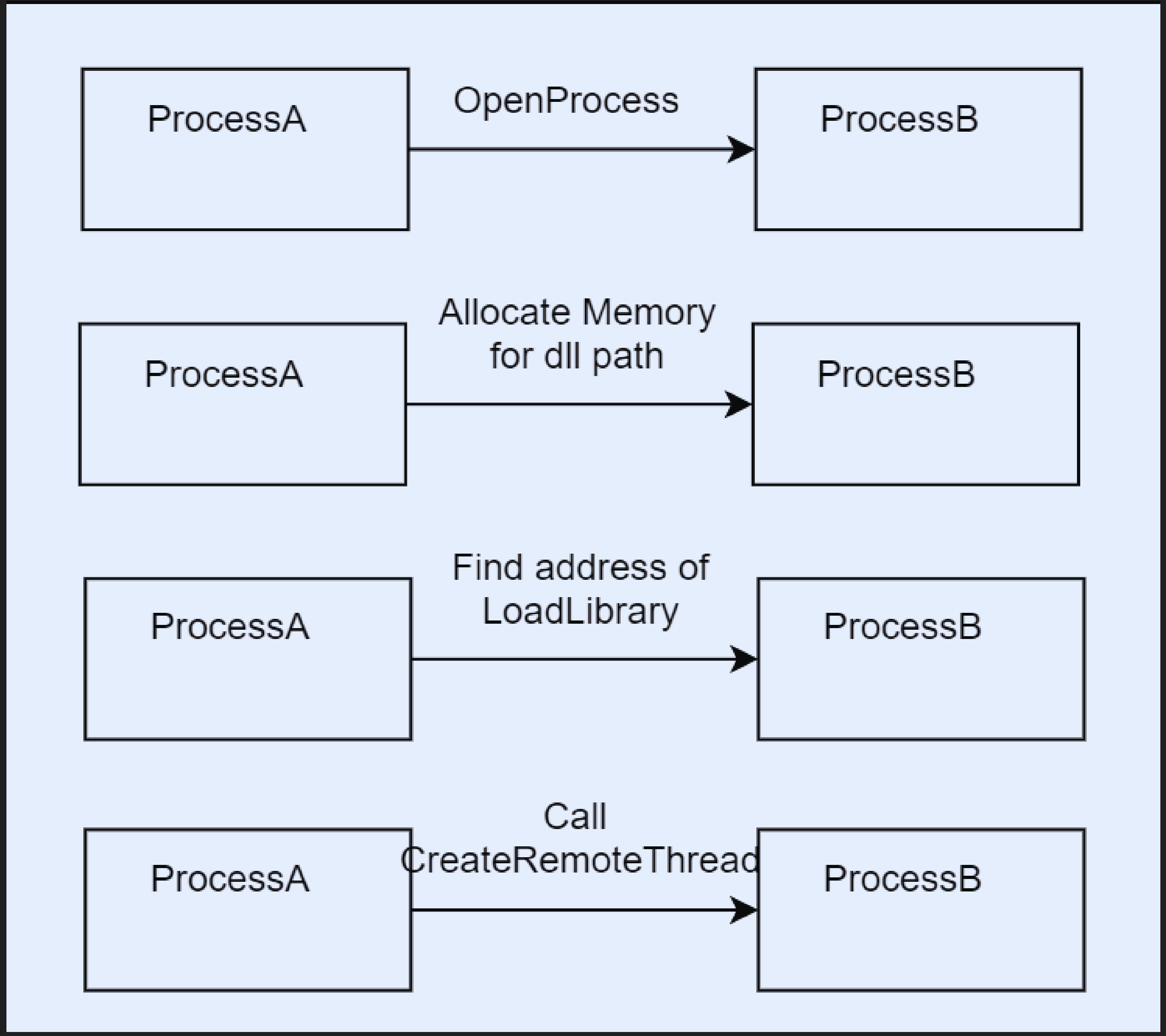
Sx-Cheats

As in the great world of hacking, injection is a very important part, whether for internal cheats or for backdoors, you cannot miss this technique.Today we will see from another angle how it works, here are the points that we will address in this article.

