An Exhaustively Analyzed IDB for ComLook

msreverseengineering.com/blog/2022/1/25/an-exhaustively-analyzed-idb-for-comlook

January 25, 2022

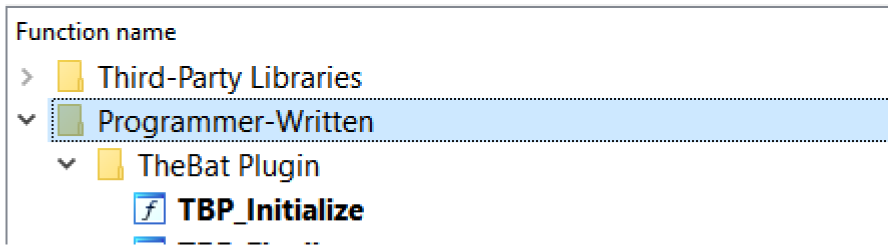
January 25, 2022 Rolf Rolles

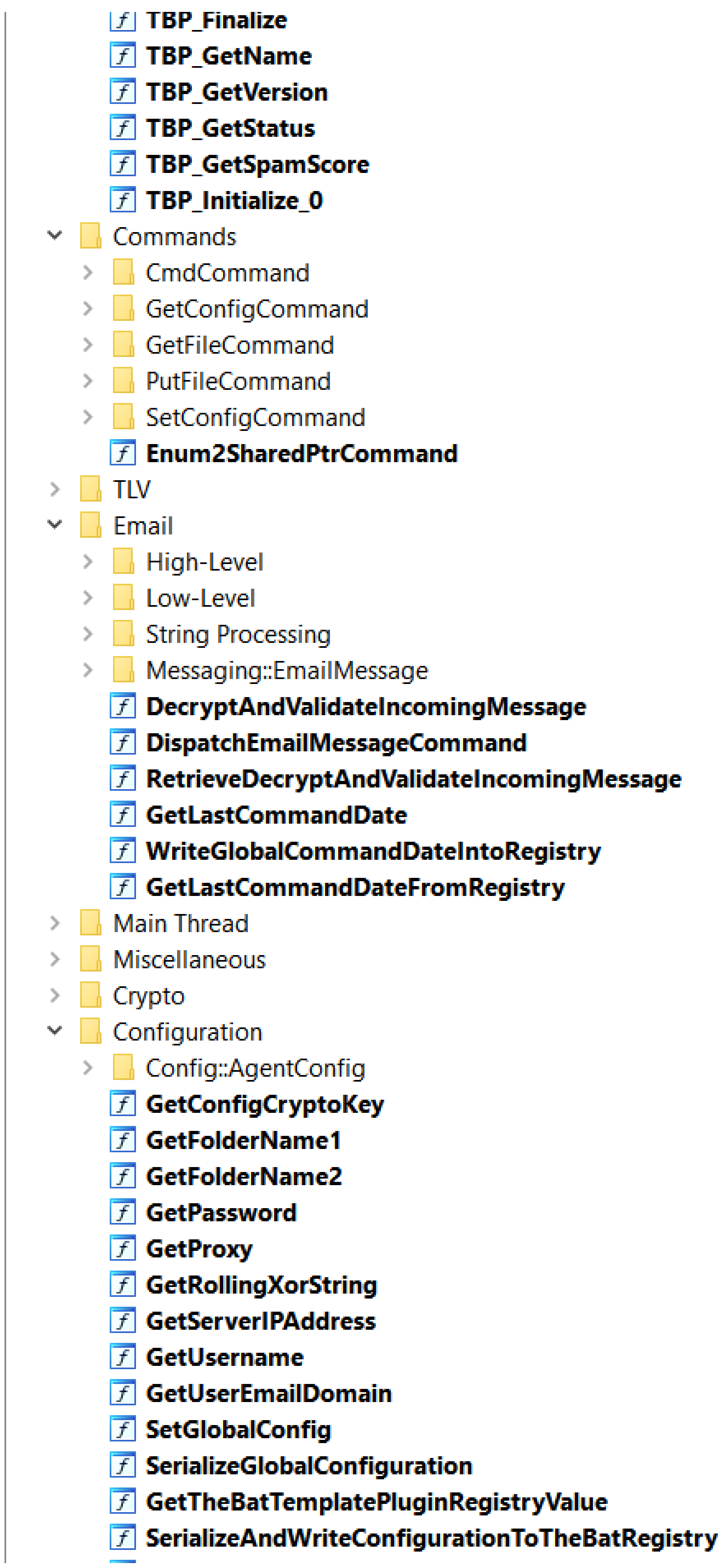
This blog entry announces the release of an exhaustive analysis of ComLook, a newly-discovered malware family about which little information has been published. It was recently discovered by ClearSky Cyber Security, and announced in a thread on Twitter. You can find the IDB for the DLL here, in which every function has been analyzed, and every data structure has been recovered.

