BÁO CÁO BÀI TẬP

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

**Tên chủ đề:** Advanced Virus Techniques

*GVHD:Phan Thế Duy.*

1. **THÔNG TIN CHUNG:**

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 | Tìm hiểu nguyên lý phát hiện sandbox (thí dụ như Cuckoo Sandbox,…) | 100% |
| 2 | Hiện thực lại mã độc chống phân tích động (trang bị thêm cho payload ban đầu) | 100% |
| 3 | Khả năng nhận biết môi trường (environmental sensivity):  + Chạy trong môi trường máy ảo,  + Chạy trong môi trường sandbox,  + Có khả năng phát hiện đang bị gỡ lỗi (debugging)  Một khi nhận biết đang bị đặt trong môi trường phân tích, nó sẽ không thực hiện hành vi, không thể hiện bản chất của mình (vd: payload không thực thi đoạn mã mục tiêu cho trước, dừng chương trình…). | 100% |

Bài tập được hoàn thành bởi : Anh Đức + Mạnh Cường

BÁO CÁO CHI TIẾT

**1.Tìm hiểu nguyên lý phát hiện sandbox (thí dụ như Cuckoo Sandbox,…)**

- Các phương pháp phát hiện sandbox, bao gồm Cuckoo Sandbox và các sandbox khác, thường dựa trên việc phân tích hành vi của máy chủ hoặc máy tính mà các ứng dụng độc hại đang chạy trên đó. Các phương pháp này có thể bao gồm:

1.Kiểm tra các giá trị hệ thống: Sandbox thường có một số giá trị hệ thống cụ thể, chẳng hạn như địa chỉ IP, tên máy tính, tên tài khoản người dùng, thư mục cài đặt Sandbox và các tệp tin hệ thống khác, do đó, các ứng dụng độc hại có thể kiểm tra những giá trị này để xác định liệu nó có đang chạy trên một môi trường sandbox hay không.

2.Kiểm tra hành vi mạng: Sandbox thường sử dụng các tường lửa ảo hoặc giám sát mạng để giám sát lưu lượng mạng. Các ứng dụng độc hại có thể phát hiện một số tính năng của các tường lửa ảo này, chẳng hạn như các cổng mạng đặc biệt được theo dõi hoặc một số lưu lượng mạng được giám sát.

3.Kiểm tra các API hệ thống: Sandbox thường có các API hệ thống đặc biệt, do đó, các ứng dụng độc hại có thể sử dụng các API này để xác định liệu nó có đang chạy trên một môi trường sandbox hay không.

4.Kiểm tra các giá trị đặc biệt: Các ứng dụng độc hại có thể kiểm tra các giá trị đặc biệt được sử dụng bởi các sandbox, chẳng hạn như các giá trị registry đặc biệt, các thông tin quản lý nhiệm vụ (Task Manager), các thông tin quản lý dịch vụ (Services Manager),...

1. **Hiện thực lại mã độc chống phân tích động (anti-debugging)**

**C1**:

* Anti-debugging techniques:
  + (malware) có thể phát hiện ra sự hiện diện của bộ gỡ rối (debugger) trên hệ thống và ẩn những hoạt động đáng ngờ của chúng để gây khó khăn cho các nhà phân tích phần mềm độc hại.
  + Một trong những cách đơn giản để phát hiện sự hiện diện của bộ gỡ rối là sử dụng hàm KERNEL32.IsDebuggerPresent trong WinAPI. Nếu hàm này trả về giá trị khác không, điều này có nghĩa là có một bộ gỡ rối đang chạy trên hệ thống và phần mềm độc hại có thể chuyển sang hoạt động ẩn để tránh bị phát hiện.

