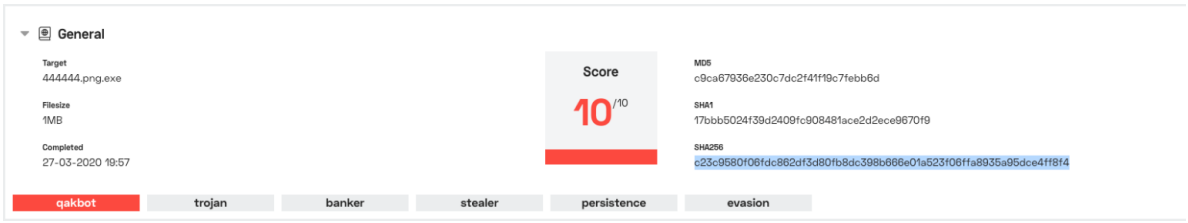
An old enemy – Diving into QBot part 1

By hackingump / March 30, 2020

While checking out the Triage Sandbox[1] I stumbled across upon QBot which I've seen already plenty of times at work at GData Cyberdefense AG[2]. This time I wanted to take a closer look at the sample myself.
The first part of this blog article dives deep into how the packer works.



Triage sandbox overview of the analysed sample

Quick summary

The packer used by this sample first allocates virtual memory and fills it with chunks of bytes from its `.text` section.

After jumping into this allocated area, the address of `GetProcAddress`[3] is determined by looping over the export table of `KernelBase.dll`. This function is then used to load further dependencies.

