

## Using Reflection to Reduce the Size of .NET Executables

**Vasian Cepa**, [cepa@informatik.tu-darmstadt.de](mailto:cepa@informatik.tu-darmstadt.de), Software Technology Group, Darmstadt University of Technology, Germany

This article presents an object-oriented technique for reducing the size of .NET executables. Current binary compressors cannot be used to pack .NET executables because .NET makes use of a specially modified PE file format.

We will rely on reflection capabilities supported by .NET to pack .NET binaries using pure C# code. The solution is general and can be used with any .NET executable, no matter in what front-end language it was written.

### 1 INTRODUCTION

In this article, we will show how reflection capabilities of the .NET framework [5]<sup>1</sup> combined with data compression can be used to reduce the size of .NET executables. Binary file compressors for Windows executables, such as UPX [14], use a similar technique to decrease the size of binary files. However, binary compressors do not work with .NET pseudo-executables. The pure .NET solution, we present here can be implemented in any .NET language and does not require any native platform code. Our prototype tool called .NETZ based on the idea presented here can be downloaded from [8].

There are several benefits of reducing the size of executables. Section 6 explains that smaller executables load faster because of fewer disk accesses. The compressed executable files also consume less hard disk space. Compressed binaries make it also more difficult to disassemble the code<sup>2</sup>, which is relatively easy for .NET because of its metadata support.

The rest of this article is organized as follows. Section 2 discusses .NET executables file format. Section 3 explains the compression techniques used in the .NETZ tool. Then, we explain how to use .NET specific reflection techniques to pack single EXE files in section 4. Section 5 extends the technique to support also DLL files. We give some performance measurements of our solution in section 6. Section 7 discusses related work and conclusions.

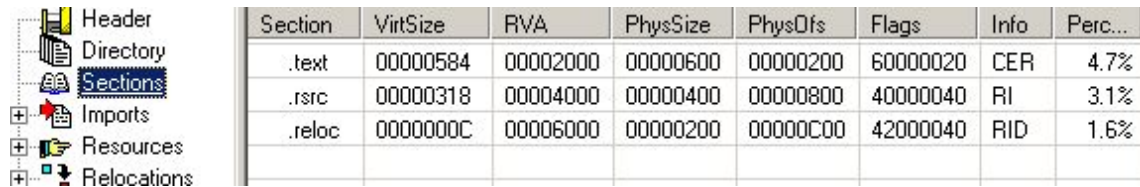
---

<sup>1</sup>All non-cited information about .NET comes from MSDN [7] documentation.

<sup>2</sup>Combined with encryption.

## 2 .NET EXECUTABLES

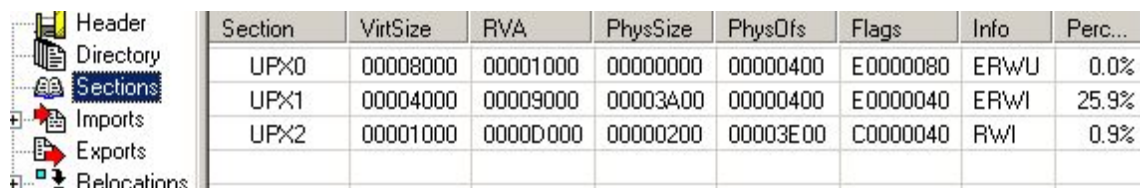
All .NET applications, despite of the front-end language used, are compiled into an *Intermediate Language (IL)* format, part of *.NET Common Language Infrastructure (CLI)* [2], which contains instructions to be executed by a stack-based virtual machine. The compiled IL is usually packed into *Assemblies*, which are physically stored into PE (portable executable) files [9]. The PE format [10] is used by all Windows executables. The CLI splits the code and resources of a .NET application in several segments called sections (Fig. 1) inside the PE file [9]. When an assembly is executed the data found in these segments is made available to the .NET runtime. This work is carried out in the EXE files by a call to `_CorExeMain` function exported by `mscorlib.dll` (*Microsoft Common Object Runtime Execution Engine*). For DLL (*Dynamic Link Libraries*) files a similar function `_CorDllMain` is called inside the usual `DllMain` [13]. The job of these functions is to properly initialize the Execution Engine (EE) of .NET framework and pass the CRI data found in the segments of the PE file to the EE.



