DLL Injection with LoadLibraryA
By βπα- κ θ745

This paper will talk about the injection of a DLL in a remote process (so not necessary to remember this will be an Dynamic) by the method known for using LoadLibrary as the point of injection.

The trick in here will be to make to a remote process load a dll that we write on the system. So from here we can advance some disadvantages:

- DLL must be on the system (we can dump it or download it inside of our injector).

- DLL can only be loaded once, because we are using LoadLibrary. Once the remote process load our DLL assign it an address, and if we try to load again LoadLibrary would just return us the address assigned before.

- The name of the DLL will appear in the list of dlls of the process.

- DLLs of 32 bits can only be injected in processes of 32 bits, and DLLs of 64 bits inside of process of 64 bits.

So before of starting you should have some knowledge about C++, Microsoft Windows, Win32Api, etc.

Injection Technique

First we are going to explain about the dll injection technique based on LoadLibrary, let's going to see the steps:

1. Get the address of the function LoadLibrary, so we will create a remote thread using as function to execute LoadLibrary.

2. Open remote process to get a handle for subsequent calls.

3. Reserve memory in the remote process with a size equals to path to dll.

4. Copy dll path to remote process.

5. Create thread in remote process to execute LoadLibrary with address of dll path string as parameter.

Because we will use ASCII, we must use ASCII winapi functions (on this case LoadLibraryA instead of LoadLibraryW).

Functions we are going to use

```cpp
HMODULE WINAPI GetModuleHandle(
  _In_opt_ LPCTSTR lpModuleName
);
```

GetModuleHandle: to get address of "kernel32.dll" in our program.
lpModuleName will be the name of this dll.

```cpp
FARPROC WINAPI GetProcAddress(
  _In_ HMODULE hModule,
  _In_ LPCSTR  lpProcName
);
```

GetProcAddress: to get address of "LoadLibraryA". hModule will be
handler given by GetModuleHandle and lpProcName will be ascii name of
function.

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

OpenProcess: to open remote process and get a handle for subsequent
calls, we will have to give as parameters the desired Access for the

process, if we want that process created by our process inherit this handle and the process id (PID) of remote process.

Opening a process is a complex question, because we will need enough permissions, usually process created by user will have the same permissions (if UAC was not activated on that process). A process that malware authors usually use to inject in, is explorer.exe, because user will have enough permissions to inject a dll there.

```cpp
LPVOID WINAPI VirtualAllocEx(
  _In_     HANDLE hProcess,
  _In_opt_ LPVOID lpAddress,
  _In_     SIZE_T dwSize,
  _In_     DWORD  flAllocationType,
  _In_     DWORD  flProtect
);
```

VirtualAllocEx: this function will allow us to get virtual memory in remote process to write path to DLL. We will have to give as parameter the handle obtained by OpenProcess, desired address (we can set to NULL and get one), size of virtual memory, allocation type (will see), al flProtect where we set permissions for that virtual memory.

```cpp
BOOL WINAPI WriteProcessMemory(
  _In_  HANDLE  hProcess,
  _In_  LPVOID  lpBaseAddress,
  _In_  LPCVOID lpBuffer,
  _In_  SIZE_T  nSize,
  _Out_ SIZE_T  *lpNumberOfBytesWritten
);
```

WriteProcessMemory: will allow us to write DLL path into virtual memory of remote process, for that reason, we use handle from OpenProcess, the address from VirtualAllocEx, buffer with the dll path, etc.

Starting with the code

Now we are going to start with the code, for this example I will use Visual Studio 2017.

Start creating a "Windows desktop" Project, and here choose "Windows console application"

So now, let's going to start programming, we will take our main function and add some headers:

```
#include "stdafx.h"
// C++ standard functions
#include <iostream>
// WinApi functions
#include <Windows.h>
```

As you can see here, I will use "iostream" for C++ standard functions (we will use std namespace) and "Windows.h" to use all the Windows api functions.

Now write main function with two parameters: argc (count of arguments) and argv (pointer to array of chars with the arguments). After this main, we will declare all our variables and some code to check program arguments are correct:

```
int main(int argc, char *argv[])
{
        std::string path_to_dll;
        std::uint32_t process_pid;
        HMODULE kernel32_address;
        FARPROC loadLibary_address;
        HANDLE procHandle;
        void *baseAddress;
        HANDLE threadHandle;
        std::uint32_t bytes_written;

        if (argc != 3)
        {
                fprintf(stderr,
                        "[-] USAGE: %s <pid> <path_to_dll>\n\n",
                        argv[0]);
                return -1;
        }
```

As you can see, we have enough variables for all our functions, and finally we check if user gave us enough arguments.

