1. **Can thiệp vào hệ thống.**

Hook API

1. **Global hay Local**

Trong quá trình chèn mã (inject dll) vào tiến trình, chúng ta có 2 giải pháp là xử lý trên toàn cục hệ thống hay trên những tiến trình nhất định.

Global hook thực hiện xử lý trên toàn hệ thống, mọi tiến trình sẽ bị tác động vào. Điều này mang lại nhiều kết quả không mong muốn. Đó là hệ thống sẽ bị làm chậm đáng kể do các xử lý sẽ phải đi qua hàm hook. Không gian bộ nhớ cũng sẽ tăng thêm ở mỗi tiến trình do phải chứa mã được chèn vào (dll inject). Nguy hiểm hơn, khi xử lý trong hàm hook gặp lỗi có thể khiến hệ thống gặp sự cố (có thể phải khởi động lại) không mong muốn.

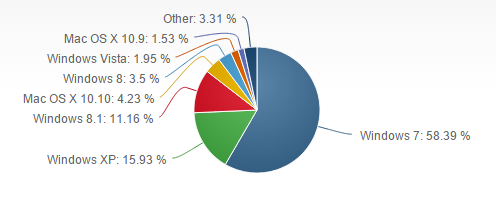
Thống kê số lượng sự kiện khi xử lý Global hook.

….

Local hook xác định tiến trình cần tác động và thực hiện xử lý trên đúng đối tượng. Tiết kiệm không gian bộ nhớ, giảm những tính toán dư thừa là lợi thế so với Global hook. Chính vì thế, nhóm nghiên cứu chọn Local hook để cài đặt phần mềm.

1. **Hệ điều hành: Windows 7.**

Net Applications hoạt động từ 1999 và trở thành website phân tích tin cậy. Theo số liệu phân tích tháng 4 năm 2015



Biểu đồ cho thấy số lượng người dùng windows 7 hiện nay rất lớn (58.39%). Windows 7 có tỷ lệ vượt trội so với 2 hệ điều hành xếp ngay sau là Windows XP (15.93%) và Windows 8.1 (11.16%). Nhóm nghiên cứu đã chọn Windows 7 làm đối tượng nghiên cứu và thực nghiệm.

1. **Thay đổi giá trị thanh ghi (register).**

Mặc định exel sử dụng cơ chế mở file DDE (Dynamic Data Exchange). Cơ chế này khiến file exel khởi động lại không chính xác.

DDE là phương thức trao đổi thông tin giữa các tiến trình của hệ điều hành Windows và OS/2 (dòng hệ điều hành được phát triển bởi IBM). DDE cho phép một chương trình đăng ký theo dõi thành phần của chương trình khác. Một ô trong Microsoft Excel có thể liên kết một tới một giá trị trong ứng dụng khác và khi giá trị này thay đổi sẽ cập nhật trên bảng tính Exel. DDE hoạt động theo mô hình client-server. Trong đó ứng dụng khởi tạo kết nối là client, ứng dụng trả lời là server (Exel).

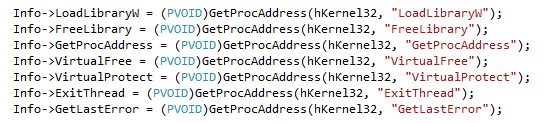
Tại sao DDE làm exel khởi động lại không chính xác?

Khi chương trình thực hiện hook hàm CreateProcess() với đối tượng là Microsoft Exel, nó gọi một lệnh command line tới file thực thi của Exel. Lúc này, lệnh command line tự động thêm 1 tham số “dde” vào cuối dòng lệnh (tham số này được mô tả ở Register). Exel sẽ ủy quyền cho DDE mở file với những tham số cần thiết. Kết quả là khi chặn hàm CreateProcess() không lấy được tham số cần thiết để mở file.

1. **Ứng dụng không khởi động thành công khi thực hiện hook (load chậm hay khi khởi động lại load sai).**

Ứng dụng mục tiêu bị crash khi khởi động. Nguyên nhân là do ứng dụng mục tiêu load không đủ các hàm cần thiết để thực hiện quá trình hook.

