

4 - 6 October, 2023 / London, United Kingdom

R2R STOMPING – ARE YOU READY TO RUN?

Jiří Vinopal <u>Check Point Research, Czechia</u>

jiriv@checkpoint.com

www.virusbulletin.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].

HelloWorld (1.0.0.0, x64, .NETCoreApp v6.0, R2R)
A 🗞 Metadata
⊿ 🖬 Headers
DOS header
▶ 🖥 NT headers
Isection headers
Directories
▶ 🔁 CLI (Cor20) header
🔺 🌇 ReadyToRun header
0x000007A8 Signature: 0x00525452 (DWord) Should be 0x00525452 (ASCII encoding for RTR)
Øx000007AC MajorVersion: 0x0005 (Word) Major version number of the ReadyToRun image format
0x000007AE MinorVersion: 0x0004 (Word) Minor version number of the ReadyToRun image format
🔺 🔁 ReadyToRun Core header
Ø 0x00000780 Flags: 0x00000008 (DWord) NONSHARED_PINVOKE_STUBS
Øx000007B4 NumberOfSections: 0x0000000B (DWord) Number of ReadyToRun Sections that follows ReadyToRun Core Header
CompilerIdentifier Crossgen2 6.0.1423.7309
▶ 🔁 ImportSections X86Base+Sse+Sse2+
RuntimeFunctions
MethodDefEntryPoints
▶ 🔁 DebugInfo
DelayLoadMethodCallThunks
▷ 🖺 AvailableTypes
▶ 🖺 InstanceMethodEntryPoints
▷ 🖺 ManifestMetadata
> 🔁 InliningInfo2
▶ 🔁 ManifestAssemblyMvids
Metadata header

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).

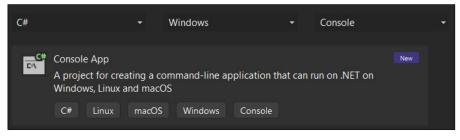


Figure 2: Visual Studio IDE – creation of new C#, Console App, .NET 6 project.

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.

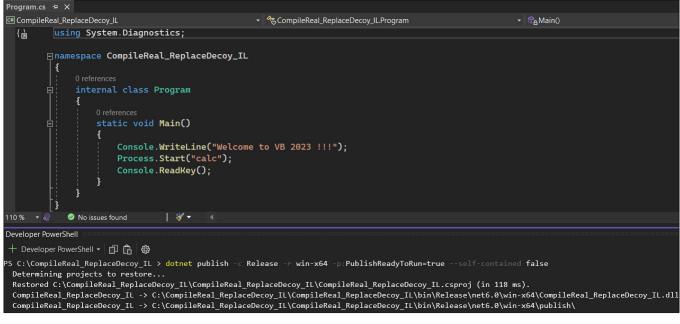


Figure 3: Building the ReadyToRun application with the dotnet publish command.

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).

Assembly Explorer 🔹 🗙	Program $ imes$			
 CompileReal_ReplaceDecoy_IL (1.0.0.0) E CompileReal_ReplaceDecoy_IL.dll 		using System; using System.Diagnostics	;	
 ▷ 몸 PE ▷ □ Type References ▷ □ References ▷ { } - ▲ { } CompileReal_ReplaceDecoy_IL ▲ Program @02000002 ▷ Base Type and Interfaces ▷ Derived Types ♥ Program(): void @06000002 ♥ Main(): void @06000001 	3 4 5 6 7 8 9 10 11 12 13 14 15	<pre>namespace CompileReal_Re { internal class Progr { private static v { Console.Writ Process.Star Console.Read } } }</pre>	ram void Main() :eLine("Welc rt("calc");	IL ome to VB 2023 !!!");
📙 🛛 💆 📒 🗧 🛛 C:\CompileReal_ReplaceDecoy_IL\CompileReal_	ReplaceDecoy_I	L\bin\Release\net6.0\win-x64\publish		- 🗆 ×
File Home Share View				~ ?
← → v ↑ 📕 « CompileReal_ReplaceDecoy_IL > bin 3	Release > ne	et6.0 > win-x64 > publish	~ U	✓ Search publish
Name	Date modi	fied Type	Size	
CompileReal_ReplaceDecoy_IL.deps.json	09.06.2023	12:14 JSON Source File	1 KB	
S CompileReal_ReplaceDecoy_IL.dll	09.06.2023	12:14 Application extension	6 KB	
CompileReal_ReplaceDecoy_IL.exe	09.06.2023	12:14 Application	145 KB	
CompileReal_ReplaceDecoy_IL.runtimeconfig.json	09.06.2023	12:14 JSON Source File	1 KB	

Figure 4: DnSpyEx: opening ReadyToRun assembly.

Edit Method Boo				×
Instructions L	ocals Exception Handlers	Before		
Body Type IL		- Code Type IL		
Keep Old I	MaxStack ✔ Init Locals Header RVA 0x18D4 Heade	r Offset 0xAD4	MaxStack 8	LocalVarSigTok 0
Index Offset	OpCode Operand			
	ldstr "Welcome to VB 2023 !!!"			
	call void [System.Console]System.Console::WriteLi			
	call class [System.Diagnostics.Process]System.Dia	gnostics.Process [System.	Diagnostics.Process]System.Di	<pre>agnostics.Process::Start(string)</pre>
	call valuetype [System.Console]System.ConsoleKeyI	info [System.Console]Syste	<pre>m.Console::ReadKey()</pre>	
				×
Instructions L	ocals Exception Handlers	After		
Body Type IL		 Code Type 		
Keep Old M	AaxStack ✔ Init Locals Header RVA 0x18D4 Heade	r Offset 0xAD4	MaxStack 8	LocalVarSigTok 0
Index Offset	OpCode Operand			
	call void [System.Console]System.Console::WriteLi			
	call valuetype [System.Console]System.ConsoleKeyI	nfo [System.Console]Syste	<pre>m.Console::ReadKey()</pre>	

Figure 5: Editing IL instructions in dnSpyEx.

Save Module			×
Main MD Writer Options Metadata Cor20 PE		Main MD Writer Options Metadata Cor20 PE	
Save PDB File Shared Method Bodies Module Type DII	Mixed-Mode Module	✓ Preserve All MD Tokens ✓ TypeRef ✓ ✓ Field ✓ ✓ Param ✓	
Filename C:\CompileReal_ReplaceDecoy_IL\CompileReal_ReplaceDecoy_I	L\bin\Release\net6.0\win-x64\publish\CompileReal_ReplaceDecoy_IL.dll	StandAloneSig	
A Run NGEN.exe if this assembly is installed in the GAC		Property TypeSpec MethodSpec	
		✓ Preserve Heap Offsets ✓ #Strings ✓ #US ✓ #Blob	
		 ✓ Create Heap Even If Empty ✓ #Strings ✓ #US ✓ #Blob ✓ #Guid 	
		Misc Options Misc Options Preserve Extra Signature Data Keep Old MaxStack Value Copy Unknown Metadata Strea	
	OK Cancel Reset	OK Cancel Reset	

Figure 6: Saving the patched module in dnSpyEx.

