Qakbot Series: API Hashing

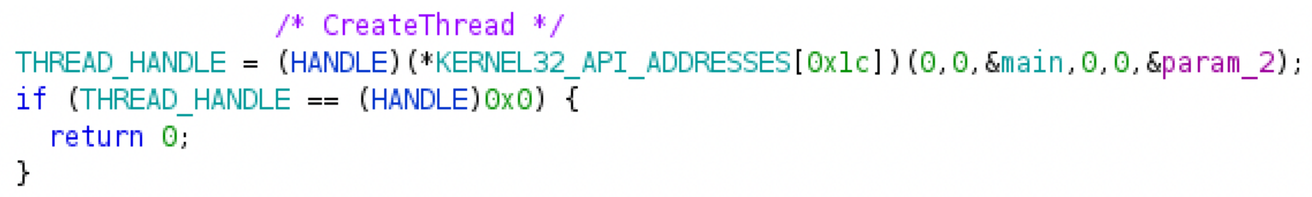
malwarology.com/2022/04/qakbot-series-api-hashing/

April 17, 2022

2022-04-17

Malware Analysis , Qakbot

In late March 2022, I was requested to analyze a software artifact. It was an instance of Qakbot, a modular information stealer known since 2007. Differently to other analyses I do as part of my daily job, in this particular case I can disclose wide parts of it with you readers. I'm addressing them in a post series. Here, I'll discuss about the Qakbot API hashing techinque based on this specific sample.



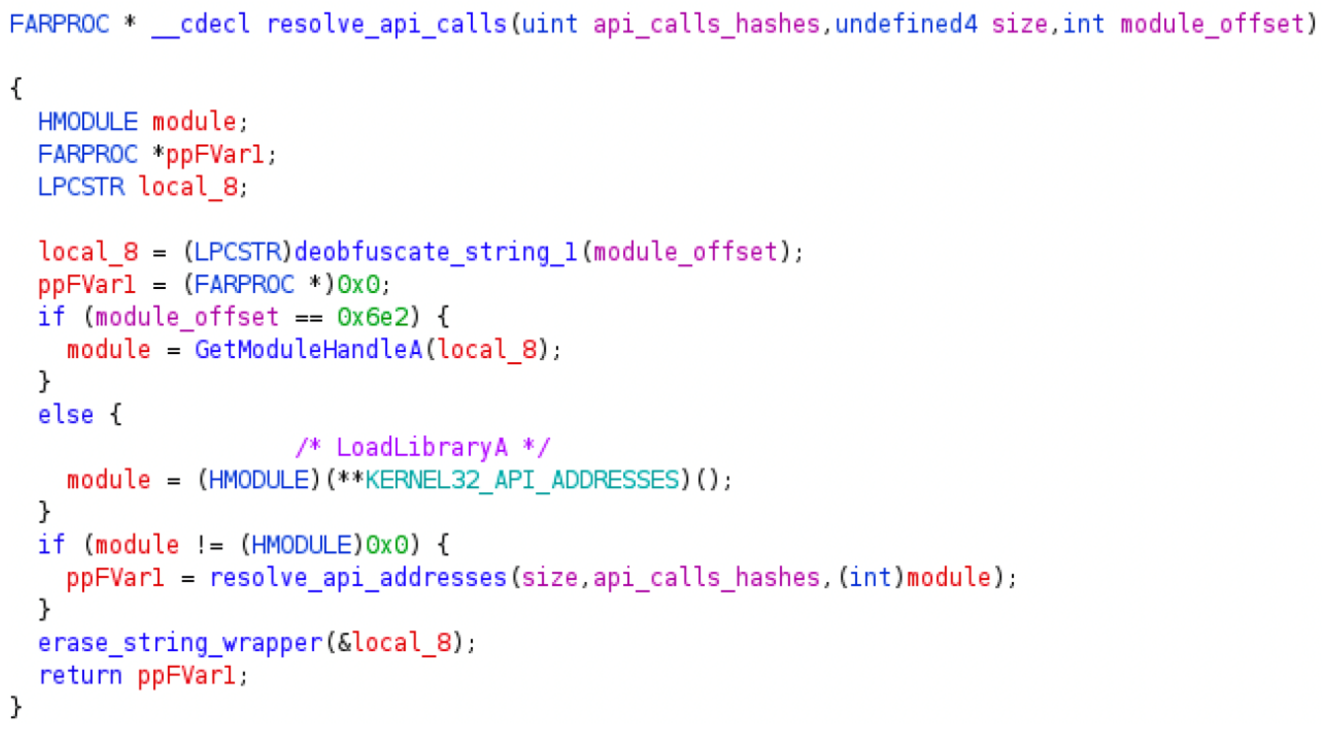
```
                /* CreateThread */
THREAD_HANDLE = (HANDLE)(*KERNEL32_API_ADDRESSES[0x1c])(0,0,&main,0,0,&param_2);
if (THREAD_HANDLE == (HANDLE)0x0) {
  return 0;
}
```

Figure 1

-

Example of protected API call in Qakbot

API hashing is an anti-analysis technique aimed at hiding a potential source of information about the malware capabilities. Analysts are used to rely on the API calls to form an hypothesis of what a given piece of code tries to do because much of the malware behavior can be inferred by looking at how it interacts with the operating system. As an example, **Figure 1** shows a call to CreateThread (kernel32.dll) located at *0xb2348b*. You cannot see any reference to CreateRemote thread into the code, besides the comment I placed to slightly improve the code readability. To understand what is happening here, I need to describe the API hashing technique implemented in the sample object of analysis. Luckly, this section is all around this topic.



```
FARPROC * __cdecl resolve_api_calls(uint api_calls_hashes,undefined4 size,int module_offset)

{
  HMODULE module;
  FARPROC *ppFVar1;
  LPCSTR local_8;

  local_8 = (LPCSTR)deobfuscate_string_1(module_offset);
  ppFVar1 = (FARPROC *)0x0;
  if (module_offset == 0x6e2) {
    module = GetModuleHandleA(local_8);
  }
  else {
                    /* LoadLibraryA */
    module = (HMODULE)(**KERNEL32_API_ADDRESSES)();
  }
  if (module != (HMODULE)0x0) {
    ppFVar1 = resolve_api_addresses(size,api_calls_hashes,(int)module);
  }
  erase_string_wrapper(&local_8);
  return ppFVar1;
}
```