Section	VirtSize	RVA	PhysSize	PhysOfs	Flags	Info	Perc...
.text	00000584	00002000	00000600	00000200	60000020	CER	4.7%
.rsrc	00000318	00004000	00000400	00000800	40000040	RI	3.1%
.reloc	0000000C	00006000	00000200	00000C00	42000040	RID	1.6%

Figure 1: Sample of a .NET executable PE sections

Binary compressors, e.g, UPX [14], use a similar technique to store the data in as compressed segments inside the EXE or DLL files (Fig. 2). The only difference is that the binary data of the original application are placed compressed in one section. Unlike .NET PE files, UPX places its own code in the first code section to access the compressed section, decompress it in memory and map it to the application address space.



Section	VirtSize	RVA	PhysSize	PhysOfs	Flags	Info	Perc...
UPX0	00008000	00001000	00000000	00000400	E0000080	ERWU	0.0%
UPX1	00004000	00009000	00003A00	00000400	E0000040	ERWI	25.9%
UPX2	00001000	0000D000	00000200	00003E00	C0000040	RWI	0.9%

Figure 2: Sample of a UPX packed DLL PE sections

The *.NET Common Language Runtime (CLR)* functions expect to be able to access PE section data directly [2]. It is impossible to modify a tool, such as UPX, to unzip the data from one section and make the data looks like as it originates from the sections expected by the CLR functions. For these reason, there are currently



no binary compressors that work with .NET executables. A possible workaround would be to compress the data segments of .NET PE file and then place starter code which decompresses the data at run-time and passes the decompressed data to the CLR functions as `byte` arrays. However, the details may be different between different .NET releases and platforms and may require CLR support. Ideally, it would be better if Microsoft supported this as option in .NET in the future. This would made .NET executables have size comparable to Java [6] JAR files.

There exists, however, another generic .NET alternative to this problem, which does not use any native platform code. We will present this pure .NET solution in this article. The technique does not modify the PE files. It works at a higher level and makes use of the fact that .NET CRI executables contain metadata and that .NET allows accessing those metadata via its Reflection API [1]. We will show first how to pack the main executable file of an application. Then, we will describe how the technique can be extended for .NET DLL files.

### 3 SELECTING A COMPRESSION LIBRARY

The first step for packing the .NET executable data is selection of a data compression library. The compression library should support data decompression in RAM and should input and output the data as `byte` arrays. The decompression library should also be relatively small in size given that it needs to be distributed with the compressed applications. The compression library does not need to be a .NET library. A native platform library can be accessed with a managed wrapper using .NET platform invoke (PInvoke). A .NET library is, however, preferable if it directly supports .NET types, e.g., `System.IO.Stream`, so that we can directly use `System.IO.MemoryStream` objects.

In our tool .NETZ, we used `#ziplib` [16] an open source implementation of several data compression algorithms implemented in .NET C#. The `#ziplib` library fulfills all the criteria above. The `#ziplib`'s license is also flexible to allow distribution of the decompression DLL as part of any open or closed source application. We chose to use the usual ZIP format [15] from the compression algorithms supported by `#ziplib`. The selection of the ZIP format was arbitrary and any other compression algorithm can be used.

The usual ZIP compression ratio for .NET executables is around 60%. We also need to distribute also the unzip code with the zipped version of an a .NET application. The size of `#ziplib` DLL (`ICSharpCode.SharpZipLib.dll`) is about 115KB. If we remove all other supported compression formats, apart from the ZIP format support and leave only the code to support unzipping, we will end up with a .NET DLL library of only 60KB<sup>3</sup>. This reduced version of the `#ziplib` library, named `zip.dll`, comes with the .NETZ tool for distribution with the compressed applications.

---

<sup>3</sup>We cannot compress the `zip.dll` library, because we need it to unzip the rest of the code.

The 60KB of `zip.dll` are the only size overhead of this method. If a .NET application is over 200KB, which is normally the case for a small .NET GUI executable, then the size of the compressed application plus the unzip library will still be under 200 KB. One could do better, however, with compression libraries written especially for this technique.

