GlobeTech LLC

Evading EDR by DLL Sideloading in C#

Leave a Comment / By gares / May 19, 2023

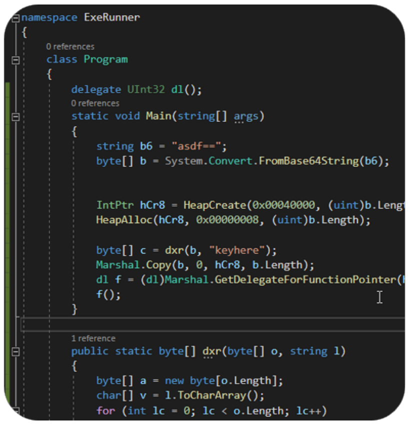
The blog post accompanies my public conference talks on this topic. Slides are available at the bottom.

One thing I learned from doing this talk is that a surprisingly high number of InfoSec professionals believe that bypassing EDR is not possible or is an exception. It should be more common knowledge that security tools can be bypassed. Regardless of what EDR you run, there are ways to get around it. This is one technique to evade those defenses by running malware within a more trustworthy process.

Instead of believing that your security tools will prevent all malware, you should know that eventually it can be bypassed, so it's better to understand some of those techniques, and how the malware operates.

What do I mean Bypass EDR?

Well if I use some sort of basic shellcode runner, like this as a starting point.
https://github.com/nettitude/PoshC2/blob/master/resources/payload-templates/csc.cs



```
namespace ExeRunner
{
    0 references
    class Program
    {
        delegate UInt32 dl();
        0 references
        static void Main(string[] args)
        {
            string b6 = "asdf==";
            byte[] b = System.Convert.FromBase64String(b6);


            IntPtr hCr8 = HeapCreate(0x00040000, (uint)b.Lengt
            HeapAlloc(hCr8, 0x00000008, (uint)b.Length);

            byte[] c = dxr(b, "keyhere");
            Marshal.Copy(b, 0, hCr8, b.Length);
            dl f = (dl)Marshal.GetDelegateForFunctionPointer(
            f();
        }

        1 reference
        public static byte[] dxr(byte[] o, string l)
        {
            byte[] a = new byte[o.Length];
            char[] v = l.ToCharArray();
            for (int lc = 0; lc < o.Length; lc++)
```

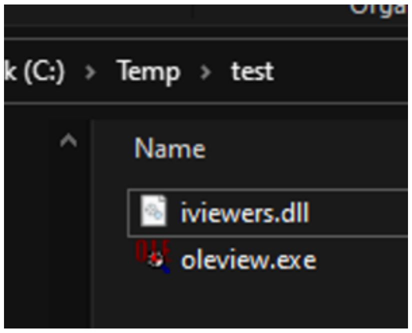
…and I compile that to an EXE, there are EDR tools that instantly look at those binaries (and any similar custom shellcode runners) and determine they are suspicious. With embedded shellcode, it's high entropy, it's a globally unique EXE file, it immediately makes network connections outbound. EDR looks at this combination and just shuts down the process.

But if I take the same style of code and compile it as a DLL, EDR gives it much less scrutiny. Static analysis scans against the DLL are much more likely to consider the file safe and pass the scan.

This means, we don't have to rewrite our shellcode runners, we just need to compile them as a DLL. Then we just need a way to load our malicious DLLs.

What is DLL Sideloading?

I feel like this used to mean something more specific. These days it seems that it is a catch all that **refers to an EXE that will load a DLL sitting alongside it**.



There are multiple ways that this can happen, like a manifest file specifying the dll location (historically, this is what I understood DLL sideloading to mean). You may also have source code with a LoadLibrary call that doesn't use an absolute path, and therefore searches for DLLs based on search order. At the end of the day, we don't care anymore (apparently).

What are we going to do?

We're going to find a Microsoft signed EXE that will 'sideload' DLLs. Then we'll place our DLL in the same directory to get execution through the DigSig binary.