Like the previous two entries in this series on ComRAT v4 and FlawedGrace, I did this analysis as part of my preparation for an upcoming class on C++ reverse engineering. The analysis took about a one and a half days (done on Friday and Saturday). ComLook is an Outlook plugin that masquerades as Antispam Marisuite v1.7.4 for The Bat!. It is fairly standard as far as remote-access trojans go; it spawns a thread to retrieve messages from a C&C server over IMAP, and processes incoming messages in a loop. Its command vocabulary is limited; it can only read and write files to the victim server, run commands and retrieve the output, and update/retrieve the current configuration (which is saved persistently in the registry). See the IDB for complete details.

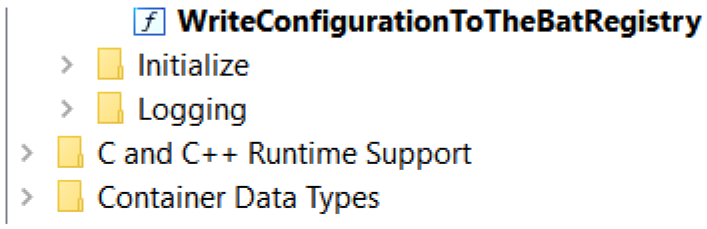
(Note that if you are interested in the forthcoming C++ training class, it is nearing completion, and should be available in Q2 2022. More generally, remote public classes (where individual students can sign up) are temporarily suspended; remote private classes (multiple students on behalf of the same organization) are currently available. If you would like to be notified when public classes become available, or when the C++ course is ready, please sign up on our no-spam, very low-volume, course notification mailing list. (Click the button that says "Provide your email to be notified of public course availability".))

This analysis was performed with IDA Pro 7.7 and Hex-Rays 32-bit. All analysis has been done in Hex-Rays; go there for all the gory details, and don't expect much from the disassembly listing. All of the programmer-created data structures have been recovered and applied to the proper Hex-Rays variables. The functionality has been organized into folders, as in the following screenshot:





```
        TBP_Finalize
        TBP_GetName
        TBP_GetVersion
        TBP_GetStatus
        TBP_GetSpamScore
        TBP_Initialize_0
  ⌄ Commands
    › CmdCommand
    › GetConfigCommand
    › GetFileCommand
    › PutFileCommand
    › SetConfigCommand
        Enum2SharedPtrCommand
  › TLV
  ⌄ Email
    › High-Level
    › Low-Level
    › String Processing
    › Messaging::EmailMessage
        DecryptAndValidateIncomingMessage
        DispatchEmailMessageCommand
        RetrieveDecryptAndValidateIncomingMessage
        GetLastCommandDate
        WriteGlobalCommandDateIntoRegistry
        GetLastCommandDateFromRegistry
  › Main Thread
  › Miscellaneous
  › Crypto
  ⌄ Configuration
    › Config::AgentConfig
        GetConfigCryptoKey
        GetFolderName1
        GetFolderName2
        GetPassword
        GetProxy
        GetRollingXorString
        GetServerIPAddress
        GetUsername
        GetUserEmailDomain
        SetGlobalConfig
        SerializeGlobalConfiguration
        GetTheBatTemplatePluginRegistryValue
        SerializeAndWriteConfigurationToTheBatRegistry
```