R2R STOMPING - ARE YOU READY TO RUN? VINOPAL

Program	x	
1		using System;
2		
3		namespace CompileReal_ReplaceDecoy_IL
4		
5		internal class Program
7		private static void Main()
8		
9		Console.WriteLine("Welcome to VB 2023 !!!");
10		Console.ReadKey();
11		
12 13		
110 % -		
Main() : vo	oid	
1		// Token: 0x06000001 RID: 1 RVA: 0x000018D4 File Offset: 0x00000CD4
2		.method private hidebysig static
3		void Main () cil managed
4		
5		
6		// Code Size: 20 (0x14) bytes
7		.maxstack 8
8		.entrypoint
9 10		/* 0x00000CD5 7201000070 */ IL 0000: ldstr "Welcome to VB 2023 !!!"
10		/* 0x00000CDA 280B00000A */ IL 0005: call void [System.Console]System.Console::WriteLine(string)
11		/* axaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
12		/ 0X00000CF 00 // IL_0008: nop /* 0X00000CF 0 00 */ IL 0008: nop
13		/* 0x00000CE1 00 */ IL 000C: nop
14		/ il_corot. http://www.corot.c
16		/ oxobooociz zobooocoo / ri_oooc. cari varuetype [355tem.consore[355tem.consorekeyinto [355tem.consore]355tem.consore[355tem.consore]355tem.consore[355tem.consore]355tem.consore[355tem.consore]355tem.consore[355tem.consore]355tem.consore[355tem.consore]355tem.consore[355tem.consore[355tem.consore]355tem.consore[355tem.consore[355tem.consore]355tem.consore[355tem.c
17		/* 0x00000CF8 2A */IL0013: ret
18		// end of method Program::Main

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).

☑ PowerShell × + ×					- 🗆 X
PS C:\CompileReal_ReplaceDecoy_IL\CompileReal > dotnet CompileReal_ReplaceDecoy_IL.dll Welcome to VB 2023 !!!	🛐 Calcı	Decoy_IL\ ^{Jlator} Indard 쯌		ease\net6. □ × ©	.0\win-x64\publish
				0	
	MC I	MR M+	M- N	∕IS M∼	
	%	CE	с	Ø	
	1⁄x	x ²	∛x	÷	
	7	8	9	×	
	4	5	6		
	1	2	3	+	
	+/_	0	,	=	

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).

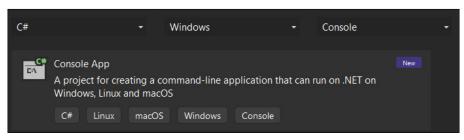


Figure 9: Visual Studio IDE – creation of new C#, Console App, .NET 6 project.

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.

13

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 !!!");
<pre>Console.WriteLine("Welcome to VB 2023 !!!");</pre>
Console.WriteLine("Welcome to VB 2023 !!!");
Console.ReadKey();

Figure 10: Decoy C# code.

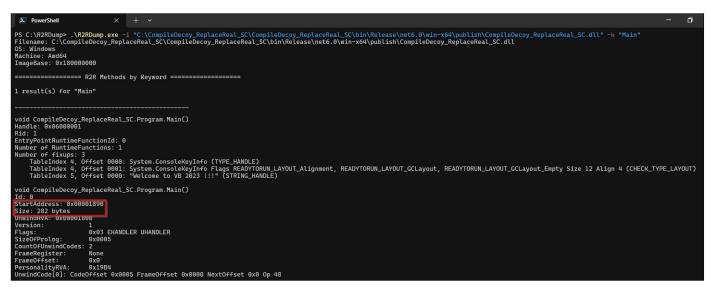


Figure 11: R2RDump tool parsing the structures of ReadyToRun assembly.

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.

📴 IDA View-A		
🛄 🛃 🖼		🔹 Export data 🛛 🕹
.text:000000180001890		
.text:000000180001890		
.text:000000180001890		Export as
.text:000000180001890	sub_180001890 proc near	hex string (unspaced)
.text:000000180001890		hex string (spaced)
.text:000000180001890	var_18= byte ptr -18h	string literal
.text:000000180001890		
.text:000000180001890	;unwind { // sub_1800019B4	C unsigned char array (hex)
.text:0000000180001890 56	push rsi	😑 C unsigned char array (decimal)
.text:000000180001891 48 83 EC 30	sub rsp, 30h	😑 initialized C variable
.text:000000180001895 48 8B 35 24 18 00 00	mov rsi, cs:qword_1800030C0	raw bytes
.text:00000018000189C 48 88 0E	mov rcx, [rsi]	Taw Dytes
.text:00000018000189F FF 15 03 18 00 00	call cs:off_1800030A8	
.text:00000001800018A5 48 88 0E .text:00000001800018A8 FF 15 FA 17 00 00	mov rcx, [rsi] call cs:off 1800030A8	
.text:00000001800018A8 FF 15 FA 17 00 00 .text:00000001800018AE 48 8B 0E		Save data to clipboard
.text:00000001800018B1 FF 15 F1 17 00 00	mov rcx, [rsi] call cs:off 1800030A8	
.text:0000001800018B7 48 8B 0E		Preview
.text:00000001800018BA FF 15 E8 17 00 00	mov rcx, [rsi] call cs:off 1800030A8	
.text:0000001800018C0 48 88 0E	call cs:ott_1000050A0 mov rcx, [rsi]	564883EC30488B3524180000488B0EFF1503180000488B0EFF15FA170000488B0EFF15F1170000488B0EFF
.text:0000001800018C3 FF 15 DF 17 00 00	call cs:off 1800030A8	15E8170000488B0EFF15DF170000488B0EFF15D6170000488B0EFF15CD170000488B0EFF15C4170000488
.text:0000001800018C9 48 88 0E	call cs:off_1000050A6 mov rcx, [rsi]	B0EFF15BB170000488B0EFF15B2170000488B0EFF15A9170000488B0EFF15A0170000488B0EFF159717000
.text:0000001800018CC FF 15 D6 17 00 00	call cs:off 1800030A8	048880EFF158E17000048880EFF158517000048880EFF157C17000048880EFF157317000048880EFF156A1 7000048880EFF156117000048880EFF155817000048880EFF154F17000048880EFF154617000048880EFF1
.text:00000001800018D2 48 8B 0E	mov rcx, [rsi]	53D170000488B0EFF1531170000488B0EFF152B170000488B0EFF1540170000488B0EFF1519170000488B0
.text:0000001800018D5 FF 15 CD 17 00 00	call cs:off 1800030A8	EFF1510170000488D4C2420FF15FD16000904883C4305EC3
.text:00000001800018DB 48 8B 0E	mov rcx, [rsi]	
.text:0000001800018DE FF 15 C4 17 00 00	call cs:off 1800030A8	
.text:0000001800018E4 48 8B 0E	mov rcx, [rsi]	
.text:0000001800018E7 FF 15 BB 17 00 00	call cs:off 1800030A8	
.text:0000001800018ED 48 8B 0E	mov rcx, [rsi]	Line:1 Column:1
.text:00000001800018F0 FF 15 B2 17 00 00	call cs:off 1800030A8	
.text:00000001800018F6 48 8B 0E	mov rcx, [rsi]	Output file export_results.txt ~
.text:00000001800018F9 FF 15 A9 17 00 00	call cs:off 1800030A8	Export Cancel
.text:00000001800018FF 48 8B 0E	mov rcx, [rsi]	Cancel

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*.

