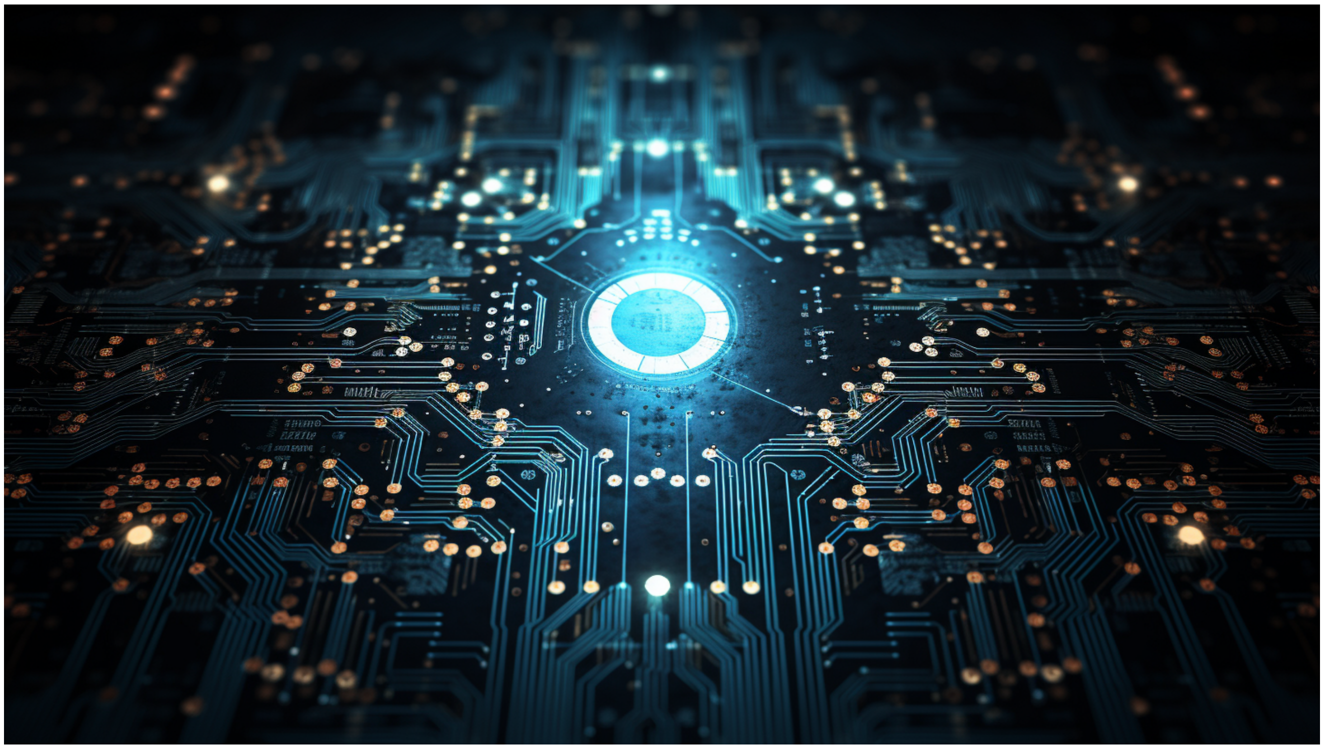
An introduction to reverse engineering .NET AOT applications

harfanglab.io/en/insidethelab/reverse-engineering-ida-pro-aot-net/



About a month ago, we started seeing reports on activities from DuckTail , a cybercrime outfit reportedly based in Vietnam. Detonating one of the samples, we observed that a new account was being created on the analysis machine, followed by an RDP connection from an operator who downloaded additional tools, stole cookies, etc.

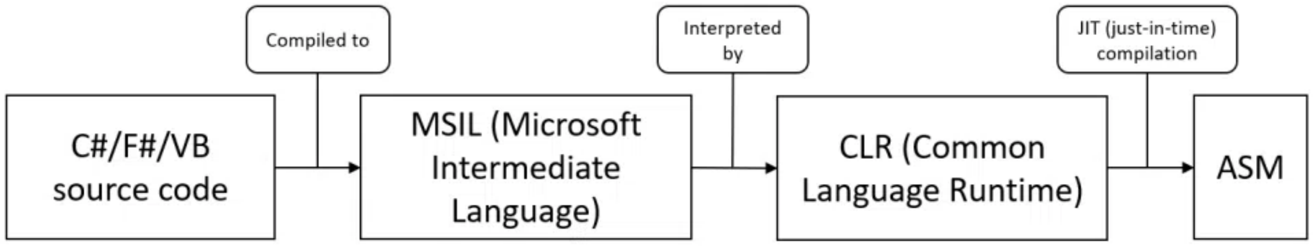
The attack itself had little unique qualities, save for the use of a little-known feature of the .NET language: AOT compilation. We decided to leave the cybercriminals aside and look into the inner workings of such programs in more detail.