Next another temporary memory is allocated, filled with decrypted code and replaces the code we started with. Finally the sample jumps back to the now decrypted payload and executes it.

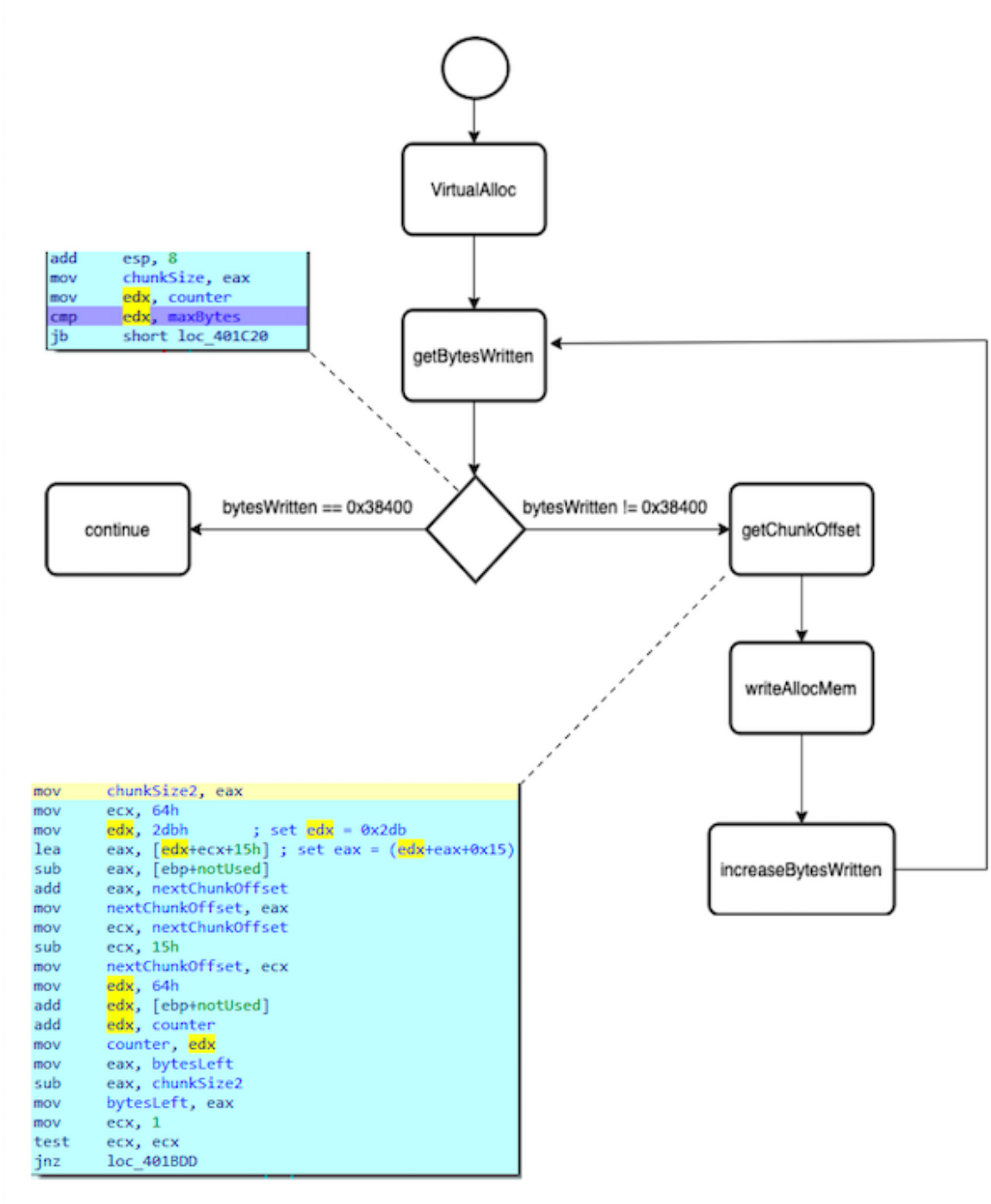
1 – Allocating VirtualAlloc

```
allocSize= dword ptr -18h
var_14= dword ptr -14h
allocAddr= dword ptr -0Ch
var_8= dword ptr -8
flProtect= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 18h
mov     [ebp+flProtect], 40h ; set executable rights
mov     [ebp+allocAddr], 0
mov     eax, dword_5D70F8
mov     [ebp+allocSize], eax
mov     [ebp+var_8], 0FFFFFFFFh
mov     ecx, VirtualAlloc ; set ecx = VirtualAlloc ptr
mov     virtAllocCpy, ecx
push    [ebp+flProtect]
push    3000h
push    [ebp+allocSize]
push    [ebp+allocAddr]
push    virtAllocCpy
pop     ecx             ; get virtAllocCpy
call    ecx             ; call VirtualAlloc
mov     [ebp+var_14], eax
```