ppileDecoy_ReplaceReal_SC.dll ×		leDecoy		ceRea	I_SC.dl	l_patch	ied* ×							
Ŏ 1 2 3 4 5 6 7 8 9 A B C D E F Ŏ123456789ABCDEF														
0h 00 02 02 CC CC CC CC CC 43 72 6F 73 73 67 65 6EÌÌÌÌÌCrossgen	0A70h												EÌÌÌÌÌCrossgen	
0h 32 20 36 2E 30 2E 31 36 32 33 2E 31 37 33 31 31 2 6.0.1623.17311	0A80h												1 2 6.0.1623.17311	
Oh 56 48 83 EC 30 48 8B 35 24 18 00 00 48 8B 0E FF VHfìOH‹5\$H‹.ÿ	0A90h												0 ™′ýùŸ.üHfäðèÀ	
0h 15 03 18 00 00 48 8B 0E FF 15 FA 17 00 00 48 8BH‹.ÿ.úH‹	0AA0h													
0h 0E FF 15 F1 17 00 00 48 8B 0E FF 15 E8 17 00 00 .ÿ.ñH<.ÿ.è	0AB0h												A <r.h<r h<rph.="" jj<="" th=""><th></th></r.h<r>	
0h 48 8B 0E FF 15 DF 17 00 00 48 8B 0E FF 15 D6 17 H<.ÿ.ßH<.ÿ.Ö.	0AC0h													
0h 00 00 48 8B 0E FF 15 CD 17 00 00 48 8B 0E FF 15H‹.ÿ.İH‹.ÿ.	0AD0h												C .A.AâíRAQH‹R ‹B<	
0h C4 17 00 00 48 8B 0E FF 15 BB 17 00 00 48 8B 0E ÄH‹.ÿ.»H‹.	0AE0h													
0h FF 15 B2 17 00 00 48 8B 0E FF 15 A9 17 00 00 48 ÿ.²H∢.ÿ.©H	0AF0h		50 8B											
0h 8B 0E FF 15 A0 17 00 00 48 8B 0E FF 15 97 17 00 <.ÿH<.ÿ	0B00h													
0h 00 48 8B 0E FF 15 8E 17 00 00 48 8B 0E FF 15 85 .H∢.ÿ.ŽH∢.ÿ			C9 0D											
0h 17 00 00 48 8B 0E FF 15 7C 17 00 00 48 8B 0E FFH‹.ÿ. H‹.ÿ	0B20h													
0h 15 73 17 00 00 48 8B 0E FF 15 6A 17 00 00 48 8B .sH‹.ÿ.jH‹	0B30h												1 HD<@.I.ĐA<.^H.ĐA	
Oh 0E FF 15 61 17 00 00 48 8B 0E FF 15 58 17 00 00 .ÿ.aH‹.ÿ.X	0B40h													
0h 48 8B 0E FF 15 4F 17 00 00 48 8B 0E FF 15 46 17 H<.ÿ.OH<.ÿ.F.	0B50h												F ARÿàXAYZH∢.éWÿÿÿ	
Oh 00 00 48 8B 0E FF 15 3D 17 00 00 48 8B 0E FF 15H<ÿ.=H<ÿ.	0B60h													
Oh 34 17 00 00 48 8B 0E FF 15 2B 17 00 00 48 8B 0E 4H<.ÿ.+H<.	0B70h		00 41											
0h FF 15 22 17 00 00 48 8B 0E FF 15 19 17 00 00 48 ÿ."H‹.ÿH	0B80h													
0h 8B 0E FF 15 10 17 00 00 48 8D 4C 24 20 FF 15 FD ‹.ÿH.L\$ ÿ.ý	0B90h													
0h 16 00 00 90 48 83 C4 30 5E C3 CC CC CC CC CC CCHfÄ0^ÄÌÌÌÌÌÌ														
0h C3 CC CC CC FF 25 D6 16 00 00 CC CC FF 25 D6 16 ÅÌÌÌÿ%ÖÌÌÿ%Ö.													6 ÂÌÌÌÿ%ÖÌÌÿ%Ö.	
0h 00 00 CC CC 33 C0 6A 03 FF 35 B2 16 00 00 FF 25ÌÌ́́3Àj.ÿ5²ÿ%	0BC0h	00 0	00 CC	CC	33 CC	6A	03 FF	35	B2 1	6 00	00 (FF 25	5ÌÌÌ3Àj.ÿ5²ÿ%	
nd Hex Bytes: ^ 56 48 83 EC 30 48 ^ 🗸 🔺 All 🛛 Options ^ 56 48 83 EC 30 48 8B 35 24 18 00 00 48 8	8B 0E FF 15 03 1	8 00 00	48 8B (DE FF 1		7 00 00) 48 8B	OE FF			00 48 8	B OE F	F 15 E8 17 00 00 48 8B 0E FF	15 DF
Results														
Address Value														
Found 1 occurrences of '56 48 83 EC 30 48 8B 35 24 18 00 00 48 8B 0E FF 15 03 18 00 00 48 8B 0E FF 15 FA 17 (00 00 48 88 0F F	E 15 E1	17 00 0	0 48 8	B OF F	E 15 E8	17 00	00 48	88 0F	FE 15	DF 17 (0 00 4	18 8B 0F FF 15 D6 17 00 00 48	8B 0F

Figure 13: Binary patching using 010 Editor.

If we try to run our ReadyToRun stomped assembly, either via its associated executable <code>CompileDecoy_ReplaceReal_SC.exe</code> located in the same folder or via issuing the <code>dotnet_CompileDecoy_ReplaceReal_SC.dll</code> 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.

➢ PowerShell × + ✓						-		×
<pre>PS C:\CompileDecoy_ReplaceReal_SC\CompileDe > dotnet CompileDecoy_ReplaceReal_SC.dll</pre>	coy_Repl		SC\bin\f	Release\ne	t6.0\win-	х64\рı	ubli	sh
		ndard 🕫		_ଅ ି ଚ				
				0				
	мс м	IR M+	M-	MS M~				
		CE						
		8						
	4							
				=				

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.

using System;							
3 namespace CompileReal_ReplaceDecoy_IL							
€ internal class Program							
private static void Main()							
Console.WriteLine("Welcome to VB 2023 !!!");							
Console.ReadKey();							
// Token: 0x06000001 RID: 1 RVA: 0x000018D4 File Offset: 0x00000CD4							
.method private hidebysig static							
void Main () cil managed							
.maxstack 8							
.entrypoint							
/# 0x00000CD5 7201000070 #/ IL 0000: ldstr "Welcome to VB 2023 !!!"							
/ 0x00000ctu3 /201000070 // IL_0000: Idlt welcome to vo 2023 ::: /* 0x00000ctu3 /201000070 // IL_0000: call void [\$ystem.console[\$ystem.console::WriteLine(string)							
/ 0x00000CDA 20000000A / it_0003; call vul [system.consule.iwiteLine(sting)							
/ 0x00000Cb 00 / it_000k nop /* 0x00000Cb 00 **/It_000k:nop							
/ 0x000000E1 00 / /I_000C:nop							
/ 0X000000L1 00 // IL_2000. nop /* 0X000000L2 2800000004 */ IL 0000: call valuetype [System.Console]System.ConsoleKeyInfo [System.Console]System.Console::ReadKey()							
/ 0x000000C2 260000000 / /1_0000. Call valuetype [3ystem.console3ystem.consolexystem.console3ystem.c							
/ 0x000000CF 20 / it_0013: ret							
/ ondoworks 2A m							

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.

Assembly Explorer 👻 🗙	Main():void ×
✓	1 // Token: 0x06000001 RID: 1 RVA: 0x000018D4 File Offset: 0x00000CD4
CompileReal_ReplaceDecoy_IL.dll	
▶ 😬 PE	3 void Main () cil managed
✓ P□ Type References	
AssemblyCompanyAttribute @01000006	
AssemblyConfigurationAttribute @01000007	
AssemblyFileVersionAttribute @01000008	
AssemblyInformationalVersionAttribute @01000009	
AssemblyProductAttribute @0100000A	9 /* 0x00000CD5 7201000070 */ IL 0000: ldstr "Welcome to VB 2023 !!!"
AssemblyTitleAttribute @0100000B	11 /* 0x00000CD 20000000 //110000 instr /* 0x00000CD 20000000 / 11 0005 call void [System.Console]System.Console::WriteLine(string)
CompilationRelaxationsAttribute @01000001	12 / # 0x00000CGA Z000000CA / it_0003. Cdit VOID [3ystem.console_system.consolemittetime(string)
Console @0100000D	13 /* 0x00000CE 00 */ 1_000E nop
ConsoleKeyInfo @0100000F	13 / 0.000000CE100 / 1_000C. nop
DebuggableAttribute @01000003	15 /* 0x000000CE2 280D00000A / /L_000D: call valuetype [System.Console]System.ConsoleKeyInfo [System.Console]System.Console::ReadKey()
DebuggingModes @01000004	16 /* 0x00000CF 26 */ IL 0012: pop
♦ ² \$ object @0100000C	17 /* 0x00000CE 2A */IL 0013: ret
Y Process @0100000E	18 } // end of method Program::Main
A C Methods	
Start(string) : Process @0A00000C	
RuntimeCompatibilityAttribute @01000002	
TargetFrameworkAttribute @01000005	
▷ □□ References ▷ { } -	
 { } CompileReal_ReplaceDecoy_IL % Program @02000002 	
 Base Type and Interfaces Derived Types 	
Program() : void @06000002	
A Main() : void @06000002	

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.

Assembly Explorer 👻 🗙	Main() : void 🗙
Assembly Explorer	<pre>Main():void X 1 // CompileDecoy_ReplaceReal_SC.Program 2 private static void Main() 3 { 4 Console.WriteLine("Welcome to VB 2023 !!!"); 5 Console.WriteLine("Welcome to VB 2023 !!!"); 6 Console.WriteLine("Welcome to VB 2023 !!!"); 7 Console.WriteLine("Welcome to VB 2023 !!!"); 8 Console.WriteLine("Welcome to VB 2023 !!!"); 9 Console.WriteLine("Welcome to VB 2023 !!!"); 10 Console.WriteLine("Welcome to VB 2023 !!!"); 10 Console.WriteLine("Welcome to VB 2023 !!!"); 10 Console.WriteLine("Welcome to VB 2023 !!!"); 10</pre>
 AssemblyProductAttribute @0100000A AssemblyProductAttribute @0100000B AssemblyTitleAttribute @0100000B CompilationRelaxationsAttribute @01000001 Console @0100000D Console &0100000D Console &0100000E DebuggableAttribute @01000003 DebuggableAttribute.DebuggingModes @01000004 Sobject @0100000C RuntimeCompatibilityAttribute @01000002 TargetFrameworkAttribute @01000005 	<pre>10 Console.WriteLine("Welcome to VB 2023 !!!"); 11 Console.WriteLine("Welcome to VB 2023 !!!"); 12 Console.WriteLine("Welcome to VB 2023 !!!"); 13 Console.WriteLine("Welcome to VB 2023 !!!"); 14 Console.WriteLine("Welcome to VB 2023 !!!"); 15 Console.WriteLine("Welcome to VB 2023 !!!"); 16 Console.WriteLine("Welcome to VB 2023 !!!"); 17 Console.WriteLine("Welcome to VB 2023 !!!"); 18 Console.WriteLine("Welcome to VB 2023 !!!"); 19 Console.WriteLine("Welcome to VB 2023 !!!"); 20 Console.WriteLine("Welcome to VB 2023 !!!"); 21 Console.WriteLine("Welcome to VB 2023 !!!"); 22 Console.WriteLine("Welcome to VB 2023 !!!"); 23 Console.WriteLine("Welcome to VB 2023 !!!");</pre>
 ▷ □ References ▷ { } - ▲ { } CompileDecoy_ReplaceReal_SC ▲ ♡ Program @02000002 ▷ □ Base Type and Interfaces ▷ □ Derived Types ♡ Program () : void @06000002 ♡ Main() : void @0600001 	<pre>23 Console.WriteLine("Welcome to VB 2023 !!!"); 24 Console.WriteLine("Welcome to VB 2023 !!!"); 25 Console.WriteLine("Welcome to VB 2023 !!!"); 26 Console.WriteLine("Welcome to VB 2023 !!!"); 27 Console.WriteLine("Welcome to VB 2023 !!!"); 28 Console.WriteLine("Welcome to VB 2023 !!!"); 29 Console.WriteLine("Welcome to VB 2023 !!!"); 30 Console.WriteLine("Welcome to VB 2023 !!!"); 31 Console.WriteLine("Welcome to VB 2023 !!!"); 32 Console.WriteLine("Welcome to VB 2023 !!!"); 33 }</pre>

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.

Options		×
Options Environment Decompiler Compiler Debugger Disassembler Text Viewer Code Editor REPL Output Window Assembly Explorer	 Enable Managed Debugging Assistants (MDA) Enable Just My Code debugging support Step over code in system modules Only step into code located in primary module Show raw structure of objects in variables windows Ignore unhandled exceptions Suppress JIT optimization on module load System modules Program modules 	×
BAML Decompiler ▶ Hex Editor Background Image Bookmarks	Hide compiler generated members Respect attributes that hide members Hide deprecated members in variables windows	•
	OK Canc	el "

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.

using System;	C:\Program Files\dotnet\dotnet.exe	П	
namespace CompileDecoy ReplaceReal SC			
	Welcome to VB 2023 !!!		
internal class Program	Welcome to VB 2023 !!!		
{ private static void Main()	Welcome to VB 2023 !!!		
privace scalic void main()	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
<pre>Console.WriteLine("Welcome to VB 2023 !!!");</pre>	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!");			
<pre>Console.WriteLine("Welcome to VB 2023 !!!");</pre>	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!"); Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!); Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!"); Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
Console.WriteLine("Welcome to VB 2023 !!!");	Welcome to VB 2023 !!!		
<pre>Console.WriteLine("Welcome to VB 2023 !!!");</pre>			
<pre>Console.WriteLine("Welcome to VB 2023 !!!");</pre>	Welcome to VB 2023 !!!		
<pre>Console.WriteLine("Welcome to VB 2023 !!!"); Console.ReadKey();</pre>			

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.

🔊 PowerShell × + ~						-		х
PS C:\CompileDecoy_ReplaceReal_SC\G > dotnet CompileDecoy_ReplaceReal_S	SC.dll 📱 Ca	lculator	L_SC\bin\ _		t6.0\win	-x64\ŗ	oubli	sh
	≡ Si	tandard	9 	ত 				
				0				
		MR M+	· М-	MS M~				
		CE						
		8	9					
	4							
				=				

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.

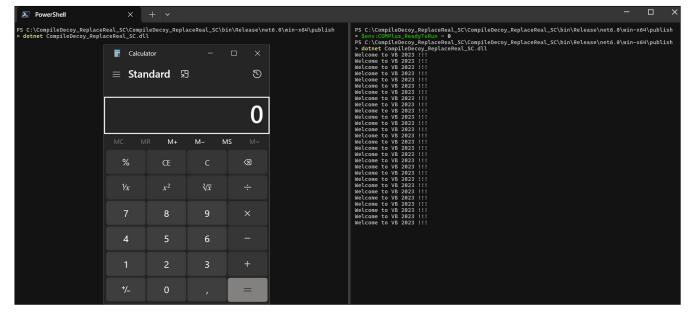


Figure 21: Normal execution of R2R stomped assembly without and with the setting of 'COMPlus ReadyToRun=0'.

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*.

