Anti-Attach trick hooking
By βπα-κ θ745

This paper will talk about a little trick to avoid a debugger to attach to a process. In this trick I will show where a debugger usually stops when attach to a process, and how to defeat them.

So let’s start checking how a debugger attach to a process.

When a debugger attach to a process, arrives to a function in Ntdll.dll, DbgUiIssueRemoteBreakin. This function will be the one which start a new thread for the debugger:

```
77F5F22C ; int __stdcall DbgUiIssueRemoteBreakin(HANDLE Handle)
77F5F22C public DbgUiIssueRemoteBreakin
77F5F22C DbgUiIssueRemoteBreakin proc near
77F5F22C
77F5F22C var_8= byte ptr -8
77F5F22C Handle= dword ptr  8
77F5F22C
77F5F22C mov     edi, edi
77F5F22E push    ebp
77F5F22F mov     ebp, esp
77F5F231 push    ecx
77F5F232 push    ecx
77F5F233 push    esi
77F5F234 push    edi
77F5F235 lea     eax, [ebp+var_8]
77F5F238 push    eax
77F5F239 xor     esi, esi
77F5F23B lea     eax, [ebp+Handle]
```

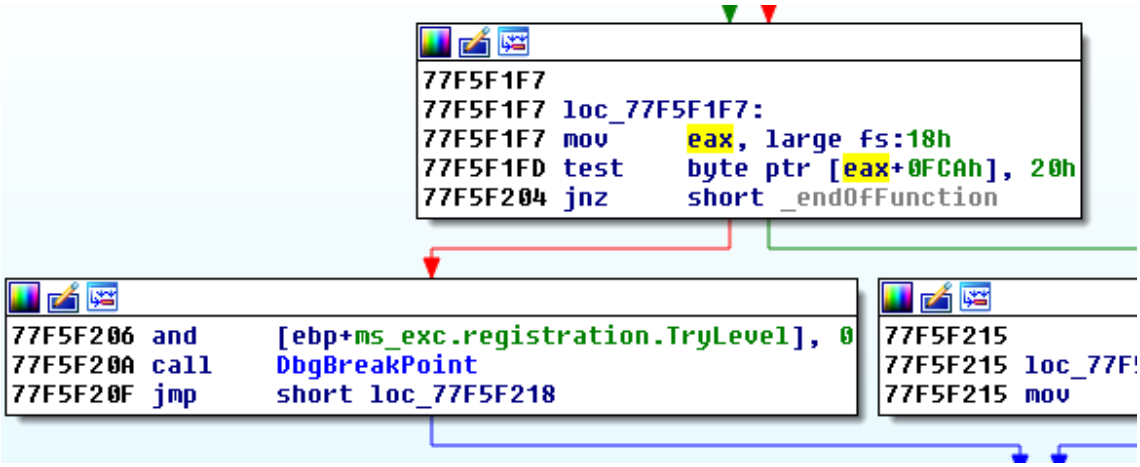
```
77F5F23E push    eax
77F5F23F push    esi
77F5F240 push    offset DbgUiRemoteBreakin
77F5F245 push    esi
77F5F246 push    4000h
77F5F24B push    esi
77F5F24C push    esi
77F5F24D push    2
77F5F24F push    esi
77F5F250 push    [ebp+Handle]
77F5F253 call    RtlpCreateUserThreadEx
77F5F258 mov     edi, eax
77F5F25A cmp     edi, esi
77F5F25C jl      short loc_77F5F266
```

As you can see, this thread will execute DbgUiRemoteBreakin, so let's dig into this function:

