BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA MẠNG MÁY TÍNH VÀ TRUYỀN THÔNG

ĐỒ ÁN CHUYÊN NGÀNH

Đề tài:
HOÀN THIỆN HỆ THỐNG PHÂN TÍCH MÃ ĐỘC 10SANDBOX

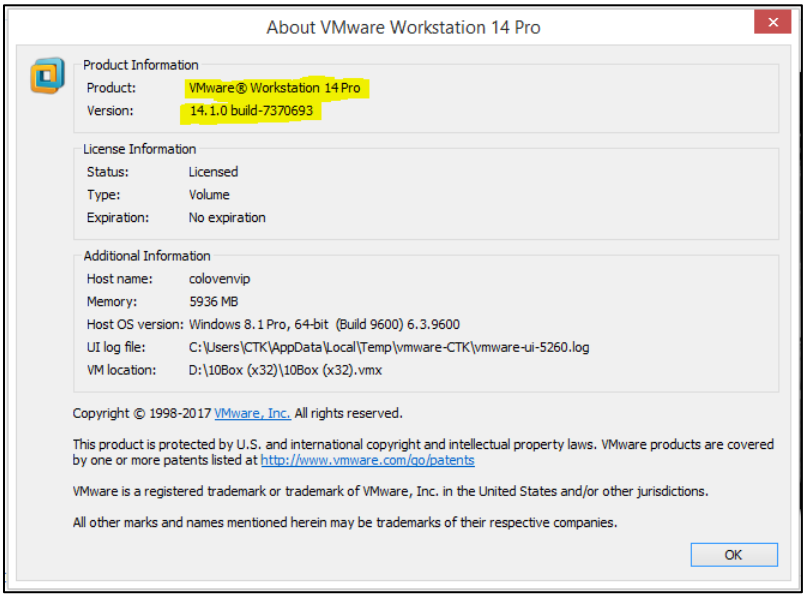
Lớp: **NT114.I21.ANTT**
Sinh viên thực hiện: **Châu Tuấn Kiệt - 15520397**
Võ Quốc Vương – 15521035
Giảng viên hướng dẫn: **TS. Phạm Văn Hậu**

Hồ Chí Minh 2018

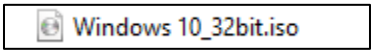
MỤC LỤC

1. Cấu hình môi trường

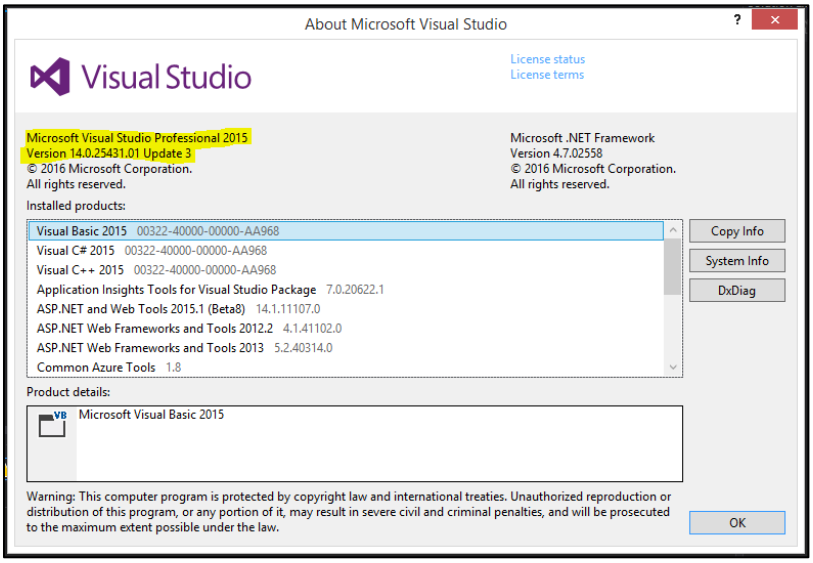
- Cần chuẩn bị:
  1. Phần mềm VMWare Workstation 14.



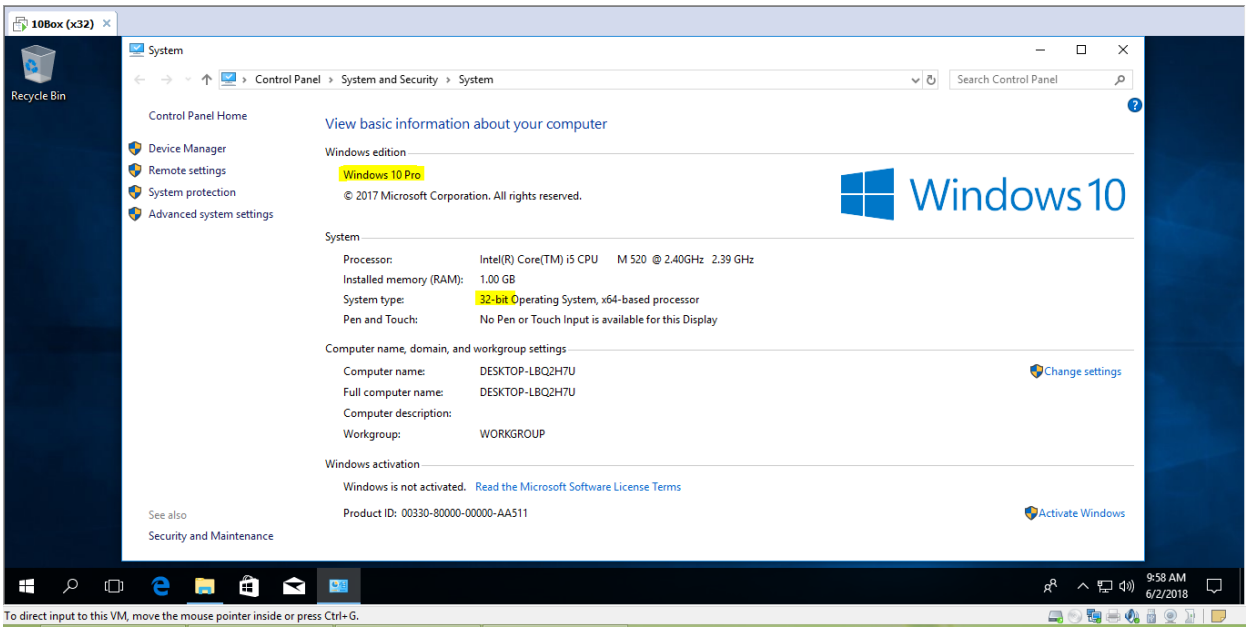
  2. Bản cài đặt Windows 10 (32-bit).



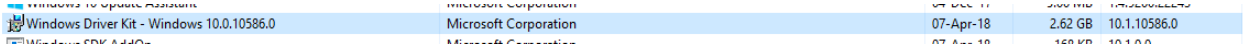
  3. Phần mềm Visual Studio 2015.



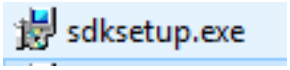
  4. Tạo một máy ảo Windows 10 (32-bit) bằng VMWare Workstation 14.

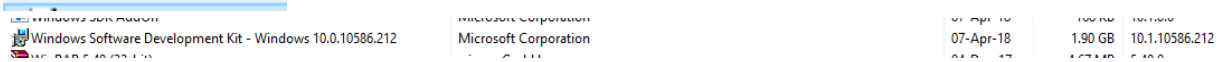


5. Cài đặt WDK phiên bản phù hợp với ảo Windows 10 .

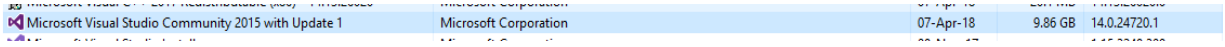


6. Cài đặt SDK cho window10:





7. Cài đặt Visual studio 2015 update 1 để phiên dịch:



- Khi thử nghiệm NetMon:

8. Sao chép các thư mục/tệp sau đây vào ổ C của máy ảo: "netmon_pdb", "netmon_guids.guid", "tracefmt.exe", "tracelog.exe", "tracepdb.exe", "OSRLOADER.exe".(*)

9. Biên dịch tệp "netmon.sln" sau đó sao chép các tệp "msnmntr.sys" và "monitor.exe" vào ổ C của máy ảo.(*)
10. Chạy chương trình "OSRLOADER.exe" và chọn tệp "msnmntr.sys" để tiến hành cài đặt trình điều khiển (driver).

11. Mở chương trình cmd.exe của Windows trong máy ảo bằng quyền Administrator, trỏ đến ổ C và thực hiện các lệnh để kiểm tra tính năng đã được cung cấp (có demo).

- Khi thử nghiệm Minispy:

12. Biên dịch tệp có tên là "minispy.sln", sau đó sao chép các tệp "minispy.sys" và "minispy.exe" và ổ C của máy ảo.(*).
13. Mở chương trình "cmd.exe" của Windows trong máy ảo bằng quyền Administrator, sau đó trỏ đến ổ C.
14. Gõ câu lệnh "net start minispy" để đăng kí driver.
15. Cuối cùng chạy câu lệnh minispy.exe .

- Khi thử nghiệm RegCtrl:

16. Biên dịch tệp "regfltr.sln" sau đó sao chép các tệp "regfltr.sys" và "regctrl.exe" vào ổ C của máy ảo.(*)
17. Mở chương trình cmd.exe của Windows trong máy ảo bằng quyền Administrator, trỏ đến ổ C và thực hiện các lệnh để kiểm tra tính năng đã được cung cấp (có demo).

(*): Các thư mục/tệp này đã được cung cấp trong đồ án

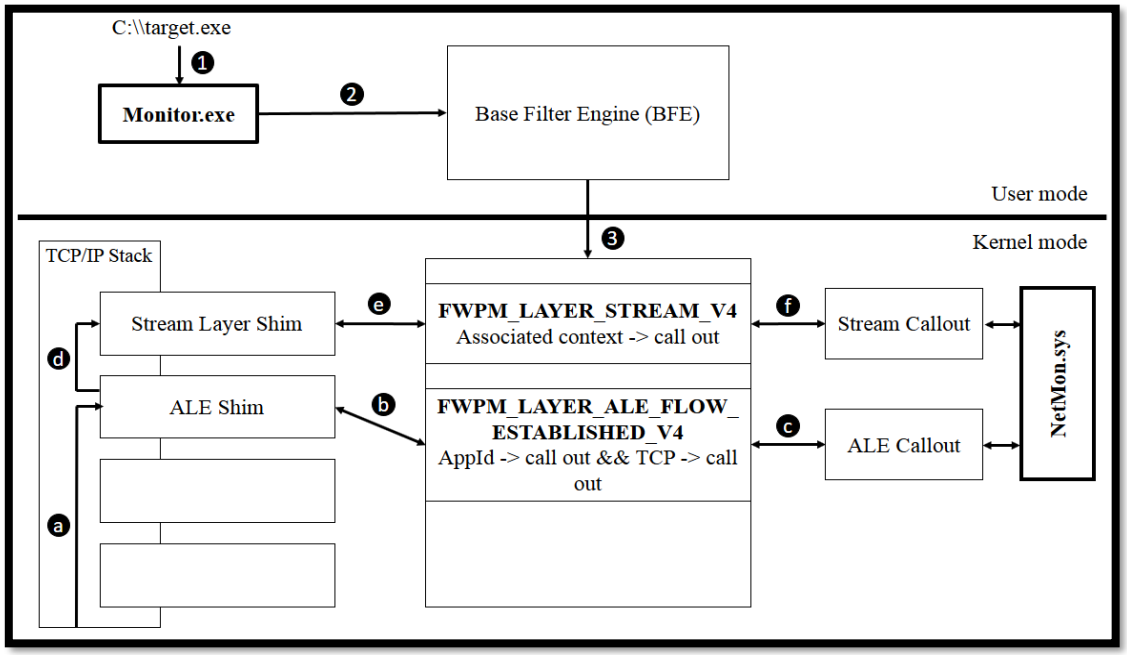
2. Network Monitor (NetMon)

- Giới thiệu chung:

Hầu hết các sandbox nổi tiếng, chẳng hạn như: Cuckoo hoặc H2Sandbox đều có mô-đun giám sát mạng riêng. Cuckoo sử dụng "tcpdump" để lắng nghe và ghi lại tất cả lưu lượng mạng, được tạo ra bởi các phần mềm độc hại đang được phân tích, qua NIC (card mạng) của các máy ảo. Còn LSP (Layered Service Provider) được sử dụng bởi H2Sandbox để chèn một lớp lên trên các API socket của Windows, vì vậy khi các phần mềm độc hại cố gắng gọi các API này đều bị bắt lại tại lớp này, được phân tích và ghi lại và sau đó chúng được chuyển tiếp tới các API thực để hoạt động mạng bình thường.