#include <windows.h>

#include <iostream>

using namespace std;

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

{

int num;

if (IsDebuggerPresent()) {

num = 0;

}

else {

num = 1;

MessageBox(NULL, L"20520890\_20520421", L"InfectionbyNT230", MB\_OK);

}

//printf("Number: %dn", num);

/\* wait \*/

getchar();

return 0;

}

[/plain]

Chỉnh sửa lại payload (thêm hàm IsDebuggerPresent()) và chèn vào y như Ex01

A screenshot of a computer

Description automatically generated

A screenshot of a computer

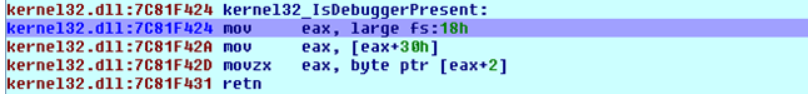
Description automatically generated

Đây là một dấu hiệu rõ ràng cho thấy chương trình thực thi sử dụng hàm để làm một việc gì đó khác biệt khi mà debugger được kết nối với chương trình thực thi. Chúng ta cũng có thể xác định được các chỉ thị chính xác được sử dụng để gọi hàm đó. Toàn bộ đồ thị Ida của hàm main thực hiện chính xác như hàm main trong mã nguồn C++ ở trên được trình bày trên hình bên dưới:

A screenshot of a computer program

Description automatically generated with medium confidence

Chúng ta có thể thấy rằng, ban đầu, chúng ta khởi tạo stack cho hàm và gọi hàm IsDebuggerPresent. Sau đó, chúng ta kiểm tra giá trị trả về trong eax so với chính nó để xác định xem giá trị true hay false được trả về. Nếu eax giữ giá trị khác không (1 trong trường hợp của chúng ta), thì cờ zero sẽ được thiết lập và hộp đầu tiên thiết lập [ebp + num] thành 0 sẽ được gọi. Vì chúng ta đang chạy chương trình dưới một trình gỡ lỗi, nhưng nếu không thì khối thiết lập [ebp + num] thành 1 sẽ được gọi. Sau đó, chúng ta chỉ đơn giản di chuyển giá trị của [ebp + num] vào thanh ghi eax và in nó ra bằng hàm printf. Nếu chúng ta đặt breakpoint trên lời gọi đến hàm IsDebuggerPresent và chạy lại chương trình, thực thi sẽ dừng ngay tại chỗ chúng ta muốn. Sau khi breakpoint được hit, chúng ta có thể vào hàm để xem hàm thực sự làm gì. Trong hình bên dưới, chúng ta có thể thấy hàm được đề cập:



Chúng ta có thể thấy rằng hàm IsDebuggerPresent rất đơn giản: đầu tiên, chúng ta load địa chỉ của thread đang hoạt động (TIB) vào thanh ghi eax, sau đó truy cập vào thành viên của cấu trúc được đặt tại offset 0x30; các cấu trúc dữ liệu PEB được đặt tại offset đó. Sau đó, chúng ta load địa chỉ của PEB vào eax và truy cập thành viên dữ liệu của nó tại offset 0x2, chứa thành viên dữ liệu được đặt tên là BeingDebugged. Như vậy, chúng ta đã thành công trong việc xem xét IsDebuggerPresent thực sự làm gì và làm như thế nào.

Chúng ta có thể thấy rằng nó rất đơn giản và không quá khó để vượt qua. Chúng ta có thể xác định rằng IsDebuggerPresent được sử dụng khi chúng ta cố gắng reverse mã nguồn của một chương trình thực thi và chương trình sẽ kết thúc, một nhánh thực thi khác được thực hiện, hoặc một điều gì đó không mong đợi xảy ra.

Trong những trường hợp như vậy, chúng ta trước tiên phải kiểm tra nếu hàm IsDebuggerPresent được gọi ở bất kỳ đâu trong chương trình thực thi. Nếu điều đó xảy ra, chúng ta có thể đơn giản là xóa các lệnh gọi hàm IsDebuggerPresent để chúng không gây phiền hà khi đảo ngược mã nguồn của chương trình thực thi. Trong khi đó, nếu chúng ta đang phát triển một chương trình và muốn sử dụng hàm gọi IsDebuggerPresent, chúng ta có thể sao chép các lệnh trên trực tiếp vào mã nguồn của chúng ta, để ta không thực sự gọi hàm IsDebuggerPresent trực tiếp, mà sử dụng các lệnh thân hàm của nó để tìm hiểu xem trình gỡ lỗi có được sử dụng để chạy chương trình thực thi hay không. Điều này chỉ là một thủ thuật khác để các reverse engineers đảo ngược mã nguồn không ngay lập tức nhận ra việc sử dụng hàm IsDebuggerPresent và làm cho việc debug phức tạp hơn một chút.

