Reverse engineering natively-compiled .NET apps

migeel.sk/blog/2023/09/15/reverse-engineering-natively-compiled-dotnet-apps/

September 15, 2023

Digging into internals of apps built with native AOT.

.NET 7 introduced a new deployment model: native ahead of time compilation. When a .NET app is compiled with native AOT, it gets compiled to a standalone native executable with its own minimal runtime to manage code execution.

This runtime is quite small and in .NET 8 it's possible to build standalone C# apps under 1 MB. To put it in perspective: the size of a native AOT Hello World in C# is closer to the size of a Rust Hello World than to a Golang Hello World, or six times smaller than similar app in Java.

It is also the first time .NET programs are not distributed in the file format defined in ECMA-335 (i.e. instructions and metadata for a virtual machine), but instead distributed as native code (PE/ELF/Mach-O file format), with native data structures same as e.g. C++. This means that none of the reverse engineering tools for .NET built over the past 20 years work with native AOT.

Unfortunately, these two aspects (small size, harder to reverse engineer) made native AOT a popular choice for malware writers as demonstrated by recent discoveries:

- https://malware.news/t/analysis-of-ms-sql-server-proxyjacking-cases/72766
- https://jfrog.com/blog/impala-stealer-malicious-nuget-package-payload/
- https://labs.withsecure.com/content/dam/labs/docs/WithSecure_Research_DUCKTAIL_Returns.pdf

This article will try to go a bit into the details of how to adapt reverse engineering to the new landscape.

Get your Ghidra and native debuggers ready

To reiterate the point from the introduction: Native AOT does not use the CLR VM file formats to store the program and its metadata. Tools that read the VM file format are not useful for native AOT executables. This leaves us with tools used for reverse engineering arbitrary native code such as native debuggers (WinDBG/VS/x64dbg on Windows, lldb/gdb on Unix-like systems) and native code analysis frameworks (Ghidra, IDA, Binary Ninja, etc.).

Since native AOT compiles into a single no-dependency executable, the amount of available metadata considerably shrinks, however there is still some metadata left (as there is with e.g. C++).

First look at a binary

If you'd like to follow along, install a .NET 8 SDK (I'm using RC1 version which was the latest available at the time of writing). You can skip installing and just download the ZIP and put the extracted location on your PATH.

Let's start with building a Hello World with native AOT:

```
$ dotnet new console --aot -o TestApp
```

This will create a new directory TestApp and drop a Hello World console app project configured for AOT compilation there.

```
$ cd TestApp
$ dotnet publish
```

Once the publish process finishes, you should see a binary under bin\Release\net8.0\win-x64\publish (I'm doing this on Windows, but this will be similar for Linux/Mac). The binary is about 1.2 MB in size and there's a file with native debug information next to it (PDB on Windows, DBG on Linux and something else on Mac). Let's take a quick look.

```
$ dumpbin bin\Release\net8.0\win-x64\publish\TestApp.exe
Microsoft (R) COFF/PE Dumper Version 14.37.32824.0
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file bin\Release\net8.0\win-x64\publish\TestApp.exe

File Type: EXECUTABLE IMAGE

  Summary

        D000 .data
       5E000 .managed
        B000 .pdata
       60000 .rdata
        1000 .reloc
        1000 .rsrc
       64000 .text
        1000 _RDATA
       31000 hydrated
```

The file looks mostly standard. The section `.managed` contains managed code (in this case "native code whose memory is managed by the garbage collector"). The `hydrated` section is uninitialized but it gets populated early during startup with runtime data structures.

The rest of sections look fairly standard, `.text` contains unmanaged code such as the garbage collector itself, or other native code that was linked into the executable by the user.

Running `strings` on the executable will find interesting things such as:

```
8.0.23.41904v8.0.0-rc.1.23419.4+92959931a32a37a19d8e1b1684edc6db0857d7de
```

(The version of commit hash of the dotnet/runtime repo that produced the executable, can be handy later.)

But also strings like `DivideByZeroException` or `get_CanWrite` giving a hope we might be able to reconstruct useful type information and method information.