Những kỹ thuật này hoạt động rất tốt trên Windows XP và Windows 7, bởi vì không có lưu lượng truy cập nền nào được tạo ra bởi hệ điều hành cùng lúc với phần mềm độc hại. Tuy nhiên, làm việc trên Windows 10 là một thách thức lớn đối với 10Box, do sự đa dạng của lưu lượng mạng - đi ra ngoài và trong các ứng dụng ngầm hoặc Windows 10 Update. Bên cạnh đó, LSP không còn được hỗ trợ bởi Microsoft do tuổi của kỹ thuật này, và việc chuyển đổi các phiên bản Windows mới. Một kỹ thuật tiên tiến được gọi là WFP (Windows Filtering Platform) là một sự thay thế rất xứng đáng cho LSP, bởi vì nó cung cấp nhiều API hơn và cho phép các nhà phát triển làm nhiều thứ hơn LSP. Đó là lý do tại sao kỹ thuật mới để giám sát mạng trên Windows 10 phải được phát triển, có khả năng nắm bắt tất cả lưu lượng truy cập được tạo ra bởi phần mềm độc hại đang được phân tích hoặc tệp thực thi, vì vậy nó không chỉ giúp giảm thiểu thiếu sót khi giám sát lưu lượng truy cập của phần mềm độc hại và hệ điều hành, nhưng cũng hoạt động hiệu quả hơn trong môi trường Windows mới. Module mới này được gọi là NetMon (Mô-đun giám sát mạng).

- Kiến trúc của NetMon:



Hình 1: Kiến trúc ủa NetMon

Hai khung nhỏ được in đậm đại diện cho các thành phần của NetMon

- "Monitor.exe" là một ứng dụng được thực thi qua các dòng lệnh, khi trỏ đến các tệp đích (mã độc hoặc tệp thực thi), ứng dụng định danh để xử lý, thêm chú thích vào bộ lọc hoặc hủy định danh, xóa chú thích sau khi xử lý xong. Nó chuyển đổi tất cả các bản ghi vào biểu mẫu có thể đọc được.
- Trình điều khiển chú thích được gọi là "NetMon.sys" chịu trách nhiệm lọc lưu lượng TCP được tạo bởi tệp đích và ghi chúng vào tệp nhật ký dạng văn bản.

- Nguyên tắc hoạt động của NetMon:

- **Nguyên tắc 1:**

a. Đầu vào của ứng dụng monitor.exe là một đường dẫn đến các tệp thực thi cần được phân tích (ví dụ: "C:\target.exe")

b. Những định danh và cấu hình của monitor.exe:

o Ba hàm chú thích:

- 1 hàm "Stream callout" ở lớp FWPM_LAYER_STREAM_V4. Việc đọc và ghi dữ liệu ở lớp này dễ dàng hơn so với các lớp khác vì tất cả các tiêu đề của lớp thấp hơn đã bị xóa hoàn toàn.
- 2 hàm "ALE callout" ở lớp FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4. Hàm đầu tiên chịu trách nhiệm thêm nội dung vào lưu lượng TCP được tạo ra từ tệp đích, trong khi hàm thứ hai thực hiện cùng một công việc, nhưng chỉ cho các tiến trình con sinh ra bởi tệp đích.

o Ba bộ lọc:

- 1 bộ lọc ở lớp FWPM_LAYER_STREAM_V4, gọi hàm "Stream callout" để ghi lưu lượng truy cập có nội dung đã được thêm bởi hàm "ALE callout".
- 2 bộ lọc ở lớp FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4 chịu trách nhiệm xác minh lưu lượng được theo dõi.

- **Nguyên tắc 2: Lưu lượng truy cập đến từ Internet (lưu lượng truy cập đến)**

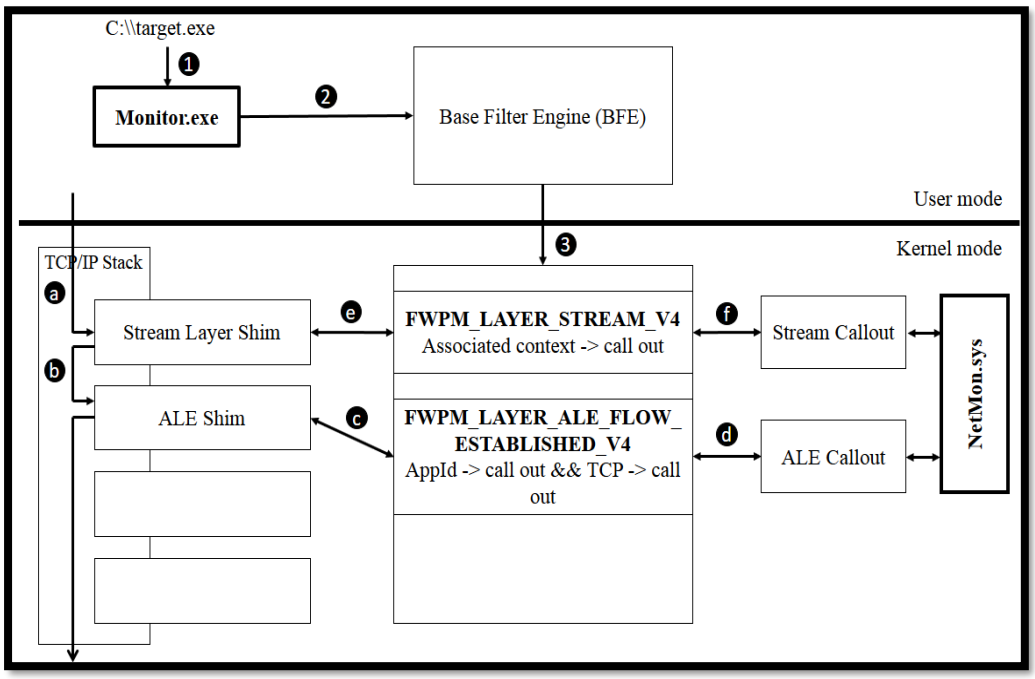
a. Bởi vì các bộ lọc đã được thiết lập ở các lớp FWPM_LAYER_STREAM_V4 và FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4, do đó hai shim đầu tiên được thông qua, lưu lượng sẽ chuyển sang "ALE shim".

b. "ALE shim" trích xuất metadata của lưu lượng truy cập và gửi đến công cụ lọc để kiểm tra ở lớp FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4.

c. Nếu metadata của lưu lượng phù hợp với bất kỳ điều kiện bộ lọc nào được chỉ định trong mỗi bộ lọc, chức năng "ALE callout" sẽ được gọi. Các hàm này thêm nội dung (thông tin bổ sung) cho lưu lượng truy cập để thông báo cho lớp FWPM_LAYER_STREAM_V4 xử lý khi nó nhìn thấy lưu lượng này. Tiếp theo, quyết định "continue" được trả về cho "ALE shim", cho phép nó chuyển lưu lượng truy cập đến "Stream Layer Shim". Nếu tiến trình thêm nội dung không thành công hoặc metadata của lưu lượng truy cập không phù hợp với bất kỳ điều kiện lọc nào, quyết định "continue" cũng được trả về vì quá trình chỉ giám sát lưu lượng truy cập, không có hành động chặn hoặc sửa đổi nào được thực hiện tại đây.

d. e,f. Cũng tương tự, công cụ lọc nhận metadata của lưu lượng truy cập từ "Stream Layer Shim". Nếu lưu lượng truy cập đi kèm với nội dung đã được thêm vào ở lớp FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4, hàm "Stream callout" được gọi, lưu lượng truy cập được ghi vào tệp dạng văn bản và quyết định "tiếp tục" được trả về cho "Shim Layer Shim", cho phép lưu lượng truy cập di chuyển đến các lớp cao hơn. Nếu quá trình thất bại, quyết định "tiếp tục" cũng được trả về để cho lưu lượng truy cập đi mà không có bất kỳ hành động ghi nhật ký nào.

- **Nguyên tắc 3: Lưu lượng truy cập ra Internet (lưu lượng gửi đi)**



Hình 2: Lưu lượng đi

a. b. Khi một kết nối TCP đã được thiết lập bởi tệp đối tượng, "Stream Layer Shim" không được gọi, nó chỉ đơn giản cho phép lưu lượng đi qua.

b. c,d. Khi ACK cuối cùng của việc bắt tay 4 chiều đã được gửi đi, các hàm "ALE callout" được gọi để thực hiện công việc thêm nội dung cho lưu lượng. (Lớp FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4 là hợp lệ sau khi nhìn thấy ACK cuối cùng của quá trình bắt tay 4 chiều).

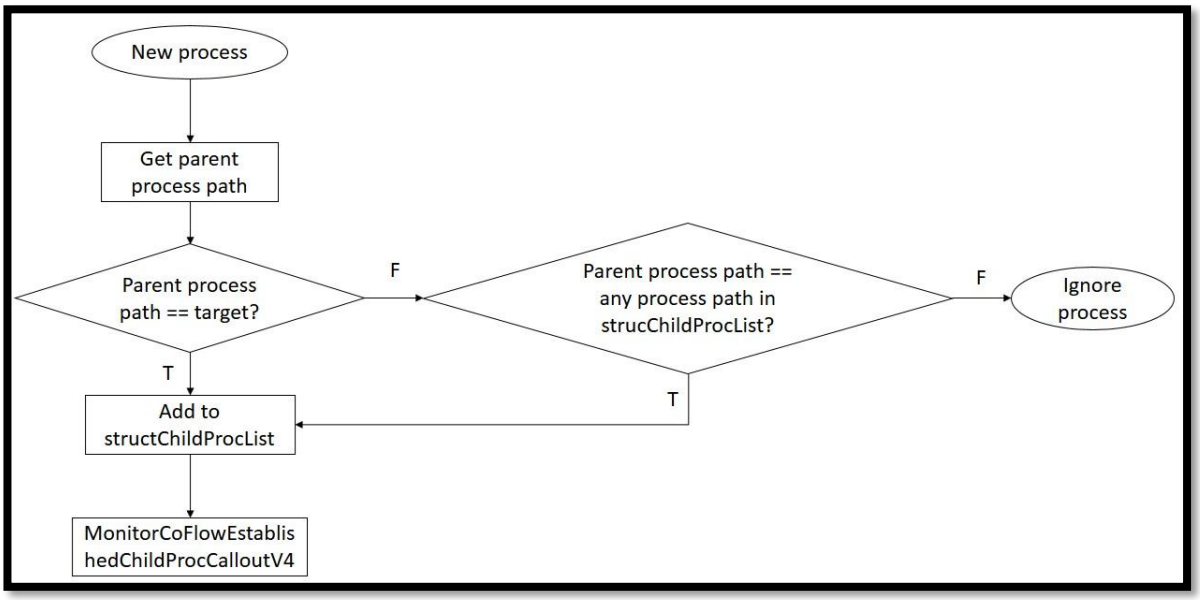
e. f. Sau khi thêm nội dung cho gói đầu tiên, tất cả các gói tiếp theo có cùng bộ dữ liệu bao gồm năm metadata: địa chỉ IP nguồn, địa chỉ IP đích, cổng nguồn, cổng đích và giao thức được coi là trong cùng một luồng. Do đó, chức năng "Stream callout" được gọi để quản lý lưu lượng truy cập này.