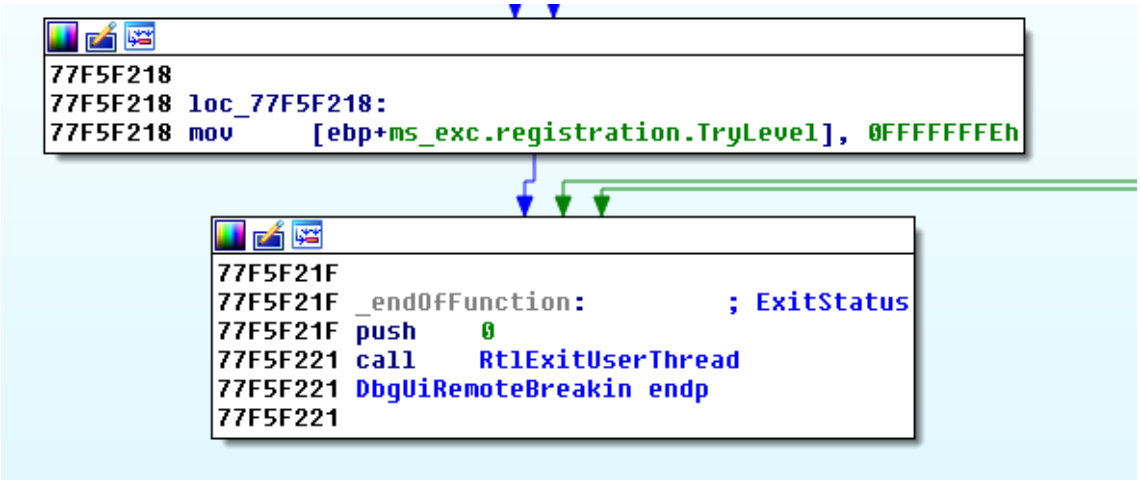
```
77F5F1D3
77F5F1D3 public DbgUiRemoteBreakin
77F5F1D3 DbgUiRemoteBreakin proc near
77F5F1D3
77F5F1D3 ms_exc= CPPEH_RECORD ptr -18h
77F5F1D3
77F5F1D3 push    8
77F5F1D5 push    offset stru_77F107E8
77F5F1DA call    __SEH_prolog4
77F5F1DF mov     eax, large fs:18h
77F5F1E5 mov     eax, [eax+30h]
77F5F1E8 cmp     byte ptr [eax+2], 0
77F5F1EC jnz     short loc_77F5F1F7
```

Here, you see that the first thing that checks this DbgUiRemoteBreakin is the flag "BeingDebugged" from PEB structure (you can find more information here: https://www.aldeid.com/wiki/PEB-Process-Environment-Block).

Let's going to see the other interesting function that is called:



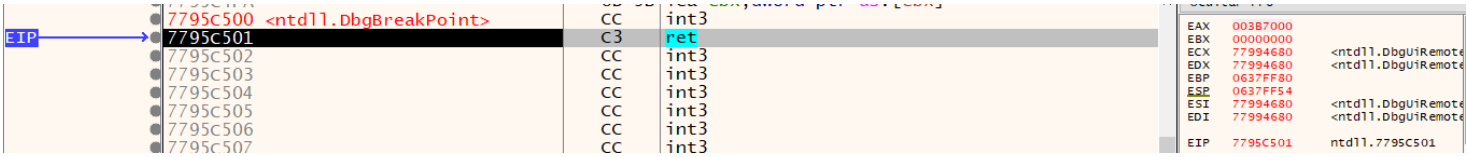
First of all, we can see how eax has the address of the TEB (Thread Information Block), and make an AND operation with the offset 0xFCAh of the TEB (you can see the value of this bit in that offset with WinDBG, this value is the offset RanProcessInit), and the value 0x20h, if result is not 0 jumps to the end of the function that it is a RtlExitUserThread:



If everything is okay, function will jump to DbgBreakPoint, that looks like this disassembly:

```
77EF4108 ; Exported entry  49. DbgBreakPoint
77EF4108
77EF4108
77EF4108
77EF4108 ; void DbgBreakPoint(void)
77EF4108 public DbgBreakPoint
77EF4108 DbgBreakPoint proc near
77EF4108 int     3               ; Trap to Debugger
77EF4109 retn
77EF4109 DbgBreakPoint endp
77EF4109
```

This function can vary depending of windows' version. This will be the function that we are going to hook.



First, let's going to see the code where function is going to jump:

```
void anti_attach()
{
	FatalAppExitA(MB_ICONWARNING, "Don't try to attach MOTHERFUCKER");
	ExitThread(0);
}
```

As MessageBoxA crashes to me, I'm going to use FatalAppExitA from kernel32.dll, you can almost use whatever you want, for example, you can try to shutdown computer and debugger will execute that code, after that an ExitThread finishing execution.

Before showing how to hook that api function, I'm going to show how to get the token for SeDebugPrivilege:

```
bool EnableSeDebugPrivilege()
{
	HANDLE process = GetCurrentProcess();
	HANDLE processToken;
	if (!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES,
&processToken))
	{
		printf("Error OpenThreadToken: %d\n", GetLastError());
		return false;
	}


	LUID privilege_value;

	if (LookupPrivilegeValue(
		NULL,
		SE_DEBUG_NAME,
		&privilege_value
	))
	{
		TOKEN_PRIVILEGES new_state;
		new_state.PrivilegeCount = 1;
		new_state.Privileges[0].Luid = privilege_value;
		new_state.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

		if(AdjustTokenPrivileges(
			processToken,
			0,
			&new_state,
			sizeof(TOKEN_PRIVILEGES),
			0,
			0
		))
		{
			printf("Obtained SeDebugPrivilege\n");
			return true;
		}
		printf("AdjustTokenPrivileges, last error: %d\n", GetLastError());
		return false;
	}
	printf("LookupPrivilegeValue, last error: %d\n", GetLastError());
	return false;


}
```

With this series of api functions, you can ask for SeDebugPrivilege, and finally enable it with AdjustTokenPrivileges, maybe it's not necessary for this, but we will enable it to modify Ntdll.dll code.