Debugging a memory allocation and virtual call

An interesting experiment to get a feel for how things work is to step through a short piece of code. Let's replace Program.cs with this:

```
using System.Runtime.CompilerServices;

class Program
{
    // Mark NoOpt/NoInline so that all of this doesn't get devirtualized
    // or inlined into managed startup code.
    [MethodImpl(MethodImplOptions.NoOptimization | MethodImplOptions.NoInlining)]
    static void Main() => Console.WriteLine(new Program().ToString());

    public override string ToString() => "Hello World!";
}
```

We `dotnet publish` again and run the program under a debugger. We will leverage the luxury of having debugging symbols for the app. When investigating malware, chances of getting hold of PDB/DBG are very low. We'll set a breakpoint on the Main line and look at the disassembly:

```
00007FF730B8FD50  push        rbp
00007FF730B8FD51  sub         rsp,30h
00007FF730B8FD55  lea         rbp,[rsp+30h]
00007FF730B8FD5A  xor         eax,eax
00007FF730B8FD5C  mov         qword ptr [rbp-8],rax
00007FF730B8FD60  lea         rcx,[TestApp_Program::`vftable' (07FF730BCC688h)]
00007FF730B8FD67  call        RhpNewFast (07FF730AF1DE0h)
00007FF730B8FD6C  mov         qword ptr [rbp-8],rax
00007FF730B8FD70  mov         rcx,qword ptr [rbp-8]
00007FF730B8FD74  call        TestApp_Program___ctor (07FF730B8FDB0h)
00007FF730B8FD79  mov         rcx,qword ptr [rbp-8]
00007FF730B8FD7D  mov         rax,qword ptr [rbp-8]
00007FF730B8FD81  mov         rax,qword ptr [rax]
00007FF730B8FD84  call        qword ptr [rax+18h]
00007FF730B8FD87  mov         rcx,rax
00007FF730B8FD8A  call        System_Console_System_Console__WriteLine_12
(07FF730B56190h)
00007FF730B8FD8F  nop
00007FF730B8FD90  add         rsp,30h
00007FF730B8FD94  pop         rbp
00007FF730B8FD95  ret
```

The code looks quite standard. There's extra register/stack shuffles because we disabled optimizations for pedagogic reasons. The symbolic names are only visible because we had debugging information. If we didn't have it, `TestApp_Program::vftable'` would only be 07FF730BCC688h.

Let's zoom in:

```
00007FF730B8FD60  lea         rcx,[TestApp_Program::`vftable' (07FF730BCC688h)]
00007FF730B8FD67  call        RhpNewFast (07FF730AF1DE0h)
```

Here we can see how allocation works – we load the address of a magical `vftable` structure describing the `Program` class and call a helper `RhpNewFast` to allocate an instance of this from the GC heap. Since .NET is open source, we can look at the details, but essentially this reads a field from the magical `vftable` structure to figure out the size of the allocation (size of a `Program` class instance), slices off a chunk from a zero initialized memory (bump allocation), and writes the address of the `vftable` in the first field of the newly allocated instance, giving the piece of memory "identity". If the bump allocator runs out of memory, there is a slow path, but it's not interesting.

`RhpNewFast` is written in assembly and rarely changes, so chances are good you'll be able to identify it even if no debugging symbols are present.

After allocating a fresh object instance, the instance constructor is called:

```
00007FF730B8FD70  mov         rcx,qword ptr [rbp-8]
00007FF730B8FD74  call        TestApp_Program___ctor (07FF730B8FDB0h)
```

Because we have debugging symbols, we can see the name of the symbol (`TestApp_Program___ctor`). If we didn't have the symbols, this would be `call 07FF730B8FDB0h`.

After the constructor returns, we do the virtual call to `ToString`. This is another interesting part:

```
00007FF730B8FD81  mov         rax,qword ptr [rax]
00007FF730B8FD84  call        qword ptr [rax+18h]
```

First we dereference the reference to the object. As we saw during allocation, this will leave us with the address of the magical `vftable` structure in `rax`. Then we call an address 0x18 bytes into the `vftable` structure. That's presumably where the address of `Program.ToString` method is stored.

The magical `vftable` structure is the vtable or "virtual method table" how we know it from C++. It lists all the virtual method addresses the type implements. It also contains additional metadata such as the size of the object instance, whether it's a struct or class, etc. In .NET world, you're pretty much guaranteed the first 3 slots of the vtable will be implementations of `object.ToString`, `object.GetHashCode` and `object.Equals` (the order of these three depends on whole program optimizations though).

