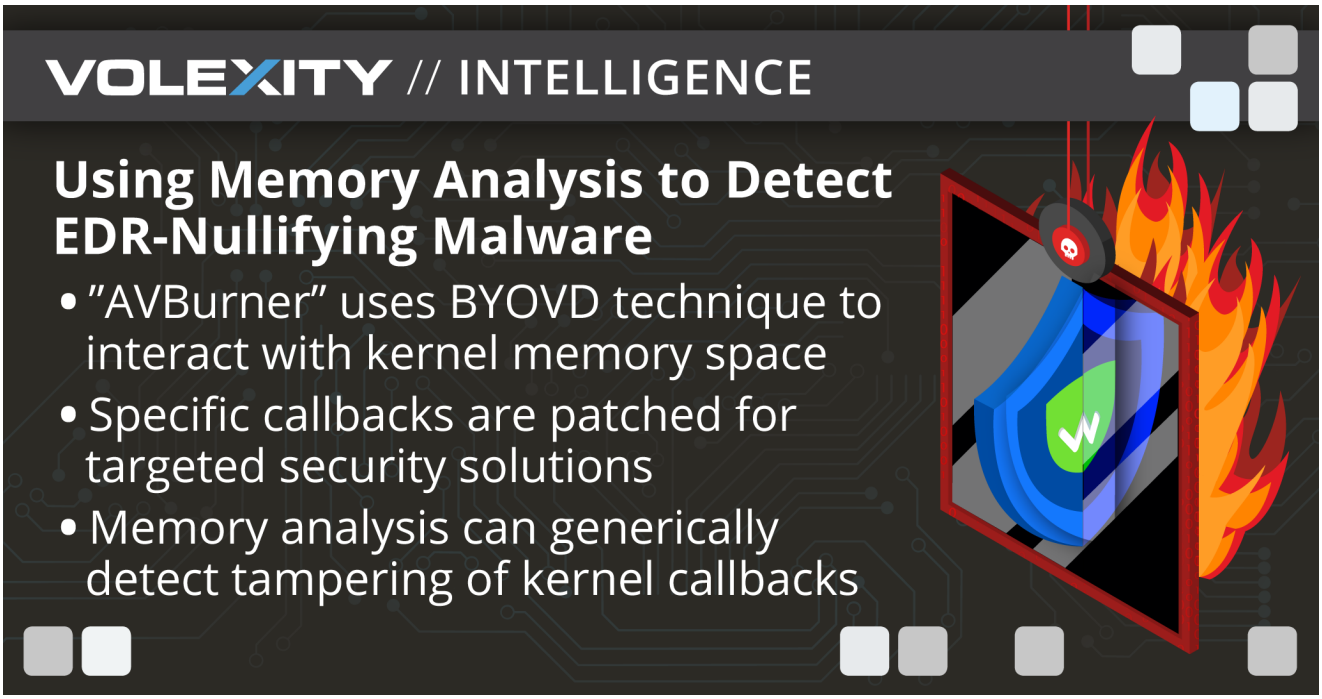
# Using Memory Analysis to Detect EDR-Nullifying Malware

volexity.com/blog/2023/03/07/using-memory-analysis-to-detect-edr-nullifying-malware/

March 7, 2023

by Paul Rascagneres, Volexity Volcano Team




In the ever-changing cybersecurity landscape, threat actors are forced to evolve and continually modify the tactics, techniques, and procedures (TTPs) they employ to launch and sustain attacks successfully. They are continually modifying their malware and command-execution methods to evade detection. The attackers in these cases are attempting to get a step ahead of security software at the most basic level.

However, some techniques take a different approach, aiming further up the stack and directly taking on security software. The most brazen methods involve leveraging various tools that directly terminate or shutdown security software. If successful, this method is effective at giving an attacker free reign on a system. However, it comes at the potential cost of alerting users or administrators that the software unexpectedly stopped reporting or was shut off. What about a technique that potentially flies a bit more under the radar?

In November 2022, Trend Micro published a blog post related a Chinese APT threat actor they called "Earth Longzhi", and that Volexity tracks as "SnakeCharmer". One tool Trend Micro described, dubbed "AVBurner", used a technique to patch process-creation callbacks in kernel memory to nullify security software running on a victim system. The end result: the security software would appear to be running fine, but in reality, it was neutered and

rendered useless. When successful, this approach allows an attacker to carry on their operations with little risk of being detected as a result of the security software. While there are several documented cases of attackers patching these callbacks, this particular technique leveraged by SnakeCharmer and others lends itself to specific methods of detection by way of memory analysis.

