A Curious Case of Malwarebytes

vx-underground.org collection // [0x1337dtm](https://twitter.com/0x00dtm)



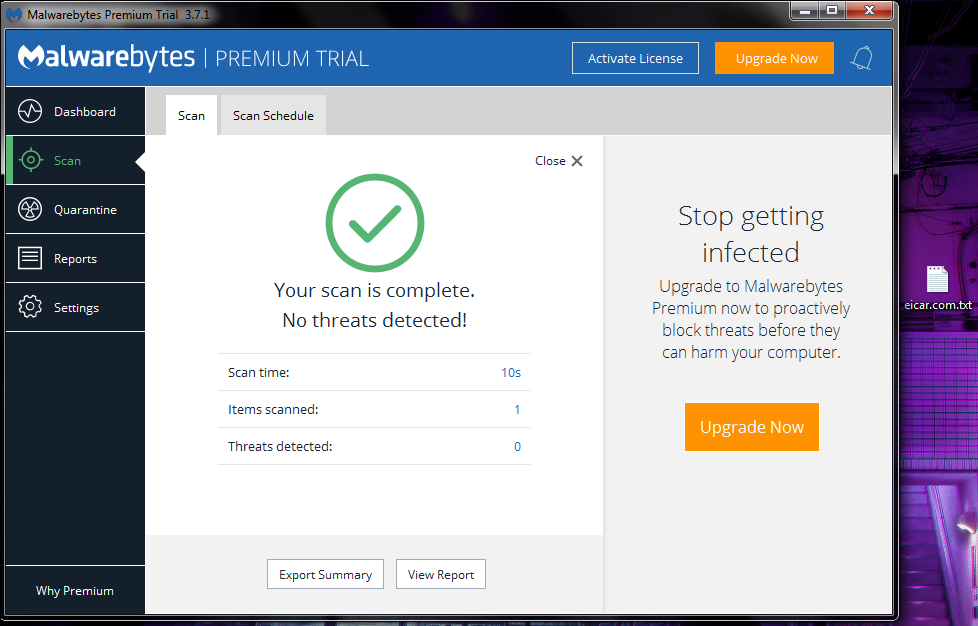
# **Introduction**

Recently, I installed Malwarebytes on my machine. I played around with it for a little and I noticed that something was off with the scanning of files on disk. Naturally, it tempted me into digging further to identify the root cause but I had also unexpectedly discovered something else that was incredibly strange. So, a couple of issues and hours of investigation later, I would like to present to you what I’ve uncovered in this journey.

# **What File Scanning?**

For those unaware, the EICAR test file is a file developed to test anti-malware solutions by intentionally triggering a detection based on the following ASCII string: X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H\*. For more information about this, please see here:<https://www.eicar.org/>

As I completed the installation of Malwarebytes, I wanted to test it with the classic EICAR file. So I dropped it into disk and waited for a detection notification… and nothing…? Malwarebytes did not bat an eyelid. I escalated it further and explicitly requested a scan on the file… and nothing!

