

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](mailto:rolf@mydomain).)

About the IDB

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

```

1 bool __fastcall sub_433C60(int *a1, int a2)
2 {
3     // [COLLAPSED] LOCAL DECLARATIONS. PRESS KEYP
4
5     v3 = *(_DWORD *)(a2 + 20);
6     if ( *((_BYTE *)a1 + 56) )
7     {
8         if ( v3 )
9             v12 = *(_QWORD *)(a1[6] + 40 * v3);
10        else
11            v12 = 0i64;
12        v4 = *(_DWORD *)(a2 + 12);
13        if ( v4 )
14            v13 = *(_QWORD *)(a1[6] + 40 * v4);
15        else
16            v13 = 0i64;
17        v5 = *(_DWORD *)(a2 + 24);
18        if ( v5 )
19            v14 = *(_QWORD *)(a1[7] + 40 * v5);
20        else
21            v14 = 0i64;
22        v6 = &v12;
23        v15 = *((_BYTE *)a2 + 36);
24        v16 = *(_WORD *)(a2 + 38);
25    }
26    else
27    {
28        if ( v3 )
29            v17 = *(_DWORD *)(a1[6] + 40 * v3);
30        else
31            v17 = 0;
32        v7 = *(_DWORD *)(a2 + 12);
33        if ( v7 )
34            v18 = *(_DWORD *)(a1[6] + 40 * v7);
35        else
36            v18 = 0;
37        v8 = *(_DWORD *)(a2 + 24);
38        if ( v8 )
39            v19 = *(_DWORD *)(a1[7] + 40 * v8);
40        else
41            v19 = 0;
42        v6 = (__int64 *)&v17;
43        v20 = *((_BYTE *)a2 + 36);
44        v21 = *(_WORD *)(a2 + 38);
45    }
46    v22 = *(_DWORD *)a2;
47    v23 = *(_DWORD *)(a2 + 4);
48    v11 = *a1;
49    v9 = *(void (__cdecl **)(int, int, int *, ch
50    v24 = 0;
51    v9(v11, 5, &v22, &v24);
52    if ( !v24 )
53        return 0;
54    v23 = a1[15];
55    return a1[15] == (*(int (__cdecl **)(int, in
56}

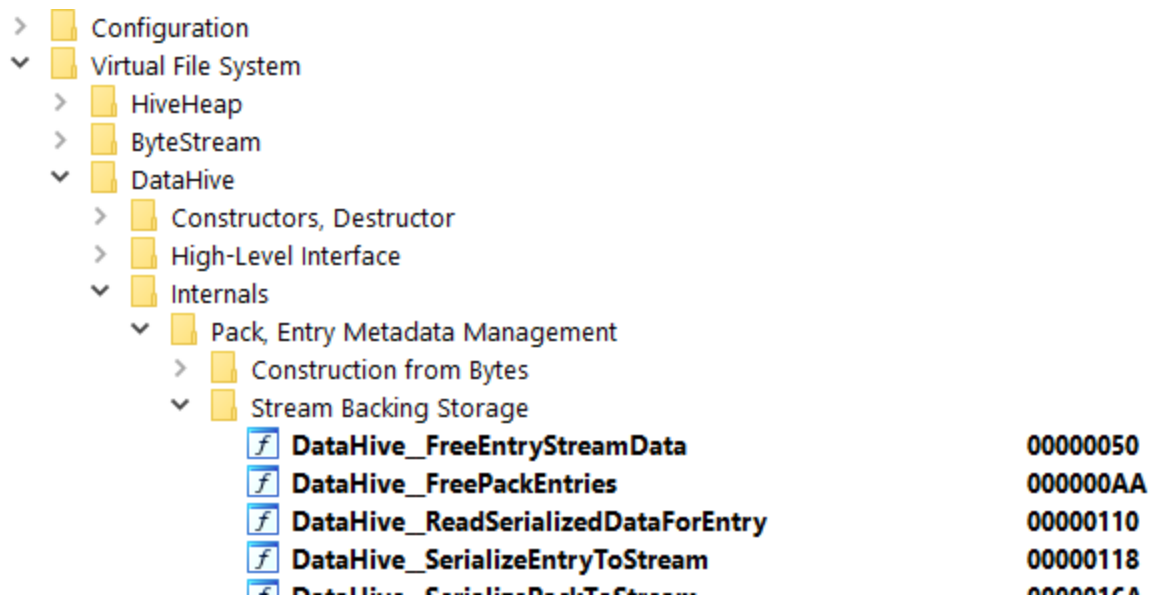
```

















































```

1 // Used when creating or modifying a DataHivePack. Creates a
2 // serialized representation of the pack metadata, and write
3 // the raw bytes into the stream. Return false on stream failure.
4 bool __fastcall DataHive::SerializePackToStream(DataHive *this, DataHivePack *aPack)
5 {
6     unsigned int vSiblingIdx; // eax
7     unsigned int vFirstChildIdx; // eax MAPDST
8     unsigned int vFirstEntryIdx; // eax MAPDST
9     DWORD *vpSerializedPack; // ebx
10    size_t (__cdecl *fpStream)(ByteStream *, StreamOperations, DWORD *, DWORD *); // eax
11    ByteStream *vpStream; // [esp-10h] [ebp-58h]
12    SerializedPack64 vSer64; // [esp+Ch] [ebp-3Ch] BYREF
13    SerializedPack32 vSer32; // [esp+28h] [ebp-20h] BYREF
