IBIMA Publishing
Communications of International Proceedings
https://ibimapublishing.com/p-articles/46SE/2025/4640025/
Vol. 2025 (36), Article ID 4640025

Custom Packer as a Method of Obfuscating .NET 8 Console Applications*

Mateusz MOHR, Piotr GÓRNY and Kazimierz WORWA

Military University of Technology, ul. gen. Sylwestra Kaliskiego 2, Warsaw, Poland.

ORCID: 0009-0008-3841-2402
ORCID: 0000-0003-3181-6320
ORCID: 0000-0002-8153-958X

Correspondence should be addressed to: Mateusz MOHR, mateusz.mohr@student.wat.edu.pl

* Presented at the 46th IBIMA International Conference, 26-27 November 2025, Ronda, Spain

Abstract

This article presents how to prepare and utilize a packer solution to obfuscate console applications developed on the .NET 8 platform. The proposed solution uses encryption of the library containing the application logic and its dynamic loading during program execution. This process ensures that key code fragments remain hidden in encrypted form, and their decryption and execution takes place exclusively in the operating memory, without saving unencrypted data on the hard drive.

The developed solution consists of three components: a library containing the program to be hidden, an encryption tool and a decryption application that loads and invokes the encrypted program's functions in memory. The article discusses the architecture of the solution, the logic behind the operation of individual components, and a practical application example illustrating the process from library preparation to its launch in the execution environment.

A case study comparing the execution times of the original and secured versions of the application showed that the use of a packer does not significantly affect computational performance. At the same time, the advantages and limitations of the adopted method were identified.

The proposed solution provides a basis for further research into code protection methods in the .NET environment, in particular by combining packer mechanisms with classic obfuscation and software security techniques.

Keywords: obfuscation, packer, .NET 8, console application.

Introduction

The .NET platform is used to develop applications in multiple programming languages, including C#. The language is compiled into intermediate language (IL) code, which is a set of hardware-independent instructions. During application execution, the IL code is compiled by a Just-In-Time (JIT) compiler into hardware-specific machine code, eliminating the need to create separate implementations for different hardware architectures [1, 2]. To enable this functionality, the files used to run applications contain clearly defined information about the

Cite this Article as: Mateusz MOHR, Piotr GÓRNY and Kazimierz WORWA, Vol. 2025 (36) "Custom Packer as a Method of Obfuscating.NET 8 Console Applications " Communications of International Proceedings, Vol. 2025 (36), Article ID 4640025, https://doi.org/10.5171/2025.4640025

included types, fields, methods and more. As a result, they are easier to analyze than binary files compiled directly into native machine code [3].

Obfuscation is a code protection technique that involves transforming code into a form that is more difficult to understand and modify. In practice, this means that an obfuscated program requires more resources, such as time, money or computing power for analysis. From this perspective, distributing a program in compiled form rather than as source code is also a form of obfuscation, since machine code is much more difficult to analyze than source code [4].

In the presented case, obfuscation involves compiling the program logic into IL code, then encrypting the resulting library file using AES (Advanced Encryption Standard) encryption and executing the program in memory using a custom loader that calls the library functions. This method is an example of a packer implementation.

A **packer** is a tool or mechanism used to compress or encrypt executable files. A file packed using a packer remains an executable file, allowing it to be easily run by the user while making it difficult to analyze its contents. Packers that encrypt data are also referred to as crypters [5, 6].

Despite the wide availability of packaging tools, implementing a custom solution may be warranted in certain situations. Most importantly, not all existing tools can support the latest versions of development platforms, such as .NET 8, or adapt to specific settings used during project compilation. What's more, a custom solution provides full control over encryption and loading mechanisms, allowing them to be tailored to the current needs of the application, while further complicating code analysis with custom methods.

The following part of the article reviews the literature related to the issue at hand, presents the architecture of the proposed solution and the roles of its various components. Then the case study analyzed the operation of the loader based on the sample code. The next section discusses the advantages and limitations of the implemented solution. The article concludes with conclusions and suggestions for future research directions in code obfuscation and packaging in .NET.

Literature Review

Code obfuscation and protection of applications from reverse engineering are important aspects of software security, especially in the context of the .NET platform, which due to its open structure and ease of decompilation (e.g., using tools such as dnSpy) requires additional protection mechanisms [7].

Code obfuscation techniques in .NET are well known and widely used. They involve renaming symbols, obfuscating program logic and hiding metadata. Tools such as ConfuserEx, Obfuscar and Dotfuscator offer various levels of protection from code analysis [8]. However, traditional obfuscation is often not enough in the face of advanced reverse engineering methods, so packer solutions are used [9].