- What is Injector ?
- What are the framework / library used
- DLL injection methods
- How does the injector work

Dll Injection, what is it ?

for the new ones, and to give the taste to certain person, the injection of dll consists in loading a dll file (.dll) in the address space of the victim process, so you will have access to the address space of the process as if it were your own. Here is a diagram that summarizes what I said.



What are the framework / library used

This project includes Qt and Asmjit, why only these 2 frameworks ?

- Qt offers the possibility of creating graphical interfaces that are pleasant to see and use while remaining at low level (C++), despite some (personal) inconvenience it remains top 1 on the market in terms of low level graphics

- AsmJit provides the ability to write in assembler (low level language) easily and we don't have to worry about opcode or host machine or worry about jmp/call, it does that for us, great no?, yes but you have to know how to use it, which is not easy

I cited the frameworks used publicly, you will see later that I use other personal "frameworks", I will come back to them on another article if you want.

Here is an asmjit repository that includes static builds and headers : github.com/Sx-Cheats/asmjit

DLL injection methods

There are plenty of known or personal injection methods, here is a small list of the best known methods :

- CreateRemoteThread
- CreateThreadEx
- QueueUserAPC
- ThreadHijacking
- RtlCreateUserThread
- ReflectiveDLLInjection (not mentioned in this article)

of course it is a method known to the general public, it is "easily" detectable by certain AV or EDR. The purpose of this article is to explain how the injection works and NOT to bypass what detects this method

How does the injector work

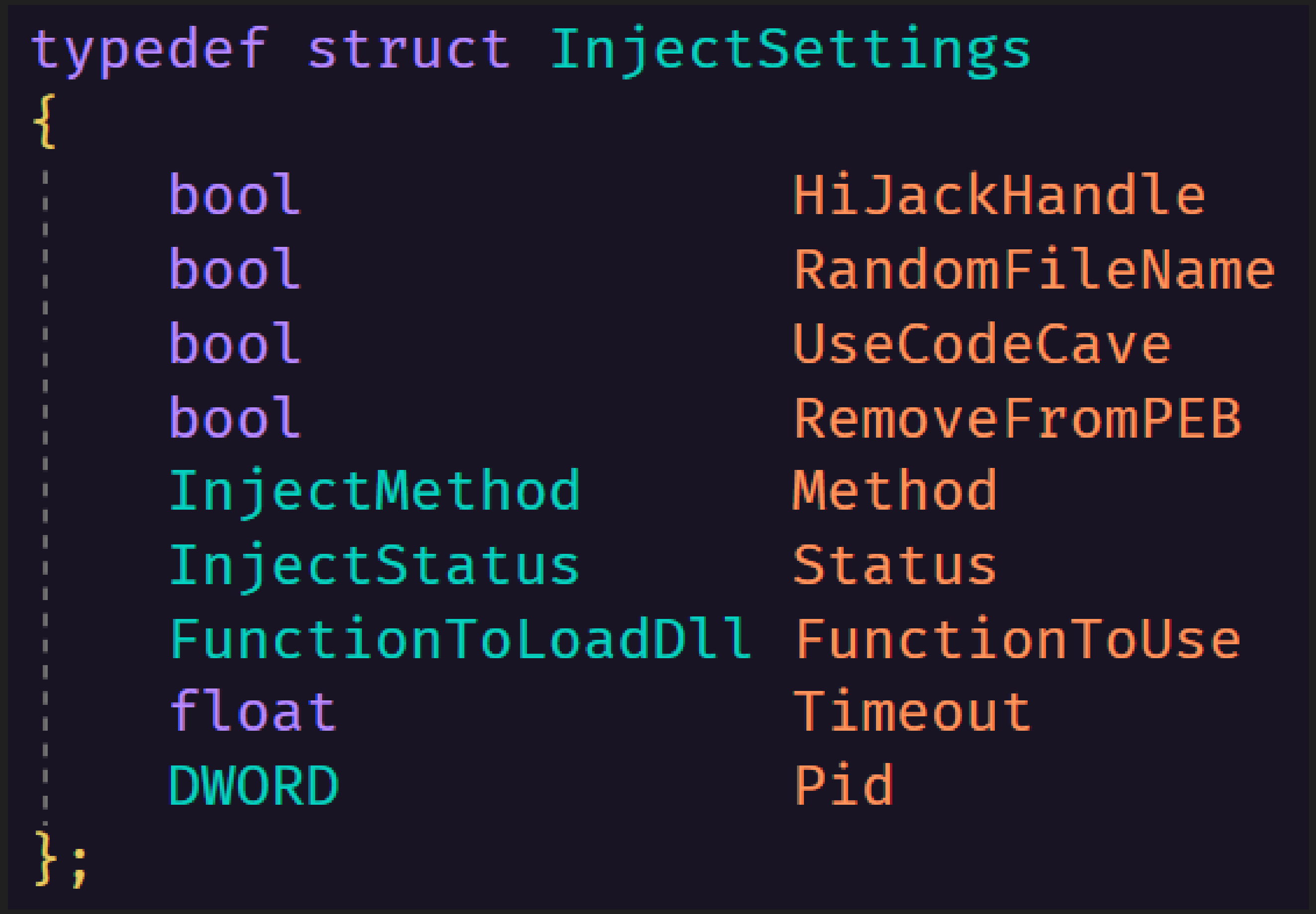
We finally get into the most crisp part of the article, but also the most difficult, follow well!. To begin with, the project is made from A to Z and has a lot of lines, I will summarize the essentials here and explain it to you