Volexity conducted research and testing to determine ways this technique of attacking endpoint detection and response (EDR) and antivirus (AV) software could reliably be detected through memory analysis. This blog:

- Examines the technique used by AVBurner, building upon the Trend Micro blog.
- Provides a refresher on the internal mechanism of the Windows callbacks and how AVBurner disables functionality of security products.
- Explains how memory analysis can identify manipulated EDR callbacks, such as those employed by AVBurner, using either via Volatility 3 or Volexity Volcano.

## Kernel Process Callbacks

In order to function, both EDR and AV solutions must effectively monitor process creation. When a process is created, security products can block it from starting or inject a library into it to monitor its behavior. To do so, most security products use a driver to register a kernel callback, most commonly using the PsSetCreateProcessNotifyRoutine() API.

Figure 1 shows the pseudocode of this function.

```
1 NTSTATUS __stdcall PsSetCreateProcessNotifyRoutine(PCREATE_PROCESS_NOTIFY_ROUTINE NotifyRoutine, BOOLEAN Remove)
2 {
3   return PspSetCreateProcessNotifyRoutine((__int64)NotifyRoutine, Remove != 0);
4 }
```

*Figure 1. PsSetCreateProcessNotifyRoutine() pseudocode*

As can be seen, this function's sole purpose is to execute another function, PspSetCreateProcessNotifyRoutine(). This API is not documented by Microsoft, but it can be understood by examining its disassembly pseudocode.

```
1  __int64 __fastcall PspSetCreateProcessNotifyRoutine(__int64 a1, unsigned int a2)
2  {
3    __int64 v2; // rsi
4    bool v4; // bl
5    __int64 v5; // rdx
6    void *v6; // rsi
7    __int64 v7; // rdi
8    struct _KTHREAD *CurrentThread; // r14
9    __int64 v9; // rbp
10   __int64 v10; // rax
11   struct _EX_RUNDOWN_REF *v11; // rdi
12
13   v2 = a2;
14   v4 = (a2 & 2) != 0;
15   if ( (a2 & 1) != 0 )
16   {
17     CurrentThread = KeGetCurrentThread();
18     --CurrentThread->KernelApcDisable;
19     v9 = 0i64;
20     while ( 1 )
21     {
22       v10 = ExReferenceCallBackBlock(&PspCreateProcessNotifyRoutine + v9);
23       v11 = (struct _EX_RUNDOWN_REF *)v10;
24       if ( v10 )
25       {
26         LODWORD(v2) = v2 & 0xFFFFFFFE;
27         if ( ExGetCallBackBlockRoutine(v10) == a1
28           && (unsigned int)ExGetCallBackBlockContext(v11) == (_DWORD)v2
29           && (unsigned __int8)ExCompareExchangeCallBack(&PspCreateProcessNotifyRoutine + v9, 0i64, v11) )
30         {
31           if ( v4 )
32             _InterlockedDecrement(&PspCreateProcessNotifyRoutineExCount);
33           else
34             _InterlockedDecrement(&PspCreateProcessNotifyRoutineCount);
35           ExDereferenceCallBackBlock(&PspCreateProcessNotifyRoutine + v9, v11);
36           KiLeaveCriticalRegionUnsafe(CurrentThread);
37           ExWaitForRundownProtectionRelease(v11);
38           ExFreePoolWithTag(v11, 0);
39           return 0i64;
40         }
41         ExDereferenceCallBackBlock(&PspCreateProcessNotifyRoutine + v9, v11);
42       }
43       v9 = (unsigned int)(v9 + 1);
44       if ( (unsigned int)v9 >= 0x40 )
45       {
46         KiLeaveCriticalRegionUnsafe(CurrentThread);
47         return 0xC000007Ai64;
48       }
49     }
50   }
```

*Figure 2. PspSetCreateProcessNotifyRoutine() pseudocode*

Figure 2 shows the pseudocode of this function when the second argument is true (i.e., when a callback is to be removed).

- Lines 19 and 44 of this pseudocode show a loop that increments the v9 variable from “0” to “0x40”. This variable is an index for the PspCreateProcessNotifyRoutine This array is used as a first argument to the ExDerefenceCallBackBlock() function and contains the callbacks. A maximum of “0x40” (64) callbacks can be defined.
- Lines 32 and 34 show two interesting variables: PspCreateProcessNotifyRoutineExCount and PspCreateProcessNotifyRoutineCount. If a callback is removed, one of these variables is decremented. The sum of these two variables is the number of defined callbacks.