VirtualAlloc routine captured in IDA

The first step itself does not decrypt any code, however it writes bytes in `0x64` chunks into virtual memory `2304` times (`0x38400 / 0x64`). The position of these chunks are calculated loop after loop and do not lie linear in the memory.



2 – Loading dependencies

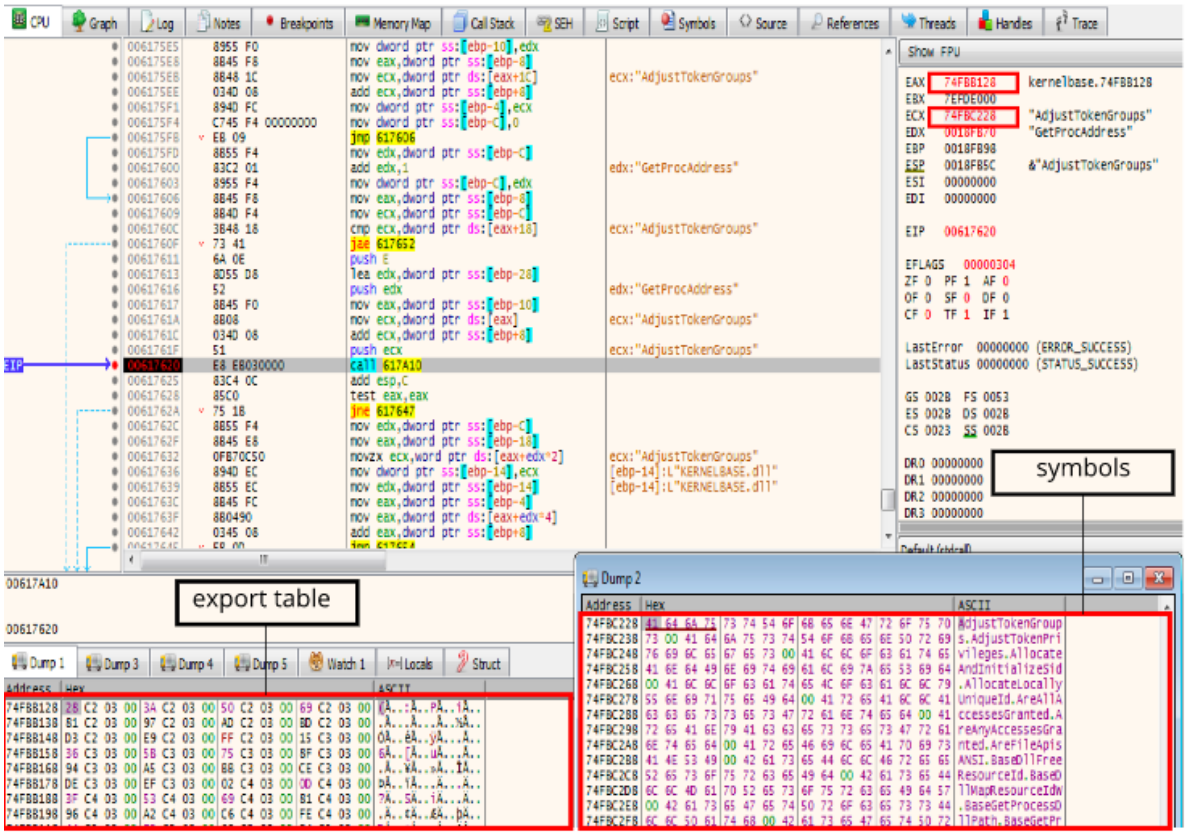
Once the virtual memory is allocated we can dump the code and load it into IDA to analyse it.
After returning the base address of the `KernelBase.dll`, the offset to the `GetProcAddress` function is determined by iterating over the export table.

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi
0000010D	00030D34	010C	0003D62E	GetNumberFormatEx
0000010E	00030CDC	010D	0003D640	GetNumberFormatW
0000010F	0002DA56	010E	0003D651	GetOEMCP
00000110	000075E2	010F	0003D65A	GetOverlappedResult
00000111	0000EA14	0110	0003D66E	GetPriorityClass
00000112	0001C9AA	0111	0003D67F	GetPrivateObjectSecurity
00000113	00011180	0112	0003D698	GetProcAddress
00000114	0001469A	0113	0003D6A7	GetProcessHeap
00000115	000146AC	0114	0003D6B6	GetProcessHeaps
00000116	0000E67D	0115	0003D6C6	GetProcessId
00000117	00012B5C	0116	0003D6D3	GetProcessIdOfThread
00000118	00031811	0117	0003D6E8	GetProcessPreferredUILanguages
00000119	0000EA7A	0118	0003D707	GetProcessTimes
0000011A	0000EEA2	0119	0003D717	GetProcessVersion
0000011B	0002296D	011A	0003D729	GetPtrCalData
0000011C	000229A6	011B	0003D737	GetPtrCalDataArray
0000011D	00007693	011C	0003D74A	GetQueuedCompletionStatus
0000011E	00007723	011D	0003D764	GetQueuedCompletionStatusEx
0000011F	0001C640	011E	0003D780	GetSecurityDescriptorControl
00000120	0001C6CD	011F	0003D79D	GetSecurityDescriptorDacl

Some exported functions of `KernelBase.dll`

Explaining this behaviour in pseudo code makes it clearer:

```
func = "GetProcAddress";
symbols = getSymbols()
for symbol in symbol:
       if symbol == func:
             return getOffsetToFunc(symbol)
```



