An Exhaustively-Analyzed IDB for FlawedGrace

msreverseengineering.com/blog/2021/3/2/an-exhaustively-analyzed-idb-for-flawedgrace

March 2, 2021

March 2, 2021 Rolf Rolles

This blog entry announces the release of an exhaustive analysis of FlawedGrace. You can find the IDB for the main executable, and for the 64-bit password stealer module, here. The sha1sum for the main executable is 9bb72ae1dc6c49806064992e0850dc8cb02571ed, and the md5sum is bc91e2c139369a1ae219a11cbd9a243b.

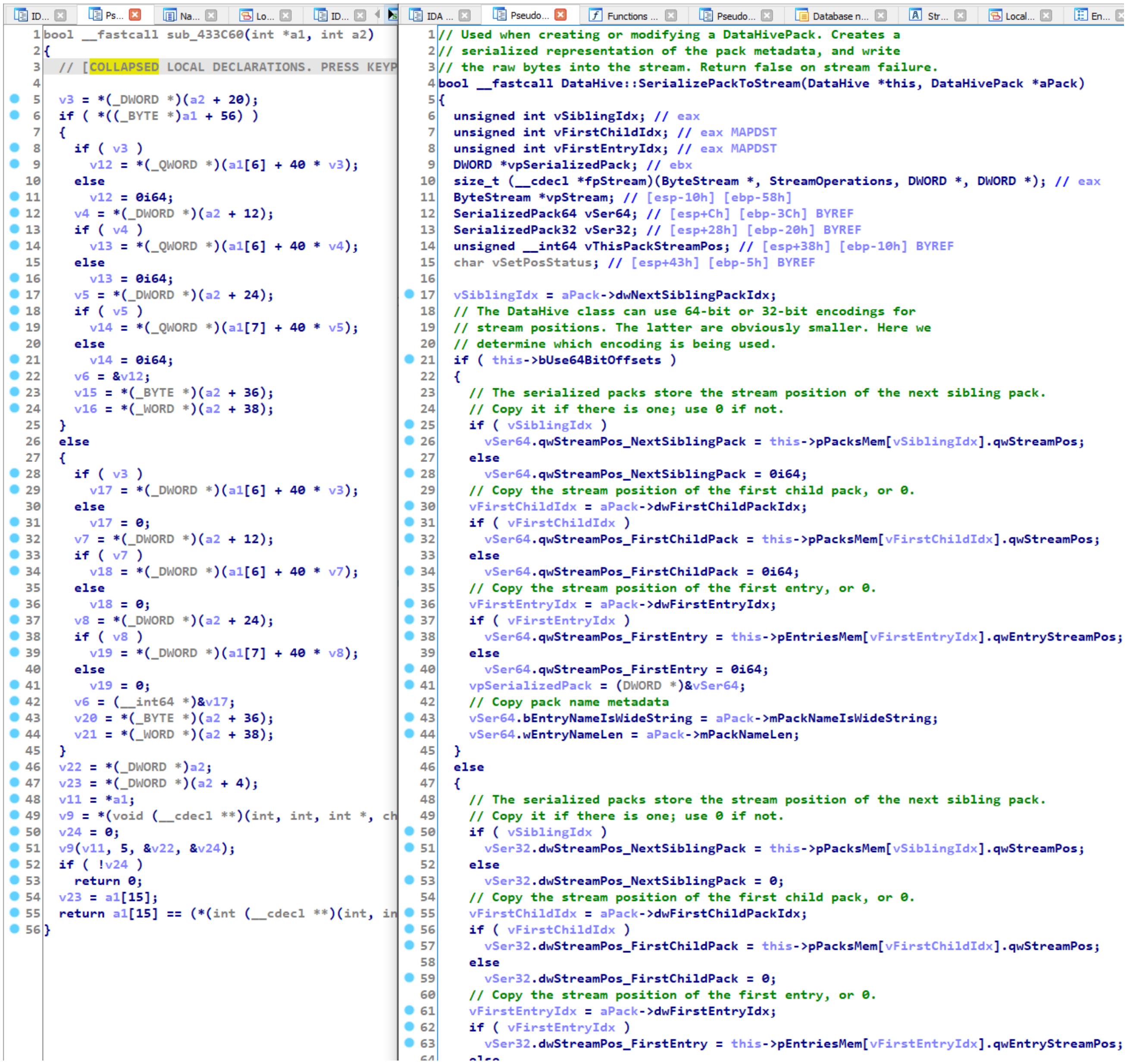
Like the previous entry in this series on ComRAT v4, I did this analysis as part of my preparation for an upcoming class on C++ reverse engineering. The analysis took about a month, and made me enamored with FlawedGrace's architecture. I have personally never analyzed (nor read the source for) a program with such a sophisticated networking component. Were I ever to need a high-performance, robust, and flexible networking infrastructure, I'd probably find myself cribbing from FlawedGrace. This family is also notable for its custom, complex virtual filesystem used for configuration management and C2 communications. I would like to eventually write a treatise about all of the C++ malware family analyses that I performing during my research for the class, but that endeavor was distracting me from work on my course, and hence will have to wait.