What is AOT?

Background on .NET

Most reverse engineers have a good opinion of .NET programs. State-of-the-art handle them very well, reconstructing original source code with a high degree of accuracy in most cases. Sure, they could always be obfuscated, but as far as reverse-engineering goes it was still on the pleasant end of the spectrum.

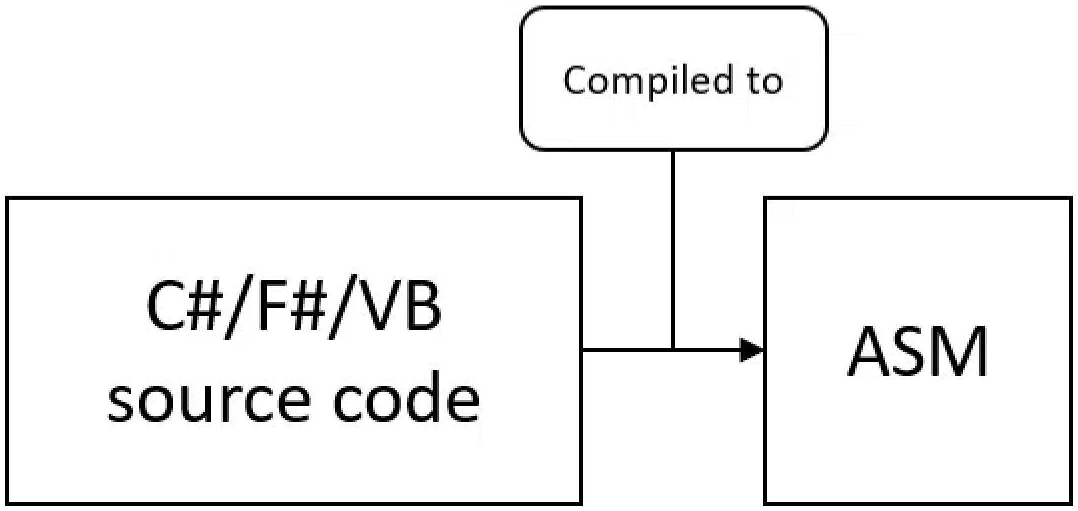
Interpreted languages such as .NET tend to contain a lot more information in the produced binaries than their native counterparts – information that is extremely helpful to the analyst. Source code written by the program author is converted to “Microsoft Intermediate Language” (MSIL) during the compilation phase, and later interpreted by a specific program (the "runtime"). The advantage this approach is that (in theory) the program written in such a language can work on any OS and CPU architecture, as long as there is a runtime available for them:


Compiled to
Interpreted by
JIT (just-in-time) compilation
C#/F#/VB source code
MSIL (Microsoft Intermediate Language)
CLR (Common Language Runtime)
ASM

Even though the “just-in-time" (JIT) compiler can leverage context-specific information to perform optimizations during the execution, it is generally admitted that the “free” portability comes at the cost of slightly reduced performance compared to programs directly compiled to architecture-specific assembly.

Introducing “ahead of time” (AOT) compilation

But what if the multiplatform aspect is of no interest to the developers? What if they know in advance which systems their application will be deployed to? In that case, there is little reason to live with the overhead caused by the intermediate language. If the development toolchain supports it, they might as well produce native binaries directly, as they would if they were writing C or C++ code. This is called AOT (“ahead of time”) compilation:



This is bad news for reverse-engineers, as the disappearance of MSIL in the chain means we have no alternative but to start our analysis at the assembly level.

A quick survey on VirusTotal shows that as of today, there are 544 “clean” AOT binaries on the platform (as in, detected by no antivirus program) against 1667 malicious ones (detected by 3 engines or more). VirusTotal’s selection bias aside, an un-scientific estimation is that around 75% of current .NET AOT samples are malicious, which makes it a decent indicator of malicious behavior. Other researchers have also noted the presence of AOT malware in the wild.

How to recognize .NET AOT binaries?

As far as I can tell, it seems that PE files generated this way have a couple of recognizable characteristics:

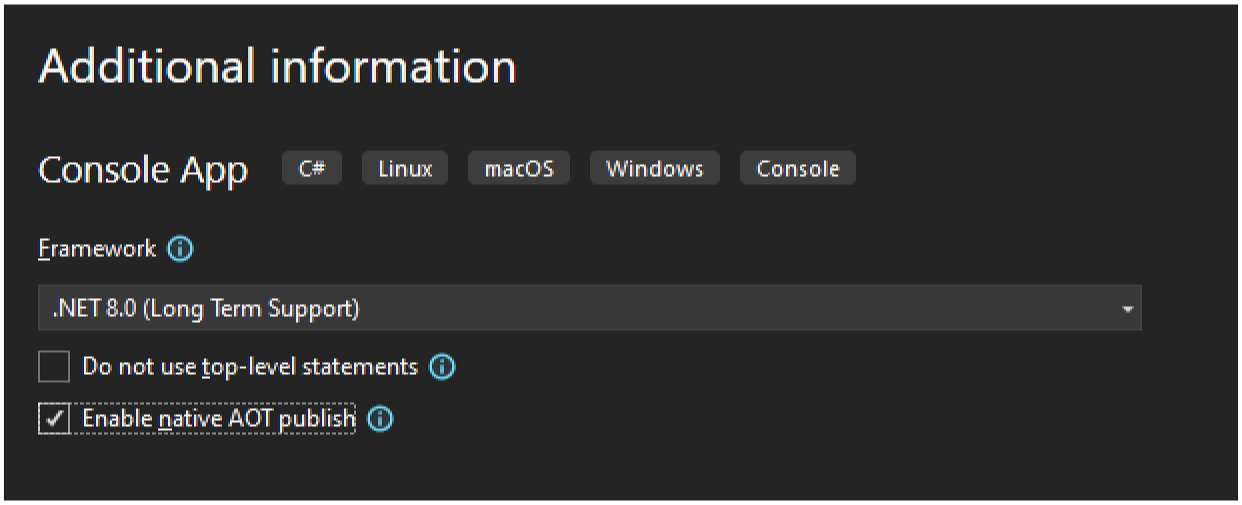
- Only one export (`DotNetRuntimeDebugHeader`)
- A section named `.managed`

Such files can be looked up easily on VirusTotal. In addition, you can find the exact .NET version which was used to compile a sample, looking for a string matching the following regular expression in the binary: `([.\– a-z0-9]*?)\d\+[a-f0-9]{40}\b`.
For instance: `8.0.0+5535e31a712343a63f5d7d796cd874e563e5ac14`, with the last part being the hash of the corresponding commit in the .NET runtime source code – always useful to detect tampered compilation timestamps.

Setting up AOT for test projects

Setting up a blank project to get acquainted with AOT code is not completely trivial, especially if you’re unfamiliar with Visual Studio. There are a few prerequisites:

Create a new C# “Console Application” project and make sure to check “Enable native AOT publish" (this option is only available in .NET 8 and later; for version 7 you must manually add the property `<PublishAot>true</PublishAot>` to your file `.csproj`):


Additional information

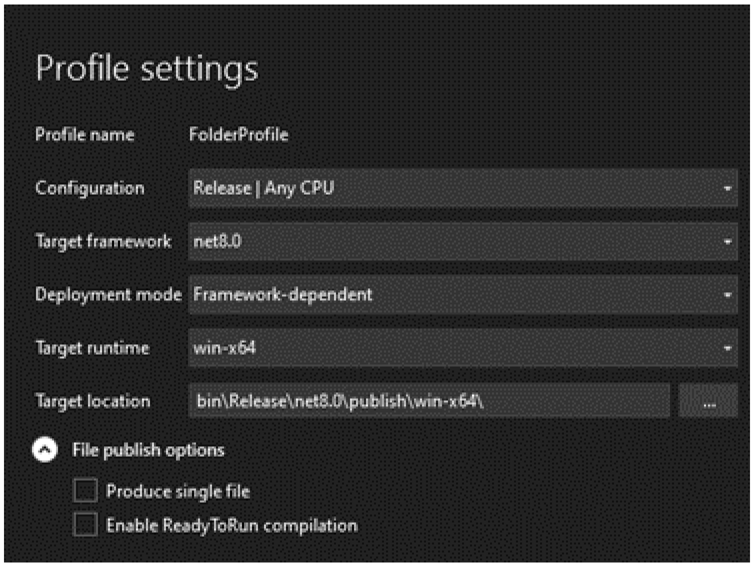
Console App C# Linux macOS Windows Console

Framework: .NET 8.0 (Long Term Support)

[ ] Do not use top-level statements

[✓] Enable native AOT publish

If you haven't touched .NET for a while, you'll notice that the "Hello World" template is now a one-liner – the `Program` class declaration appears to be implicit now. Feel free to restore the explicit code, as it makes things a little clearer when we look at the corresponding assembly code. In order to generate the AOT binary, right-click on your project and select "Publish". For our purposes, the easiest option seems to be publishing to a folder on disk. There are a few more options to specify for the build, mainly the target CPU architecture (x64 in this case):



Profile settings

Profile name	FolderProfile
Configuration	Release | Any CPU
Target framework	net8.0
Deployment mode	Framework-dependent
Target runtime	win-x64
Target location	bin\Release\net8.0\publish\win-x64\

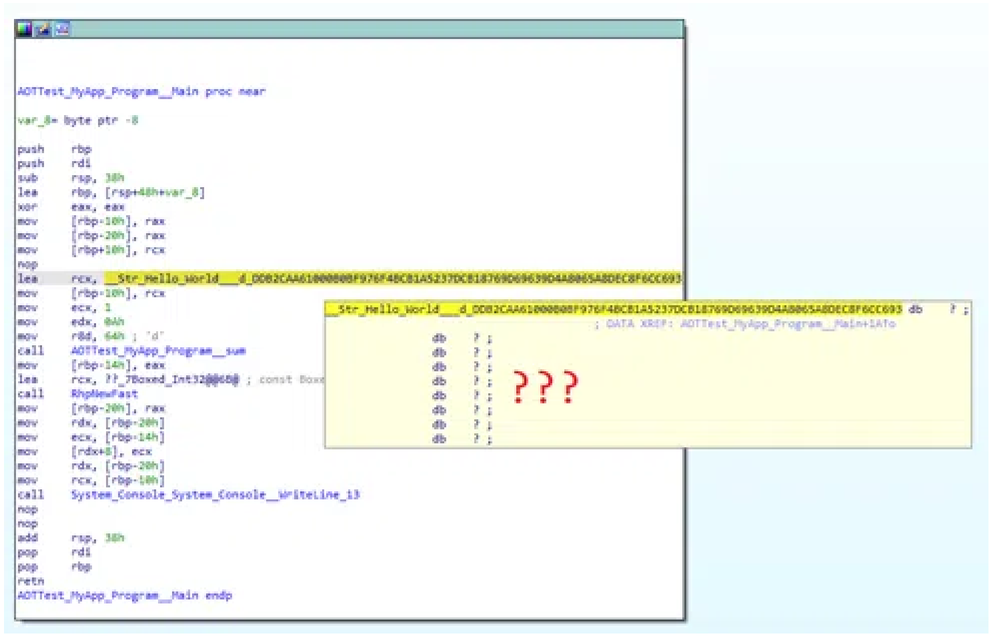
File publish options
- ☐ Produce single file
- ☐ Enable ReadyToRun compilation

Once this is taken care of, clicking the "Publish" button should generate a self-contained binary in the specified folder. Our one-line "Hello World" project weighs around 1.2MB (3MB when optimizations are disabled), which is terrible news because it means the program embeds a lot of library code.

IDA Pro and AOT in practice

Looking at sample programs leads us to a few observations:

- .NET AOT looks a lot like C++ code, which is no surprise since Visual Studio required the C++ component to generate AOT files.
- The calling convention for x64 appears to be pretty standard, using the registers RCX, RDX, R8 and R9 and then the stack to pass arguments.
- IDA doesn't have signatures for .NET runtime functions compiled ahead of time, so nothing is recognized.
- Pointers to strings lead to uninitialized memory in a section called `hydrated` 😱



```
AOTTest_MyApp_Program__Main proc near

var_8= byte ptr -8

push    rbp
push    rdi
sub     rsp, 38h
lea     rbp, [rsp+40h+var_8]
xor     eax, eax
mov     [rbp-10h], rax
mov     [rbp-20h], rax
mov     [rbp+10h], rcx
nop
lea     rcx, __Str_Hello_World___d_DDB2CAA61000000F976F48CB1A5237DCB18769D69639D4A8065A8DEC8F6CC693
mov     [rbp-10h], rcx
mov     ecx, 1
mov     edx, 0Ah
mov     r8d, 64h ; 'd'
call    AOTTest_MyApp_Program__sum
mov     [rbp-14h], eax
lea     rcx, ??_7Boxed_Int32@@6B@ ; const Boxe
call    RhpNewFast
mov     [rbp-20h], rax
mov     rdx, [rbp-20h]
mov     ecx, [rbp-14h]
mov     [rdx+8], ecx
mov     rdx, [rbp-20h]
mov     rcx, [rbp-10h]
call    System_Console_System_Console__WriteLine_13
nop
nop
add     rsp, 38h
pop     rdi
pop     rbp
retn
AOTTest_MyApp_Program__Main endp
```

```
__Str_Hello_World___d_DDB2CAA61000000F976F48CB1A5237DCB18769D69639D4A8065A8DEC8F6CC693 db    ? ;
                                        ; DATA XREF: AOTTest_MyApp_Program__Main+1Ato
                db    ? ;
                db    ? ;
                db    ? ;
                db    ? ;
                db    ? ;
                db    ? ;
                db    ? ;
                db    ? ;
```

The latter phenomenon can be traced back to an optimization which aims at reducing the binary size during the compilation step. What it means for us is that we will be forced to use a debugger to figure out what strings are being manipulated during runtime.

Identifying library functions

Let's assume we're now working on an unknown .NET AOT sample. Our first order of business is to obtain names for the library functions which represent over 95% of the binary. Without those, it's safe to say that we're not going to make it.

To achieve this, we follow an approach similar to what Hex-Rays has been doing with Golang and their go2pat utility – generate a .NET AOT binary which contains all possible imports, then extract all symbols and generate byte patterns for them. Our approach doesn't need to be as generic however: we know which .NET version we're targeting thanks to a string present in the target binary (see previous title). What we need next is to create a binary containing as many library functions as possible so we can make signatures for them. Unfortunately for us, the AOT compiler trims all unused code and there is no way to turn this off.

We can work around this issue, because even though we need to *use* the functions we're importing, there's no need to do anything meaningful with them to prevent the compiler from optimizing them out. But how to generate such code? I have zero experience writing .NET, but you know who does? Language models. After a few incremental queries, I generated a source file we can compile into a 10MB AOT binary – along with its PDB file containing all associated symbols. There's probably a lot missing still, but this should be plenty of library code to begin with.

Creating IDA Pro signatures

The next step is to create signatures for known .NET AOT functions. For this, we can use FLARE's idb2pat.py script to generate a `.pat` for the ~31000 functions contained in my test program. The `.pat` file then needs to be converted to a `.sig` likely to be used by IDA; I'm about to go into detail on this aspect, as I know that most reversers don't often have the opportunity to do so.
(more complete instructions are available here).

First, you have to download FLAIR utilities from the Hex-Rays download center . Presently, we have a `.pat` which contains entries like this:

```
488D05........488D0D........837908017501C3488BD0E9.............. 00 0000 001D
:00000000 __GetNonGCStaticBase_System_Collections_System_SR :00000015
loc_140001015 ^00000003 ?__NONGCSTATICS@System_Collections_System_SR@@
^0000000A off_14094E4D0 ^00000019
S_P_CoreLib_System_Runtime_CompilerServices_ClassConstructorRunner__CheckStati
cClassConstructionReturnNonGCStaticBase
```

Each line is essentially a byte-pattern like those you would find today in Yara rules, followed by lengths and checksums, the name of the function as well as referenced names. The full documentation for this format can be found in `pat.txt`in the FLAIR folder. The following command compiles those human-readable-ish patterns into binary signatures:

```
sigmake.exe input.pat output.sig -n "[description]"
```

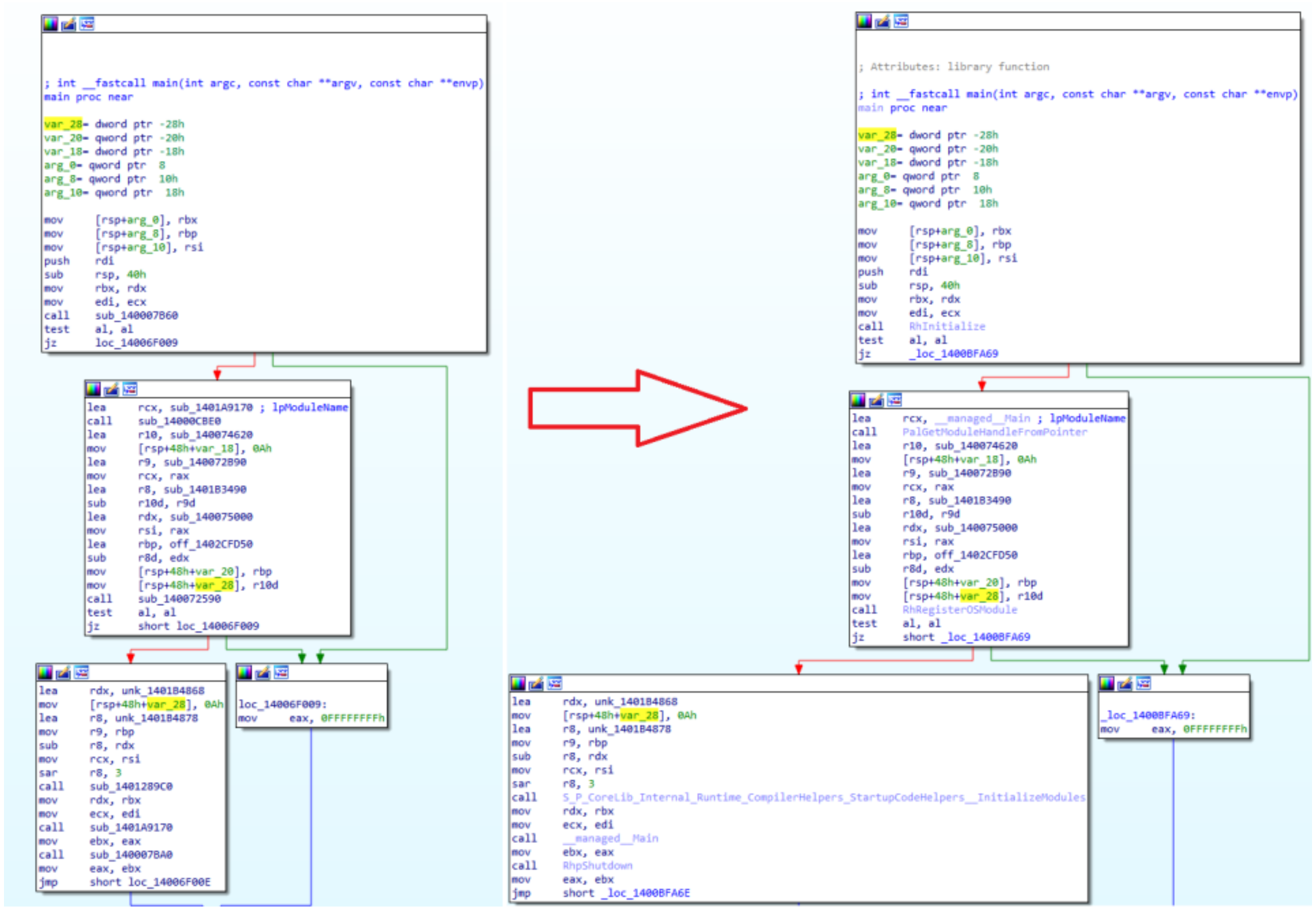
However, odds are that this will not immediately work as `idb2pat` created the patterns automatically, and collisions are likely to happen (~800 in my case). The process for resolving them involves editing the `.exc` file created by `sigmake` in the same folder, and manually indicating which function to keep:

```
;--------- (delete these lines to allow sigmake to read this file)

; add '+' at the start of a line to select a module

; add '-' if you are not sure about the selection

; do nothing if you want to exclude all modules


memcpy_CopyDown_Intel                              BD 6EF1 [...]

memcpy_CopyDown_amd                                BD 6EF1 [...]


Bool__System_IConvertible_ToByte                   00 0000 [...]

Bool__System_IConvertible_ToInt16                  00 0000 [...]
```

```
Bool__System_IConvertible_ToInt32            00 0000 [...]

Bool__System_IConvertible_ToSByte            00 0000 [...]

Bool__System_IConvertible_ToUInt16           00 0000 [...]

Bool__System_IConvertible_ToUInt32           00 0000 [...]

Bool__System_IConvertible_ToUInt64           00 0000 [...]


[...]
```

As you will see, in most cases the functions seem to be similar enough that losing the name due to ambiguity would be a shame. At this stage, I just use the search and replace function of a text editor to add a "+" character at the start of each line following a blank one (replace `^$\r\n` by `\r\n\+`). Do not forget to delete the first lines of the file as instructed, and run again `sigmake`.

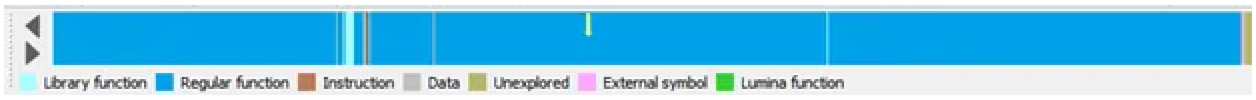
We can now place the obtained `.sig` file in the `$IDAUSR/sig/pc`folder and apply it manually via the `File > Load File > FLIRT Signature File`.



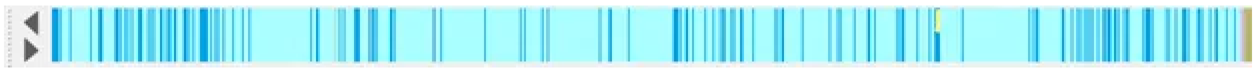
```
; int __fastcall main(int argc, const char **argv, const char **envp)
main proc near

var_28= dword ptr -28h
var_20= qword ptr -20h
var_18= dword ptr -18h
arg_0= qword ptr  8
arg_8= qword ptr  10h
arg_10= qword ptr  18h

mov     [rsp+arg_0], rbx
mov     [rsp+arg_8], rbp
mov     [rsp+arg_10], rsi
push    rdi
sub     rsp, 40h
mov     rbx, rdx
mov     edi, ecx
call    sub_140007B60
test    al, al
jz      loc_14006F009

lea     rcx, sub_1401A9170 ; lpModuleName
call    sub_14000CBE0
lea     r10, sub_140074620
mov     [rsp+48h+var_18], 0Ah
lea     r9, sub_140072B90
mov     rcx, rax
lea     r8, sub_1401B3490
sub     r10d, r9d
lea     rdx, sub_140075000
mov     rsi, rax
lea     rbp, off_1402CFD50
sub     r8d, edx
mov     [rsp+48h+var_20], rbp
mov     [rsp+48h+var_28], r10d
call    sub_140072590
test    al, al
jz      short loc_14006F009

lea     rdx, unk_1401B4868
mov     [rsp+48h+var_28], 0Ah
lea     r8, unk_1401B4878
mov     r9, rbp
sub     r8, rdx
mov     rcx, rsi
sar     r8, 3
call    sub_1401289C0
mov     rdx, rbx
mov     ecx, edi
call    sub_1401A9170
mov     ebx, eax
call    sub_140007BA0
mov     eax, ebx
jmp     short loc_14006F00E

loc_14006F009:
mov         eax, 0FFFFFFFFh
```

```
; Attributes: library function

; int __fastcall main(int argc, const char **argv, const char **envp)
main proc near

var_28= dword ptr -28h
var_20= qword ptr -20h
var_18= dword ptr -18h
arg_0= qword ptr  8
arg_8= qword ptr  10h
arg_10= qword ptr  18h

mov     [rsp+arg_0], rbx
mov     [rsp+arg_8], rbp
mov     [rsp+arg_10], rsi
push    rdi
sub     rsp, 40h
mov     rbx, rdx
mov     edi, ecx
call    RhInitialize
test    al, al
jz      _loc_1400BFA69

lea     rcx, __managed__Main ; lpModuleName
call    PalGetModuleHandleFromPointer
lea     r10, sub_140074620
mov     [rsp+48h+var_18], 0Ah
lea     r9, sub_140072B90
mov     rcx, rax
lea     r8, sub_1401B3490
sub     r10d, r9d
lea     rdx, sub_140075000
mov     rsi, rax
lea     rbp, off_1402CFD50
sub     r8d, edx
mov     [rsp+48h+var_20], rbp
mov     [rsp+48h+var_28], r10d
call    RhRegisterOSModule
test    al, al
jz      short _loc_1400BFA69

lea     rdx, unk_1401B4868
mov     [rsp+48h+var_28], 0Ah
lea     r8, unk_1401B4878
mov     r9, rbp
sub     r8, rdx
mov     rcx, rsi
sar     r8, 3
call    S_P_CoreLib_Internal_Runtime_CompilerHelpers_StartupCodeHelpers__InitializeModules
mov     rdx, rbx
mov     ecx, edi
call    __managed__Main
mov     ebx, eax
call    RhpShutdown
mov     eax, ebx
jmp     short _loc_1400BFA6E

_loc_1400BFA69:
mov         eax, 0FFFFFFFFh
```

The results are really satisfying, as the navigator bar at the top of the screen goes from this:



...to this:



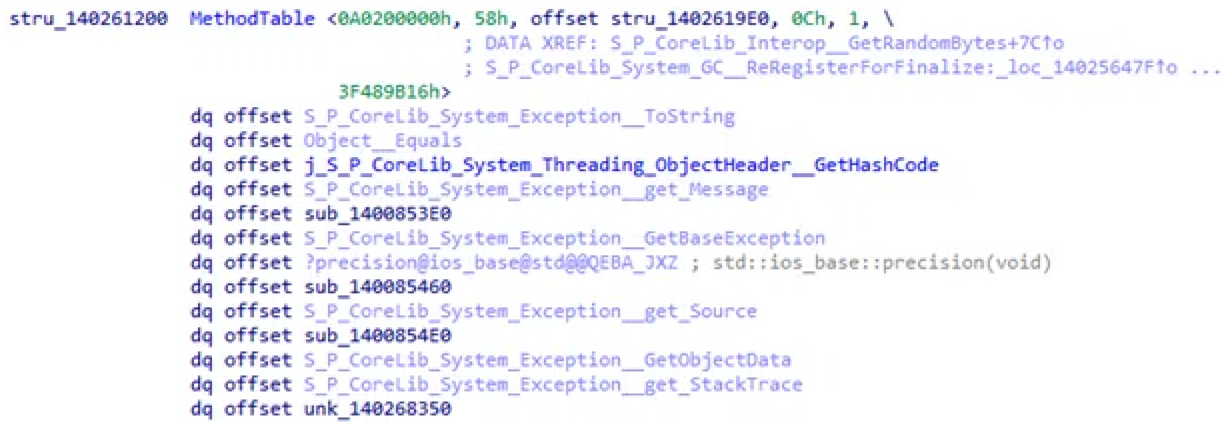
You can find my `.sig` here , but I would only recommend it for binaries built with .NET 7.0.

What next?

