In Process Execute Assembly and Mail Slots

teamhydra.blog/2020/10/12/in-process-execute-assembly-and-mail-slots

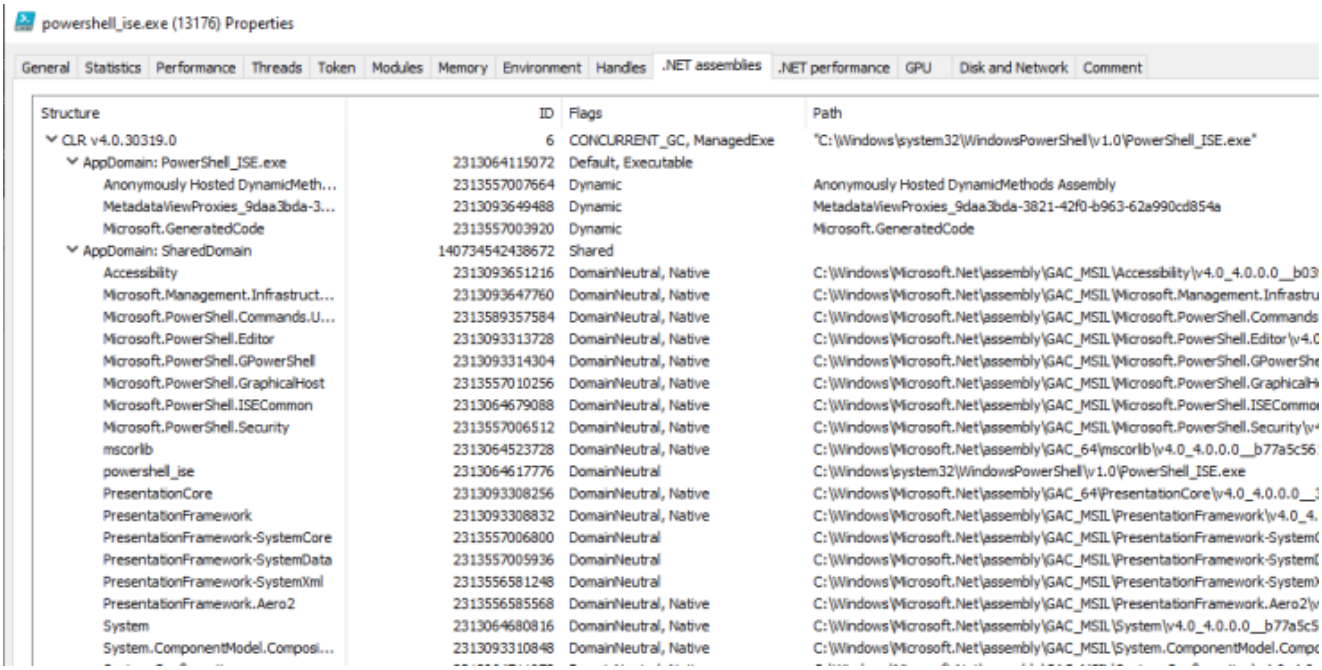
By N4kedTurtle October 12, 2020

While working on our team's internal implant I wanted to implement the ability to execute .Net assemblies in memory. However, by far the most common way of doing this is spawning a new process, executing the .Net assembly inside that process, and sending the response over a pipe to the launching process. This is the way Cobalt Strike introduced in 2018 and does provide a lot of flexibility. However, creating a new process feels expensive and I wanted the option to execute the assembly from within my own process. I also wanted to explore other avenues of writing and capturing the output from the assembly while still remaining in memory. This post and the included PoC are the result of me prototyping how I wanted to go about accomplishing these tasks.

