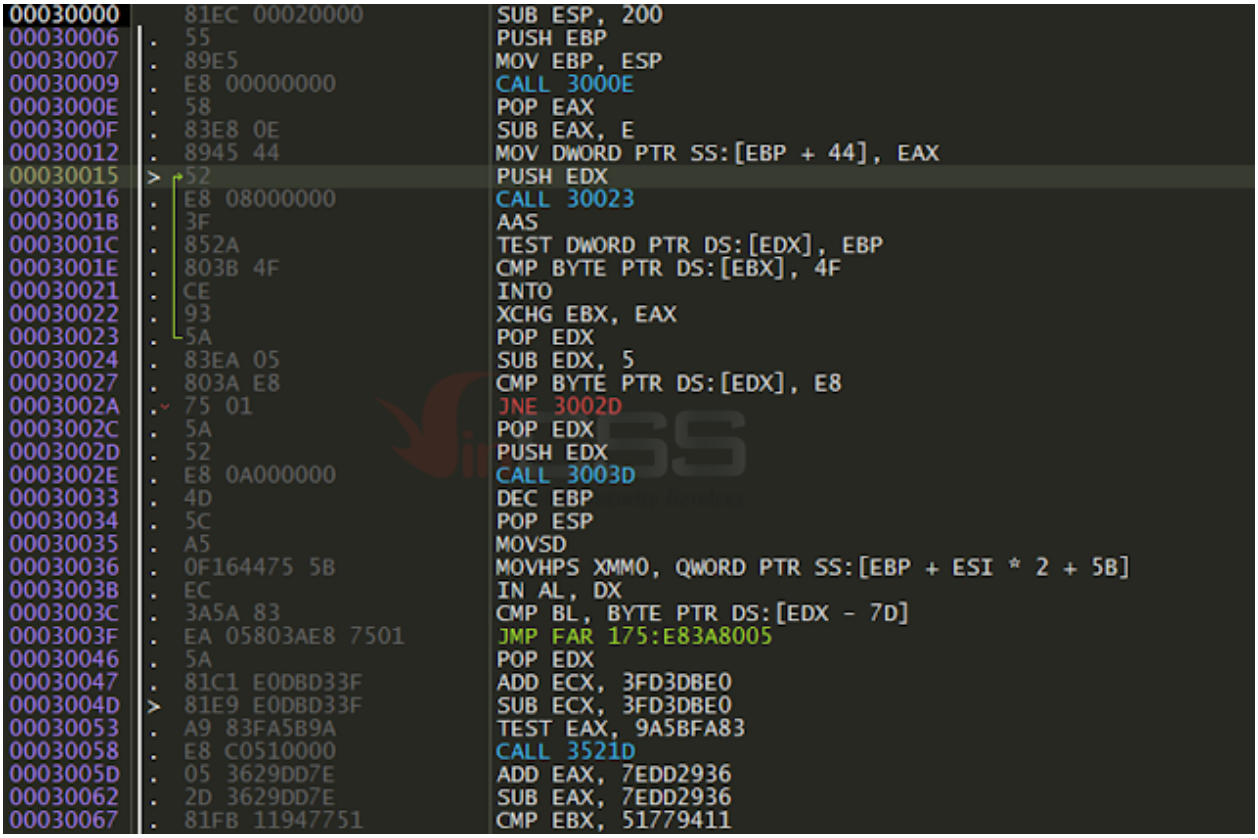


Hình 5. *Dump shellcode với x64dbg*

Với shellcode đã dump, có nhiều công cụ hỗ trợ để phân tích. Trong bài này, chúng tôi sử dụng **jmp2it** để load shellcode lên và debug bằng x64dbg. Chạy jmp2it với command như sau:

“Jmp2it.exe shellcode.bin 0x0 pause”

Tiến hành attach x64dbg vào process **jmp2it.exe** để debug shellcode. Shellcode lúc này được load ở địa chỉ **0x30000**.