(Note that if you are interested in the forthcoming C++ training class, it probably will be available in Q3/Q4 2021. 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".))

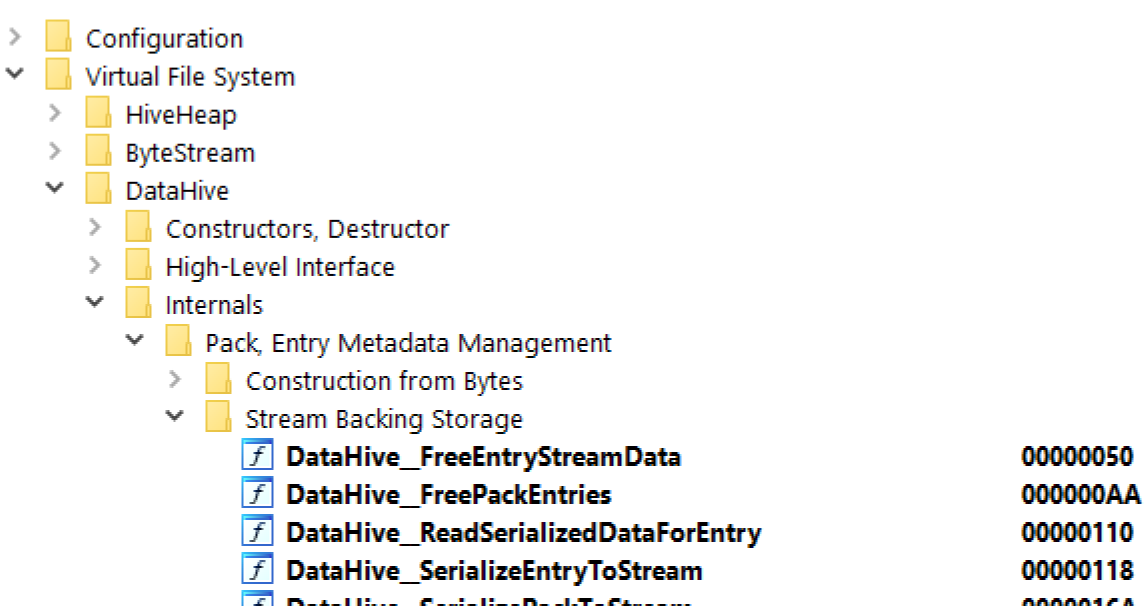
(Note that I am looking for a fifth and final family (beyond ComRAT, FlawedGrace, XAgent, and Kelihos) to round out my analysis of C++ malware families. If you have suggestions -- and samples, or hashes I can download through Hybrid-Analysis -- please send me an email at rolf@ my domain.)

