How to Analyze Malware for Technical Writing

By by Alexey Kleymenov | January 25, 2022

Archived: 2026-04-05 16:30:43 UTC

Analyzing New Malware

In the ever-changing world of cybersecurity, new threats appear and evolve on a regular basis. Sharing information about them is an important part of fighting cybercrime and keeping people and organizations safe. To do so efficiently, being prepared will make the best use of your—and your team's—time when analyzing an emerging threat.

In this blog, we cover various situations that researchers encounter when they need to publish their findings and provide some suggestions on how to approach them, along with a suggested workflow for approaching the analysis most efficiently. Finally, we apply this strategy to analyze a ransomware sample.

Challenges and Solutions

When a new threat emerges, there are a few common challenges that researchers face during analysis. Here are a few ways to handle them so you can produce clear and purposeful findings.

Urgency

In many cases, there is a relatively narrow window of time in which to release the publication, if we want the topic to be hot and the corresponding material to be relevant.

The solution is to focus on the most important questions that need answers.

- Who are the potential readers of the article? How will they benefit from reading it?
- How will the time costs associated with each section compare to its benefits?

Beginning your work by answering these questions will help shape the material in the right direction and manage time properly.

Novelty

For many attacks that hit the news, the related malware may not yet have been analyzed by other researchers. This increases the amount of work required to understand all parts of the relevant functionality, as there is little to no information to use as a starting point.

To address this issue, it is worth remembering that in many cases, modern malware families and attacker groups already have some roots. Tracking these connections allows researchers to find previous iterations of similar projects and reduce the amount of time required to understand malware's functionality.

Complexity

The consequences of simple cyberattacks aren't generally big enough to attract the attention of the public. What that means for researchers is that if something is worth writing an article about, it's likely to be quite complex and therefore time-consuming to analyze.

The solution here might be to split the big task into smaller tasks. Apart from prioritizing based on the article's focus, it also allows the analysis to done by a group, with different people focusing on different parts of functionality. Exchanging knowledge on a regular basis about what has already been covered will help the team to be efficient and not waste time analyzing the same parts multiple times.

Suggested Workflow

Here is a common workflow that should allow researchers to approach the analysis of new executable samples efficiently and effectively.

The second step, Behavioral Analysis, refers to the blackbox-style analysis that generally involves the execution of a sample under various monitoring tools and on sandboxes. The Dynamic Analysis step refers the use of a debugger to execute instructions.

Steps	Actions
1. Triage	• Collect as much easily-accessible open information as possible. This can come from existing articles, public sandbox reports, or other vendors' detection • Check for the presence of high-entropy blocks, import table or syscalls and strings to understand if it likely to be packed or not. • Check if some official (non-malicious) packers were used by using packer detection tools.
2. Behavioral Analysis	• Conduct this analysis if it is easy to restore the lab environment after execution. • It may not be necessary if good public sandbox reports are already available. • Keep in mind that, often, behavioral analysis doesn't show the full picture. • It may not go as expected because of anti-RE techniques involved.
3. Unpacking – Optional	• Not necessarily present, some malware developers prefer to only use obfuscation. • For official packers, there are multiple existing unpacking tools and scripts already available. • Ideally, the unpacked sample should remain executable to make the dynamic analysis easy. Otherwise, get as much unpacked code and

Steps	Actions
	data as possible.
4. Static and Dynamic Analysis of the Actual Functionality	• This step only becomes possible once the unpacking is done (if it was necessary). • Generally, strings and APIs give the maximum information and serve as important landmarks to facilitate navigation within the samples. • Keep the markup accurate: rename functions, create structures, define enums and leave comments where necessary. • Debugging is mainly needed to decrypt/decode/decompress code and data and resolve APIs. Static analysis is generally enough for the rest.