Tóm lại, NetMon hoạt động như thế nào với giao thức TCP / IP đều đã được giải thích chi tiết. Tiếp theo, tất cả các chức năng quan trọng trong mã nguồn của NetMon sẽ được trình bày để cung cấp thêm thông tin về chi tiết của NetMon.

- Giám sát tiến trình con:

Theo như dự kiến, tất cả lưu lượng truy cập được tạo bởi tệp đích, được nhập vào monitor.exe, được theo dõi và ghi lại vào tệp dạng văn bản. Điều này cũng có nghĩa là chỉ giám sát quá trình đã được tạo ra bởi chính tệp đích. Tuy nhiên, mã độc luôn cố giấu chúng ta, nó làm bất cứ điều gì để cho quá trình phân tích trở nên khó khăn hơn, chẳng hạn như: sinh ra nhiều tiến trình con hơn, tạo các tệp thực thi mới để thực hiện các công việc của nó thay vì tệp gốc đang được phân tích. Do đó, NetMon sẽ có khả năng không chỉ giám sát tệp đích, mà còn tất cả các tiến trình con của nó được sinh ra trong quá trình thực hiện.

Thuật toán được trình bày dưới đây hỗ trợ NetMon để giám sát các tiến trình con được sinh ra:



Hình 3. Thuật toán bổ sung tiến trình con

NetMon theo dõi tất cả các tiến trình con mới được sinh ra sau khi cài đặt trình điều khiển NetMon.sys bằng cách sử dụng hàm gọi lại SetCreateProcessNotifyRoutineEx [reference needed].

Với mỗi tiến trình mới, NetMon trích xuất đường dẫn của tiến trình gốc, sau đó so sánh nó với đường dẫn của đối tượng đã nhập trong monitor.exe. Nếu bằng nhau, tiến trình mới chắc chắn là tiến trình con của đối tượng, sau đó nó được thêm vào "structChildProcList" - danh sách lưu trữ tất cả các đường dẫn đang được giám sát. Nếu không bằng nhau, NetMon sẽ so sánh giữa đường dẫn của tiến trình gốc với từng thành viên trong structChildProcList để kiểm tra xem nó có thuộc về bất kỳ tiến trình nào trong danh sách hay không.

Nếu tất cả các so sánh không phù hợp, tiến trình mới sẽ được bỏ qua và tiếp tục chạy như bình thường.

Tất cả các tiến trình con trong structChildProcList đều được giám sát bởi hàm chú thích MonitorCoFlowEstablishedChildProcCalloutV4.

- Chức năng ghi nhật ký:

NetMon.sys sử dụng bộ truy tìm WPP để ghi lưu lượng truy cập. Bộ truy tìm WPP là một bản cập nhật lớn của bộ WMI, được sử dụng để ghi các sự kiện bằng

định dạng nhị phân, và chuyển đổi chúng thành dạng văn bản. Vì vậy, nó rất hữu ích để gỡ lỗi và có thể được coi là một sự thay thế cho DbgPrint, bởi vì WPP dễ sử dụng và cung cấp nhiều chức năng hơn.

WPP bao gồm bộ cung cấp dấu vết và theo dõi sự tiêu thụ. Bộ cung cấp dấu vết chính là trình điều khiển NetMon vì nó tạo ra các sự kiện (hoặc các nhật ký được ghi lại). Tất cả các thông tin này chỉ được hiển thị trong TraceView hoặc Tracelog, được gọi là dấu vết tiêu thụ.

Xác định GUID là bắt buộc khi sử dụng WPP:

```
#define WPP_CONTROL_GUIDS \
    WPP_DEFINE_CONTROL_GUID(MsnMntrNotify,(65BDB7FF,7C71,44CF,92D9,54F5B844747A),   \

        WPP_DEFINE_BIT(TRACE_CLIENT_SERVER)                \
        WPP_DEFINE_BIT(TRACE_PEER_TO_PEER)                 \
        WPP_DEFINE_BIT(TRACE_UNKNOWN)                      \
        WPP_DEFINE_BIT(TRACE_ALL_TRAFFIC) )
```

GUID được tạo ra thông qua công cụ "Create GUID", được xây dựng trong Visual Studio. Theo sau GUID là 4 macro WPP_DEFINE_BIT (), được gọi là cờ. Cờ được sử dụng để lọc thông điệp.

Hàm này giống với DbgPrint, nhưng hỗ trợ việc sử dụng cờ. Ví dụ:
`DoTraceMessage(TRACE_CLIENT_SERVER, "r3: notify.c: MonitorNfInitialize");`

Hàm trên in ra thông báo "r3: notify.c: MonitorNfInitialize" với cờ TRACE_CLIENT_SERVER. Điều đó có nghĩa là khi thiết lập theo dõi lượng tiêu dùng để lọc cờ TRACE_CLIENT_SERVER, sẽ thấy thông báo. Có 4 cấp độ cờ, được sử dụng để in 4 loại thông điệp. NetMon.sys sử dụng cờ TRACE_CLIENT_SERVER để in thông tin gỡ lỗi và TRACE_PEER_TO_PEER để in và ghi lưu lượng truy cập như sau:

```
if (!Inbound)
   {
          if ((getData != NULL))
          {
                DoTraceMessage(TRACE_PEER_TO_PEER, "Request (%d): %s", streamLength,
(const char*)getData);


          }
                if ((postData != NULL))
                DoTraceMessage(TRACE_PEER_TO_PEER, "Request (%d): %s", streamLength,
(const char*)postData);
   }
   else
   {
          if ((httpData != NULL))
                DoTraceMessage(TRACE_PEER_TO_PEER,     "Response     (%d):     %s",
streamLength, (const char*)httpData);
   }
```

Do đó, đối với mỗi lưu lượng phù hợp, hàm DoTraceMessage ghi lại payload của nó vào tệp.

Tracelog được chọn để ghi lại tất cả thông báo được tạo bởi hàm DoTraceMessage vào một tệp. Tracelog được bắt đầu sau khi tất cả các chú thích đã được định danh (trong monitor.exe) bằng cách sử dụng lệnh:
“c:\tracelog.exe -start netmon2 -guid "c:\netmon_guids.guid" -f "c:\netmon_log.etl" -flag 2”.

```
if(argc == 3)
	{
		if (_wcsicmp(argv[1], L"monitor") == 0)
		{
			// Add callouts
			MonitorAppAddCallouts();

			// Start to monitor
			dStatus = MonitorAppDoMonitoring(argv[2]);

			// Start tracing
			printf("\n\nStarting tracelog...\n\n");
			StartTraceview(L"c:\\tracelog.exe -start netmon2 -guid
\"c:\\netmon_guids.guid\" -f \"c:\\netmon_log.etl\" -flag 2");

			return dStatus;

		}
	}
```

Các lệnh trên có nghĩa là: bắt đầu tracelog với một phiên mới "netmon2" và sử dụng GUID tại c:\netmon.guids.guid, sau đó tất cả đầu ra được đặt trong c:\netmon_log.etl. Gắn cờ 2 có nghĩa là chỉ lọc thông điệp có cờ TRACE_PEER_TO_PEER.

Vì nhật ký được ghi ở dạng nhị phân nên cần thêm một bước nữa để chuyển đổi thành định dạng văn bản để đọc dễ dàng hơn. Để làm điều đó, NetMon.sys sử dụng tracefmt.exe để phân tích cú pháp nhị phân thành một tệp văn bản sau khi tất cả các chú thích đã bị xóa (trong chức năng MonitorAppProcessArguments).

```
if ( wcslcmp(argv[1], L"delcallouts") == 0)
            {
                  dStatus = MonItorAppRemoveCallouts();

                  // Stop tracing
                  printf("\n\nStopping tracelog...\n\n");
                  StartTracevIew(L"c:\\tracelog.exe -stop netmon2");

                  // Generate log file
                  printf("\n\nStarting tracefmt...\n\n");
                  StartTracevIew(L"c:\\tracefmt.exe     c:\\netmon_log.etl     -p
c:\\netmon_pdb -o c:\\netmon_log.txt");

                  return dStatus;
            }
```

Lệnh của tracefmt như sau:
"c:\tracefmt.exe c:\netmon_log.etl -p c:\netmon_pdb -o c:\netmon_log.txt"

Có nghĩa là chạy tracefmt.exe để phân tích cú pháp tệp nhật ký tại c:\netmon_log.etl, bằng cách sử dụng tệp pdb tại c:\netmon_pdb, sau đó lưu tệp văn bản đầu ra tại c:\netmon_log.txt (tệp pdb chứa tất cả thông tin về WPP , được tạo tự động khi biên dịch trình điều khiển).

Hàm StartTraceView sử dụng hàm API "_wsystem" để chạy các lệnh như thường làm trong cmd.exe.

```
void StartTracevIew(const wchar_t *cmd)
{
	Int IStatus;
	errno_t err;
	Int errCode;

	// Flush all streams
	_flushall();

	// Start tracevIew to log traffIc
	IStatus = _wsystem(cmd);

	// Check If the _wsystem() Is falIed
	if (IStatus == -1)
	{
		err = _get_errno(&errCode);
		prIntf("_wsystem falIed = %d\n", err);
	}
}
```

Cuối cùng, sau khi giám sát lưu lượng được tạo bởi tệp đích, tệp nhật ký văn bản được lưu trữ tại c:\netmon_log.txt, chứa tất cả lưu lượng truy cập để phân tích sau này.

3. File System Monitor (Minispy)

- Một số đặc trưng của mô hình I/O.

Mô hình quản lý I/O đưa ra một giao diện thống nhất cho tất cả driver ở mức độ kernel, bao gồm ở mức thấp nhất , mức trung gian và hệ thống tệp tin driver. Tất cả các yêu cầu I/O đến các drivers đều được thực hiện qua 1 gói tin yêu cầu có tên IRPs-I/O (request packets).

Mô hình I/O chia thành 2 lớp sau:

* I/O System: Chịu trách nhiệm giao tiếp với phần cứng:

Là một tập hợp các thành phần mức kernel quản lý các drivers thiết bị chạy trong hệ thống và thông tin liên lạc giữa các ứng dụng và drivers thiết bị.

