An Exhaustively-Analyzed IDB for ComRAT v4

msreverseengineering.com/blog/2020/8/31/an-exhaustively-analyzed-idb-for-comrat-v4

September 1, 2020

September 1, 2020 Rolf Rolles

This blog entry announces the release of an exhaustive analysis of ComRAT v4. You can find the IDBs here.

More specifically, an IDB for the sample with hash 0139818441431C72A1935E7F740A1CC458A63452, which was mentioned in the ESET report (see especially its attached PDF), and which is available online on Hybrid Analysis. All of the analysis has been performed in Hex-Rays 64-bit, so the results will be less interesting to IDA users who do not own Hex-Rays 64-bit. That is to say, if you open the IDB, you should definitely use Hex-Rays to view the function decompilations, as that is where all of the naming and commenting has taken place. It is rich with detail, in comparison to the disassembly listing's barrenness.

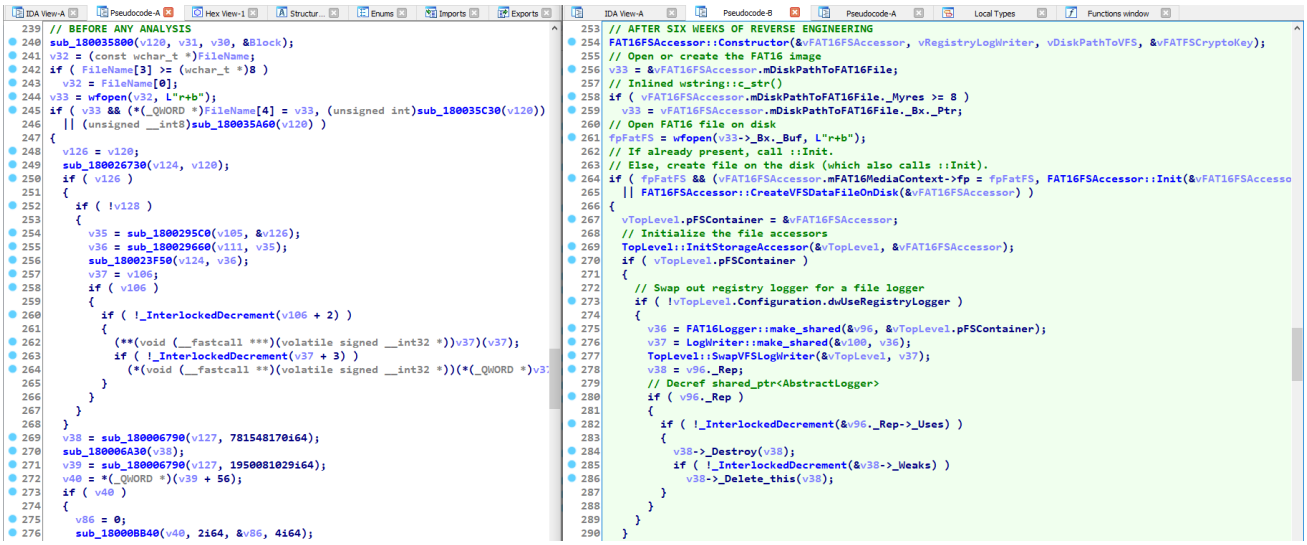
This analysis took roughly six weeks of full-time work. I have spent the pandemic working on a new training class on C++ reverse engineering; part of the preparation includes large-scale analysis of C++ programs. As such, ESET's report of ComRAT's use of C++ caught my eye. ComRAT has a beautiful architecture, and many sophisticated components, all of which I believe deserve a detailed report unto themselves. I had begun writing such a report, but decided that it was side-tracking me from my ultimate goals with my new training class. Hence, I had decided to wait until the class was ready, and release a collection of reports on the software architectures of C++ malware families (perhaps as a book) after I was done. Thus, my write-up on ComRAT's architecture will have to wait. You can consider this release, instead, as a supplement to the ESET report.

(Note that if you are interested in the forthcoming C++ training class, it probably will not be available for roughly another year. 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".))

(Note also that I have more analyses like this waiting to be released. FlawedGrace and XAgent are ready; Kelihos is in progress. If you can provide me with a bundle of Careto SGH samples, preferably Windows 64-bit, please get in touch.)