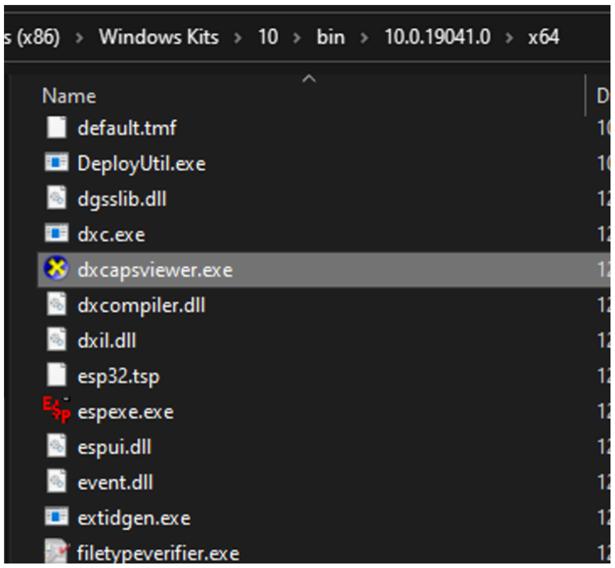
Check out these references about DLL Sideloading. **Hijack Libs** is basically what inspired me to start writing my malware this way, and let to this talk / post.

Mandiant, hijacklibs.net, CyberReason

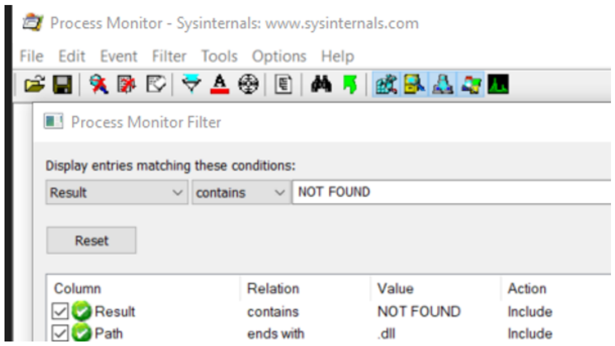
Finding a vulnerable EXE

To start, I get some signed binaries. I like the Microsoft SDK for signed EXEs. Particularly because they don't typically exist on disk by default (which is huge for certain EDR detection capabilities), but they commonly NEED DLLs alongside them to function. The only way for that to happen is for both to be brought down to disk in a random directory and executed together. Since they're signed by MS, they tend to be inherently more trusted.

Now that we have some EXEs, I run the EXE while watching with the SysInternals tool Procmon. Process Monitor will let you see what actions a process is taking. Using this, we can check for the process looking for a .DLL file where the result is NOT FOUND. Or sometimes I'll run the exe and just monitor the directory that I'm running it from so I can see all the actions that it takes. In either case, I create a new empty directory, and place the EXE there with nothing else in that directory. For example, monitor C:\Temp\exes\ location only, and now any exe you run from that directory, you'll get to see the actions it takes.


s (x86) › Windows Kits › 10 › bin › 10.0.19041.0 › x64

Name
default.tmf
DeployUtil.exe
dgsslib.dll
dxc.exe
dxcapsviewer.exe
dxcompiler.dll
dxil.dll
esp32.tsp
espexe.exe
espui.dll
event.dll
extidgen.exe
filetypeverifier.exe


Process Monitor - Sysinternals: www.sysinternals.com

Process Monitor Filter

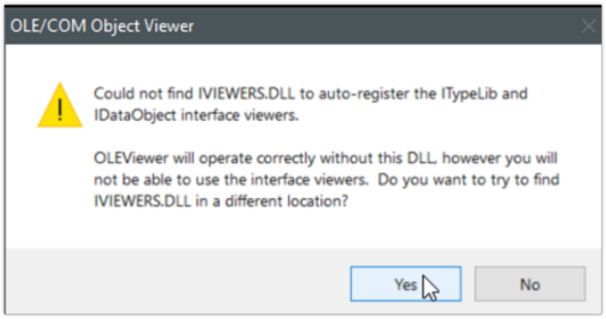
Display entries matching these conditions:
Result | contains | NOT FOUND

Column	Relation	Value	Action
✓ Result	contains	NOT FOUND	Include
✓ Path	ends with	.dll	Include

Testing with OleView

We're going to copy oleview into C:\temp\exes. Oleview is a good example tool for a couple of reasons. One, it isn't normally on disk somewhere, and that is actually a pretty good detection point. Some EDRs are smart enough to witness when an EXE already lives in System32. Since a normal user has a PATH entry that would find EXEs in System32, some EDRs will outright block those exes from running outside of that directory since it's likely they're just being abused for DLL sideloading.