- The ExCompareExchangeCallBack() API shows a mask (“0xFFFFFFFF`FFFFFFF0”) must be applied to the address stored in the PspCreateProcessNotifyRoutine

WinDBG can be used to confirm these conclusions. Figure 3 shows the PspCreateProcessNotifyRoutineExCount and PspCreateProcessNotifyRoutineCount values.

```
0: kd> db nt!PspCreateProcessNotifyRoutineExCount L1
fffff801`4ff4fef8  06
0: kd> db nt!PspCreateProcessNotifyRoutineCount L1
fffff801`4ff4fefc  07
```

*Figure 3. The count of defined callbacks*

In this example, there are 13 callbacks defined on this system.

Figure 4 shows the PspCreateProcessNotifyRoutine array (13 callbacks) and the called functions for the first three callbacks.

```
0: kd> dps nt!PspCreateProcessNotifyRoutine
fffff801`4fb43ba0  ffff8702`82063bcf
fffff801`4fb43ba8  ffff8702`822941bf
fffff801`4fb43bb0  ffff8702`855f1cff
fffff801`4fb43bb8  ffff8702`8560311f
fffff801`4fb43bc0  ffff8702`856e6def
fffff801`4fb43bc8  ffff8702`857f96ff
fffff801`4fb43bd0  ffff8702`84cf69af
fffff801`4fb43bd8  ffff8702`84cfe7bf
fffff801`4fb43be0  ffff8702`845c88cf
fffff801`4fb43be8  ffff8702`857fc89f
fffff801`4fb43bf0  ffff8702`84cfa87f
fffff801`4fb43bf8  ffff8702`8582c44f
fffff801`4fb43c00  ffff8702`863a456f
fffff801`4fb43c08  00000000`00000000
fffff801`4fb43c10  00000000`00000000
fffff801`4fb43c18  00000000`00000000
0: kd> dps ffff8702`82063bcf & 0xFFFFFFFFFFFFFFF0 L2
ffff8702`82063bc0  00000000`00000020
ffff8702`82063bc8  fffff801`4f8fd2f8 nt!ViCreateProcessCallback
0: kd> dps ffff8702`822941bf & 0xFFFFFFFFFFFFFFF0 L2
ffff8702`822941b0  00000000`00000020
ffff8702`822941b8  fffff804`ad945120 cng!CngCreateProcessNotifyRoutine
0: kd> dps ffff8702`855f1cff & 0xFFFFFFFFFFFFFFF0 L2
ffff8702`855f1cf0  00000000`00000020
ffff8702`855f1cf8  fffff804`ad679a00 ksecdd!KsecCreateProcessNotifyRoutine
```

*Figure 4. 13 defined callbacks and the definition of the first three*

In this example, each time a process is created, nt!ViCreateProcessCallback(), cng!CngCreateProcessNotifyRoutine(), ksecdd!KsecCreateProcessNotifyRoutine(), and the 10 others are called.

Armed with this knowledge, one can further analyze AVBurner and its approach to tampering with these callbacks.

## AVBurner Kernel Process Callback Bypass

| | |
|---|---|
| **Name(s)** | execute.exe |
| **Size** | 158.0KB (161792 Bytes) |
| **File Type** | Win64 EXE |
| **MD5** | 494cc48a9856cf5b46fb13bcd68c256f |
| **SHA1** | 39727e755b2806fc2ed5204dae4572a14b2d43d1 |
| **SHA256** | 4b1b1a1293ccd2c0fd51075de9376ebb55ab64972da785153fcb0a4eb523a5eb |

AVBurner is designed to disable callbacks from the kernel space. A userland application cannot modify kernel memory, so the malware authors include a vulnerable driver, RTCore64.sys, to read and write into this protected memory space. This driver has been previously reported as being used for the same purpose by ransomware groups. The threat actor in this case is not a ransomware attacker.

This technique of using an older, vulnerable driver to load malicious code was famously used by Turla for the purposes of loading a malicious rootkit. A public GitHub repository, KDU, owned by hFiref0x, documents a list of drivers that can be abused for this “Bring Your Own Vulnerable Driver” (BYOVD) technique.