Cơ bản, chúng ta tạo một thread ở bên trong ứng dụng mục tiêu. Thread này có nhiệm vụ load các hàm cần thiết để phục vụ cho mục đích hook. Do thread được tạo này không load đầy đủ hàm cần thiết dẫn đến hiện tượng crash và ứng dụng mục tiêu khởi động không thành công.



Update DLL version. Thực hiện một bước hỗ trợ quá trình load các hàm cần thiết.

Chúng ta tạo thêm một thread có chức năng tương tự thread được tạo từ xa để load các hàm quan trọng vào process. Đảm bảo ứng dụng mục tiêu chứa các hàm thực hiện hook.

1. **Xác định và thay đổi thông tin file**

Vấn đề đặt ra là sau khi chèn được mã vào ứng dụng (process), làm cách nào xác định được thông tin của file cần bảo vệ. Tiếp đó, chúng ta cần biến đổi nội dung file (mã hóa) nhằm bảo vệ dữ liệu quan trọng bên trong. Có giải pháp nào giải quyết được vấn đề này?

Hệ điều hành sử dụng hàm ReadFile() để đọc thông tin file

ReadFile()..

Với những tham số trên, hàm ReadFile() cung cấp cho ta đầy đủ dữ liệu của một file để phân tích và đưa ra biện pháp xử lý. Sử dụng phương thức hook chặn hàm ReadFile() và lấy các tham số cần thiết. Sau đó, tùy vào mục tiêu có thể thay đổi dữ liệu để hàm ReadFile() gốc thực hiện với dữ liệu đã được thay đổi (áp dụng để giải mã dữ liệu).

Bên cạnh đó, hệ điều hành gọi hàm WriteFile() khi cần ghi dữ liệu xuống một file xác định.

BOOL WINAPI WriteFile(

\_In\_        HANDLE       hFile,

\_In\_        LPCVOID      lpBuffer,

\_In\_        DWORD        nNumberOfBytesToWrite,

\_Out\_opt\_   LPDWORD      lpNumberOfBytesWritten,

\_Inout\_opt\_ LPOVERLAPPED lpOverlapped

);

Trong đó:

* hFile là handle của file hoặc I/O device được ghi xuống.
* lpBuffer là con trỏ tới dữ liệu được ghi xuống file hoặc device.
* nNumberOfBytesToWrite là số byte dữ liệu được ghi.
* lpNumberOfBytesWritten là con trỏ tới biến lưu số byte dữ liệu được ghi xuống thành công.
* lpOverlapped là con trỏ tới một cấu trúc nếu file được mở với cờ **FILE\_FLAG\_OVERLAPPED, ngoài ra tham số này có thể bằng NULL**

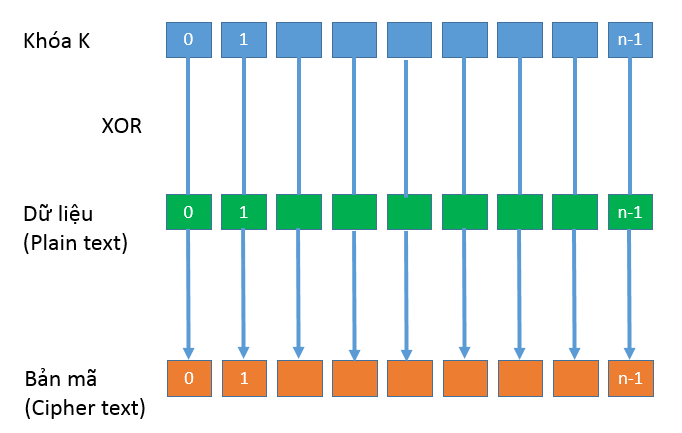
Tương tự như ReadFile(), hàm WriteFile() cũng có các thông tin của file và dữ liệu. Chỉ có điều dữ liệu này sẽ được ghi xuống file khi hàm WriteFile() thực thi thành công.

1. **Thuật toán mã hóa**

Lựa chọn thuật toán mã hóa đối xứng AES (**Advanced Encryption Standard)**. Thuật toán mã hóa đối xứng cho tốc độ mã hóa nhanh hơn rất nhiều so với thuật toán mã hóa đối xứng. Mã hóa đối xứng đủ an toàn với chiều dài khóa (key) nhất định (đối với AES đảm bảo an toàn với khóa dài 256 bit).

1. **Chế độ mã hóa (Mode cipher)**

Áp dụng tư tưởng stream cipher. Tức là mã hóa từng byte (hoặc bit) của dữ liệu với byte tương ứng của khóa.



1. **Mã hóa và giải mã**
   1. **Mã hóa**

Sử dụng AesCrytoServiceProvider (**Namespace:**  [System.Security.Cryptography](https://msdn.microsoft.com/en-us/library/system.security.cryptography(v=vs.110).aspx), **Assembly:**  System.Core (in System.Core.dll) tạo ra khóa mã hóa (key) K từ password, IV và dữ liệu khởi tạo.

Dữ liệu cần mã hóa (plain text) sẽ được chia thành các khối (block) có độ dài bằng độ dài khóa K. Tiến thành xor dữ liệu với khóa theo từng byte thu được dữ liệu mã hóa (cipher text).

* 1. **Giải mã**

Đọc thông tin mã hóa (metadata) lấy thông tin cần thiết: quyền truy cập (giải mã), IV…

Tạo khóa giải mã (tương tự khóa mã hóa) K. Trong trường hợp này khóa giải mã chính là khóa mã hóa (vì thực hiện phép xor để mã dữ liệu).

Chia nhỏ dữ liệu mã hóa (cipher text) thành từng khối (block) có kích thước bằng khóa K. Thực hiện phép xor dữ liệu mã hóa và khóa thu được dữ liệu ban đầu (plain text).

1. **Thông tin mã hóa (metadata)**

Với mỗi file mã hóa cần chứa thông tin để giải mã. Thông tin mã hóa chứa thông tin liên quan đến thao tác mã hóa và giải mã. Vấn đề đặt ra là lưu thông tin mã hóa này chứa những gì, lưu ở đâu, như thế nào?

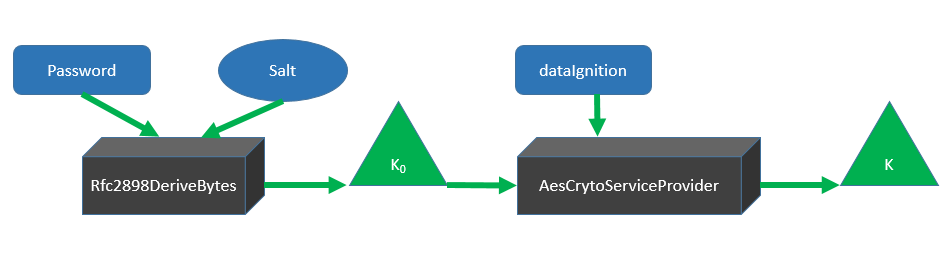
* 1. **Thông tin mã hóa chứa những gì?**

Quá trình mã hóa và giải mã dữ liệu của chương trình đơn giản là thực hiện phép xor tuần tự khóa (K) và dữ liệu. Vì vậy, thông tin mã hóa sẽ phải chứa dữ liệu liên quan đến việc hình thành khóa.

Chúng ta sử dụng AesCrytoServiceProvider mã hóa một khối dữ liệu để tạo khóa (K). AesCrytoServiceProvider tạo ra IV (giá trị khởi tạo ban đầu) và cần được nhận vào giá trị Key (K0).

Để tăng tính bảo mật, K0 của AesCrytoServiceProvider sẽ được tạo thành từ 2 thành phần: password (mật khẩu-thông tin bí mật) và salt (muối-tăng tính ngẫu nhiên). Sử dụng lớp Rfc2898DeriveBytes với phương thức khởi tạo sử dụng 2 tham số password và salt để tạo K0.

Cuối cùng, AesCrytoServiceProvider có khóa K0 sẽ mã hóa một khối dữ liệu (dataIgnition) tạo thành khóa K.



* 1. **Lưu thông tin mã hóa ngay trong file được mã hóa.**

Yêu cầu này đặt ra một số vấn đề quan trọng cần xử lý:

* + 1. Ghi thêm thông tin vào file trong khi hook

Vấn đề đặt ra là có thể ghi thêm thông tin vào file cần mã hóa khi chặn hàm WriteFile() hay không?

Cấu trúc hàm WriteFile()

BOOL WINAPI WriteFile(

\_In\_        HANDLE       hFile,

\_In\_        LPCVOID      lpBuffer,

\_In\_        DWORD        nNumberOfBytesToWrite,

\_Out\_opt\_   LPDWORD      lpNumberOfBytesWritten,

\_Inout\_opt\_ LPOVERLAPPED lpOverlapped

);

Trong đó:

* hFile là handle của file hoặc I/O device được ghi xuống.
* lpBuffer là con trỏ tới dữ liệu được ghi xuống file hoặc device.
* nNumberOfBytesToWrite là số byte dữ liệu được ghi.
* lpNumberOfBytesWritten là con trỏ tới biến lưu số byte dữ liệu được ghi xuống thành công.
* lpOverlapped là con trỏ tới một cấu trúc nếu file được mở với cờ **FILE\_FLAG\_OVERLAPPED, ngoài ra tham số này có thể bằng NULL**

Khi thay đổi dữ liệu ghi xuống bởi hàm WriteFile() cần thay đổi 2 tham số lpBuffer và nNumberOfBytesToWrite. Thao tác ghi dữ liệu sẽ thực hiện bình thường. Cần lưu ý là hàm WriteFile không thay đổi được vị trí bắt đầu ghi dữ liệu (thường gọi là offset).

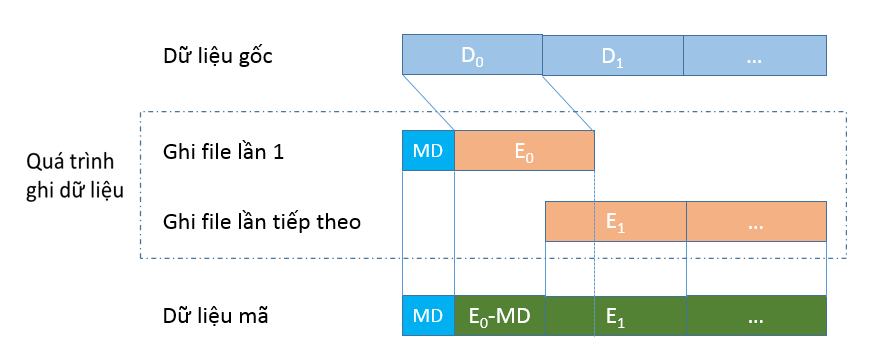
* + 1. **Lưu thông tin mã hóa ở vị trí nào của file?**

Khi xác định có thể lưu thông tin mã hóa vào file, quyết định vị trí của thông tin mã hóa rất quan trọng. Bởi vì quá trình ghi dữ liệu xuống một file (thực chất là gọi hàm WriteFile()) được chia nhỏ. Khi đó, một thay đổi nhỏ trong một lần ghi xuống file cũng có thể ảnh hưởng đến cả quá trình ghi file và tạo ra file lỗi hoặc crash ứng dụng thực hiện WriteFile().

Giả sử một file F được ghi xuống thiết bị USB (hoặc ổ đĩa) có dữ liệu được chia nhỏ thành N phần liên tiếp (N>1), mỗi phần có kích thước là Di byte. Mỗi phần dữ liệu này được ghi xuống bởi hàm WriteFile(). Hàm WriteFile nhận dữ liệu, mã hóa và thực hiện ghi xuống thiết bị USB. Dữ liệu mã hóa E có kích thước đúng bằng dữ liệu gốc, vì thao tác mã hóa thực chất là thực hiện phép xor.

