DLL là một thư viện liên kết động chứa các đoạn mã/dữ liệu để phục vụ cho các chương trình trong máy tính. Các function trong window API đều chứa trong DLL.

Một số DLL quan trọng như: Kernel32.dll chứa các function về quản lý bộ nhớ, tiến trình, luông.; User32.dll chứa các function phục vụ giao diện người dùng như tạo cửa sổ và gửi thông điệp; GDI32.dll chứa các function để vẽ các hình ảnh đồ họa và hiển thị văn bản.

Mục đích của DLL:

1. **Dùng để mở rộng ứng dụng:** code trong dll có thể nạp vào vùng địa chỉ của tiến trình từ đó tiến trình có thể xem xét thực hiện hành động nào theo yêu cầu.
2. **Dùng để đơn giản hóa công tác quản lý dự án**: khi dự án có nhiều module riêng lẻ thì dll giúp cho việc quản lý phat triển dễ dàng hơn do ít phải can thiệp vào product. Nhưng cần giới hạn dll ở mức nhất định do việc quá nhiều dll sẽ khiến cho ứng dụng mất nhiều thời gian để nạp code từ rất nhiều file được lưu trong ổ đĩa.
3. **Tiết kiệm RAM:** mỗi dll đều được chia sẻ trong cùng 1 page trong RAM và các ứng dụng cùng dùng chung dll sẽ không cần thêm RAM để nạp lại code trong dll.
4. **Chia sẻ các nguồn tài nguyên**: như dialog, string, icon..
5. **Tạo điều kiện cho việc cục bộ hóa ứng dụng**
6. **Giải quyết sự khác nhau trong các nền tảng.**
7. **Có thể dùng cho các mục đích đặc biệt khác nhau.**

DLL và vùng địa chỉ tiến trình

một tiến trình là một phiên bản đang chạy của một chương trình và chương trình đó bắt đầu dưới dạng tệp EXE trên đĩa.

Về cơ bản, DLL là một tệp trên đĩa (thường có phần mở rộng DLL) bao gồm dữ liệu toàn cục, các hàm được biên dịch và tài nguyên, trở thành một phần của quy trình trong chương trình của bạn. Nó được biên dịch để tải tại một địa chỉ mà bạn muốn và nếu không có xung đột với các DLL khác, tệp sẽ được ánh xạ tới cùng một địa chỉ ảo trong chương trình của bạn. DLL có các function được **export** khác nhau và chương trình khách (client - chương trình đã tải DLL ở vị trí đầu tiên) **import** các chức năng đó. Windows đối sánh các loại **import** và **export** khi nó tải DLL. Win32 DLL cho phép **exported** các biến toàn cục cũng như các hàm.

Trong Win32, mỗi tiến trình nhận được bản sao riêng của các biến toàn cục đọc / ghi của DLL. Nếu bạn muốn chia sẻ bộ nhớ giữa các tiến trình, bạn phải sử dụng tệp ánh xạ bộ nhớ hoặc khai báo phần dữ liệu được chia sẻ như được mô tả trong Windows nâng cao của tác giả Jeffrey Richter (Microsoft Press, 1997). Bất cứ khi nào DLL của bạn yêu cầu bộ nhớ heap, bộ nhớ đó sẽ được cấp phát từ heap của tiến trình máy khách (client).

cách thức code, build & load dll;

export function;

Một DLL chứa một bảng các hàm đã xuất. Các chức năng này được xác định với thế giới bên ngoài bằng tên tượng trưng của chúng và (tùy chọn) bởi các số nguyên được gọi là số thứ tự. Bảng hàm cũng chứa địa chỉ của các hàm trong DLL. Khi chương trình khách lần đầu tiên tải DLL, nó không biết địa chỉ của các hàm mà nó cần gọi, nhưng nó biết các ký hiệu hoặc thứ tự. Sau đó, quá trình liên kết động sẽ xây dựng một bảng kết nối các lệnh gọi của máy khách (client) với các địa chỉ hàm trong DLL. Nếu bạn chỉnh sửa và xây dựng lại DLL, bạn không cần phải xây dựng lại chương trình khách của mình trừ khi bạn đã thay đổi tên hàm hoặc chuỗi tham số. hiểu một cách đơn giản, bạn sẽ có một tệp EXE **import** các hàm từ một hoặc nhiều DLL. Trong thực tế, nhiều DLL gọi các hàm bên trong các DLL khác. Do đó, một DLL cụ thể có thể có cả **import** và **export**.

