FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of master's thesis

Title:	Memory Explorer for .NET
Student:	Bc. Michal Žůrek
Supervisor:	Ing. Josef Kokeš, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2023/2024

Instructions

1) Study the .NET platform from the point of view of a reverse engineer: Tools, CIL (Common Intermediate Language), structure, memory layout, class structure, data type metadata.
2) Focus on the .NET's garbage collector. Research its known behavior and internals.
3) Propose a design of an application that would help a reverse engineer locate and manipulate data in the memory space of a .NET application.
4) Implement a proof-of-concept (POC) of this application.
5) Test your POC against a selection of real-world applications, including obfuscated ones.
6) Discuss your results.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 9 February 2023 in Prague.

Master's thesis

MEMORY EXPLORER FOR .NET

Bc. Michal Žůrek

Faculty of Information Technology
Department of Information Security
Supervisor: Ing. Josef Kokeš, Ph.D.
May 4, 2023

Czech Technical University in Prague
Faculty of Information Technology
© 2023 Bc. Michal Žůrek. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Žůrek Michal. *Memory Explorer for .NET*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

Acknowledgements vii

Declaration viii

Abstract ix

Summary x

Acronyms xi

Introduction 1

1 .NET Overview 3
- 1.1 History of .NET 3
- 1.2 ECMA and ISO Standards 4
- 1.3 Programming Languages 4
- 1.4 Intermidiate Language 5
- 1.5 .NET Implementations 6

2 Tools 9
- 2.1 Visual Studio 9
- 2.2 Watching .NET process heap using Visual Studio 2022 10
- 2.3 Dotnet CLI Diagnostics Utilities 13
- 2.4 ProcDump 15
- 2.5 WinDBG and SOS 15
- 2.6 Listing .NET objects using WinDBG and SOS 17
- 2.7 ILSpy 20
- 2.8 dnSpy and dnSpyEx 21
- 2.9 PerfView 21

3 Garbage Collector 25
- 3.1 Virtual Memory and Cooperation with Operating System 25
- 3.2 Types of Heap 26
- 3.3 GC Variants and Configuration 26
- 3.4 Object Layout and Metadata 27
- 3.5 Allocations 28
- 3.6 Generations in GC 28
- 3.7 Garbage Collection Triggers 30
- 3.8 Garbage Collection Steps 31
- 3.9 Garbage Collection Mark Phase 32
- 3.10 Garbage Collection Plan Phase 33
- 3.11 Garbage Collection Sweep Phase 35
- 3.12 Garbage Collection Relocate and Compact Phase 35
- 3.13 Interactions with Runtime Interactions 36

4 Obfuscation 37
4.1 Motivation . . . 37
4.2 Obfuscation techniques . . . 37
4.2.1 Symbols protection . . . 38
4.2.2 String Protection . . . 38
4.2.3 Codeflow Change . . . 38
4.2.4 Operation and Constant Substitution . . . 38
4.2.5 Virtual Machine . . . 39
4.2.6 Metadata Destroy . . . 39
4.2.7 Anti-Design Time . . . 39
4.3 Available Obfuscators . . . 39
4.3.1 Obfuscar . . . 39
4.3.2 MindLated . . . 40
4.3.3 Z00bfuscator . . . 40
4.3.4 Eazfuscator.NET . . . 40
4.3.5 CSharpObfuscator . . . 40
4.4 Deobfuscators . . . 40
4.5 Impact of Obfuscation on Program Memory Footprint . . . 40
4.6 Obfuscator demonstration . . . 41

5 .NET Memory Explorer Implementation 45
5.1 Methods of searching objects in .NET memory . . . 45
5.1.1 Manual parsing . . . 45
5.1.2 SOS Hosting . . . 47
5.1.3 GC Dumps using Diagnostic API . . . 47
5.2 .NET Memory Explorer Implementation . . . 48
5.2.1 .NET Diagnostic Interface Client . . . 49
5.2.2 .NET Type Information . . . 50
5.2.3 MethodTable . . . 51
5.2.4 EEClass . . . 51

6 Testing 53
6.1 Testing program . . . 53
6.2 Browsing objects . . . 54
6.3 Modifying objects . . . 58
6.4 paint.net . . . 59
6.5 Discovering Classes by Data . . . 61
6.6 Obfsucated application . . . 62

7 Conclusion 65

Contents of the Attached Medium 71

List of Figures

2.1 Visual Studio 2022 10
2.2 Take Snapshot button in Diagnostic Tool in Visual Studio 2022 11
2.3 Data Types and memory usage statistics shown in Visual Studio 2022 12
2.4 Listing of Instances of TestItem class in Visual Studio 2022 13
2.5 Quick Watch of instance content in Visual Studio 2022 14
2.6 Legacy version of WinDBG attached to the testing program 16
2.7 New version of WinDBG attached to the testing program 17
2.8 Objects on managed heaps listed by WinDBG with SOS loaded 18
2.9 Objects on managed heap dumped using `dumobj` command 19
2.10 IL Code generated of GetMagicNumber method shown using ILSpy 20
2.11 C# Decompiled code of GetMagicNumber method shown using ILSpy 21
2.12 dnSpyEx attached to the simple testing program 22
2.13 gcdump object visualisation in PerfView 23