AVBurner follows the same logic outlined in the *Kernel Process Callbacks* section to identify the callback array and disable specific callbacks. Specifically, it has the following workflow:

- Check the OS version, as this is required for the next step.
- Abuse RTCore64.sys to identify the PspCreateProcessNotifyRoutine array. The identification of this array is based on a byte pattern, which differs according to the OS version (hence the requirement for the previous step).
- Abuse RTCore64.sys to parse the array in order to get the list of currently defined callbacks.
- For each registered callback address, gets the location of any SYS file in the directory C:\Windows\ or its subdirectories. If the SYS file is not located in this directory it is skipped.
- Checks the metadata of the SYS file to see if the file matches specific criteria. This is to identify whether or not the callback must be removed.

- If the callback is identified as one that should be removed, AVBurner abuses RTCore64.sys to replace the callback address with “0x00000000`00000000” which effectively disables the callback.

This specific sample of AVBurner targets any drivers with the string “360” in their metadata description, meaning it was almost certainly configured to prevent security products by Qihoo 360 from functioning correctly. Figure 5 summarizes AVBurner’s workflow.

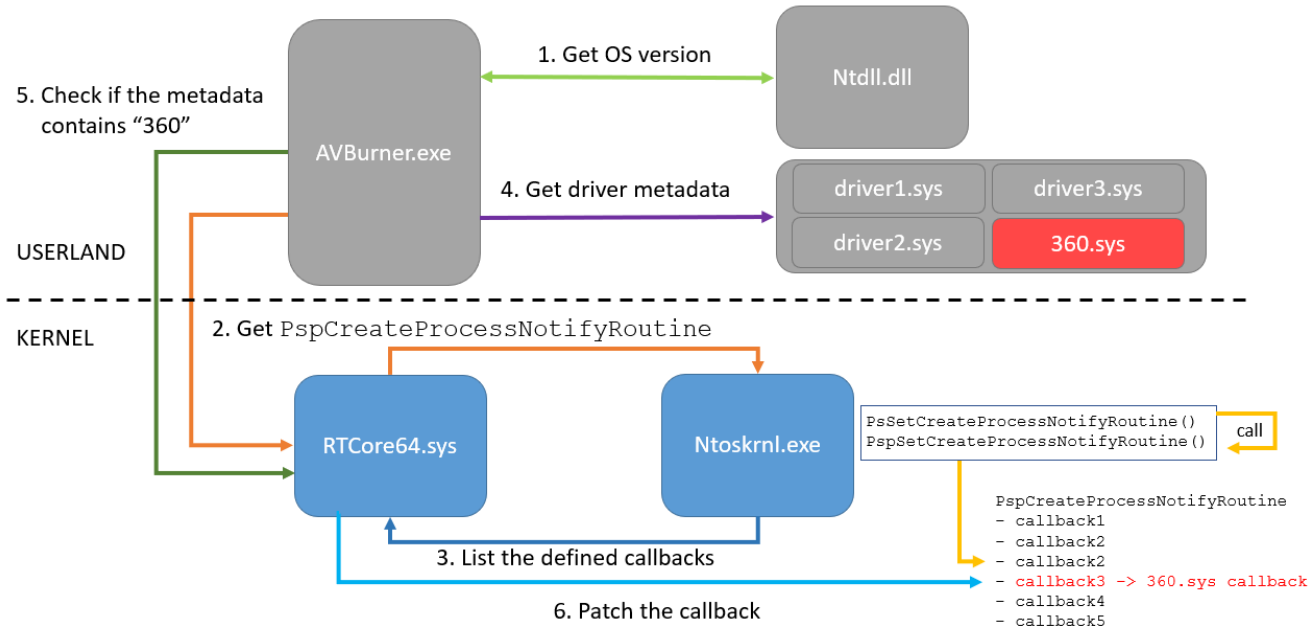



*Figure 5. AVBurner workflow*

Since the Qihoo 360 security product cannot be downloaded and installed for free, for the purposes of demonstration in this report Volexity patched AVBurner to target the free version of Immunet, which embeds Cisco AMP. Note that the approach employed by AVBurner could be used to target any security product using the same APIs, and the use of Cisco AMP in this post is simply for illustrative purposes. Some products may have anti-tampering mechanisms to prevent or detect this style of attack.