Đây không phải là vấn đề vì quá trình liên kết động có thể xử lý các phụ thuộc chéo. Trong mã DLL, bạn phải khai báo rõ ràng các hàm đã xuất của mình như sau:

\_\_declspec(dllexport) int MyFunction(int n);

Giải pháp thay thế là liệt kê các hàm đã xuất của bạn trong tệp [DEF] định nghĩa mô-đun, nhưng điều đó thường rắc rối hơn. Về phía máy khách, bạn cần khai báo các lần import tương ứng như sau:

\_\_declspec(dllimport) int MyFunction(int n);

Nếu bạn đang sử dụng C ++, trình biên dịch sẽ tạo ra một tên được trang trí cho MyFunction (), mà các ngôn ngữ khác không thể sử dụng. Những tên được trang trí này là những tên dài mà trình biên dịch tạo ra dựa trên tên lớp, tên hàm và các kiểu tham số. Chúng được liệt kê trong tệp MAP của dự án. Nếu bạn muốn sử dụng tên thuần MyFunction (), bạn phải viết các khai báo theo cách sau:

extern "C" \_\_declspec(dllexport) int MyFunction(int n);

extern "C" \_\_declspec(dllimport) int MyFunction(int n);

Theo mặc định, trình biên dịch sử dụng quy ước truyền đối số \_\_cdecl, có nghĩa là chương trình gọi sẽ bật các tham số ra khỏi ngăn xếp. Một số ngôn ngữ máy khách (client) có thể yêu cầu quy ước stdcall, quy ước này thay thế quy ước gọi Pascal và có nghĩa là hàm được gọi bật ra ngăn xếp. Do đó, bạn có thể phải sử dụng công cụ sửa đổi stdcall trong khai báo xuất DLL của mình. Chỉ có khai báo **import** là không đủ để tạo liên kết máy khách với DLL. Dự án của chương trình khách (client) phải chỉ định thư viện **import** (LIB) cho trình liên kết và chương trình khách phải thực sự chứa lệnh gọi đến ít nhất một trong các hàm đã imported của DLL. Câu lệnh gọi đó phải nằm trong một đường dẫn thực thi trong chương trình.

**Liên kết ngầm và Liên kết rõ ràng**

Phần trước chủ yếu mô tả liên kết ngầm, đó là những gì bạn là một lập trình viên C ++ có thể sẽ sử dụng cho các tệp DLL của bạn. Khi bạn tạo DLL, trình liên kết tạo tệp LIB, tệp này chứa mọi ký hiệu và thứ tự (tùy chọn) của DLL, nhưng không có mã. Tệp LIB là một đại diện cho DLL được thêm vào dự án của chương trình khách (client) . Khi bạn xây dựng (liên kết tĩnh) máy khách, các ký hiệu đã nhập được khớp với các ký hiệu đã export trong tệp LIB và các ký hiệu (hoặc thứ tự) đó được liên kết với tệp EXE. Tệp LIB cũng chứa tên tệp DLL (nhưng không phải tên đường dẫn đầy đủ của nó), được lưu trữ bên trong tệp EXE. Khi máy khách được tải, Windows sẽ tìm và tải DLL, sau đó liên kết động nó theo ký hiệu hoặc theo thứ tự.