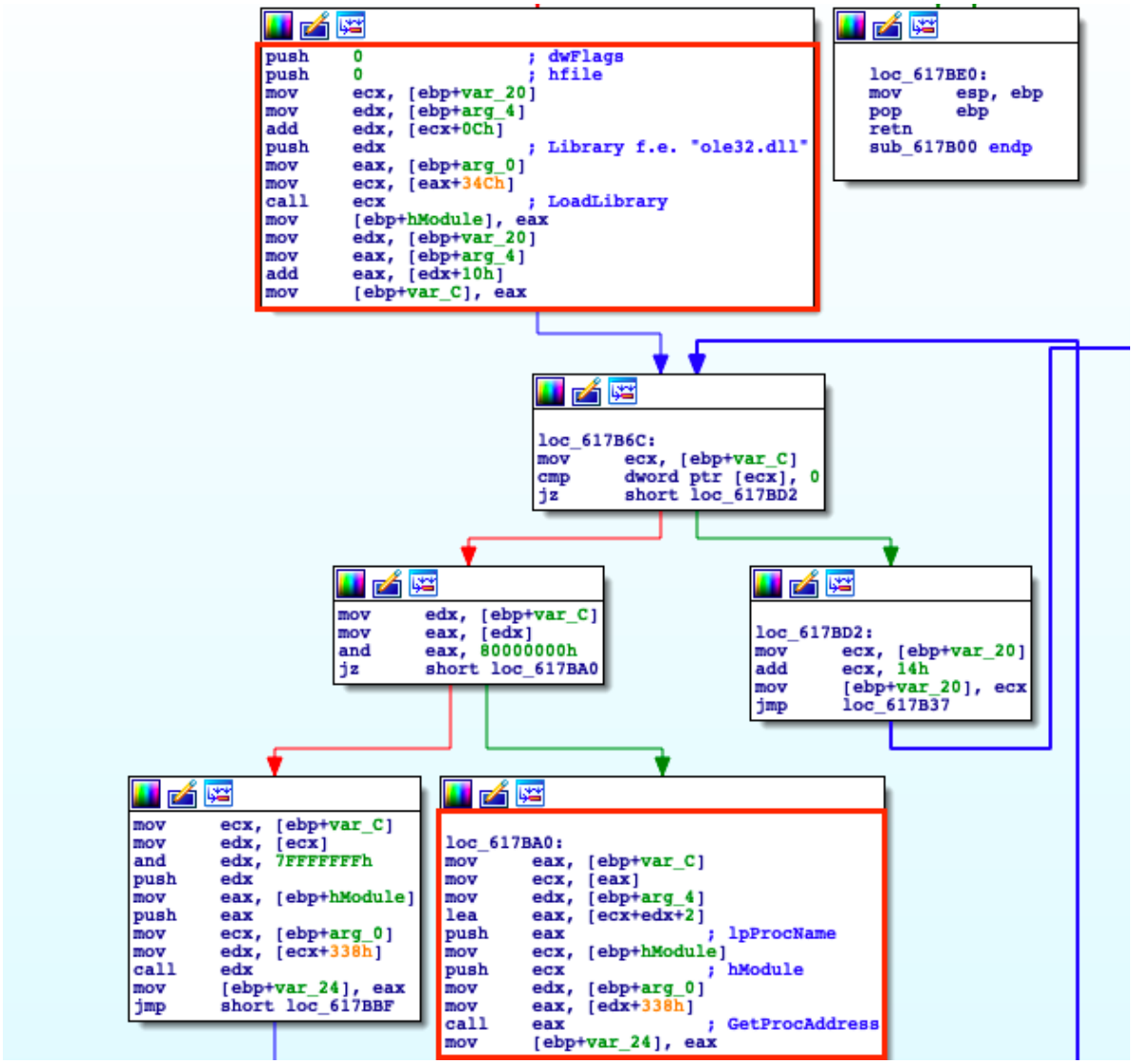
Searching for GetProcAddress in the debugger

With `GetProcAddress` the location of `LoadLibrary` is returned. By using these two functions the packer is now able to write offsets of needed library functions into memory.

3 – Decrypt the code

In the third step the actual payload is being prepared. `VirtualAlloc`[4] sets up another memory area which is used to hold decrypted code temporarily. After the decryption is finished a fully unpacked PE file lies now in memory. The PE sections we started with are zero'ed and replaced with the new decrypted sections.