3.1 The layout of objects grouped into generations 29
3.2 The layout of objects after marking objects in Generation 0 29
3.3 The layout of objects and their assignment to generations after sweep 30
3.4 The layout of objects and their assignment to generations after compact 31
3.5 Sequence of GC phases execution 32
3.6 Detail of sequence of GC phases execution 32
3.7 Example topology with one GC root and four objects 33
3.8 Heap conversion to plgus and gaps 34

4.1 Decompiled obfuscated `TestItem` class 42
4.2 Decompiled MindLated obfuscated class `TestItem` with all possible obfuscations . 43
4.3 Decompiled MindLated obfuscated explicit constructor `TestItem` 43
4.4 Decompiled MindLated obfuscated `GetMagicNumber` method 44

5.1 The class hierarchy of .NET runtime structures describing data types 52

6.1 The simple testing program used for testing .NET Memory Explorer 53
6.2 Listing of saved objects in testing program 54
6.3 Heap Dump Details in .NET Memory Explorer 55
6.4 `TestItem` classes listing in .NET Memory Explorer 55
6.5 `TestItem` instances listing in .NET Memory Explorer 56
6.6 Detail of selected `TestItem` instance 56
6.7 Binary content of selected `TestItem` instance 57
6.8 Object referencing selected `TestItem` instance 57
6.9 Objects referenced by the array of `TestItem` 58
6.10 Objects referencing array of `TestItem` 58
6.11 Externaly modfied value and reference in testing program 59
6.12 paint.net and testing scene 60
6.13 Direct2D related Data Types in paint.net 60
6.14 PaintDotNet related Data Types in paint.net 61

6.15 Search string in paint.net memory 61
6.16 `PNG` string search in paint.net memory 62
6.17 LayerProperties object in paint.net memory 62
6.18 LayerProperties object in paint.net memory 63
6.19 Search for data in obfuscated program memory 63
6.20 Search for data in obfuscated program memory 63
6.21 Search for data in obfuscated program memory 64
6.22 Search for object address in obfuscated program memory 64

7.1 .NET Memory Explorer 65

List of code listings

2.1 TestItem class used in testing program 11
2.2 Installing SOS by 16
4.1 `System.TypeLoadException` exception generated by the faulty program 42

I want to thank my supervisor, Ing. Josef Kokeš, Ph.D., for his support and valuable feedback on this topic. I would also thank my family for their support and patience throughout writing this thesis and my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity

In Prague on May 4, 2023 . . . .

Abstract

This thesis aims to design and implement a computer program that allows reverse engineering of the memory of a process running on .NET technology and locating and manipulating objects in the memory of a process. The thesis describes the .NET technology and its generally available implementations. In the thesis, existing tools for listing objects in the .NET process memory are described. The thesis describes behaviour of the Garbage Collector, which is used to manage memory in the .NET environment. The thesis contains a description of obfuscators available for obfuscating .NET programs and an evaluation of their impact on the obfuscated program. The thesis includes a description of possible approaches to analyze .NET process memory. It contains a detailed description of the Event Pipe approach, which was used in the tool developed as a part of this thesis. Finally, the thesis evaluates achieved results when the tool was used to analyze the memory of testing programs, commonly used programs, and obfuscated programs.

Keywords .NET, memory, garbage collector, reverse engineering, process, analysis

Abstrakt

Cílem práce bylo navrhnout a implementovat počítačový program, který umožní reverzně analyzovat paměť procesu využívající technologii .NET a umožní vyhledávat a upravovat objekty v paměti tohoto procesu. Práce obsahuje popis technologie .NET i její běžně dostupné implementace. V práci jsou popsány existující nástroje pro výpis objektů v paměti procesů .NET a je zde popsáno chování Garbage Collectoru, který se v prostředí .NET používá pro správu paměti. Obsahem práce je i popis dostupných obfuskátorů určených pro aplikace v .NET včetně vyhodnocení jejich dopadu na obfuskovaný program. Práce obsahuje možné metody analýzy obsahu paměti procesů v .NET a detailně popisuje metodu pomocí rozhraní Event Pipe, která byla vybrána k implementaci nástroje, který vznikl jako součást této práce. Na závěr práce obsahuje popis dosažených výsledků při použití vyvinutého nástroje při analyzování paměti testovacího program, běžně používaných aplikací i obfuskovaných aplikací.

Klíčová slova .NET, paměť, garbage collector, reverzní inženýrství, proces, analýza

Summary

Motivation

The motivation for doing this thesis was the lack of an analysis tool capable of interpreting objects in the memory of the .NET process for searching in the process memory for known data while not knowing anything about the class structure of an analyzed program. Another motivation was to learn more details about .NET and especially Garbage Collector internals.

Goal

The goal of the thesis is to find methods for accessing metadata about objects in the .NET process and develop a tool that will allow a user to view objects in that process, interpret their metadata as well as their content and allows user to search the memory for data while giving the user information about the class structure that holds such searched data. The thesis aims to find possible approaches for accessing information from .NET process memory, evaluate their advantages and disadvantages, and select a strategy that will be implemented in the tool. This thesis's last goal is to evaluate the final implementation of the tool against testing programs, commonly used and obfuscated programs.