## 4 PACKING .NET EXECUTABLES

The process of creating a .NET self-contained packed executable follows the logical steps shown in Fig. 3. The .NETZ tool automates all the steps explained next and produces as output directly only the packed EXE file.

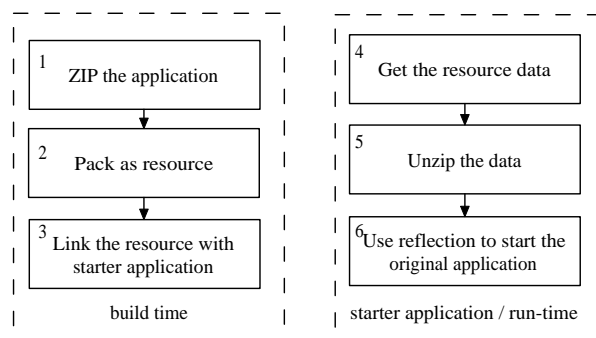


Figure 3: Steps for packing a .NET executable file

To ease the discussion of Fig. 3, we will suppose we have a .NET application named `app.exe`. As a first step, we compress `app.exe` as `app.zip` by using `#ziplib` programmatically. The .NETZ tool compresses the .NET executables as raw ZIP streams, without any ZIP directory information to keep the end size smaller. The .NETZ's compressed files cannot be read directly by normal ZIP applications expecting the ZIP file directory to be present.

The compressed data will be packed as part of a starter application `starter.exe`, which will decompress and start the original `app.exe` at run-time. The easiest way to pack `app.zip` as part of the `starter.exe` in .NET is to pack it as a .NET managed resource file. Managed resources of a .NET application are packed along with other IL data in the `.text` section of the PE file [9]. Other resources are packed in the data section. The following code will produce a valid .NET resource file `app.resources` (step 2 of Fig. 3):

```

FileStream fs = new FileStream("app.zip",
    FileMode.Open, FileAccess.Read);
byte[] data = new byte[fs.Length];
fs.Read(data, 0, data.Length);
  
```



```
fs.Close();
ResourceWriter rm = new ResourceWriter("app.resources");
rm.AddResource("appdata1", data);
rm.Close();
```

We have omitted the error handling code from all the C# examples to keep them simple. We named the packed resource to `appdata1` in order to access it later<sup>4</sup>. After we created the resource file, we need to create the `starter.exe` application. The starter application will load the resource file, get the compressed data, unzip them in memory and use reflection to start the `app.exe` application. The code invoked by the `starter.exe`'s `Main(string[] args)` method to access the original packed resource data is (step 4 of Fig. 3):

```
ResourceManager rm = new ResourceManager("app",
    this.GetType().Assembly);
byte[] data = (byte[])rm.GetObject("appdata1");
```

In step 5 of Fig. 3, we unzip the data in memory. The code for `#ziplib` is<sup>5</sup>:

```
string zipPath = "app.exe";
MemoryStream zipFile = new MemoryStream(data);

ZipFile zf = new ZipFile(zipFile);
ZipEntry ze = zf.GetEntry(zipPath);
Stream zs = zf.GetInputStream(ze);
byte[] uzdata = new byte[ze.Size];
zs.Read(uzdata, 0, uzdata.Length);
```

We use an instance of `System.IO.MemoryStream` to pass the zipped data to the `System.IO.Stream` format expected by `#ziplib`. We can then create a .NET assembly from the byte array (step 6 of Fig. 3):

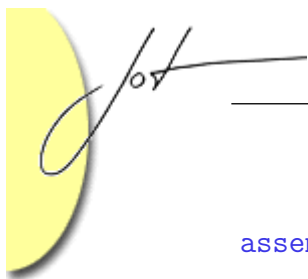
```
Assembly assembly = Assembly.Load(uzdata);
```

Once we have an assembly, the easiest way to properly activate `app.exe` is to invoke its entry point, which corresponds to the `Main(string[] args)` method in the original `app.exe`, passing to it the original command line arguments passed to the `Main(string[] args)` method of the starter:

---

<sup>4</sup>The .NETZ tool uses a unique global identifier (GUI) string to name the compressed EXE inside the resource file.