I/O System cùng với drivers thiết bị cho phép ứng dụng giao tiếp với drivers và thực hiện các yêu cầu từ thiết bị. Các I/O System có trách nhiệm chuyển tiếp yêu cầu từ ứng dụng đến drivers thiết bị chịu trách nhiệm để thực hiện.

I/O System cũng được phân chia thành các lớp. Điều này có nghĩa là mỗi thiết bị có thể tương tác với nhiều drivers thiết bị bằng việc xếp chồng các drivers thiết bị. Điều này cho phép tạo ra một drivers chung, sử dụng để quản lý dữ liệu trên thiết bị.

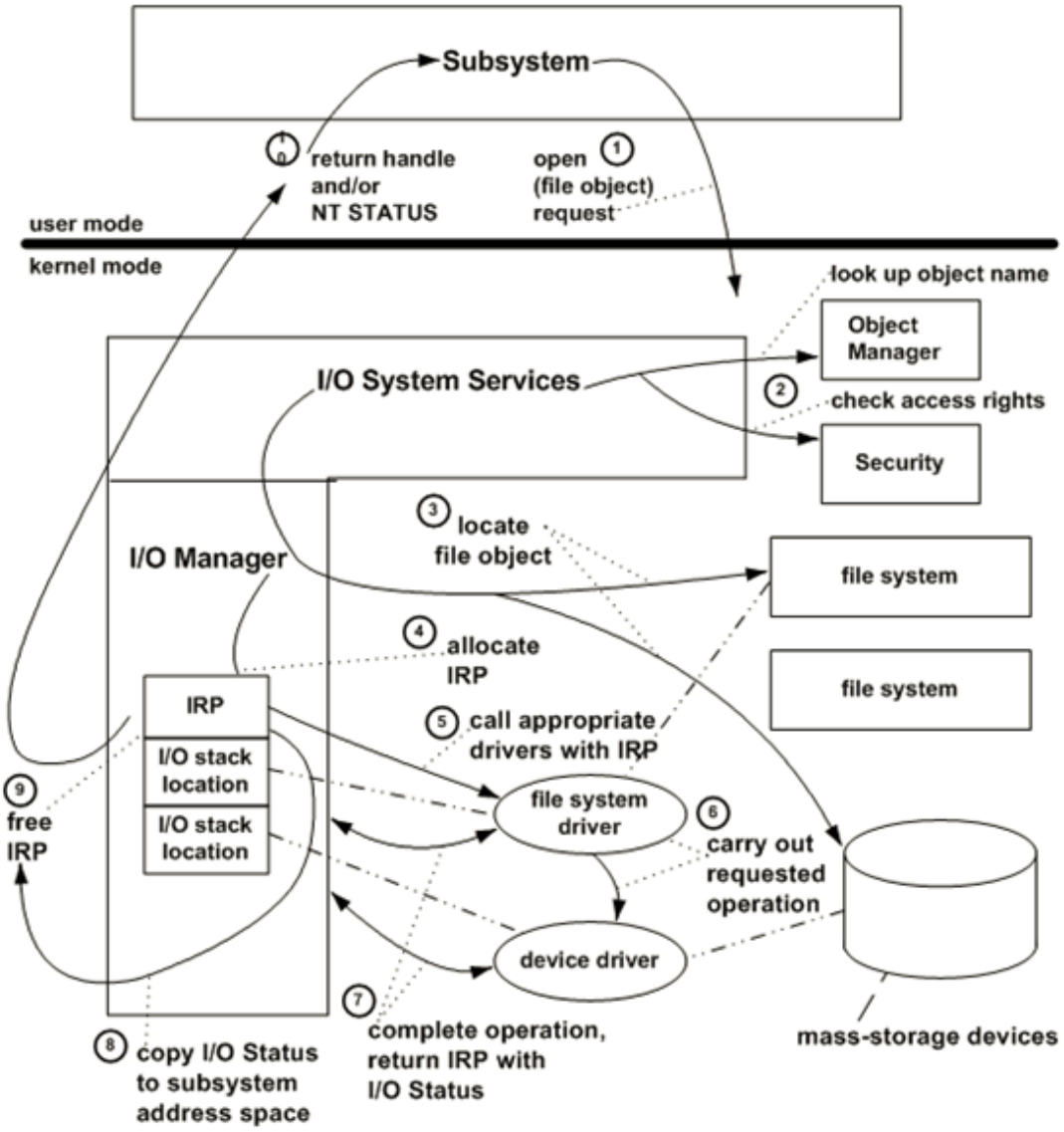
Chính vì điều này mà chúng ta có thể dễ dàng thêm các driver bọ lọc đề giám sát hoặc sửa đổi thông tin liên lạc giữa các driver và các ứng dụng.

* I/O Subsystem: Chịu trách nhiệm với các thực thi có liên quan đến đồ họa và xử lý các thông tin nhập vào của người dùng.

I/O Subsystem (hay Win32 Subsystem) là thành phần chịu trách nhiệm về mọi khía cạnh Giao diện người dùng. Bao gồm từ những engine đồ họa ở mức thấp, giao diện đồ họa thiết bị (GDI – graphics device interface), và các thành phần ở mức User như cửa sổ, menu,… hay xử lý thông tin người dùng đưa vào.

- Chu trình của một yêu cầu I/O.

Hình minh họa dưới đây cho chúng ta một cái nhìn tổng quan về những gì sẽ xảy ra khi một Subsystem mở một đối tượng tệp tin, đại diện cho tệp tin dữ liệu trong ứng dụng. (Nguồn Microsoft).



Hình 4: Mô hình khi một Subsystem mở một đối tượng tệp tin

Các bước thực hiện:

1. Subsystem gọi một dịch vụ của I/O System để mở một tệp tin đã có tên.
2. Thành phần quản lý I/O gọi đến thành phần quản lý đối tượng (Object Manager) để tìm kiếm các tệp tin có tên và giúp giải quyết bất kỳ liên kết nào cho đối tượng tệp tin (file object). Nó cũng gọi đến các tham chiếu kiểm tra xem Subsystem có chính xác quyền để mở đối tượng tệp tin không.
3. Nếu Volume – khu vực lưu trữ truy cập đơn lẻ – chưa được gắn vào, thành phần quản lý I/O sẽ tạm ngừng yêu cầu mở tệp tin. Thay vào đó là việc gọi đến các hệ thống tệp tin khác cho đến khi xác định được rằng tệp tin cần mở được

lưu trữ trong một thiết bị lưu trữ nào đó. Khi Volume được gắn, thành phần quản lý I/O sẽ tiếp tục lại yêu cầu.

4. Thành phần quản lý I/O cấp phát vùng nhớ và khởi tạo một IRP cho yêu cầu mở tệp tin. Đối với drivers, yêu cầu mở tương đương với yêu cầu khởi tạo.

5. Thành phần quản lý I/O gọi driver của hệ thống tệp tin thông qua các IRP. Các driver của hệ thống tệp tin truy cập I/O Stack Location để xác định hoạt động phải thực hiện và kiểm tra các thông số.

6. Các driver xử lý các IRP và hoàn thiện hoạt động I/O được yêu cầu. Thành phần quản lý I/O và các thành phần khác của hệ điều hành cung cấp việc gọi đến các thủ tục hỗ trợ ở kernel mode.

7. Các driver sẽ trả IRP về cho thành phần quản lý I/O để thiết đặt trạng thái vào/ra của yêu cầu là thành công hay lí do thất bại.

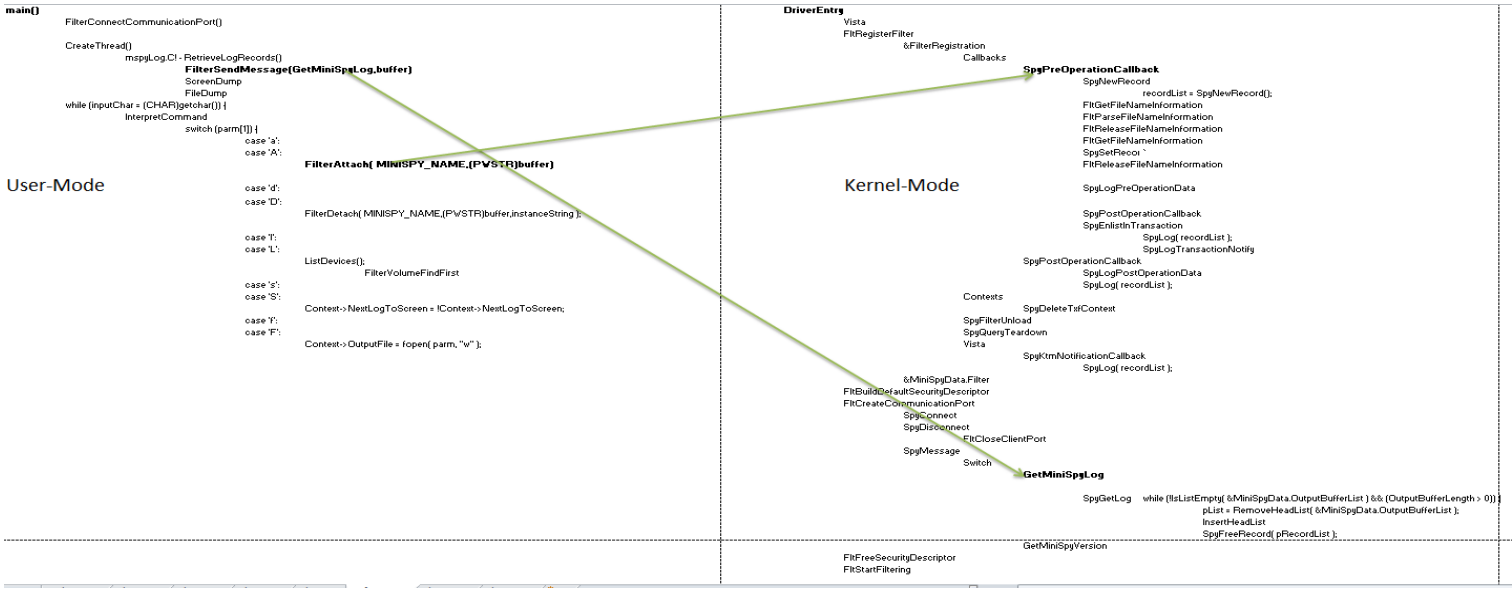
8. Thành phần quản lý I/O lấy được trạng thái vào/ra từ IRP, nên nó có thể trả về thông tin trạng thái thông qua Subsystem được bảo vệ tới lời gọi ban đầu.

9. Thành phần quản lý I/O sẽ giải phóng IRP khi đã hoàn thành.

10. Nếu thành công, thành phần quản lý I/O sẽ trả về một handle cho đối tượng tệp tin. Còn nếu có lỗi, nó sẽ trả về trạng thái thích hợp cho Subsystem.

Sau khi mở thành công một đối tượng tệp tin, Subsystem sẽ sử dụng handle được trả về để xác định đối tượng tệp tin trong yêu cầu tiếp theo cho các hoạt động như đọc, ghi, hay yêu cầu điều khiển của thiết bị vào/ra. Điều này được thực hiện bằng việc Subsystem gọi đến dịch vụ của I/O System. Thành phần quản lý I/O sẽ định tuyến các yêu cầu này tương tự các IRP đến các driver thích hợp gọi 1 dịch vụ của I/O System để mở 1 tệp tin đã có tên.

- Cơ chế hoạt động của Monitor file system.




```
NTSTATUS
DriverEntry (
    _In_ PDRIVER_OBJECT DriverObject,
    _In_ PUNICODE_STRING RegistryPath
    )
{

   status = FltRegisterFilter( DriverObject,

                                   &FilterRegistration,
                                   &MiniSpyData.Filter );
  if (!NT_SUCCESS( status )) {
          leave;
      }

  status  = FltBuildDefaultSecurityDescriptor( &sd,FLT_PORT_ALL_ACCESS );

       if (!NT_SUCCESS( status )) {
           leave;
       }

  status = FltCreateCommunicationPort( MiniSpyData.Filter,
                                       &MiniSpyData.ServerPort,
                                       &oa,
                                       NULL,
                                       SpyConnect,
                                       SpyDisconnect,
                                       SpyMessage,
                                       1 );

       FltFreeSecurityDescriptor( sd );

       if (!NT_SUCCESS( status )) {
           leave;
       }
```

Kernel-Mode

- Bất kì minifilter driver nào cũng gọi FltRegisterFilter trong chương trình DriverEntry để thêm chính nó vào danh sách các minifilter drivers đã được đăng ký trước đó, cung cấp cho Filter Manager danh sách các hàm callback cũng như là 1 số thông tin khác về minifilter driver.
- Tiếp theo chúng ta tiến hành thiết lập port để truyền thông, giao tiếp. Nhưng trước hết chúng ta phải thiết lập 1 kênh truyền thông an toàn bằng cách gọi hàm FltBuildDefaultSecurityDescriptor(). Hàm này có mục đích cho phép tất cả user sử dụng truyền thông trên 1 port giao tiếp, ở đây có 1 tham số là FLT_PORT_ALL_ACCESS. Nghĩa là các chương trình user có thể kết nối đến port để truyền thông. Sau khi kết nối thì user có thể thực hiện bất cứ hoạt động nào mà không bị chăn.
- FltCreateCommunicationPort định nghĩa 3 hàm được yêu cầu cho port này. 3 hàm này là:

SpyConnect: user-mode và kernel-mode thiết lập kết nối khi gọi hàm này.

```
NTSTATUS
SpyConnect(
    _In_ PFLT_PORT ClientPort,
    _In_ PVOID ServerPortCookie,
    _In_reads_bytes_(SizeOfContext) PVOID ConnectionContext,
    _In_ ULONG SizeOfContext,
    _Flt_ConnectionCookie_Outptr_ PVOID *ConnectionCookie
    )

{    PAGED_CODE();

    UNREFERENCED_PARAMETER( ServerPortCookie );
    UNREFERENCED_PARAMETER( ConnectionContext );
    UNREFERENCED_PARAMETER( SizeOfContext);
    UNREFERENCED_PARAMETER( ConnectionCookie );
    FLT_ASSERT( MiniSpyData.ClientPort == NULL );
    MiniSpyData.ClientPort = ClientPort;
    return STATUS_SUCCESS;
}
```

SpyDisconnect: Khi user-mode và kernel-mode disconnected thì gọi hàm này.

```
VOID
SpyDisconnect(    _In_opt_ PVOID ConnectionCookie   )
{

    PAGED_CODE();
    UNREFERENCED_PARAMETER( ConnectionCookie );
    FltCloseClientPort( MiniSpyData.Filter, &MiniSpyData.ClientPort );
}
```

SpyMessage: user-mode và kernel-mode truyền thông với nhau thì gọi hàm này.

```
NTSTATUS
SpyMessage (
    _In_ PVOID ConnectionCookie,
    _In_reads_bytes_opt_(InputBufferSize) PVOID InputBuffer,
    _In_ ULONG InputBufferSize,
    _Out_writes_bytes_to_opt_(OutputBufferSize,*ReturnOutputBufferLength) PVOID OutputBuffer,
    _In_ ULONG OutputBufferSize,
    _Out_ PULONG ReturnOutputBufferLength
    )

{

switch (command) {
            case GetMiniSpyLog:

                if ((OutputBuffer == NULL) || (OutputBufferSize == 0)) {

                    status = STATUS_INVALID_PARAMETER;
                    break;
                }

                     status = SpyGetLog( OutputBuffer,
                                         OutputBufferSize,
                                         ReturnOutputBufferLength );

                break;
            case GetMiniSpyVersion:
                if ((OutputBufferSize < sizeof( MINISPYVER )) ||
                    (OutputBuffer == NULL)) {
                    status = STATUS_INVALID_PARAMETER;
                    break;
                }
                if (!IS_ALIGNED(OutputBuffer,sizeof(ULONG))) {
                    status = STATUS_DATATYPE_MISALIGNMENT;
                    break;
                }

}
```

- GetMiniSpyLog: Lây ra tất cả các log record
- GetMiniSpyVersion: Trả về version của filter driver.

Tất cả những hàm trên đều là callback function, vì thế mà không thể gọi chúng trong chương trình user. Vậy trên user chúng ta cần sử dụng 2 APIs đó là: FilterConnectCommunicationPort () và FilterSendMessage

User-mode:

```
int _cdecl
main (
    _In_ int argc,
    _In_reads_(argc) char *argv[]
    )
{

printf( "Connecting to filter's port...\n" );

    hResult = FilterConnectCommunicationPort( MINISPY_PORT_NAME,
                                              0,
                                              NULL,
                                              0,
                                              NULL,
                                              &port );

    if (IS_ERROR( hResult )) {

        printf( "Could not connect to filter: 0x%08x\n", hResult );
        DisplayError( hResult );
        goto Main_Exit;
    }

}
```

FilterConnectCommunicationPort: Mở 1 connection đến server port đã được tạo bởi file system minifilter.

```
int _cdecl
main (
    _In_ int argc,
    _In_reads_(argc) char *argv[]
    )
{

       ........

       pintf( "Creating logging thread...\n" );

       thread = CreateThread( NULL,

                         0,
                           RetrieveLogRecords,
                           (LPVOID)&context,
                           0,
                           &threadId);

        if (!thread) {
        result = GetLastError();
        printf( "Could not create logging thread: %d\n", result );
        DisplayError( result );
        goto Main_Exit;
       ..........
    }
```

CreateThread() Tạo ra 1 thread để lắng nghe và đọc các log records mà hàm RetrieveLogRecords() trả về từ MiniSpy.sys.

```
DWORD
WINAPI
RetrieveLogRecords(
    _In_ LPVOID lpParameter
    )

{      ........

        hResult = FilterSendMessage( context->Port,
                                     &commandMessage,
                                     sizeof( COMMAND_MESSAGE ),
                                     buffer,
                                     sizeof(alignedBuffer),
                                     &bytesReturned );

      ..........
    }

}
```

Trong hàm RetrieveLogRecords có sử dụng 1 hàm tên là FilterSendMessage() hàm này có mục đích là gửi 1 message đến kernel-mode minifilter thông qua port đã được khai báo trong tham số context->Port, Tin nhắn này sẽ được truyền xuống các chương trình callback của minifilter. Tuy vào input của tham số &commandMessage mà kernel sẽ gọi tới callback nào để thực hiện. Sau khi thực hiện xong thì biến buffer sẽ nhận dữ liệu trả về của minifilter. Buffer này là 1 mảng các LOG_RECORD structures được định nghĩa bởi Microsoft.

```
typedef struct _LOG_RECORD {

    ULONG Length;           // Length of log record.  This Does not include
    ULONG SequenceNumber;   // space used by other members of RECORD_LIST

    ULONG RecordType;       // The type of log record this is.
    ULONG Reserved;         // For alignment on IA64

    RECORD_DATA Data;
    WCHAR Name[];           //  This is a null terminated string

} LOG_RECORD, *PLOG_RECORD;
```

Tiếp theo là danh sách các tham số mà user có thể input vào :

```
DWORD
InterpretCommand (    _In_ int argc,    _In_reads_(argc) char *argv[],    _In_ PLOG_CONTEXT
Context    )
{for (parmIndex = 0; parmIndex < argc; parmIndex++) {

        parm = argv[parmIndex];

        if (parm[0] == '/') {
            switch (parm[1]) {
            case 'a':
            case 'A':
                parmIndex++;
                if (parmIndex >= argc) {
                    goto InterpretCommand_Usage;
                }
                parm = argv[parmIndex];
                printf( "    Attaching to %s... ", parm );

                bufferLength = MultiByteToWideChar( CP_ACP,
                                                    MB_ERR_INVALID_CHARS,
                                                    parm,
                                                    -1,
                                                    (LPWSTR)buffer,
                                                    BUFFER_SIZE/sizeof( WCHAR ) );
                hResult = FilterAttach( MINISPY_NAME,
                                        (PWSTR)buffer,
                                        NULL, // instance name
                                        sizeof( instanceName ),
                                        instanceName );
                break;
```

Với tham số /a là chọn ổ đĩa cần moniter file system

- FilterAttach() gắn 1 minifilter vào 1 volume (ổ đĩa).

```
DWORD
InterpretCommand (    _In_ int argc,    _In_reads_(argc) char *argv[],    _In_ PLOG_CONTEXT
Context    )
{           case 'l':

            case 'L':

                ListDevices();

                break;

            case 's':
            case 'S':
                if (Context->NextLogToScreen) {
                    printf( "    Turning off logging to screen\n" );
                } else {
                    printf( "    Turning on logging to screen\n" );
                }
                Context->NextLogToScreen = !Context->NextLogToScreen;
                break;

            case 'f':
            case 'F':
                if (Context->LogToFile) {

                    printf( "    Stop logging to file \n" );
                    Context->LogToFile = FALSE;
                    assert( Context->OutputFile );
                    _Analysis_assume_( Context->OutputFile != NULL );
                    fclose( Context->OutputFile );
                    Context->OutputFile = NULL;

                } else {

                    parmIndex++;

                    if (parmIndex >= argc) {
                        goto InterpretCommand_Usage;
                    }
                    parm = argv[parmIndex];
                    printf( "    Log to file %s\n", parm );

                    if (fopen_s( &Context->OutputFile, parm, "w" ) != 0 ) {
                        assert( Context->OutputFile );
                    }

                    Context->LogToFile = TRUE;
                }
                break;

            default:
               goto InterpretCommand_Usage;
            }
```