Process

At first, research of existing tools for dumping .NET memory will be done with investigations of their internals and approaches which they use for retrieving and interpreting data from target memory. Next, the behaviour of the Garbage Collector in common .NET runtimes will be investigated with focus on the memory footprint caused by this Garbage Collector. Possible approaches to dump process memory and their interpretation will be analyzed, and their advantages and disadvantages will be evaluated. The result of this analysis will be base for a decision on the implementation used in the tool, which will be created as part of this thesis. Finally, the implemented tool will be evaluated against testing programs, commonly used programs and obfuscated programs.

Outcome

The outcome of the thesis is a .NET Memory Explorer tool with a Graphical User Interface which connects to the .NET process using EventPipe diagnostics API. Too can connect to the process at any time and take a heap dump of the memory containing reliable information about objects in the target .NET process. The tool can search the memory of the process and find objects containing searched term. The tool can evaluate objects in the binary and read internal data type structures from the process to allow parsing object content. Developed tools targets reverse engineering use cases and enable users to rename program identifiers and label objects for a better understanding of the object topology in the program.

Conclusion

The goals of the thesis were successfully realized. The thesis contains an analysis of possible methods for accessing objects in the target process memory. It includes a description of evaluated existing tools which supports providing information about objects in the process memory. As part of the work, internal structures of .NET Core runtime were analyzed, and analysis outputs were used for interpreting object content in the developed tool. Finally, the program was successfully used to analyze the memory of testing programs, commonly used .NET programs and even obfuscated programs.

Acronyms

CLI Common Language Infrastructure
CLS Common Language Specification
GC Garbage Collector
EE Execution Engine
ETW Event Tracing for Windows
GUI Graphical User Interface
IDE Integrated Development Environment
IL Intermediate Language
IPC Inter Process Communication
JIT Just-in-Time
LINQ Language-Integrated Query
LOH Large Object Heap
OS Operating System
SOH Small Object Heap
SOS Son of Strike
WCF Windows Communication Foundation
WPF Windows Presentation Foundation

Introduction

.NET and its ecosystem is one of the most popular toolchains used by millions of developers worldwide to develop, build, run and host applications. Every year more and more new applications use this technology as their base, as well as existing applications are getting larger. This growth is followed not only by the larger impact of this technology but also by a larger number of malicious applications using this technology as well as a larger number of attacks compromising the underlying runtime of the application. .NET is a technology that internally contains a lot of interesting techniques like JIT compiler and garbage collection which are hidden for most developers, but these technologies are here, and we can see their side effects even we most of the time do not interact with them directly. This thesis describes the internals of .NET runtime with a focus on Garbage Collector. It contains a description of the .NET Memory Explorer tool, which was created as part of this thesis. This tool allows looking at the managed heap of the .NET process externally. It will enable its user to search for data and information in memory of this process and highlight the target application's structure and hierarchy of objects. It is designed to support security analysis, and its features targets reverse engineering use cases. The thesis also contains a description of achieved results when the tool was used for analysing memory of small programs developed for testing purposes of this thesis, large real-life applications and also some obfuscated applications.

Since Microsft made the core parts of .NET Open Source in recent years, much information described in this thesis comes from analysis of the source codes directly, but due to the complexity of the topic, much other information was taken from other sources like very valuable book Pro .NET Memory Management book [1] written by Konrad Kokosa. Some information presented here also comes from online resources like Microsoft blogs published directly by .NET developers. In the case of information based on analysis of source codes from the GitHub project, a single point in Git history is used. In the case of the .NET Runtime project (which will be referred most of the time), all information refers to commit with id `340508f5c750d190090be93f0c731b1a8055a354`, which was the latest commit in the .NET Runtime project at the time of beginning writing this thesis. From a practical point of view, it is better to always use the latest sources instead of sources fixed to this commit, but in case of using the latest sources, you need to pay attention and, for example, remember to adjust the line numbers mentioned in this thesis because these numbers can change in time.

The thesis is split into six chapters. The first chapter contains a description of .NET technology in overall. It includes a description of the .NET background, history, and related standards, as well as an explanation of real-life implementations of these standards. The second chapter will show available tools for handling .NET applications. It will describe tools for development, debugging, performance monitoring and tuning, reverse engineering and security analysis. The third chapter will introduce the garbage collector as the critical component for managing memory in the .NET application. It will describe its operation, some of its internal structures and

the overall memory footprint of the garbage collector, which is needed to understand dumping objects from the managed heap of the .NET application. The fifth chapter will move into more security-related topics. It will describe publicly known obfuscators for .NET applications and show how they modify data, which we can see in tools like a tool created as part of this thesis and generally available tools described in the second chapter. The fifth chapter will provide more details about the .NET Memory Explorer tool developed as part of this thesis, including a description of its main classes and its theory of operation with a focus on factors related to dumping and parsing external process memory.