Liên kết rõ ràng thích hợp hơn cho các ngôn ngữ thông dịch như Microsoft Visual Basic, nhưng bạn có thể sử dụng nó từ C ++ nếu cần. Với liên kết rõ ràng, bạn không sử dụng tệp import; thay vào đó, bạn gọi hàm Win32 LoadLibrary (), chỉ định tên đường dẫn của DLL làm tham số. LoadLibrary () trả về một tham số HINSTANCE mà bạn có thể sử dụng trong cuộc gọi tới GetProcAddress (), nó sẽ chuyển đổi một biểu tượng (hoặc một thứ tự) thành một địa chỉ bên trong DLL. Giả sử bạn có một DLL exports một hàm như sau:

extern "C" \_\_declspec(dllexport) double SquareRoot(double d);

Dưới đây là một ví dụ về liên kết rõ ràng của trình khách (client) với hàm:

typedef double (SQRTPROC)(double);

HINSTANCE hInstance;

SQRTPROC\* pFunction;

VERIFY(hInstance = ::LoadLibrary("c:\\winnt\\system32\\mydll.dll"));

VERIFY(pFunction = (SQRTPROC\*)::GetProcAddress(hInstance, "SquareRoot"));

double d = (\*pFunction)(81.0); // Call the DLL function

Với liên kết ngầm định, tất cả các tệp DLL được tải khi client được tải, nhưng với liên kết rõ ràng, bạn có thể xác định thời điểm các tệp DLL được tải và không tải. Liên kết rõ ràng cho phép bạn xác định DLL sẽ tải trong thời gian chạy. Ví dụ: bạn có thể có một DLL với tài nguyên chuỗi bằng tiếng Anh và một tệp khác với tài nguyên chuỗi bằng tiếng Tây Ban Nha. Ứng dụng của bạn sẽ tải DLL thích hợp sau khi người dùng chọn một ngôn ngữ.

các kỹ thuật inject dll:

**DLL Injection** là quá trình chèn code vào một tiến trình (process) đang chạy. Code được sử dụng ở đây là ở dạng thư viện liên kết động (DLL). Tuy nhiên không phải chỉ chèn được code dạng DLL, chúng ta có thể chèn code ở nhiều dạng khác như exe, handwritten, … Điều quan trọng là chúng ta có đủ quyền hệ thống để thao tác với tiến trình của ứng dụng khác hay không.

Thực ra Windows API đã cung cấp cho chúng ta một vài các hàm để can thiệp và thao tác vào những chương trình khác cho mục đích Debug. Chúng ta sẽ tận dụng các API này để thực hiện chèn DLL. Ta sẽ chia DLL injection thành 4 bước sau:

**– Can thiệp vào process  
– Cấp phát một vùng nhớ trong process  
– Copy toàn bộ DLL hoặc đường dẫn đến DLL vào vùng nhớ đó và xác định vị trí của vùng nhớ  
– Process thực thi DLL**

Mỗi bước có thể được thực hiện bởi một hoặc nhiều các kỹ thuật, sẽ được tóm tắt bằng hình ảnh bên dưới. Quan trọng là chúng ta hiểu được chi tiết các kỹ thuật và các mặt tích cực, tiêu cực của chúng để sử dụng trong các trường hợp khác nhau.

Để process thực thi DLL chúng ta có một vài lựa chọn CreateRemoteThread(), NtCreateThreadEx(), … Chúng ta thực hiện các bước cấp phát và copy để có không gian bộ nhớ của process và chuẩn bị nó để bắt đầu thực thi DLL.  
Có 2 cách phổ biến là LoadLibraryA() và nhảy đến DLLMain.

LoadLibraryA :  
LoadLibraryA() là hàm trong kernel32.dll để nạp DLL, file thực thi hoặc các loại thư viện khác. Tham số truyền vào của hàm là tên DLL. Nghĩa là chúng ta chỉ cần cấp phát một vùng nhớ, ghi đường dẫn đến DLL và chọn điểm bắt đầu thực thi là địa chỉ của hàm LoadLibraryA(), tham số truyền vào là địa chỉ của vùng nhớ chứa đường dẫn đến DLL.  
Nhược điểm chính của hàm LoadLibraryA() là nó sẽ đăng ký DLL với chương trình nên sẽ dễ bị phát hiện (mỗi chương trình đều có một bảng các DLL sẽ nạp). Và một điều nữa là DLL chỉ được nạp lên chứ không được thực thi.