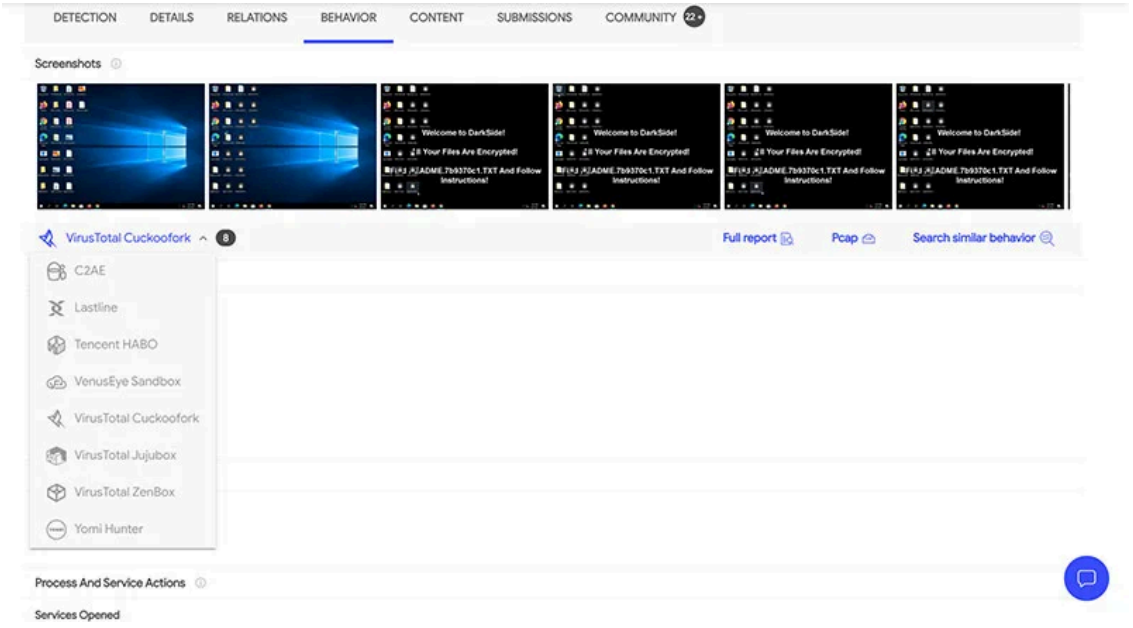
Applying the Workflow to Malware Analysis

Let's take a look at a DarkSide ransomware sample, which we analyzed earlier this year:

> 0a0c225f0e5ee941a79f2b7701f1285e4975a2859eb4d025d96d9e366e81abb9

Step 1: Triage

At the time of analysis, the sample had already been uploaded to Virustotal, so all cybersecurity community members could benefit from access and were able to see AV vendors' detections as well as the sandbox logs in the Behavior tab. Note that there are now multiple sandboxes supported in Virustotal, so try a few to find a good report.

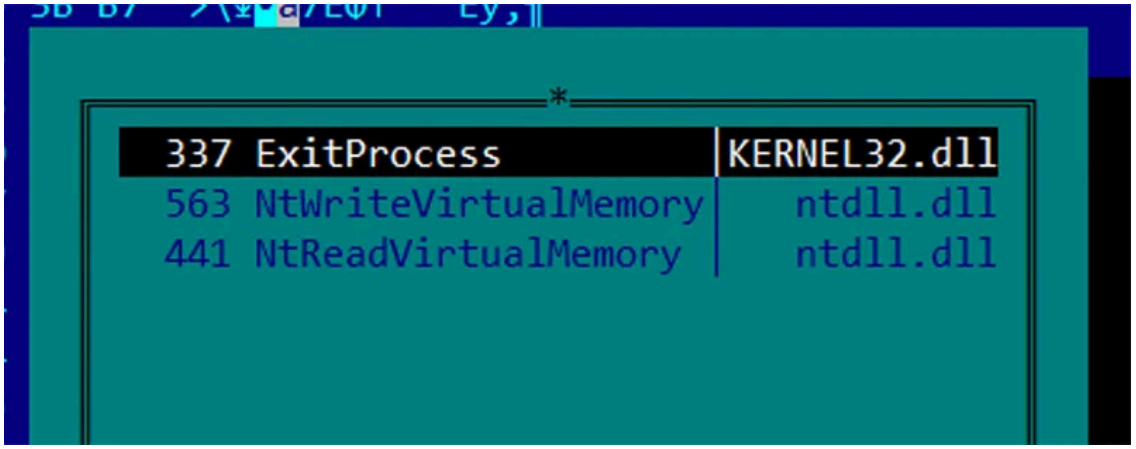


Multiple sandbox options on Virustotal.

A quick look at the sample in the hex editor reveals that there is a high-entropy block at the end. There are multiple things it could be: the next stage payload or another module, a blob containing encrypted strings or configuration, etc. Static analysis will be required to understand it.