Chapter 1
.NET Overview

The first chapter of this thesis overall describes the history and basics of the .NET technology. In the beginning, there is a briefly described history of .NET, and it's an evolution of both the commonly used implementations .NET Framework and .NET Core, their relations, differences and shared properties. This chapter describes standards related to .NET, C# and other programming languages used in the .NET world. The chapter describes the Immidiate Language bytecode used by .NET runtimes and describes relations between programming languages, Intermediate Lanauge and interpreter. The chapter contains a detailed description of currently available .NET runtime implementations and how differences of runtimes affect tools created as part of this thesis.

1.1 History of .NET

The first beta version of .NET was released at the beginning in 2000 [2]. It was referred as .NET Framework 1.0. .NET Framework 1.0 had several successors referred as 1.1, 2.0, 3.0, 3.5, 4, 4.5, ending with 4.8.1 which is the latest version of .NET Framework as of writing this thesis in March 2023 [3]. Since Windows Vista, many of these versions were included in the Windows operating system by default, and they are strictly backward compatible [4]. .NET Framework was historically proprietary and closed source project, but it was always free for use. Later in 2007 [2] Microsoft opened source codes of .NET Framework 3.5, but they originally opened it only partially, and some important parts of the .NET environment like Language-Integrated Query (LINQ) and Windows Communication Foundation (WCF) remained originally closed source.

In 2016 [5] Microsoft released a new implementation of .NET referred to as .NET Core. Its numbering did not follow versioning of .NET Framework and started with .NET Core 1.0 at the same time when .NET Framework 4.6.1 was available. .NET Core reimplemented many parts of .NET and is not fully compatible with .NET Framework. In opposition to .NET Framework, .NET Core was always open source, and since the begining, it supported building and running on Linux. It is the first time since .NET code was able to run on Linux with a framework based on Microsoft implementation. Natural support for Linux introduced several limitations to the framework, and some concepts used in the original .NET Framework were removed or not implemented in .NET Core. .NET Core was referred as .NET Core until version 3.1, which was the latest version under this name. Microsoft decided to skip versions with major number 4 to prevent confusion with the original .NET Framework, which was versioned by major number 4 at that time. Instead version 3.1 was followed by 5.0 and the product was renamed from .NET Core to just .NET (without the Core and Framework suffix in the name). The latest .NET version at the time of writing this thesis is version .NET 7.0, and at the time of writing this text we can

see the first occurrence of preview version 8.0.

.NET is based on several technologies. It is not only the runtime, but .NET also define standard libraries. Both .NET Framework and .NET Core define their libraries. As mentioned above, there are some differences in APIs of the .NET Framework and .NET Core with minor differences. For this reason, Microsoft introduced .NET Standard [6] as an API subset supported by both .NET Framework and .NET Core and some other implementations of .NET like Universal Windows Platform (UWP), Xamarin and Mono. This standard allowed us to write code running on all these .NET runtimes.

1.2 ECMA and ISO Standards

While most of the development was historically done by Microsoft, some original development was also done by Intel. The effort from this initial development was later standardised using ECMA standards. ECMA standards covered Common Language Infrastructure (CLI) and C# programming language. IOS/IEC standards later adopted both [7]. There are two important ECMA (and related ISO) standards. Both standards are currently 6th editions.

1. **ECMA-334** [8]: This standard describes C# programming language. This standard describes C# constructs, syntax, fundamental data types, naming rules, operators and their priorities, description of automatically generated exceptions. Generally, it contains specifications related to C#, but it says nothing about other languages supported by the .NET ecosystem and nothing about .NET libraries, API, runtime and implementation details in general. It also describes eight undefined behaviour situations when writing code for .NET in C#. [8]

2. **ECMA-335** [9]: This language-independent standard contains details about Common Language Infrastructure CLI, a fundamental part of the .NET. It contains a description of Common Language Specification (CLS), data types metadata definitions of classes, methods, types, requirements on execution engine (runtime), and details about generics support. It also contains a detailed description of Intermediate Language (IL) instructions which I will describe later. It also contains a description of required libraries implemented and provided by implementation (which is not described in this document directly. These APIs are described in the attached 161593 lines long XML file). This standard also specifies the format of debugging symbol file (which is in standard referred as Portable CILDB File, usually known as PDB). Finally, it contains some requirements for implementing IL assemblers (probably for standardising the syntax of IL assembly text files). [9]

1.3 Programming Languages

Currently, the primary programming language used in .NET is C#, but .NET is not limited to this language. In general, .NET works in a way that source code is compiled into ECMA-335 standardised binary code referred as Intermidiate Language (IL). ECMA-335 has no requirement for a programming language, and C# (described in ECMA-334) is only one option. Except for C# there are two additional languages officially supported by Microsoft:

1. Visual Basic .NET

2. F#

Both languages have their own specification (independent of any ECMA standard). Visual Basic Specification is available to download at the Microsoft Docs site in DOCX format [10]. F# Specification is available to download in PDF format at F# website [11].

Except for these three officially supported languages, there are several abandoned languages supported by Microsoft in history (for example, Python via IronPython project), and few community-managed projects exist. For example, the Open Source compiler PeachPie [12] allows writing .NET applications in PHP. Wikipedia [13] and [14] mention 35 languages with existing compilers for making programs written in these languages running in the .NET ecosystem. These 39 languages range from general purpose languages like C#, functional programming languages like F#, educational programming language Small Basic, domain-specific languages Axum for highly parallel applications, metaprogramming languages like Script.NET, business-oriented languages like Data Business Language (DBL) and programming language optimised for graph transformations GrGen.NET. Note that many of these languages are deprecated and non-maintained. Programs written in all these languages can be compiled to IL and later processes by .NET implementations.