Step 1

The injector starts by checking if it is started as an administrator and then gives the privileges to inject without error.

Step 2

It checks and initializes the parameters with the "__check_param__" function, this is what the parameters look like:



```
typedef struct InjectSettings
{
    bool              HiJackHandle
    bool              RandomFileName
    bool              UseCodeCave
    bool              RemoveFromPEB
    InjectMethod      Method
    InjectStatus      Status
    FunctionToLoadDll FunctionToUse
    float             Timeout
    DWORD             Pid
};
```

- HiJackHandle ? Allows to "steal" an already open handle to the process
- RandomFileName ? Allows you to clone the dll with a random name
- UseCodeCave ? Allows to "steal" memory space from the victim process
- RemoveFromPEB ? Hide the dll from the PEB
- Method ? Function used to call the function that loads the dll
- Status ? Current injection status
- FunctionToUse ? Function used to load the dll
- Timeout ? Time before abandoning the injection
- Timeout ? The PID of a victim process

- "__check_param__" essentially checks if the process exists, if the path of the Dll exists as well as the correct extension (.dll) and finally "RandomFileName" is activated (to load a copy of the dll with a random name)

Step 3

the third step is to build a proxy for our dll (using the "__build_proxy__" function), but why a proxy?, that's fine with us because in the future we can write so much more without worrying about where or what calls to make , and some processes could directly detect calls made to the dll loader.

what does the "__build_proxy__" function do ? this is what she does :