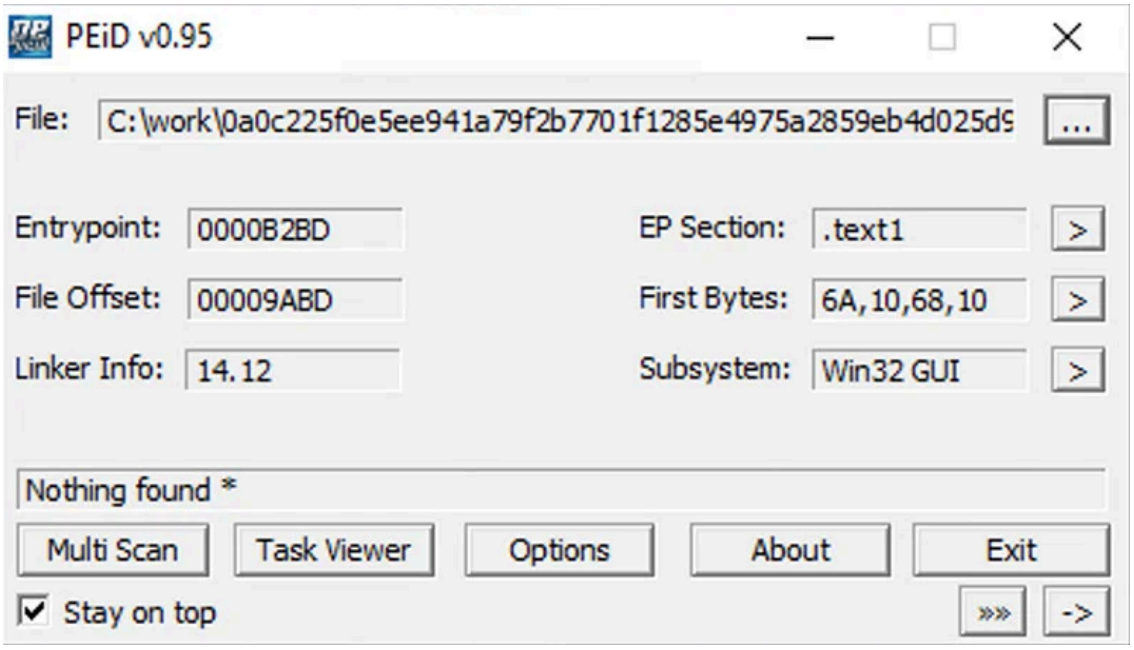
A high-entropy block.

There are pretty much no meaningful strings and APIs:



Very few entries in the import table.

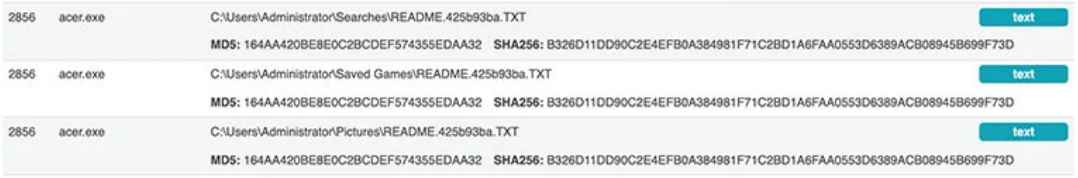
This is a strong indicator that the sample is obfuscated with APIs resolved dynamically and strings encrypted. Running a packer detection tool (PEiD with custom community signatures) confirms that there is no indication that public packers have been used in this case.



PEiD did not identify any known packers.

Step 2: Behavioral Analysis

By the time the analysis began, the sample had already been submitted to various public sandboxes by other community members, so lots of information could be taken from there.



File activity in the public any.run report.

Step 3: Unpacking

Checking cross-references to the high-entropy block in the disassembler, we can see that this doesn't seem to be the next stage payload as there is no control transfer to it or related blocks. In addition, a quick look around the disassembly confirms that the sample is indeed obfuscated rather than packed with multiple APIs resolved dynamically by hashes and with strings encrypted.

```
sub_401943 proc near
push    ebx
push    ecx
push    edx
push    esi
push    edi
push    1E2B04A4h
push    3B98045Eh
call    sub_401820
mov     dword_420978, eax
push    288B0588h
push    3B98045Eh
call    sub_401820
```

API resolution by hashes.

```
push    [ebp+var_C]
push    8
push    dword_4103B6
call    dword_420648
mov     [ebp+var_10], eax
cmp     [ebp+var_10],dword_420648    dd ?
jz      short loc_402
```

A call to the not-yet-resolved API.

Step 4: Static and Dynamic Analysis of the Actual Functionality

In order to be able to efficiently navigate the disassembly, we need to make APIs and strings easily readable.