1.4 Intermidiate Language

The intermediate language is the assembly language used by .NET applications. As mentioned above, programs are compiled from the source programming language (for example, C#) to this IL language, which is later executed by runtime. While it is assembly, it supports more advanced instructions, which are not known in processor assembly languages like x86-64 or ARM64 assembly. IL contains, for example, instructions for handling objects. Since the .NET is object-oriented and object design is highly integrated into the .NET, IL is naturally object-aware. There are instructions like, for example, instruction `initobj` (initialise object at address), `ldfld` (load field of an object), `stsfld` (store to a static field of class) and many other. At runtime interpreter executes these instructions (in fact, it is not an interpreter, but it is native code compiled by JIT instead), which of course took, executes more native (for example, x86-64) instructions to execute (or less in case of JIT optimisations) single object-oriented IL instruction. Except object-oriented instructions, there are also some additional instructions usually not present in processor instruction sets, like instructions for passing arguments when calling a function (or method). For this purpose, IL has `starg` (store value in argument slot) instruction, `ldarg` (load argument to stack), `ldarda` (load argument address to stack), and `arglist` (get argument list) instructions. Finally, there are standard instructions similar to processor instructions, like instruction `add`, which adds two numbers. In the case of IL one instruction supports multiple width operands and single `add` instruction supports 32 and 64-bit numbers. In opposition handling unsigned numbers is done using different (`add.un`) instruction. IL is stack-based. For example, mentioned `add` instruction works in a way that two values took (pop) from the stack and the output value push to the stack. But the stack is not the only memory. As mentioned above, there is, for example, support for argument-passing instructions. If the user wants to sum two arguments of the method, he needs to load the argument to stack using `ldarg` instruction, then call `add` instruction which pops the two values from the stack and push one output value to the stack. Then the user can process output with additional instructions that will take value from the stack. For example, you can store it in a local variable (IL supports local variables similarly to arguments) using `stloc` instruction. Finally, IL has support for exceptions. Exceptions are represented by objects (which are also supported by IL as mentioned above). IL has instruction `throw` for throwing an exception. This instruction consumes an exception object from the stack. IL can throw exceptions internally, which is the case, for example, of the `div` instruction used for dividing numbers. One of these expressions is generated in the case of integer division by zero (note that this instruction support float division and, in case of float division by zero, returns NaN float value instead of generating an exception).

While there is significant flexibility in IL, there are some restrictions defined by Common Language Specification (CLS). For example, it is a case of throwing exceptions, as mentioned above. Throwing is done using `throw` instruction discussed above. In terms of IL, this instruction pops one object from the stack. IL does not restrict the object data type, but CLS

specifies such a requirement. It can be any instance of any class (or null, which causes throwing `System.NullReferenceException` internally). But CLS, a specification for making codes written in different languages interoperable, specifies that object passed to throw an exception must be an instance of `System.Exception` class, or it must be an instance of class inheriting from `System.Exception` class.

1.5 .NET Implementations

As briefly discussed in the section History of .NET, there are multiple implementations of .NET available, like mentioned .NET Framework and .NET Core. This implementation brings its own (slightly different) runtime and libraries [1]. Except these, there are several other alternative implementations of the .NET. The most popular alternative implementation is Mono which was mostly used on non-Windows platforms in history. Currently, official implementation .NET Core also allows running on Linux. However, Mono still live as part of Xamarin mobile framework (which enables the development to .NET apps for Android and iOS based mobile phones) and in case of applications running on Linux which rely on API which is not implemented in .NET Core (for example, GUI frameworks Windows Forms and WPF are not). Mono is one of the four .NET implementations officially supported by Microsoft [15]. The fourth implementation officially supported by Microsoft is the Universal Windows Platform (UWP).

Many .NET implementations existed in history but were deprecated and abandoned. They are not listed in Microsoft documentation, but they are listed, for example, in [1]. One of them is, for instance, SilverLight, which allowed running .NET apps in web browsers and .NET Micro Framework, which allowed running .NET applications on microcontrollers.