Hình 6. *Shellcode thực hiện tác vụ độc hại*

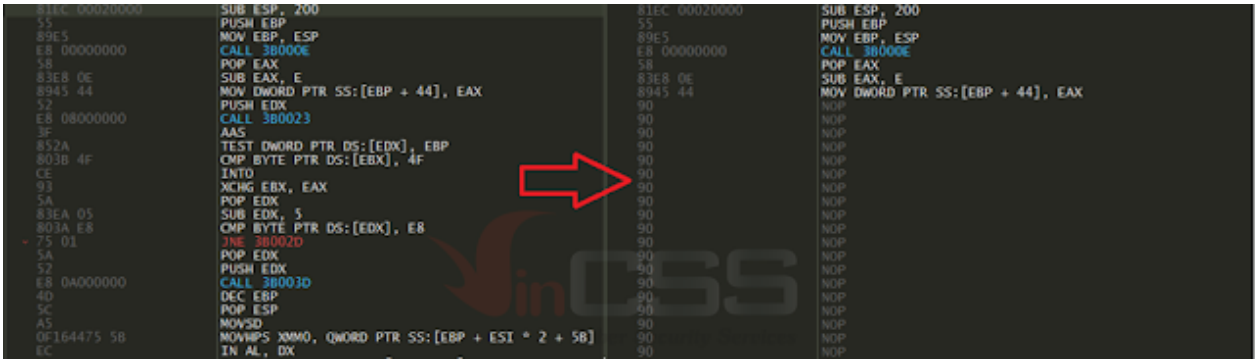
Như trên hình, shellcode mới này cũng bị obfuscate. Lúc này, có hai lựa chọn:
- Trace từng lệnh để debug.
- Viết tool/script để deobfuscate đoạn shellcode.

Vì shellcode là khá lớn nên ở đây chúng tôi lựa chọn phương án viết script để thực hiện deobfuscate đoạn shellcode này.

3.1. Deobfuscate shellcode

Để viết được script thực hiện deobfuscate đoạn shellcode, chúng ta cần hiểu một số pattern của đoạn shellcode. Ví dụ như sau:
- Pattern Push Reg/Pop Reg
- Pattern Inc Reg/Dec Reg
- Pattern Add Reg, Const/ Sub Reg, Const
- Pattern Push Reg/Xor Reg, Const/ Pop Reg
- Pattern CLD, CLC
- …

Các pattern trên tương ứng với lệnh **NOP**. Sau khi xác định được các pattern này, dùng python để viết script deobfuscate. Chi tiết script xem tại đây. Để sử dụng, lựa chọn đoạn mã cần deobfuscate và chạy script.

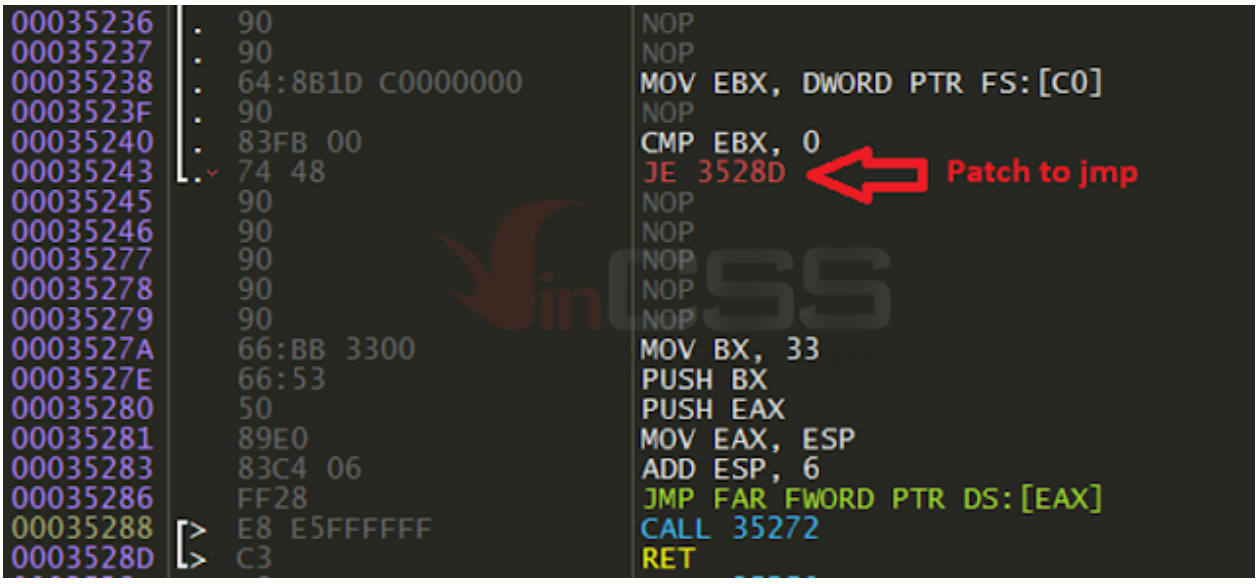


Hình 7. *Trước và sau khi deobfuscated*

**Lưu ý: script trên tạm thời bỏ qua tìm kiếm pattern 2 bytes vì pattern ngắn, không có hiệu quả nhiều trong việc obfuscate và rất dễ nhầm lẫn khi deobfuscate bằng cách search/replace byte pattern. Để xác định pattern 2 bytes một cách chính xác thì nên dùng pattern bằng asm. Hiện tại x64dbg chưa support tốt các pattern bằng asm.*

3.2. Heaven's gate

Shellcode áp dụng kỹ thuật heaven's gate[1] để làm rối trong quá trình debug. Đây là kỹ thuật thực thi mã từ x86 sang x64 bằng lệnh far jmp. Bằng cách đơn giản check ở địa chỉ **FS:[0xC0]** để xem hệ thống có phải là x64 hay không? Nếu là x64, shellcode dùng kỹ thuật heaven's call. Để debug tiếp được trơn tru trên x86, chúng tôi tiến hành patch lệnh nhảy sau lệnh so sánh để "lừa" shellcode thực thi trên x86.