Now first part, we will open remote process and check for errors, this will be easy as this code:

```
process_pid = std::atoi(argv[1]);
path_to_dll = argv[2];
// First start opening remote process
procHandle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, process_pid);
if (procHandle == NULL)
        throw std::exception("[ERROR] Error in
        OpenProcess",GetLastError());
```

We request the operating system to give us the handle of the process, requesting the desired access "PROCESS_ALL_ACCESS", with this we'll can write to remote process, read, create new thread, etc.

```
// Allocate virtual memory in remote process
baseAddress = VirtualAllocEx(procHandle,
        NULL,
        path_to_dll.size(),
        MEM_COMMIT | MEM_RESERVE,
        PAGE_READWRITE);
if (baseAddress == NULL)
        throw std::exception("[ERROR - VirtualAllocEx]",GetLastError());
```

This code snippet show how we can allocate virtual memory on remote process, for that instead of using VirtualAlloc, we will use VirtualAllocEx to

specify a handle to the remote process, so the allocated memory will be in remote. We set MEM_COMMIT and MEM_RESERVE, to get the memory and reserve it for later, and finally PAGE_READWRITE (because we will write there right now).

And as we said, is time to write the dll path in the remote process, for that task we will use WriteProcessMemory, specifying handle we obtained in OpenProcess:

```
if (!WriteProcessMemory(procHandle,
        baseAddress,
        path_to_dll.c_str(),
        path_to_dll.size(),
        reinterpret_cast<SIZE_T*>(&bytes_written)))
{
        throw std::exception("Error writing in remote process");
}
```

Here we are joining the pieces of the puzzle, we are using handle of OpenProcess, the path to the dll, size of that path, and finally the address of bytes_written.

Now we will need address of "LoadLibraryA", we can do it in this way:

```
kernel32_address = GetModuleHandleA("Kernel32");
loadLibary_address = GetProcAddress(kernel32_address, "LoadLibraryA");
```

Get first pointer to kernel32 and finally address of LoadLibraryA (one day we will try to reverse engineer "GetProcAddress" a function that does more than function resolving).

Finally, we are going to create a thread in remote process, to do this, we have to order the process, to create the thread in a specific function, this function will be "LoadLibraryA", and we can specify parameters for the function, in this case, the address where we copied the DLL path.

So let's going to holy dive into the last part of the code:

```
threadHandle = CreateRemoteThread(procHandle,
        NULL,
        0,
        reinterpret_cast<LPTHREAD_START_ROUTINE>(loadLibary_address),
        baseAddress,
        0,
        NULL);
if (threadHandle == NULL)
        throw std::exception("Error creating remote thread");

    return 0;
}
```

This will create a thread and load our DLL, once our DLL is loaded, will execute DLL main, and one of the parameters of that dll main will be `DWORD ul_reason_for_call` this will be our value for the switch case, where will select between four reasons:

- DLL_PROCESS_ATTACH: The DLL is being loaded into the virtual address space of the current process as a result of the process starting up or as a result of a call to **LoadLibrary**.

- DLL_THREAD_ATTACH: The current process is creating a new thread. When this occurs, the system calls the entry-point function of all DLLs currently attached to the process. The call is made in the context of the new thread.

- DLL_THREAD_DETACH: A thread is exiting cleanly.

- DLL_PROCESS_DETACH: The DLL is being unloaded from the virtual address space of the calling process because it was loaded unsuccessfully or the reference count has reached zero (the processes has either terminated or called FreeLibrary one time for each time it called LoadLibrary).

We will use DLL_PROCESS_ATTACH to execute something in our testing DLL, we will just open a MessageBoxA with a message.