LOADING A CLR

> "The .NET Framework provides a run-time environment called the common language runtime, which runs the code and provides services that make the development process easier."
>
> *https://docs.microsoft.com/en-us/dotnet/standard/clr*

The Common Language Runtime is hosted within a native process and is where .Net assemblies are loaded and run. Honestly, I'm not going to go that deeply in this post on what each of these concepts are because it would be very long and frankly I would probably get it wrong . I will provide MSDN links at each applicable stage and code so you can explore for yourself. If you open up PowerShell and then use a tool like Process Hacker you can see the loaded CLR, the app domains, and assemblies that are loaded within.



A CLR is not loaded by default into a process so if we want to execute a .Net assembly within our process the first thing we need to do is load a CLR.

```
HRESULT hr;
ICLRMetaHost* pMetaHost = NULL;
ICLRRuntimeInfo* pRuntimeInfo = NULL;
BOOL bLoadable;
hr = CLRCreateInstance(CLSID_CLRMetaHost, IID_ICLRMetaHost,
(LPVOID*)&pMetaHost);
hr = pMetaHost->GetRuntime(L"v4.0.30319", IID_ICLRRuntimeInfo,
(LPVOID*)&pRuntimeInfo);
hr = pRuntimeInfo->IsLoadable(&bLoadable);
hr = pRuntimeInfo->GetInterface(CLSID_CorRuntimeHost, IID_ICorRuntimeHost,
(LPVOID*)&g_Runtime);
hr = g_Runtime->Start();
```

Now we have loaded a CLR into our process. We need an application domain into which our assembly will be loaded.

APPLICATION DOMAINS

> Application domains provide an isolation boundary for security, reliability, and versioning, and for unloading assemblies. Application domains are typically created by runtime hosts, which are responsible for bootstrapping the common language runtime before an application is run.
>
> *https://docs.microsoft.com/en-us/dotnet/framework/app-domains/application-domains*

Application domains are a bit like a process within a process. They can have their own threads, work similarly to processes in terms of isolation, and each can run with its own security level. We are just going to use the default application domain for this blog as creating your own app domain take a bit more code and explanation.

```
IUnknownPtr pUnk = NULL;
_AppDomainPtr pAppDomain = NULL;
hr = g_Runtime->GetDefaultDomain(&pUnk);
hr = pUnk->QueryInterface(IID_PPV_ARGS(&pAppDomain));
```

Now we have created and started the CLR and have a pointer to the default app domain interface.

LOADING THE ASSEMBLY

Now that we have our app domain we can load the assembly.

```
bounds[0].cElements = (ULONG)assembly.size();
bounds[0].lLbound = 0;
psaBytes = SafeArrayCreate(VT_UI1, 1, bounds);
SafeArrayLock(psaBytes);
memcpy(psaBytes->pvData, assembly.data(), assembly.size());
SafeArrayUnlock(psaBytes);
hr = pAppDomain->Load_3(psaBytes, &pAssembly);
```

EXECUTING THE ASSEMBLY

Finally, we are able to execute the assembly! This is very easy to do if you want to execute a specific exported function from a dll. However, we want to be able to execute common .Net offensive testing tools like Rubeus, Seatbelt, etc. which are commonly used as exes so we need to do a bit extra. (Credit to https://github.com/b4rtik/metasploit-execute-assembly/blob/master/HostingCLR_inject/HostingCLR/HostingCLR.cpp for some of this).

```
hr = pAssembly->get_EntryPoint(&pEntryPt);
if (args.empty())
{
vtPsa.parray = SafeArrayCreateVector(VT_BSTR, 0, 0);
}
else
{
w_ByteStr = (wchar_t*)malloc((sizeof(wchar_t) * args.size() + 1));
mbstowcs(w_ByteStr, (char*)args.data(), args.size() + 1);
szArglist = CommandLineToArgvW(w_ByteStr, &nArgs);
vtPsa.parray = SafeArrayCreateVector(VT_BSTR, 0, nArgs);
for (long i = 0; i < nArgs; i++)
{
BSTR strParam1 = SysAllocString(szArglist[i]);
SafeArrayPutElement(vtPsa.parray, &i, strParam1);
}
}
psaArguments = SafeArrayCreateVector(VT_VARIANT, 0, 1);
hr = SafeArrayPutElement(psaArguments, &rgIndices, &vtPsa);
hr = pEntryPt->Invoke_3(vtEmpty, psaArguments, &vReturnVal);
```

Now, if all goes well….