Implementation defines many things, and most implementation details are not specified in the specification and are implementation-defined. For example, Just-in-Time (JIT) compilation is unnecessary, and anyone can implement its own non-JIT .NET IL interpreter (runtime), which would fully comply with ECMA-335. The implementation also specifies essential parts for the tool created as part of this thesis. It is Memory Management and Garbage Collector (GC). GC is not a mandatory part of the .NET runtime. As you can implement an interpreter without JIT, you can implement runtime with a different allocator without using the GC concept. Instead, you can, for example, implement reference counting [1] based memory management, and instead of cleaning memory by the garbage collector, you can clean objects whenever the counter (which will be part of every object) reaches zero references. It is possible, but this thesis's author is unaware of any .NET implementation using anything different from GC for managing memory. While all current (and especially active and supported) implementations use GC, each uses slightly different implementations of GC. It is especially significant in the case of Mono which uses an entirely different GC compared to mainline .NET Framework and .NET Core implementations. From a practical point of view, the implementation of Garbage Collector significantly affects the program's memory footprint. This affects the main topic of this thesis - a tool for analysing external memory. Naturally, this tool must be implementation specific unless there is an interface for exposing information about memory (in fact, there is some interface, but unluckily it is not standardised outside the mainline .NET world, and it is also implementation specific). For example, assume a dummy tool that looks at the memory of the target process. This tool parses some internal structures of GC, and finally, it finds all objects managed by target memory management. But suppose this tool is executed against a process running different implementations of memory management (different implementations of GC). In that case, it will probably find nothing or meaningless information because the internal metadata of other implementations is entirely different. Alternatively, the tool can support the detection of target runtime and implement multiple strategies of parsing internal GC structures for supporting various GC implementations, but this kind of tool *never* can support *all* implementations because you always can create a new implementation that this tool is unable to handle. This thesis and tool were created to target mainline .NET Core running on the Windows platform. Based on the

information provided in this section, it will naturally be unable to detect any object in memory of a process that runs .NET using Mono or other non-mainline Microsoft .NET runtime.

Chapter 2
Tools

This chapter describes tools for developing, processing, debugging and monitoring .NET processes. The chapter will focus on tool features related to this thesis, and these features are highlighted in more detail. The chapter primarily focuses on features supporting dumping memory and watching its data. For more general information about these tools, I recommend visiting their official websites, GitHub projects and their official documentation.

The following tools will be described in this chapter. Some tools ca be categorised into multiple categories. They are included in the most related ones.

1. Development Tools:
   a. Visual Studio
2. Debugging Tools:
   a. WinDBG
   b. dotnet-cli diagnostics utilities
   c. ProcDump
3. Reversing Tools:
   a. ILSpy
   b. dnSpy
4. Monitoring Tools:
   a. PerfView

2.1 Visual Studio

The first tool mentioned in this section is the most popular tool in the .NET. It is Microsoft Visual Studio. It is a tool mainly (but not only) used for the development of .NET applications. It is Integrated Development Environment (IDE). Visual Studio was also used to develop a tool created as part of this thesis. Most of the features of this IDE target development needs. Visual Studio has tools for debugging and testing as well. Visual Studio can be used for the development of both .NET and native (for example, C++) applications or applications combining both paths and interacting, for example, using Platform Invoke (P/Invoke) or applications running .NET app using CLR Hosting (CLR Hosting is used by users who want to integrate .NET to their existing

non-.NET applications). Visual Studio has an integrated debugger. It is mostly designed for debugging applications with available source codes, but it can be partially used for debugging applications without sources available (for example, applications created by someone else). This is a case of making a security analysis of programs in the .NET. In this case, Visual Studio features are minimal, and dnSpy (which is described later in this chapter) tool can provide more information for the user in this case. In opposition, some tools, like advanced options for breaking the debugger on exceptions and the IntelliTrace tool, can still be handy even without having source codes available. The following screenshot in Figure 2.1 shows Visual Studio 2022.