← → • ↑ 📮 > This PC > Local Disk (C:) > CompileDecoy_Repla	ceReal_SC > CompileD	ecoy_ReplaceF	eal_SC → bin → Release → net€	5.0 > v	vin-x64 → publish		~ Ŭ	O Search publis
Name	Date modified	Туре	Size						
Extracted_bundle	12.06.2023 3:44	File folder							
CompileDecoy_ReplaceReal_SC.exe	12.06.2023 3:40	Application		2 430 KB					
📙 🛃 📒 = C:\CompileDecoy_ReplaceReal					A	ssembly Explorer			
File Home Share View				~	? *	P 🖬 福 🎾 🍬 🖽 🔡 🐐 D D			
← → マ ↑ 📙 « Release > net6.0 > w	vin-x64 > publish > Extra	cted_bundle							
Name	Date modified	Туре				P Q metadata	🕲 Navig		
CompileDecoy_ReplaceReal_SC.deps.json	12.06.2023 3:44	JSON Source File	27 KB			Win32 resources CompileDecoy_ReplaceReal_SC.deps.json		n Containing Folder	
CompileDecoy_ReplaceReal_SC.dll	12.06.2023 3:44	Application extens				 CompleDecoy_ReplaceReal_SC.deps.json * CompileDecoy_ReplaceReal_SC (1.0.0.0, x64, .f 	Сору	Full Path	
CompileDecoy_ReplaceReal_SC.runtimec	12.06.2023 3:44	JSON Source File				CompileDecoy ReplaceReal SC.runtimeconfig	Extra	ct Bundle Contents to Folder	··· 📐
Microsoft.CSharp.dll	12.06.2023 3:44	Application extens	1 019 KB			Microsoft.CSharp (6.0.0.0, x64, .NETCoreApp ve	🗙 Remo	ove Item from List	
Microsoft.VisualBasic.Core.dll	12.06.2023 3:44	Application extens	1 217 KB			Microsoft.VisualBasic.Core (11.0.0.0, x64, .NETC)	Show	Assembly Dependency Diac	Iram
Microsoft.VisualBasic.dll	12.06.2023 3:44	Application extens	18 KB			• O Microsoft.VisualBasic (10.0.0.0, msil, .NETCore/			
Microsoft.Win32.Primitives.dll	12.06.2023 3:44	Application extens	26 KB			P BINICIOSOT. WITTELETITITITIVES (0.0.0.0, X04, 14ETC)	A Prop		
Microsoft.Win32.Registry.dll	12.06.2023 3:44	Application extens	86 KB			Microsoft.Win32.Registry (6.0.0.0, x64, .NETCore.	App v6.0,		
🗟 mscorlib.dll	12.06.2023 3:44	Application extens	58 KB						
📓 netstandard.dll	12.06.2023 3:44	Application extens	100 KB			 Interstandard (2.1.0.0, msil, .NETCoreApp v6.0) System.AppContext (6.0.0.0, msil, .NETCoreApp 			
168 items 1 item selected 6,00 KB				B		System.AppContext (6.0.0.0, msil, INETCORAPP			

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 im	hage
Usage:	
R2RDump [options]	
Options:	
-i,in <in></in>	Input file(s) to dump. Expects them to by ReadyToRun images
-o,out <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>	Query method by exact name, signature, row ID or token
-k,keyword <keyword></keyword>	Search method by keyword
-f,runtimefunction <runtimefunction></runtimefunction>	Get one runtime function by id or relative virtual address
-s,section <section></section>	Get section by keyword
unwind	Dump unwindInfo
gc	Dump gcInfo and slot table
––pgo	Dump embedded pgo instrumentation data
<pre>sc,sectionContents</pre>	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-path>	PDB output path forcreate-pdb
create-perfmap	Create PerfMap
perfmap-path <perfmap-path></perfmap-path>	PerfMap output path forcreate-perfmap
perfmap-format-version <perfmap-format-version></perfmap-format-version>	PerfMap format version forcreate-perfmap [default: 1]
-r,reference <reference></reference>	Explicit reference assembly files
referencePath,rp <referencepath></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.

PS C:\222Dump>. \R22Dump.exe -i.\CompileDecoy_DeplaceReal_SC.dllheadersc Filename: C: Z22Dump\CompileDecoy_ReplaceReal_SC.dll DS: Windows Machine: Anddu ImageBase: 0:10000000 =============================	SectionRVA: 0x000030A0 (12448) SectionSize: 0 bytes Flags: DCade Type: Unknown EntrySize: 8 SignatureRVA: 0x00000000 (0) AuxillaryUnatRVA: 0x0000000 (0)
Signature: 0x00525452 (RTR) Relative/IrtualAddress: 0x00001648 Sign: 140 bytesion: 0x0005 Najo: 0x00000 Plag: 0x0000000 - READYTORUM_FLAG_MONSharedPInvokeStubs - READYTORUM_FLAG_MONSharedPInvokeStubs	SectionRVA: 0x000030A0 (12448) SectionSize: 16 bytes Flags: Rcdo Type: Statu Bpatch SignatureNvk: 0x000039E0 (12512) AuxiliaryDataRVA: 0x000039E0 (12512) AuxiliaryDataRVA: 0x0000130E0 (12512) AuxiliaryDataRVA: 0x0000130E0 (12512) AuxiliaryDataRVA: 0x000030E0 (12512) AuxiliaryDataRVA: 0x0000000 AuxiliaryDataRVA: 0x0000000 AuxiliaryDataRVA: 0x00000000 AuxiliaryDataRVA: 0x000000000
11 sections	_REF_TOKEN) R(48) SectionRVA: 0x00003080 (12464)
Type: CompilerIdentifier (100) RelativeVirtualAddress: 0x00001870 Size: 24 bytes Crossgen2 6.0.1623.1731	SectionSize: 16 bytes Flags: PCode Type: Unknown EntrySize: 0 8080830E5 (13520) Auddel SyntherRMA. BeroBooles (0) Auddel SyntherRMA. BeroBooles (0) Auddel SyntherRMA. BeroBooles SignatureRVA: 8x00003107 System ConsoleKeyInfo (TYPE_HANDLE) +0008 (SBBS) Section: 8x000008000 SignatureRVA: 8x00003107 System ConsoleKeyInfo Flags REDYTORUL, LAYOUT_Alignm +0008 (SBBS) Section: 8x000008000 SignatureRVA: 8x00003107 System ConsoleKeyInfo Flags REDYTORUL, LAYOUT_Alignm +0008 (SBBS) Section: 8x000008000 SignatureRVA: 8x00003107 System ConsoleKeyInfo Flags REDYTORUL, LAYOUT_Alignm
Type: ImportSections (101) Rolaivu/itualdaress: 0x00003000 Size: 120 bytes SectionRV: 0x00003070 (12400) SectionSize: 0 bytes Fugs: Focdo Type: StubDispatch EntrySize: 8 SignatureRVA: 0x00001808 (6) AuxiliaryDatANVA: 0x00001808 (6920)	SectionSize RooBeastoc (12480) SectionSize: 8 bytes Flags: None Type: StringHandle EntrySize: 8 3098096062520 SignatureRVA: 0x0003060602520 AuxiliaryDataRVA: 0x000306060080 (0) +0000 (30C0) Section: 0x0000000 (0)
SectionRVA: 6x00003078 (12408) SectionSize: 40 bytes Thays: Lage: EntrySize: 8 SignatureRVA: 6x00003080 (8) AuxiliaryDataNVA: 6x00003080 (8) Section: 6x00008000 (8) Section: 6x00008000 SignatureRVA: 6x000830FM ODDLE (HELPER) H030 (3080) Section: 6x00000000 SignatureRVA: 6x000830FM H030 (3080) Section: 6x00000000 SignatureRVA: 6x0000310F H030 (3080) Section: 6x00000000 SignatureRVA: 6x0000310F H050MLTY.ROUTINE (ELPER)	Type: RuntimeFunctions (102) PelatiuwVirtunlAddress: 0x000018uC Size: 20 bytes Index StartRVA EndRVA UnwindRVA e 0e001900 000015ha 000015ha 1 000015b0 000015ha 000015h4
SectionRVA: 0x000030A0 (12448) SectionSize: 0 bytes	Fype: metmoderinityFounds (1857) RelativeVirualAddress: 0x000016E0 Size: 12 bytes

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.

