Unmanaged .NET Patching

outflank.nl/blog/2024/02/01/unmanaged-dotnet-patching

Kyle Avery

February 1, 2024

To execute .NET post-exploitation tools safely, operators may want to modify certain managed functions. For example, some C# tools use the .NET standard library to terminate their process after execution. This may not be an issue for fork&run implementations that spawn a sacrificial process, but executing in-process will terminate an implant. One could write a small .NET program that resolves and patches these functions, but we were interested in an unmanaged approach (i.e. a unmanaged implant executing managed code in-process). While our example targets `System.Environment.Exit`, a similar technique should work for any managed function.

In January 2022, I uploaded a functional example of this approach to my personal GitHub. However, the implementation was a part of a larger project, and I've received a few questions about the technique, so I created this standalone example and writeup. You can find the proof-of-concept code here: https://github.com/outflanknl/unmanaged-dotnet-patch.

Resolving Function Pointers from Managed Code

To better understand the process of resolving managed function pointers, let's start by writing a C# implementation. This idea was first demonstrated by Peter Winter-Smith, in his post Massaging your CLR. First, the program describes the target method using its class, name, and binding constraints. Binding constraints describe attributes of a function, such as accessibility and scope.

```
Type exitClass = typeof(System.Environment);
string exitName = "Exit";
BindingFlags exitBinding = BindingFlags.Static | BindingFlags.Public;
```

The `System.Type` class provides several overloads for `GetMethod` that accept different information to describe a target method. The following code resolves the `System.Reflection.MethodHandle` value for the `Exit` function. This handle points to metadata about the method, not the implementation. One member function of this method handle, `GetFunctionPointer`, will return the implementation start address.

```
MethodInfo exitInfo = exitClass.GetMethod(exitName, exitBinding);
RuntimeMethodHandle exitRtHandle = exitInfo.MethodHandle;
IntPtr exitPtr = exitRtHandle.GetFunctionPointer();
```

As you may have realized, targeting static methods is much simpler than targeting instance methods. It is still possible to target instance methods, but patching may be more difficult in some circumstances. Fortunately, we needed a patch for `System.Environment.Exit`, a static method.

Resolving Function Pointers from Unmanaged Code

Now that we have a strategy to resolve function pointers from managed code, we can move on to an unmanaged implementation. The unmanaged COM interfaces for .NET can resolve and execute managed methods. The approach described below mirrors the managed approach, using COM to resolve and execute the required reflection methods.

Loading Managed Libraries

First, our program must resolve the .NET standard library, mscorlib. We can then use this pointer to resolve any .NET framework classes. The following code will find the default AppDomain and then execute `Load_2` to resolve mscorlib.

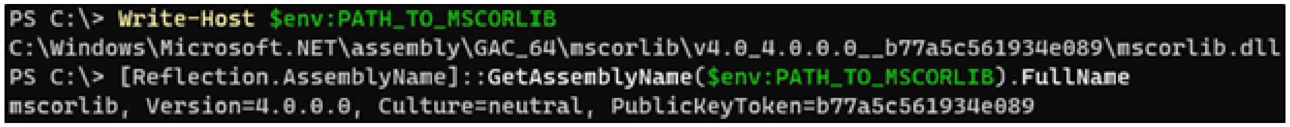
```
IUnknownPtr appDomainUnk;
corRtHost->GetDefaultDomain(&appDomainUnk);

_AppDomain* appDomain;
appDomainUnk->QueryInterface(IID_PPV_ARGS(&appDomain));

_Assembly* mscorlib;
appDomain->Load_2(SysAllocString(L"mscorlib, Version=4.0.0.0"), &mscorlib);
```

If you’re attempting to patch a method outside of mscorlib, you must also load that assembly. If the system you are targeting has only one version of the .NET framework, you should be able to load mscorlib using its name alone. Specify the version or full name for production tools to ensure they load the correct assembly. You can retrieve the full name of an assembly on disk using PowerShell:

```
[Reflection.AssemblyName]::GetAssemblyName(<Assembly Path>).FullName
```



```
PS C:\> Write-Host $env:PATH_TO_MSCORLIB
C:\Windows\Microsoft.NET\assembly\GAC_64\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.dll
PS C:\> [Reflection.AssemblyName]::GetAssemblyName($env:PATH_TO_MSCORLIB).FullName
mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

Resolving Managed Functions

The code below implements our previous managed approach using COM to resolve and invoke the same methods. First, we describe the `Exit` method using its class name, name, and binding constraints to resolve its method info pointer.

```
_Type* exitClass;
mscorlib->GetType_2(SysAllocString(L"System.Environment"), &exitClass);

_MethodInfo* exitInfo;
BindingFlags exitFlags = (BindingFlags)(BindingFlags_Public | BindingFlags_Static);
exitClass->GetMethod_2(SysAllocString(L"Exit"), exitFlags, &exitInfo);
```

Next, we resolve the `MethodHandle` property and retrieve the value for `Exit`. The unmanaged syntax differs significantly from our managed equivalent because `MethodHandle` is an instance property of the `MethodInfo` class.