Nhảy đến DLLMain (hoặc một entry point khác):  
Một phương thức thay thế cho LoadLibraryA() là nạp toàn bộ DLL vào vùng nhớ và xác định offset tới DLL entry point. Sử dụng phương thức này sẽ tránh được việc đăng ký DLL với chương trình (tàng hình) và thực thi DLL trong process.

**Can thiệp vào Process:**

Đầu tiên chúng ta cần lấy được handle của process để có thể thao tác với nó. Bước này chúng ta sẽ sử dụng hàm *OpenProcess()*. Chúng ta cũng cần những yêu cầu về quyền truy cập để thực thi các tác vụ dưới đây. Những quyền truy cập mà chúng ta cần sẽ khác nhau đối với các phiên bản Windows, tuy nhiên hầu hết là như dưới đây:

hHandle = OpenProcess( PROCESS\_CREATE\_THREAD |

                    PROCESS\_QUERY\_INFORMATION |

                    PROCESS\_VM\_OPERATION |

                    PROCESS\_VM\_WRITE |

                    PROCESS\_VM\_READ,

                    FALSE,

                    procID );

### ****Cấp phát vùng nhớ****

Trước khi chèn bất kì thứ gì vào process khác chúng ta đều cần có một chỗ để đặt nó vào. Chúng ta sẽ sử dụng hàm VirtualAllocEx() để thực hiện công việc đó.  
VirtualAllocEx() lấy dung lượng vùng nhớ cần cấp phát làm tham số truyền vào. Nếu sử dụng hàm LoadLibraryA(), chúng ta cần cấp phát vùng nhớ để ghi đường dẫn đến DLL, còn nếu sử dụng phương thức nhảy đến DLLMain thì cần cấp phát vùng nhớ đủ lớn để ghi toàn bộ DLL vào.

Sử dụng đường dẫn đến DLL sẽ phải sử dụng hàm LoadLibraryA() cùng với những hạn chế ta đã nói ở trên. Nhưng nó là một phương pháp rất phổ biến.  
Cấp phát đủ vùng nhớ để ghi đường dẫn đến DLL vào:

GetFullPathName(TEXT("mydll.dll"),

              BUFSIZE,

              dllPath, //Đường dẫn đến DLL sẽ được lưu vào đây

              NULL);

dllPathAddr = VirtualAllocEx(hHandle,

                       0,

                       strlen(dllPath),

                       MEM\_RESERVE|MEM\_COMMIT,

                       PAGE\_EXECUTE\_READWRITE);

Sử dụng toàn bộ code trong DLL chúng ta sẽ không cần sử dụng hàm LoadLibraryA() và sẽ tránh được các hạn chế trên.

Đầu tiên chúng ta sẽ lấy handle của DLL bằng hàm CreateFileA() và tính toán kích thước của DLL bằng hàm GetFileSize(), cuối cùng và đưa vào hàm VirtualAllocEx():

GetFullPathName(TEXT("mydll.dll"),

              BUFSIZE,

              dllPath, //Đường dẫn đến DLL sẽ được lưu vào đây

              NULL);

hFile = CreateFileA( dllPath,

                 GENERIC\_READ,

                 0,

                 NULL,

                 OPEN\_EXISTING,

                 FILE\_ATTRIBUTE\_NORMAL,

                 NULL );

dllFileLength = GetFileSize( hFile,

                       NULL );

remoteDllAddr = VirtualAllocEx( hProcess,

                          NULL,

                          dllFileLength,

                          MEM\_RESERVE|MEM\_COMMIT,

                          PAGE\_EXECUTE\_READWRITE );

### ****Copy DLL và xác định địa chỉ****

Giờ chúng ta có thể copy đường dẫn hoặc toàn bộ DLL đến vùng nhớ của process.

