4 - 6 October, 2023 / London, United Kingdom

R2R STOMPING – ARE YOU READY TO RUN?

Jiří Vinopal

Check Point Research, Czechia

jiriv@checkpoint.com

ABSTRACT

What if I told you that the reality you perceive with your very own eyes is not always what it seems? That the .NET code you witness executing within your beloved managed debugger, such as *dnSpy*/*dnSpyEx*, may not necessarily be the same code that operates outside of its bounds?

.NET application startup time and latency can be improved by compiling application assemblies as ReadyToRun (R2R) format files, which is a form of ahead-of-time (AOT) compilation. Binaries compiled this way contain similar native code to what JIT would produce, but they are larger because they contain both Intermediate Language (IL) code and the native version of the 'same code'. Or at least, that's what the documentation says.

This paper introduces a new method for running hidden implanted code in ReadyToRun (R2R) compiled .NET binaries. The method focuses on the possibility of altering R2R compiled binaries in such a way that the original IL code of the assembly differs from the pre-compiled native code, which is a part of the produced binary too. Because of the .NET optimization, the pre-compiled native code will be prioritized and will run, ignoring the original IL code.

Furthermore, because of the debugging experience, the default optimization settings of managed debuggers such as *dnSpy*/*dnSpyEx* differ, resulting in different code execution comparing normal execution of the altered R2R compiled binary and execution in the context of the managed debugger.

This paper will focus on the following:

- Introduction to R2R stomping
- Implementation of R2R stomping with an explanation of the internals
- The resulting problems that will affect the work of the reverse engineer
- Techniques and tools to reverse engineer R2R stomped assemblies
- Possible ways of detecting R2R stomping

INTRODUCTION TO R2R STOMPING

Before we dive into the ReadyToRun compilation format of dotnet applications, a little recap about .NET in general is needed.

The dotnet framework, originally created by *Microsoft*, is an open-source, cross-platform environment for building many different types of applications. Specific programming and scripting languages run on top of the framework (C#, F#, VB.NET, PowerShell). When it was first introduced in 2002 as the '.NET Framework', it was a *Windows*-only platform and closed-source. Two years later, *Ximian* introduced the first open-source, cross-platform version of the .NET Framework, known as 'Mono Project'. It took some time for *Microsoft* to react and bring its own open-source, cross-platform version, '.NET Core' (2016). This *Microsoft* solution evolved into its successor, '.NET' (.NET 5 in 2020). As the ReadyToRun format of dotnet compilation was first introduced in .NET Core 3.0, the technique introduced in this paper (R2R stomping) targets dotnet versions from .NET Core 3.0+ to .NET 5+.

Usually, a regular .NET assembly only contains a managed code (also known as Intermediate Language, IL code, MSIL, CIL), which needs to be compiled and interpreted into its form of native code by the just-in-time compiler (JIT) after the application starts. As the usage of the dotnet environment to build many different types of applications is becoming more and more popular, a lot of pressure has been put on improvements regarding its latency and relatively slow application startup time, caused by JIT.

Since JIT compilation is the main cause of slow startup time and speed of execution, logical solutions that help us target this problem are, in general, reducing the amount of code that needs to be JIT-compiled, or avoiding JIT usage at all. Such solutions are coming up with different types of compilation formats for dotnet assemblies that generally use a form of ahead-of-time (AOT) compilation.

The main formats of AOT compilation are:

- NGEN – .NET Framework only, considered to be a somewhat fragile solution [1].
- ReadyToRun – From .NET Core 3.0+, reducing the need for JIT by pre-compilation.
- Native AOT – From .NET 7+, full native format (PE + CPU code), no need for .NET runtime to be installed, no usage of JIT, no IL code and .NET metadata [2].

Once the application assemblies are compiled in a ReadyToRun (R2R) format, a form of AOT, resulting binaries contain similar native code to what JIT would produce, but they are larger because they contain both Intermediate Language (IL) code and the native version of the 'same code' [3]. Because this format still depends on the original dotnet metadata of assembly, they are also a part of the produced binary.

So, in general, such binaries conform to CLI file format as described in ECMA-335 [4] but enrich it with a 'ManagedNativeHeader' pointing to a specific 'READYTORUN_HEADER' followed by other structures needed for successful execution of pre-compiled native code. The signature field of 'READYTORUN_HEADER' is always set to 0x00525452 (ASCII encoding for 'RTR'). The signature can be used to distinguish ReadyToRun images from other CLI images with 'ManagedNativeHeader' (e.g. NGen images) [5].

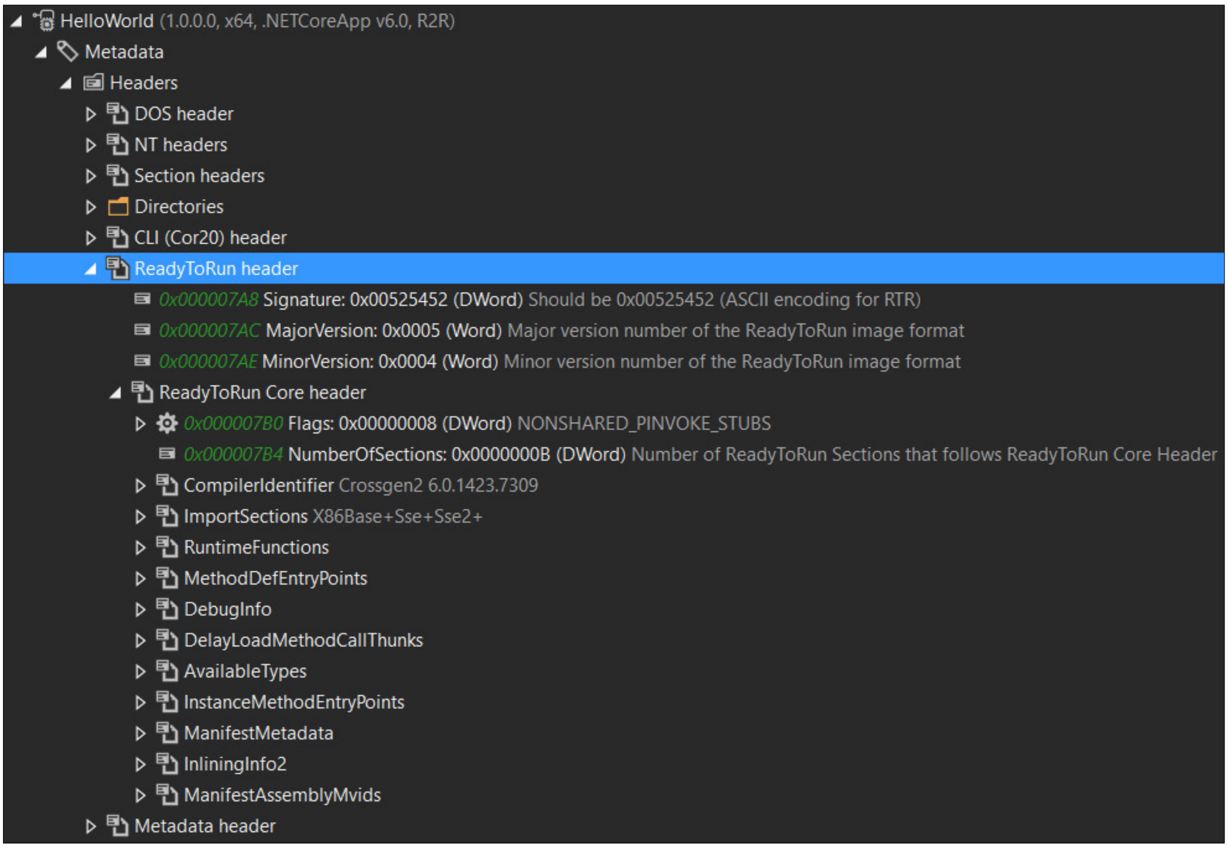


Figure 1: The ReadyToRun header structure parsed in the dotPeek tool.

The 'R2R stomping' method focuses on the possibility of altering R2R compiled binaries in such a way that the original IL code of the assembly will differ from the pre-compiled native code, which is a part of the produced binary too. Because of the .NET optimization, the pre-compiled native code will be prioritized and run, ignoring the difference to the original IL code of such assembly.

Furthermore, the default optimization settings of managed debuggers such as *dnSpy*/*dnSpyEx* differ (suppressing the JIT optimization), resulting in different code execution comparing normal execution of the altered R2R compiled binary and execution in the context of the managed debugger.

IMPLEMENTATION OF R2R STOMPING

As already mentioned, the main idea behind the R2R stomping implementation is to modify the original code of compiled assembly in a way that the capability and behaviour of the method's IL code would differ from the pre-compiled native code.

Such modifications could be done in two ways:

- Compile real – replace with decoy: replacement of the compiled IL code, leaving the original pre-compiled code.
- Compile decoy – replace with real: replacement of the pre-compiled native code, leaving the original IL code.

During the implementation of R2R stomping, we need to keep in our mind that either the original IL code or the pre-compiled native code we decide to preserve still depends on the original metadata of the dotnet assembly. In other words, we must be very careful not to change the metadata in a way that could later result in failure during the execution.

Despite the fact that we chose the *Windows* OS, x64, and .NET 6 as the targeted environment for our implementation example, we were able to successfully test the R2R stomping method in a wide range of dotnet runtimes (supporting ReadyToRun), from .NET Core to .NET 7 across different architectures and OS platforms (*Windows*, *Linux*, *macOS*).

It is worth noting that the R2R stomping could be further combined with different compilation settings, such as those producing dotnet bundle (single-file) or self-contained assembly [6]. In the implementation shown, these compilation formats were omitted to simplify the explanation of R2R stomping, but once they are applied, they would make analysis of the file more difficult regarding R2R stomped methods.

Compile real – replace with decoy

In this implementation, the target code for a replacement is the IL code of the produced assembly, leaving the pre-compiled native code intact.

We will start with the creation of a new project in Visual Studio IDE [7], selecting C#, Console App, and building on top of .NET (in our case, .NET 6).

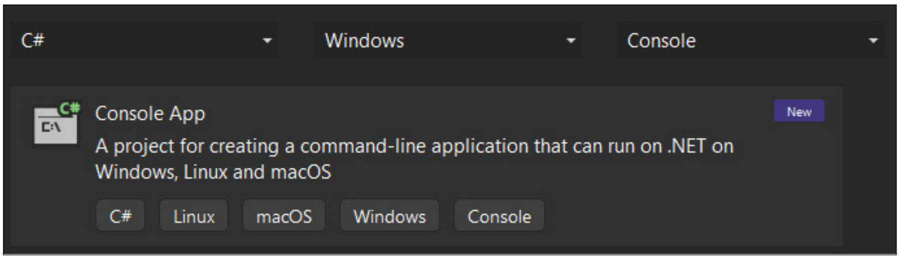


Figure 2: Visual Studio IDE – creation of new C#, Console App, .NET 6 project.

To build our non-self-contained, ReadyToRun application, we can directly specify the 'PublishReadyToRun' flag to the dotnet publish command `dotnet publish -c Release -r win-x64 -p:PublishReadyToRun=true --self-contained false`.

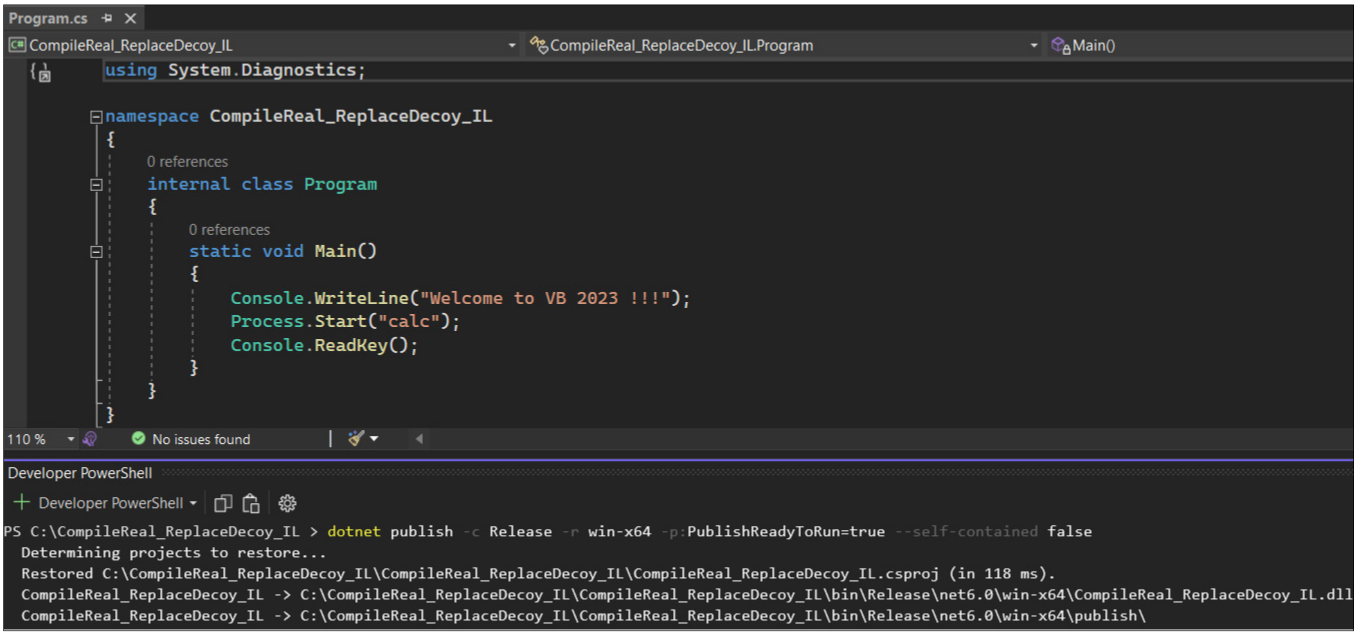


Figure 3: Building the ReadyToRun application with the dotnet publish command.

To demonstrate the modification of the IL code, we can simply replace the `Process.Start("calc")` method invocation and its appropriate IL code with nops instructions. To achieve this, we can choose either the GUI-based tool *dnSpyEx* [8] or the programmatic way using libraries such as *AsmResolver* [9] or *dnlib* [10]. Whichever approach we choose, preserving as much from the original metadata and PE structure as possible is important so as not to strip the pre-compiled native code from the dotnet module.

Approach using dnSpyEx

1. First, open the compiled ReadyToRun assembly in the *dnSpyEx*, as shown in Figure 4.
2. Next, edit the IL instructions related to the `Process.Start("calc")` method invocation – replace with nops instructions, as shown in Figure 5.
3. Save the patched module – preserve as much as possible and make sure the 'Mixed-Mode Module' option is checked, as shown in Figure 6.

The newly created ReadyToRun stomped assembly will not reveal any evidence of code related to the creation of the `calc` process either in the IL view or in the decompiled view of C# code (see Figure 7).

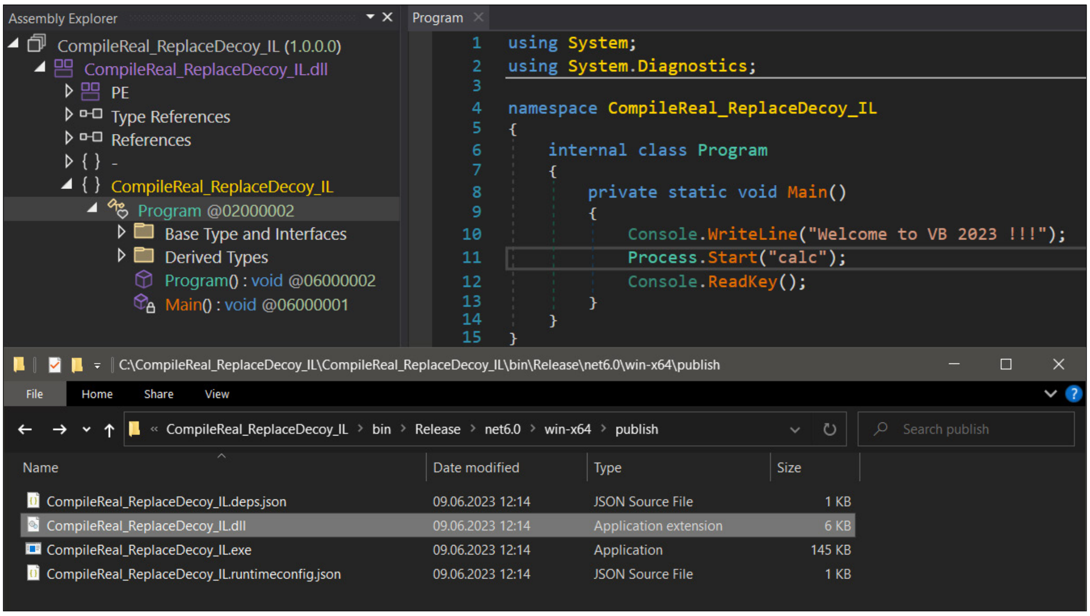


Figure 4: DnSpyEx: opening ReadyToRun assembly.

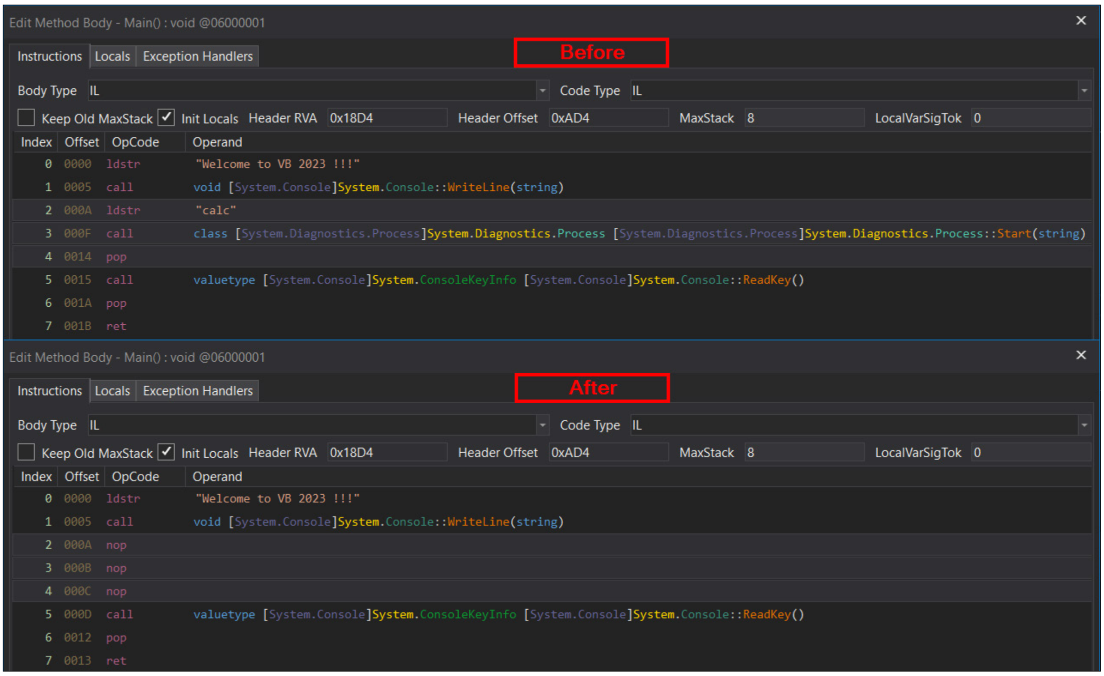


Figure 5: Editing IL instructions in dnSpyEx.

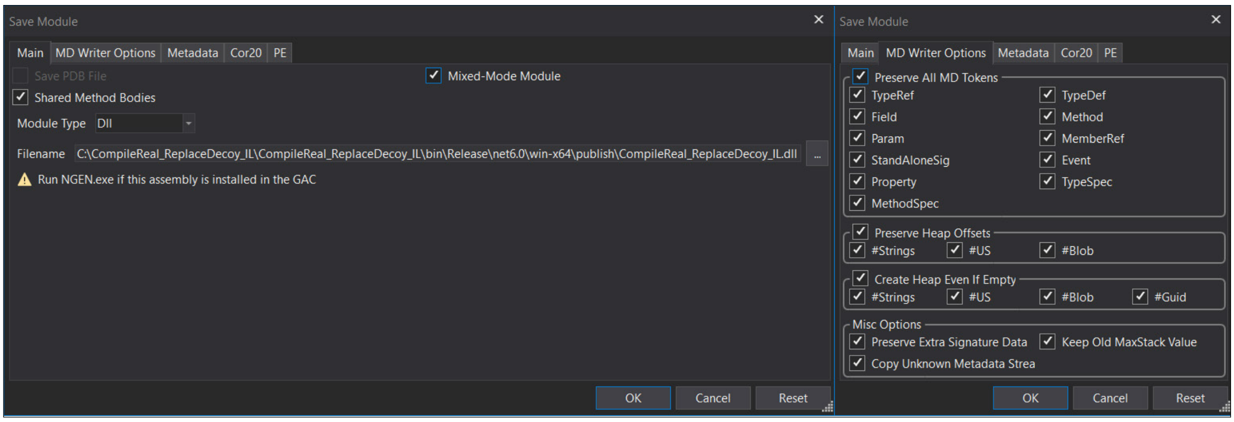


Figure 6: Saving the patched module in dnSpyEx.

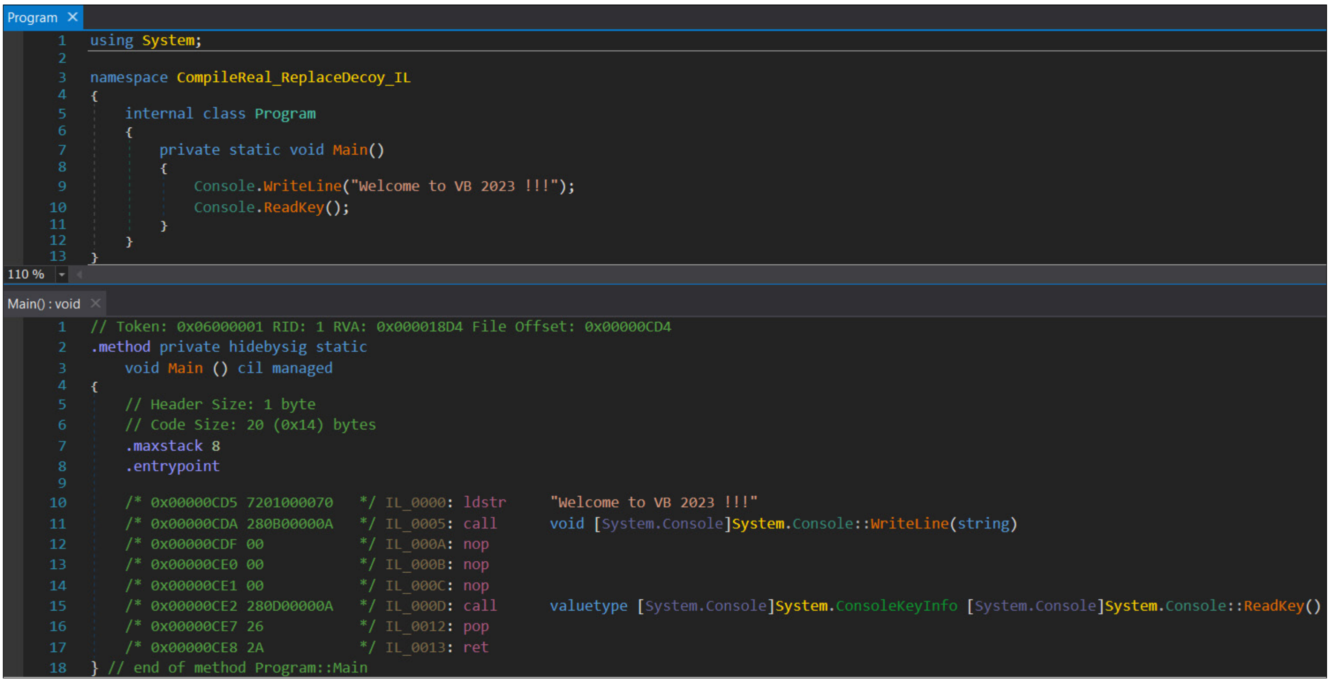


Figure 7: C# view and IL view of the ReadyToRun stomped assembly.

However, once we try to run our patched ReadyToRun assembly normally, either via its associated executable `CompileReal_ReplaceDecoy_IL.exe` located in the same folder or via issuing the `dotnet CompileReal_ReplaceDecoy_IL.dll` command from a command prompt, we can spot that our pre-compiled native code was executed, ignoring the difference to the patched IL code (process `calc.exe` started).

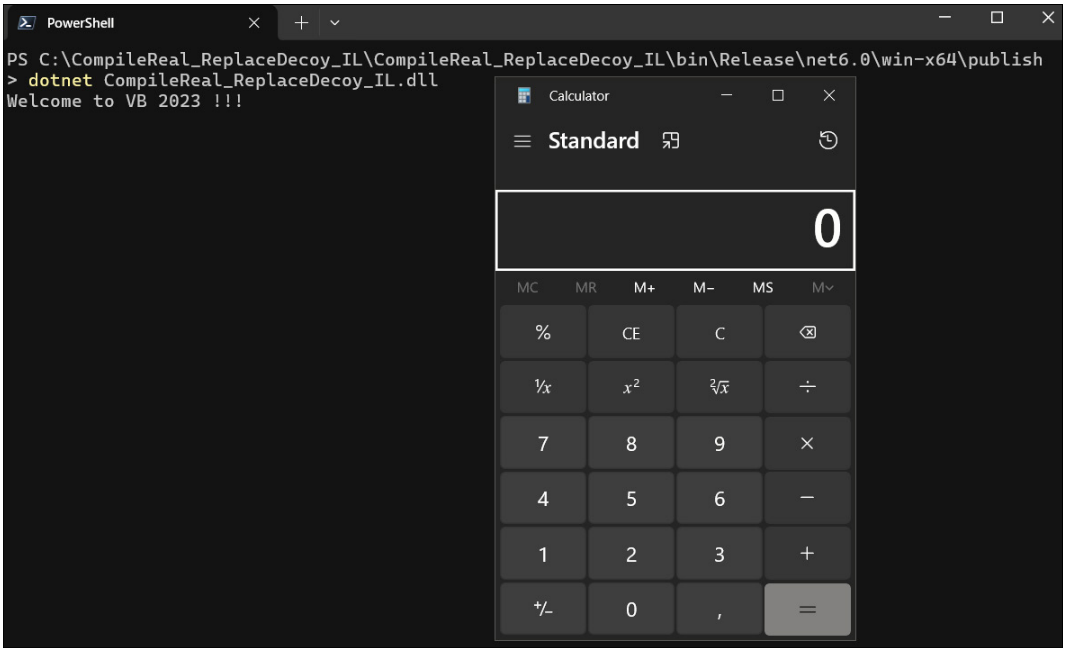


Figure 8: Triggering the execution of the pre-compiled native code.

Programmatic approach using dnlib

Generally, the logic behind the programmatic way of patching is the same as in the case we have already covered using *dnSpyEx*. Since we need a simple solution that is able to preserve not only the original dotnet metadata but also the pre-compiled code and its related structures that are a part of PE, using *dnlib* is probably the most suitable solution. *Dnlib* provides a native writer and its appropriate options that are able to preserve everything we need [10].

The following is example usage of *dnlib* (via PowerShell) to patch the original ReadyToRun application:

```
[Reflection.Assembly]::LoadFrom("C:\dnlib.dll") | Out-Null
$original = "C:\CompileReal_ReplaceDecoy_IL.dll"

$moduleDef = [dnlib.DotNet.ModuleDefMD]::Load($original)
$mainMethod = $moduleDef.Types.Methods.Where{$_.Name -like "Main"}[0]
$inst = $mainMethod.MethodBody.Instructions.Where{$_.Operand.FullName -like
"*Process::Start*"}[0]
$instIndex = $mainMethod.MethodBody.Instructions.IndexOf($inst)
$nopInst = [dnlib.DotNet.Emit.Instruction]::Create([dnlib.DotNet.Emit.OpCodes]::Nop)

$mainMethod.MethodBody.Instructions[$instIndex-1] = $nopInst
$mainMethod.MethodBody.Instructions[$instIndex] = $nopInst
$mainMethod.MethodBody.Instructions[$instIndex+1] = $nopInst

$nativeModuleWriterOptions = [dnlib.DotNet.Writer.
NativeModuleWriterOptions]::new($moduleDef, $true)
$nativeModuleWriterOptions.MetadataOptions.Flags = [dnlib.DotNet.Writer.
MetadataFlags]::PreserveAll
$moduleDef.NativeWrite($original + "_patched.dll", $nativeModuleWriterOptions)
```

Example usage of dnlib (via PowerShell) to patch the original ReadyToRun application:

Compile decoy – replace with real

In this implementation, the target code for a replacement is the pre-compiled native code of the produced assembly, leaving the IL code intact.

We will start with the creation of a new project in Visual Studio IDE, selecting C#, Console App, and building on top of .NET (in our case, .NET 6).

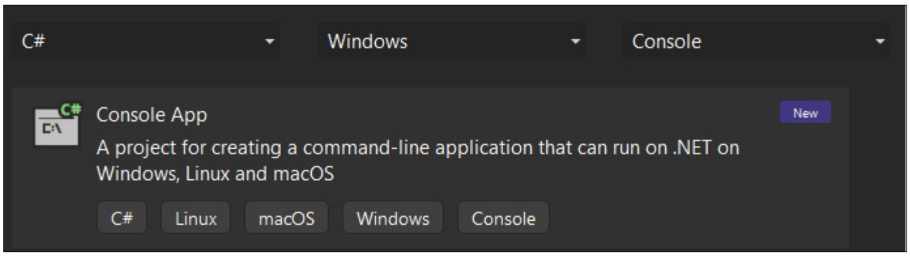


Figure 9: Visual Studio IDE – creation of new C#, Console App, .NET 6 project.

Normally, despite being native, the pre-compiled code of the ReadyToRun application still depends on metadata of the dotnet assembly that needs to be resolved before the code starts executing.

This time, the subject of replacement is the pre-compiled native code, so one of the most suitable solutions could be to replace it with some memory-independent shellcode specific to the targeted OS platform and architecture.

Such an implanted native shellcode will make sure that we are not using any kind of metadata from our targeted dotnet assembly that cannot be resolved. To make our demonstration easy and clear, we can create a decoy C# code that will result in a pre-compiled native code being large enough to make our shellcode fit in easily.

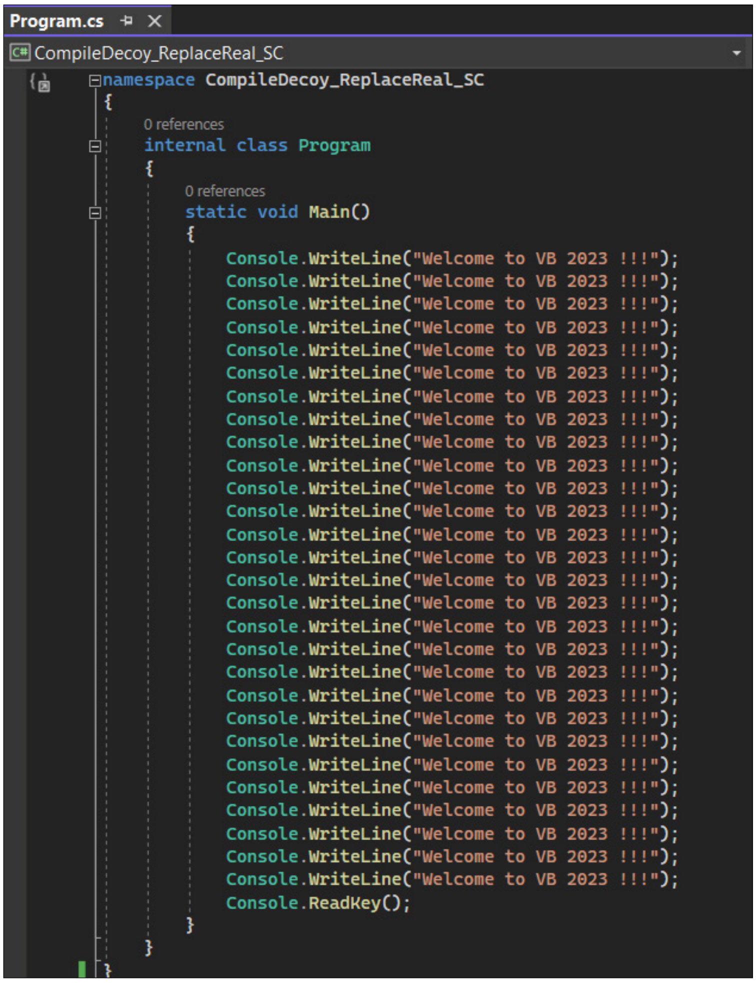
The resulting decoy IL code that will be a part of the produced R2R assembly can be further modified or replaced (we need it just to create space for the shellcode that will be implanted in place of the pre-compiled code).

Figure 10 shows the decoy C# code.

To build our non-self-contained, ReadyToRun application, we can directly specify the 'PublishReadyToRun' flag to the dotnet publish command `dotnet publish -c Release -r win-x64 -p:PublishReadyToRun=true --self-contained false`.

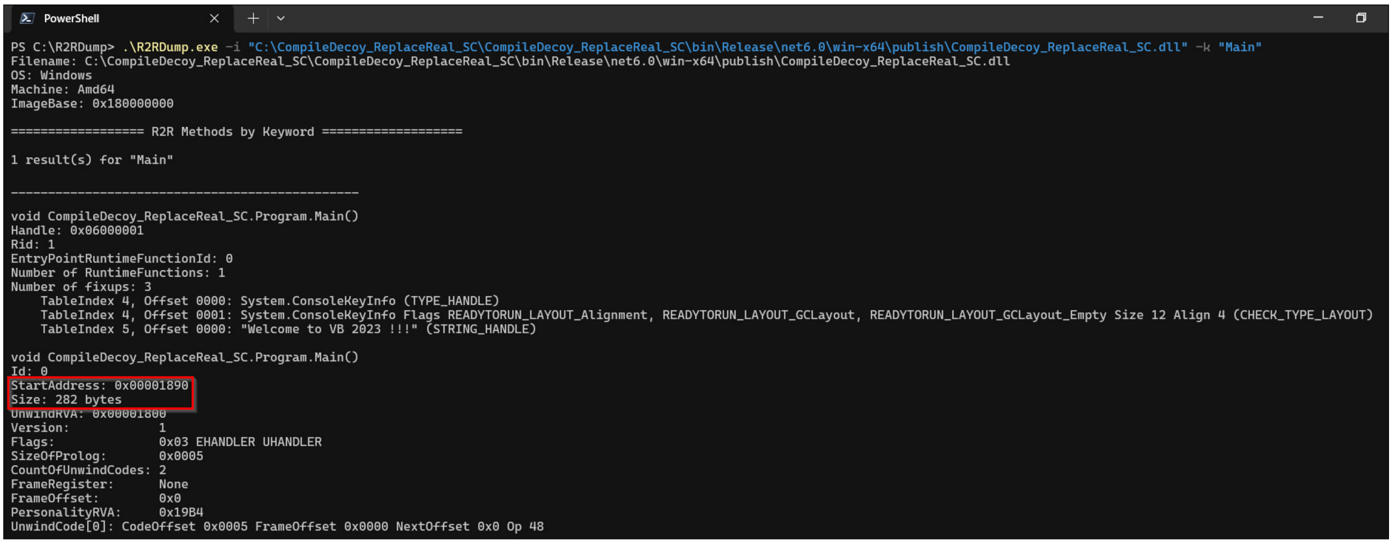
When we have built the ReadyToRun assembly, we need to locate the pre-compiled native code of the `Main()` method, which is a part of this assembly, and find out information about its size. There are several ways to accomplish this, but the most straightforward is to use a tool called *R2RDump* (more about this tool will be covered later) [11].

Figure 11 shows the *R2RDump* tool parsing the structures of the ReadyToRun assembly.



```
Program.cs
CompileDecoy_ReplaceReal_SC
namespace CompileDecoy_ReplaceReal_SC
{
    0 references
    internal class Program
    {
        0 references
        static void Main()
        {
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.WriteLine("Welcome to VB 2023 !!!");
            Console.ReadKey();
        }
    }
}
```

Figure 10: Decoy C# code.



```
PowerShell
PS C:\R2RDump> .\R2RDump.exe -i "C:\CompileDecoy_ReplaceReal_SC\CompileDecoy_ReplaceReal_SC\bin\Release\net6.0\win-x64\publish\CompileDecoy_ReplaceReal_SC.dll" -k "Main"
Filename: C:\CompileDecoy_ReplaceReal_SC\CompileDecoy_ReplaceReal_SC\bin\Release\net6.0\win-x64\publish\CompileDecoy_ReplaceReal_SC.dll
OS: Windows
Machine: Amd64
ImageBase: 0x180000000

================== R2R Methods by Keyword ===================

1 result(s) for "Main"

-----------------------------------------------

void CompileDecoy_ReplaceReal_SC.Program.Main()
Handle: 0x06000001
Rid: 1
EntryPointRuntimeFunctionId: 0
Number of RuntimeFunctions: 1
Number of fixups: 3
    TableIndex 4, Offset 0000: System.ConsoleKeyInfo (TYPE_HANDLE)
    TableIndex 4, Offset 0001: System.ConsoleKeyInfo Flags READYTORUN_LAYOUT_Alignment, READYTORUN_LAYOUT_GCLayout, READYTORUN_LAYOUT_GCLayout_Empty Size 12 Align 4 (CHECK_TYPE_LAYOUT)
    TableIndex 5, Offset 0000: "Welcome to VB 2023 !!!" (STRING_HANDLE)

void CompileDecoy_ReplaceReal_SC.Program.Main()
Id: 0
StartAddress: 0x00001890
Size: 282 bytes
UnwindRVA: 0x00001800
Version:            1
Flags:              0x03 EHANDLER UHANDLER
SizeOfProlog:       0x0005
CountOfUnwindCodes: 2
FrameRegister:      None
FrameOffset:        0x0
PersonalityRVA:     0x19B4
UnwindCode[0]: CodeOffset 0x0005 FrameOffset 0x0000 NextOffset 0x0 Op 48
```

Figure 11: R2RDump tool parsing the structures of ReadyToRun assembly.

We can clearly see that, in this case, the pre-compiled code of the `Main()` method is located on the RVA address 0x00001890 with a size of 282 bytes.

A native disassembler like *IDA* [12] could be used to find and extract 282 opcode bytes of the pre-compiled native code on RVA address 0x00001890. These opcode bytes will serve the purpose of a pattern search during binary patching.

Figure 12 shows the *IDA* disassembler being used to extract the opcode bytes of pre-compiled native code.

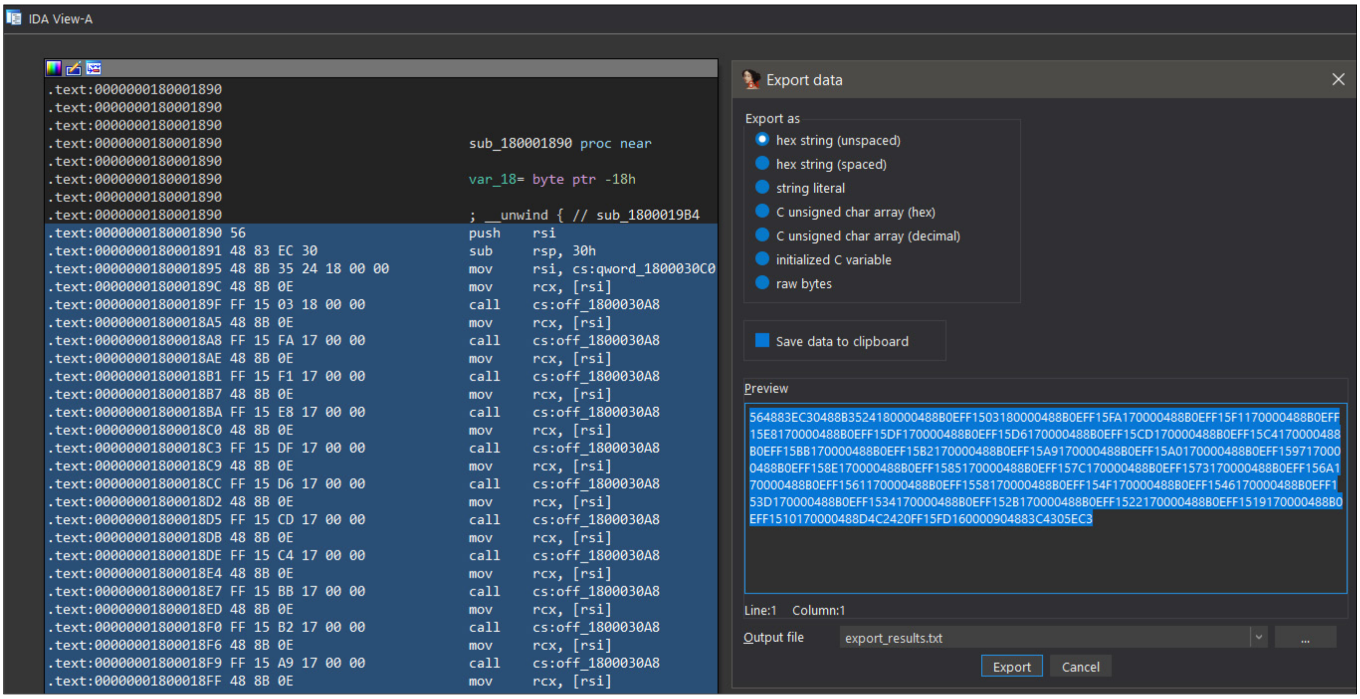


Figure 12: IDA disassembler used to extract the opcode bytes of pre-compiled native code.

To generate an example of memory-independent shellcode that will replace the pre-compiled native code of R2R assembly, MsfVenom (a Metasploit standalone payload generator) could be used [13]. Issuing the command below will result in 282 bytes of 64-bit *Windows* shellcode with the purpose of spawning a new process, `calc.exe`:

```
.\msfvenom.bat -p windows/x64/exec CMD=calc.exe -f raw --smallest --nopsled 6 -o calc.sc
```

Once we have both the opcode bytes pattern of the pre-compiled native code of the assembly and the shellcode, we can use any tool to search for the pattern and perform the raw binary patching. We decided to use the *010 Editor*.

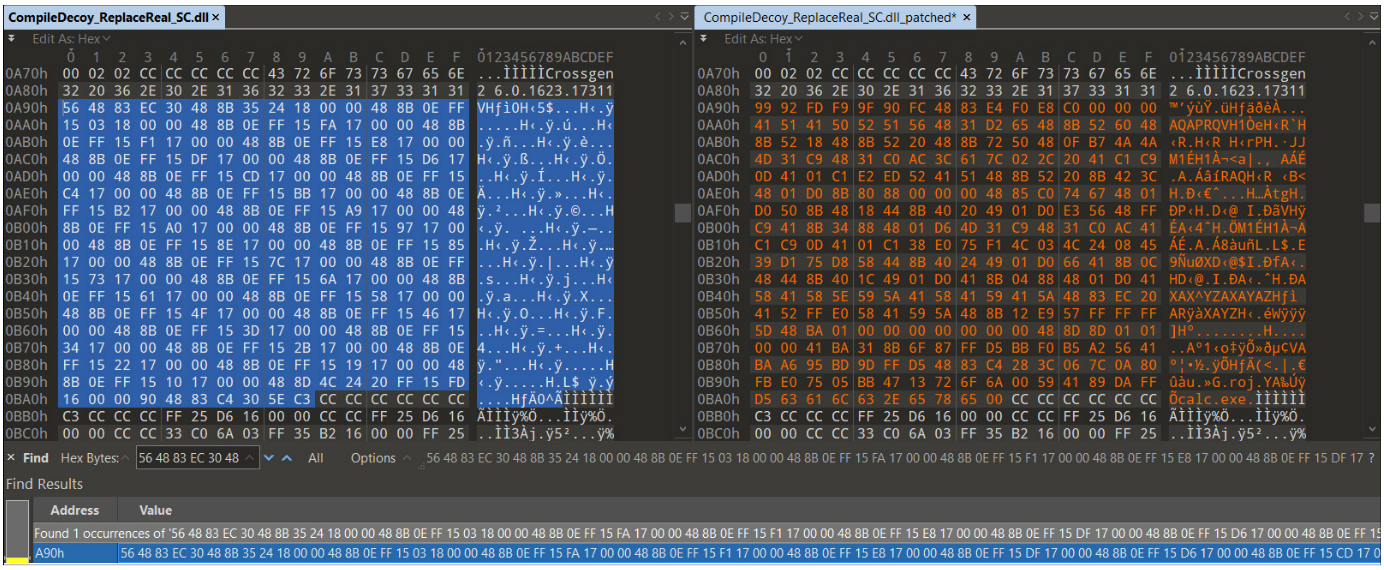


Figure 13: Binary patching using 010 Editor.

If we try to run our ReadyToRun stomped assembly, either via its associated executable `CompileDecoy_ReplaceReal_SC.exe` located in the same folder or via issuing the `dotnet CompileDecoy_ReplaceReal_SC.dll` command from a command prompt, we can spot that our shellcode implanted in the place of the original pre-compiled native code has been executed, ignoring the difference to the original decoy IL code (process `calc.exe` started).

Figure 14 shows the execution of the implanted shellcode being triggered.

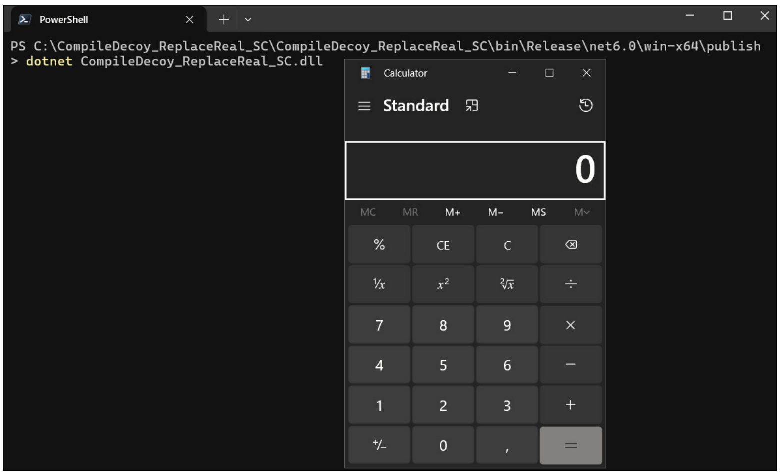


Figure 14: Triggering the execution of the implanted shellcode.

Despite the fully manual method of the above-mentioned implementation, most of the steps can be automated with a programmatic approach.

PROBLEMS AFFECTING REVERSE ENGINEERING

Usually, when it comes to the analysis of dotnet assembly, a significant number of researchers will stay on the level of IL code or interpreted decompiled C# code. To be honest, who would use a tool other than *dnSpy*/*dnSpyEx*?

When it comes to analysis or reverse engineering of R2R stomped assembly, one must go deeper; as we have seen earlier in this paper, the shenanigans are on the level of native code.

The main problems we are dealing with can be summarized as follows:

- We see a different code from the one that is executed (static analysis problems)
- We debug a different code from the one that is executed out of debugger context (dynamic analysis problems)
- Other compilation formats can be applied to complicate the analysis (complicating the analysis)

To cover the problems affecting the work of reverse engineers, we will use the examples of R2R stomped applications we covered in the 'Implementation of R2R stomping' section.

Static analysis problems

When we try to examine the IL code or the interpreted decompiled C# code of the R2R stomped assembly, we will not see any sign of strangeness at first sight.

For example, the R2R stomped program that was replacing/modifying the IL code and leaving the pre-compiled code intact (in the 'Compile real – replace with decoy' section) in *dnSpyEx* is shown in Figure 15.

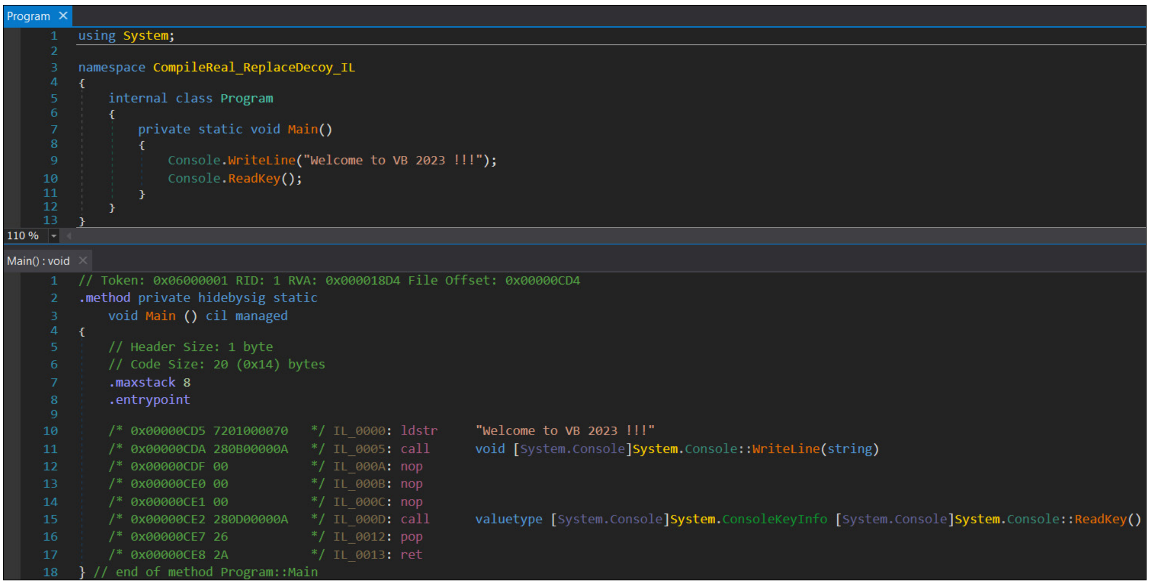


Figure 15: C# view and IL view of the ReadyToRun stomped assembly (pre-compiled code intact).

One could say that the nops instructions look suspicious, but it is important to note that these nops instructions can be removed completely.

Those who are fairly aware of dotnet internals could say that the dotnet metadata related to referenced types are showing types that are not used by the IL code at all (they are still used by the pre-compiled native code that was left intact).

While that is a good point, in a very complicated program where only one of the methods is a target for the R2R stomping, the unused referenced types could easily be overlooked.

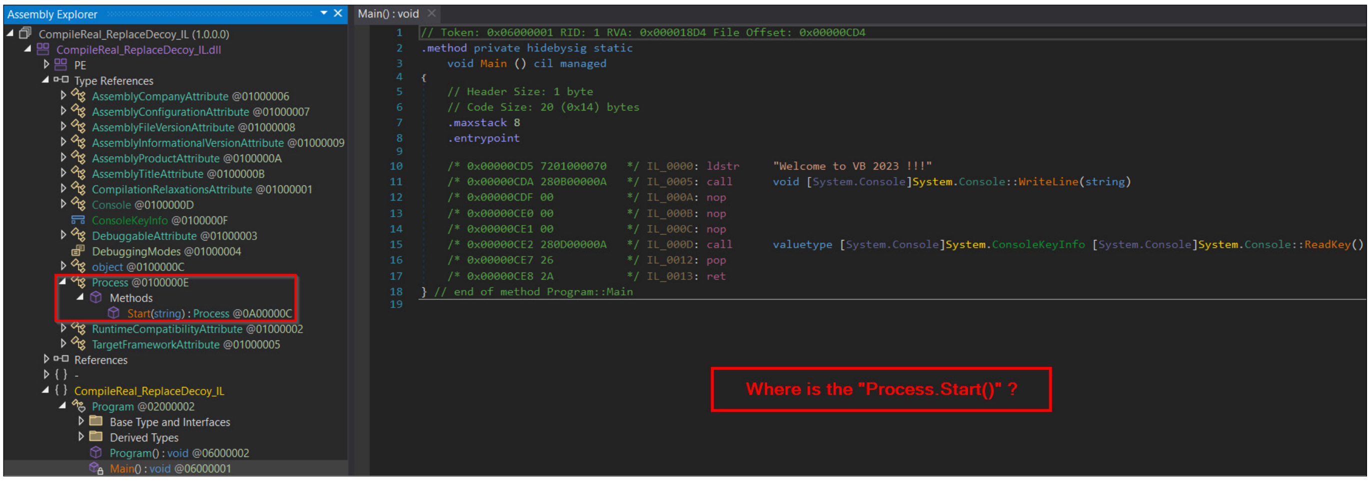


Figure 16: Reference types check of R2R stomped assembly (unused referenced types).

Also, what about the case we saw in the 'Compile decoy – replace with real' section? In that case, we left the original IL code intact and replaced the pre-compiled native code with shellcode, so metadata related to referenced types are accurate.

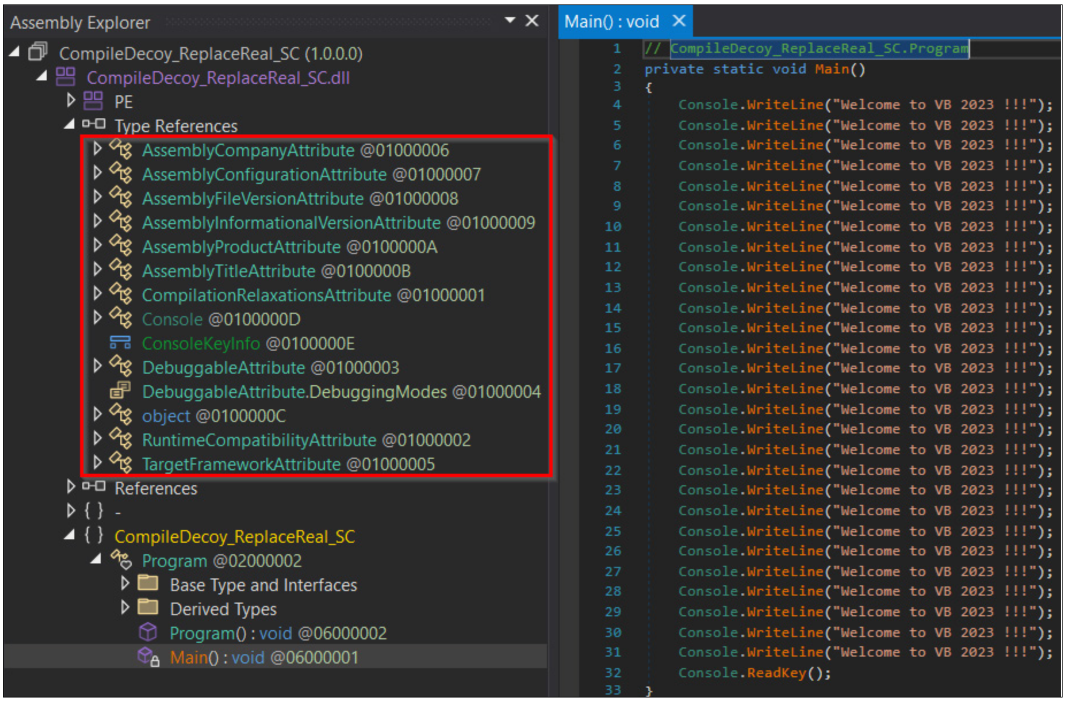


Figure 17: R2R stomped assembly with accurate referenced types.

Dynamic analysis problems – debugging

When it comes to debugging dotnet assemblies, one could hardly imagine using a tool other than *dnspy*/*dnSpyEx* [8].

When we try to run/debug our patched ReadyToRun application in *dnSpyEx*, we will find a different code executing from that which executes in normal execution. This is because the default settings of *dnSpyEx* are suppressing the JIT optimization (to preserve the debugging experience), forcing JIT (Just-In-Time) compilation of the presented IL code, and omitting execution of the pre-compiled native code.

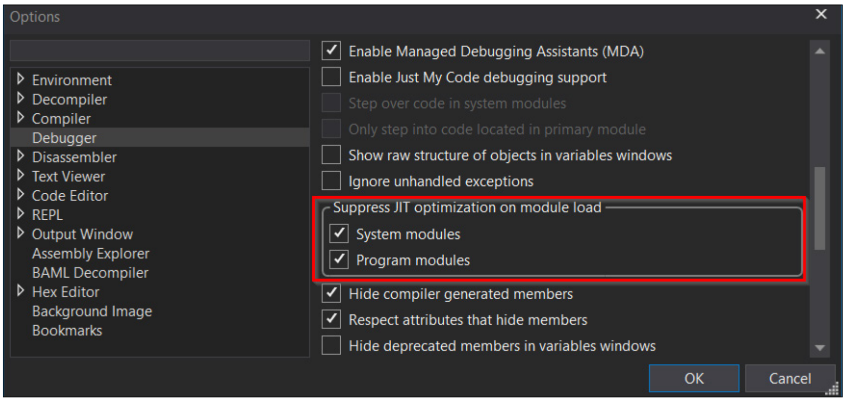


Figure 18: Default dnSpyEx settings - suppressing the JIT optimization.

We immediately notice that, once we try to debug/run the R2R stomped application shown in the 'Compile decoy – replace with real' section (the original IL code intact, the pre-compiled native code replaced with shellcode) in *dnSpyEx*, the `calc.exe` process is not started.

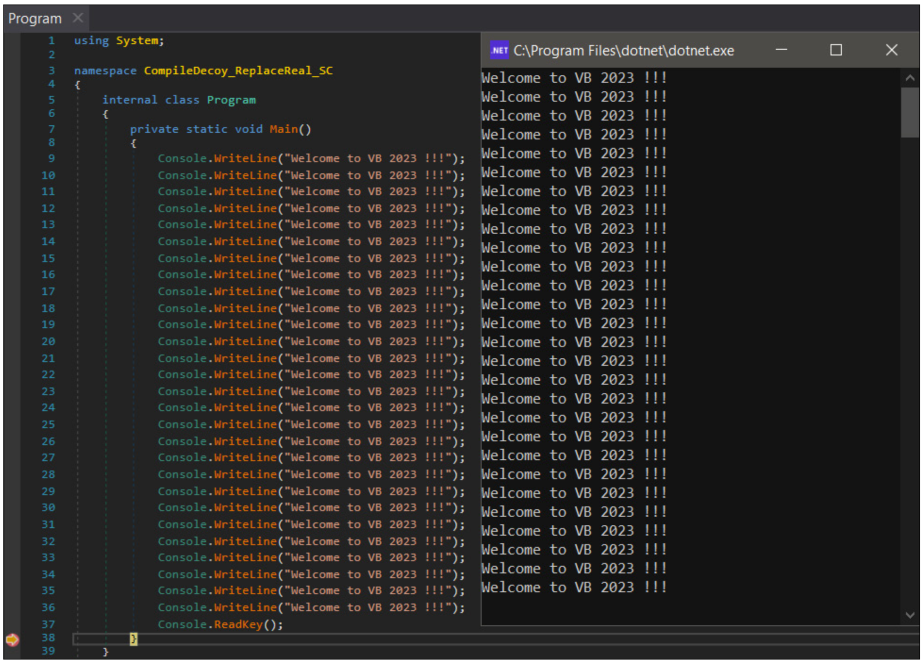


Figure 19: R2R stomped assembly running in the context of dnSpyEx – forced JIT of the IL code.

But once we try to run it out of the debugger context (normal execution), we can see that, because of the .NET optimization, the shellcode (implanted in place of the original pre-compiled native code) is prioritized and executed.

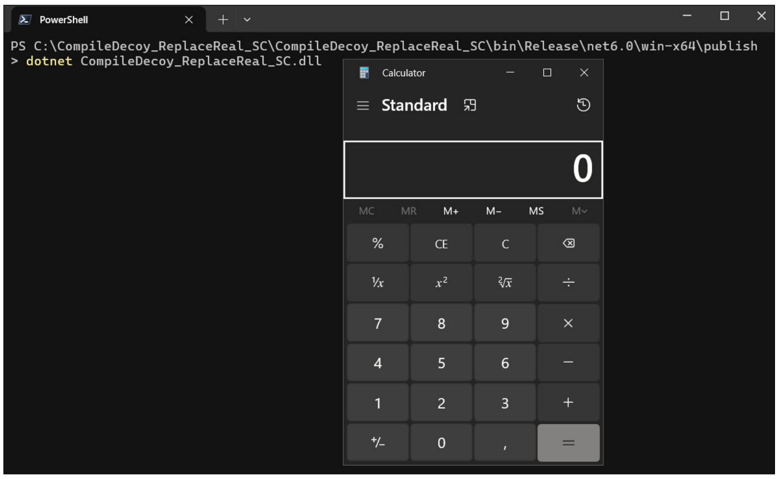


Figure 20: Triggering execution of the implanted shellcode – normal execution.

Because of the debugging experience, the suppression of JIT optimization is quite the expected setting. As a point of interest, we can replicate the behaviour of *dnSpyEx* default settings, effectively turning off AOT optimization, in normal execution. This can be accomplished by setting our targeted process's environment variable `COMPlus_ReadyToRun=0`.

The normal execution without and with setting the environment variable `COMPlus_ReadyToRun=0` can be seen below.

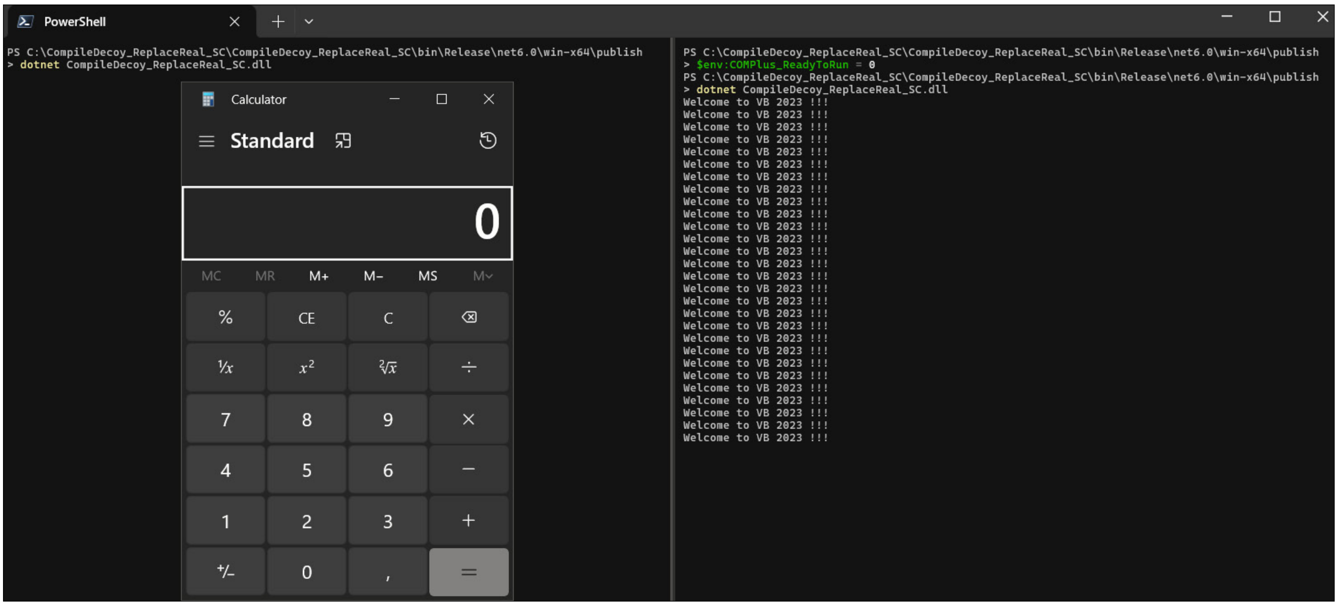


Figure 21: Normal execution of R2R stomped assembly without and with the setting of 'COMPlus_ReadyToRun=0'.

Further complicating the analysis

Different compiler settings can be applied to complicate the analysis of the R2R stomped assembly, resulting in different compilation formats of the produced ReadyToRun application.

An example of such compiler settings could be a combination of dotnet bundle file format (single-file) and self-contained options [6].

These settings could result in one native executable (because of the single-file compiler option) that contains the dotnet assemblies in its overlay. In addition to our main module, a significant part of the dotnet assemblies could represent a targeted dotnet runtime that was bundled into the single-file format (because of the self-contained option).

When dealing with such a program, we are struggling with the same issues as covered before, but also with the problem of detecting this form of compilation and extraction of the assemblies from the overlay of the dotnet bundle (single-file).

Even though these compilation formats are out of the scope of this paper (not directly related to R2R stomping), the extraction of dotnet assemblies from the dotnet bundle overlay (single-file) can be accomplished by using the appropriate tools that understand the dotnet bundle file format, either via GUI-based tools such as *ILSpy* [14] or *dotPeek* [15] or via a programmatic approach using *AsmResolver*.

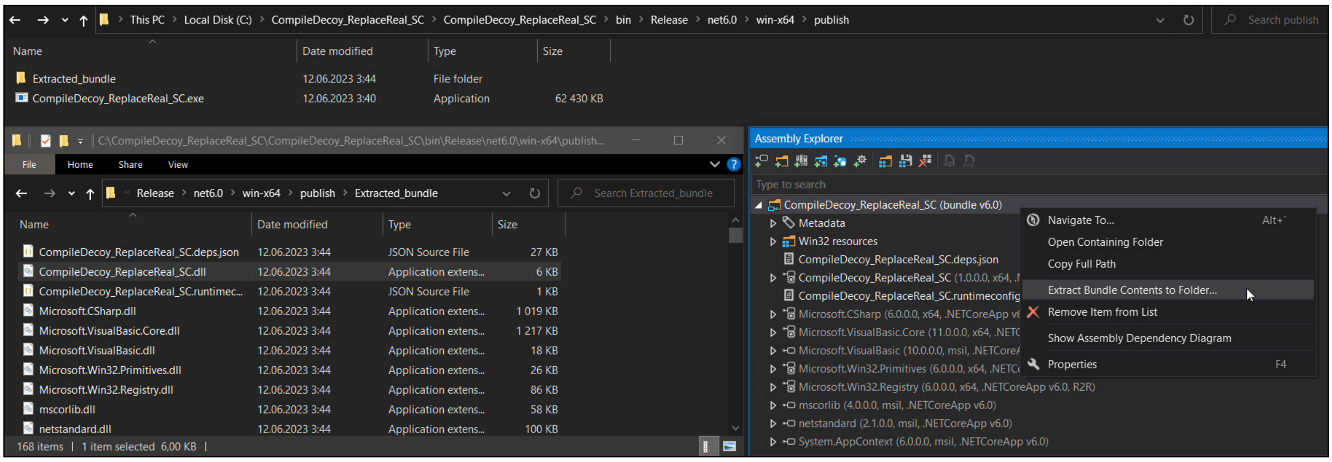


Figure 22: Extraction of dotnet bundle in the dotPeek tool.

TECHNIQUES AND TOOLS TO REVERSE ENGINEER R2R STOMPED ASSEMBLIES

The analysis and reverse engineering of R2R stomped assemblies require a different approach to the one we are used to going with when it comes to ordinary dotnet assembly. We need a different toolset to analyse the parts of ReadyToRun assembly related to AOT compilation and its result. Unfortunately, there is no 'one-size-fits-all' solution, but several tools are very helpful for particular tasks.

In general, these tasks can be divided into:

- Parsing the ReadyToRun assembly structure (*R2RDump*, *dotPeek*)
- Showing the IL code and interpreted decompiled C# code (*ILSpy*, *dnSpyEx*, *dotPeek*)
- Locating and disassembling the pre-compiled native code (*R2RDump*, *ILSpy*)

To demonstrate the use of a specific tool regarding a particular task, the R2R stomped application outlined in the 'Compile decoy – replace with real' section (replacement of the pre-compiled native code, leaving the original IL code) was chosen.

Parsing the ReadyToRun assembly structure

Proper parsing of the R2R assembly is crucial as related structures provide important information that helps with analysis and reverse engineering. An example of information we can obtain is a list of methods that were pre-compiled to their native form, enriched with details about location and size.

Among the most reliable tools that understand the R2R assembly structure, parse it, and can present this information meaningfully, are *R2RDump* and *dotPeek*.

R2RDump is a command-line utility, and part of its dotnet runtime source code is available on its *GitHub* repository [11]. This tool is not a part of the dotnet runtime installer, so if we need to get it, we must compile it on our own. The maintenance of this tool is regular, and because of that, it can provide the most comprehensive information about ReadyToRun assemblies.

The available options for the *R2RDump* tool are shown in Figure 23.

```
PS C:\R2RDump> .\R2RDump.exe -h
Description:
  Parses and outputs the contents of a ReadyToRun image

Usage:
  R2RDump [options]

Options:
  -i, --in <in>                                     Input file(s) to dump. Expects them to by ReadyToRun images
  -o, --out <out>                                   Output file path. Dumps everything to the specified file except for help message and exception messages
  --raw                                             Dump the raw bytes of each section or runtime function
  --header                                          Dump R2R header
  -d, --disasm                                      Show disassembly of methods or runtime functions
  --naked                                           Naked dump suppresses most compilation details like placement addresses
  --hide-offsets, --ho                              Hide offsets in naked disassembly
  -q, --query <query>                               Query method by exact name, signature, row ID or token
  -k, --keyword <keyword>                           Search method by keyword
  -f, --runtimefunction <runtimefunction>           Get one runtime function by id or relative virtual address
  -s, --section <section>                           Get section by keyword
  --unwind                                          Dump unwindInfo
  --gc                                              Dump gcInfo and slot table
  --pgo                                             Dump embedded pgo instrumentation data
  --sc, --sectionContents                           Dump section contents
  -e, --entrypoints                                 Dump list of method / instance entrypoints in the R2R file
  -n, --normalize                                   Normalize dump by sorting the various tables and methods (default = unsorted i.e. file order)
  --hide-transitions, --ht                          Don't include GC transitions in disassembly output
  -v, --verbose                                     Dump disassembly, unwindInfo, gcInfo and sectionContents
  --diff                                            Compare two R2R images
  --diff-hide-same-disasm                           In matching method diff dump, hide functions with identical disassembly
  --create-pdb                                      Create PDB
  --pdb-path <pdb-path>                             PDB output path for --create-pdb
  --create-perfmap                                  Create PerfMap
  --perfmap-path <perfmap-path>                     PerfMap output path for --create-perfmap
  --perfmap-format-version <perfmap-format-version> PerfMap format version for --create-perfmap [default: 1]
  -r, --reference <reference>                       Explicit reference assembly files
  --referencePath, --rp <referencePath>             Search paths for reference assemblies
  --sb, --signatureBinary                           Append signature binary to its textual representation
  --inlineSignatureBinary, --isb                    Embed binary signature into its textual representation
  -?, -h, --help                                    Show help and usage information
```

Figure 23: Available options for the R2RDump tool.

An example of *R2RDump* usage that provides information about the R2R header and content of each presented section is shown in Figure 24.

If one would prefer a GUI-based tool, *dotPeek* is the one to go with. Despite the fact that it cannot provide as detailed information as *R2RDump*, it can be considered a suitable alternative.

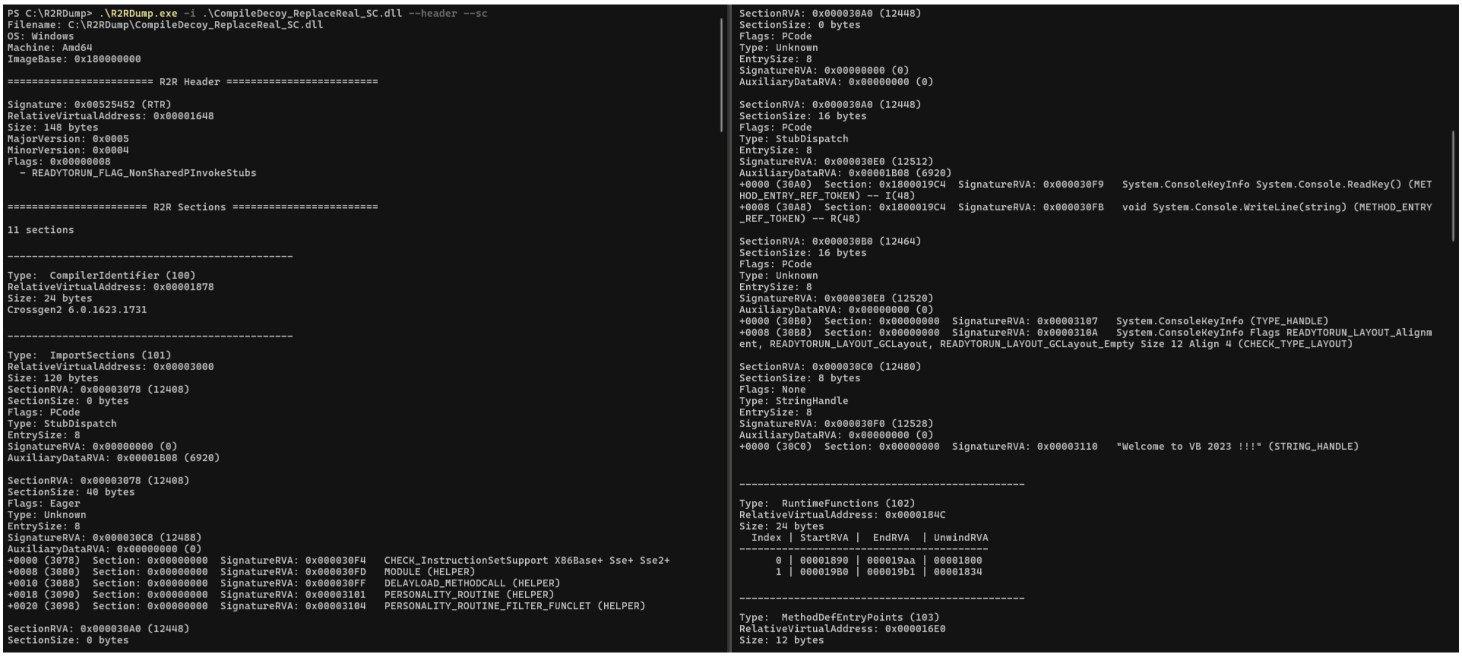


Figure 24: Parsing R2R header and content of sections with the R2RDump tool.

Showing the IL code and interpreted decompiled C# code

As we described earlier, with abusing of R2R stomping, certain IL code or the pre-compiled native code is modified. To be able to see the IL code of such methods is another important part of the analysis.

Most researchers are already aware of tools like *dnSpyEx*, *ILSpy* and *dotPeek* that have the ability to show the IL code and its reconstructed decompiled C# code. This task is probably the only one that is common when analysing an ordinary dotnet assembly.

The engine from *ILSpy* is running under the hood of the *dnSpyEx* tool to reconstruct both the IL code and decompiled C# code. An example of both of these views side-by-side can be seen in Figure 25.

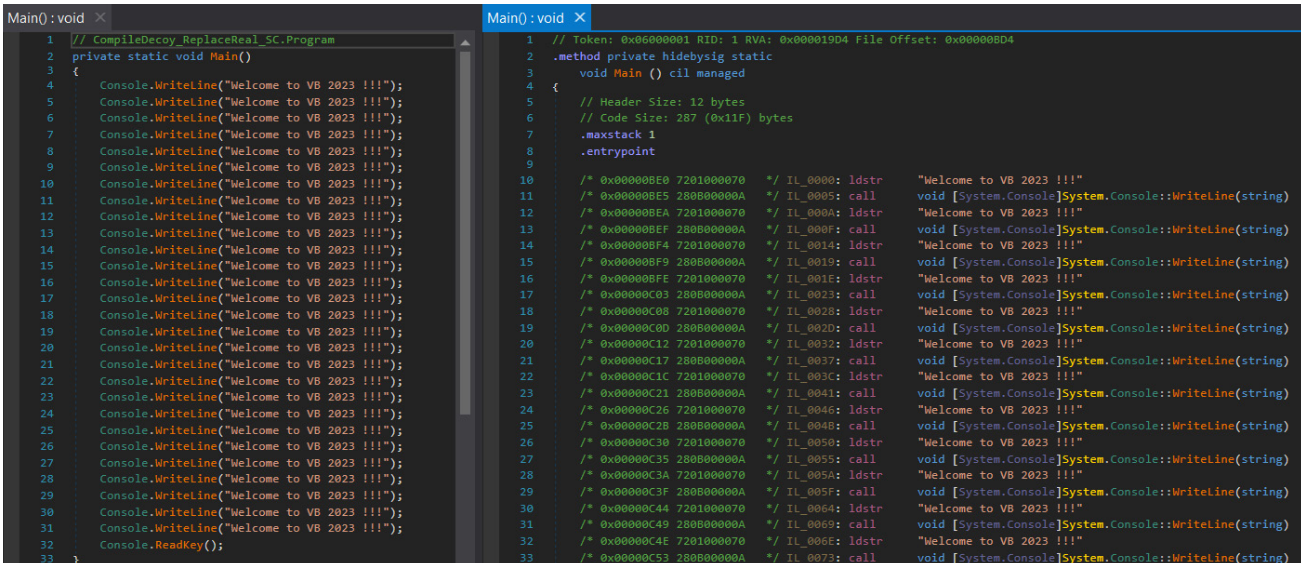


Figure 25: IL and C# code views in the dnSpyEx tool.

Locating and disassembling the pre-compiled native code

The last but most important part of the analysis regarding R2R stomping is being able to locate and see the disassembly of methods that were pre-compiled to their native form.

When it comes to this task, a limited number of tools can be used. Such tools need to understand the R2R assembly structure and must be able to properly parse it to use certain information that can later serve to locate and process the pre-compiled native code and present it in its disassembly form. The most useful tools that can be used to accomplish this task are *R2RDump* and *ILSpy*.

We have already mentioned the *R2RDump* tool, but we did not cover its ability to reconstruct and present the disassembly of certain methods that were pre-compiled to their native form. An example of using this tool to do so can be seen in Figure 26 below (showing the disassembly of R2R stomped assembly, `Main` method).

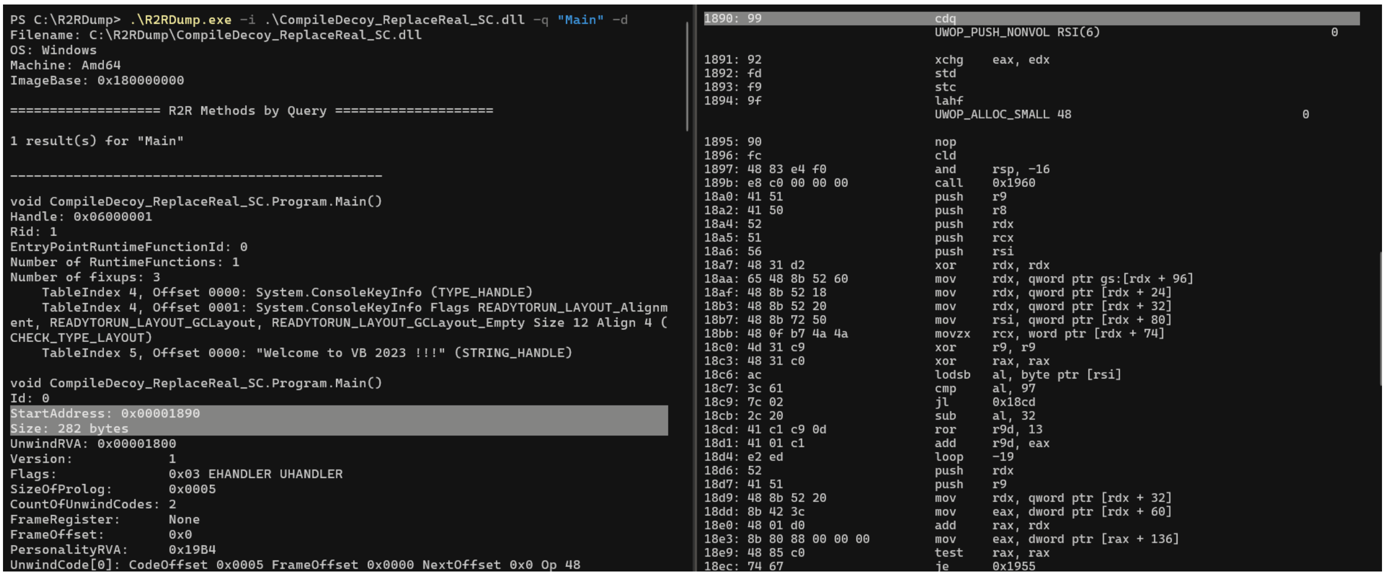


Figure 26: Using the R2RDump to show the disassembly of the 'Main' method.

ILSpy is an industry-changing tool regarding dotnet analysis. It is not so well known, but it also understands the R2R assembly format well enough to be able to interpret the disassembly code of pre-compiled methods. By selecting a method that was pre-compiled to native code and switching the view to one named 'ReadyToRun', we can investigate the disassembly associated with the selected method.

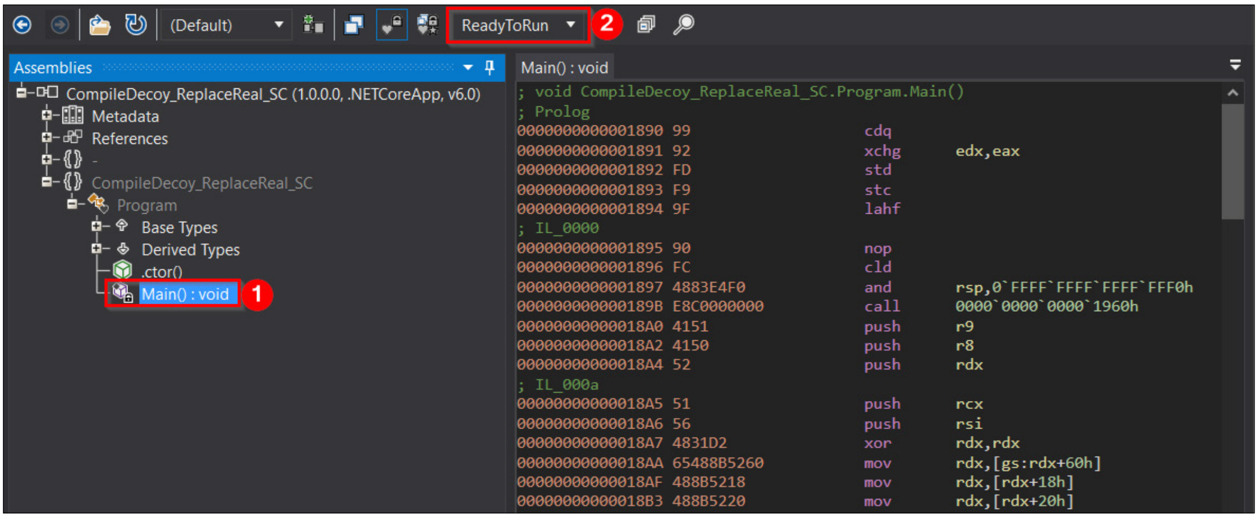


Figure 27: Using ILSpy to show the disassembly of the R2R stomped method.

DETECTING R2R STOMPING

Before we jump to possible ways of detecting the R2R stomping technique, we need to start with a general detection of the ReadyToRun form of compilation. Recognizing this kind of format with a manual or automated approach is a relatively easy task.

For the manual approach to R2R format detection, tools like *dotPeek* or *ILSpy* should be our first choice because they tell us immediately what we are dealing with. As they even understand the dotnet bundle file format, there is no problem if such an option was set during the compilation of the R2R application (they can extract the content of the bundle).

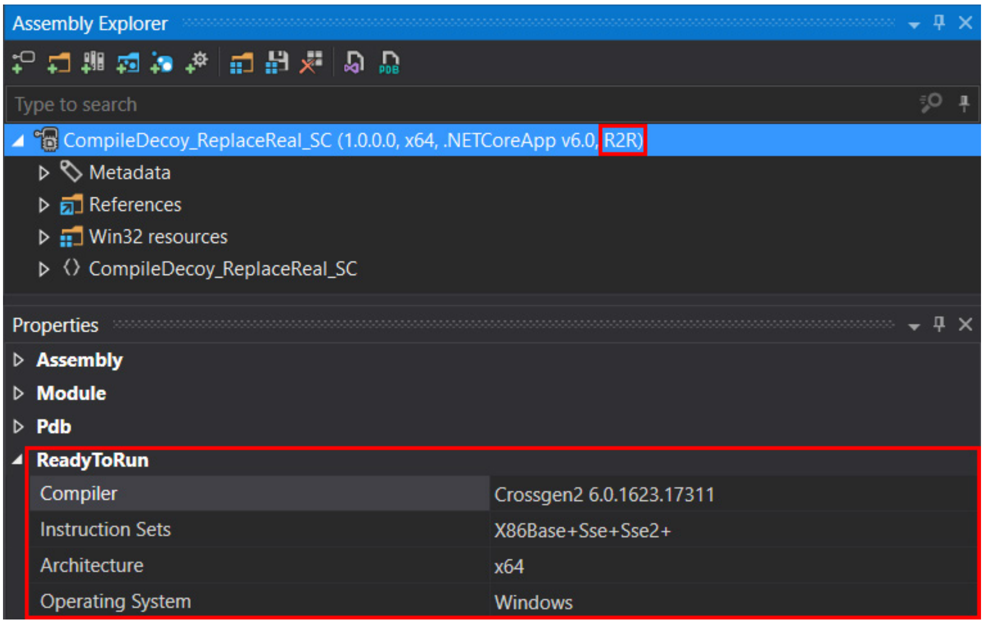


Figure 28: Detection of R2R assembly in the dotPeek tool.

The ReadyToRun compiled binaries enrich the CLI file format with a 'ManagedNativeHeader' pointing to a specific 'READYTORUN_HEADER'. The signature field of 'READYTORUN_HEADER' is always set to 0x00525452 (ASCII encoding for 'RTR'). The RVA address and size of 'ManagedNativeHeader' are a part of the .NET Directory. All these findings can be used to create an effective YARA rule [16] that can be used for automated detection of the ReadyToRun dotnet format. An example of such a YARA rule is shown below.

```
import "pe"

rule r2r_assembly
{
   meta:
      author = "jiriv"
      description = "Detects dotnet binary compiled as ReadyToRun - form of ahead-of-time 
(AOT) compilation"
   condition:
      // check if valid PE
      uint16(0) == 0x5a4d and uint16(uint32(0x3c)) == 0x4550 and
      // check if dotnet -> .NET Directory is present
      pe.data_directories[pe.IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR].virtual_address != 0 and
      // check if ManagedNativeHeader exists -> ManagedNativeHeader RVA is not 0 inside .NET 
Directory
      uint32(pe.rva_to_offset(pe.data_directories[pe.IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR].
virtual_address) + 0x40) != 0 and
      // check if it is R2R -> RTR magic signature is present (0x00525452 == "RTR" in ascii)
      uint32(pe.rva_to_offset(uint32(pe.rva_to_offset(pe.data_directories[pe.IMAGE_DIRECTORY_
ENTRY_COM_DESCRIPTOR].virtual_address) + 0x40))) == 0x00525452
}
```

Generally, the manual detection of R2R stomping is based on an investigation of the difference between the method's IL code and its appropriate pre-compiled native code.

We mentioned earlier that no tool could be considered an 'all-in-one' solution for analysing and detecting R2R stomping, but *ILSpy* is very likely the closest to it [14]. *ILSpy* understands the R2R format and is able to show us the IL code, interpreted decompiled C# code, and even the disassembly of the pre-compiled native code. Furthermore, it can deal with other compilation formats such as bundle (single-file) and self-contained dotnet. With all of these capabilities, it became the main utility for manual detection and analysis of R2R stomping. It is worth noting that even though the *ILSpy* engine runs under the hood of *dnSpyEx*, the above-mentioned features are not implemented.

An example of manual detection of R2R stomping using *ILSpy* can be seen in Figure 29 below, where we use the application outlined in the 'Compile decoy – replace with real' section (replacement of the pre-compiled native code, leaving the original IL code).

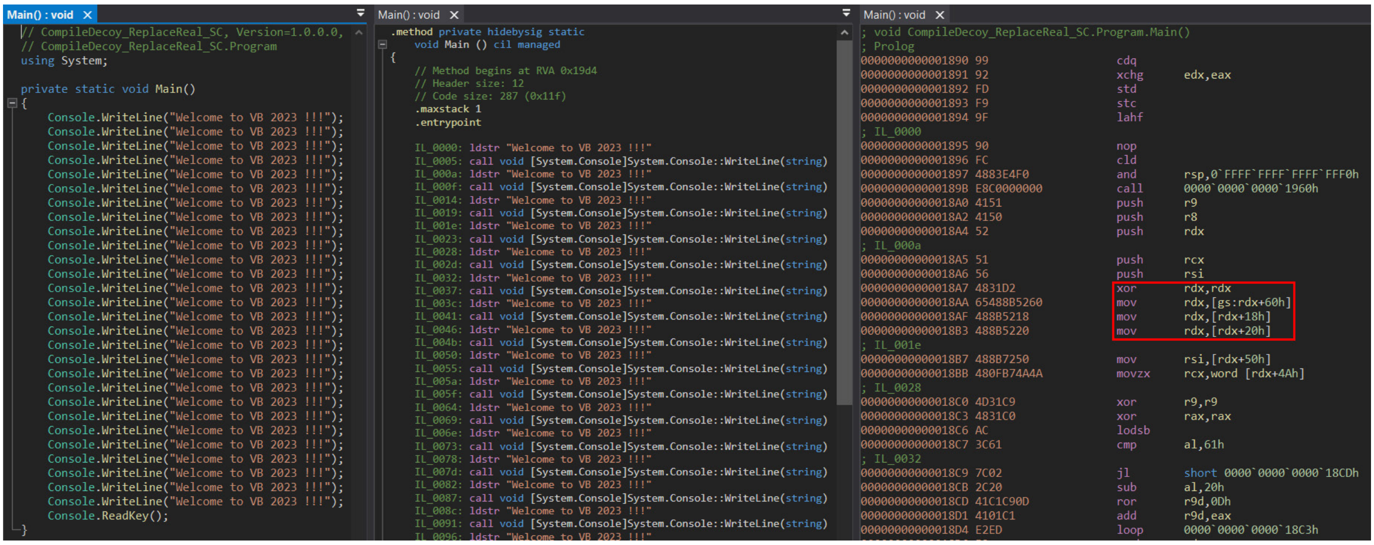


Figure 29: R2R stomping – implanted shellcode.

With the side-by-side views, we can immediately see that something is really wrong with the pre-compiled native code of the `Main` method. One could hardly imagine a situation where the pre-compiled code would result in something lacking a typical function prologue and even manipulating with PEB structure (Process Environment Block). We would expect something like that shown in Figure 30 below (the original, not stomped R2R assembly).

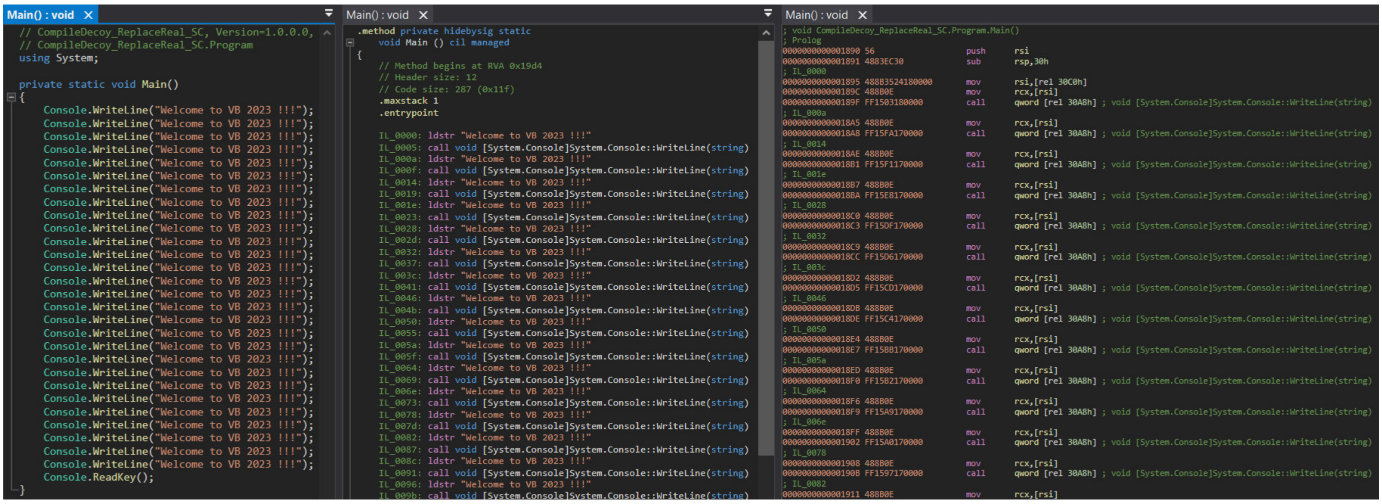


Figure 30: The original, not stomped R2R assembly.

When it comes to manual detection of R2R stomping regarding our second example application described in the 'Compile real – replace with decoy' section (replacement of the compiled IL code, leaving the original pre-compiled code), we can spot relatively easily the missing reference to the `Process.Start()` method in IL and the C# code view.

Of course, the more complicated programs we have, the harder it will be to reveal the R2R stomping technique. The manual approach will always be time-consuming, but in most cases, the most reliable way to reveal R2R assemblies affected by stomping.

If we want to try to automate the detection of R2R stomping, no simple and 100% reliable solution is ready for production. As we have already seen, the logic behind the R2R stomping detection needs to cover several different scenarios. We have covered implanted shellcode and modified IL code with decoy instructions, but there is always space for other imagination.

One can hardly think about the implementation of such detection logic with just some signature-based solution, like YARA.

The most promising solution would be using a programmatic approach with the help of libraries (e.g. *dnlib*, *AsmResolver*, *iced* [14]) that understand the dotnet assembly structure, metadata, IL code, and are also able to disassemble the

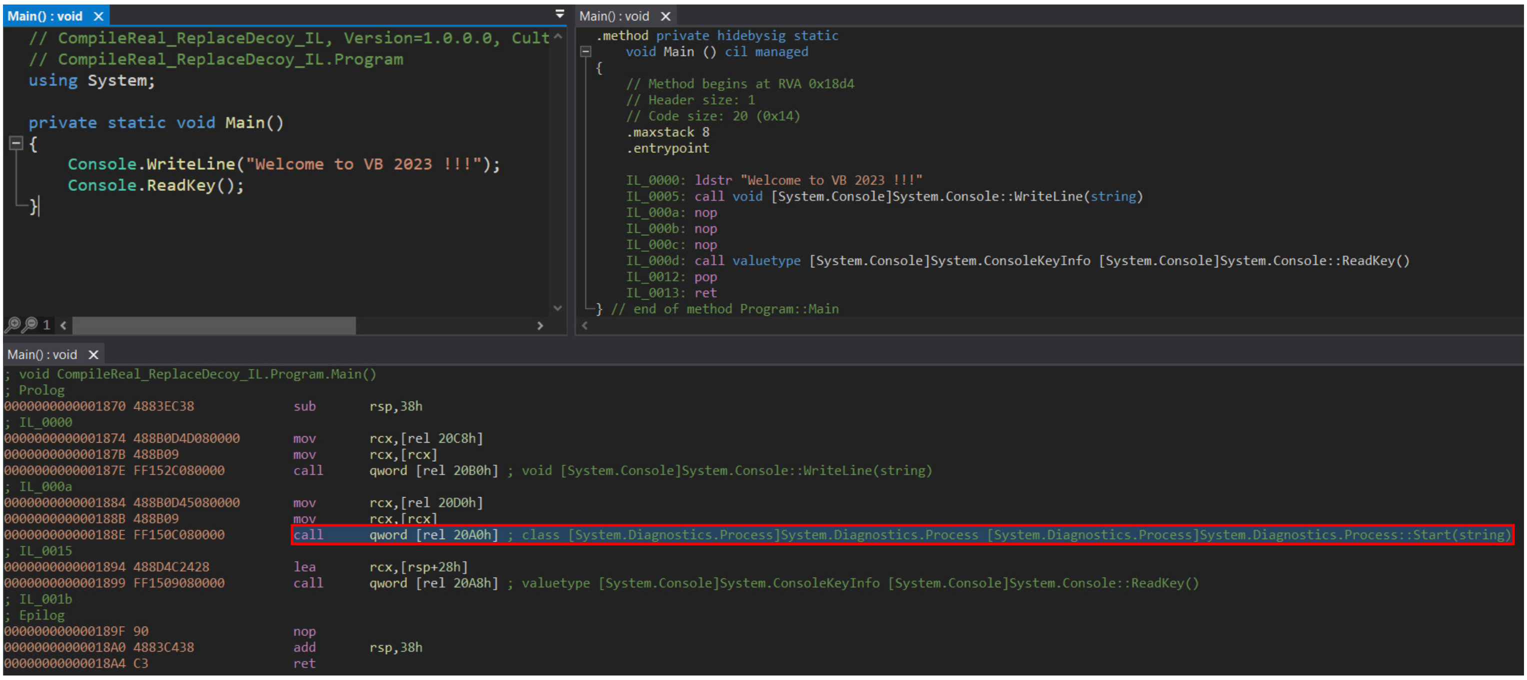


Figure 31: R2R stomping – patched IL code (pre-compiled intact).

pre-compiled native code. This would be as reliable as our implemented logic that would need to predicate how the resulting pre-compiled code of methods should look across all different platforms and architectures.

This is an example of a case where prevention would be a much more reliable and easy-to-implement solution. If we thought about some computed hash of the IL code and its pre-compiled code that would be added to the R2R assembly structure and verified upon execution by dotnet runtime, there would be no R2R stomping (until next time – R2R hash stomping).

CONCLUSION

This paper has introduced a new method for running hidden implanted code in ReadyToRun (R2R) compiled .NET binaries, R2R stomping. We have covered its implementation details, focusing on the internal processing of dotnet runtime and resulting problems that harden reverse engineering. In the final sections, we introduced several tools and techniques that can be effective and useful for the analysis of R2R stomped applications and described how to use them for detection.

Despite the fact that there is no static, automated detection mechanism ready for production yet, in the case of implanting a malicious code via the R2R stomping technique, the behavioural-based detection should not be affected. R2R stomping could affect the work of researchers, but it is not an evasion technique. As of now, we have not found any evidence of use of R2R stomping in the wild, but we cannot exclude the possibility of it already being part of some advanced arsenals.

REFERENCES

[1] NGEN. https://learn.microsoft.com/en-us/dotnet/framework/tools/ngen-exe-native-image-generator.

[2] Native AOT. https://learn.microsoft.com/en-us/dotnet/core/deploying/native-aot/?tabs=net7.

[3] ReadyToRun Compilation. https://learn.microsoft.com/en-us/dotnet/core/deploying/ready-to-run.

[4] ECMA-335. https://www.ecma-international.org/publications-and-standards/standards/ecma-335/.

[5] ReadyToRun File Format. https://github.com/dotnet/runtime/blob/main/docs/design/coreclr/botr/readytorun-format.md.

[6] Single-file deployment. https://learn.microsoft.com/en-us/dotnet/core/deploying/single-file/.

[7] Visual Studio IDE. https://visualstudio.microsoft.com/.

[8] DnSpy/DnSpyEx. https://github.com/dnSpyEx/dnSpy.

[9] AsmResolver. https://github.com/Washi1337/AsmResolver.

[10] Dnlib. https://github.com/0xd4d/dnlib.

[11] R2RDump. https://github.com/dotnet/runtime/tree/main/src/coreclr/tools/r2rdump.

[12] IDA (Hex-Rays). https://hex-rays.com/ida-pro/.

[13] MsfVenom. https://docs.metasploit.com/docs/using-metasploit/basics/how-to-use-msfvenom.html.

[14] ILSpy. https://github.com/icsharpcode/ILSpy.

[15] DotPeek. https://www.jetbrains.com/decompiler/.

[16] YARA. https://github.com/VirusTotal/yara.

[17] Iced. https://github.com/icedland/iced.

[18] ReadyToRun Overview. https://github.com/dotnet/runtime/blob/main/docs/design/coreclr/botr/readytorun-overview.md.