If we run oleview from our newly created directory and monitor with proc mon we can see a series of DLLs that it checks for, but the "Result" is NOT FOUND. This is our big tip off. These are the DLLs we should review.


OLE/COM Object Viewer

Could not find IVIEWERS.DLL to auto-register the ITypeLib and IDataObject interface viewers.

OLEViewer will operate correctly without this DLL, however you will not be able to use the interface viewers. Do you want to try to find IVIEWERS.DLL in a different location?

Oleview gives us an error message though, a very verbose message that calls out the fact that it couldn't find iviewers.dll. Even if you didn't run Procmon, you'd already be able to guess that oleview probably sideloads iviewers.dll. Most EXEs aren't going to be so obvious, so procmon is the way to go, and we can see iviewers.dll in the results from procmon as well.

Inspecting the original DLL

Back in the SDK folder we can find the original iviewers.dll and we can use this to look at the functions that are normally available through this DLL.

DLLs export certain functionality to a program. Our exe expects this functionality to be present when it loads the dll. If oleview loads a dll and the exported functionality seems to be missing, it will immediately error out and never actually load or execute the DLL. We can explore the "Export Address Table" to identify what these functions are, so we can understand what functionality a DLL normally offers for programs to utilize.

Using tools like CFF Explorer, PE-Bear, or objdump on the *nix command line, we can have the DLL parsed so we can explore it. The PE structure is parsed and displayed but that can be intimidating to look at if you've never seen it before. All we really care about this time is the "Exports" which will show us what functions exist inside of that DLL. These are the functions that an application can utilize when it loads the DLL.

Writing our own DLL

Often PoCs are written in C /C++. I prefer C#, but this preference causes some problems for us. To understand this, we need to understand the difference between managed and unmanaged code.

When we write C or C++, we have to compile it for the machine to understand our code. Compiling these languages will result in machine code, or native code. This is code that the CPU understands and can execute. Machine code, native code, unmanaged code

Difference between machine code and byte code. copmiling down to native code, vs compiling down to byte code, and then needing a CLR for the rest.

Methods written in C# won't exist in an export address table. c# .NET assemblies don't need that same adverstisement to function. the result is oleview won't be able to use our DLL. It will load it, look for the EAT, and not find any functions. It will assume the DLL doesn't have those functions, and fail to call the DLL at all.

C# DLL- No Exports and Manually Exporting Methods

To review this, and prove ourselves correct, we can create a C# based DLL and create methods inside it. Even though our method names might be the same as, once we compile it and open the DLL with something like CFF explorer, we'll see that there is no "Export Directory". So a standard C# .NET Assembly doesn't export functions the same way unmanaged code does. You'll notice that there is a .NET Directory, so it recognizes that we're looking at a different structure here.

But the C# DLL is compiled down to byte-code, so we can use an Intermediate Language disassembler to ‘reverse’ the compiled code back to IL. This will show us a close resemblance of the source code. If we review the IL, we can find our methods. Inside the method, we have the ability to add a ‘.export’ descriptor. This one change to our IL will publish the method as a exported function. With this change, we need to re-assemble the IL as a DLL again, and we can once again review the DLL in something like CFF explorer.

This time, we'll see that it's still a .NET assembly of course, but now we have an Export Directory, with the function name properly exported. This is enough for an application to believe that the functionality is present, it won't do any checks to see that the code does what it expects.

C# DLL- DLLExport Nuget Package

While we certainly could do this manually, we don't have to. There is more than one way to make this easier for you, but I like the Nuget package DLLExport. If you install that package to your project, you can add the '[DLLExport]' decorator to your C# method and the methods will be properly exported to the EAT.