Main() : void $ imes$	Main() : void 🗙
1 // CompileDecoy_ReplaceReal_SC.Program	1 // Token: 0x06000001 RID: 1 RVA: 0x000019D4 File Offset: 0x00000BD4
2 private static void Main()	 method private hidebysig static
3 (3 void Main () cil managed
<pre>4 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	
<pre>5 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	
<pre>6 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	
<pre>7 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	
8 Console.WriteLine("Welcome to VB 2023 !!!");	8 .entrypoint
<pre>9 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	
<pre>10 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	10 /* 0x000006E0 7201000070 */ IL_0000: ldstr "Welcome to VB 2023 !!!"
<pre>11 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	11 /* 0x000008E5 280600000A */ IL_0005: call void [System.Console]System.Console::WriteLine(string)
<pre>12 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	12 /* 0x00000BEA 7201000070 */ IL_000A: ldstr "Welcome to VB 2023 !!!"
13 Console.WriteLine("Welcome to VB 2023 !!!");	13 /* 0x00000BEF 280800000A */ IL_000F: call void [System.Console]System.Console::WriteLine(string)
<pre>14 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	14 /* 0x00000BF4 7201000070 */ IL_0014: ldstr "Welcome to VB 2023 !!!"
<pre>15 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	15 /* 0x00000BF9 280B00000A */ IL_0019: call void [System.Console]System.Console::WriteLine(string)
<pre>16 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	16 /* 0x00000BFE 7201000070 */ IL_001E: ldstr "Welcome to VB 2023 !!!"
<pre>17 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	<pre>17 /* 0x00000C03 280B00000A */ IL_0023: call void [System.Console]System.Console::WriteLine(string)</pre>
18 Console.WriteLine("Welcome to VB 2023 !!!");	
<pre>19 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	19 /* 0x00000C0D 280B00000A */ IL_002D: call void [System.Console]System.Console::WriteLine(string)
20 Console.WriteLine("Welcome to VB 2023 !!!");	20 /* 0x00000C12 7201000070 */ IL_0032: ldstr "Welcome to VB 2023 !!!"
<pre>21 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	21 /* 0x00000C17 280B00000A */ IL_0037: call void [System.Console]System.Console::WriteLine(string)
<pre>22 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	22 /* 0x00000C1C 7201000070 */ IL_003C: ldstr "Welcome to VB 2023 !!!"
23 Console.WriteLine("Welcome to VB 2023 !!!");	23 /* 0x00000C21 280B00000A */ IL_0041: call void [System.Console]System.Console::WriteLine(string)
24 Console.WriteLine("Welcome to VB 2023 !!!");	24 /* 0x00000C26 7201000070 */ IL_0046: ldstr "Welcome to VB 2023 !!!"
<pre>25 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	25 /* 0x00000C2B 280B00000A */ IL_004B: call void [System.Console]System.Console::WriteLine(string)
<pre>26 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	26 /* 0x00000C30 7201000070 */ IL_0050: ldstr "Welcome to VB 2023 !!!"
<pre>27 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	27 /* 0x00000C35 280B00000A */ IL_0055: call void [System.Console]System.Console::WriteLine(string)
28 Console.WriteLine("Welcome to VB 2023 !!!");	
<pre>29 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	29 /* 0x00000C3F 280B00000A */ IL_005F: call void [System.Console]System.Console::WriteLine(string)
<pre>30 Console.WriteLine("Welcome to VB 2023 !!!");</pre>	30 /* 0x00000C44 7201000070 */ IL_0064: ldstr "Welcome to VB 2023 !!!"
31 Console.WriteLine("Welcome to VB 2023 !!!");	31 /* 0x00000C49 280B00000A */ IL_0069: call void [System.Console]System.Console::WriteLine(string)
32 Console.ReadKey();	32 /* 0x00000C4E 7201000070 */ IL_006E: ldstr "Welcome to VB 2023 !!!"
33 }	33 /* 0x00000C53 280800000A */ IL 0073: call void [System.Console]System.Console::WriteLine(string)

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).

PS C:\R2RDump> . <mark>\R2RDump.exe -i .\CompileDecoy_ReplaceReal_SC.dll -q "Main" -</mark> d Filename: C:\R2RDump\CompileDecoy_ReplaceReal_SC.dll	1890: 99	cdq UWOP_PUSH_NONVOL RSI(6)	Θ
OS: Windows Machine: Amd64	1891: 92	xchg eax, edx	
ImageBase: 0x180000000	1892: fd 1893: f9	std stc lahf	
======================================	1894: 9f		Э
1 result(s) for "Main"	1895: 90 1896: fc	nop cld	
	1895. FC 1897: 48 83 e4 f0 1895: e8 c0 00 00 00	and rsp, -16 call 0x1960	
void CompileDecoy_ReplaceReal_SC.Program.Main()	1895. 28 C6 86 86 88 18a0: 41 51 18a2: 41 50	push r9 push r8	
Handle: 0x06000001 Rid: 1	18a4: 52 18a5: 51	push rox push rcx	
EntryPointRuntimeFunctionId: 0 Number of RuntimeFunctions: 1	1845: 51 18a6: 56 18a7: 48 31 d2	push rcx push rsi xor rdx, rdx	
Number of fixups: 3 TableIndex 4, Offset 0000: System.ConsoleKevInfo (TYPE_HANDLE)	18a7: 48 31 02 18aa: 65 48 8b 52 60 18af: 48 8b 52 18	mov rdx, rdx mov rdx, qword ptr gs:[rdx + 96] mov rdx, qword ptr [rdx + 24]	
TableIndex 4, Offset 0001: System.ConsoleKeyInfo Flags READYTORUN_LAYOUT_Alignm	18b3: 48 8b 52 20 18b7: 48 8b 72 50	mov rdx, qword ptr [rdx + 32] mov rsi, qword ptr [rdx + 80]	
ent, READYTORUN_LAYOUT_GCLayout, READYTORUN_LAYOUT_GCLayout_Empty Size 12 Align 4 (CHECK_TYPE_LAYOUT)	18bb: 48 0f b7 4a 4a 18c0: 4d 31 c9	movzx rcx, word ptr [rdx + 74] xor r9, r9	
TableIndex 5, Offset 0000: "Welcome to VB 2023 !!!" (STRING_HANDLE)	1860: 40 51 C9 18c3: 48 31 c0 18c6: ac	xor rax, rax lodsb al, byte ptr [rsi]	
void CompileDecoy_ReplaceReal_SC.Program.Main() Id: 0	1867: 3c 61 18c9: 7c 02	cmp al, 97 jl 0x18cd	
StartAddress: 0x00001890 Size: 282 bytes	18cb: 2c 20 18cd: 41 c1 c9 0d	sub al, 32 ror r9d, 13	
UnwindRVA: 0x00001800 Version: 1	18d1: 41 01 c1 18d4: e2 ed	add r9d, eax loop -19	
Flags: 0x03 EHANDLER UHANDLER	18d6: 52 18d7: 41 51	push rdx push r9	
SizeOfProlog: 0x0005 CountOfUnwindCodes: 2	18d9: 48 8b 52 20 18dd: 8b 42 3c	mov rdx, qword ptr [rdx + 32] mov eax, dword ptr [rdx + 60]	
FrameRegister: None FrameOffset: 0x0	18e0: 48 01 d0 18e3: 8b 80 88 00 00 00	add rax, rdx mov eax, dword ptr [rax + 136]	
PersonalityRVA: 0x19B4 UnwindCode[0]: CodeOffset 0x0005 FrameOffset 0x0000 NextOffset 0x0 Op 48	18e9: 48 85 c0 18ec: 74 67	test rax, rax	

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.

💿 💿 🍅 🕖 (Default) 🔹 🕯 📲 🚅 👯 ReadyT	ōRun 🔻 2 🗃 🔎			
Assemblies 👻 👎	Main() : void			₹
Assemblies Assemblies Assemblies Assemblies CompileDecoy_ReplaceReal_SC (1.0.0.0, .NETCoreApp, v6.0)	Main():void ; void CompileDecoy_ReplaceReal_SC.Pr ; Prolog 000000000001890 99 000000000001891 92 0000000000001891 92 0000000000001893 F9 0000000000001895 90 0000000000001895 90 0000000000001895 FC 0000000000001895 E3C000000 0000000000001895 E3C000000 0000000000001895 4151 0000000000018A4 4151 0000000000018A4 52 ; IL_000a 0000000000018A5 51 0000000000018A7 4831D2 0000000000018A7 6548852560	ogram.Main cdq xchg std stc lahf cld and call push push push push push push push	<pre>() edx,eax rsp,0`FFFF`FFFF`FFFF`FFF0h 0000`0000`0000`1960h r9 r8 rdx rcx rcx rsi rdx,fdx rdx,[gs:rdx+60h]</pre>	▼ <
	00000000000018AF 488B5218 00000000000018B3 488B5220	mov mov	rdx,[rdx+18h] rdx,[rdx+20h]	

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).