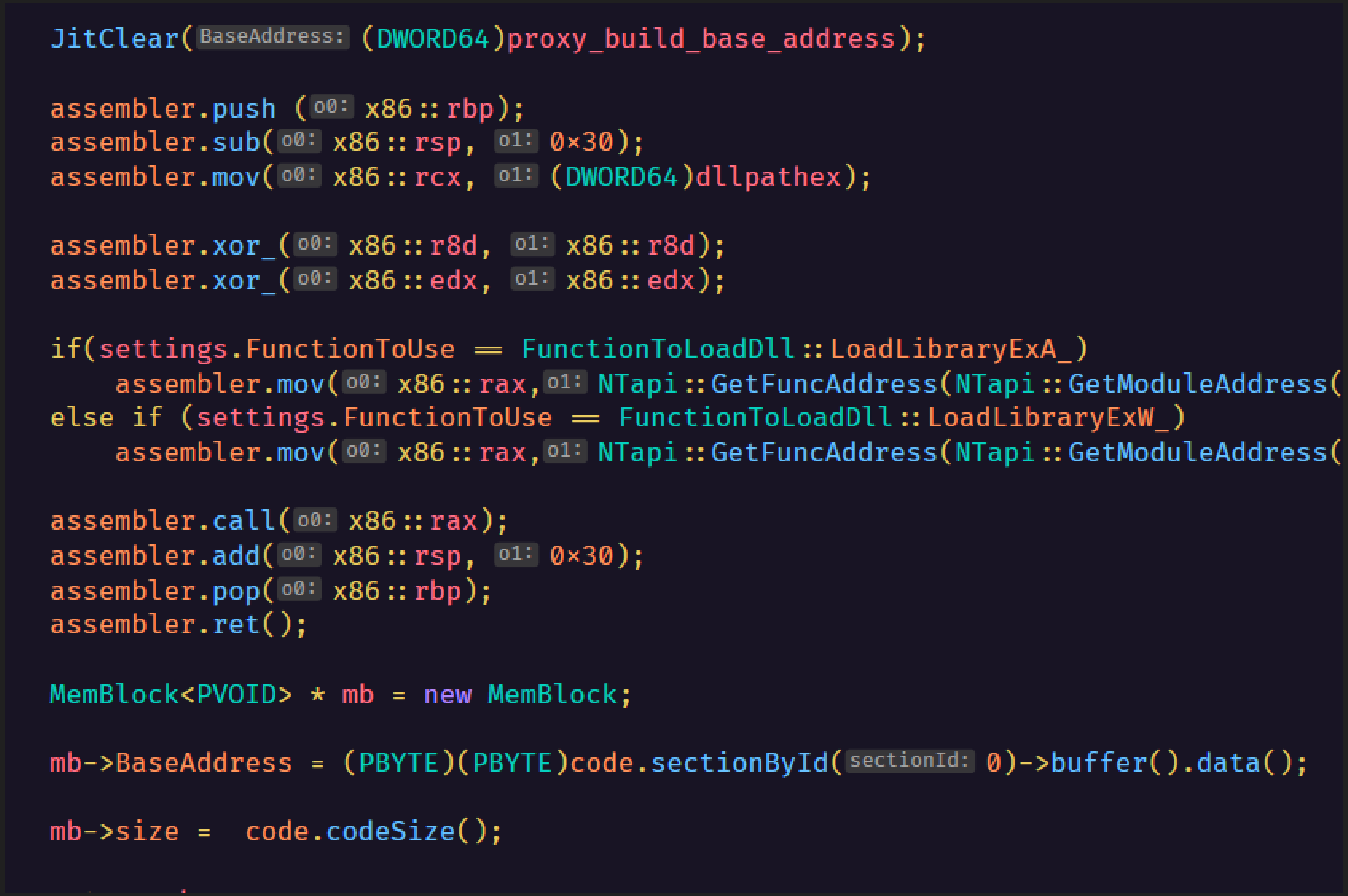
- if "HiJackHandle" is "true", it will try to spoof a Handle already open in the victim process, if it is "false" or unsuccessful it will open a handle to the victim process.

- if "UseCodeCave" is "true" it will try to "steal" memory from the victim process's address space, if it is "false" or finds no memory to "steal" it will allocate memory.

- it writes the path of the dll

- then comes a very useful function which is "__build_caller_dll_loader__" which will build the proxy.

- then comes a very useful function which is "__build_caller_dll_loader__" which will build the proxy. How to build it? well as said above I use 2 frameworks, the first which is QT for the visual and AsmJit to write in assembler easily, here we will use AsmJit, here is an extract of the proxy built with AsmJit :