* Anti-debugging

**C2: Detecting a debugger using PEB:**

* Sử dụng cấu trúc PEB để phát hiện xem một trình gỡ rối (debugger) có đang được sử dụng để chạy chương trình hay không. Khi quá trình được tạo bằng CreateProcess API và nếu cờ tạo được thiết lập là DEBUG\_ONLY\_THIS\_PROCESS, thì một trường đặc biệt sẽ được thiết lập trong cấu trúc PEB trong bộ nhớ. Hàm detectDebugger được sử dụng để truy cập trường này và xác định xem trình gỡ rối có được sử dụng hay không. Nếu phát hiện trình gỡ rối đang được sử dụng, chương trình sẽ hiển thị một hộp thoại cảnh báo. Nếu không, nó sẽ hiển thị một thông báo khác cho biết trình gỡ rối không được phát hiện.

Tương tự cách trên

[plain]  
#include <windows.h>

#define WIN32\_LEAN\_AND\_MEAN

int detectDebugger()

{

\_\_asm

{

ASSUME FS : NOTHING

MOV EAX, DWORD PTR FS : [18]

MOV EAX, DWORD PTR DS : [EAX + 30]

MOVZX EAX, BYTE PTR DS : [EAX + 2]

RET

}

}

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

{

int num;

if (detectDebugger()) {

num = 0;

MessageBox(NULL, L"Debugger detected !!!", L"InfectionbyNT230", MB\_OK);

}

else {

num = 1;

MessageBox(NULL, L"20520890\_20520421", L"InfectionbyNT230", MB\_OK);

}

//printf("Number: %dn", num);

}

[/plain]

**ANTI-VM**

* Vì các mẫu phần mềm độc hại thường được phân tích trong một môi trường cô lập như máy ảo, nên chúng sẽ bao gồm bảo vệ chống máy ảo hoặc đơn giản là thoát ra khi chạy trên một môi trường cô lập.
* Các kỹ thuật sau có thể được sử dụng để phát hiện xem một mẫu đang chạy trong một máy ảo hay không:
  + Phát hiện dựa trên thời gian
  + Phát hiện dựa trên những hiệu ứng để lại trên hệ thống (Artifacts)
* Trong đó, phát hiện dựa trên thời gian sử dụng Time Stamp Counter (TSC), đây là một thanh ghi 64-bit được tìm thấy trên tất cả các bộ xử lý x86 từ Pentium trở lên. Nó đếm số chu kỳ từ khi thiết bị được khởi động.
* Khi mã được giả lập, sẽ có sự khác biệt về thời gian đếm giữa máy ảo và máy thật. Vì vậy, phát hiện các sự khác biệt này có thể giúp xác định xem một mẫu phần mềm độc hại có đang chạy trong một môi trường ảo hay không.

#include <iostream>

using namespace std;

int main()

{

unsigned int time1 = 0;

unsigned int time2 = 0;

\_\_asm

{

RDTSC

MOV time1, EAX

RDTSC

MOV time2, EAX

}

if ((time2 - time1) > 100)

{

//cout << "VM Detected" << endl;

return 0;

}

//cout << "VM not present" << endl;

MessageBox(NULL, L"20520890\_20520421", L"InfectedbyNT230", MB\_OK);

return 0;

}

**C2: Artifact based detection**

* Malware sử dụng sự hiện diện của cấu hình máy ảo dựa trên các file, network hoặc device artifacts. Malware thường kiểm tra sự hiện diện của những artifacts này để phát hiện sự hiện diện của debugger hoặc môi trường ảo.
* Trong trường hợp tốt nhất sẽ là artifacts, Vmware tạo ra các khóa registry cho Bộ điều khiển đĩa ảo, có thể được tìm thấy trong registry bằng cách sử dụng khóa sau đây. HKLMSYSTEMCurrentControlSetServicesDiskEnum như “SCSIDisk&Ven\_VMware\_&Prod\_VMware\_Virtual\_S&Rev\_1.04&XXX&XXX”

int main()

{

TCHAR lszValue[100];

HKEY hKey;

if (RegOpenKeyEx(HKEY\_LOCAL\_MACHINE, \_T("SYSTEM\\CurrentControlSet\\Services\\DiskEnum"), 0, KEY\_READ, &hKey) == ERROR\_SUCCESS)