```
_Type* methodInfoClass;
mscorlib->GetType_2(SysAllocString(L"System.Reflection.MethodInfo"), &methodInfoClass);

_PropertyInfo* methodHandleProp;
BindingFlags methodHandleFlags = (BindingFlags)(BindingFlags_Instance | BindingFlags_Public);
methodInfoClass->GetProperty(SysAllocString(L"MethodHandle"), methodHandleFlags, &methodHandleProp);

VARIANT methodHandlePtr = {0};
methodHandlePtr.vt = VT_UNKNOWN;
methodHandlePtr.punkVal = exitInfo;

SAFEARRAY* methodHandleArgs = SafeArrayCreateVector(VT_EMPTY, 0, 0);
VARIANT methodHandleVal = {0};
methodHandleProperty->GetValue(methodHandlePtr, methodHandleArgs, &methodHandleVal);
```

Finally, the program can resolve and execute `GetFunctionPointer`. Again, the unmanaged syntax looks quite different because it is an instance method of the `RuntimeMethodHandle` class.

```
_Type* rtMethodHandleType;
mscorlib->GetType_2(SysAllocString(L"System.RuntimeMethodHandle"), &rtMethodHandleType);

_MethodInfo* getFuncPtrMethodInfo;
BindingFlags getFuncPtrFlags = (BindingFlags)(BindingFlags_Public | BindingFlags_Instance);
rtMethodHandleType->GetMethod_2(SysAllocString(L"GetFunctionPointer"), getFuncPtrFlags, &getFuncPtrMethodInfo);

SAFEARRAY* getFuncPtrArgs = SafeArrayCreateVector(VT_EMPTY, 0, 0);
VARIANT exitPtr = {0};
getFuncPtrMethodInfo->Invoke_3(methodHandleValue, getFuncPtrArgs, &exitPtr);
```

Patching the Function

The address of `System.Environment.Exit` should now be stored in `exitPtr.byref`. We can disable the function by patching a “return” instruction at the beginning of its implementation. The return instruction on x86 and x86_64 is `0xC3`, so the same patch should work regardless of the .NET assembly and system architectures. The following code demonstrates a simple patching technique. The memory protection of our target is modified to allow modification and then restored.

```
DWORD oldProt = 0;
BYTE patch = 0xC3;

printf("[U] Exit function pointer: 0x%p\n", exitPtr.byref);

VirtualProtect(exitPtr.byref, 1, PAGE_EXECUTE_READWRITE, &oldProt);
memcpy(exitPtr.byref, &patch, 1);
VirtualProtect(exitPtr.byref, 1, oldProt, &oldProt);
```

This solution, while straightforward, could lead to issues with tools that rely on `System.Environment.Exit` to terminate execution. In this case, a different patch may be more appropriate, but that topic is beyond the scope of this post.

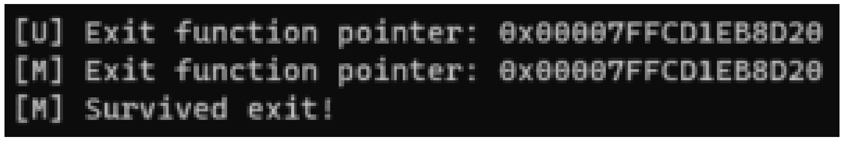
We can use the following .NET program to test our patch. This program will use managed code to find the function address and compare it to the address from our unmanaged implementation.

```
Type exitClass = typeof(System.Environment);
string exitName = "Exit";
BindingFlags exitBinding = BindingFlags.Static | BindingFlags.Public;

MethodInfo exitInfo = exitClass.GetMethod(exitName, exitBinding);
RuntimeMethodHandle exitRtHandle = exitInfo.MethodHandle;
IntPtr exitPtr = exitRtHandle.GetFunctionPointer();

Console.WriteLine("[M] Exit function pointer: 0x{0:X16}", exitPtr.ToInt64());
System.Environment.Exit(0);
Console.WriteLine("[M] Survived exit!");
```

Executing this assembly with the unmanaged host program from the POC repository should produce the following result. Both implementations locate the same address, and the .NET program successfully survives a call to `Exit`.



```
[U] Exit function pointer: 0x00007FFCD1EB8D20
[M] Exit function pointer: 0x00007FFCD1EB8D20
[M] Survived exit!
```

Credits and References

- The managed implementation used to patch `System.Environment.Exit` comes from Peter Winter-Smith. His MDSec blog “Massaging your CLR” inspired our unmanaged approach.
- I published my PoC on Outflank’s GitHub: https://github.com/outflanknl/unmanaged-dotnet-patch