UNPACK YOUR TROUBLES[1]: .NET PACKER TRICKS AND COUNTERMEASURES

Marcin Hartung
ESET, Poland

Email hartung@eset.pl

ABSTRACT

Nowadays, .NET samples are increasingly common, necessitating specialized techniques for processing and analysis, especially when obfuscation is used: .NET packers have many tricks up their sleeves, but fortunately we do too.

A skilled researcher can often glance inside 'good old-fashioned' native executables and see what they do despite protection with strong packers. .NET files, however, are different.

Analysing clean .NET files with dedicated tools shows us almost everything, but if the file is obfuscated we sometimes see nothing at all. In .NET analysis we face one main obstacle – complex runtime technology, which introduces some level of abstraction and therefore makes debugging harder.

This paper combines the analysis of methods collected from various sources with techniques originating with the author's own experience, in order to improve sample management. It describes simple tricks for getting strings after packer decryption or logging APIs used as well as some more sophisticated examples.

All the problems addressed relate to real cases often encountered in the context of commercial packers or of custom protectors used by malware.

Such tricks can be used for single analyses for adding breakpoints in locations of interest or as building blocks for constructing a powerful tool for analysing .NET samples.

1. INTRODUCTION

The analysis of large volumes of .NET samples is a growing problem. The escalating growth in numbers of observed detections of managed[2] samples is illustrated in Figure 1.

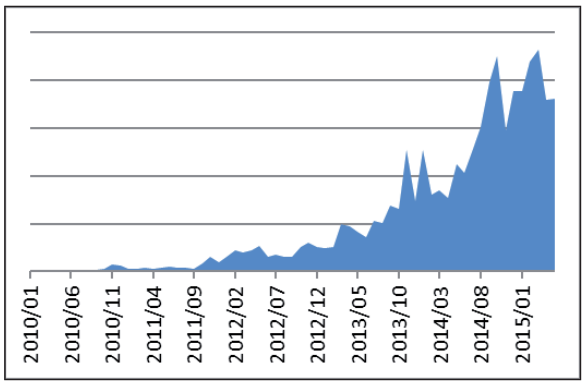

Figure 1: .NET malware – growing number of detections.

[1] http://en.wikipedia.org/wiki/Pack_Up_Your_Troubles_in_Your_Old_Kit-Bag.

[2] Managed code is another term for .NET samples.

Samples are often protected with packers, which makes reverse engineering harder.

This paper describes methods for the reverse engineering of .NET samples, the problems encountered if these files are obfuscated, and some tricks that can help in analysis.

2. ANALYSIS – STATE OF THE ART

A compiled .NET executable is still object-oriented, which means that the whole structure of the file – consisting of classes, fields and methods – is stored within the file itself [1].

Furthermore, the names of the objects created by programmers are also present in the executable (see Figure 2).

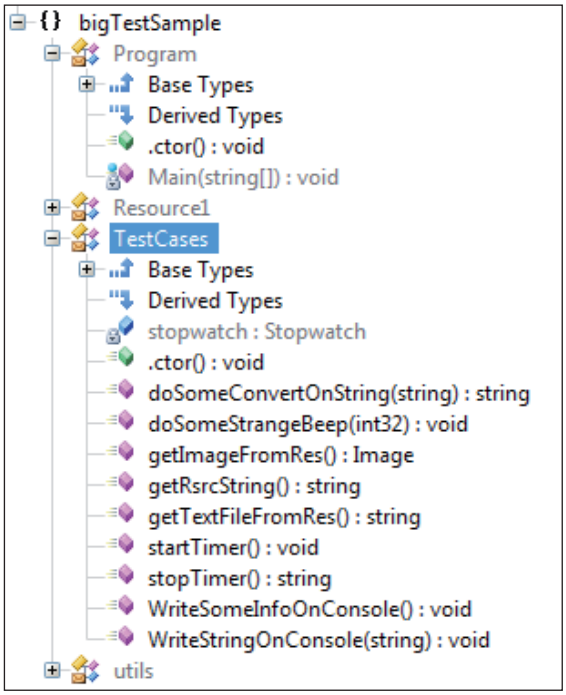

Figure 2: User names for objects.

The presence of these names facilitates the reverse engineering of .NET files. This situation is analogous to analysing native *Windows* samples with a .pdb symbols file.

Programming logic is maintained by generating intermediate code called CIL [1]. When the sample is executed it is compiled a second time to native code.

2.1. Static analysis

There are many good tools that help with the static analysis of .NET files. An open-source example is *IlSpy* [2], which can show the structure of the analysed program (Figure 2) as well as method CIL code or decompiled logic in C# or VB (see Figure 3).

```
// bigTestSample.utils
public static int getRand()
{
    Random random = new Random();
    return random.Next(100);
}
```

Figure 3: Decompiled user method.

Sometimes it is necessary to go deeper and look into .NET metadata. This is where information about program structure is kept – it is implemented by describing the framework hierarchy with tables as shown in Figure 4; *CFF Explorer* [3] is a useful tool for analysing it.

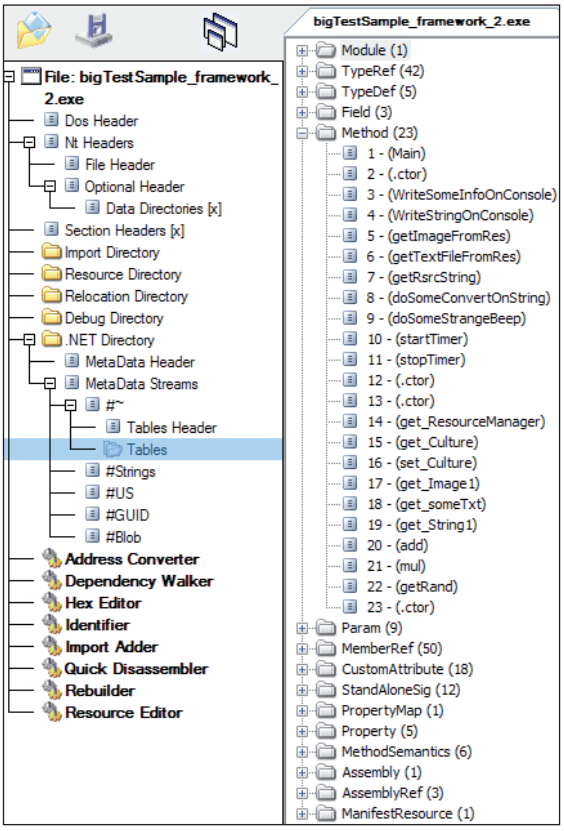

Figure 4: .NET file format.

The reverse engineering of .NET samples that have not been packed is quick and easy, since they can be decompiled trivially – it is almost like reading the source files of a project. This explains why not only malicious .NET samples but also clean .NET projects are so often obfuscated.

It is sometimes difficult to perform static analysis when the samples are packed and it is often the case that static analysis does not provide any information at all.

2.2 Debug

Debugging from the protector code up to the decrypted user code is a good option for handling packed native samples.

Debugging is problematic in managed code because of intermediate CIL method code, which is compiled a second time to its native equivalent during execution if the program flow requires it.

There are many layers of runtime logic that are involved in that process, so sometimes it is hard to find interesting parts of user code because the .NET runtime is so large.

There are some tools that allow debugging in 'mixed mode'. This enables breakpoints to be set on CIL code, but debugging to be done in native mode [4]. Another approach is tracing code in a decompiler, which some plug-ins make possible [2]. These tools are helpful but sometimes they can't cope with protected samples.

A particularly resistant debugger for obfuscation is *WinDbg*, which contains a special extension for debugging managed programs – SOS.dll [5]. It is not able to trace into or decompile CIL code, but it can analyse and list internal .NET runtime objects, which helps in deep research.

Our tricks for analysing packed .NET samples (described in section 4) are shown using the example of *WinDbg* with the SOS.dll plug-in.

2.3 Emulating

Emulating samples is a widely used approach in the anti-malware industry. In .NET it is an especially promising technique because of the availability of runtime sources [6, 7]. A complete emulator is a powerful tool, but building one is a laborious and complicated process. It exceeds the scope of this article but some reference is made to emulating packed .NET samples.