For APIs, this is very easy to achieve with dynamic analysis as all the APIs are resolved in a single function. Therefore, letting it execute until the end will give us all the APIs' addresses. To propagate their names to the pointers, use standard renimp.idc script shipped as part of IDA Pro.

```
.data:00420704 ; BOOL (__stdcall *SetFileAttributesW)(LPCWSTR lpFileName, DWORD dwFileAttributes)
.data:00420704 SetFileAttributesW dd ?                 ; DATA XREF: resolve_APIs+46↑o
.data:00420704                                         ; StartAddress+2E↑r ...
.data:00420708 ; DWORD (__stdcall *GetFileAttributesW)(LPCWSTR lpFileName)
.data:00420708 GetFileAttributesW dd ?                 ; DATA XREF: sub_4031C2+15↑r
.data:00420708                                         ; StartAddress+1F↑r ...
.data:0042070C ; HANDLE (__stdcall *FindFirstFileExW)(LPCWSTR lpFileName, FINDEX_INFO_LEVELS fInfo
.data:0042070C FindFirstFileExW dd ?                   ; DATA XREF: find_all_files+95↑r
.data:0042070C                                         ; find_recycle_and_delete+68↑r ...
.data:00420710 ; BOOL (__stdcall *FindNextFileW)(HANDLE hFindFile, LPWIN32_FIND_DATAW lpFindFileDa
.data:00420710 FindNextFileW   dd ?                    ; DATA XREF: find_all_files+CA↑r
.data:00420710                                         ; find_recycle_and_delete+BF↑r ...
.data:00420714 ; BOOL (__stdcall *FindClose)(HANDLE hFindFile)
.data:00420714 FindClose       dd ?                    ; DATA XREF: find_all_files+D7↑r
.data:00420714                                         ; find_recycle_and_delete+CC↑r ...
.data:00420718 ; BOOL (__stdcall *MoveFileExW)(LPCWSTR lpExistingFileName, LPCWSTR lpNewFileName,
.data:00420718 MoveFileExW     dd ?                    ; DATA XREF: send_file_to_worker_to_encrypt
```

Resolved APIs' names.

This approach won't work for strings, as they're decrypted on an ad-hoc basis just before being used, rather than in a single place. Therefore, to make them easily visible, scripting will be required. In our blog on Darkside, we have already provided such a script that will attempt to find all the encrypted strings and decrypt them.

```
mov     [ebp+cbData], 0
push    offset unk_40DFC1
call    decrypt_block
mov     esi, eax
test    esi, esi
jz      loc_401EE9
lea     eax, [ebp+dwDispos
push    eax                 ;
lea     eax, [ebp+Handle]
push    eax                 ;
push    0                   ;
push    20119h              ;
push    0                   ;
push    0                   ; lpClass
push    0                   ; Reserved
push    esi                 ; lpSubKey
push    80000002h           ; hKey
call    RegCreateKeyExW
```

```
unk_40DFC1      db  46h ; F
                db  79h ; y
                db    6
                db 0B7h ; ·
                db    2
                db  4Bh ; K
                db 0B2h ; ²
                db  6Bh ; k
                db  92h ; ’
                db 0DDh ; Ý
```

Before string decryption.

```
mov     [ebp+cbData], 0
push    offset cryptogr ; "SOFTWARE\\Microsoft\\Cryptography"
call    decrypt_block
mov     esi, eax
test    esi, esi
jz      loc_401EE9
lea     eax, [ebp+dwDisposition]
push    eax             ; lpdwDisposition
lea     eax, [ebp+Handle]
push    eax             ; phkResult
push    0               ; lpSecurityAttributes
push    20119h          ; samDesired
push    0               ; dwOptions
push    0               ; lpClass
push    0               ; Reserved
push    esi             ; lpSubKey
push    80000002h       ; hKey
call    RegCreateKeyExW
```

After string decryption.

That's it. Now when both strings and APIs are visible, the only thing left to engineer is to carefully go through cross references and keep the markup for the corresponding functions describing all potentially interesting information (subject to the target audience) in the article.

Conclusion

Knowledge sharing is an important part of the cybersecurity field that allows us to quickly adapt to new threats and minimize their associated risks. By properly focusing our efforts, we can improve the quality of this process and make the world a safer place.

Extra Tips

- **Know your audience** – the content of the technical blog post (and the corresponding questions to answer) will be very different from a news article for the general public
- **Consider teamwork to speed up the process** – Asking for help if at an early stage helps increase the total time available for the analysis
- **Have your templates ready** – simple scripts to decrypt / decode / decompress the data may help avoid unnecessary delays

---

Source: https://www.nozominetworks.com/blog/how-to-analyze-malware-for-technical-writing/