{

DWORD valueSize = sizeof(lszValue);

if (RegQueryValueEx(hKey, \_T("0"), NULL, NULL, (LPBYTE)lszValue, &valueSize))

{

if (\_tcsstr(lszValue, \_T("VMware")))

{

//std::cout << "VMware Detected" << std::endl;

}

}

RegCloseKey(hKey);

}

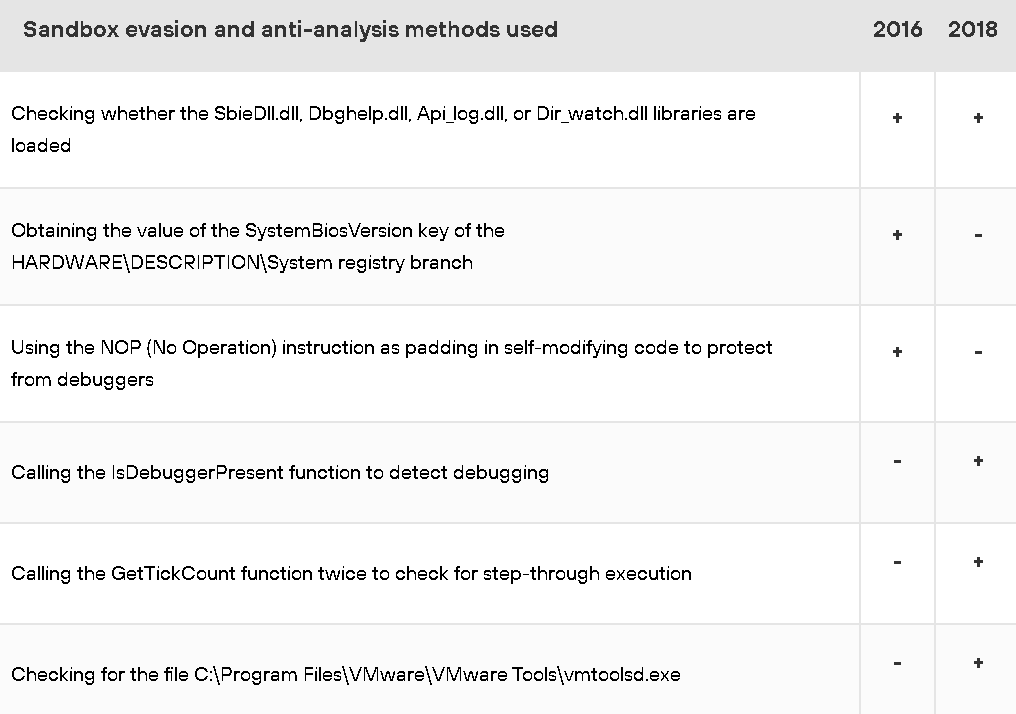
MessageBox(NULL, L"20520890\_20520421", L"InfectedbyNT230", MB\_OK);

return 0;