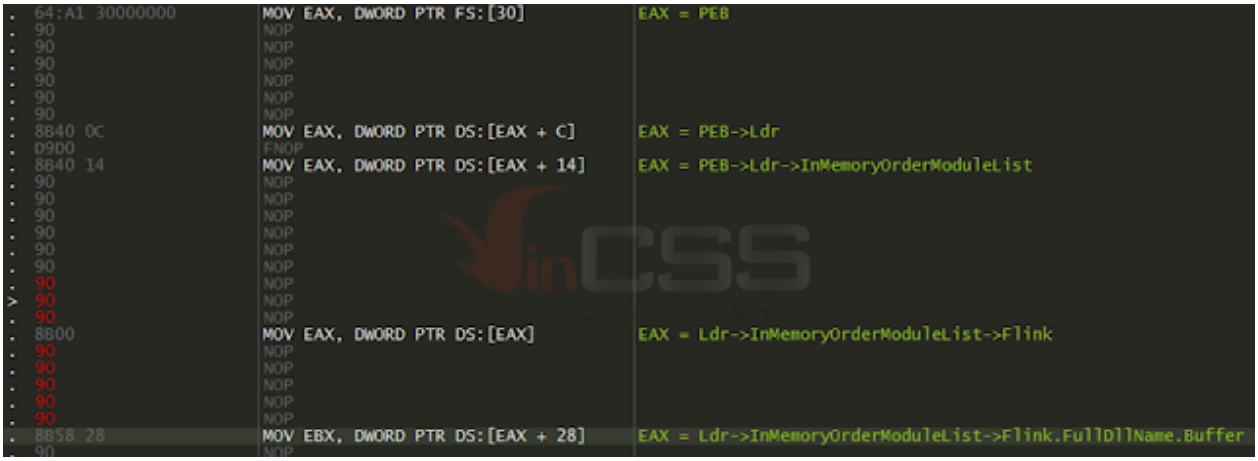


Hình 8. *Đoạn code thực thi Heaven's gate*

3.3. Resolve API

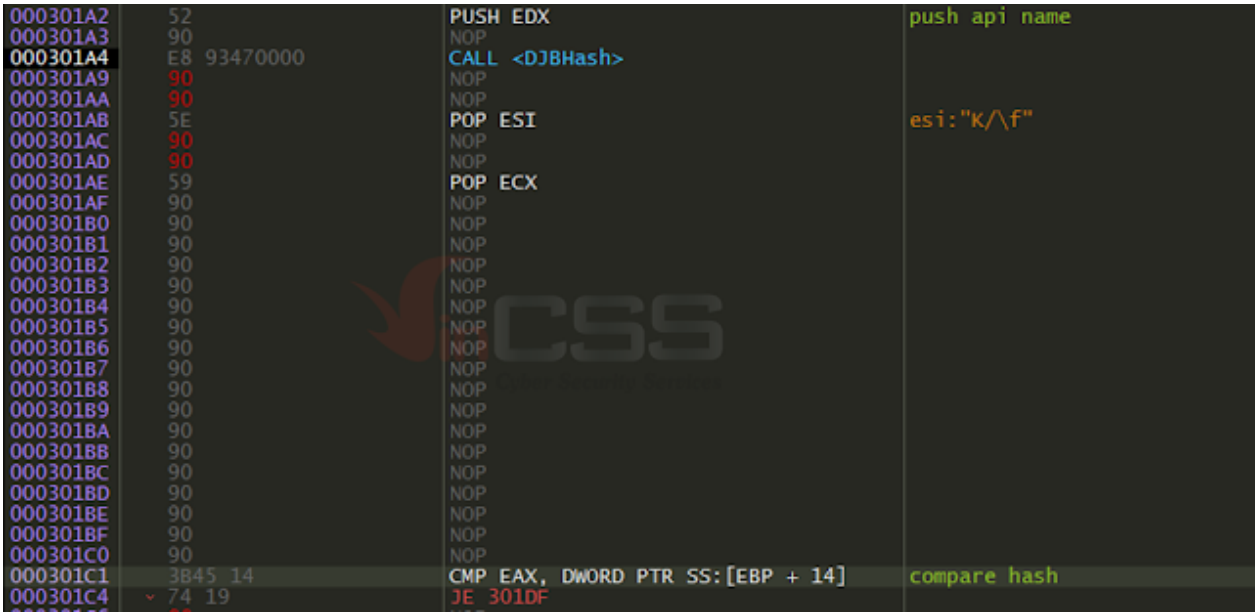
Có thể nói **GetProcAddress** là một API quan trọng để shellcode có thể tìm và gọi các

hàm API khác. Để tìm địa chỉ của hàm **GetProcAddress** cần có địa chỉ của **kernel32.dll**. Shellcode ở trên sẽ tiến hành truy xuất tới **PEB->Ldr->InMemoryOrderModuleList** và lấy địa chỉ của module tương ứng với tên **kernel32.dll**.