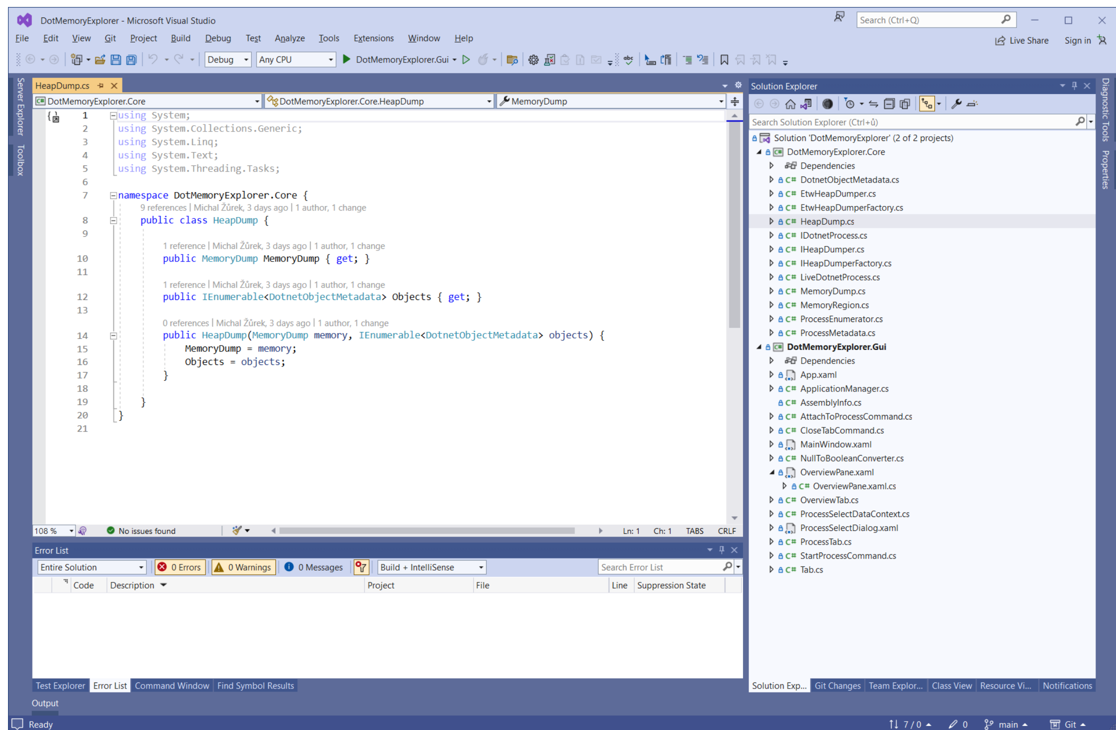


Figure 2.1 Visual Studio 2022

2.2 Watching .NET process heap using Visual Studio 2022

Visual Studio has a tool which allows users to see objects on the heap, similar to .NET Memory Explorer tool, which was created as part of this thesis. This tool can work on a memory dump of non-live processes or a snapshot of the memory of the live process. Making memory snapshots of live processes is possible in `Diagnostics Windows` (in case of missing this window, the window can be enabled in the menu using `Debug` > `Windows` > `Show Diagnostic Tools`). The option for making a snapshot is highlighted on the screenshot in Figure 2.2.

After creating a snapshot, the new snapshot appears in the Memory Usage tab in the Diagnostic Tools window. The objects (Diff) column contains a clickable link that opens details about memory snapshots after clicking. In the newly opened window is recommended to unmark `Collapse small object types` checkbox; otherwise, some objects will be hidden (this checkbox is useful for memory usage and performance troubleshooting use cases). In the example shown in this section, I will use DebugeeWpf, which is a simple program created as part of this thesis for testing purposes. It is a simple program with GUI which allows users to add items to the internal store and later print them. Items consist of one string, number and link to another item.

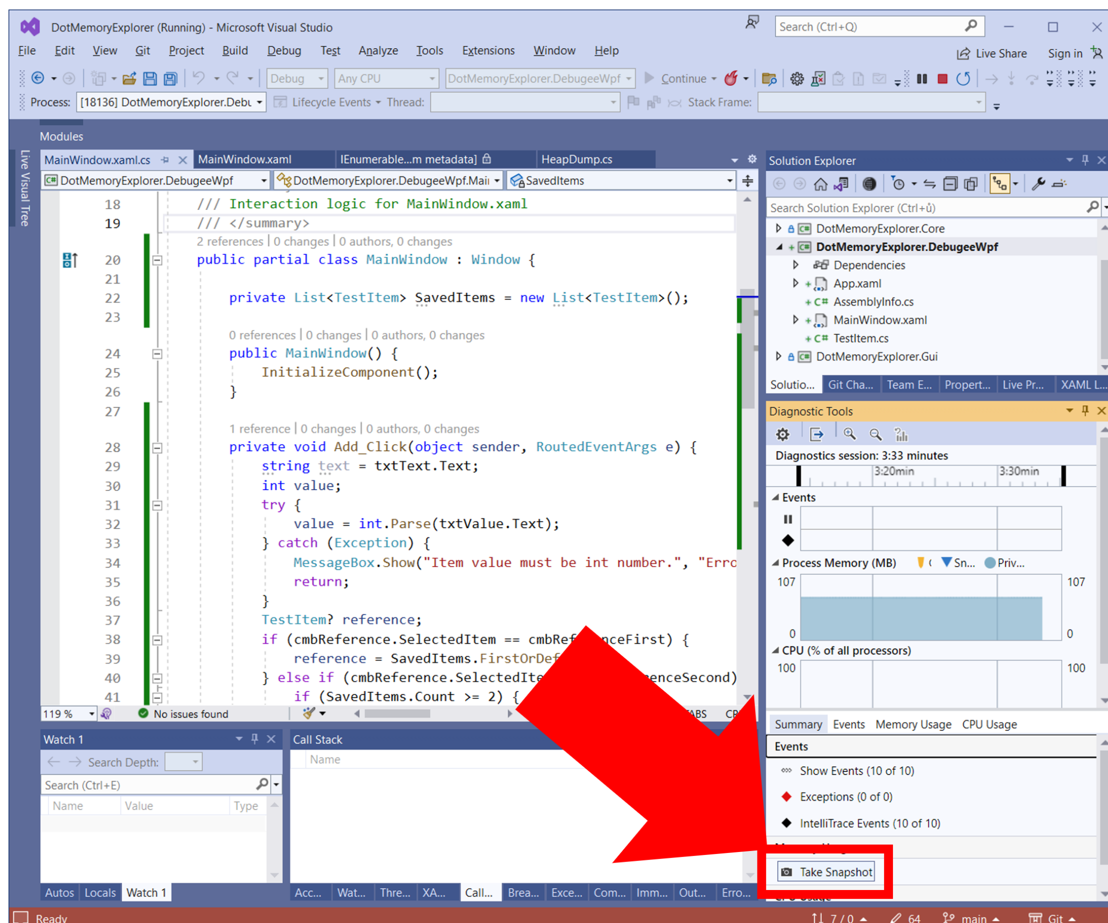


Figure 2.2 Take Snapshot button in Diagnostic Tool in Visual Studio 2022

This program is written in C#. Its primary purpose is to test that their internal objects are detectable by tools like the .NET Memory Explorer, created as part of this thesis or currently in Visual Studio. Since now Visual Studio is used, we naturally need to know about the analysed program. Items of this simple program are represented internally using `TestItem` class. It is a very simple class with one method and three fields (not properties) for storing data inside the class. Its implementation is shown in the following code listing 2.1.