As will be described in the section about specific obfuscations, .NET protectors often put their code in many specific places in the program – e.g. every string load or API call. The decryption process is performed for every object the moment it is used – meaning that if the emulator doesn't reach that point, the information won't be obtained. This is problematic because program flow can be determined by triggers such as user action or some special network connection, which sometimes can't be simulated in an artificial environment.

A .NET runtime emulator can provide a lot of interesting data, but sometimes it won't help with packed samples.

2.4 Deobfuscators

.NET protectors effectively hide interesting parts of packed samples but are themselves quite simple programs. Every file packed with a popular protector can be pre-processed with a static deobfuscator. The most popular static deobfuscator is the open-source *de4dot* [8]. After unpacking, samples look almost the same as they did before obfuscation and they are ready for analysis.

Unfortunately, static deobfuscators are not a perfect solution. They need updating for every new version of a packer. If samples are protected with more than one packer, sometimes the output is not as good as we would like. But the main problem is custom packers – often seen in malware – where each version of the protector is used only for a few samples and writing an unpacker for every variant is pointless.

3. WHAT PACKERS REALLY DO

Methods of obfuscation used by .NET protectors can be described in three categories:

- basic
- hiding user code
- related to .NET runtime structures.

3.1 Basic – rename symbols

As shown in Figure 2, the .NET executable contains all user names of classes, methods and fields. Stripping these names is a basic feature of every .NET packer. Some example of renaming are shown in Figure 5.

```
internal class Registration
{
    private bool registrationFlag = false;
    private bool amIRegistered()
    {
        return this.registrationFlag;
    }
    private void registerMe(bool val)
    {
        this.registrationFlag = val;
    }
}
```

```
internal class cvbnc
{
    private bool zdfvjkl = false;
    private bool kjjgfja()
    {
        return this.zdfvjkl;
    }
    private void dfhhcfg(bool cvb)
    {
        this.zdfvjkl = cvb;
    }
}
```

Figure 5: Renaming user symbols.

There is even a simple obfuscator built into Visual Studio – Dotfuscator Community Edition [9]. It can only change these user names, which confirms how important this obfuscation is.

At this point it should be noted that after stripping these labels, they cannot be retrieved. For the compilation process it does not matter whether the names are descriptive or a random string of ASCII symbols.

3.2 Hide CIL code

In this section we describe methods for hiding CIL code. The first method is very similar to the approach known from native packers such as UPX. Then we compare the differences between native and .NET packers for the code entry point.

Two last tricks are distinctive for .NET – they use runtime internals.

3.2.1 Assembly.Load

Old-style native packers typically cover original code with one or a few layers of protection so as to hide it from static analysis (analysis where samples aren't run). For unpacking, the best option is often to debug the packer code until the jump to the original entry point (OEP) is reached – at that moment the interesting part of the file is unpacked and can be dumped and analysed.

Sometimes we can see a similar approach in .NET – original code is encrypted and/or compressed – during execution the whole decrypted file is loaded into runtime. APIs such as Reflection.Assembly.Load [10] are typically used for this action (see Figure 6).

The situation described above is used by the simplest packers like RPX [11] and Netshrink [12]. Sometimes malware also uses packing to hide payload (e.g. some Bladabindi families). More complex packers might use this feature only as the first layer of protection.

This obfuscation can be handled by setting a breakpoint during debugging near to the call to the Reflection.Assembly.Load API – the whole decrypted file will be loaded so it can simply be dumped. Emulation should also help in simpler cases.

```
byte[] rawAssembly;
using (MemoryStream memoryStream = new MemoryStream())
{
    manifestResourceStream.CopyTo(memoryStream);
    rawAssembly = memoryStream.ToArray();
}
Assembly assembly = Assembly.Load(rawAssembly);
Console.WriteLine("Assembly loaded\n");
assembly.EntryPoint.Invoke(null, new object[]
{
    args
});
Console.WriteLine("Assembly invoked\n");
```

Figure 6: Example of Assembly.Load usage.

3.2.2 Entry point

In native code the entry point is an RVA value which points to some place in the code. Old-fashioned packers add some extra decrypting code at the last section (sometimes adding a few new sections), set the entry point there and, after decryption, jump to the OEP.

A compiled .NET file retains the whole structure of classes described with some internal tables so the EntryPointToken value in the CLI header [1] can't be a data address – it is a token to the method that will be executed first. See Figure 7.

SizeOfUninitializedData	000000A4	Dword	00000000	
AddressOfEntryPoint	000000A8	Dword	000046BE	.text
BaseOfCode	000000AC	Dword	00002000	
Flags	00000218	Dword	00000003	
EntryPointToken	0000021C	Dword	06000001	
Resources RVA	00000220	Dword	00002420	

Figure 7: Native (up) and managed (down) entry point.

.NET packers rarely change the entry point to their code because they have better options, like type initializer.

3.2.3 .cctor() – type initializer

Type initializer is a powerful and extremely useful feature that can be used in a .NET packer. At runtime there is a special static method named .cctor, which is used by the class to initialize itself (static arrays, etc.). The method is called once for each type (class, interface or value type) before giving access to any of its methods or fields [1].

How can packers use it? This is a perfect place to put the code responsible for unpacking the currently used class. It runs before every called method, so if the method Program::Main() is set as entry point, the method Program::.cctor() will be executed immediately beforehand.

The initializer of <Module> class is an even better solution for packers. This is an invisible public space in the .NET file structure and contains all global fields and methods in the project [1]. If <Module> has the .cctor() method, then this type initializer is executed before any other class initializer or entry point method. We can see some decryption methods being called from <Module>::.cctor() in Figure 8.

Packers like Confuser or ConfuserEx[3] [13] use this feature to hide the CIL code of user and packer methods. All instruction opcodes are replaced with a crypted equivalent – only

[3] Confusers are powerful open-source packers – they can be used for learning the basics of .NET protectors.

```
.class private auto ansi '<Module>'
{
    // Methods
    .method private hidebysig specialname rtspecialname static
        void .cctor () cil managed
    {
        // Method begins at RVA 0x2050
        // Code size 6 (0x6)
        .maxstack 8

        IL_0000: call void eDyvLNuENDt5Xi1U2C::ewwmj81c4$P25()
        IL_0005: ret
    } // end of method '<Module>'::.cctor

} // end of class <Module>
```

Figure 8: <Module>::.cctor() – first loaded function.

decrypting methods, called from <Module>::.cctor(), are represented with real code. Decoding is done at the beginning of execution, so decrypted method CIL code will be ready at the time of compiling to native code.

The type initializer feature is similar to the well-known TLS callback trick used by native malware and packers [14]. Both solutions keep the original entry point but allow the addition of some code that is executed beforehand.

So if you want to analyse a packed .NET sample but don't know how to begin, checking the <Module>::.cctor() function can be a good place to start.

3.2.4 Hook compileMethod()

ICorJitCompiler::compileMethod() is the main routine used to ask the .NET runtime to create native code for a method [6]. This is done before the first execution of the method. Input arguments are descriptions of managed methods with CIL code – output is a compiled native equivalent.

This is an interesting place for analysing code execution – methods appear in the calling order and there is access to both kinds of code, CIL and native. Some simple analysis tools make use of this by adding some hooks and performing some analyses on CIL code [15, 16].

.NET packers also 'know' about it. Eziriz [17] and CodeWall [18] remove CIL code from user methods – there is nothing to see in static analysis. To restore the code they hook compileMethod() at runtime – they prepare the original code from some decryption and transfer it as input parameters for compiling. The functions responsible for setting hooks are called from the <Module>::.cctor() initializer.

3.3 .NET special

This section describes some of the tricks that are specialized for .NET – tricks fit for high-level file format.

3.3.1 User strings crypto

As noted before, .NET files keep file format descriptors in separate components. There is the metadata stream named '#US', which is a physical representation of logical user strings [1]. Separate strings can be accessed by a special token that points to the beginning of the string entry – used by CIL opcode Ldstr.

Many packers offer the concealment of these strings. Single opcode, which loads strings on the method stack, is changed to some logic for decryption of the hidden string. This is done via a call to a decrypt function that returns a string value – it is also loaded onto the method stack so the method flow is the same (see Figure 9).

```
ldstr "for line 0"
call void [mscorlib]System.Console::WriteLine(string)

ldc.i4 -1666591847
call !!0 '<Module>'::''<string>(uint32)
call void [mscorlib]System.Console::WriteLine(string)
```

Figure 9: Ldstr opcode and crypted string pattern.