<sup>5</sup>The actual .NETZ implementation contains optimized code.



```
assembly.EntryPoint.Invoke(null, new object[] {args});
```

We used *null* as the first object argument of `Invoke` because `Main` is a static method and properly packed the original arguments as an object array. Alternatively, we can rely on reflection code to find the types in the assembly and invoke methods on them. This can be useful when the `app.exe` has no entry point, or when we want to invoke any other methods. The .NETZ tool generated applications also check whether the `Main()` method of the compressed application supports command-line arguments or not, to invoke the proper `Main()` version.

To compile `starter.exe` from `starter.cs` and pack the zipped data resource with it (step 3 of Fig. 3) we can use the following C# compiler command (supposing it is an windows executable):

```
csc /t:winexe /out:starter.exe starter.cs AssemblyInfo.cs  
/r:zip.dll /res:app.resources /win32icon:App.ico
```

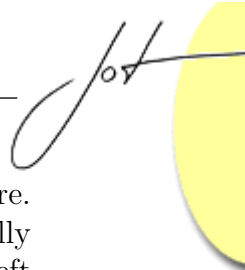
We can rename `starter.exe` back to `app.exe` later if we like. This way, we distribute `starter.exe` and `zip.dll` which are both smaller in size than `app.exe` alone.

We used two additional files `AssemblyInfo.cs` and `App.ico` as part of the compiler command. They both come from the original `app.exe` data. The `AssemblyInfo.cs` file contains version information about `app.exe` in the *Visual Studio* format. If we have the source of `app.exe` we can reuse `AssemblyInfo.cs` from it. Our .NETZ tool, however, uses reflection over `app.exe` to discover the assembly properties and generates a suitable `AssemblyInfo.cs` in the *Visual Studio* format. The .NETZ tool also extracts the main icon (`App.ico`) automatically from the original PE executable file and reuses it. .NETZ compiles the generated starter application programmatically using `.NET System.CodeDom.Compiler.ICodeCompiler` interface with the `CSharpCodeProvider`.

## 5 PACKING .NET DLL-S

If the `app.exe` application depends on other DLL-s, we normally do not need to do anything. However, sometimes we may prefer to compress also some of the DLL files. The technique that we will describe next works only for applications that make use of *.NET XCOPY paradigm* that is, when the DLL files are used by a single application. This technique will not work if the DLL files are placed in Global Assembly Cache (GAC), or shared by more than one application which is not aware of the technique.

An exception to this rule are the shared types used with .NET remoting [11]. In this case the common classes and interfaces should be placed in separate DLL shared



by the client and server, which is not compressed using the technique explained here. The reason is that the .NET CLR uses the shared types directly to automatically generate part of the marshalling code. The shared types should, therefore, be left uncompressed despite that both client and server applications could be both packed with the .NETZ tool.

.NET has a build-in mechanism for resolving types and assemblies. When the build-in mechanism fails to find an assembly we can provide the .NET CLR with an assembly of our own. This functionality is exposed by a hook in the `System.AppDomain` class. Every .NET application executes in an *application domain*. A single process may contain more than one application domain. The application domains are isolated from each other. We need to handle `AssemblyResolve` event for the current application domain:

```
AppDomain currentDomain = AppDomain.CurrentDomain;
currentDomain.AssemblyResolve += new
    ResolveEventHandler(MyResolveEventHandler);
```

This code need to be placed into the `Main` method of the starter application. In order for this event to be activated successfully at the right time, we have to place the `app.exe` assembly activation code described in section 4 in another separate method that will be called by the starter's `Main` method.

To illustrate the idea of packing .NET DLLs, let us suppose that `lib.dll` is a DLL file required by `app.exe` that we like to compress. First, we would link `app.exe` with the unzipped version of `lib.dll` as normally. Then, we would compress the `lib.dll` as `lib.zip`. We could pack `lib.zip` as a resource file with the starter application, as we did with the `app.zip`. This can be preferable if we want to have a single executable file which contains also the DLL files, an option not offered by the .NET CLR linkers. Alternatively, we may leave the packed DLL as separate file so we can update the DLL easier to a newer version. The .NETZ tool supports both these options.

Our custom `AssemblyResolve` handler will be called by .NET only when a type is missing. For this reason, we need to rename the packed DLL file to something different from the original file name `lib.dll`. If we were to leave the original name then .NET CLR will try to load the compressed file as it were uncompressed. The compressed file will then look like a corrupted DLL file to .NET CLR. We will have then an missing type run-time exception.

We need to rename the compressed version of `lib.dll` to something else. We can leave the name `lib.zip` or be creative and rename it, for example, to `libz.dll` as the .NETZ tool does<sup>6</sup>. The code to activate the DLL in `MyResolveEventHandler`

---

<sup>6</sup>Other alternatives based on this idea are possible. For example, we can save the `lib.zip` data in a SQL database table as a *blob* field and retrieve the data from there.

is shown next. We suppose that the zipped DLL is a file in the same directory as the starter application:

```
public static Assembly MyResolveEventHandler(object sender,
    ResolveEventArgs args) {
    int i = args.Name.IndexOf(',');
    string dllName = args.Name.Substring(0, i);

    // the dllName will equal "lib" in our example
    // we map it to the zipped file name
    dllName += ".dllz";

    // read the file and unzip the data as above
    // code omitted ...
    byte[] uzdata = ...

    return Assembly.Load(uzdata);
}
```

Additional information, e.g., the version of the expected DLL will be available as part of `args.Name` string passed to this event by the framework. This way, the types found in the zipped DLL will be resolved to the `AppDomain`. This method can also be used with EXE or any other .NET module files. We need to access some type defined in a module which cannot be resolved by .NET CLR for this event to fire.

The example code shown above was drastically simplified to illustrate the main points of the idea. The .NETZ tool generated applications can find the compressed DLL files in resources or in directories emulating the way .NET CLR finds the normal DLLs. The .NETZ tool implementation is fully compatible with .NET CLR search strategy and supports, for example, DLLs for multiple cultures (localization of application DLLs for different languages) and private DLL application paths.

Another important issue when implementing a custom `AssemblyResolve` handler is that .NET does not cache the supplies assemblies. .NET CLR caches the assemblies it loads itself so that the types in them are properly resolved without loading multiple copies of the same assembly. For custom `AssemblyResolve` handlers, assembly caching is, however, the responsibility of the programmer. .NET CLR may ask continuously for the same assembly, every time it need to resolve a type (even though the type may have been resolved before). When assembly caching is implemented, we should be careful to return the same physical assembly instance every time we are asked for the same assembly. Returning different instances will cause many copies of the same assembly to be loaded. Types of different loaded instances of the same assembly will be treated as if they were different types<sup>7</sup>. For

<sup>7</sup>Thanks to Taylor Brown (tbrown@ncsoft.com) for pointing this out.





this reason, .NETZ implements assembly caching as a `Hashtable` indexed by a key made up of all assembly identification data, such as, name, version, and culture.

## 6 PERFORMANCE MEASUREMENTS

We want to compare the startup time of .NET EXE files (a) non-compressed, using the normal PE files and (b) compressed files, which are unzipped in memory as described in section 4. One way to do these measurements is to use a model of the startup time of a .NET application. We will use the model of Fig. 4 in order to measure the startup time of .NET EXE-s files.

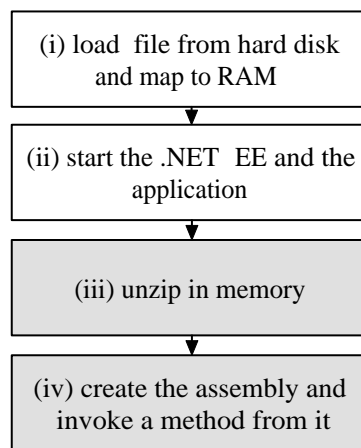


Figure 4: Time model of a .NET application startup

The last two segments (iii) and (iv) in Fig. 4 are present only with EXE files unzipped in memory (case b). The segment (ii) is the time .NET execution engine is made ready and started. This time is the same for both cases, compressed and not compressed, so we can ignore it. The time span of the segment (iv) is also too small in comparison with (iii) so we can also ignore it. We will compare time (i) for case the (a) above with time (i) + (iii) for case (b). In order not be effected by the buffer size when we read from a file we will use a buffer as long as the length of the file (`fi.Length`):

```
FileInfo fi = new FileInfo(file);  
byte[] buff = new byte[fi.Length];  
FileStream fs = new FileStream(file, FileMode.Open,  
    FileAccess.Read, FileShare.Read, (int)fi.Length);  
fs.Read(buff, 0, buff.Length);  
MemoryStream ms = new MemoryStream(buff);
```

The size of zipped files used is about 64% smaller than the unzipped version. Fig. 5 shows the average results of 25 tests for different file sizes from about 0.5 to 80 MB. Because of the simplification in our time model the time data should be treated as relative to each other and not as absolute values.

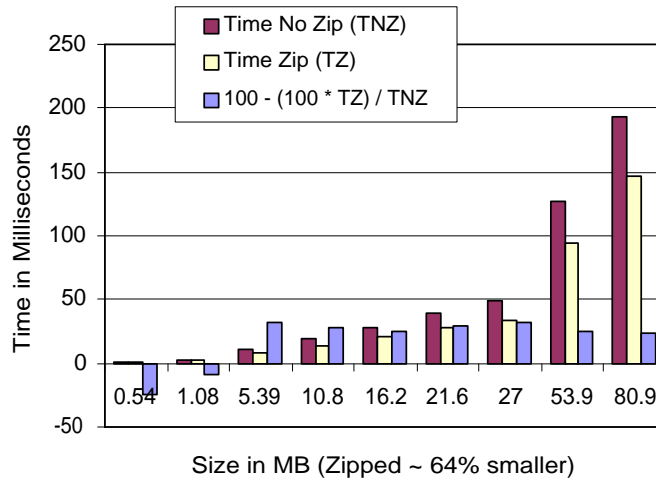


Figure 5: Performance measurements

The first of three data columns for each file measurement is the time required to load the file when it is not zipped (case a). The second one corresponds to loading the zipped file and unzipping it in memory (case b). The third column calculates the gain percentage. For small files under 5MB<sup>8</sup> the gain is negative because the compression technique is slower than for loading uncompressed files. The worst negative value for executables of 500KB is about 24.9%. However, for files bigger than 5MB we gain about 28% in average. For files bigger than 80MB we have more than 90% gain. A test case for files about 100MB (107.81MB) gave the data triple: 2003.98, 190.67, 90.49. We did not show this sample in Fig. 5 because this big gain rate will make it impossible to show clearly the values of the other samples. It is unclear why the gain grows so significantly for files bigger than 100MB. A possible cause could be that the operating system code or disk-cache is optimized to load faster small and medium size files, which make most of the used files.

The data of Fig. 5 shows that using compression makes loading of files faster because of fewer hard disk accesses by the operating system. We used #zlib [16] for these tests, however, compression code written especially for this technique could perform better.

<sup>8</sup>Again, because of the used model the real threshold value may be slightly different.



## 7 CONCLUSIONS

We presented here an OO .NET solution for reducing the size of .NET executables. This technique allows packing .NET EXE and DLL files in a compressed form, reducing the time it takes to load them and the size consumed in hard disk by .NET binary packages.

The compression of binary executables to reduce their size and improve the loading time is a technique used by binary compressors, e.g., UPX [14]. We, however, did not modify the PE file sections with native code. We made use of the fact that .NET framework exposes reflection [3] capabilities. We used the .NET object-oriented reflection capabilities to implement a pure .NET solution for reducing the size of .NET executables.

The technique we used for compressing .NET applications does not work with *.NET Compact Framework (CF)* [17]. The .NET CF does not implement the methods of the `Assembly` and `AppDomain` classes that make this technique possible. However, the devices that run .NET CF use compression by default for storing programs and data in the non-volatile device memory ([17] reports up to 2:1 compression ratio). This means that this technique may not be directly needed in .NET CF devices<sup>9</sup>.

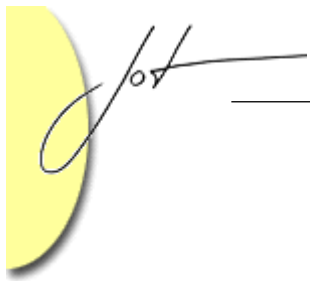
Resolving assemblies and providing custom assembly files to the .NET CLR, is done via `System.AppDomain` class. This is similar to creating custom class loaders in Java [4]. There are however subtle differences between the two models which are beyond the scope of this article. Java supports compression by default as part of the JAR (ZIP based) format. Java JAR files are used somehow similarly to PE files in .NET. A JAR file contains not only the code, but also resources and other meta-information as part of its manifest file.

There are several benefits of our specific .NET solution:

- It is a pure .NET object-oriented solution. We used C# here to demonstrate the code, but it could easy written in any .NET language. The code does not use of any particular information about the PE format, which makes it portable for example to be used also with Mono [12] executables in other platforms apart from Windows<sup>10</sup>.
- It does not change the programming style. Applications and their DLL-s files can be developed as usual. Only when we need to pack them we use the .NETZ tool. This can be part of the release build only and does not add any time to the usual development building process.
- The .NETZ generated starter is quite generic and independent of any particular .NET EXE or DLL files written in any .NET language. The solution

<sup>9</sup>Double-compression has usually no effect on size reduction.

<sup>10</sup>Not tested.



produces reduced size executables, which are undistinguishable from the original non-compressed ones to the end user.

- It helps to hide application code and protect investment in code. .NET disassemblers make it easy to view source code because of the .NET metadata/reflection support. The zipped resources are more difficult to disassemble. Combined properly with encryption and decryption in memory of sensitive application parts this technique can be used to hide sensitive intellectual properties in .NET applications making it harder to find the real code.

There are also a few liabilities. We require a starter application which is aware of this technique. This is problematic for DLL files. Another application not aware of the technique will not be able to use the packed DLLs. This limitation prevents compressed DLLs from being installed into .NET Global Assembly Cache (GAC) where they can be shared system-wide. Despite this, the technique presented here is easy to implement and offers potential benefits to .NET developers.



## REFERENCES

- [1] .NET Reflection API. *MSDN: <ms-help://MS.VSCC/MS.MSDNVS/cpref/html/frlrSystemReflection.htm>*, 2001.
- [2] ECMA-335 Common Language Infrastructure (CLI). <http://www.ecma-international.org/publications/standards/ecma-335.htm>, 2002.
- [3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern - Oriented Software Architecture - A System of Patterns*. Wiley, 1996.
- [4] Stuart Dabbs Halloway. *Component Development for the Java Platform*. Addison-Wesley, 2002.
- [5] J. Prosis. *Programming Microsoft .NET*. Microsoft Press, 2002.
- [6] K. Arnold, J. Gosling, D. Holmes. *Java Programming Language*. Addison Wesley, 2000.
- [7] .NET Framework MSDN Documentation. <ms-help://MS.VSCC/MS.MSDNVS/Netstart/html/sdkstart.htm>, 2002.
- [8] .NETZ - .NET Executables Compressor Tool. <http://www.st.informatik.tu-darmstadt.de/static/staff/Cepa/tools/netz/index.html>, 2004.
- [9] G. Nutt. *Distributed Virtual Machines: Inside the Rotor CLI*. Addison-Wesley, 2002-4.
- [10] M. Pietrek. Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format. *MSDN Magazine*, February 2002.
- [11] I. Rammer. *Advanced .NET Remoting (C# Edition)*. APress, 2002.
- [12] Mono .NET Implementation. <http://www.mono-project.com/about/index.html>, 2004.
- [13] Shared Source Common Language Infrastructure. <http://msdn.microsoft.com/net/sscli/>, 2004.
- [14] Ultimate Packer for eXecutables. <http://upx.sourceforge.net/>, 1996-2004.
- [15] ZIP File Format Specification. [http://www.pkware.com/products/enterprise/white\\_papers/appnote.txt](http://www.pkware.com/products/enterprise/white_papers/appnote.txt), 1989-2004.
- [16] #ziplib Homepage. <http://www.icsharpcode.net/>, 2003.
- [17] P. Yao and D. Durant. *.NET Compact Framework Programming with C#*. Addison Wesley, 2004.

## ABOUT THE AUTHORS



**Vasian Cepa** is a PhD student and research assistant at the Chair of Software Technology Group at the Darmstadt University of Technology, Germany. He can be reached at [cepa@informatik.tu-darmstadt.de](mailto:cepa@informatik.tu-darmstadt.de). See also <http://www.st.informatik.tu-darmstadt.de>.