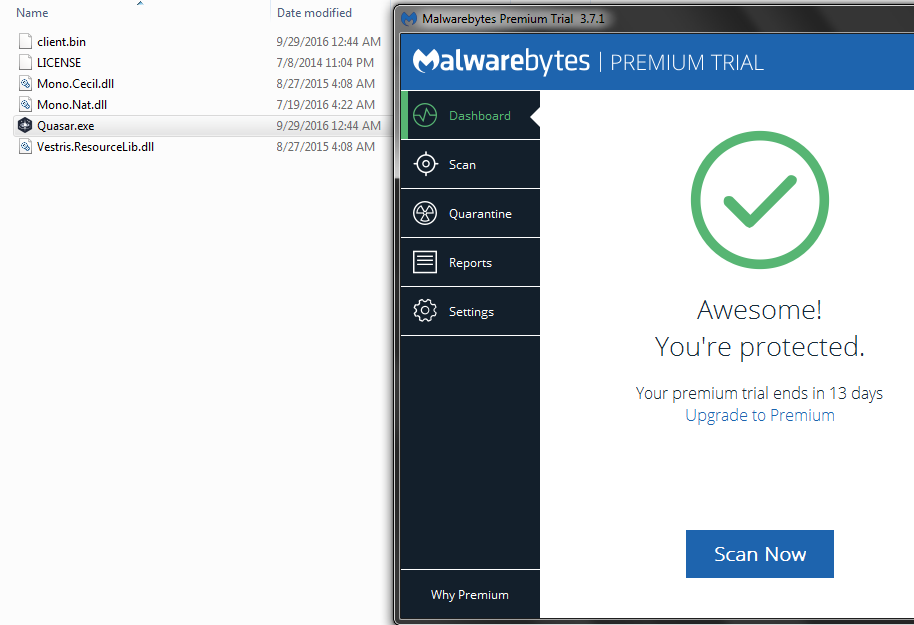


After Googling the issue, here is the summary in the [FAQ 8](https://forums.malwarebytes.com/topic/191650-malwarebytes-3-frequently-asked-questions/?do=findComment&comment=1077438) of why Malwarebytes does not detect EICAR files:

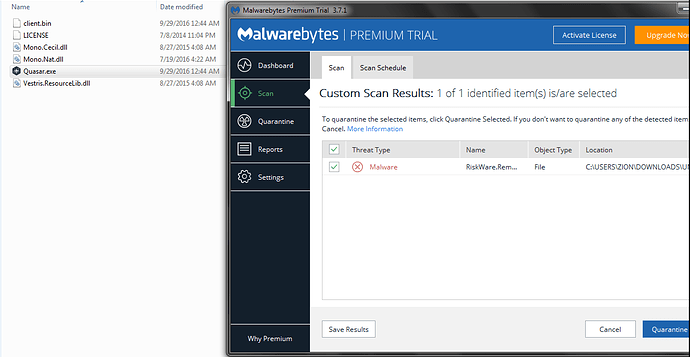
*So in summary, MB3 already incorporates world-class, next-generation anti-malware technologies. Our combination of signature-less and rules-based layered approach is far more effective than using AV signatures. Malwarebytes is able to prevent 0-minute threats and attacks without updates, even script-based, file-less, and other advanced attacks . We won’t detect EICAR because EICAR is not representative of either today’s threat environment or security needs.* - MalwareBytes

Weird flex, but okay.

After this unique incident, I decided to try and see if it would detect *anything* that was dropped to disk so I decided to find something that *should* guarantee a detection: Quasar RAT. I downloaded it and unzipped it, waiting in anticipation for the detection notification… and nothing! Again!



I performed a manual scan on the *Quasar.exe* file to check if Malwarebytes was actually functional at all and, lo and behold, it picked it up!



# **Investigating the Issue**

Of course, I was not impressed by what I have seen. I wanted to investigate what was causing this problem to resolve it. Having messed with a little bit of kernel driver development before, I knew where to look.

## **Minifilter Callback Operations**

Windows has a special type of drivers called *Minifilters* which are used for file system operations. To register as a minifilter, the driver must use the [FltRegisterFilter](https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/fltkernel/nf-fltkernel-fltregisterfilter) registration function. One of the parameters specify the registration context which is a struct that holds the relevant information to be provided to the kernel.

|  |
| --- |
| typedef struct \_FLT\_REGISTRATION {  USHORT Size;  USHORT Version;  FLT\_REGISTRATION\_FLAGS Flags;  const FLT\_CONTEXT\_REGISTRATION \*ContextRegistration;  const FLT\_OPERATION\_REGISTRATION \*OperationRegistration; // <--  PFLT\_FILTER\_UNLOAD\_CALLBACK FilterUnloadCallback;  PFLT\_INSTANCE\_SETUP\_CALLBACK InstanceSetupCallback;  PFLT\_INSTANCE\_QUERY\_TEARDOWN\_CALLBACK InstanceQueryTeardownCallback;  PFLT\_INSTANCE\_TEARDOWN\_CALLBACK InstanceTeardownStartCallback;  PFLT\_INSTANCE\_TEARDOWN\_CALLBACK InstanceTeardownCompleteCallback;  PFLT\_GENERATE\_FILE\_NAME GenerateFileNameCallback;  PFLT\_NORMALIZE\_NAME\_COMPONENT NormalizeNameComponentCallback;  PFLT\_NORMALIZE\_CONTEXT\_CLEANUP NormalizeContextCleanupCallback;  PFLT\_TRANSACTION\_NOTIFICATION\_CALLBACK TransactionNotificationCallback;  PFLT\_NORMALIZE\_NAME\_COMPONENT\_EX NormalizeNameComponentExCallback;  PFLT\_SECTION\_CONFLICT\_NOTIFICATION\_CALLBACK SectionNotificationCallback; } FLT\_REGISTRATION, \*PFLT\_REGISTRATION; |

Within this struct, there is one member that is interesting: OperationRegistration.

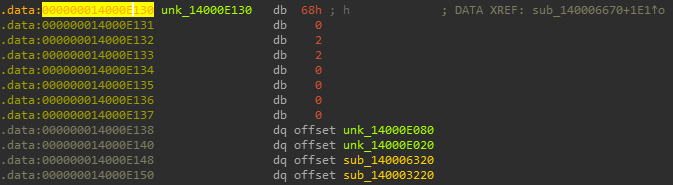
|  |
| --- |
| typedef struct \_FLT\_OPERATION\_REGISTRATION {  UCHAR MajorFunction;  FLT\_OPERATION\_REGISTRATION\_FLAGS Flags;  PFLT\_PRE\_OPERATION\_CALLBACK PreOperation;  PFLT\_POST\_OPERATION\_CALLBACK PostOperation;  PVOID Reserved1; } FLT\_OPERATION\_REGISTRATION, \*PFLT\_OPERATION\_REGISTRATION; |

This struct describes the type of operation that will be registered as a callback (MajorFunction) and the two functions that will handle the callback (PreOperation and PostOperation). The PreOperation handles the callback *before* the operation is performed and the PostOperation handles the callback *after* the operation is performed. This struct is used in an array that may specify multiple types of operations.

## 

## **The Tip of the Iceberg**

Using this knowledge, I discovered the registration structure used for FltRegisterFilter:

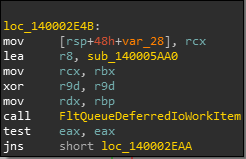


If we match the offsets of the above struct definition, we can deduce that the two green unk\_XXX values are the ContextRegistration and OperationRegistration respectively. We are interested in the second one:



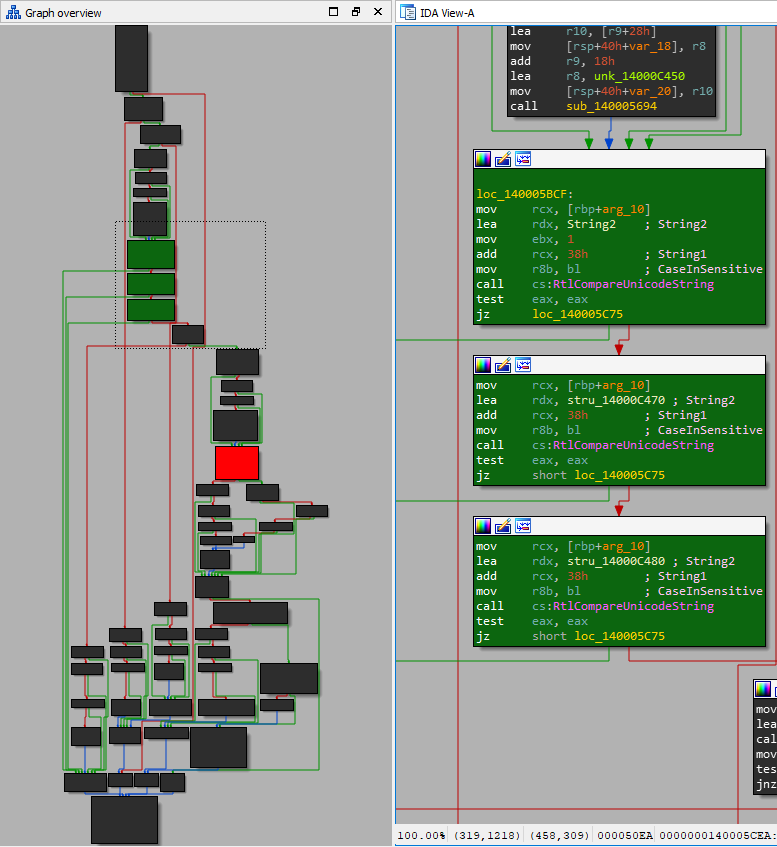
The figure above shows the OperationRegistration struct with the operations IRP\_MJ\_CREATE (file handle opens) and IRP\_MJ\_ACQUIRE\_FOR\_SECTION\_SYNCHRONIZATION. What’s strange here is that there is no registered callback registration for [IRP\_MJ\_WRITE](https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/irp-mj-write) (file writes) nor [IRP\_MJ\_CLEANUP](https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/irp-mj-cleanup) (file handle closes). This could track malicious byte patterns being written to a file as well as being able to scan a file after it has been opened and potentially modified. Perhaps this was the issue for failing to have scanned Quasar?

Anyway, the IRP\_MJ\_CREATE specifies both PreOperation and PostOperation. The PostOperation is only used for clean up so we are not interested in that. Let’s have a look at the PreOperation. The function is actually quite small and didn’t contain any relevant information about file scanning but I noticed the following deferred routine:

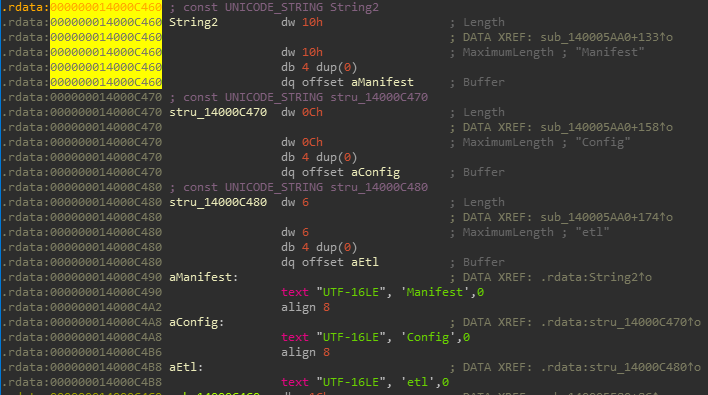


## **Taken By Surprise**

If we jump into sub\_140005AA0, we can see this:



In the green boxes, there are calls to [RtlCompareUnicodeString](https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-rtlcompareunicodestring), with the first argument as the file extension and the second argument as **hardcoded** strings! Here are the three strings:



Looking back at the flow chart, RtlCompareUnicodeString is defined to return zero if the two provided strings are the same. We can follow the green branches that satisfy this return value and see that execution flows all the way to the end of the function. The red box on the right contains the function that scans the file. If we wanted to bypass the file scanning function, all we need to do is to rename the file extension to Manifest, Config, or etl (case-insensitive)!



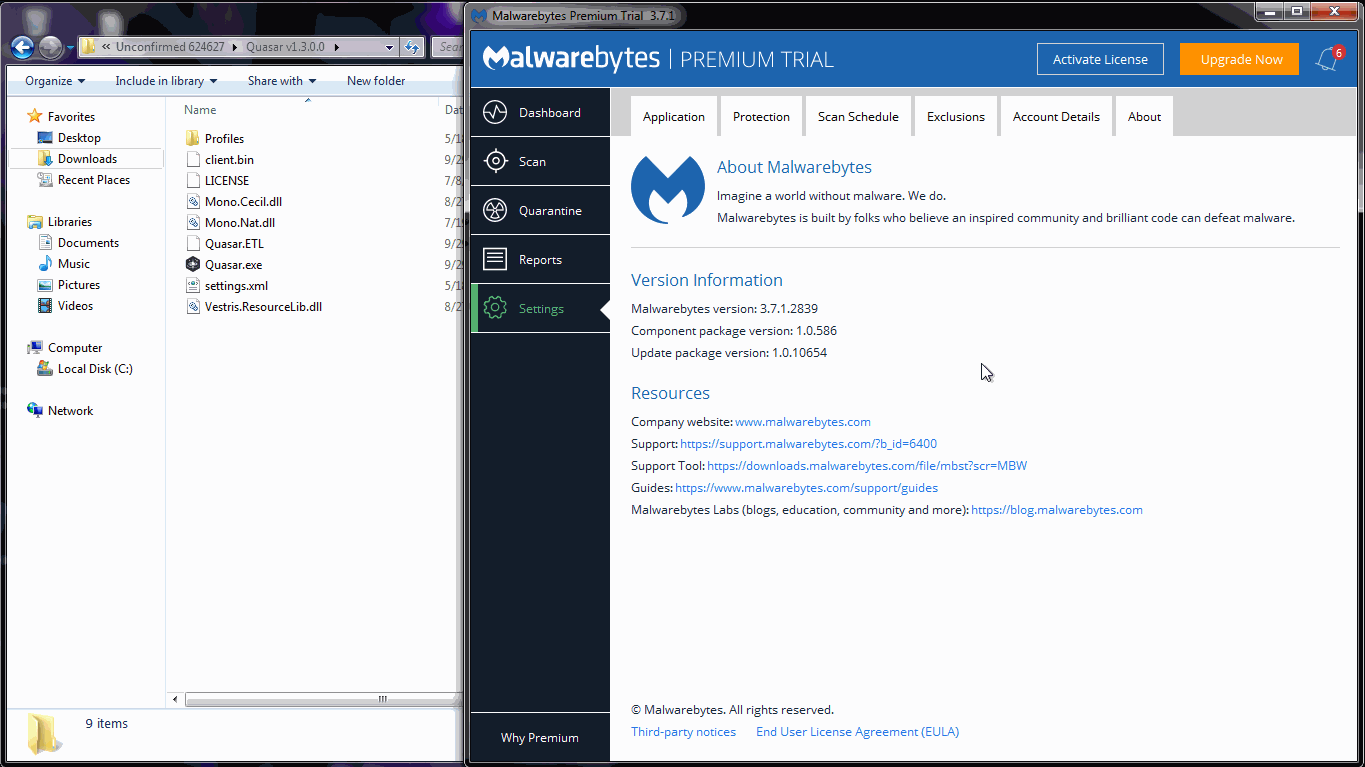
It is actually possible to run executable files despite lacking the exe extension name. I believe that the PATHTEXT environment variable plays a role in the [CreateProcess](https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa) function. The (my) PATHTEXT variable is defined as:

|  |
| --- |
| .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC |

From my educated guess, if the executing file has an unassociated file extension, it will iterate through these file types in order (left to right) and verify by analysing the file’s header. In the case of an exe, it will reach the .EXE value, be recognised as an executable by the MZ header signature, and then attempted to be executed as such. Correct me if I’m wrong.

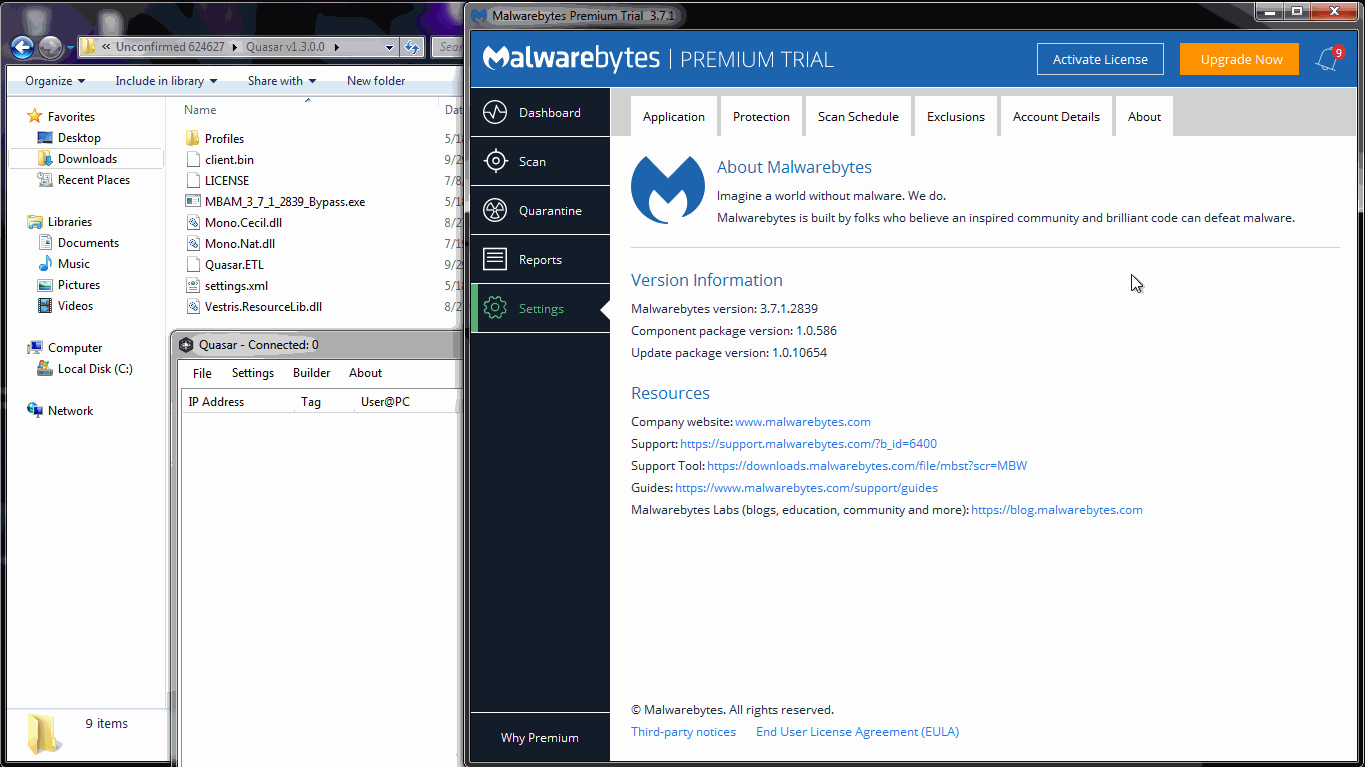
# **Demonstration**

In the following GIF, I will show that malicious files dropped to disk do not get detected. Then, by changing the file name extensions to etl and manifest, Malwarebytes will not also not see them.



## **Dropper PoC**

I’ve also developed a PoC dropper designed to automate this process.



# **Conclusion**

This journey started out quite strange and became even stranger. I have no idea why the implementation is missing some file system operation callbacks. It may explain why Malwarebytes is not scanning files when they are written to disk. I have even less of an idea as to why it was decided that these file extensions were whitelisted from scanning. Perhaps it was an optimisation of some sort? Maybe it was assumed that they weren’t executable? Let me know what you think.

As always, you can find the PoC here on my GitHub:<https://github.com/NtRaiseHardError/Antimalware-Research/tree/master/Malwarebytes>.

*– dtm*