We have to create a new Project in visual studio, and inside of process choose Dynamic link library (DLL), and in the file dllmain.cpp, copy this code:

```
#include "stdafx.h"
#include <Windows.h>

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
            MessageBoxA(NULL,
                    "Injection Done",
                    "BetaPiAlpha",
                    MB_OK);
            break;
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```

Now, compile both Project (Debug or release) and going to test in a VM of 32 bits:
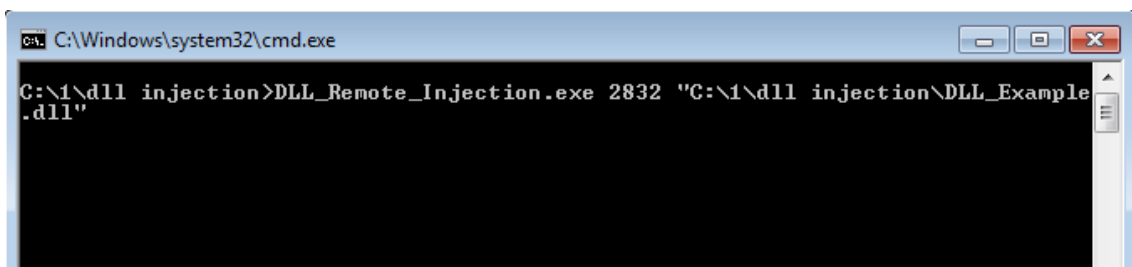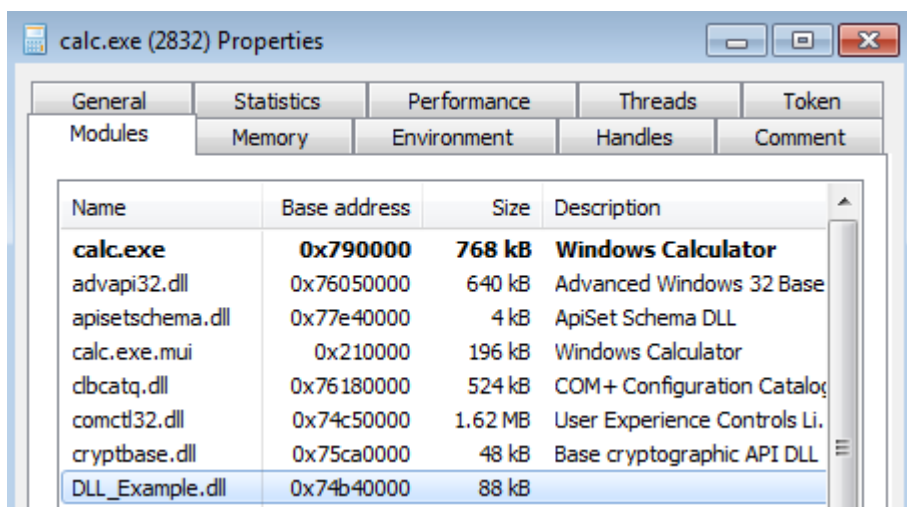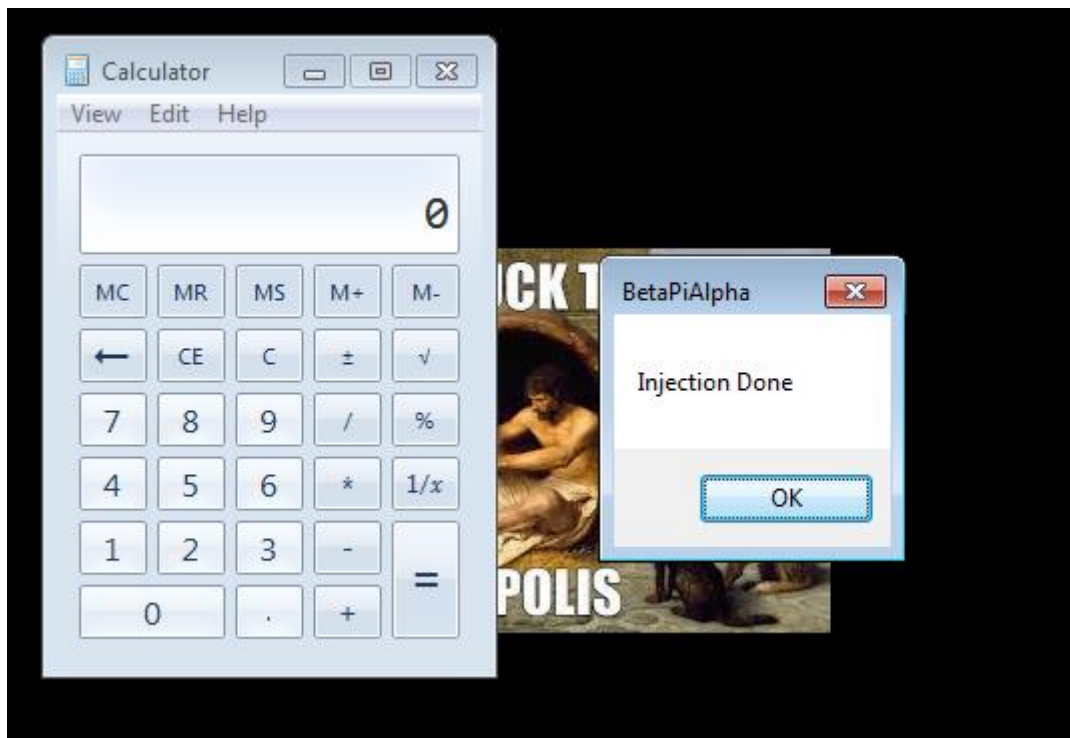
As you can see, here we have injected the dll, create remote thread, and finally we can see in process hacker, how the DLL has been injected with the name given. If we want to inject again the dll, we should change the name of the dll (if not LoadLibraryA will return the address of the actual dll).

Final tips

If you have problem with some visual studio dll as msvcp140.dll, you can check this post in stackoverflow:
https://stackoverflow.com/questions/32998902/msvcp140-dll-missing

The path to the DLL must be complete path from root, and once you have injected code into othe process you will can access remote process memory, do IAT hooking, read values, modify values, etc.

In next papers we will see how to do the DLL loading manually, copying headers, sections, resolving relocs and imports, etc.

"Then you will know the truth, and the truth will set you free." John 8:32