Khi đã có vùng nhớ cần thiết, chúng ta sẽ sử dụng hàm WriteProcessMemory() để thực hiện công việc ghi:

Đường dẫn DLL:

WriteProcessMemory( hHandle,

                dllPathAddr,

                dllPath,

                strlen(dllPath),

                NULL);

Toàn bộ DLL: Chúng ta cần đọc DLL trước khi ghi nó vào vùng nhớ của process

lpBuffer = HeapAlloc( GetProcessHeap(),

                  0,

                  dllFileLength);

ReadFile( hFile,

        lpBuffer,

        dllFileLength,

        &dwBytesRead,

        NULL );

WriteProcessMemory( hProcess,

                lpRemoteLibraryBuffer,

                lpBuffer,

                dllFileLength,

                NULL );

Xác định điểm bắt đầu thực thi:  
Đường dẫn DLL và LoadLibraryA():  
Chúng ta sẽ xác định địa chỉ của hàm LoadLibraryA() và chuyển nó đến hàm thực thi cùng với tham số truyền vào là địa chỉ vùng nhớ chứa đường dẫn đến DLL. Để lấy địa chỉ của hàm LoadLibraryA(), ta sẽ sử dụng GetModuleHandle() và GetProcAddress():

loadLibAddr = GetProcAddress(GetModuleHandle(TEXT("kernel32.dll")), "LoadLibraryA");

Toàn bộ DLL và DLLMain  
Bằng cách này chúng ta sẽ tránh được việc đăng ký DLL với chương trình. Tuy nhiên phần khó thực hiện là lấy entry point của DLL khi nó được ghi vào trong vùng nhớ. May mắn là chúng ta đã có sẵn hàm tìm entry point DLL của Stephen Fewer – người đi đầu trong kỹ thuật DLL Injection ở đây:

https://github.com/stephenfewer/ReflectiveDLLInjection/

dwReflectiveLoaderOffset = GetReflectiveLoaderOffset(lpBuffer);

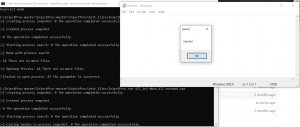
### ****Thực thi DLL****

Hiện tại, DLL đã nằm trong vùng nhớ của process và chúng ta đã có địa chỉ của vùng nhớ đó. Việc cuối cùng cần làm là cho process thực thi nó. Có một vài cách để process thực thi DLL nhưng trong phạm vi bài viết này Ta chỉ trình bày cách sử dụng CreateRemoteThread(). CreateRemoteThread() cũng là cách được sử dụng rộng dãi nhất.

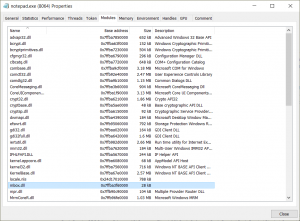
rThread = CreateRemoteThread(hTargetProcHandle, NULL, 0, lpStartExecAddr, lpExecParam, 0, NULL);

WaitForSingleObject(rThread, INFINITE);

WaitForSingleObject() để chắc chắn rằng DLL đã được thực thi trước khi Windows thực thi các công việc tiếp theo của process.  
Sau đây là một vài hình ảnh khi Ta demo kỹ thuật.



Thư viện mbox.dll Ta inject vào notepad đã có chứa file mbox.dll



### ****Cách phát hiện****

Có nhiều cách để phát hiện ra kỹ thuật DLL Injection, nhưng cách phổ biến là kiểm tra danh sách các module import và tìm các vùng nhớ có quyền đọc ghi của process.

Trên đây Ta đã trình bày chi tiết kỹ thuật DLL Injection. Ta đánh giá đây là một kỹ thuật rất mạnh mẽ, linh hoạt trong nhiều trường hợp và cũng khá là đơn giản để sử dụng, hầu như có thể áp dụng trong mọi trường hợp mà bạn muốn. Hẹn gặp lại mọi người trong những bài viết sau.