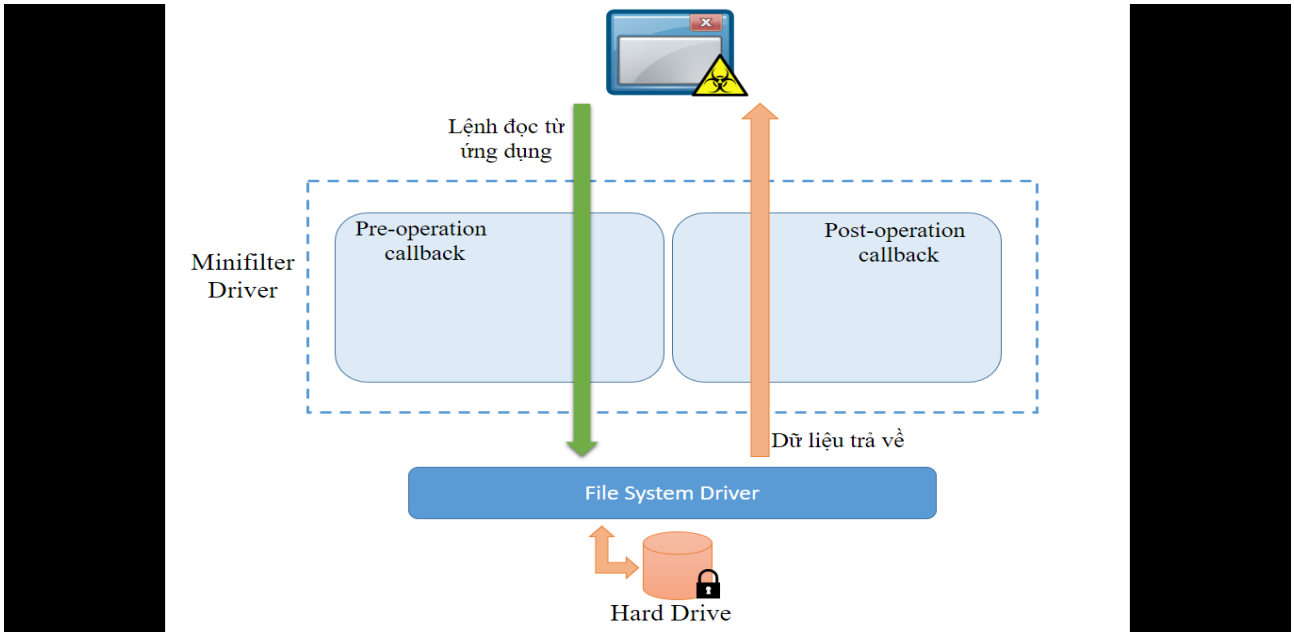
Trường hợp tham số truyền vào là /l hoặc /L : In ra danh sách tất cả các ổ đĩa đang được moniter file system, sử dụng hàm ListDevices().

```
void
ListDevices(     VOID     )
{

    hResult = FilterVolumeFindFirst( FilterVolumeBasicInformation,

                                          volumeBuffer,
                                          sizeof(buffer)-sizeof(WCHAR),
                                          &volumeBytesReturned,
                                          &volumeIterator );
```

FilterVolumeFindFirst() hàm này sẽ mở 1 xử lý tìm kiếm , sau đó trả về thông tin của volume đầu tiên trong danh sách các volumes mà filter manager quản lý. Sau khi xử lý được thiết lập thì chúng ta có thể sử dụng hàm FilterVolumeFindFirst () lại lần nữa để in ra volumes tiếp theo trong danh sách . Chính vì để in ra được tất cả danh sách thì cần có 1 vòng lặp.

- Trường hợp tiếp theo là /s hoặc /S: HIện thị các log ra màn hình console (/s) hay là không (/S).
- Trường hợp /f hoặc /F: Thực hiện việc ghi log vào file (/f) hay là không (/F).

Kernel-mode:



Hình 5: Mô hình thể hiện tiến trình đọc dữ liệu từ ổ đĩa.

- Pre-operation Callback.

```
SpyPreOperationCallback (
    _Inout_ PFLT_CALLBACK_DATA Data,
    _In_ PCFLT_RELATED_OBJECTS FltObjects,
    _Flt_CompletionContext_Outptr_ PVOID *CompletionContext
    )
{
       recordList = SpyNewRecord();
    if (recordList) {
        if (FltObjects->FileObject != NULL) {
            status = FltGetFileNameInformation( Data,
                                                FLT_FILE_NAME_NORMALIZED |
                                                MiniSpyData.NameQueryMethod,
                                                &nameInfo );

        } else {
            status = STATUS_UNSUCCESSFUL;
        }
   if (Data->Iopb->MajorFunction == IRP_MJ_CREATE) {

            RtlInitEmptyUnicodeString( &ecpData,
                                       ecpDataBuffer,
                                       MAX_NAME_SPACE/sizeof(WCHAR) );

            SpyParseEcps( Data, recordList, &ecpData );

            ecpDataToUse = &ecpData;
        }

       if (NULL != nameInfo) {

            FltReleaseFileNameInformation( nameInfo );
        }

  if (Data->Iopb->MajorFunction == IRP_MJ_SHUTDOWN) {
            SpyPostOperationCallback( Data,
                                      FltObjects,
                                      recordList,
                                      0 );

            returnStatus = FLT_PREOP_SUCCESS_NO_CALLBACK;

        } else {

            *CompletionContext = recordList;
            returnStatus = FLT_PREOP_SUCCESS_WITH_CALLBACK;
        }
```

Hàm này sẽ ghi log lại tất cả các hoạt động pre-operation callbacks truy xuất đến file system driver. FltGetFileNameInformation() Để truy xuất tên file trong lệnh hiện tại, minifilter sẽ gọi hàm này

+ Data: là cấu trúc FLT_CALLBACK_DATA của FileObject mà minifilter muốn truy xuất tên.

+ FLT_FILE_NAME_NORMALIZED | MiniSpyData.NameQueryMethod : Tên của File sẽ được trả về kèm theo đường dẫn tới file đó và tất cả các short name sẽ được chuyển thành long name.

+ Khi mà minifilter đã sử dụng xong thì nó sẽ giải phóng vùng nhớ của cấu trúc chứa thông tin tên File bằng cách gọi hàm FltReleaseFileNameInformation().

+ Tất cả pre-operation callbacks đều trả về FLT_PRE_OPERATION_CALLBACK_STATUS

* FLT_PREOP_SUCCESS_WITH_CALLBACK: Lệnh đã hoàn thành và minifilter muốn gọi post-operation callback.

* FLT_PREOP_SUCCESS_NO_CALLBACK: Lệnh đã hoàn thành nhưng minifilter không muốn gọi post-operation callback.

- Post-operation Callback.

```
FLT_POSTOP_CALLBACK_STATUS
SpyPostOperationCallback (
    _Inout_ PFLT_CALLBACK_DATA Data,
    _In_ PCFLT_RELATED_OBJECTS FltObjects,
    _In_ PVOID CompletionContext,
    _In_ FLT_POST_OPERATION_FLAGS Flags
    )
{

    UNREFERENCED_PARAMETER( FltObjects );
    recordList = (PRECORD_LIST)CompletionContext;
    if (FlagOn(Flags,FLTFL_POST_OPERATION_DRAINING)) {
        SpyFreeRecord( recordList );
        return FLT_POSTOP_FINISHED_PROCESSING;
   }
    SpyLogPostOperationData( Data, recordList );
    tagData = Data->TagData;
    if (tagData) {

        reparseRecordList = SpyNewRecord();

        if (reparseRecordList) {

            RtlCopyMemory( &reparseRecordList->LogRecord.Data,
                           &recordList->LogRecord.Data,
                           sizeof(RECORD_DATA) );

            reparseLogRecord = &reparseRecordList->LogRecord;

            copyLength = FLT_TAG_DATA_BUFFER_HEADER_SIZE + tagData->TagDataLength;

            if(copyLength > MAX_NAME_SPACE) {

                copyLength = MAX_NAME_SPACE;
            }
            RtlCopyMemory(
                &reparseRecordList->LogRecord.Name[0],
                tagData,
                copyLength
                );

            reparseLogRecord->RecordType |= RECORD_TYPE_FILETAG;
            reparseLogRecord->Length += (ULONG) ROUND_TO_SIZE( copyLength, sizeof( PVOID ) );
        }
    }
    SpyLog( recordList );
    if (reparseRecordList) {
        SpyLog( reparseRecordList );
    }

       if ((FltObjects->Transaction != NULL) &&
        (Data->Iopb->MajorFunction == IRP_MJ_CREATE) &&
        (Data->IoStatus.Status == STATUS_SUCCESS)) {
             SpyEnlistInTransaction( FltObjects );
    }

    return FLT_POSTOP_FINISHED_PROCESSING;

}
```

+ Chương trình này sẽ nhận tất cả các post-operation callbacks. Sau đó tiến hành ghi lại log thông qua các them số mà user input .

+ Nếu minifilter trả về FLT_PREOP_SUCCESS_WITH_CALLBACK từ pre-operation callback, nó đảm bảo rằng chỉ nhận chính xác một completion callback cho mỗi pre-operation. Post-operations có thể trả về FLT_POSTOP_CALLBACK_STATUS. Với giá trị là:

* FLT_POSTOP_FINISHED_PROCESSING —Minifilter đã hoàn thành các tiến trình cần thiết trong post-operation.

* FLT_POSTOP_STATUS_MORE_PROCESSING_REQUIRED — Minifilter vẫn chưa hoàn thành tiến trình, khi hoàn thành thì hàm FltCompletePendedPostOperation()sẽ được gọi.

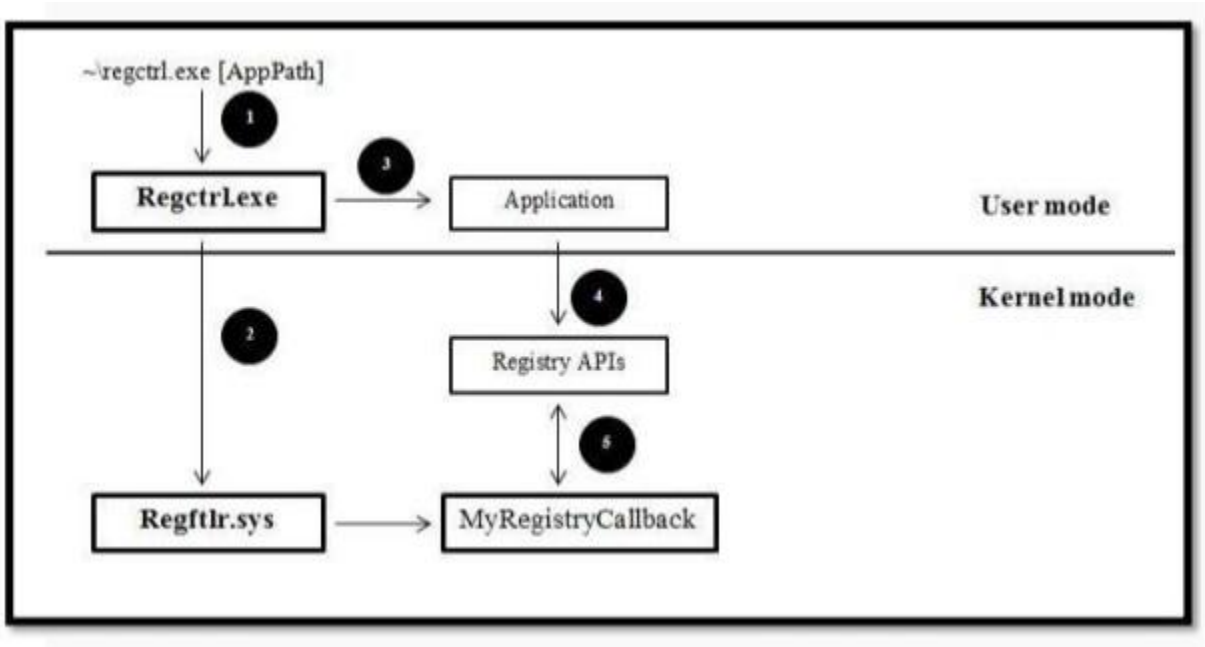
+ Các lệnh được gọi dựa theo mã IRP_MJ và thêm một vài mã đại diện cho Fast I/O. Sẽ có một cờ hiệu ở trong dữ liệu trả về để mà xác định sự khác nhau giữa Irp-based, FsFilter-based hay Fast I/O based.