About the Analysis

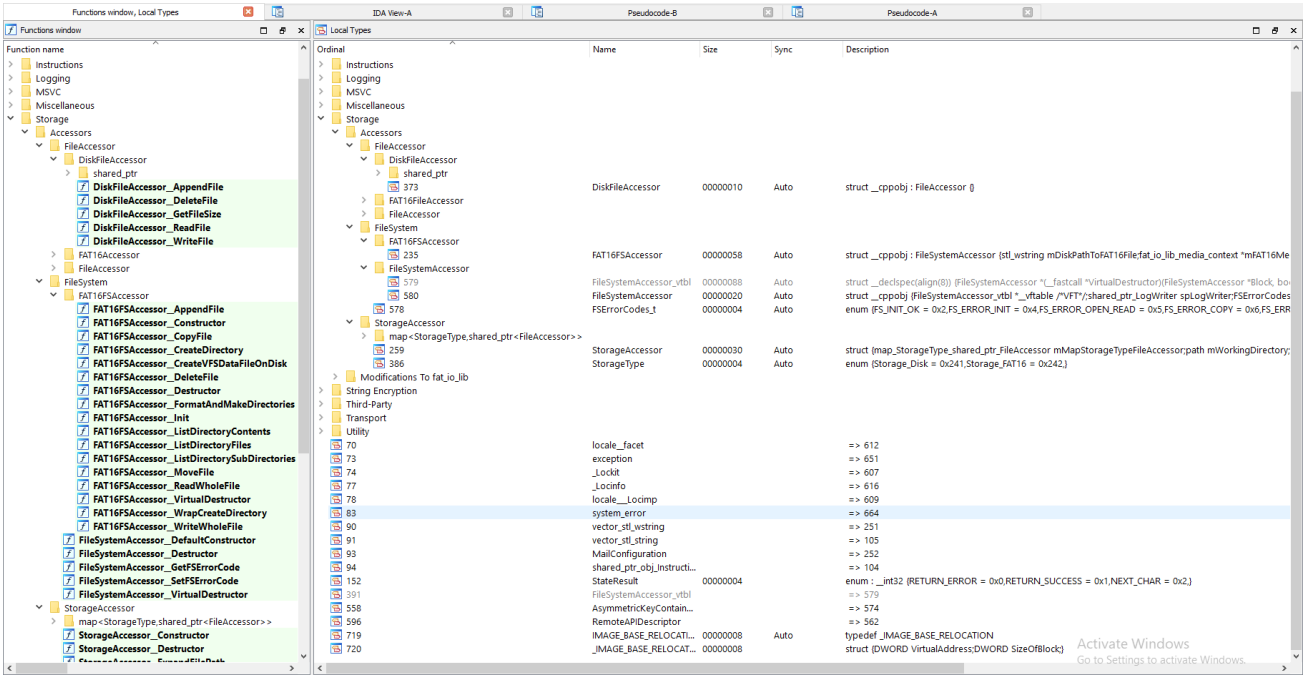
This analysis was conducted purely statically, without access to RTTI, or any other form of debug information. The only external information I had was the ESET report. I have reverse engineered every function in the binary that is not part of the C++ standard library, and some of those that are. To get an idea of what the sample looks like before and after analysis, here's a screenshot of the binary freshly loaded into IDA on the left, versus the analyzed one on the right. See if you can spot the difference:



```
// BEFORE ANY ANALYSIS
sub_180035800(v120, v31, v30, &Block);
v32 = (const wchar_t *)FileName;
if ( FileName[3] >= (wchar_t *)8 )
  v32 = FileName[0];
v33 = wfopen(v32, L"r+b");
if ( v33 && (*(_QWORD *)FileName[4] = v33, (unsigned int)sub_180035C30(v120))
  || (unsigned __int8)sub_180035A60(v120) )
{
  v126 = v120;
  sub_180026730(v124, v120);
  if ( v126 )
  {
    if ( !v128 )
    {
      v35 = sub_1800295C0(v105, &v126);
      v36 = sub_180029660(v111, v35);
      sub_180023F50(v124, v36);
      v37 = v106;
      if ( v106 )
      {
        if ( !_InterlockedDecrement(v106 + 2) )
        {
          (**(void (__fastcall ***)(volatile signed __int32 *))v37)(v37);
          if ( !_InterlockedDecrement(v37 + 3) )
            (*(void (__fastcall **)(volatile signed __int32 *))(*(_QWORD *)v37
        }
      }
    }
  }
  v38 = sub_180006790(v127, 781548170i64);
  sub_180006A30(v38);
  v39 = sub_180006790(v127, 1950081029i64);
  v40 = *(_QWORD *)(v39 + 56);
  if ( v40 )
  {
    v86 = 0;
    sub_18000BB40(v40, 2i64, &v86, 4i64);
```

```
// AFTER SIX WEEKS OF REVERSE ENGINEERING
FAT16FSAccessor::Constructor(&vFAT16FSAccessor, vRegistryLogWriter, vDiskPathToVFS, &vFATFSCryptoKey);
// Open or create the FAT16 image
v33 = &vFAT16FSAccessor.mDiskPathToFAT16File;
// Inlined wstring::c_str()
if ( vFAT16FSAccessor.mDiskPathToFAT16File._Myres >= 8 )
  v33 = vFAT16FSAccessor.mDiskPathToFAT16File._Bx._Ptr;
// Open FAT16 file on disk
fpFatFS = wfopen(v33->_Bx._Buf, L"r+b");
// If already present, call ::Init.
// Else, create file on the disk (which also calls ::Init).
if ( fpFatFS && (vFAT16FSAccessor.mFAT16MediaContext->fp = fpFatFS, FAT16FSAccessor::Init(&vFAT16FSAccesso
  || FAT16FSAccessor::CreateVFSDataFileOnDisk(&vFAT16FSAccessor) )
{
  vTopLevel.pFSContainer = &vFAT16FSAccessor;
  // Initialize the file accessors
  TopLevel::InitStorageAccessor(&vTopLevel, &vFAT16FSAccessor);
  if ( vTopLevel.pFSContainer )
  {
    // Swap out registry logger for a file logger
    if ( !vTopLevel.Configuration.dwUseRegistryLogger )
    {
      v36 = FAT16Logger::make_shared(&v96, &vTopLevel.pFSContainer);
      v37 = LogWriter::make_shared(&v100, v36);
      TopLevel::SwapVFSLogWriter(&vTopLevel, v37);
      v38 = v96._Rep;
      // Decref shared_ptr<AbstractLogger>
      if ( v96._Rep )
      {
        if ( !_InterlockedDecrement(&v96._Rep->_Uses) )
        {
          v38->_Destroy(v38);
          if ( !_InterlockedDecrement(&v38->_Weaks) )
            v38->_Delete_this(v38);
        }
      }
    }
  }
}
```