14    unsigned __int64 vThisPackStreamPos; // [esp+38h] [ebp-10h] BYREF
15    char vSetPosStatus; // [esp+43h] [ebp-5h] BYREF
16
17    vSiblingIdx = aPack->dwNextSiblingPackIdx;
18    // The DataHive class can use 64-bit or 32-bit encodings for
19    // stream positions. The latter are obviously smaller. Here we
20    // determine which encoding is being used.
21    if ( this->bUse64BitOffsets )
22    {
23        // The serialized packs store the stream position of the next sibling pack.
24        // Copy it if there is one; use 0 if not.
25        if ( vSiblingIdx )
26            vSer64.qwStreamPos_NextSiblingPack = this->pPacksMem[vSiblingIdx].qwStreamPos;
27        else
28            vSer64.qwStreamPos_NextSiblingPack = 0i64;
29        // Copy the stream position of the first child pack, or 0.
30        vFirstChildIdx = aPack->dwFirstChildPackIdx;
31        if ( vFirstChildIdx )
32            vSer64.qwStreamPos_FirstChildPack = this->pPacksMem[vFirstChildIdx].qwStreamPos;
33        else
34            vSer64.qwStreamPos_FirstChildPack = 0i64;
35        // Copy the stream position of the first entry, or 0.
36        vFirstEntryIdx = aPack->dwFirstEntryIdx;
37        if ( vFirstEntryIdx )
38            vSer64.qwStreamPos_FirstEntry = this->pEntriesMem[vFirstEntryIdx].qwEntryStreamPos;
39        else
40            vSer64.qwStreamPos_FirstEntry = 0i64;
41        vpSerializedPack = (DWORD *)&vSer64;
42        // Copy pack name metadata
43        vSer64.bEntryNameIsWideString = aPack->mPackNameIsWideString;
44        vSer64.wEntryNameLen = aPack->mPackNameLen;
45    }
46    else
47    {
48        // The serialized packs store the stream position of the next sibling pack.
49        // Copy it if there is one; use 0 if not.
50        if ( vSiblingIdx )
51            vSer32.dwStreamPos_NextSiblingPack = this->pPacksMem[vSiblingIdx].qwStreamPos;
52        else
53            vSer32.dwStreamPos_NextSiblingPack = 0;
54        // Copy the stream position of the first child pack, or 0.
55        vFirstChildIdx = aPack->dwFirstChildPackIdx;
56        if ( vFirstChildIdx )
57            vSer32.dwStreamPos_FirstChildPack = this->pPacksMem[vFirstChildIdx].qwStreamPos;
58        else
59            vSer32.dwStreamPos_FirstChildPack = 0;
60        // Copy the stream position of the first entry, or 0.
61        vFirstEntryIdx = aPack->dwFirstEntryIdx;
62        if ( vFirstEntryIdx )
63            vSer32.dwStreamPos_FirstEntry = this->pEntriesMem[vFirstEntryIdx].qwEntryStreamPos;
64    }