About the IDB

Here are some screenshots. First, a comparison of the unanalyzed executable versus the analyzed one:

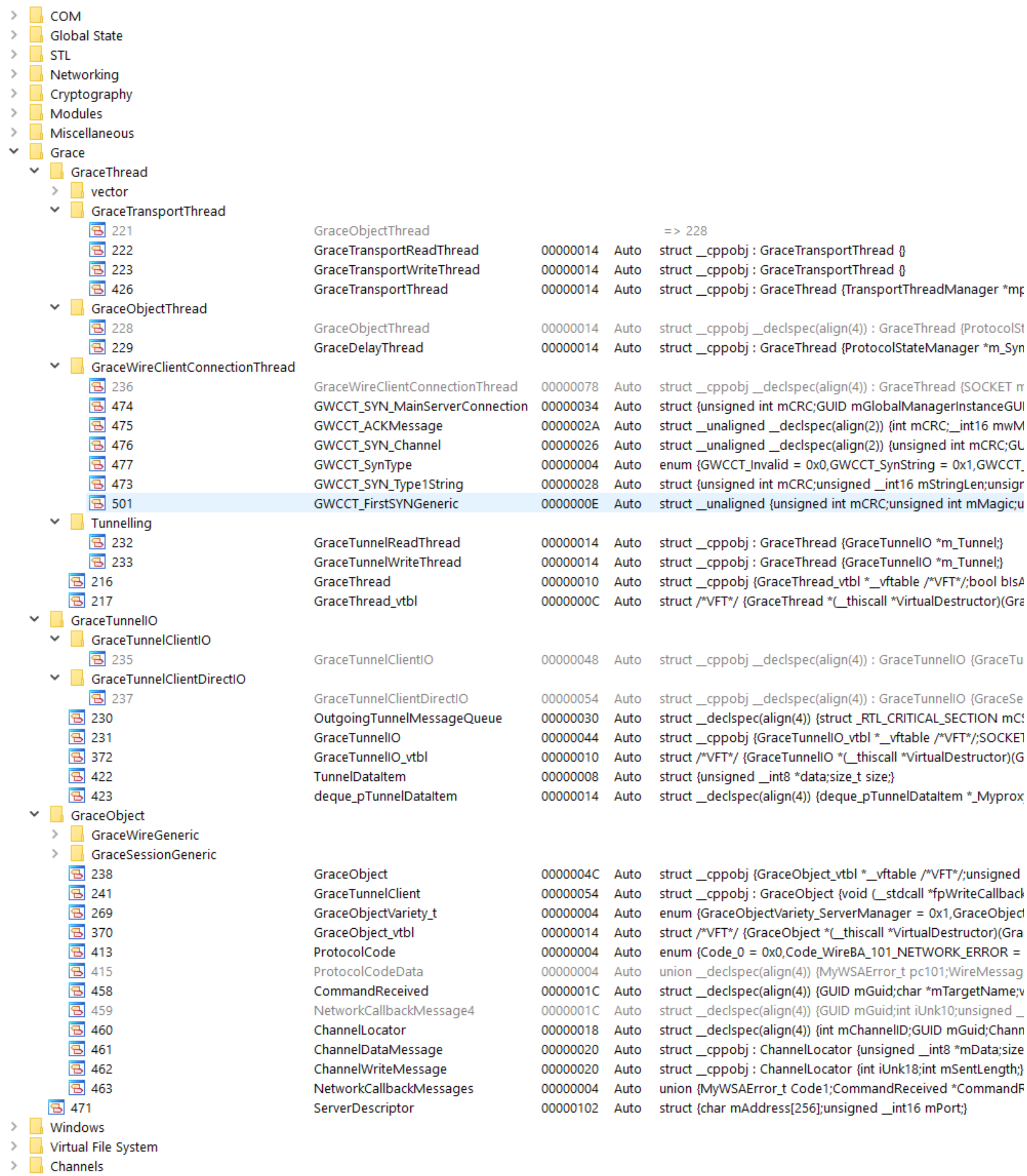


Next, IDA's function folders should make it easy to find the parts that interest you:



```
          DataHive_SerializePackToStream                          0000016A
          DataHive_WriteSerializedDataForEntry                    000001C0
        > Index Management
        > Removal
      > Stream Data Management
      > Retrieval
        IterativeCrackDataHivePathA                               0000009D
        IterativeCrackDataHivePathW                               000000A9
> Backdoor Commands
v Channels
    > RDP
    > Download
    > Upload
    > GenericChannelDescriptor
      EnqueueDataHiveAsChannelWriteEntry                          00000048
      ShutdownChannelByID                                         00000055
      EnqueueChannelWriteEntry                                    00000078
      ShutdownChannelsOfType                                      000000B1
      ChannelThreadProc                                           000000F2
      CreateChannel                                               00000117
      g_List_GenericChannelDescriptor__EnqueueChannelMessage      0000014D
v Grace
    > GraceThread
    > GraceTunnelIO
    > GraceObject
> Network
> Thread Procedures
v Global State
    v ProtocolStateManager
        > GraceObjectManager
        > ProtocolEventManager
          ProtocolStateManager__RemoveDelayedEventsByID           0000007C
          ProtocolStateManager__AcquireGraceObjectBySerial        00000086
          ProtocolStateManager__AddDelayedEvent                   0000008F
          ProtocolStateManager__DequeueAndExecuteProtocolEvent    000000A7
          ProtocolStateManager__DequeueProtocolEvent              000000A8
          ProtocolStateManager__RegisterNewEvent                  000000C1
          ProtocolStateManager__RemoveEventByID                   000000C6
          ProtocolStateManager__RemoveDelayedEventsByObject       000000DE
          ProtocolStateManager__EnqueueProtocolEventByID          000000E1
          ProtocolStateManager__RemoveGraceObjectFromTrackedSet   000000E8
          ProtocolStateManager__CleanupAndRemoveGraceObject       0000015E
          ProtocolStateManager__Cleanup                           00000168
          ProtocolStateManager__Destructor                        0000016D
          ProtocolStateManager__Constructor                       000001D4
          ProtocolStateManager__ProcessDelayedEvents              0000027D
    > TransportThreadManager
    > TransportManager
```

Finally, the local types window contains all of the reconstructed data structures:



About the Analysis