Packer solutions, known for their use in low-level environments (e.g. UPX for C/C++ applications), take a slightly different form in .NET. They involve encrypting the entire library (assembly) and dynamically loading it into memory at application startup. This approach effectively makes it impossible to analyze executable files without running the program [10].

One of the key aspects of effective packers is the encryption of binary data. Studies indicate that using AES or RSA encryption to hide the contents of a DLL library can effectively make it difficult for unauthorized persons to extract data [11]. In addition, memory protection techniques, such as restricting memory page permissions (e.g., execute/no-read), can provide an extra layer of security. [12, 13].

With the development of the .NET platform, especially version 8, there have been changes in JIT performance, application publication formats (single file publish, AOT) and file system interaction. These changes require adjustments to existing application protection techniques and create the need for new, more complex packers that are compatible with the latest runtime features [14].

The existing literature provides a solid theoretical and practical basis for further research on code obfuscation and protection techniques in .NET. Effectively protecting .NET applications from reverse engineering requires a combination of several techniques: obfuscation, encryption, dynamic loading and memory protection. Packer solutions offer the potential for a higher level of protection than traditional obfuscation, but their effectiveness depends on implementation and awareness of .NET runtime mechanisms.

Based on this knowledge, this article presents a proprietary solution that can be used to secure the code of .NET console applications.

Solution Architecture

The proposed solution consists of three main components that work together to secure the application: an application logic library, an encryption tool and a loader. Figure 1 illustrates the overall operation of the packager.

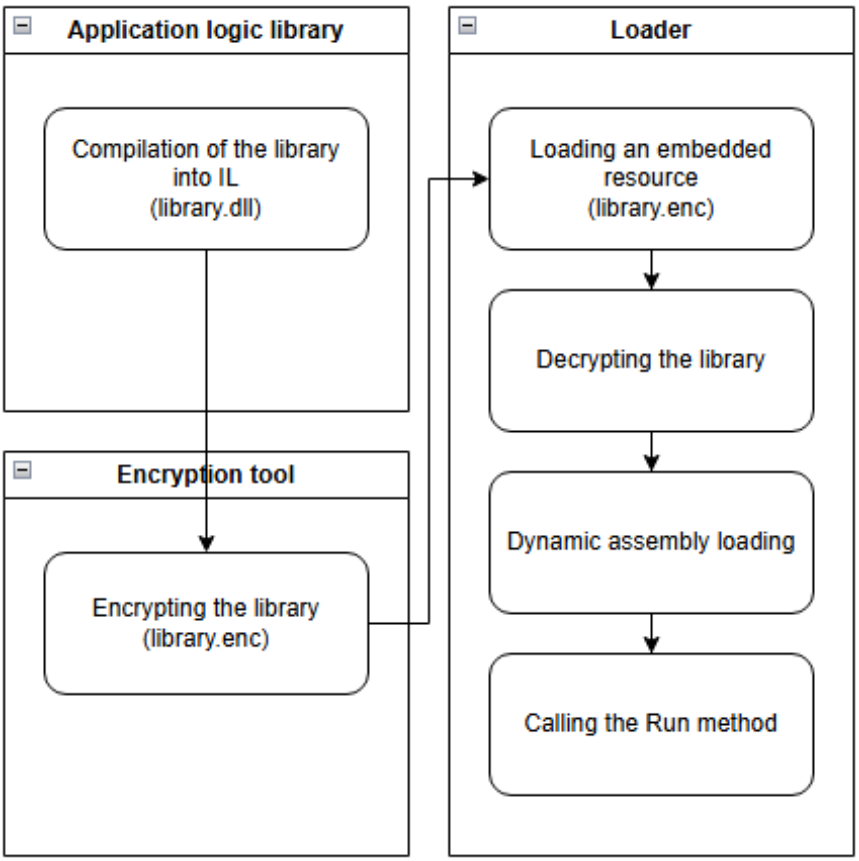

Fig. 1. Packer operation diagram.

Application logic library - the first and most important component of the solution is the library containing the application logic to be protected. This library is compiled as a separate module (DLL file containing IL code). In order to prepare the console application in such a way that it is compatible with the developed solution, it should be written as a library, replacing the Main function with the Run method, which is called by the loader. The Run method should be declared public and static to allow it to be easily located and executed using the loader's reflection mechanisms.