```

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



	 DataHive_SerializePackIOStream	000016A
	 DataHive_WriteSerializedDataForEntry	000001C0
	>  Index Management	
	>  Removal	
	>  Stream Data Management	
	>  Retrieval	
	 IterativeCrackDataHivePathA	0000009D
	 IterativeCrackDataHivePathW	000000A9
>	 Backdoor Commands	
▼	 Channels	
	>  RDP	
	>  Download	
	>  Upload	
	>  GenericChannelDescriptor	
	 EnqueueDataHiveAsChannelWriteEntry	00000048
	 ShutdownChannelByID	00000055
	 EnqueueChannelWriteEntry	00000078
	 ShutdownChannelsOfType	000000B1
	 ChannelThreadProc	000000F2
	 CreateChannel	00000117
	 g_List_GenericChannelDescriptor_EnqueueChannelMessage	0000014D
▼	 Grace	
	>  GraceThread	
	>  GraceTunnelIO	
	>  GraceObject	
>	 Network	
>	 Thread Procedures	
▼	 Global State	
	▼  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:

>	COM				
>	Global State				
>	STL				
>	Networking				
>	Cryptography				
>	Modules				
>	Miscellaneous				
>	Grace				
>	GraceThread				
>	vector				
>	GraceTransportThread				
	221	GraceObjectThread		=> 228	
	222	GraceTransportReadThread	00000014	Auto	struct __cppobj : GraceTransportThread {}
	223	GraceTransportWriteThread	00000014	Auto	struct __cppobj : GraceTransportThread {}
	426	GraceTransportThread	00000014	Auto	struct __cppobj : GraceThread {TransportThreadManager *mp
>	GraceObjectThread				
	228	GraceObjectThread	00000014	Auto	struct __cppobj __declspec(align(4)) : GraceThread {ProtocolSt
	229	GraceDelayThread	00000014	Auto	struct __cppobj : GraceThread {ProtocolStateManager *m_Syn
>	GraceWireClientConnectionThread				
	236	GraceWireClientConnectionThread	00000078	Auto	struct __cppobj __declspec(align(4)) : GraceThread {SOCKET m
	474	GWCCCT_SYN_MainServerConnection	00000034	Auto	struct {unsigned int mCRC;GUID mGlobalManagerInstanceGUI
	475	GWCCCT_ACKMessage	0000002A	Auto	struct __unaligned __declspec(align(2)) {int mCRC;__int16 mwM
	476	GWCCCT_SYN_Channel	00000026	Auto	struct __unaligned __declspec(align(2)) {unsigned int mCRC;GU
	477	GWCCCT_SynType	00000004	Auto	enum {GWCCCT_Invalid = 0x0,GWCCCT_SynString = 0x1,GWCCCT_
	473	GWCCCT_SYN_Type1String	00000028	Auto	struct {unsigned int mCRC;unsigned __int16 mStringLen;unsig
	501	GWCCCT_FirstSYNGeneric	0000000E	Auto	struct __unaligned {unsigned int mCRC;unsigned int mMagic;u
>	Tunnelling				
	232	GraceTunnelReadThread	00000014	Auto	struct __cppobj : GraceThread {GraceTunnelIO *m_Tunnel;}
	233	GraceTunnelWriteThread	00000014	Auto	struct __cppobj : GraceThread {GraceTunnelIO *m_Tunnel;}
	216	GraceThread	00000010	Auto	struct __cppobj {GraceThread_vtbl * _vftable /*VFT*/;bool blsa
	217	GraceThread_vtbl	0000000C	Auto	struct /*VFT*/ {GraceThread *_thiscall *VirtualDestructor}(Gr
>	GraceTunnelIO				
>	GraceTunnelClientIO				
	235	GraceTunnelClientIO	00000048	Auto	struct __cppobj __declspec(align(4)) : GraceTunnelIO {GraceTu
>	GraceTunnelClientDirectIO				
	237	GraceTunnelClientDirectIO	00000054	Auto	struct __cppobj __declspec(align(4)) : GraceTunnelIO {GraceSe
	230	OutgoingTunnelMessageQueue	00000030	Auto	struct __declspec(align(4)) {struct RTL_CRITICAL_SECTION mC!
	231	GraceTunnelIO	00000044	Auto	struct __cppobj {GraceTunnelIO_vtbl * _vftable /*VFT*/;SOCKET
	372	GraceTunnelIO_vtbl	00000010	Auto	struct /*VFT*/ {GraceTunnelIO *_thiscall *VirtualDestructor}(G
	422	TunnelDataItem	00000008	Auto	struct {unsigned __int8 *data;size_t size;}
	423	deque_pTunnelDataItem	00000014	Auto	struct __declspec(align(4)) {deque_pTunnelDataItem *_Myprox
>	GraceObject				
>	GraceWireGeneric				
>	GraceSessionGeneric				
	238	GraceObject	0000004C	Auto	struct __cppobj {GraceObject_vtbl * _vftable /*VFT*/;unsigned
	241	GraceTunnelClient	00000054	Auto	struct __cppobj : GraceObject {void (_stdcall *fpWriteCallback
	269	GraceObjectVariety_t	00000004	Auto	enum {GraceObjectVariety_ServerManager = 0x1,GraceObject
	370	GraceObject_vtbl	00000014	Auto	struct /*VFT*/ {GraceObject *_thiscall *VirtualDestructor}(Gra
	413	ProtocolCode	00000004	Auto	enum {Code_0 = 0x0,Code_WireBA_101_NETWORK_ERROR =
	415	ProtocolCodeData	00000004	Auto	union __declspec(align(4)) {MyWSAError_t pc101;WireMessag
	458	CommandReceived	0000001C	Auto	struct __declspec(align(4)) {GUID mGuid;char *mTargetName;v
	459	NetworkCallbackMessage4	0000001C	Auto	struct __declspec(align(4)) {GUID mGuid;int iUnk10;unsigned _
	460	ChannelLocator	00000018	Auto	struct __declspec(align(4)) {int mChannelID;GUID mGuid;Chann
	461	ChannelDataMessage	00000020	Auto	struct __cppobj : ChannelLocator {unsigned __int8 *mData;size
	462	ChannelWriteMessage	00000020	Auto	struct __cppobj : ChannelLocator {int iUnk18;int mSentLength;}
	463	NetworkCallbackMessages	00000004	Auto	union {MyWSAError_t Code1;CommandReceived *CommandF
	471	ServerDescriptor	00000102	Auto	struct {char mAddress[256];unsigned __int16 mPort;}
>	Windows				
>	Virtual File System				
>	Channels				

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.