Now that the library functions are identified properly, the original repository can serve as documentation – let's start with RhpNewFast, a crucial function tasked with allocating objects. It receives a pointer to a MethodTable structure which represents the object to allocate. We need to figure out what is being instantiated, but obviously we do not have names for static structures. The decompiled code is littered with calls like:

```
v4 = RhpNewFast(&stru_140261200);
```

`MethodTable` structures (or `EETypes`) contain basic information about the corresponding object in their header (such as its size, related types, and flags controlling the garbage collector behavior), followed by a function table reminiscent of C++ vtables:



```
stru_140261200  MethodTable <0A0200000h, 58h, offset stru_1402619E0, 0Ch, 1, \
                                        ; DATA XREF: S_P_CoreLib_Interop__GetRandomBytes+7C↑o
                                        ; S_P_CoreLib_System_GC__ReRegisterForFinalize:_loc_14025647F↑o ...
                             3F489B16h>
                dq offset S_P_CoreLib_System_Exception__ToString
                dq offset Object__Equals
                dq offset j_S_P_CoreLib_System_Threading_ObjectHeader__GetHashCode
                dq offset S_P_CoreLib_System_Exception__get_Message
                dq offset sub_1400853E0
                dq offset S_P_CoreLib_System_Exception__GetBaseException
                dq offset ?precision@ios_base@std@@QEBA_JXZ ; std::ios_base::precision(void)
                dq offset sub_140085460
                dq offset S_P_CoreLib_System_Exception__get_Source
                dq offset sub_1400854E0
                dq offset S_P_CoreLib_System_Exception__GetObjectData
                dq offset S_P_CoreLib_System_Exception__get_StackTrace
                dq offset unk_140268350
```

Thankfully these methods are identified properly with our signature file, and we can deduce that the object being instantiated is some sort of Exception (specifically, `OutOfMemoryException` even though this is not visible here).

Can we do better? .NET actually has RTTI capabilities. While I'm sure there's a way to write a Python script that recovers all type information (but haven't dug into how such information is stored), we can score quick wins using a debugger. Intercept calls to the `S_P_CoreLib_System_Type__GetTypeFromEETypePtr` (or its wrapper, `Object__GetType`), put a pointer to the desired `MethodTable` in ECX, and wait for the call to `NativeFormatRuntimeNamedTypeInfo__ToString` to get your desired answer in the return value (just follow EAX in the heap).

Wrap-up: a generic approach for unknown frameworks

There's no doubt that .NET AOT programs will give reversers a hard time, *especially* when compared to their MSIL counterparts. However, using the techniques described above, we were able to recover symbols as well as typing information. This brings us back to a situation close to that of analyzing programs in Go (but with a functional decompiler). From here on I'd recommend following the approach I use for Golang, i.e. using a debugger to inspect calls to standard library functions and try to reconstruct the original code in this way.

This blog post also demonstrates a generic approach that can be followed when attempting to analyze programs written in a new language (which .NET AOT is, from our reversing perspective). I always tend to go for quick and dirty approaches at first, but if you'd like to know more about AOT internals, I recommend checking out Michal Strehovsky's blog post on the subject.

As a closing note, I will share my opinion that AOT malware will become more commonplace in the future, because of two main factors:

- Microsoft is making it easier with new .NET versions to publish AOT applications;
- It's a free "make my app annoying to reverse" button – word is simply going to spread.

Feel free to reach out on Twitter if you have other useful techniques when analyzing these binaries!

Indicators of compromise

`9ba3b2ce74d60e0960be0e2544f7497339f1f115db93afb94e5512a8c990f63f`
`BotGetKeyChromium`

`268aec06d44359b21bfe1c0c13abb75d1e37add2c8512acb6e0a0835b939b9b9`
`BotGetKeyChromium`

`9b8a1424cd299629e8dccdb1c7c4f3caad78fecec083c9e27b6a3dc281d5b1ca`
`BotGetKeyChromium`

`6d689bfc12d18a6e4dae9309e3260f71d93de1fb9864f8545cbc30a24e181b1f`
`BotGetKeyChromium`

`7fd054a810f5d942bc18d91d8e31285b484982bf5c8ace0c12c8ad64b0f183d4`
`BotGetKeyChromium`

```
9ba3b2ce74d60e0960be0e2544f7497339f1f115db93afb94e5512a8c990f63f
BotGetKeyChromium

1e082ed9733b033a0c9b27a0d1146397771b350b013ea3e9fba228e1400a263f ResetMainBot

ab8d86ac204d9c9ae689d87b9d2f7319b38125f7659ff2ba7cbfed13cbf0a13d ResetMainBot
```

Yara rules

```
import "pe"
rule NET_AOT {
    meta:
        description = "Detects .NET binaries compiled ahead of time (AOT)"
        author = "HarfangLab"
        distribution = "TLP:WHITE"
        version = "1.0"
        last_modified = "2024-01-08"
        hash = "5b922fc7e8d308a15631650549bdb00c"

    condition:
        pe.is_pe and pe.number_of_exports == 1 and
        pe.exports("DotNetRuntimeDebugHeader") and
        pe.section_index(".managed") >= 0
}
```

Published January 15, 2024 Last updated on January 29, 2024