Hình 9. *Truy cập vào PEB lấy base tương ứng*

Sau khi có địa chỉ của **kernel32.dll**, shellcode tiếp tục tìm địa chỉ của API bằng API hash. Ở đây hash được sử dụng là DJB hash và giá trị hash của hàm "**GetProcAddress**" là **0xCF31BB1F**[2].



Hình 10. *Đoạn code hash tên API và tiến hành so sánh*

Dựa vào địa chỉ của **kernel32.dll** và API **GetProcAddress**, shellcode tiến hành resolve một loạt các API sau:

- LoadLibraryA
- TerminateProcess
- EnumWindows

- ZwProtectVirtualMemory
- DbgBreakPoint
- DbgUIRemoteBreakin

3.4. Anti attach

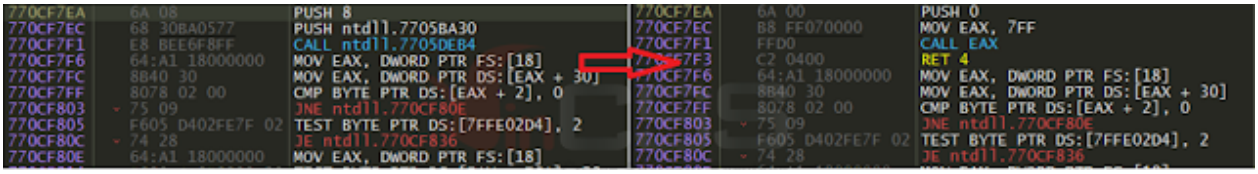
Shellcode gọi hàm **ZwProtectVirtualMemory** để đặt quyền **PAGE_EXECUTE_READWRITE** cho section **.text** của **ntdll**, sau đó tiến hành patch các hàm API **DbgBreakPoint** và **DbgUIRemoteBreakin** để anti attach.

```
00034D62  .  8B4424 18      MOV EAX, DWORD PTR SS:[ESP + 18]
00034D66  .  FC             CLD
00034D67  .  C600 90        MOV BYTE PTR DS:[EAX], 90          pDbgBreakPoint[0] = 0x90
00034D6A  .  90             NOP
00034D6B  .  90             NOP
00034D6C  .  90             NOP
```

Hình 11. *Code patch hàm DbgBreakPoint thành NOP*

```
00034D96  .  8B4424 1C      MOV EAX, DWORD PTR SS:[ESP + 1C]
00034D9A  .  89FF           MOV EDI, EDI
00034D9C  .  C600 6A        MOV BYTE PTR DS:[EAX], 6A          pDbgUIRemoteBreakin[0] = 0x6A
00034D9F  .  D9D0           FNOP
00034DA1  .  C640 01 00     MOV BYTE PTR DS:[EAX + 1], 0       pDbgUIRemoteBreakin[1] = 0x0
00034DA5  .  90             NOP
00034DA6  .  90             NOP
00034DA7  .  C640 02 B8     MOV BYTE PTR DS:[EAX + 2], B8      pDbgUIRemoteBreakin[2] = 0xB8
00034DAB  .  90             NOP
00034DB7  .  90             NOP
00034DB8  .  8B95 9C000000  MOV EDX, DWORD PTR SS:[EBP + 9C]
00034DBE  .  90             NOP
00034DBF  .  90             NOP
00034DC0  .  90             NOP
00034DC1  .  8950 03        MOV DWORD PTR DS:[EAX + 3], EDX    *(DWORD*)(pDbgUIRemoteBreakin
00034DC4  .  90             NOP
00034DE3  .  C640 07 FF     MOV BYTE PTR DS:[EAX + 7], FF      pDbgUIRemoteBreakin[7] = 0xFF
00034DE7  .  90             NOP
00034DE8  .  90             NOP
00034DE9  .  C640 08 D0     MOV BYTE PTR DS:[EAX + 8], D0      pDbgUIRemoteBreakin[8] = 0xD0
00034DED  .  90             NOP
00034DEE  .  90             NOP
00034DEF  .  90             NOP
00034DF0  .  C640 09 C2     MOV BYTE PTR DS:[EAX + 9], C2      pDbgUIRemoteBreakin[9] = 0xC2
00034DF4  .  90             NOP
00034DF5  .  90             NOP
00034DF6  .  90             NOP
00034DF7  .  90             NOP
00034DF8  .  90             NOP
00034DF9  .  90             NOP
00034DFA  .  90             NOP
00034DFB  .  90             NOP
00034DFC  .  90             NOP
00034DFD  .  90             NOP
00034DFE  .  90             NOP
00034DFF  .  90             NOP
00034E00  .  C640 0A 04     MOV BYTE PTR DS:[EAX + A], 4       pDbgUIRemoteBreakin[0xA] = 4
00034E04  .  90             NOP
00034E05  .  90             NOP
00034E06  .  C640 0B 00     MOV BYTE PTR DS:[EAX + B], 0       pDbgUIRemoteBreakin[0xB] = 0
```