The native AOT codebase calls the vtable structure `MethodTable` or `EEType` interchangeably and you can learn more about it by looking at the implementation of the writer or the reader. (Be warned, there is also a MethodTable in the CoreCLR virtual machine, but layout of that one is different.)

While the `MethodTable` data structure contains a lot of information, the extremely useful one such as names of types is not readily available. Other things that are **not** available are:

- List of all the methods (we can at least list the addresses of the virtual ones, as when reversing C++ though)
- List of all the fields (however, the GC information that prefixes `MethodTable` is able to tell us at what offsets within an object instance are GC pointers - it's better than nothing)
- Containing assembly of the type
- Etc.

Dehydrated data

Additional hurdle is that `MethodTable` data structures get laid out into the `hydrated` segment of the executable that is defined as zero init. There is a small piece of code early in the startup path that populates this segment with actual data. Static analysis tools therefore will have extra trouble interpreting the contents of MethodTables, unless this gets dumped from the memory.

Data dehydration was added in this pull request and describes what happens better than what I could do here. But essentially this data is stored in a more compact form in the file format and gets inflated at runtime. One could potentially simulate this inflation in static analysis tools by locating the data blob that has it, starting from the RTR header. However, this file format is not part of any ABI, it's likely to change, and might need to be updated every year for new .NET versions.

Reflection data structures

While the information about names is not easily accessible, it is still there, as we saw in the strings dump. Reflection keeps track of all type names since in .NET one can just call `object.GetType` on anything and ask about the name of it.

A data blob that maps MethodTable data structures to metadata handles is linked from the RTR header, and so is the metadata blob itself. One could in theory use the metadata reading APIs to reconstruct symbolic names for all MethodTables in the program. However, none of these formats or APIs are meant for public consumption and will likely change with every major .NET release.

Dedicated malware writer could also publish their app with `IlcDisableReflection` property set to true, which will turn on a reflection disabled mode that doesn't generate any reflection metadata. This mode is not supported or documented outside of the dotnet/runtime repo.

Stack trace data structures

Similarly, as we saw in the `strings` dump, information about method names should also be present. The sole reason why it's present is backtrace generation – when an exception is thrown, the developer can call `ToString` on it or access the `StackTrace` property to obtain the textual stack trace. This is implemented by keeping a map between native method addresses and metadata that allows constructing names and signatures. This is similar to how reflection data is generated and the file formats are same (these are also referenced from the RTR header). Let's try it:

```
using System.Runtime.CompilerServices;

class Program
{
    // Mark NoOpt/NoInline so that all of this doesn't get devirtualized
    // or inlined into managed startup code.
    [MethodImpl(MethodImplOptions.NoOptimization | MethodImplOptions.NoInlining)]
    static void Main() => Console.WriteLine(new Program().ToString());

    public override string ToString() => throw new Exception();
}
```

(We updated the previous program by having `ToString` throw an exception and leave it unhandled.)

```
Unhandled Exception: System.Exception: Exception of type 'System.Exception' was
thrown.
   at Program.ToString() + 0x24
   at Program.Main() + 0x37
```

Notice that the app was able to print names and signatures of the involved methods. This will still work even if we delete the debug information by deleting the PDB/DBG file.

However, a user can set StackTraceSupport property to `false` at the time of publishing their app to disable generating this data (stack trace data generation is on by default). Then the program will print this instead:

```
Unhandled Exception: System.Exception: Exception of type 'System.Exception' was
thrown.
   at TestApp!<BaseAddress>+0x9dab4
   at TestApp!<BaseAddress>+0x9da77
```

If the app was built like this, our chances of reconstructing the names or signatures of methods drops close to zero. Some method names might still be available within reflection metadata, but the list of methods that are visible from reflection is typically very small - the compiler aggressively strips it unless trimming analysis tells it to keep it.

Summary

To sum up, analyzing .NET binaries compiled with native AOT requires similar skills to analyzing e.g. C++. Some information is there (like unwinding, limited type information, etc.), but we can forget about luxuries such as being able to break down types into individual fields and monitor their access. Fields basically dissolve into instruction access (we can guess something could be an `int` if it gets read as a 4-byte). Method names will disappear if stack trace data is turned off. And type names can also disappear if reflection is turned off.