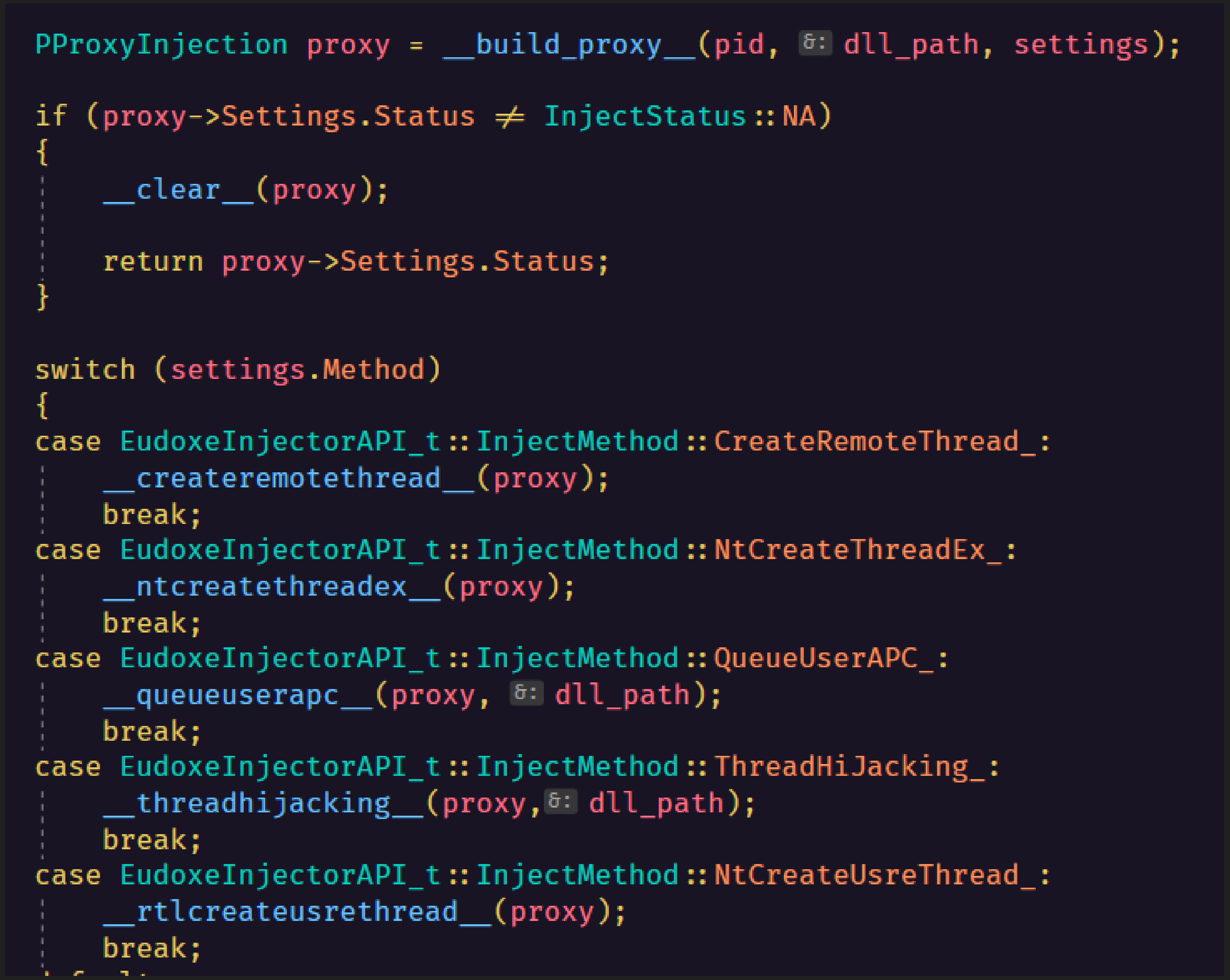
```
JitClear(BaseAddress: (DWORD64)proxy_build_base_address);

assembler.push (o0: x86::rbp);
assembler.sub(o0: x86::rsp, o1: 0x30);
assembler.mov(o0: x86::rcx, o1: (DWORD64)dllpathex);

assembler.xor_(o0: x86::r8d, o1: x86::r8d);
assembler.xor_(o0: x86::edx, o1: x86::edx);

if(settings.FunctionToUse == FunctionToLoadDll::LoadLibraryExA_)
    assembler.mov(o0: x86::rax,o1: NTapi::GetFuncAddress(NTapi::GetModuleAddress(
else if (settings.FunctionToUse == FunctionToLoadDll::LoadLibraryExW_)
    assembler.mov(o0: x86::rax,o1: NTapi::GetFuncAddress(NTapi::GetModuleAddress(

assembler.call(o0: x86::rax);
assembler.add(o0: x86::rsp, o1: 0x30);
assembler.pop(o0: x86::rbp);
assembler.ret();

MemBlock<PVOID> * mb = new MemBlock;

mb->BaseAddress = (PBYTE)(PBYTE)code.sectionById(sectionId: 0)->buffer().data();

mb->size =  code.codeSize();
```

  - To summarize this nice screenshot, we build the Proxy by taking "proxy_build_base_address" as a base to calculate the function calls (call/jmp, simplified) then we write our pointer, which will point to the path of our Dll, then we write the address of the function that will load our dll, and finally we retrieve the base of the proxy that will be our entry point as well as its size.