With the DLLExport Nuget package installed, we can compile the new DLL with the methods exported. The newly compiled DLL can be renamed to iviewers.dll and now we'll be able to load it into oleview if we keep them in the same directory. The execution shows our msfvenom Message Box shellcode being loaded inside of oleview.

Oleview PoC

The moment of truth, if we take our newly compiled DLL and rename it to iviewers.dll, then we can copy it into the same directory as oleview. This should be enough to get our execution. Oleview will launch, look for iviewers.dll, find our dll, check that the exports it expects are actually present, and finally execute.

Why would this work

So we created an export… why does this work? We're still loading oleview, which has no idea that bytecode is a thing. In fact, if we look at oleview running, we can see the clr.dll isn't even loaded, so it doesn't have any way to process managed code like our C# DLL.

The thing is, when the OS loads the DLL, it recognizes that structure as byte-code, and already knows that in order to process it, we'll need the CLR. As a result clr.dll will automatically be loaded into the process and our C# dll will execute as expected.

Cool, but I don't write code

Yeah, you don't really have to. There are great tools out there, and when it comes to DLL sideloading, I've really liked the Freeze project by Optiv. The project lets you specify an export, and some shellcode, and it will create a DLL with that export, which encrypting your shellcode inside the DLL.
I also really like the pull request on the project right now, which allows you to specify multiple exports, or specify a source DLL. In that case, you can specify the original DLL, and your shellcode from your favorite C2 server. The result is a DLL that you can use for DLL sideloading.

Didn't this just break Oleview?

Absolutely, but in our case, we may just want a C2 beacon executed. We're only doing it with oleview to gain extra trust from our security tools by executing within a legitimate tool.

We could, however, keep application functionality. In that case, we're looking for a DLL proxy. While this is a different technique, it shares a lot of the same concepts. An application can call our DLL expecting certain functionality, and the proxy DLL will be able to run our shellcode, but then pass along the requests to the ACTUAL original DLL, which maintains the original applications requests and functionality.

I like using the project SharpDLLProxy for this, this tool take shellcode and source DLL, but the output is actually just source code for a proxy DLL.

Blue Team

I won't pretend to be a wise blue teamer, but there are some possible detection options and threads to pull on.

One of the first detection points I noticed is that some EDRs will detect and outright block on DLL sideloading of binaries would normally have been found in System32. This makes sense to me, since every user has the PATH environment variable defined. If we copy a utility from System32 out to our own directory, and DLL sideload with it, those EDRs can basically assume that normal operation would have let the user resolve these binaries to the already-on-disk files inside of System32, so if we're calling something else, it's likely a DLL sideloading attempt.

That's why I like to bring in MS Signed files that aren't already on disk. The expectation for those files is that they will be written to any random directory. Not quite as obvious, but even the DLL will make a difference. For example, if we use oleview, but we opt to choose aclui.dll for the sideload… that might be a little suspicious to some EDRs, since aclui.dll already lives in System32.

To test your EDR, you could copy something like relpost.exe from System32, and build your own reagent.dll (examples from hijacklib) and see your EDRs reaction to understand what it is capable of.

You might also consider looking at signed binaries that are loading DLLs that aren't signed. For example, the original iviewers.dll is also a Microsoft signed file. Our version of iviewers is obviously not signed by Microsoft.

When it comes to C# tradecraft, you might want to watch for clr.dll being loaded into a process, especially if you can understand where that is normal and when it isn't. This is useful beyond just our PoC, but for any beacon that loads C# based malware. For example, a beacon process that loads something like seatbelt or rubues, would have the side effect of image loading clr.dll .

If you can, baseline what normal looks like for your exes running in 'non-standard' directories. Potentially more difficult, but attempt to do the same for your DLLs based on image load events. I've also heard there may be opportunity in identifying newly written DLLs that are immediately loaded into another process. Pairing down all of this information has resulted in decent success in teams I've seen.

Presentation Slides

DLL-Sideloading

Download

← Previous Post

Next Post →

Leave a Comment

Your email address will not be published. Required fields are marked *

Type here..

Name*

Email*

Website

☐ Save my name, email, and website in this browser for the next time I comment.

Post Comment »