Encryption tool - the second component is an encryption program that acts as a packer during the data preparation phase. It is a standalone program that is executed once and takes a compiled library of application logic as input to generate an encrypted output file. The tool can use any cryptographic algorithm. The presented solution uses the AES-256 algorithm.

Loader - the last component is the loader application, which is distributed as an application executable file. It contains an encrypted file embedded in the program as an embedded resource. Once launched, the loader performs the following steps:

- Resource reading – the loader locates the embedded file within its resources and reads it using the Assembly.GetManifestResourceStream method.

- Decryption – the loader decrypts the library data and stores it in memory as a byte array.

- Dynamic Loading – the loader loads the assembly from memory using the Assembly.Load(byte[]) method. In .NET 8, this operation loads the module into an AssemblyLoadContext, similar to loading it from a file, with the key difference being that no file is required on disk [15, 16, 17].

- Calling the Library's Entry Point – With the assembly loaded, the loader uses reflection to locate and invoke the Run method. As a result, the code within the protected library is executed.

After these steps, the loader passes control to the library (protected console application).

Description of operation

To better understand how the custom packer works, it is helpful to analyze a specific use case. For this purpose, we will use a specially prepared library named P00_BubbleSortLib.dll, which contains the BubbleSorter class with the public static void Run(string[] args) method. This method takes a set of numbers as input, sorts them, and then prints out the sorted list to the console.

The entire process of preparing and running the program can be described in a few steps:

Step 1 - The encryption tool is launched, with the path to the P00_BubbleSortLib.dll file provided as input, and the name for its encrypted version specified as P00_BubbleSortLib.enc.

Step 2 - In Visual Studio, the encrypted library P00_BubbleSortLib.enc is added to the loader project as an embedded resource. The appropriate encryption key and initialization vector are also provided to ensure that the library can be correctly decrypted and loaded. Once configured, the loader application can be compiled and published.

Step 3 - When the program is executed via the CMD console with a set of numbers to be sorted (e.g., 7, 5, 1, 3, 6), the loader locates the P00_BubbleSortLib.enc resource and loads its contents into memory.

```
var currentAssembly = Assembly.GetExecutingAssembly();

using var stream = currentAssembly.GetManifestResourceStream("NetPack." + resourceName + ".enc");

if (stream == null)

{

    Console.WriteLine("Resource not found: " + resourceName);

    return;

}

using var ms = new MemoryStream();

stream.CopyTo(ms);

byte[] encryptedBytes = ms.ToArray();
```

Step 4 - With the encrypted data in memory, the loader uses a predefined function static byte[] DecryptAes(byte[] cipherBytes) to decrypt the library and stores it in memory as a byte array. At this point, all IL byte sequences and metadata are present in plaintext form in memory but have not been written to disk.

Step 5 – The loader uses the method Assembly.Load(byte[]) to load the library from the byte array. This call returns an Assembly object representing the now-loaded P00_BubbleSortLib.dll module.

```
var loadedAssembly = Assembly.Load(decryptedDllBytes);

Console.WriteLine($"Library loaded: {loadedAssembly.FullName}");
```