We successfully executed the assembly inside our process! Great! Only there is one big problem. All of that output to the console, which is generally not what we want as offensive security testers. Now, you could go through and modify every assembly you are going to use to make sure they use a string and get the response from the previous code. However, that is boring, so let's try something different.

MAILSLOTS

Mailslots! I have been trying to figure out a decent way to use these for a while. They seem so useful but they are very restrictive and there tends to be a better way to accomplish whatever I have been doing.

> A mailslot is a pseudofile that resides in memory, and you use standard file functions to access it. The data in a mailslot message can be in any form, but cannot be larger than 424 bytes when sent between computers. Unlike disk files, mailslots are temporary. When all handles to a mailslot are closed, the mailslot and all the data it contains are deleted.
>
> *https://docs.microsoft.com/en-us/windows/win32/ipc/about-mailslots*

To highlight the important stuff, mailslots are ***in memory***, can be ***read over the network***, and ***all the data is deleted*** when all handles are closed . Also, you can broadcast a message across the domain to all processes with the same mailslot name. BUT, you are restricted to 424 bytes (400 bytes across domain). Yeah, that is a big but that not even Sir Mix-a-lot likes.

Mailslots are really easy to implement for in process / interprocess comms. So easy that the examples on MSDN can be copy / pasted into code at will.

Now all that is left to do is redirect the output from stdout / stderr into our shiny new mailslot and we will be done.

```
g_OrigninalStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
g_OrigninalStdErr = GetStdHandle(STD_ERROR_HANDLE);
HANDLE hFile = CreateFileA(SlotName, GENERIC_WRITE, FILE_SHARE_READ,
(LPSECURITY_ATTRIBUTES)NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
(HANDLE)NULL);
SetStdHandle(STD_OUTPUT_HANDLE, hFile);
SetStdHandle(STD_ERROR_HANDLE, hFile);
SetStdHandle(STD_OUTPUT_HANDLE, g_OrigninalStdOut);
SetStdHandle(STD_ERROR_HANDLE, g_OrigninalStdErr);
ReadSlot(outputString);
printf("Output from string = %s", outputString.c_str());
```

And finally.

POC || GTFO

This code has everything from above but laid out better and with more safety checks. POC

OPERATIONAL THOUGHTS

So, how useful is all this? Obviously, execute assembly has been hugely impactful on post-PowerShell-gets-detected-by-everything operations. The ability to do it in process is especially useful if you are writing your own tooling and it is nice to have the option of not creating a new process. Standard tradecraft still applies (bypass amsi, etw, etc) and you need to be aware that you are often not running in a process that normally loads a CLR.

Mailslots? I'm not sure. In this case they are useful since it avoids more common ways of doing this kind of thing like named pipes and the size restrictions don't apply when using them in process. I think it could be useful to use something like this to run .Net assemblies (Seatbelt) on a target host and receive the output on your current host. I also think there is possibly space for asymmetric comms channels (commands over mailslot, responses over a more robust channel) or as a way to trigger persistence in certain situations. I would love to know if others are using these for anything.

DEFENSIVE CONSIDERATIONS

A fair bit has been written about detecting execute assembly / .Net offensive tools. This is still one where I feel like the adversaries have the advantage, but visibility is improving constantly. Some resources:
https://blog.f-secure.com/detecting-malicious-use-of-net-part-1/
https://blog.f-secure.com/detecting-malicious-use-of-net-part-2/
https://redcanary.com/blog/detecting-attacks-leveraging-the-net-framework/
https://www.mdsec.co.uk/2020/06/detecting-and-advancing-in-memory-net-tradecraft/