Figure 6 shows the callback array before and after AVBurner execution.

```
0: kd> dps nt!PspCreateProcessNotifyRoutine
fffff801`4fb43ba0  ffff8702`82063bcf
fffff801`4fb43ba8  ffff8702`822941bf
fffff801`4fb43bb0  ffff8702`855f1cff
fffff801`4fb43bb8  ffff8702`8560311f
fffff801`4fb43bc0  ffff8702`856e6def
fffff801`4fb43bc8  ffff8702`857f96ff
fffff801`4fb43bd0  ffff8702`84cf69af
fffff801`4fb43bd8  ffff8702`84cfe7bf
fffff801`4fb43be0  ffff8702`845c88cf
fffff801`4fb43be8  ffff8702`857fc89f
fffff801`4fb43bf0  ffff8702`84cfa87f
fffff801`4fb43bf8  ffff8702`8582c44f
fffff801`4fb43c00  ffff8702`863a456f
fffff801`4fb43c08  00000000`00000000
fffff801`4fb43c10  00000000`00000000
fffff801`4fb43c18  00000000`00000000
```

```
0: kd> dps nt!PspCreateProcessNotifyRoutine
fffff801`4fb43ba0  ffff8702`82063bcf
fffff801`4fb43ba8  ffff8702`822941bf
fffff801`4fb43bb0  ffff8702`855f1cff
fffff801`4fb43bb8  ffff8702`8560311f
fffff801`4fb43bc0  ffff8702`856e6def
fffff801`4fb43bc8  ffff8702`857f96ff
fffff801`4fb43bd0  ffff8702`84cf69af
fffff801`4fb43bd8  ffff8702`84cfe7bf
fffff801`4fb43be0  00000000`00000000
fffff801`4fb43be8  00000000`00000000
fffff801`4fb43bf0  ffff8702`84cfa87f
fffff801`4fb43bf8  ffff8702`8582c44f
fffff801`4fb43c00  ffff8702`863a456f
fffff801`4fb43c08  00000000`00000000
fffff801`4fb43c10  00000000`00000000
fffff801`4fb43c18  00000000`00000000
```

*Figure 6. Callback array before AVBurner execution (left) and after (right)*

Two callbacks are removed from the list and the address is replaced by "0x00000000`00000000". Figure 7 shows the values before the AVBurner execution; as expected, these callbacks are related to AMP.

```
0: kd> dps ffff8702`845c88cf & 0xFFFFFFFFFFFFFFF0 L2
ffff8702`845c88c0       00000000`00000020
ffff8702`845c88c8       fffff804`aeb34fd0   CiscoAMPCEFWDriver+0x4fd0
0: kd> dps ffff8702`857fc89f & 0xFFFFFFFFFFFFFFF0 L2
ffff8702`857fc890       00000000`00000020
ffff8702`857fc898       fffff804`aeb55d68   CiscoSAM+0x5d68
```

*Figure 7. Callback before AVBurner execution*

This post is limited to process creation monitoring, but AVBurner also supports manipulation of additional callbacks, such as thread creation and image load callbacks.

## Detecting Disappearing Callbacks with Volatility 3

Volatility 3 natively lists the currently registered callbacks with the windows.callbacks command. Figure 8 shows the command's output before and after the execution of AVBurner; as expected, the two Cisco AMP callbacks disappear.