Step 6 – From the loaded library, the loader retrieves the BubbleSorter class type and then its Run method, which it invokes, passing in the arguments supplied to the loader when it was started.

```
var type = loadedAssembly.GetType(resourceName + "." + mainClassName);

if (type == null)

{

    Console.WriteLine("Class not found: " + mainClassName);

    return;

}

var runMethod = type.GetMethod("Run", BindingFlags.Public | BindingFlags.Static);

if (runMethod == null)

{

    Console.WriteLine("Run method not found!");

    return;

}

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

The steps above show how a small C# program can dynamically load protected code at runtime. The key aspect of this process is that the file containing the application logic never appears in plain text on disk. It is decrypted and executed entirely in memory. Figure 2 illustrates the program's execution of this scenario.

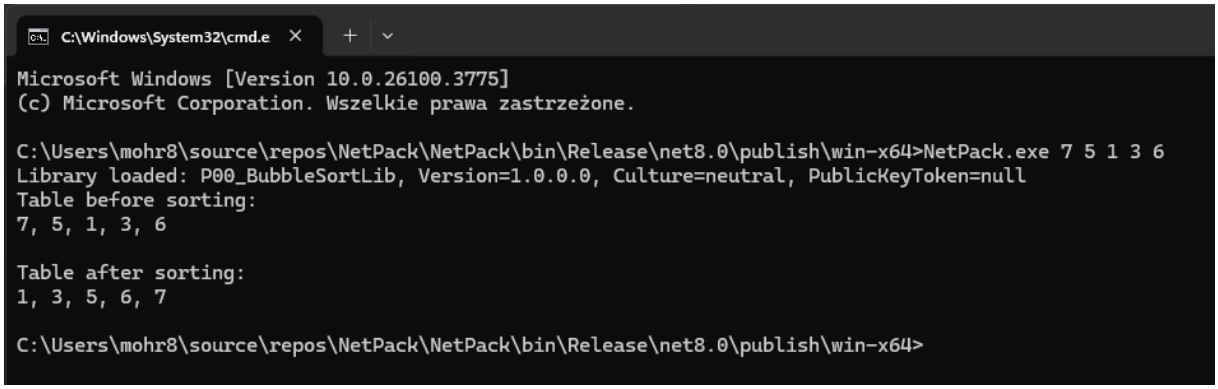

Fig. 2. Screenshot of program execution in the command prompt

Case study

The purpose of the experiment was to compare the execution time of an application performing hundreds of millions of arithmetic operations in its original version and after it was modified using a prepared packer. Both variants of the application were run under identical conditions, and the execution time was measured in the program from the call of the first function performing calculations. To ensure the reliability of the results, each measurement was repeated 10 times, and the average value given in the table below was calculated. The test was conducted using a virtual machine running Windows 11 with 4 GB of RAM and a 2-core processor. The application includes a configuration variable that determines a factor that multiplies the number of operations performed. The results are shown in Table 1.

Table 1. Program execution times

Factor multiplying the number of operations	**The time taken by the original console application to perform calculations**	**Time to perform calculations after applying packer**
1	86 ms	85 ms
10	815 ms	849 ms
25	1677 ms	1685 ms
50	3412 ms	3472 ms
100	6601 ms	6570 ms

The results presented clearly indicate that the use of the packer does not significantly affect the execution time of the called application. This can be seen very well in Figure 3, where the execution times of the two applications overlap. This is due to the fact that once called, the way the program works is no different compared to a classic console application.

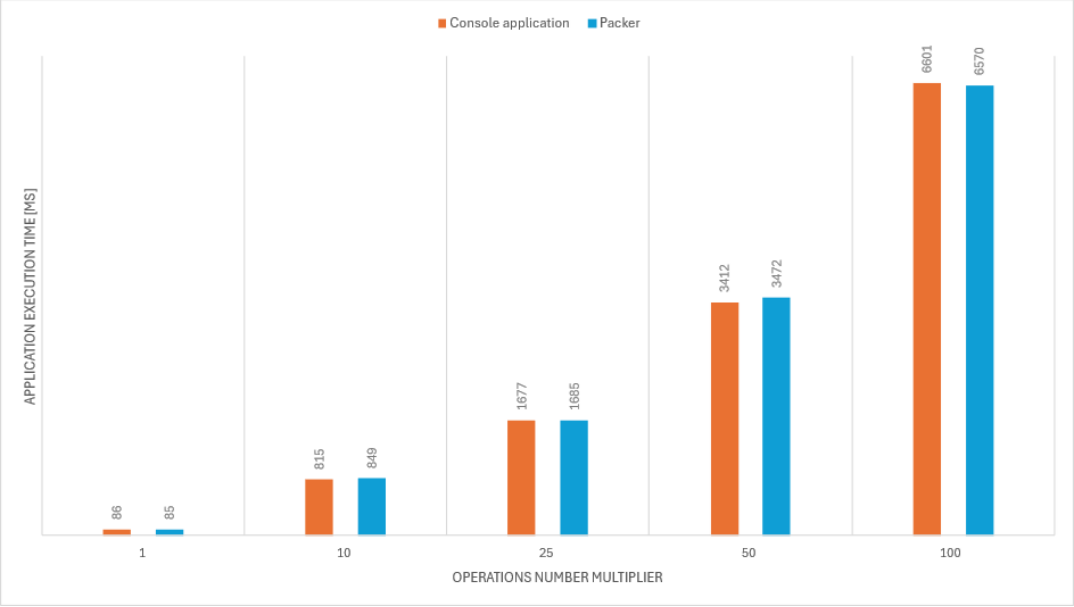

Fig. 3. Program execution times

Advantages and Limitations of the Approach

The proposed obfuscation method using a packer offers several advantages, but also has limitations that should be considered when implementing your own solution.

Advantages:

- Protection against static analysis by encrypting the critical portion of the code.

- Transparency for the end-user, who launches the application through an .exe file that behaves like a normal console program.

- Simple implementation, allowing the packer to be adapted to many projects and extended as needed.

- Czas wykonywania operacji nie zmienia się pomimo zastosowanej ochrony.

Limitations:

- Encryption key embedded in the application - depending on your requirements, you may want to apply additional obfuscation techniques to hide the key inside the program or supply it at runtime based on properties of the environment in which the program is executed.

- Lack of runtime protection - it is advisable to further secure the protected code, e.g., by obfuscating the IL before it is encryptesd.

Conclusions

This article presents a complete packer solution for .NET 8 applications, consisting of a library containing application logic, an encryption program and a loader. The solution effectively demonstrates the concept of code packing, which makes static analysis much more difficult. Its main advantages are simplicity and operational transparency.