Well, let’s going to dig, in the code to hook that API function. First of all, we need to know the address in runtime, for that, we can use the next APIs:

```C++
HMODULE WINAPI LoadLibrary(
  _In_ LPCTSTR lpFileName
);
```

LoadLibraryA to get a handle to Ntdll.dll.

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

GetProcAddress to get the address of the function we are going to hook.

So these would be the first lines:

```
HMODULE ntdll = LoadLibraryA("ntdll.dll");
FARPROC DbgBreakPoint = GetProcAddress(ntdll, "DbgBreakPoint");
```

Once we have the address its necessary to change permissions on Ntdll to modify it, even if we change just some bytes, it will modify the permissions of all the page usually 4KB (because of how it works the permissions on Windows, based on pages of memory).

Let’s see the code:

```
VirtualProtect(reinterpret_cast<LPVOID>(DbgBreakPoint), 100,
PAGE_EXECUTE_READWRITE, reinterpret_cast<PDWORD>(&oldProtect));
```

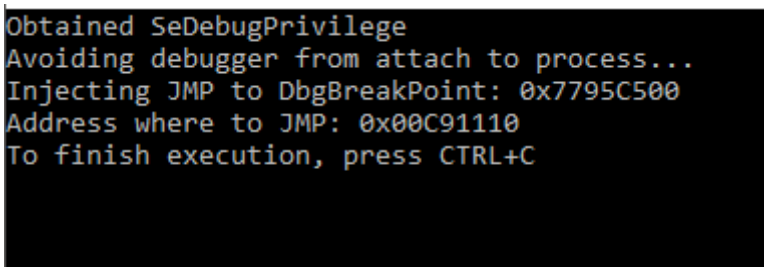
And finally the injection to jump to the address, this time instead of using JMP <displacement> I used the push-ret technique, so I just push the address of the function on the stack and finally ret to that address.

```
std::uint8_t PUSH = 0x68;
memcpy(reinterpret_cast<void *>(reinterpret_cast<std::uintptr_t>(DbgBreakPoint)),
&PUSH, 1);

std::uintptr_t address_to_jmp = (reinterpret_cast<std::uintptr_t>(anti_attach));
memcpy(reinterpret_cast<void*>(reinterpret_cast<std::uintptr_t>(DbgBreakPoint)+1)
, &address_to_jmp, sizeof(std::uintptr_t));

std::uint8_t RET = 0xC3;
memcpy(reinterpret_cast<void *>(reinterpret_cast<std::uintptr_t>(DbgBreakPoint) +
5), &RET, 1);
```

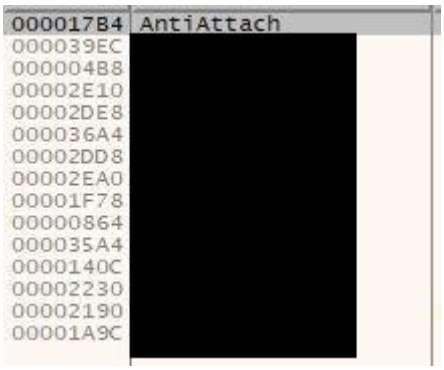
And finally a Sleep function to wait for debugger:

```
printf("To finish execution, press CTRL+C");
Sleep(INFINITE);
```