Các mã IRP_MJ được cung cấp là:

- Tất cả các mã lệnh IRP_MJ từ IRP_MJ_CREATE đến IRP_MJ_PNP
- IRP_MJ_FAST_IO_CHECK_IF_POSSIBLE
- IRP_MJ_NETWORK_QUERY_OPEN
- IRP_MJ_MDL_READ
- IRP_MJ_MDL_READ_COMPLETE
- IRP_MJ_PREPARE_MDL_WRITE
- IRP_MJ_MDL_WRITE_COMPLETE
- IRP_MJ_VOLUME_MOUNT
- IRP_MJ_VOLUME_DISMOUNT
- IRP_MJ_ACQUIRE_FOR_SECTION_SYNCHRONIZATION
- IRP_MJ_RELEASE_FOR_SECTION_SYNCHRONIZATION
- IRP_MJ_ACQUIRE_FOR_MOD_WRITE

4. Registry Monitor (RegCtrl)

- Kiến trúc của RegCtrl:



Hình 6: Mô hình thể hiện kiến trúc RegCtrl.

RegCtrl bao gồm hai thành phần:

Thứ nhất là một ứng dụng chạy bằng dòng lệnh (RegCtrl.exe) ở chế độ người dùng (user-mode) với đầu vào là đường dẫn đến tệp đích hoặc chương trình, được dùng để cài đặt và nạp trình điều khiển, thực thi tệp đối tượng, gỡ trình điều khiển khi hoàn tất và phân tích nhật ký.

Thứ hai là trình điều khiển chế độ kernel (RegFltr.sys) dùng để đăng ký chức năng gọi lại và thông tin của những nhật ký.

- Tổng quan hoạt động của RegCtrl:

Người dùng nhập đường dẫn đích của đối tượng cho Regctrl.exe bằng cách sử dụng cmd.exe với lệnh "regctrl.exe [AppPath]".

- RegCtrl cài đặt và nạp trình điều khiển Regfltr.sys và dùng hàm gọi lại MyRegistryCallback để giám sát các registry API

- Nếu RegCtrl không thể tải trình điều khiển, nó sẽ thoát.

- Sau khi tải trình điều khiển thành công, RegCtrl tạo tiến trình con mới với đường dẫn của tệp đích.

- Trong quá trình xử lý, nếu tập tin đích sử dụng các registry API (như Reg… Key, hoặc Nt… Key), hàm MyRegistryCallback nhận ba tham số đầu vào bao gồm nội dung callback, loại Reg_notify_class và một con.

- Cơ chế hoạt động của RegCtrl:

  RegCtrl.exe

  RegCtrl.exe được phát triển bằng ngôn ngữ lập trình C++ và dựa trên Bộ lọc Registry của Windows-driver-samples-master.

  Hàm wmain() khai báo các biến được sử dụng thêm. Sau đó, hàm UtilLoadDriver() cài đặt và tải trình điều khiển (regfltr.sys). Nếu chương trình không thể tải trình điều khiển, chương trình sẽ thoát.

```
INT __cdecl
wmain(
    _In_ ULONG argc,
    _In_reads_(argc) LPCWSTR argv[]
    )
{

    BOOL Result;
    char s[8];
    STARTUPINFO si;

    UNREFERENCED_PARAMETER(argc);
    UNREFERENCED_PARAMETER(argv);

    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&process_info, sizeof(process_info));
```

```
    Result = UtilLoadDriver(DRIVER_NAME,
                            DRIVER_NAME_WITH_EXT,
                            WIN32_DEVICE_NAME,
                            &g_Driver);

    if (Result != TRUE) {
        ErrorPrint("UtilLoadDriver failed, exiting...");
        exit(1);
    }
...
}
```

Người dùng có thể nhập đường dẫn của tệp đích bằng cách sử dụng cú pháp regctrl.exe [AppPath]. Sau khi tải trình điều khiển thành công, RegCtrl.exe sử dụng hàm CreateProcess () để chạy tệp đích.

Sau đó, phần ghi nhật ký được bắt đầu. Nó dừng lại và tạo nhật ký có thể đọc được khi chương trình đích đã thoát.

Nhật ký có thể đọc được (sử dụng tracefmt.exe) là một phần rất quan trọng vì tracelog.exe chỉ tạo tệp nhật ký nhị phân.

```
INT __cdecl

wmain(

    _In_ ULONG argc,

    _In_reads_(argc) LPCWSTR argv[]

    )

{

...

       if (argc == 2) {

              if (!CreateProcess(NULL, (char *)argv[1], NULL, NULL, FALSE, 0, NULL,
NULL, &si, &process_info)) {

                     printf("\nCreateProcess failed (%d).", GetLastError());

                     goto Exit;

              }


              printf("\nCreated process with pid: %lu", process_info.dwProcessId);

              //Starting logging

              printf("\nStarting tracelog...\n");
```

```
            StartTraceview(L"c:\\tracelog.exe      -start      regctrl      -guid
\"c:\\regctrl_guids.guid\" -f \"c:\\regctrl_log.etl\" -flag 1");

            //Waiting until child process exits
            WaitForSingleObject(process_info.hProcess, INFINITE);

            //Close process and thread handle
            CloseHandle(process_info.hProcess);
            CloseHandle(process_info.hThread);

            //Stop logging when process exits
            printf("\nStopping tracelog...\n");
            StartTraceview(L"c:\\tracelog.exe -stop regctrl");

            //Generating log
            printf("\nStarting tracefmt...\n");
            StartTraceview(L"c:\\tracefmt.exe        c:\\regctrl_log.etl        -p
c:\\regctrl_pdb -o c:\\regctrl_log.txt");
      }
...
}
```

Cuối cùng, RegCtrl.exe gỡ trình điều khiển và thoát.

```
INT __cdecl
wmain(
    _In_ ULONG argc,
    _In_reads_(argc) LPCWSTR argv[]
    )
{
...
```

```
Exit:
        UtilUnloadDriver(g_Driver, NULL, DRIVER_NAME);
        printf("Driver unloaded\n");
        return 0;
}
```

Hàm StarTraceview () sử dụng API "_wsystem" để chạy lệnh như trong cmd.exe.

```
void StartTraceview(const wchar_t *cmd)
{
        int iStatus;
        errno_t err;
        int errCode;

        // Flush all streams
        _flushall();

        // Start traceview to log traffic
        iStatus = _wsystem(cmd);

        // Check if the _wsystem() is failed
        if (iStatus == -1)
        {
                err = _get_errno(&errCode);
                printf("_wsystem failed = %d\n", err);
        }
}
```

RegFltr.sys

RegFltr.sys là một trình điều khiển KMDF (một trong các framework được khai báo trong Windows Driver Framework) sử dụng một số thư viện trong Windows Driver Kit và REG_NOTIFY_CLASS để thực hiện quá trình giám sát.

```
    #include <ntifs.h>
  #include <ntstrsafe.h>
#include <wdmsec.h>
```

Giống như các trình điều khiển khác, RegFltr.sys cũng có các thành phần cơ bản:

- DriverEntry: một chức năng chính của trình điều khiển, là điểm khởi đầu khi trình điều khiển được tải, chịu trách nhiệm tạo đối tượng trình điều khiển, đăng ký gọi lại bằng cách sử dụng CmRegisterCallbackEx() và bắt đầu truy tìm WPP.

```
WPP_INIT_TRACING(DriverObject, RegistryPath);

...

Status = CmRegisterCallbackEx(MyRegistryCallback, &AltitudeString, DriverObject,
NULL, &g_CmCookie, NULL);


      if (!NT_SUCCESS(Status)) {

            ErrorPrint("Register MyRegistryCallback Failed");

      }
```

- DeviceUnload: thực hiện làm sạch trước khi trình điều khiển được gỡ bỏ.

Bên cạnh các thành phần cơ bản, RegFltr.sys có một chức năng giám sát bất kỳ hoạt động nào liên quan đến Registry có tên MyRegistryCallback.

MyRegistryCallback () có ba tham số: CallbackContext, Argument1 và Argument2.

- CallbackContext: giá trị mà trình điều khiển chuyển đến tham số Context của CmRegisterCallbackEx.
- Argument1: giá trị REG_NOTIFY_CLASS đã nhập để xác định loại hoạt động registry đang được thực hiện.
- Argument2: một con trỏ trỏ đến một cấu trúc chứa thông tin cụ thể cho kiểu hoạt động registry. Kiểu cấu trúc phụ thuộc vào REG_NOTIFY_CLASS trong Argument1.

```
NTSTATUS MyRegistryCallback(
        _In_      PVOID CallbackContext,
        _In_opt_  PVOID Argument1,
        _In_opt_  PVOID Argument2
)
{
---
}
```

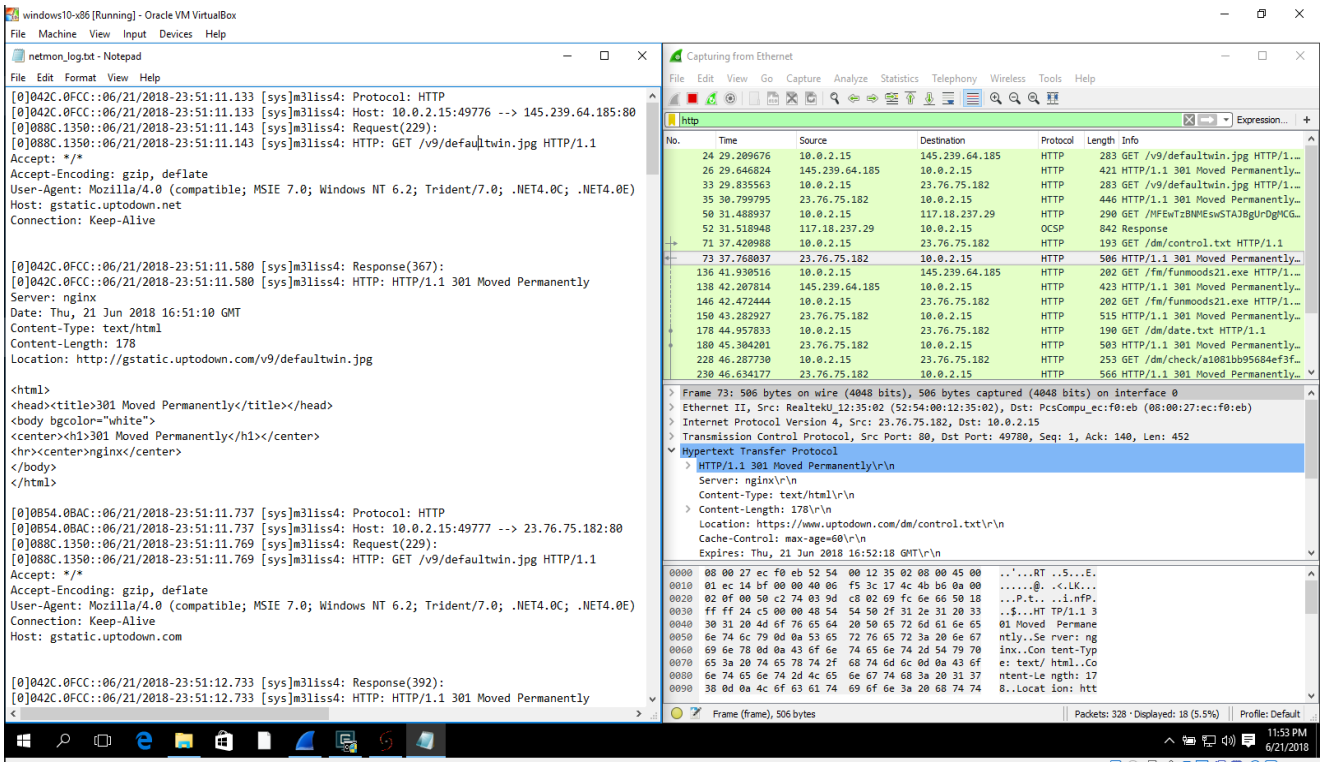
Sử dụng chuyển đổi cho Reg_notify_class để quyết định những gì được thực hiện ở mỗi thông báo. Vì mục tiêu chính là theo dõi các hoạt động registry, nên chỉ in ra thông tin từ cấu trúc cụ thể cho loại hoạt động registry.

```
switch (NotifyClass)
        {
        case RegNtPreCreateKeyEx:
                PreCreateInfo = (PREG_CREATE_KEY_INFORMATION)Argument2;

                DoTraceMessage(TRACE_INFO,  "MyRegistrytCallback:  Create  key  %wZ.",
PreCreateInfo->CompleteName);
                break;

        case RegNtPreSetValueKey:
                PreSetValueInfo = (PREG_SET_VALUE_KEY_INFORMATION)Argument2;

                DoTraceMessage(TRACE_INFO, "MyRegistryCallback: Set value name %wZ.",
PreSetValueInfo->ValueName);
---
```