Like the previous analysis of ComRAT v4, this analysis was conducted purely statically. Like the previous, I have reverse engineered every function in the binary that is not part of the C++ standard library, and some of those that are. Like the previous, all analysis was conducted in Hex-Rays, so you will not find anything particularly interesting in the plain disassembly listing. Unlike the previous, this binary had RTTI, meaning that I was given the names and inheritance relationships of classes with virtual functions.

Each C++ program that I devote significant time to analyzing seems to present me with unique challenges. With ComRAT, those were scale and usage of modern additions to the STL that had been previously unfamiliar to me. With XAgent, it was forcing myself to muddle through the subtleties of how MSVC implements multiple inheritance. For FlawedGrace, those challenges were:

- Extensive use of virtual functions and inheritance, beyond anything I've analyzed previously. Tracing the flow of data from point A to point B often involved around a dozen different object types and virtual function calls, sometimes more. You can see an example of this in the database notepad, where I describe the RDP tunneling implementation.

- A type reconstruction burden that seemed to never end. FlawedGrace has one of the highest ratios of custom types to program size of anything I've analyzed. In total, I manually reconstructed 178 custom data types across 454 programmer-written functions, which you will find in the Local Types window.

- Having to reverse engineer a complex virtual file system statically, with no sample data. You can find the relevant code in the functions window, under the folder path Modalities\Standalone\Virtual File System. I suspect this was written by a different team than the networking component, given the difference in coding styles: i.e., the VFS was written in plain C, with some features that mimic VTables.

- Having to confront, as a user, the challenges that reverse engineering tools have with x86/Windows programs (in contrast to x64) with regards to stack pointer analysis and 64-bit integers.

- Having to brush up on my network programming skills. For example, I had forgotten what the “Nagle algorithm” was. It’s clear that the server-side component is derived from the same codebase. However, the server portion of the code was not present in the binary, so I could not analyze it.

FlawedGrace makes proficient use of C++ features and the STL, and its authors are experts in concurrent programming and networking. However, it is mostly written in an older style than ComRAT was; for example, it does not use <memory>. Here is a list of the STL data types used, in descending frequency of usage:

- <atomic>

- thread

- list<T>

- map<K,V>

- deque<T>
- set<T>
- vector<T>

I hope you enjoy the IDB.