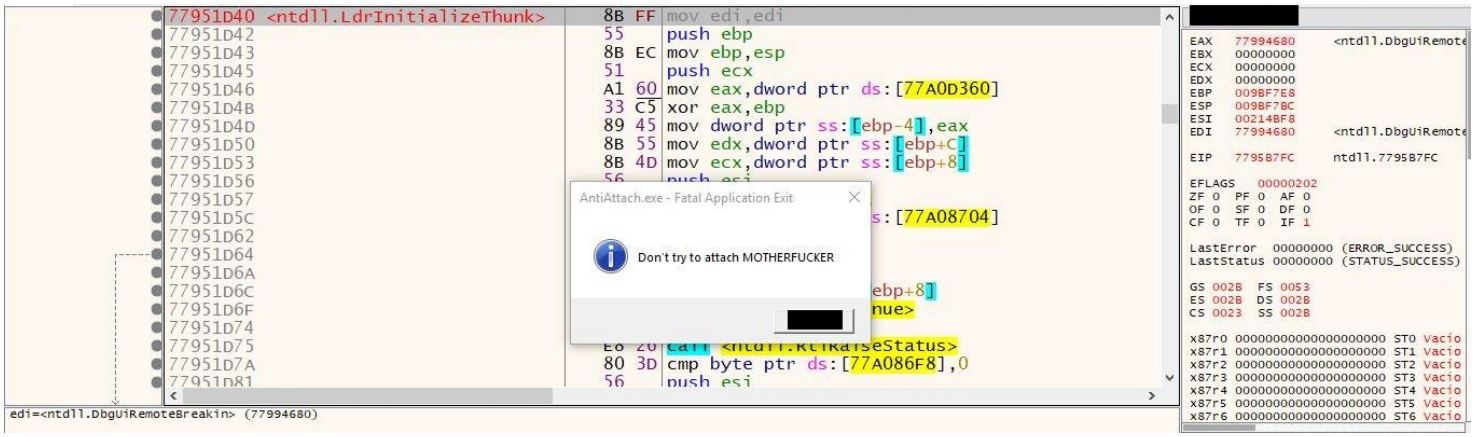
Well, let’s going to compile and try it:



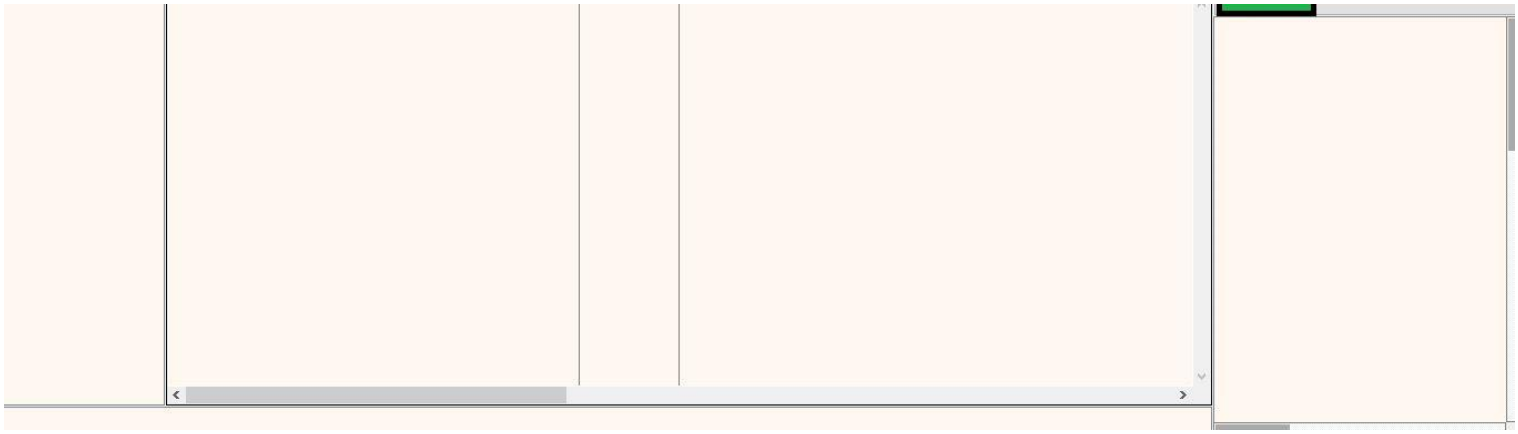
And now open the debugger, and try to attach:



Let’s attach to the process:



In the moment we attach to the process, the message box appears. So press OK in the message box, to see how the ExitThread execute:



So we’ve seen that with a simple API Hooking, we can modify the behaviour of the debugger because the debugger use this api hooked to synchronize and stop to debug, we could hook another function for example we can hook DbgUiRemoteBreakin, and would be the same behaviour.

Well, thanks for reading, and we will see other anti-debugging tricks in another paper.

Thank you.

"The path of the righteous man is beset on all sides by the inequities of the selfish and the tyranny of evil men. Blessed is he who, in the name of charity and good will, shepherds the weak through the valley of darkness, for he is truly his brother's keeper and the finder of lost children. And I will strike down upon thee with great vengeance and furious anger those who attempt to poison and destroy my brothers. And you will know my name is the Lord when I lay my vengeance upon thee." Ezekiel 25:17 (Pulp Fiction)