Thread Injection

By Nick Cano

This tutorial for Thread Injection is intended for x86 processes. Due to slight differences in registers it wont work on x64 processes, but it can easily be converted.

Introduction:

Code-caving is the practice of injecting machine code into a remote process and making it execute. In this tutorial, I will cover a method of code-caving which I like to call thread injection. Thread injection is a seven step process.

1. Detect target process
2. Identify main thread
3. Suspend main thread
4. Obtain thread context
5. Create and write the code-cave
6. Spoof instruction pointer to execute the code-cave
7. Resume the thread, continue execution, and free memory

Step 1: *Detect target process*

The first step to creating a code-cave is identifying the process into which we want to inject our code. There is literally a multitude of ways to doing this – find the one that works best for your program. For examples sake, I will simply find the process by window name. To do this, I utilize two WinAPI functions: *FindWindow()* and *GetWindowThreadProcessId()*.

```
DWORD FindProcessByWindowName(char* windowName)
{
	DWORD procID = NULL;
	HWND window = FindWindowA(NULL, windowName);

	if (window)
		GetWindowThreadProcessId(window, &procID);

	return procID;
}
```

Step 2: *Identify main thread*

Our next step is to identify the main thread of our process. This can also be done a number of ways. The easiest method is to call *GetWindowThreadProcessId()* again, but I personally prefer to pull it from a structure called the TIB (*Thread Information Block)*. The TIB is a windows structure which holds data about the currently running thread. The TIB is stored by the FS process register, and can be obtained by reading the FS register. One may ask themselves *"If we need to use a register to identify the TIB, wouldn't that mean we would have to code-cave to obtain it?"* The answer is no. Every process on the system stores the TIB at the same memory location. Moreover, the TIB container a pointer to itself at offset 0x18. What this means is that we can simply find the TIB for our current process, obtain its memory location and read data at that address from our target process.

```
DWORD pointerTID;
_asm
{
        MOV EAX, FS:[0x18]
        MOV [pointerTID], EAX
}
```

Above you can see code which will locate the address of the TIB. From here there are two ways we can proceed:

1. Read a 4-byte value from pointerTID + 0x20 *(the structural offset for CurrentThreadID is 0x20 bytes)*
2. Create a structure defining the TIB and read the whole block

Both of the methods mentioned above will be shown below, but this tutorial will proceed using the second method.

```
DWORD threadID;
HANDLE hProcess = OpenProcess(PROCESS_VM_READ, false, procID);
ReadProcessMemory(hProcess, (LPVOID)(pointerTID + 0x20), &threadID, 4, NULL);
CloseHandle(hProcess);

return threadID;
```

The method shown above will suffice in nearly all cases. It is, however, helpful to have the entire TIB in case we want to work with certain things like the PEB *(Process Enviroment Block)*. For this reason, I will show how to read the TIB into a structure as a proof-of-concept. However, since we only need certain data, I will only use a partial structure.

```
struct partialTIB
{
        DWORD SEHFrame;
        DWORD StackTopPointer;
        DWORD StackBottomPointer;
        DWORD Unknown;
        DWORD FiberData;
        DWORD ArbitraryDataSlot;
        DWORD LinearAddressOfTIB;
        DWORD EnviromentPointer;
        DWORD ProcessID;
        DWORD CurrentThreadID;
};
```

```
partialTIB TIB;
HANDLE hProcess = OpenProcess(PROCESS_VM_READ, false, procID);
ReadProcessMemory(hProcess, (LPVOID)pointerTID, &TIB, sizeof(partialTIB), NULL);
CloseHandle(hProcess);

return TIB;
```

Step 3: *Suspend main thread*

Now that we have detected our target thread, we must suspend its execution. To do this, we use two WinAPI functions: *OpenThread()* and *SuspendThread()*. Since we must open the thread again to get its context and resume it, we can leave it opened and store the HANDLE for efficiency.

```
HANDLE OpenAndSuspendThread(DWORD threadID)
{
        DWORD ACCESS =
        THREAD_GET_CONTEXT | THREAD_SUSPEND_RESUME | THREAD_SET_CONTEXT;

        HANDLE thread = OpenThread(ACCESS, false, threadID);
        SuspendThread(thread);
        return thread;
}
```

Step 4: *Obtain thread context*

This is another very simple step. We will call one WinAPI function, *GetThreadContext()*, in order to obtain the control context of the thread. The control context will give us the current instruction pointer which we will use in our code-cave to return back to regular execution.

```
CONTEXT RetriveThreadControlContext(HANDLE thread)
{
        CONTEXT threadContext;
        threadContext.ContextFlags = CONTEXT_CONTROL;
        GetThreadContext(thread, &threadContext);
        return threadContext;
}
```

Step 5: *Create and write the code-cave*

This next step is very trivial. Creating the proper code-cave will all depend on what you want it to achieve. In most cases, we will want to preserve our registers and flags using *PUSHAD* and *PUSHFD*, restoring them after we execute our code. That is, however, a new tutorial in itself. As a proof-of-concept, I will simply create a code-cave which does nothing but return back into normal execution. In order to simply return to execution, we must make the process think that code at the address we obtained earlier *(the current instruction from the thread context)* was a piece of code that called our cave. We do this by doing a *PUSH* which will put the instruction pointer on the top of the stack. To then return execution to that code, we execute a RETN operation. This simple code-cave should be the wrapper for any more advanced code-caves, which would be nested after the *PUSH* and before the *RETN*. The first step to creating the previously explained code-cave is to allocate enough memory to hold all of the code.

```
LPVOID codeCave =
        VirtualAllocEx(process, NULL, 6,
        MEM_COMMIT, PAGE_EXECUTE_READWRITE);
```

Once we have allocated the memory, our next step is to write the code-cave to it. As with previous tasks, this task can also be done a number of ways. The most common of which is to write a naked function using inline-assembly and copy the code into a buffer using *memcpy()*. However, I prefer writing my code-caves in bycode – it becomes much easier to add dynamic values into the code.

```
DWORD push = 0x68;
DWORD retn = 0xC3;

WriteProcessMemory(process, codeCave, &push, 1, NULL); // "PUSH" opcode
WriteProcessMemory(process, (LPVOID)((DWORD)codeCave+1), &InstructPtr 4, NULL); //return adr
WriteProcessMemory(process, (LPVOID)((DWORD)codeCave+5), &retn, 4, NULL); //"RETN" opcode
```

Step 6: *Spoof instruction pointer to execute the code-cave*

The last step before resuming thread execution is telling the thread to execute our code-cave. We do this by using *SetThreadContext()* with the context we obtained earlier. The only thing we must do is change the instruction pointer to the address of our codecave.

```
threadContext.Eip = (DWORD)codeCave;
threadContext.ContextFlags = CONTEXT_CONTROL;
SetThreadContext(thread, &threadContext);
```

Step 7: *Resume the thread, continue execution, and free memory*

Our final step requires us to call *ResumeThread(),* Wait for our code-cave to execute, and clean up the mess we left behind. It is safe to assume that our code-cave will have executed within two seconds, so we will wait two seconds before using *VirtualFreeEx()* to free our previously allocated memory. We will also close any handles we left open.

```
ResumeThread(thread);
Sleep(2000);

VirtualFreeEx(process, codeCave, 6, MEM_DECOMMIT);
CloseHandle(process);
CloseHandle(thread);
```

Conclusion:

That's it! We've successfully hijacked a running thread, spoofed it's context and executed our own code in a remote process. You should be able to find the full project, along with further tutorials, on the same host as this tutorial.