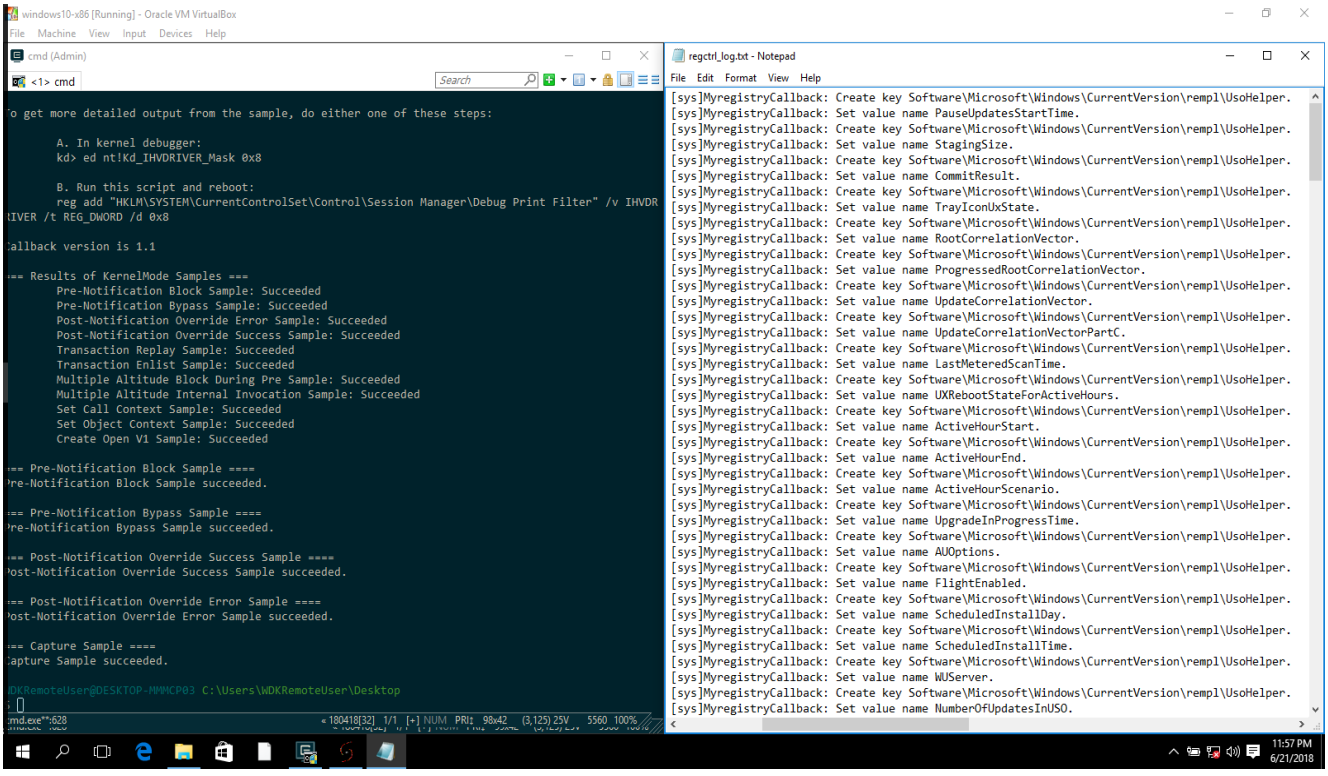
Sau đó, trả về STATUS_SUCCESS để cho chương trình biết rằng nó có thể tiếp tục xử lý hoạt động registry.

5. Kết quả phân tích mã độc Virus.Win32.uptodown.b.exe

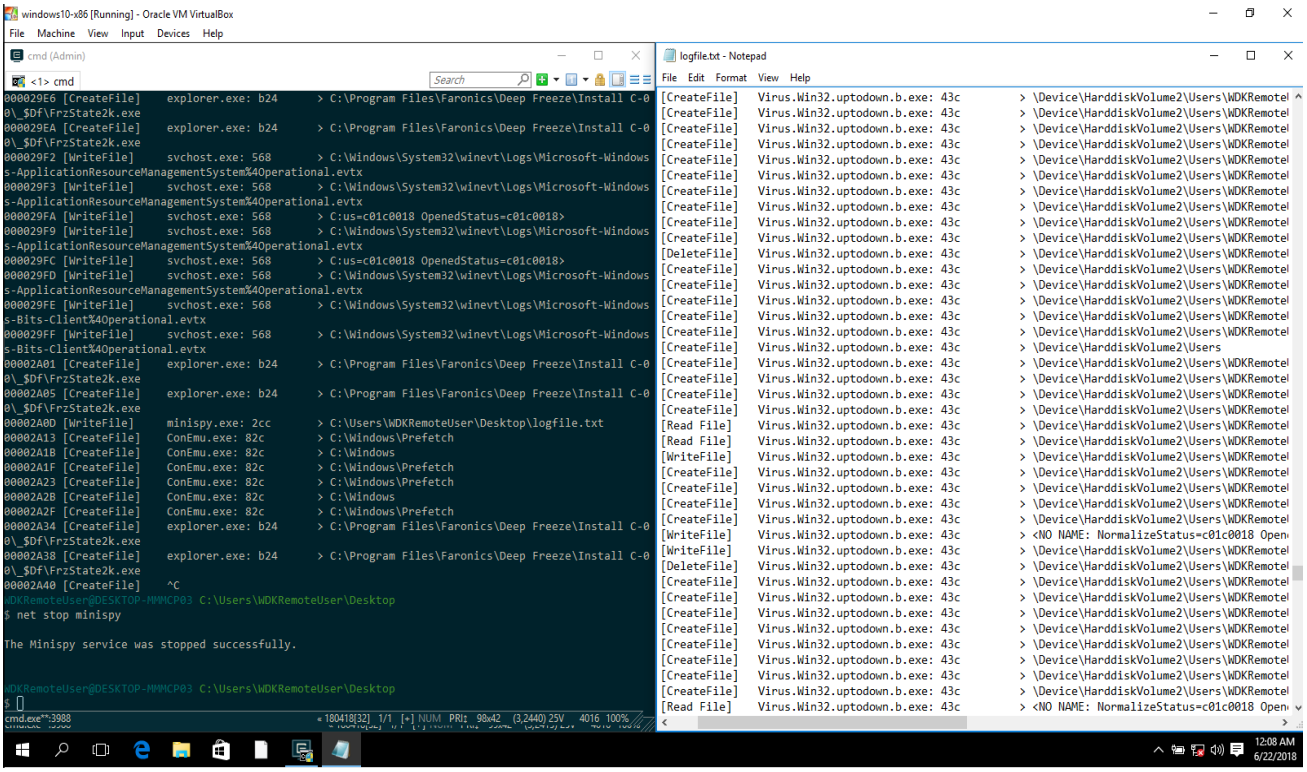
- Network monitor



- Registry monitor



- File system monitor



6. So sánh các sandbox và kết luận

10sandbox này có thể làm được như sau:

- Bắt được tất cả các traffic ra vào card mạng như là GET ,POST request, HTTP responses, SMTP,FTP , IRC của đối tượng và tất cả các tiến trình con sinh ra từ đối tượng đó. Qua đó giúp cho việc nhận biết đối tượng download những gì về máy hay upload những gi lên internet.
- Theo dõi được sự thay đổi của register mà đối tượng gây ra như là tạo khóa mới hay là set giá trị. Giúp nhận biết đối tượng đã đăng kí những thanh ghi nào với hệ điều hành từ đó nhận ra mối nguy hiểm của nó.
- Có thể xem sự thay đổi của file và folder, các hoạt động sửa đổi, xóa, tạo, đọc file mà đối tượng gây ra. Biết được đối tượng đã tạo ra những file gì, chèn vào những file nào trên hệ thống.

Trên thế giới đã có rất nhiều sandbox, nhưng những loại sandbox đó thì chỉ hổ trở trên window XP và window 7, còn 10box làm việc trên window 10.

So sánh với Cuckoo thì 10box có file log rõ ràng , dễ đọc hơn bởi vì nó ghi được tất cả các log mà chỉ có file đó tạo ra. Trong khi Cuckoo nó sử dụng TCPdump, nó có thể bắt được tất cả các traffic ra vào card mạng, dẫn đến nó không phân tích được gói tin nào của process nào. Giả sử như nếu có nhiểu process thì file log sẽ rất nhiều và gây khó khăn cho việc phân tích.

So sánh với H2Sandbox thì 10Box phân tích mẫu trên hệ điều hành Windows 10, sử dụng Windows Filtering Platform, một công nghệ mới được hỗ trợ bởi Windows, để giám sát các hoạt động mạng, có module đăng ký cung cấp thêm chi tiết về hoạt động registry.

Trong tương lai, sandbox sẽ được cải thiện công cụ phân tích.

Bằng cách nâng cấp trình điều khiển, 10Box không chỉ giám sát hoạt động mạng và registry mà còn giám sát hoạt động liên quan đến tiến trình và tệp hệ thống.

Tăng lên số lượng các tiến trình con mà 10Box có thể giám sát. Tạo tệp nhật ký dễ đọc hơn và phát triển một công cụ chuyện đổi định dạng văn bản (.txt) sang định dạng khác (chẳng hạn như HTML, PDF ...)

Số lượng các tiến trình con mà NetMon có thể giám sát được giới hạn ở mức 10.

Không chỉ giám sát các hoạt động registry của ứng dụng đích, RegCtrl cũng giám sát các hoạt động registry của hệ điều hành Windows, vấn đề này gây ra các thông tin không cần thiết được ghi vào tệp nhật ký.