}

**Anti-Sandbox :**

* Sandbox thường sử dụng các sự kiện (callbacks) để giám sát các hoạt động của chương trình. Tuy nhiên, việc giám sát trực tiếp sẽ làm thay đổi mã nguồn của hệ thống và dễ bị phát hiện.
  + Do đó, các sandbox thường sử dụng các phương pháp giám sát gián tiếp bằng cách chặn các lệnh hệ thống hoặc các chức năng thư viện. Nhưng điều này cũng có nhược điểm là không thể giám sát các lệnh thực thi giữa các chức năng. Kẻ tấn công có thể tận dụng điểm yếu này bằng cách sử dụng mã chậm hoặc các kỹ thuật khác để thực thi các lệnh giữa các chức năng mà không bị phát hiện.
* "Chẩn đoán" - Diagnosing sandbox.
  + Kỹ thuật này bao gồm việc quét các đặc điểm của hệ thống và máy ảo để xác định môi trường sandbox. Sandboxes đã sử dụng các kỹ thuật để che giấu môi trường của chúng để đánh lừa malware, nhưng những giải pháp này thường là ngắn hạn.
  + Phân tích mã tĩnh cung cấp khả năng phát hiện hiệu quả nhất, tuy nhiên các chương trình malware đã sử dụng hành vi động để làm cho việc phát hiện trở nên khó khăn đối với các sandbox sử dụng phân tích mã tĩnh.
* "Checking for human pulse", là chương trình độc hại sẽ kiểm tra sự tương tác của người dùng trên hệ thống trước khi thực thi.
  + Khi có sự tương tác của con người như di chuyển chuột hoặc nhấn nút, chương trình độc hại sẽ dừng thực thi để tránh bị phát hiện bởi môi trường sandbox. Việc tương tác của con người trên hệ thống là rất khó để mô phỏng trong môi trường ảo, do đó, điều này khiến cho các chương trình độc hại khó bị phát hiện bởi sandbox.
* Một trong các kỹ thuật khác là trì hoãn việc thực thi của mã độc, để làm cho việc phát hiện của sandbox trở nên khó khăn hơn.
  + Kỹ thuật khác thứ 2 là kiểm tra trạng thái của user trong hệ thống trước khi thực thi chương trình độc hại. Điều này có thể làm cho sandbox khó phát hiện được chương trình độc hại do các hoạt động của con người, chẳng hạn như sự di chuyển chuột hoặc nhấn nút, khó mô phỏng được.
  + Một kỹ thuật khác để tránh bị phát hiện là tiến hành mã độc "bí mật", bằng cách tạo ra những giả lập tạm thời và không thực hiện bất kỳ hoạt động độc hại nào trong quá trình này. Nó giúp tránh bị phát hiện bởi sandbox, trong khi vẫn giữ lại khả năng thực hiện các hành động độc hại sau đó. Tuy nhiên, các nhà phát triển sandbox cũng đã phát triển các kỹ thuật để phát hiện và phân tích chương trình độc hại, chẳng hạn như phân tích tĩnh và phân tích động.
* sandbox không được coi là giải pháp hoàn hảo cho việc phát hiện malware.
  + Malware đã tìm ra cách để né tránh việc phát hiện bởi sandbox bằng cách sử dụng các kỹ thuật như (stalling code).
    - Được sử dụng để kéo dài thời gian thực thi của malware, làm cho hệ thống phân tích tự động tưởng rằng chương trình không có hành động độc hại và bỏ qua nó.
    - Tuy nhiên, thực tế là mã độc vẫn đang chạy, và khi nó chạy trên hệ thống thật sự, nó có thể gây hại cho hệ thống. Điều này làm cho việc phát hiện malware bằng sandbox trở nên khó khăn hơn.



**C1:** Kiểm tra xem các thư viện SbieDll.dll, Dbghelp.dll, Api\_log.dll hoặc Dir\_watch.dll có được tải lên hay không

int main()

{

HMODULE modules[] = { GetModuleHandle(L"SbieDll.dll"), GetModuleHandle(L"Dbghelp.dll"), GetModuleHandle(L"Api\_log.dll"), GetModuleHandle(L"Dir\_watch.dll") };

int numModules = sizeof(modules) / sizeof(HMODULE);

for (int i = 0; i < numModules; i++) {

if (modules[i] != NULL) {

std::wcout << L"Library " << i + 1 << L" is loaded" << std::endl;

}

else {

MessageBox(NULL, L"20520890\_20520421", L"InfectionbyNT230", MB\_OK);

}

}

return 0;

}

Các thư viện này thường được sử dụng trong môi trường sandbox và khi chương trình phát hiện được rằng chúng được tải lên, nó có thể cho rằng đang chạy trong môi trường sandbox và thực hiện một số hành động để tránh bị phát hiện hoặc phá vỡ môi trường sandbox đó.

Tuy nhiên, để tránh bị phát hiện, các malware có thể kiểm tra xem các thư viện này đã được tải lên hay chưa và nếu chưa thì chúng có thể tự động tải chúng xuống và sử dụng để đánh lừa các chương trình phát hiện môi trường sandbox. Ngoài ra, chúng có thể sử dụng các kỹ thuật khác để tránh bị phát hiện trong môi trường sandbox, như chạy chậm code, tạo ra các tập tin giả lập, thực thi một số mã ngẫu nhiên hoặc không thực hiện hành động độc hại trong môi trường sandbox.