Hình 12. *Code patch hàm DbgUIRemoteBreakin*



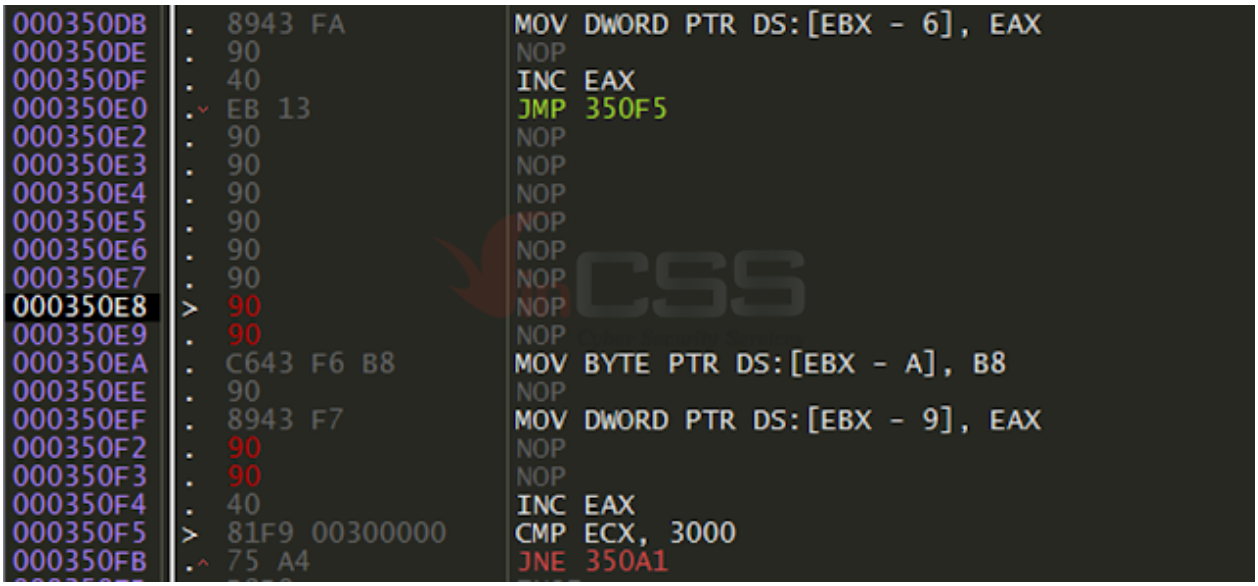
Hình 13. *Trước và sau khi patch hàm DbgUIRemoteBreakin*

3.5. Restore hook/breakpoint tại các hàm Zw*/Nt*

Shellcode thực hiện scan pattern "**B9 ?? ?? ?? ?? 8D 54 24 04**" và "**33 C9 8D 54 24 04**" trong code của thư viện **ntdll**. Đây là pattern nằm trong các hàm Zw*/Nt* gọi đến system call. Sau khi tìm thấy pattern này, shellcode sẽ khôi phục lại 5 bytes đầu tiên của các hàm:

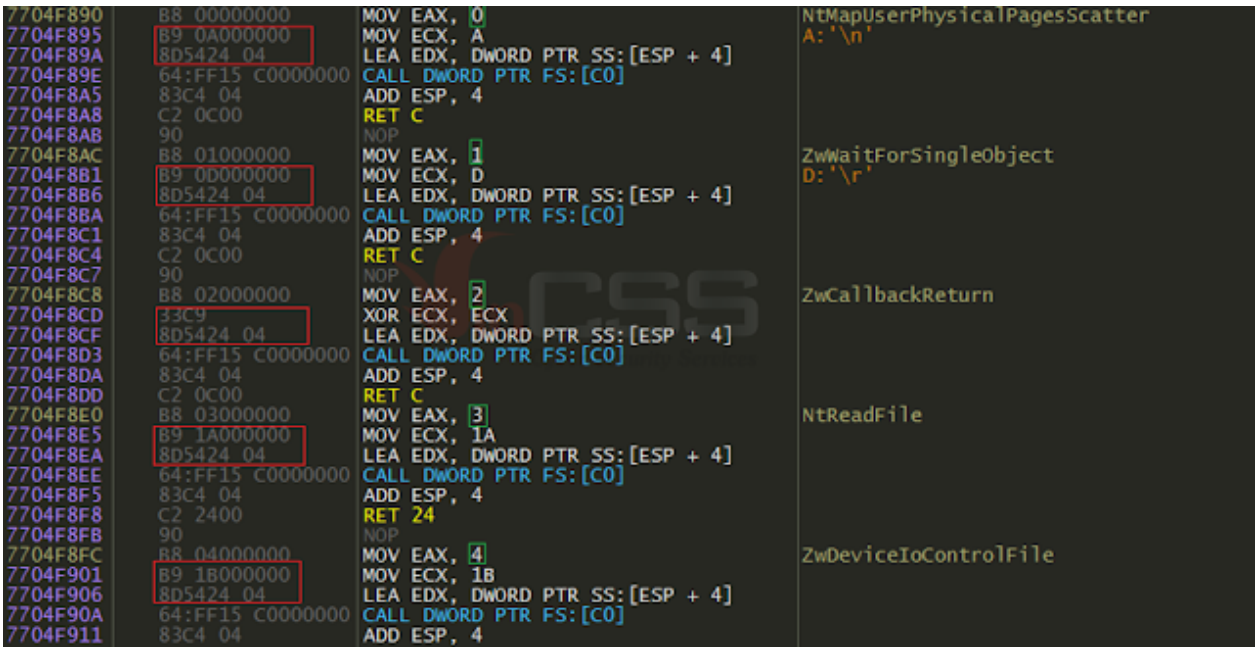
```
00035049  > 90              NOP
0003504A  . 90              NOP
0003504B  . 90              NOP
0003504C  . 90              NOP
0003504D  . 90              NOP
0003504E  . 90              NOP
0003504F  . 43              INC EBX
00035050  . 39C3            CMP EBX, EAX
00035052  .v 0F84 CD000000  JE 35125
00035058  . 90              NOP
00035059  . 90              NOP
0003505A  . 803B B8         CMP BYTE PTR DS:[EBX], B8
0003505D  .^ 75 EA          JNE 35049
0003505F  . 90              NOP
00035060  . 837B 01 00      CMP DWORD PTR DS:[EBX + 1], 0
00035064  .^ 75 E3          JNE 35049
00035066  . 90              NOP
00035067  . 90              NOP
00035068  . 807B 05 B9      CMP BYTE PTR DS:[EBX + 5], B9
0003506C  .^ 75 DB          JNE 35049
0003506E  . D9D0            FNOP
00035070  . 90              NOP
00035071  . 90              NOP
00035072  . BA 8D542404     MOV EDX, 424548D
0003508F  . 90              NOP
00035090  . 83C3 0A         ADD EBX, A
00035093  . 31C9            XOR ECX, ECX
00035095  . 90              NOP
00035096  . B8 01000000     MOV EAX, 1
0003509B  . 90              NOP
0003509C  . 90              NOP
0003509D  . 90              NOP
0003509E  . 90              NOP
0003509F  . 90              NOP
000350A0  . 90              NOP
000350A1  > 90              NOP
000350A2  . 90              NOP
000350A3  . 41              INC ECX
000350A4  . 43              INC EBX
000350A5  . 3B13            CMP EDX, DWORD PTR DS:[EBX]
000350A7  .v 75 4C          JNE 350F5
000350A9  . 90              NOP
000350AA  . 66:817B FE C933 CMP WORD PTR DS:[EBX - 2], 33C9
000350B0  .v 74 0A          JE 350BC
000350B2  . 90              NOP
000350B3  . 90              NOP
000350B4  . 807B FB B9      CMP BYTE PTR DS:[EBX - 5], B9
000350B8  .v 74 2E          JE 350E8
```

Hình 14. *Đoạn code search pattern*



Hình 15. *Đoạn code khôi phục lại 5 byte đầu tiên của API*

Một điểm hay cần học ở kỹ thuật này là các hàm Zw*/Nt* có các system call number tăng dần theo luồng từ trên xuống dưới. Nhờ đặc điểm này, shellcode có thể khôi phục lại lệnh "**mov eax, system call number**" một cách chính xác:



Hình 16. *Pattern được khoanh màu đỏ và system call number được khoanh màu xanh*

*Bug: Nếu hàm API với system call number là 0 hoặc dòng lệnh "**lea edx, dword ptr ss: [esp+4]**" bị thay đổi (hook, breakpoint,…), shellcode sẽ hủy toàn bộ system call number*