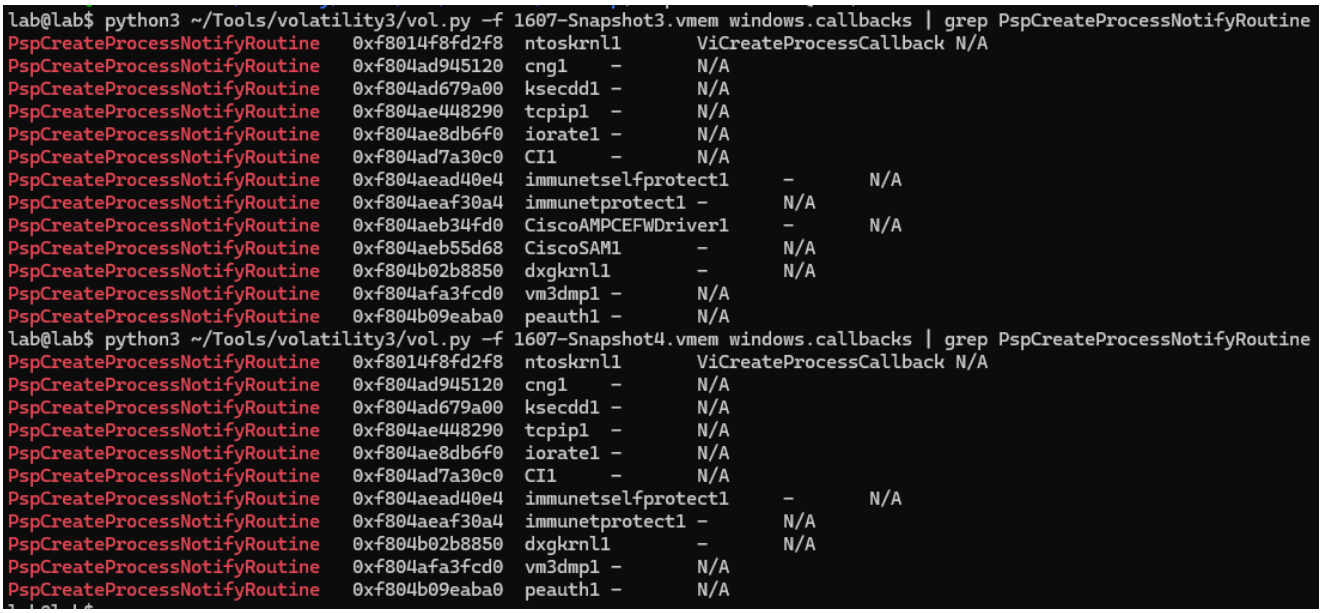



*Figure 8. Volatility 3 windows.callback output*

Two approaches can be used to identify the callback anomaly:

- Get the count of callbacks (see next section for details on how to do this) and compare the value with the windows.callbacks output.
- Maintain a list of EDR or AV modules that register process creation callbacks, and check if the driver is loaded but a callback that should be registered is missing.

## Volshell: Getting the Count of Callbacks

Volatility 3 provides a shell that supports symbols to query the kernel objects. The following code shows how to get the values for PspCreateProcessNotifyRoutineExCount and PspCreateProcessNotifyRoutineCount (Figure 9). Note that color has been added for emphasis.

```
lab@lab$ volshell.py -f 1607-Snapshot4.vmem -w
Volshell (Volatility 3 Framework) 2.4.1
Readline imported successfully  PDB scanning finished

    Call help() to see available functions

    Volshell mode        : Windows
    Current Layer        : layer_name
    Current Symbol Table : symbol_table_name1
    Current Kernel Name  : kernel

(layer_name) >>> kernel = self.context.modules[self.config["kernel"]]
(layer_name) >>> kvo = context.layers[kernel.layer_name].config["kernel_virtual_offset"]
(layer_name) >>> ntkrnlmp = context.module(kernel.symbol_table_name, layer_name=kernel.layer_name, offset=kvo)
(layer_name) >>> hex(ntkrnlmp.get_symbol("PspCreateProcessNotifyRoutineCount").address)
'0x747efc'
(layer_name) >>> hex(ntkrnlmp.get_symbol("PspCreateProcessNotifyRoutineExCount").address)
'0x747ef8'
(layer_name) >>> ntkrnlmp.object(object_type="unsigned int", offset=0x747efc)
7
(layer_name) >>> ntkrnlmp.object(object_type="unsigned int", offset=0x747ef8)
6
```

*Figure 9. Highlighted example command-line output showing the number of callbacks that should exist*

The offsets of the variables, PspCreateProcessNotifyRoutineExCount and PspCreateProcessNotifyRoutineCount, (denoted in purple and gold) match what is shown in Figure 9; the sum is 13. However, there are only 11 defined callbacks shown when listing them using windows.callbacks (Figure 10).

```
lab@lab$ vol.py -f 1607-Snapshot4.vmem windows.callbacks | grep PspCreateProcessNotify | wc -l
11
```

*Figure 10. Highlighted Volatility output showing only 11 callbacks are still defined*

## Detecting Tampered EDR Callbacks with Volexity Volcano

Although these malicious modifications are possible to detect with Volatility, many analysts prefer a more robust, out-of-the-box solution, especially in time-sensitive engagements. Volexity Volcano can help assess if systems are trustworthy, even if AV and EDR products report being healthy and fully operational. Figure 11 shows a summary of some of the IOCs that triggered on the AVBurner memory sample. It points out that two kernel modules, CiscoAMPCEFWDriver.sys and CiscoAMP.sys, are affected by the malware.