Crypted string data is kept in a manifest resource, packer static data fields, or even in the #US stream, but changed to a protected form.

3.3.2 Resource resolver

Here is an interesting example of how a packer can use some high-level runtime features to hide some part of a file by hiding managed resources with a special resolver.

There is a runtime-defined event, AppDomain.ResourceResolve [10], which occurs if there are errors during the loading of the .NET resource (manifest resources are part of metadata and are described with the ManifestResource table [1]). So the packer can remove user resources and add its decrypt&load method as a resolver. During static analysis user resources are not visible but the code used is the same.

This feature can be seen in SmartAssembly [19].

3.3.3 Anti-decompiler

Decompiling the CIL code into high-level programming language makes .NET reverse engineering faster and more pleasant, so it is great a place for packers to show themselves.

.NET metadata is very complicated and sometimes tricks can be used to confuse decompilers. ConfuserEx [13] uses an undocumented bit flag which misleads metadata parsing. Eziriz [17] adds some fake code to methods which is never executed (the first opcode is a jump over the problematic block of data), but it is enough to make decompilation impossible.

If we still want to analyse a sample that is obfuscated with anti-decompile tricks we must prepare it – often tools like *de4dot* are enough to settle this problem.

3.3.4 Flow obfuscation

Obfuscation on CIL opcodes can sometimes be subtle. The main goal is to make static analysis difficult but the program's original order of execution must remain the same.

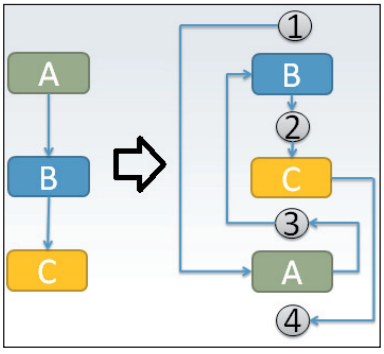

Figure 10: Flow obfuscation.

An example of the basic flow obfuscation is shown in Figure 10.

Basic blocks will be executed in the same order but physically are located elsewhere in the method code. This offers good protection against an automatic search for known code patterns.

The CIL code with obfuscated flow is larger and more difficult to analyse but its execution time can still be the same – basic obfuscation patterns are resolved during compilation to native code.

More sophisticated methods of flow obfuscation can mislead a decompiler and analysing in CIL opcodes becomes necessary. Sometimes packers also move some part of code to other methods to hide some APIs or logic.

4. OUR TRICKS

We have described some of the tricks used by .NET. Now it is time for ours. After research into obfuscators we can see some similar patterns of protection, which can be used during analysis.

Proofs of concept for the methods described are shown with scripts for *WinDbg*. This debugger is chosen because of its ability to cope with packed samples and the SOS.dll plug-in, which is irreplaceable for looking into internal runtime structures. Good tutorials that can help with getting used to this tool are [20] and [21].

4.1 Start working with SOS.dll

For debugging we must configure our *WinDbg* – add a path to symbols and load the SOS.dll plug-in. The important thing is to check which version of runtime is used by the sample, which determines which CLR library is in use: versions 2.0 up to 3.5 work with mscorwks.dll; clr.dll is used by version 4.0 and newer. This is needed for the appropriate set-up. See Figure 11.

```
$$ runtime v2+
sxe ld mscorwks
g
.loadby sos mscorwks
---------------------
$$ runtime v4+
sxe ld clr
g
.loadby sos clr
```

Figure 11: WinDbg setup.

During the debugging of a sample, our main tool is the setting of breakpoints in interesting places to get some data (in an automatic analysis tool it can be replaced by the hooks).

For methods in CLR (mscorwks.dll or clr.dll) there are simple commands like bp or bs for managing breakpoints [22].

An interesting issue is setting breakpoints in the runtime API – which is mainly located in mscorlib.dll. This library is kept and loaded as a native image [23], which makes setting a breakpoint with the bp command and symbol impossible. Fortunately, the SOS.dll plug-in has a special feature – the bpmd command [20]. This allows us to set breakpoints in a native image as we do in a regular native dll – only with the API name.

If we want to use another debugger or find a place for some hooks, we don't have this convenience. In this case the structure of mscorlib_ni.dll must be parsed (complicated and undocumented) or we can use *WinDbg* bpmd once to find an interesting place in that particular runtime version, and use byte patterns found in other tools.

Solutions were tested for two runtime versions:

- 2.0.50727 – as v2+
- 4.0.30319 – as v4+

The logic for other versions should be similar but our scripts may need some modification.

4.1 API

The APIs used by a program are always interesting during reverse engineering.

CIL method code in .NET executable files refers to APIs by tokens. There are special metadata tables to translate them, such as MemberRef and TypeRef for runtime calls and MethodDef for user methods [1], which can also be used during static analysis.

Packers rarely hide it but even if those data are available, there is still no information about which APIs will really be called and in what order.

Debugging helps here – even in using sophisticated obfuscation, the call to the required API must be executed eventually. Runtime uses some logic for analysing and providing calls requested by user code by tokens. And this is a perfect place to set a breakpoint.

There is a MethodTable::MapMethodDeclToMethodImpl function in CLR which is called for most of the requested methods. It works on an internal runtime structure named MethodDesc [6]. SOS.dll has a special command to list it: dumpmd [5].

```
$$ runtime v2+
bp mscorwks!MethodTable::MapMethodDeclToMethodImpl
 "!dumpmd dwo (esp + 4)"
---------------------
$$ runtime v4+
bp clr!MethodTable::MapMethodDeclToMethodImpl
 "!dumpmd ecx"
```

Figure 12: WinDbg command for log API.

After setting the breakpoint, as shown in Figure 12, the program stops as each call is resolved and logs the method named, as shown in Figure 13. Changes in implementation come from differences in passing arguments in other runtime versions.

```
Method Name: System.Text.Encoding.get_UTF8()
Class:       6f49c6c4
MethodTable: 6f835e9c
mdToken:     06003516
Module:      6f431000
IsJitted:    yes
CodeAddr:    6f7bc730
```

Figure 13: API info.

The method described is generic and works for called runtime APIs as well as user-defined functions. We only get truly

resolved methods in the correct sequence in which they were called.

4.2 User strings

User-defined strings in programs provide other interesting information. As discussed in section 3.3.1, packers hide them from static analysis.

The first idea for getting that data during program execution is to find the place in runtime where string objects are created and analyse it. There is System::String class in mscorlib.dll, which includes a few constructors. At first sight this is a perfect place for a breakpoint, but after a closer look the special flag 'internalcall' can be seen on those definitions. Implementation of a method marked in such a way is provided by runtime [1].

After a deep search the function responsible for creating strings in runtime can be found, but it does not meet our expectations in terms of accessing user-defined strings. The method in CLR creates many runtime literal objects so the data we're interested in is a small percentage of the entire reported information.

A better solution is to look into string decryption logic in particular packers. As mentioned previously, sometimes there is some decryption from the protected container. It is usually necessary to change some data kept as byte arrays into string objects. Three solutions were found as a result of our research into many packers, see Figure 14.

```
$$ many samples use this one
$$ bpmd set breakpoint, bs modify it
!bpmd mscorlib.dll  System.String.CreateStringFromEncoding
bs 1 da @eax

$$ used by yano & babel
!bpmd mscorlib.dll System.String.Intern
bs 2 du @ecx + 0xc

$$ used by deepSea (sometimes also by system)
!bpmd mscorlib.dll System.Text.StringBuilder.ToString
bs 3 du poi(esi+4) + 0xc
```

Figure 14: WinDbg commands for getting packed user strings.

The first method covers the majority of tested packers including confusers, SmartAssembly and many others. The second was seen in Yano and Babel. The last one is used in DeepSea. With these breakpoints set, we get user strings the moment they are decrypted.

These breakpoints can sometimes also be triggered by regular program flow or runtime internal operations, so reported data are not always decrypted strings from the #US stream.

If strings are not obfuscated in the file (kept as plain text in the #US stream) but we want to log it in order to use it in the program, we can add another breakpoint – see Figure 15.

```
$$ runtime v2+
bp mscorwks!GlobalStringLiteralMap::GetStringLiteral
"r @$t0 = poi(ebx + 4) & 0xffff; du poi(ebx) l$t0"
-------------|--------
$$ runtime v4+
bp clr!StringLiteralMap::GetStringLiteral
"r @$t1 = poi(ebp + 8);
r @$t0 = poi($t1 + 4) & 0xffff ; du poi($t1) l$t0"
```

Figure 15: WinDbg commands for getting regular user strings.

4.3 LoadAssembly

As was described in paragraph 3.2.1, .NET packers sometimes decrypt some payload and load the whole new MZ file with dedicated APIs. A script to handle the most common case is shown in Figure 16.

```
$$ runtime v2+
$$ loaded MZ file
bp mscorwks!CLRMapViewOfFileEx + 0x26 "da eax"
---------------------
$$ runtime v4+
$$ MZ file to load
bp clr!AssemblyNative::LoadFromBuffer
"dd (edx - 4) l1; da edx"
```

Figure 16: WinDbg commands for getting loaded assemblies.

The loaded file appears at the moment of loading into runtime. Details of this process can be seen in runtime sources [6, 7].

5. CONCLUSIONS

Packer obfuscation makes it harder to reverse engineer .NET files. Almost every packer can rename user symbols, make decompiling impossible, or encrypt user strings. Debugging is also difficult – complex runtime is a big obstacle.

The solution is to use tricks that can help in particular situations. These are typically acquired with experience but sometimes it is good to get some advice – like from this paper.

The methods described here can be used during the process of debugging or in building an automated analysis system. They cover problems with hidden strings and APIs but a similar approach can be used for other problems.

REFERENCES

[1] Standard ECMA-335 Common Language Infrastructure (CLI). http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[2] IlSpy. http://ilspy.net/.

[3] CffExplorer. http://www.ntcore.com/exsuite.php.

[4] PEBrowseDbg64 Interactive. http://www.smidgeonsoft.prohosting.com/pebrowse-pro-interactive-debugger.html.

[5] SOS.dll (SOS Debugging Extension). https://msdn.microsoft.com/en-us/library/bb190764%28v=vs.110%29.aspx.

[6] .NET Home. https://github.com/Microsoft/dotnet.

[7] Shared Source Common Language Infrastructure 2.0 Release. https://www.microsoft.com/en-us/download/details.aspx?id=4917.

[8] de4dot. https://github.com/0xd4d/de4dot.

[9] Dotfuscator Community Edition 3.0. https://msdn.microsoft.com/en-us/library/ms227240%28vs.80%29.aspx.

[10] .NET Development. https://msdn.microsoft.com/en-us/library/ff361664%28v=vs.110%29.aspx.

[11] RPX – Rugland Packer for (.Net) eXecutables. https://rpx.codeplex.com/.

[12] .netshrink. http://www.pelock.com/products/netshrink.

[13] ConfuserEx. https://yck1509.github.io/ConfuserEx/.

[14] How Malware Defends Itself Using TLS Callback Functions. https://isc.sans.edu/diary/How+Malware+Defends+Itself+Using+TLS+Callback+Functions/6655.

[15] .NET Internals and Code Injection. http://www.ntcore.com/files/netint_injection.htm.

[16] SJITHook. https://github.com/UbbeLoL/SJITHook.

[17] Eziriz. http://www.eziriz.com/.

[18] CodeWall. http://www.codewall.net/.

[19] SmartAssembly. http://www.red-gate.com/products/dotnet-development/smartassembly.

[20] How Windows Debuggers Work: Managed-Code Debugging. https://www.microsoftpressstore.com/articles/article.aspx?p=2201303&seqNum=3.

[21] .NET malware: De-obfuscation, decryption and debugging - tips and tricks. http://h30499.www3.hp.com/t5/HP-Security-Research-Blog/NET-malware-De-obfuscation-decryption-and-debugging-tips-and/ba-p/6463402.

[22] Debugger Commands. https://msdn.microsoft.com/en-us/library/windows/hardware/ff539170%28v=vs.85%29.aspx.

[23] NGen Revs Up Your Performance with Powerful New Features. https://msdn.microsoft.com/pl-pl/magazine/cc163808%28en-us%29.aspx.