As	sembly Explorer			×		
*C) ᅺ 🍿 🔯 🎾 🐐 🗮 🔛 👫 🖄 🖑					
Ту	pe to search		ç٥	Щ.		
	🐻 CompileDecoy_ReplaceReal_SC (1.0.0.0, x64, .NETC	CoreApp v6.0 <mark>, R2R)</mark>				
	Metadata					
	References					
	Win32 resources					
	CompileDecoy_ReplaceReal_SC					
Pr	operties		• 7	×		
⊳	Assembly					
⊳	Module					
⊳	Pdb					
4	ReadyToRun					
	Compiler	Crossgen2 6.0.1623.17311				
	Instruction Sets X86Base+Sse+Sse2+					
	Architecture	x64				
	Operating System	Windows				

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) + 0 \times 40 != 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).

Main(): void ×	▼ Main():void ×	Ŧ	Main() : void 🗙		
<pre>// CompileDecoy_ReplaceReal_SC, Version=1.0.0.0,</pre>					n ()
<pre>// CompileDecoy_ReplaceReal_SC.Program</pre>	🖻 void Main () cil managed				
using System;			000000000001890 99		w.
			000000000001891 92	xchg	edx,eax
private static void Main()	// Header size: 12 // Code size: 287 (0x11f)		000000000001892 FD		
□ {	.maxstack 1		000000000001893 F9		
Console.WriteLine("Welcome to VB 2023 !!!");	entrypoint		0000000000001894 9F		
Console.WriteLine("Welcome to VB 2023 !!!");	rener jpozne				
Console.WriteLine("Welcome to VB 2023 !!!");	IL_0000: ldstr "Welcome to VB 2023 !!!"		000000000001895 90		
Console.WriteLine("Welcome to VB 2023 !!!");	IL_0005: call void [System.Console]System.Console::WriteLine(string)	000000000001896 FC		
Console.WriteLine("Welcome to VB 2023 !!!");	IL_000a: ldstr "Welcome to VB 2023 !!!"		0000000000001897 4883E4F0		rsp,0`FFFF`FFFF`FFFF`FFF0h
Console.WriteLine("Welcome to VB 2023 !!!");	<pre>IL_000f: call void [System.Console]System.Console::WriteLine(string</pre>)	00000000000189B E8C0000000		0000`0000`0000`1960h
Console.WriteLine("Welcome to VB 2023 !!!");	IL_0014: ldstr "Welcome to VB 2023 !!!"		0000000000018A0 4151		r9
Console.WriteLine("Welcome to VB 2023 !!!");	IL_0019: call void [System.Console]System.Console::WriteLine(string IL_001e: ldstr "Welcome to VB 2023 !!!")	00000000000018A2 4150		r8
Console.WriteLine("Welcome to VB 2023 !!!");	IL_001e: Idstr Welcome to VB 2023 !!! IL 0023: call void [System.Console]System.Console::WriteLine(string	、 III	0000000000018A4 52		rdx
Console.WriteLine("Welcome to VB 2023 !!!");	IL 0028: ldstr "Welcome to VB 2023 !!!"	'			
Console.WriteLine("Welcome to VB 2023 !!!");	IL_002d: call void [System.Console]System.Console::WriteLine(string	Ϋ́	0000000000018A5 51		rcx
Console.WriteLine("Welcome to VB 2023 !!!");	IL 0032: ldstr "Welcome to VB 2023 !!!"	1	0000000000018A6 56	push	rsi
Console.WriteLine("Welcome to VB 2023 !!!");	IL_0037: call void [System.Console]System.Console::WriteLine(string)	00000000000018A7 4831D2		rdx,rdx
Console.WriteLine("Welcome to VB 2023 !!!");	IL_003c: ldstr "Welcome to VB 2023 !!!"		00000000000018AA 65488B5260		rdx,[gs:rdx+60h]
Console.WriteLine("Welcome to VB 2023 !!!");	<pre>IL_0041: call void [System.Console]System.Console::WriteLine(string</pre>)	0000000000018AF 488B5218		rdx,[rdx+18h]
Console.WriteLine("Welcome to VB 2023 !!!");	IL_0046: ldstr "Welcome to VB 2023 !!!"		0000000000018B3 488B5220	mov	rdx,[rdx+20h]
Console.WriteLine("Welcome to VB 2023 !!!");	IL_004b: call void [System.Console]System.Console::WriteLine(string IL 0050: ldstr "Welcome to VB 2023 !!!")			
Console.WriteLine("Welcome to VB 2023 !!!");	IL 0055: call void [System.Console]System.Console::WriteLine(string	、 III	0000000000018B7 488B7250		rsi,[rdx+50h]
Console.WriteLine("Welcome to VB 2023 !!!");	IL 005a: ldstr "Welcome to VB 2023 !!!"	,	00000000000018BB 480FB74A4A		rcx,word [rdx+4Ah]
Console.WriteLine("Welcome to VB 2023 !!!");	IL 005f: call void [System.Console]System.Console::WriteLine(string	x			101 000
Console.WriteLine("Welcome to VB 2023 !!!");	IL 0064: ldstr "Welcome to VB 2023 !!!"		00000000000018C0 4D31C9		r9,r9
Console.WriteLine("Welcome to VB 2023 !!!");	IL_0069: call void [System.Console]System.Console::WriteLine(string)	00000000000018C3 4831C0		rax,rax
Console.WriteLine("Welcome to VB 2023 !!!");	IL_006e: ldstr "Welcome to VB 2023 !!!"		0000000000018C6 AC		
Console.WriteLine("Welcome to VB 2023 !!!");	IL_0073: call void [System.Console]System.Console::WriteLine(string		0000000000018C7 3C61		al,61h
Console.WriteLine("Welcome to VB 2023 !!!");	IL_0078: ldstr "Welcome to VB 2023 !!!"				
Console.WriteLine("Welcome to VB 2023 !!!");	IL_007d: call void [System.Console]System.Console::WriteLine(string IL_0082: ldstr "Welcome to VB 2023 !!!")	0000000000018C9 7C02		short 0000`0000`0000`18CDh
Console.WriteLine("Welcome to VB 2023 !!!");	IL_0082: Idstr Welcome to VB 2023 !!!" IL 0087: call void [System.Console]System.Console::WriteLine(string	Ň	0000000000018CB 2C20		al,20h
Console.WriteLine("Welcome to VB 2023 !!!");	IL 008c: ldstr "Welcome to VB 2023 !!!"		0000000000018CD 41C1C90D		r9d,0Dh
Console.ReadKey();	IL 0091: call void [System.Console]System.Console::WriteLine(string)	0000000000018D1 4101C1		r9d,eax
	IL 0096: 1dstr "Welcome to VB 2023 !!!"		0000000000018D4 E2ED	loop	0000`0000`0000`18C3h

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).