Although I believe that the IDB could probably be loaded in versions of IDA prior to 7.5, I nevertheless recommend using IDA 7.5 to view it. The reason for that is because I have made extensive use of 7.5's new "folders" feature to organize the functions and local types windows, which I found massively useful for large-scale reverse engineering. Those two windows have a nearly identical organization; if you were to dock the windows side-by-side, you would see something like this:



As a result of this analysis, I wrote many Hex-Rays plugins, and devised a number of techniques in C++ reverse engineering that were new to me. Eventually, I will publish on topics such as the following:

- A Hex-Rays plugin for navigating virtual function cross-references
- Reverse engineering STL containers, the easy way
- A Hex-Rays plugin for virtual inheritance
- Tips for reverse engineering multiple inheritance
- Automated creation of VTable structure types
- Automation for detecting inlined functions, and the addition of stock comments

ComRAT uses a lot of C++ features; a mostly complete list follows. If you're interested in learning how to reverse engineer C++ programs, you might do well to study how I analyzed the parts of the binary that interact with them.

- Inheritance
- Polymorphism (virtual functions)
- Custom templates
- Multiple and virtual inheritance (due to iostreams)

- STL, listed in descending order of usage frequency:
    - shared_ptr<T>
    - vector<T>
    - string
    - wstring
    - locale
    - unique_ptr<T>
    - wstringstream
    - stringstream
    - fstream
    - list<T>
    - map<K,V>
    - regex
    - wstring_convert
    - random

Notes on the Sample

1. Although the use of Gmail as a covert channel was a major aspect of the ESET report, I could not get my hands on any samples that had that feature. However, this sample does contain some of the Gmail communication code -- the Gumbo library is compiled into it, and the configuration in the virtual file system contains a "mail" subdirectory, with similar entries to those in the ESET report. Perhaps that feature was still in development, or was deliberately not compiled into my sample for whatever reason.

2. One striking feature of the ESET report was that their sample had RTTI information compiled into it, which provided the names of many of the classes used within ComRAT. I.e., section 4.3 of the ESET report mentions specific class names, as created by the ComRAT programmers. However, my sample had no such RTTI information. Therefore, all of my analysis had to be done from scratch. I used the few names provided in the report as a guide when creating my own.

3. To the extent I was able to verify their claims, everything in the ESET report is accurate. There are a few minor technical details in my sample that were different, but are barely worth mentioning, and might have legitimately changed between the creation of my sample and the non-public one they analyzed.