



« Back to home

# Hiding your .NET - COMPlus\_ETWEnabled

Posted on 2020-06-06 Tagged in [reversing](#), [redteam](#), [etw](#)

The process of disabling ETW is something that I first looked at back in March after trying to figure out just how some defenders were detecting in-memory Assembly loads (<https://blog.xpnsec.com/hiding-your-dotnet-etw/>). There have since been several other posts with clever and improved methods of bypassing this kind of detection from some awesome researchers including [Cneeliz](#), [BatSec](#) and [modexp](#). Each method relies on manipulating the ETW subsystem itself, from intercepting and manipulating calls to the usermode function **EtwEventWrite** or the kernel function **NtTraceEvent**, and even parsing and manipulating the ETW registration table to avoid any code patching.

It turns out however that there is also a further method of disabling ETW in .NET, strangely exposed by setting an environment variable of **COMPlus\_ETWEnabled=0**:

Want to stop ETW from giving up your loaded .NET assemblies to that pesky EDR, but can't be bothered patching memory? Just pass `COMPlus_ETWEnabled=0` as an environment variable during your `CreateProcess` call 😊 [pic.twitter.com/wXWeSdt0li](https://pic.twitter.com/wXWeSdt0li)

— Adam Chester (@xpn) June 5, 2020

Now since posting this method I have been asked quite a few questions, mostly focusing on how this particular setting was found as well as how it works. So in this short post I wanted to cover some of those details for anyone interested in peeking under the hood.

Before we begin to look at this however it is worth commenting on the fact that ETW was never meant to serve as a security control, which helps to explain some of the things shown in this post. Its primary purpose is as a debugging tool, but as attackers have evolved their payload execution techniques, it seems that some defenders turned to the power of ETW as a way of surfacing information on events like .NET Assembly loads. It is because of ETW's original purpose that we come across things like **COMPlus\_ETWEnabled** which might seem like a toggle for security auditing... when in fact it's just a simple way to turn off some debugging functionality.

## So what is this COMPlus\_ prefix thing?

**COMPlus\_** prefixed settings provide developers with a number of configuration options which can be set at runtime with various levels of impact on the CLR, from loading alternative JITters, to tweaking performance and even dumping the IL of a method. Being provided via environment variables or registry values, many of these settings are undocumented, with one of the best resources for understanding each being the CoreCLR source, specifically [clrconfigvalues.h](https://github.com/dotnet/coreclr/blob/master/src/config/configvalues.h).

Now if you take a quick look at this file you might notice that **COMPlus\_ETWEnabled** isn't present. It actually turns out that this was removed from the CoreCLR in an earlier commit [here](#) on 31 May 2019.

As with many undocumented features, these settings provide some interesting functionality for us attackers. It's worth pointing out however that unsurprisingly these settings don't always follow the **COMPlus\_** prefix naming as we can see from [this](#) FullDisclosure post from 2007 which used a setting named **COR\_PROFILER** to achieve a UAC bypass in MMC.exe.




So now that we know just why these settings exist and how to list them, let's look at how this particular one was found.

## Finding COMPlus\_ETWEnabled

Although there are a lot of settings shown in the CoreCLR source, many do not apply to the standard .NET Framework we're all familiar with.

To determine which **COMPlus\_** settings apply to the .NET Framework, we can simply hunt within **clr.dll** for any references, such as the **COMPlus\_AltJit** setting documented in **clrconfigvalues.h**.

Removing the prefix and performing a simple string search in IDA is enough to give us an indication that **AltJit** is likely present within **clr.dll**:

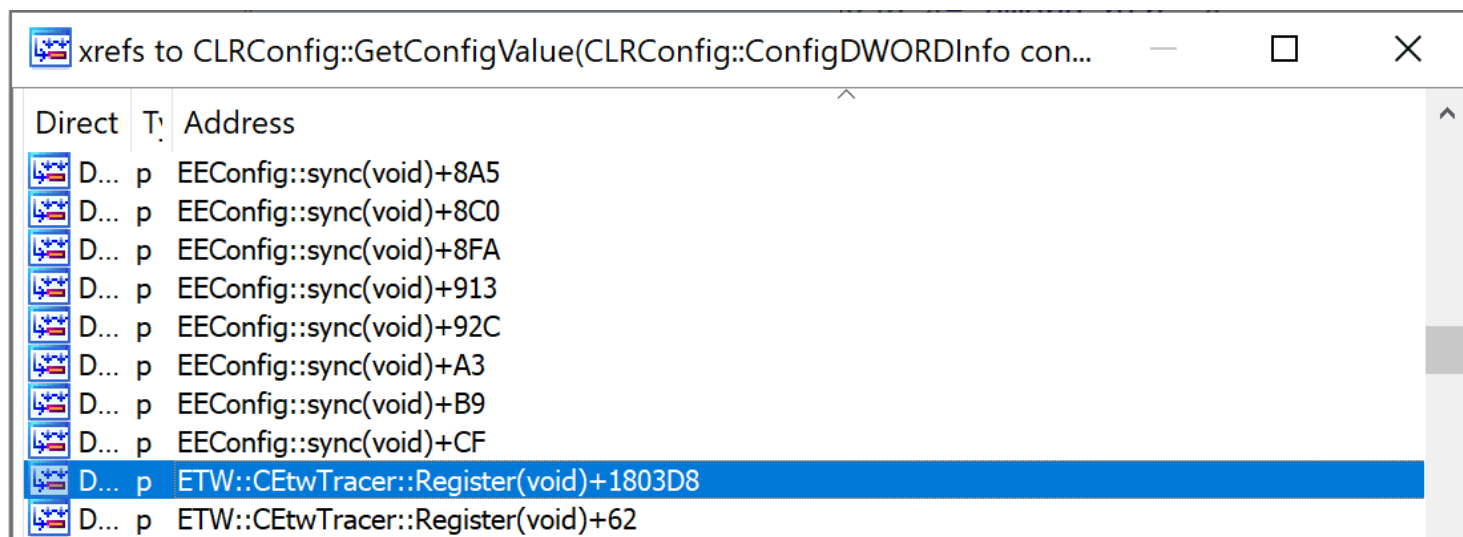
Address	Length	Type	String
 .text:1016D7... 0000000E	0000000E	C (16 bits) - UTF-16LE	AltJit
 .text:101EE5... 00000016	00000016	C (16 bits) - UTF-16LE	AltJitName
 .text:101F8F48 00000016	00000016	C (16 bits) - UTF-16LE	AltJitNgen

Following references to this string leads us to a method of **CLRConfig::GetConfigValue** which is passed our setting name as a parameter to retrieve the value:

```
loc_1016D687:
mov     ecx, esi
call    ?SetCpuInfo@EEJitManager@@AAEXXZ ; EEJitManager::SetCpuInfo(void)
mov     [esi+6Ch], edi
lea     edx, [ebp+var_10]
mov     ecx, offset ?EXTERNAL_JitName@CLRConfig@@2UConfigStringInfo@1@B ; COMPlus_JitName
mov     [ebp+var_18], edi
mov     ?g_JitLoadData@@3UJIT_LOAD_DATA@@A, 1F4h ; JIT_LOAD_DATA g_JitLoadData
mov     [ebp+var_10], edi
call    ?GetConfigValue@CLRConfig@@SGJABUConfigStringInfo@1@PAPAG@Z ; CLRConfig::GetConfigValue(CLRConfig::ConfigStringInfo const &,ushort * *)
test    eax, eax
js      loc_102E6C09
```

Taking this and searching for overloads gives us several other methods which are also used to access similar configuration settings during runtime:

And as Microsoft provide the appropriate PDB file for the CLR, walking through each of these methods and taking a look at the Xrefs is enough to indicate which references are likely to be interesting, for example:



Finally, following the reference and looking at the arguments passed to `CLRConfig::GetConfigValue` is enough to give you the setting used:

## What does COMPlus\_ETWEnabled do exactly?

Now that we know that this setting exists in the .NET Framework, how does this disable event tracing? Well let's look at a CFG in IDA which should show you immediately why this works to disable ETW:



Here we can clearly see the 2 code paths which depend on the **COMPlus\_ETWEnabled** value returned from **CLRConfig::GetConfigValue**. If this setting exists and returns a **0** value, the CLR will jump past the block of ETW registrations shown in blue, where **\_McGenEventRegister** is simply a wrapper around the **EventRegister** API call.

Digging further by taking one of these provider GUID's, we see something familiar:

This is of course referencing the GUID **{e13c0d23-ccbc-4e12-931b-d9cc2eee27e4}** which we used in our [previous post](#) when unhooking ETW and is [documented](#) by Microsoft as the CLR ETW provider:

So hopefully this sheds some light on this strange but cool setting... Essentially, toggling it just forces the CLR to skip past the point of registering to the .NET ETW providers which is required to surface events.

Now for defending against this, I'll defer to [Roberto Rodriguez's](#) awesome set of notes which detail a number of detections and mitigations which can be used to detect and protect an environment... check them out [here](#).

**Update 07/06/2020** - I have created a quick environment variable spoofing POC which uses a similar practice to [Argument Spoofing](#) to mask environment variables passed to a process on launch. This can be found [here](#) and is designed to hide the **COMPlus\_ETWEnabled=0** variable from **CreateProcess**: