

Exploiting a vulnerable Minifilter Driver to create a process killer

 antonioparata.blogspot.com/2024/02/exploiting-vulnerable-minifilter-driver.html

Bring Your Own Vulnerable Driver (BYOVD) is a technique that uses a vulnerable driver in order to achieve a specific goal. BYOVD is often used by malware to terminate processes associated with security solutions such as an EDR. There are many examples of open-source software that (ab)use a vulnerable driver for this purpose. One the most used driver is the Process Explorer driver. In this case we cannot talk about a vulnerability since it is a feature of the application to permit process termination from its UI.

BYOVD is gaining more and more attention since attackers understood that it's a better strategy to terminate the EDR process instead than relying on obfuscation techniques in order to evade EDR detection.

In this blog post I'll analyze a signed driver that can be used to create a program able to terminate a specific process from the kernel. The driver is quite old but nevertheless usable. The driver hash is **023d722cbbdd04e3db77de7e6e3cfeabcef21ba5b2f04c3f3a33691801dd45eb** (*probmon.sys*).

Exploiting a Minifilter Signed Driver

The mentioned driver is a signed minifilter driver part of a security solution. One of the imported function is *ZwTerminateProcess*, so my goal is to check if it is possible to call this function on an arbitrary process.

The driver starts by calling the *FltRegisterFilter* function in order to register the filter. Next, a communication port is created by calling *FltCreateCommunicationPort*. The call specifies the parameter *MessageNotifyCallback*, implying that a user mode application can communicate with the minifilter by using the *FilterSendMessage* function. This callback does not expose the access to the *ZwTerminateProcess* function, but it is necessary in order to satisfy the needed preconditions.

After the creation of the communication port, the driver sets a process creation notification function by calling the function *PsSetCreateProcessNotifyRoutine*. The specified callback checks that the third argument of the callback, named *Create*, is false, if not, the function returns immediatly. This implies that only process termination are monitored by the driver. Under specific conditions, the notification callback function will call the *ZwTerminateProcess* function.

In order to terminate a process with the vulnerable driver, there are two preconditions that must be satisfied:

1. The handle of the process to terminate is read from a global variable. We have to set this variable, otherwise when the driver tries to terminate a process a *KeBugCheckEx* will be called generating a BSOD
2. The *ZwTerminateProcess* is called only if the value of the process ID calling into the minifilter is the same of the one associated with a global variable.

Set the target process handle

This requirement is satisfied by sending a message to the communication port by using the struct from Figure 1.

```
#[repr(C)]
struct CommandSetPidToTerminate {
    command_type: u32,
    pid_to_kill: u32
}
```

Figure 1. Set Target Process Handle Message Structure

In this case the *command_type* parameter must assume value 3. This will cause the *ZwOpenProcess* to be called by using the *pid_to_kill* parameter, and the result assigned to the above mentioned global variable (let's call it *process_handle_to_terminate*).

Enable process termination

The second precondition involves a check on a global variable (let's call it *it_s_a_me*, you will understand why I choose this name in a moment). The value of this variable must be the same of the process ID that is exiting (remember that the callback is monitoring for process termination). This check is performed in the *PsSetCreateProcessNotifyRoutine* notification callback function. As before, this can be achieved by using the struct from Figure 2.

```
#[repr(C)]
struct CommandEnableTermination {
    command_type: u32,
    data_count: u32,
    my_pid: u32
}
```

Figure 2. Set Global Variable To Enable Process Termination

In this case the *command_type* parameter must assume value 1. The *data_count* is used to copy the data that follow this parameter. In our case it is ok to set 1 as value (1 DWORD is copied) and set as value of the field *my_pid* our PID. In this way, our PID is written to the *it_s_a_me* global variable, satisfied our second precondition.

Triggering process termination

At this point we have set the handle of the process to terminate (variable *process_handle_to_terminate*) and we can reach the *ZwTerminateProcess* function thanks to the variable *it_s_a_me*.

When our process will exit, the *PsSetCreateProcessNotifyRoutine* notification callback will be called, the PID check will be satisfied by verifying that the variable *it_s_a_me* is equals to the process ID that is exiting, triggering the *ZwTerminateProcess* on the *process_handle_to_terminate* process. All this means that when our process killer program will exit, the target process will be killed :)

Source Code

Considering the plethora of such programs available on Github, releasing one more shouldn't be a huge problem. You can find the source code using the analyzed driver in my Github account:

<https://github.com/enkomio/s4killer>

Be consciuos that the driver is registered by using the flag `FLTFL_REGISTRATION_DO_NOT_SUPPORT_SERVICE_STOP` implying that the minifilter is not unloaded in response to service stop requests. In addition, the code `STATUS_FLT_DO_NOT_DETACH` is returned when you try to unload the driver with *fltmc*. In order to unload the driver you have to reboot your machine.

Conclusion

The goal of this blog post was to demonstrate how the malware use BYOVD technique in order to kill EDR processes. I analyzed a previously unknow vulnerable driver (to the best of my knowledge of course) demonstrating how a minifilter can also be abused for such purpose.

Bonus

I'm currently focused on BYOVD technique used by malware to kill processes, so I haven't searched for more vulnerabilities in the driver. However, there is a nice buffer overflow in it but I'm unsure if it is exploitable or not :)