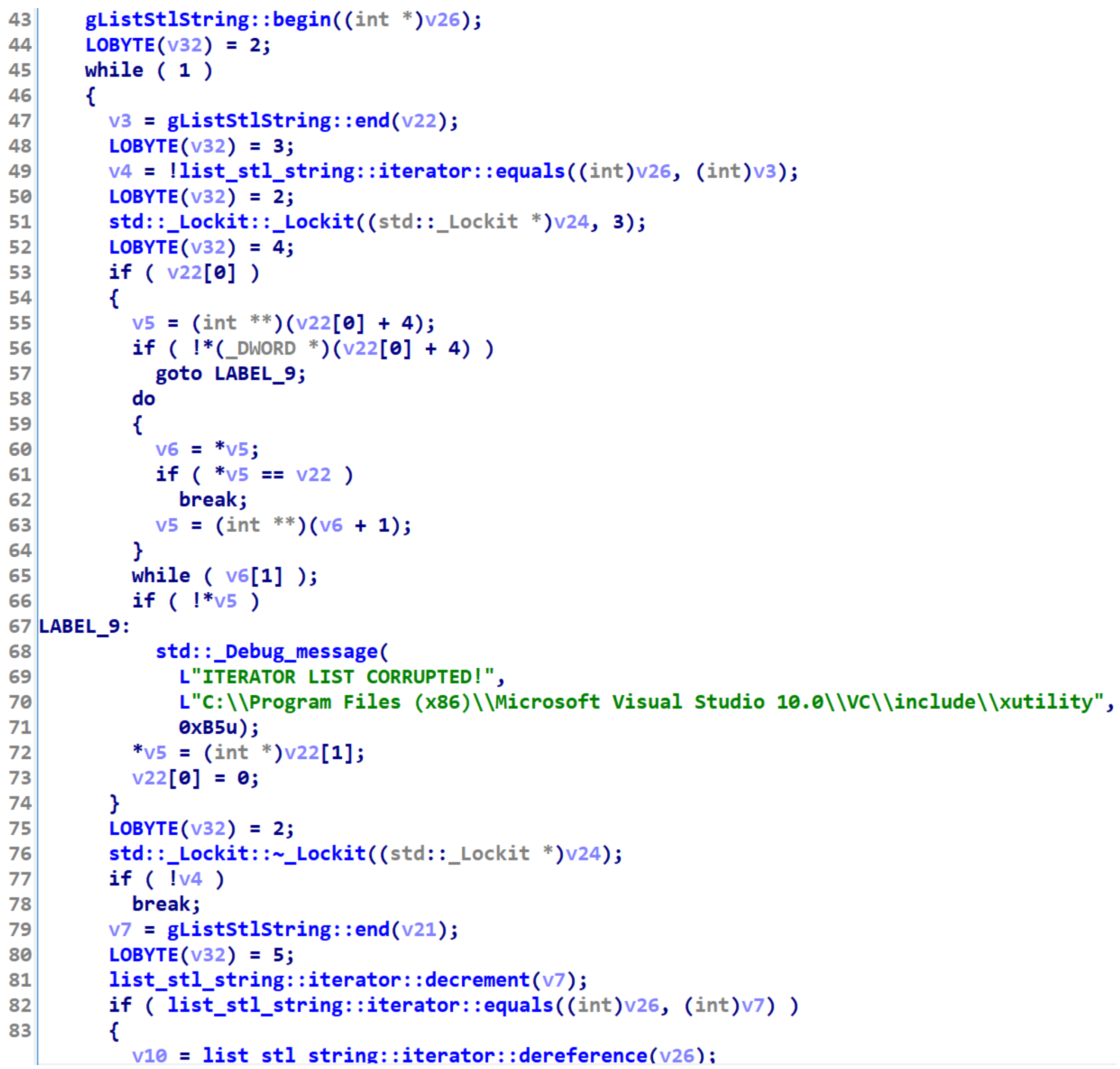


WriteConfigurationToTheBatRegistry
Initialize
Logging
C and C++ Runtime Support
Container Data Types

The binary was compiled with MSVC 10.0 with RTTI, and uses standard C++ template containers:

- string/wstring
- shared_ptr
- vector
- list
- map

The primary difficulty in analyzing this sample was that it was compiled in debug mode. Although this does simplify some parts of the analysis (e.g., error message contain the raw STL typenames), it also slows the speed of comprehension due to a lack of inlining, and includes a huge amount of code to validate so-called C++ debug iterators. This makes locating the real programmer-written functionality more time-consuming. STL functions involving iterators that, in a release build, would have consumed less than a page of decompilation, often consumed five or more pages due to debug iterator validation.



```
    gListStlString::begin((int *)v26);
    LOBYTE(v32) = 2;
    while ( 1 )
    {
      v3 = gListStlString::end(v22);
      LOBYTE(v32) = 3;
      v4 = !list_stl_string::iterator::equals((int)v26, (int)v3);
      LOBYTE(v32) = 2;
      std::_Lockit::_Lockit((std::_Lockit *)v24, 3);
      LOBYTE(v32) = 4;
      if ( v22[0] )
      {
        v5 = (int **)(v22[0] + 4);
        if ( !*(_DWORD *)(v22[0] + 4) )
          goto LABEL_9;
        do
        {
          v6 = *v5;
          if ( *v5 == v22 )
            break;
          v5 = (int **)(v6 + 1);
        }
        while ( v6[1] );
        if ( !*v5 )
LABEL_9:
          std::_Debug_message(
            L"ITERATOR LIST CORRUPTED!",
            L"C:\\Program Files (x86)\\Microsoft Visual Studio 10.0\\VC\\include\\xutility",
            0xB5u);
        *v5 = (int *)v22[1];
        v22[0] = 0;
      }
      LOBYTE(v32) = 2;
      std::_Lockit::~_Lockit((std::_Lockit *)v24);
      if ( !v4 )
        break;
      v7 = gListStlString::end(v21);
      LOBYTE(v32) = 5;
      list_stl_string::iterator::decrement(v7);
      if ( list_stl_string::iterator::equals((int)v26, (int)v7) )
      {
        v10 = list_stl_string::iterator::dereference(v26);
```