| Indicator | Artifact | Reason |
|---|---|---|
| Disabled AV/EDR Callbacks | Kernel Module | CiscoAMPCEFWDriver.sys |
| Disabled AV/EDR Callbacks | Kernel Module | CiscoSAM.sys |
| Disabled Services | Service | wuauserv (Windows Update) - %systemroot%\system32\svchost.exe -k netsvcs |
| Explicit Debug | Privilege | SeDebugPrivilege (windbg.exe pid 5752) |
| Explicit Debug | Privilege | SeDebugPrivilege (x64dbg.exe pid 1340) |
| Explicit Debug | Privilege | SeDebugPrivilege (livekd64.exe pid 6872) |

*Figure 11. Volcano IOC summary includes two Disabled AV/EDR Callbacks*

In addition to the IOC summary, Volcano offers more specific details on the artifacts themselves (including the full path on disk to the kernel module) and an explanation of exactly what functionality has been disabled. Color-coded labels and associated notes are automatically added to the artifacts to bring these alerts to the analyst's attention.

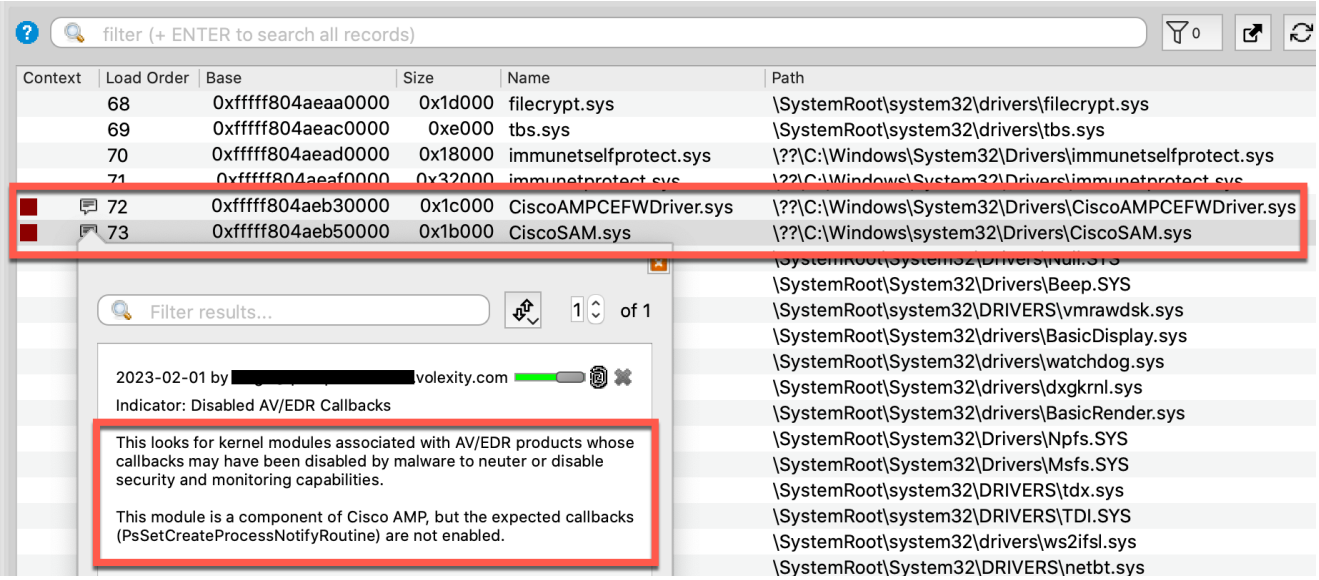



*Figure 12. Zooming in on specific details for CiscoSAM.sys*

Another valuable feature of Volcano that defenders can leverage against the previously described BYOVD technique is listing recently unloaded kernel modules. In most cases, malware will not need a vulnerable driver in the long term. It will load the driver, use it, then unload it quickly to avoid further detection. For debugging purposes, Windows stores unload events in memory, which Volcano can query. Although the vulnerable driver was referred to by its original name, RTCore64.sys, the malware drops it to disk as a.sys. Figure 13 shows how Volcano preserves this activity for forensic purposes and provides an unload timestamp for incorporation into timelines.