*của **ntdll**.*

3.6. Set hidden thread

Shellcode gọi hàm **ZwSetInformationThread** với tham số **0x11** để hide thread[3] trước debugger:

```
00030588    6A 00           PUSH 0
0003058A    90              NOP
0003058B    90              NOP
0003058C    6A 00           PUSH 0
0003058E    90              NOP
0003058F    6A 11           PUSH 11                 ThreadHideFromDebugger
00030591    90              NOP
00030592    90              NOP
00030593    6A FE           PUSH FFFFFFFE           current thread
00030595    90              NOP
00030596    90              NOP
00030597    90              NOP
00030598    90              NOP
00030599    90              NOP
0003059A    90              NOP
0003059B    90              NOP
0003059C    90              NOP
0003059D    90              NOP
0003059E    90              NOP
0003059F    90              NOP
000305A0    90              NOP
000305A1    FFD0            CALL EAX                Call ZwSetInformationThread
```

***Hình 17.** Đoạn mã shellcode set hidden thread*

3.7. Kiểm tra breakpoint

Shellcode sử dụng một hàm check trước khi gọi API. Nội dung hàm check như sau:

- Gọi **NtGetThreadContext** để check hardware breakpoint[3].
- Kiểm tra opcode **0xCC (int 3)** để phát hiện software breakpoint[3].

```
0003578D    90              NOP
0003578E    8B87 00500000   MOV EAX, DWORD PTR DS:[EDI + 5000]
00035794    90              NOP
00035795    90              NOP
00035796    90              NOP
00035797    90              NOP
00035798    90              NOP
00035799    90              NOP
0003579A    90              NOP
0003579B    90              NOP
0003579C    90              NOP
0003579D    8378 04 00      CMP DWORD PTR DS:[EAX + 4], 0       Dr0
000357A1    0F85 B0000000   JNE 35857
000357A7    90              NOP
000357A8    8378 08 00      CMP DWORD PTR DS:[EAX + 8], 0       Dr1
000357AC    0F85 A5000000   JNE 35857
000357B2    D9D0            FNOP
000357B4    8378 0C 00      CMP DWORD PTR DS:[EAX + C], 0       Dr2
000357B8    0F85 99000000   JNE 35857
000357BE    90              NOP
000357BF    90              NOP
000357C0    8378 10 00      CMP DWORD PTR DS:[EAX + 10], 0      Dr3
000357C4    0F85 8D000000   JNE 35857
000357CA    90              NOP
000357CB    8378 14 00      CMP DWORD PTR DS:[EAX + 14], 0      Dr6
000357CF    0F85 82000000   JNE 35857
000357D5    D9D0            FNOP
000357D7    8378 18 00      CMP DWORD PTR DS:[EAX + 18], 0      Dr7
000357DB    75 7A           JNE 35857
```

Hình 18. *Code phát hiện hardware breakpoint*

Hình 19. *Code phát hiện software breakpoint*

3.8. Create process và code injection

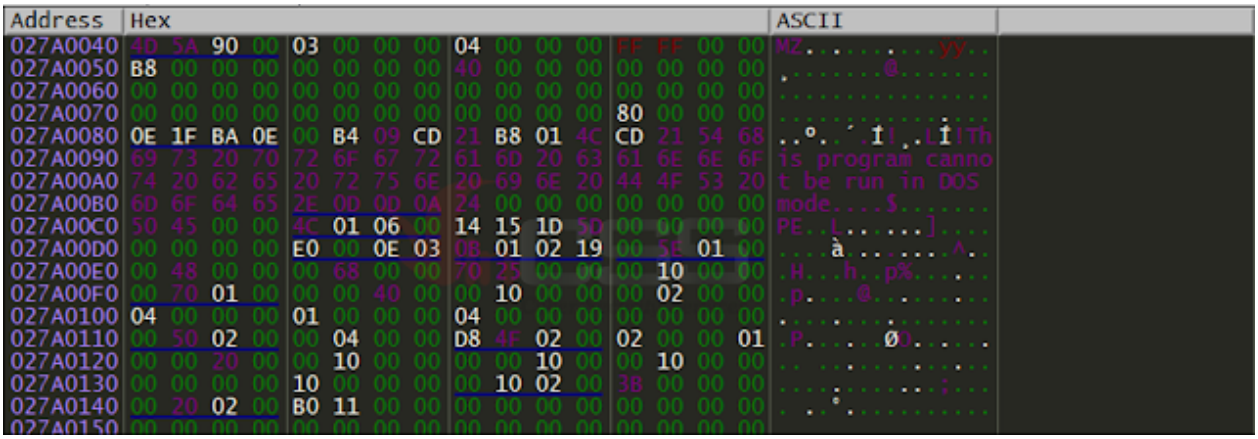
Cuối cùng shellcode thực hiện code injection bằng cách:

· Gọi **CreateProcessInternalW** với flag là **CREATE_SUSPENDED**.
· Gọi **ZwUnmapViewOfSection** với base là **0x400000.**
· Sử dụng **ZwCreateSection**/**NtMapViewOfSection** để cấp phát bộ nhớ.
· Nếu **ZwCreateSection**/**NtMapViewOfSection** bị lỗi, chuyển sang dùng API **ZwAllocVirtualMemory** để cấp phát bộ nhớ.
· Gọi **NtWriteVirtualMemory** để inject shellcode mới vào bộ nhớ.
· Cuối cùng gọi hàm **NtSetThreadContext**/**NtSetThreadContext**/ **NtResumeThread** để chạy shellcode và **TerminateProcess** để exit.

3.9. Phân tích shellcode thứ hai

Đoạn shellcode thứ hai thực chất là đoạn shellcode thứ nhất với config khác. Thay vì inject shellcode như đoạn shellcode thứ nhất, mục đích của đoạn shellcode này là tải về payload, decrypt và thực thi. Công việc chính nó thực hiện bao gồm:

· Sử dụng hàm của thư viện **wininet.dll** để download payload từ địa chỉ **https://drive[.]google[.]com/uc?export=download&id=1zEuX2HZcVvTYp7wzGtD1IXOSVLTBWVUe**
· Giải mã payload và map vào memory để thực thi. Payload này có kích thước **0x1AA40**, 40 bytes đầu tiên là phần header của payload.

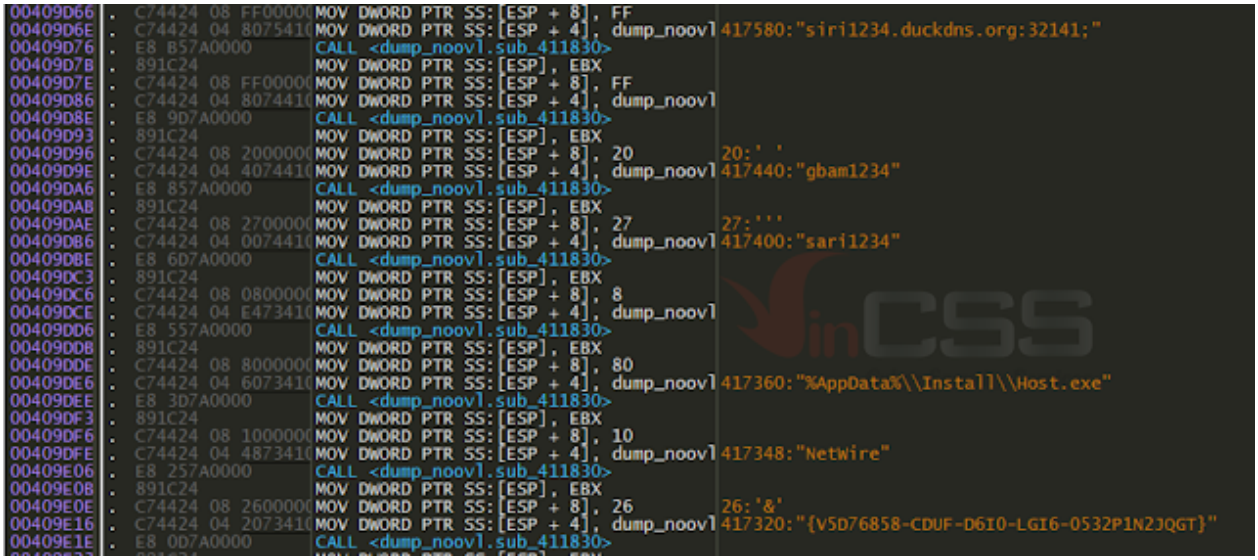


Hình 20. *Payload được decrypt trong memory*

Dump toàn bộ với size là **0x1AA00** sẽ thu được payload cuối cùng là **Netwire**.

3.10. Bonus

Khi load payload Netwire đã dump bằng x64dbg và đặt breakpoint tại **0x409E8A,** sẽ thu được config đã decrypted:



Hình 21. *Config được decrypt trong memory*

4. Tài liệu tham khảo

Các nguồn tham khảo được sử dụng làm tư liệu cho bài viết:

Để tiện theo dõi, chúng tôi cung cấp bài phân tích dưới dạng PDF:

File Name: CSS-RD-ADV-200304-011_Unpack crypter của malware Netwire bằng x64dbg.pdf
File Hash (SHA-256): 717efd6b8dd9a8a40ee34386311ab0f5689eb1f5f8fbd6df30b9cfdd8abe02c0

Dang Dinh Phuong
R&D Center - VinCSS (a member of Vingroup)