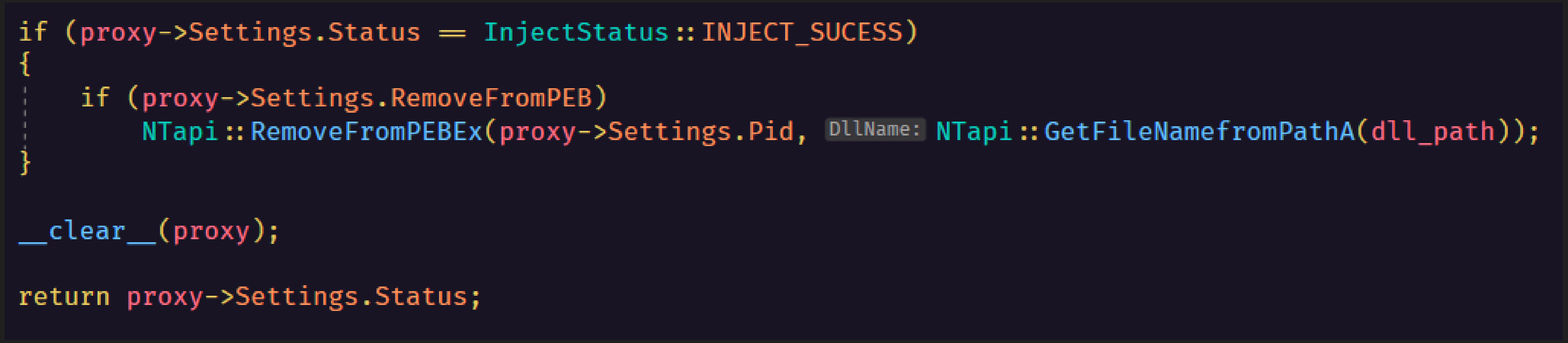
  - then when the proxy has finished being built with AsmJit, then we write it in the process victim.

- We have finished ? NO !, we have one last step, which you guessed it remains to call the entry point of our proxy.

- The following includes 5 functions that can be used to call our breakpoint, it would take too long to describe them one by one. So I'm just going to put a screenshot and do a summary



```cpp
PProxyInjection proxy = __build_proxy__(pid, &: dll_path, settings);

if (proxy->Settings.Status ≠ InjectStatus::NA)
{
    __clear__(proxy);

    return proxy->Settings.Status;
}

switch (settings.Method)
{
case EudoxeInjectorAPI_t::InjectMethod::CreateRemoteThread_:
    __createremotethread__(proxy);
    break;
case EudoxeInjectorAPI_t::InjectMethod::NtCreateThreadEx_:
    __ntcreatethreadex__(proxy);
    break;
case EudoxeInjectorAPI_t::InjectMethod::QueueUserAPC_:
    __queueuserapc__(proxy, &: dll_path);
    break;
case EudoxeInjectorAPI_t::InjectMethod::ThreadHiJacking_:
    __threadhijacking__(proxy,&: dll_path);
    break;
case EudoxeInjectorAPI_t::InjectMethod::NtCreateUsreThread_:
    __rtlcreateusrethread__(proxy);
    break;
```



```cpp
if (proxy->Settings.Status == InjectStatus::INJECT_SUCESS)
{
    if (proxy->Settings.RemoveFromPEB)
        NTapi::RemoveFromPEBEx(proxy->Settings.Pid, DllName: NTapi::GetFileNamefromPathA(dll_path));
}

__clear__(proxy);

return proxy->Settings.Status;
```

- As shown in the screenshot above, it checks if the build was successful if not cleans up our passage. Then he looks at the method to use to load our dll. If the status is "INJECT_SUCESS" and if "RemoveFromPEB" is "true" he hide the dll then cleans the traces of our passage

- WE'VE FINALLY DONE !