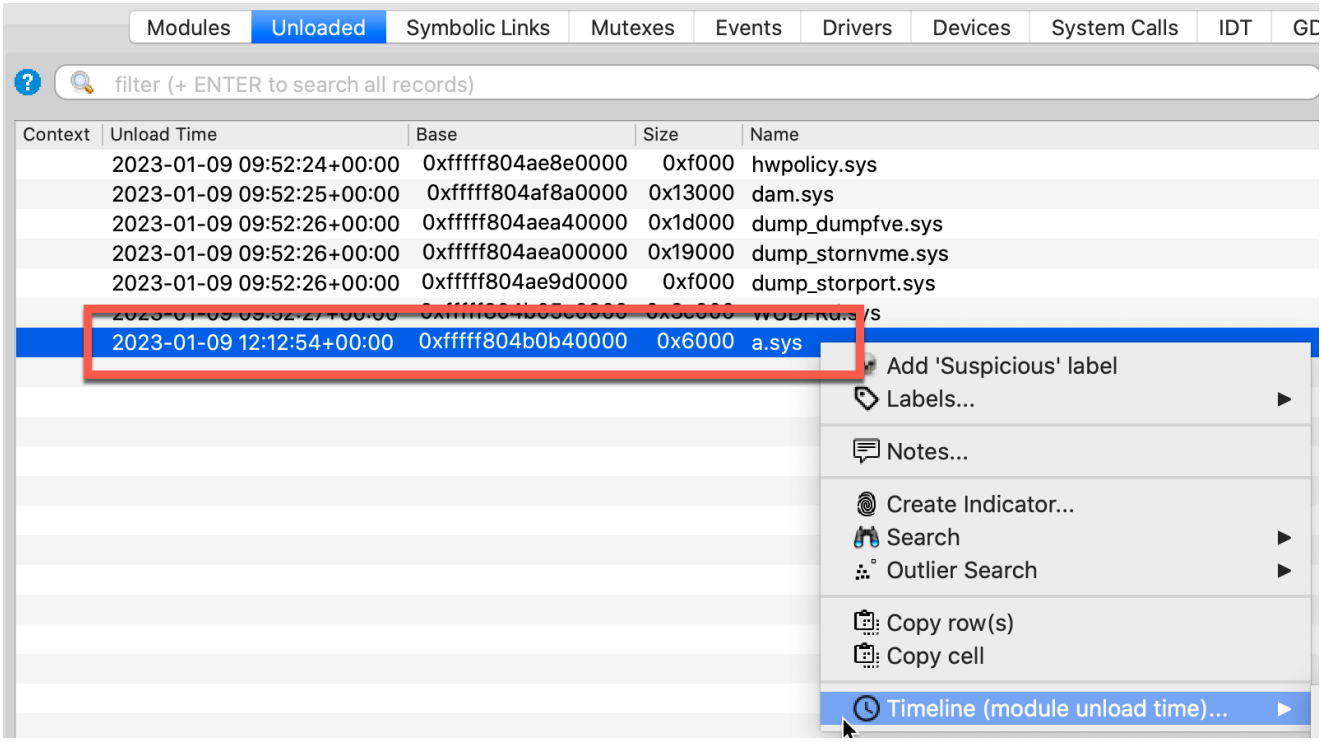



*Figure 13. Volcano showing details of unloaded Kernel modules*

## Conclusion

This blog posts describes how process creation callbacks work on Windows, and how AVBurner can bypass modern EDR and AV solutions by patching kernel memory. In this case, the malware was only used to disable monitoring process creation, but the same approach can be used to disable other callbacks used for events like thread creation and image loading.

Having detection capabilities related to AVBurner is useful. However, it is even better to have robust techniques for broadly detecting whether or not security products have been tampered with in this manner. Callback manipulation can be identified by analyzing the memory of the system, either through automated detections in Volexity Volcano, or through use of Volatility 3 and Volshell.

To generically identify BYOVD related attacks, Volexity recommends the following:

- Monitor for creation of the file RTCore64.sys and the other vulnerable drivers listed in the KDU project, as they are commonly used by threat actors.
- If possible, consider enforcing the mitigations recommended by Microsoft here.

To detect the specific malware referenced in this blog post, Volexity recommends the following:

Use the YARA rule available on GitHub to identify instances of AVBurner.

*Volexity's Threat Intelligence research, such as the content from this blog, is published to customers via its Threat Intelligence service and was covered by in MAR-20230112 and original activity related to this threat actor was discussed in TIB-20211124.*

*Volexity's leading memory analysis product, Volexity Volcano, detects the EDR and AV evasion technique discussed in this post through the "Disabled AV/EDR Callbacks" indicator.*

*If you are interested in learning more about these products and services, please do not hesitate to contact us.*