The analysis also revealed limitations, mainly that the decryption key is stored in the application, and during execution the decrypted code remains vulnerable to advanced analysis. Nevertheless, the packer presents a

significant obstacle to potential attackers and can protect the software well enough to make an attack economically unviable.

Future work on code protection, in addition to improving the packer itself, should focus on securing the loader's code (especially securing the decryption key) and the IL code of the called libraries through additional obfuscation techniques.

In summary, the described packer shows that with relatively little effort (in the context of commercial projects) and using the mechanisms available in .NET 8, the level of application security can be significantly increased.

Acknowledgements

This work was financed/co-financed by Military University of Technology under research project UGB 531-000023-W500-22.

References

- Introduction to .NET, https://learn.microsoft.com/en-us/dotnet/core/introduction [accessed 7 May 2025].
- .NET glossary, https://learn.microsoft.com/en-us/dotnet/standard/glossary [accessed 7 May 2025].
- Melissa Eckardt, .NET Deobfuscation, https://cyber.wtf/2025/04/07/dotnet-deobfuscation/ [accessed 7 May 2025].
- Collberg C., Nagra J. Surreptitious Software, Addison-Wesley Professional, 2009, p. 12-14.
- Mario M. Introduction to Packers, https://mariom3.github.io/malware-analysis/introduction-to-packers/ [accessed 7 May 2025].
- Understanding Packers, Crypters, and Protectors in Malware, https://101.school/courses/introduction-to-malware-analysis/modules/10-anti-reverse-engineering/units/1-packers-crypters-and-protectors [accessed 7 May 2025].
- Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, Edgar Weippl. Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis?, ACM Computing Surveys (CSUR), Volume 49, Issue 1, Article No.: 4, P. 1 – 37 https://doi.org/10.1145/2886012
- Alessandro Mantovani, Simone Aonzo, Xabier Ugarte-Pedrero, Alessio Merlo, Davide Balzarotti. Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem. NDSS 2020, Network and Distributed System Security Symposium, 23-26 February 2020, San Diego, CA, USA, Feb 2020, San Diego, United States. https://www.ndss-symposium.org/ndss-paper/prevalence-and-impact-of-low-entropy-packing-schemes-in-the-malware-ecosystem/
- Dotfuscator: C# Encryption & Obfuscation for App Security, https://www.preemptive.com/dotfuscator [accessed 24 July 2025].
- Alkhateeb, E., Ghorbani, A. & Habibi Lashkari, A. (2024), A survey on run-time packers and mitigation techniques. Int. J. Inf. Secur. 23, 887–913
- Implementing AES Encryption With C#, https://www.milanjovanovic.tech/blog/implementing-aes-encryption-with-csharp [accessed 27 July 2025].
- Santosh Nagarakatte Milo M. K. Martin Steve Zdancewic, Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety, Computer and Information Science, University of Pennsylvania,
- https://www.cis.upenn.edu/~stevez/papers/NMZ12.pdf [accessed 25 July 2025]
- Reto Achermann, Nora Hossle, Lukas Humbel, Daniel Schwyn, David Cock, Timothy Roscoe; Secure Memory Management on Modern Hardware, Systems Group, Department of Computer Science, ETH Zurich, https://retoachermann.ch/static/papers/achermann-2020-smm.pdf [accessed 25 July 2025].
- What's new in .NET 8., https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-8 [accessed 20 July 2025]
- Assembly.Load Method, https://learn.microsoft.com/en-us/dotnet/api/system.reflection.assembly.load?view=net-8.0 [accessed 7 May 2025].
- About System.Runtime.Loader.AssemblyLoadContext, https://learn.microsoft.com/en-us/dotnet/core/dependency-loading/understanding-assemblyloadcontext [accessed 7 May 2025].
- Managed assembly loading algorithm, https://learn.microsoft.com/en-us/dotnet/core/dependency-loading/loading-managed [accessed 7 May 2025].