Main() : void ×	Main():void ×	Main(): void ×		
// CompileDecoy ReplaceReal SC, Version=1.0.0.0, 🔥 .method private hidebysig static			; void CompileDecoy_ReplaceReal_SC.Program.Main()	
// CompileDecoy ReplaceReal SC.Program	<pre>void Main () cil managed</pre>			
using System;		0000000000001690 56 000000000001691 4883EC30	push	rsi
using system;		; IL 0000		rsp,30h
private static void Main()		000000000001895 48883524180000		rsi,[rel 30C0h]
		00000000000189C 488B0E		rcx,[rsi]
Console.WriteLine("Welcome to VB 2023 !!!");	.maxstack 1	00000000000189F FF1503180000		<pre>qword [rel 30A8h] ; void [System.Console]System.Console::WriteLine(string)</pre>
Console.WriteLine("Welcome to VB 2023 !!!");	.entrypoint	; IL_000a 000000000018A5 488B0E		rcx,[rsi]
Console.WriteLine("Welcome to VB 2023 !!!");	IL 0000: ldstr "Welcome to VB 2023 !!!"	0000000000018A8 FF15FA170000	call	<pre>gword [rel 30A8h] ; void [System.Console]System.Console::WriteLine(string)</pre>
Console.WriteLine("Welcome to VB 2023 !!!");	IL 0005: call void [System.Console]System.Console::WriteLine(string)			
Console.WriteLine("Welcome to VB 2023 !!!");	IL 000a: ldstr "Welcome to VB 2023 !!!"	0000000000018AE 488B0E		rcx,[rsi]
Console.WriteLine("Welcome to VB 2023 !!!");	IL 000f: call void [System.Console]System.Console::WriteLine(string)	00000000000018B1 FF15F1170000 ; IL 001e		<pre>qword [rel 30A8h] ; void [System.Console]System.Console::WriteLine(string)</pre>
Console.WriteLine("Welcome to VB 2023 !!!");		00000000000018B7 488B0E		rcx.[rsi]
Console.WriteLine("Welcome to VB 2023 !!!");	<pre>IL_0019: call void [System.Console]System.Console::WriteLine(string)</pre>			<pre>qword [rel 30A8h] ; void [System.Console]System.Console::WriteLine(string)</pre>
Console.WriteLine("Welcome to VB 2023 !!!); Console.WriteLine("Welcome to VB 2023 !!!");		; IL_0028		
	<pre>IL_0023: call void [System.Console]System.Console::WriteLine(string)</pre>	0000000000018C0 48880E 000000000018C3 FF15DF170000	mov call	<pre>rcx,[rsi] qword [rel 38A8h] ; void [System.Console]System.Console::WriteLine(string)</pre>
Console.WriteLine("Welcome to VB 2023 !!!");	IL 0028: ldstr "Welcome to VB 2023 !!!" IL 002d: call void [System.Console]System.Console::WriteLine(string)	; IL 0032		quora (rez bonon) y tozo (bybeen console jayacen consoler run zeezne (arrang)
Console.WriteLine("Welcome to VB 2023 !!!");	IL 0032: ldstr "Welcome to VB 2023 !!!"	0000000000018C9 488B0E		rcx,[rsi]
Console.WriteLine("Welcome to VB 2023 !!!");	IL 0037: call void [System.Console]System.Console::WriteLine(string)	0000000000018CC FF15D6170000		<pre>qword [rel 30A8h] ; void [System.Console]System.Console::WriteLine(string)</pre>
Console.WriteLine("Welcome to VB 2023 !!!");	IL 003c: ldstr "Welcome to VB 2023 !!!"	; IL_003c 000000000018D2 488B0E		rcx,[rsi]
Console.WriteLine("Welcome to VB 2023 !!!");	IL 0041: call void [System.Console]System.Console::WriteLine(string)	0000000000001802 4688002 0000000000001805 FF15CD170000	call	<pre>qword [rel 30A8h] ; void [System.Console]System.Console::WriteLine(string)</pre>
Console.WriteLine("Welcome to VB 2023 !!!");				
Console.WriteLine("Welcome to VB 2023 !!!");	IL_004b: call void [System.Console]System.Console::WriteLine(string)	00000000000018DB 48880E 00000000000018DE FF15C4170000		rcx,[rsi]
Console.WriteLine("Welcome to VB 2023 !!!");				<pre>qword [rel 30A8h] ; void [System.Console]System.Console::WriteLine(string)</pre>
Console.WriteLine("Welcome to VB 2023 !!!");	<pre>IL_0055: call void [System.Console]System.Console::WriteLine(string)</pre>	0000000000018E4 488B0E		rcx.[rsi]
Console.WriteLine("Welcome to VB 2023 !!!");	IL_005a: ldstr "Welcome to VB 2023 !!!" IL 005f: call void [System.Console]System.Console::WriteLine(string)	0000000000018E7 FF15BB170000		<pre>qword [rel 30A8h] ; void [System.Console]System.Console::WriteLine(string)</pre>
Console.WriteLine("Welcome to VB 2023 !!!");	IL 0064: ldstr "Welcome to VB 2023 !!!"	; IL_005a 0000000000018ED 48880E		
Console.WriteLine("Welcome to VB 2023 !!!");	IL 0069: call void [System.Console]System.Console::WriteLine(string)	00000000000018ED 48880E 00000000000018F0 FF1582170000	mov call	<pre>rcx,[rsi] gword [rel 38A8h] ; void [System.Console]System.Console::WriteLine(string)</pre>
Console.WriteLine("Welcome to VB 2023 !!!");	IL 006e: ldstr "Welcome to VB 2023 !!!"	; IL 0064		ding free pointing a nore following parentening retrie (arrang)
Console.WriteLine("Welcome to VB 2023 !!!");	IL_0073: call void [System.Console]System.Console::WriteLine(string)	0000000000018F6 488B0E		rcx,[rsi]
Console.WriteLine("Welcome to VB 2023 !!!");		0000000000018F9 FF15A9170000 ; IL 005e		<pre>qword [rel 30A8h] ; void [System.Console]System.Console::WriteLine(string)</pre>
Console.WriteLine("Welcome to VB 2023 !!!");	<pre>IL_007d: call void [System.Console]System.Console::WriteLine(string)</pre>	0000000000018FF 488B0E		rcx.[rsi]
Console.WriteLine("Welcome to VB 2023 !!!");	IL_0082: ldstr "Welcome to VB 2023 !!!"	0000000000001902 FF15A0170000		<pre>qword [rel 30A8h] ; void [System.Console]System.Console::WriteLine(string)</pre>
Console.WriteLine("Welcome to VB 2023 !!!");	<pre>IL_0087: call void [System.Console]System.Console::WriteLine(string) IL_008c: ldstr "Welcome to VB 2023 !!!"</pre>			
Console.WriteLine("Welcome to VB 2023 !!!");	<pre>IL_008c: Idstr "Welcome to V8 2023 !!!" IL_0091: call void [System.Console]System.Console::WriteLine(string)</pre>	0000000000001908 48860E 0000000000001908 FF1597170000	mov call	<pre>rcx,[rsi] gword [rel 30A8h] ; void [System.Console]System.Console::WriteLine(string)</pre>
Console.ReadKey();	IL 00991: call Void [System.Console]System.Console::WriteLine(string)	; IL 0082		dword [ler sowoul : vorm [system.consore]system.consore::writerine(string)
·}	IL 009b: call void [System.Console]System.Console::WriteLine(string)	000000000001911 488B0E		rcx,[rsi]

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

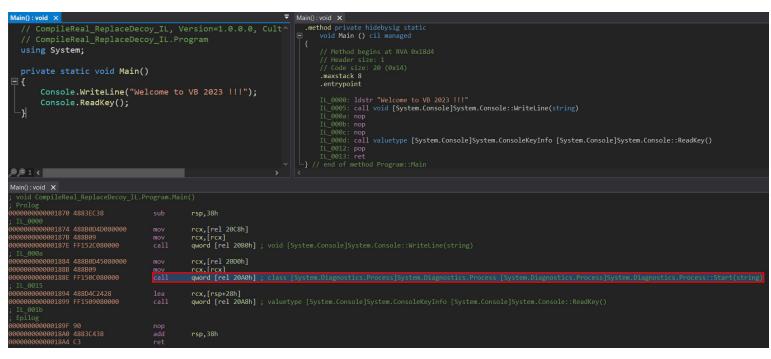


Figure 31: R2R stomping – patched IL code (pre-compiled intact).

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/readytorunformat.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/readytorunoverview.md.