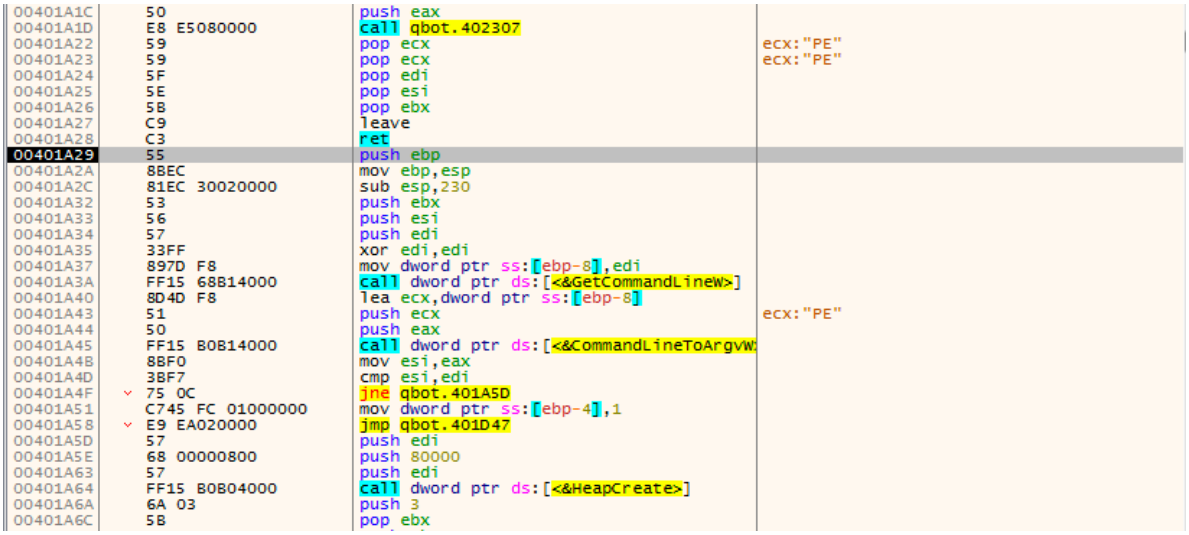
Some exported functions are still missing. In order to determine their position the same trick is used which I already explained in the second step. This time though, different libraries are used.