**C2**: Gọi hàm GetTickCount hai lần để kiểm tra việc thực thi step-through

#include <windows.h>

#include <iostream>

#include <iostream>

#include <Windows.h>

#include <tchar.h>

#define WIN32\_LEAN\_AND\_MEAN

using namespace std;

int main()

{

DWORD tickCount1 = GetTickCount();

DWORD tickCount2 = GetTickCount();

if (tickCount2 - tickCount1 > 10) {

std::cout << "Step-through execution detected" << std::endl;

}

else {

//std::cout << "Normal execution detected" << std::endl;

MessageBox(NULL, L"20520890\_20520421", L"InfectionbyNT230", MB\_OK);

}

return 0;

}

* Việc gọi hàm GetTickCount hai lần là để kiểm tra việc thực thi step-through được sử dụng để phát hiện xem một chương trình có đang chạy trong môi trường sandbox hay không.
  + Môi trường sandbox thường được thiết lập để giới hạn thời gian thực thi chương trình. Khi một chương trình chạy trong môi trường này, việc gọi GetTickCount lần đầu tiên sẽ trả về một giá trị thời gian, và nếu thời gian thực thi của chương trình vượt quá một giới hạn nhất định, chương trình sẽ bị chấm dứt.
  + Tuy nhiên, nếu chương trình được chạy trong một môi trường step-through, việc gọi GetTickCount lần thứ hai sẽ trả về một giá trị khác biệt so với lần đầu tiên, vì các môi trường step-through thường cho phép chương trình dừng lại và tiếp tục thực thi từng bước một.
  + Việc kiểm tra này cho phép các malware phát hiện xem chương trình của họ đang chạy trong môi trường sandbox hay không, và nếu phát hiện ra rằng chương trình đang chạy trong môi trường này, chúng có thể thay đổi hành vi của chương trình để tránh bị phát hiện và chấm dứt.
* Do đó, việc sử dụng GetTickCount để kiểm tra việc thực thi step-through là một kỹ thuật anti-sandbox phổ biến trong các phần mềm độc hại.

**In a word** : Trong môi trường bình thường, khi gọi hàm GetTickCount liên tiếp thì hai lần gọi đó sẽ trả về giá trị gần giống nhau, thường là chỉ khác nhau 1 đơn vị thời gian (milisecond).

Tuy nhiên, khi một mã độc được chạy trong môi trường step-through, thì các câu lệnh sẽ được thực thi từng bước một, thường là thông qua các breakpoint được đặt ra trước đó.

Khi đó, khi gọi hàm GetTickCount liên tiếp thì sẽ trả về các giá trị khác nhau, đôi khi còn khá lớn, bởi vì việc thực thi các câu lệnh mất thời gian hơn so với môi trường bình thường.

Việc kiểm tra sự khác biệt giữa hai lần gọi hàm GetTickCount có thể được sử dụng để phát hiện một môi trường đang chạy trong chế độ step-through, và giúp cho mã độc có thể tránh phát hiện và tránh bị chặn bởi các giải pháp bảo mật.

* An-ti sandbox

**Kết quả :**

1. **Anti-sandbox**

Trước : [Cuckoo Sandbox (cert.ee)](https://cuckoo.cert.ee/analysis/4065158/summary/)

[VirusTotal - File - 8ceb33c19e44db227aa24b142a318271ac4c14d1ad224e91fa93fc0deb6595f9](https://www.virustotal.com/gui/file/8ceb33c19e44db227aa24b142a318271ac4c14d1ad224e91fa93fc0deb6595f9?nocache=1)

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Sau :

[VirusTotal - File - fefeac4c10bbe237cc6c861229ecaacbd2a366ac4fbd04a3862b62bd7a778093](https://www.virustotal.com/gui/file/fefeac4c10bbe237cc6c861229ecaacbd2a366ac4fbd04a3862b62bd7a778093)

[Cuckoo Sandbox (cert.ee)](https://cuckoo.cert.ee/analysis/4065164/summary/)

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

**Anti-VM:**

Trước:

A screenshot of a computer

Description automatically generated with medium confidence

Sau:

A picture containing text, software, computer, screenshot

Description automatically generated

**Anti-debugging**:

Trước:

A screenshot of a computer

Description automatically generated

Sau:

A screenshot of a computer

Description automatically generated

1. [↑](#footnote-ref-1)