Figure 2

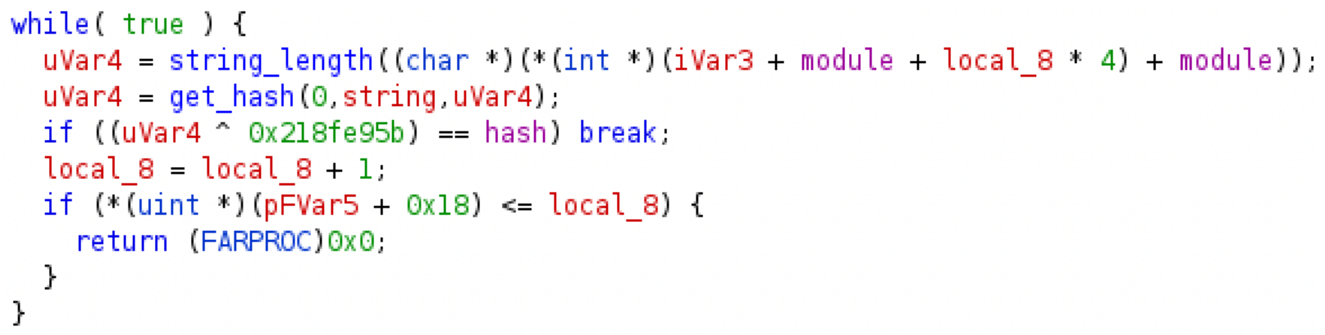
-
Qakbot API de-hashing main function

Qakbot attempts to invoke functions exported by 11 dynamic-link libraries: kernel32.dll, ntdll.dll, advapi32.dll, netapi32.dll, shell32.dll, shlwapi.dll, user32.dll, wininet.dll, urlmon.dll, crypt32.dll, and wtsapi32.dll. I know that because I found tracks of those libraries into the strings. Don't panic if you won't find those strings into the sample because they are obfuscated. If you'are interested in knowing how Qakbot obfuscates the strings, I address the topic in this post. From those strings, I was able to discover the function responsible for the API de-hashing. For API de-hashing I mean the process allowing to retrieve API addresses starting from rather anonymous numbers called hashes. That function is located at *0xb3050a* and it is showed in **Figure 2**.

That function expects three arguments. The very first argument is an hash table, namely a vector of 32 bits hashes. A very important point to make clear here is that there is a dedicated and hardcoded hash table for every DLL mentioned above. The second argument is the size of the hash table. The last argument is the offset used by the function *deobfuscate_string_1* to de-obfuscate the name of the DLL. The goal of the de-hashing function consists in allocating a new table containing the memory address of all the required APIs for the given DLL. Once such an address table is allocated, invocations for an API may occur by just pointing at the corresponding entry in the table. **Figure 1** shows an example of this type of invocation where the offset *0x1c2* is aligned with the memory address of

CreateThread into the kernel32.dll table (actually, the offset for CreateThread is *0x70*, as it is possible to observe into the assembly listing at *0xb2348b*. For some reason, Ghidra decompiler reconstructed a different offset).

As a side note, *0x6e2* is the offset for kernel32.dll therefore the first condition in the listing of **Figure 2** checks if the caller has requested an API exported by kernel32.dll. Indeed, if that is the case then it is sufficient to invoke GetModuleHandleA to obtain an handle to the DLL because kernel32.dll is mapped in the memory space of every process at loading time. Otherwise, if the requested module isn't kernel32.dll, LoadLibraryA is invoked to obtain the same result. Notice that in the latter case the API invocation occurs via the kernel32 address table where LoadLibraryA lies at offset 0.



```
while( true ) {
  uVar4 = string_length((char *)(*(int *)(iVar3 + module + local_8 * 4) + module));
  uVar4 = get_hash(0,string,uVar4);
  if ((uVar4 ^ 0x218fe95b) == hash) break;
  local_8 = local_8 + 1;
  if (*(uint *)(pFVar5 + 0x18) <= local_8) {
    return (FARPROC)0x0;
  }
}
```

Figure 3

-
Hash checking to identify the requested export name

The actual de-hashing occurs in a sub-function of *resolve_api_address* located at *0xb303ca*. This function is responsible for resolving the address of a specific API of a given module starting from an hash. The code snippet showed in **Figure 3** highlights the hash check. That loop enumerates all the export names of a module and ends in two cases:

- When the hash provided to resolve_api_address is equal to the hash of an export name xor-ed with an hardcoded constant (*0x218fe95b*). In this case there is a hit and the export name is later provided to GetProcAddress to obtain the corresponding API address.
- The provided hash doesn't correspond to any export name of the module. In this case the function returns NULL.

I close this post by sharing the mapping between hashes and all the API functions requested by this Qakbot sample. You will find the mapping here. Each row in this file contains the library exporting the API function, the offset into the address table, the hash, and finally the API function name. The offset may turn useful for analysis purposes because it is often hardcoded as showed in **Figure 1**.

As always, if you want to share comments or feedbacks (rigorously in broken Italian or broken English) do not esitate to drop me a message at **admin[@]malwarology.com**.