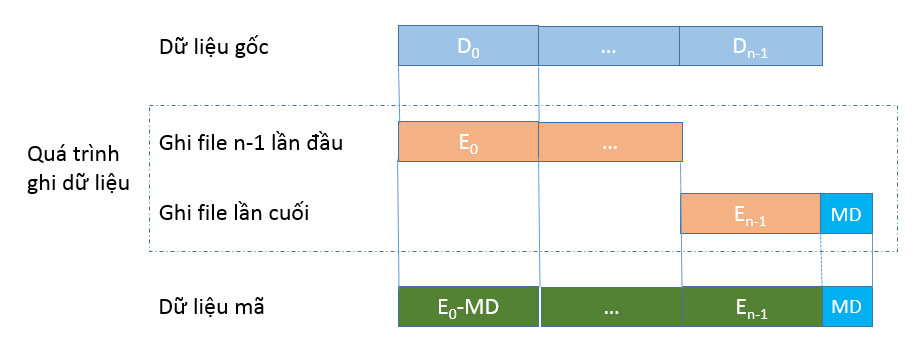
**Trường hợp 1:** thêm MD byte dữ liệu vào đầu file F (MD < D0). Lúc này tổng kích thước cần ghi xuống lần đầu tiên là T = E­0­ + MD byte (E0 = D0). Để lưu được T byte, hàm WriteFile() phải thay đổi tham số kích thước được ghi để có thể lưu thành công.

Trong quá trình can thiệp hàm WriteFile() có thể thay đổi giá trị tham số kích thước file được ghi để thực hiện thao tác này. Theo đó, vị trí bắt đầu ghi tiếp theo sẽ phải dịch một khoảng bằng D0 – MD = E0 – MD byte. Thế nhưng, hàm WriteFile không có tham số lưu vị trí bắt đầu ghi tiếp theo (offset). Lần ghi dữ liệu thứ 2 (dữ liệu D­1), hệ thống giữ nguyên vị trí ghi ban đầu dẫn tới việc dữ liệu ở lần ghi đầu tiên bị ghi đè.



**Trường hợp 2:** thêm MD byte dữ liệu vào cuối file F. Trong lần ghi dữ liệu cuối cùng (dữ liệu DN-1), chúng ta ghi tổng cộng DN-1 + MD byte. Thao tác này chỉ cần thay đổi tham số kích thước được ghi và không quan tâm đến vị trí ghi dữ liệu tiếp theo (vì là lần ghi cuối cùng).

Khi đọc dữ liệu file F’ (file F được mã hóa), chỉ cần đọc MD byte thông tin mã hóa ở cuối file và tiến hành giải mã dựa theo thông tin mã hóa này.



* + 1. **Microsoft Office không cho phép mở file đã mã hóa.**

Đối với các file thuộc phiên bản Microsoft Office 2007 trở về sau, ta thêm một phần dữ liệu vào cuối file thì khi các ứng dụng mở file lên sẽ thông báo các file đã bị lỗi và hỏi rằng chúng ta có “tin tưởng” file này để mở lên và repair hay không, nếu ta chọn tin tưởng thì lúc đó file mới được mở bình thường.

Đối với các loại file thuộc các chương trình khác thì việc đó không gây ảnh hưởng gì.

* 1. **Lưu thông tin mã hóa ra một file riêng biệt.**

Việc lưu thông tin mã hóa riêng biệt với file cần bảo vệ sẽ giải quyết vấn đề lỗi cấu trúc file. File lưu thông tin mã hóa (FT) có thể chứa thông tin rác và sắp xếp có cấu trúc nhằm bảo vệ thông tin bên trong.

* + 1. **Mã hóa và giải mã**

Trong trường hợp này, quá trình mã hóa và giải mã đơn giản hơn rất nhiều so với giải pháp ghi thông tin mã hóa vào file cần bảo vệ.

Cả 2 thao tác mã và giải mã đều thực hiện đọc thông tin mã hóa từ file. Lọc những thông tin cần thiết (bỏ dữ liệu rác, ghép nối thông tin…) để thực thi.

Sử dụng thông tin mã hóa để tạo khóa (key). Tiến hành mã hóa và giải mã với khóa đã tạo.

* + 1. **Mất file chứa thông tin mã hóa**

Vấn dề: mất file chứa thông tin mã hóa sẽ không thể giải mã chính xác vì không có thông tin IV.

Giải pháp:

Hook hàm DeleteFile, ngăn không cho người dùng xóa file thông tin mã hóa.

Vì file chứa thông tin mã hóa được ẩn đi và không tốn nhiều dung lượng bộ nhớ nên việc tồn tại của nó không ảnh hưởng đến công việc.