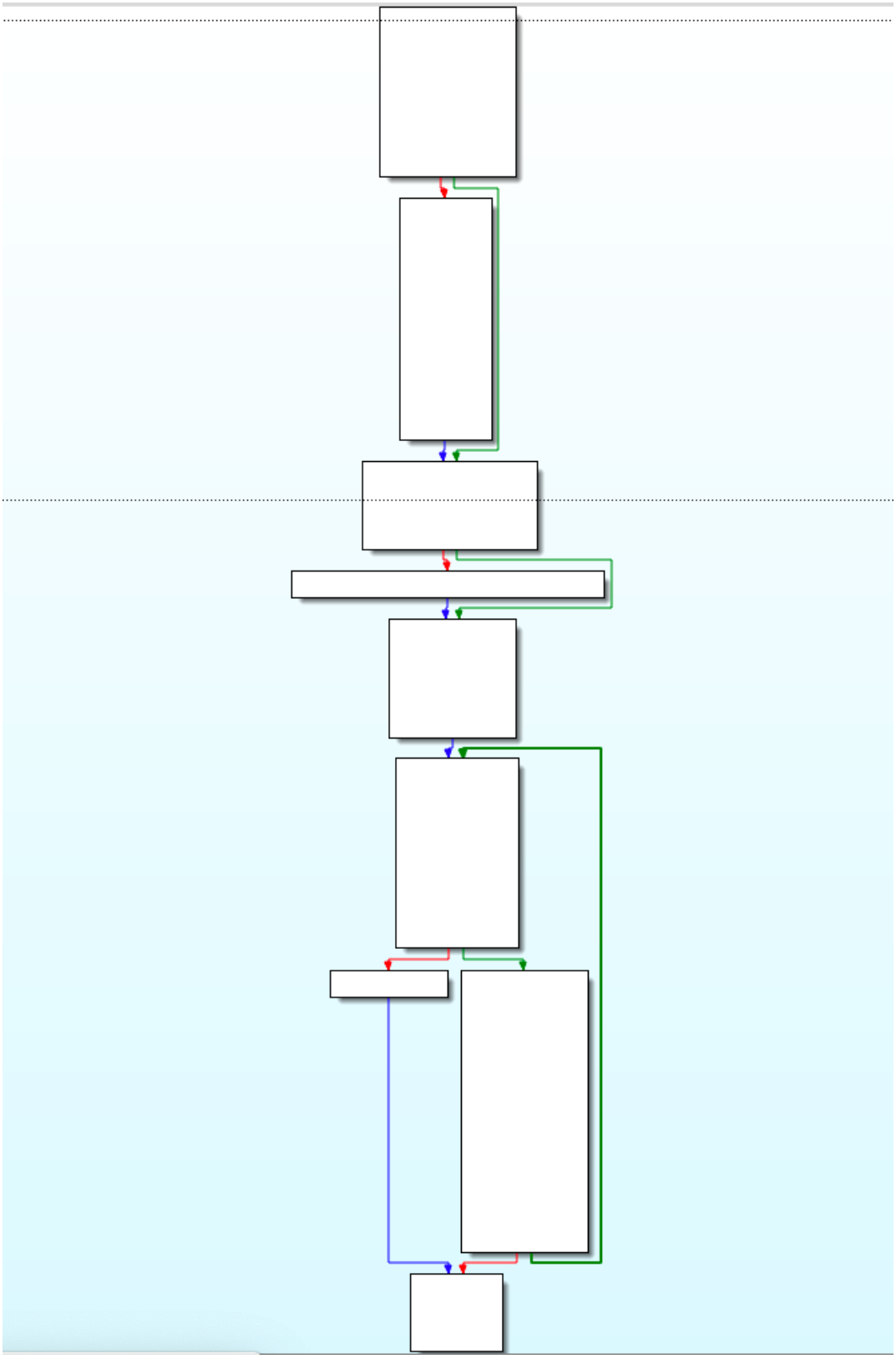
Determining position of final dependencies

4 – Returning to the payload

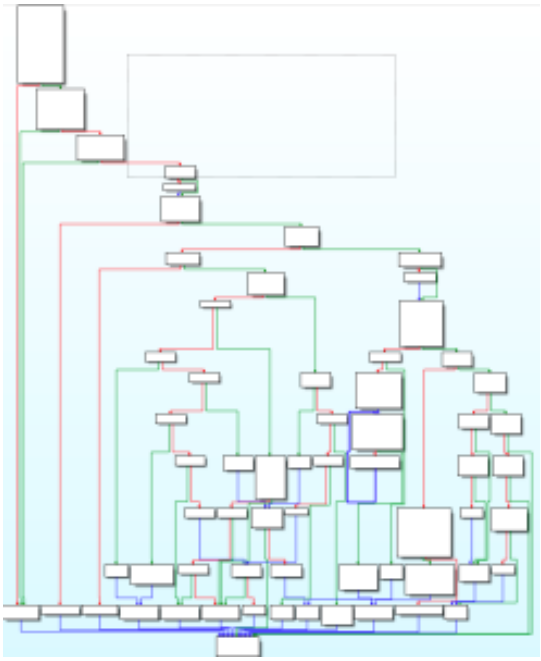
All that is left now is to return to the unpacked sample via return instruction because the return address is still written onto the stack.



Return back to where we started at



Graph overview of start func packed



Graph overview of start func unpacked

5 – IoCs

Sample SHA256	c23c9580f06fdc862df3d80fb8dc398b666e01a523f06ffa8935a95dce4ff8f4

← Previous Post

Next Post →

Follow Me

Recent Posts

The DLL Search Order And Hijacking It

PEB: Where Magic Is Stored

Catching Debuggers with Section Hashing

Taming Virtual Machine Based Code Protection – 2

DGAs – Generating domains dynamically