Code listing 2.1 TestItem class used in testing program

```
public class TestItem
{
    public string? Name;
    public int Value;
    public TestItem? Reference;

    public int GetMagicNumber() {
        if (Name == null) {
            return Value + 42;
        } else {
            return Name.Length + Value;
        }
    }
```

```
}
```

After adding some items in the debuggee GUI user can make a new snapshot of memory in Visual Studio, open it and see objects of the map, including instances as shown on the screenshot in figure 2.3.

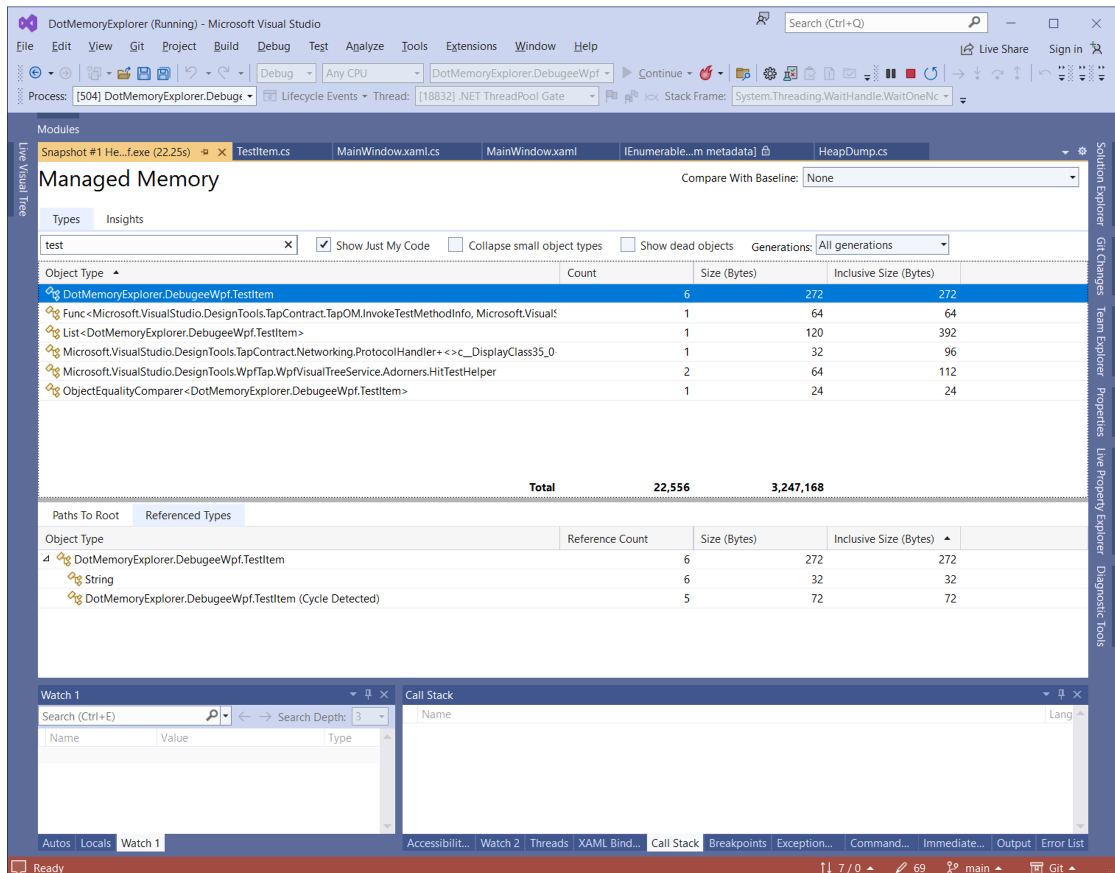


Figure 2.3 Data Types and memory usage statistics shown in Visual Studio 2022

The tool can show Paths to Root which contains a list of paths to some GC Roots. Term GC Root will be described in more detail later in chapter Garbage Collector, but in simplicity, it is one of few places that contain a reference to objects on the heap. These roots contains a reference to objects on the heap, and these objects can contain references to other objects. This tool shows the path from GC Root to objects of selected data type. There could possibly be more roots containing paths to objects, and also, one path can be used by more objects which are in the case of this program.

The tool also shows referenced types and a count of objects referenced by objects of selected data type.

The memory analysis tool offers the option to view all instances of the object on the heap, as shown in the screenshot in figure 2.4.

Users can view instances by double-clicking the data type in the main view of the heap snapshot window. The tool will list all objects found in the heap and their addresses. To show the object's content, the user can right-click the object and select `Quick Watch` to view the selected object content. The content of one instance is shown in the following screenshot in figure 2.5.

While this tool offers the option to view objects on a .NET heap, it is not mainly designed for this task. Its main job is to help developers analyse memory and performance-related issues like

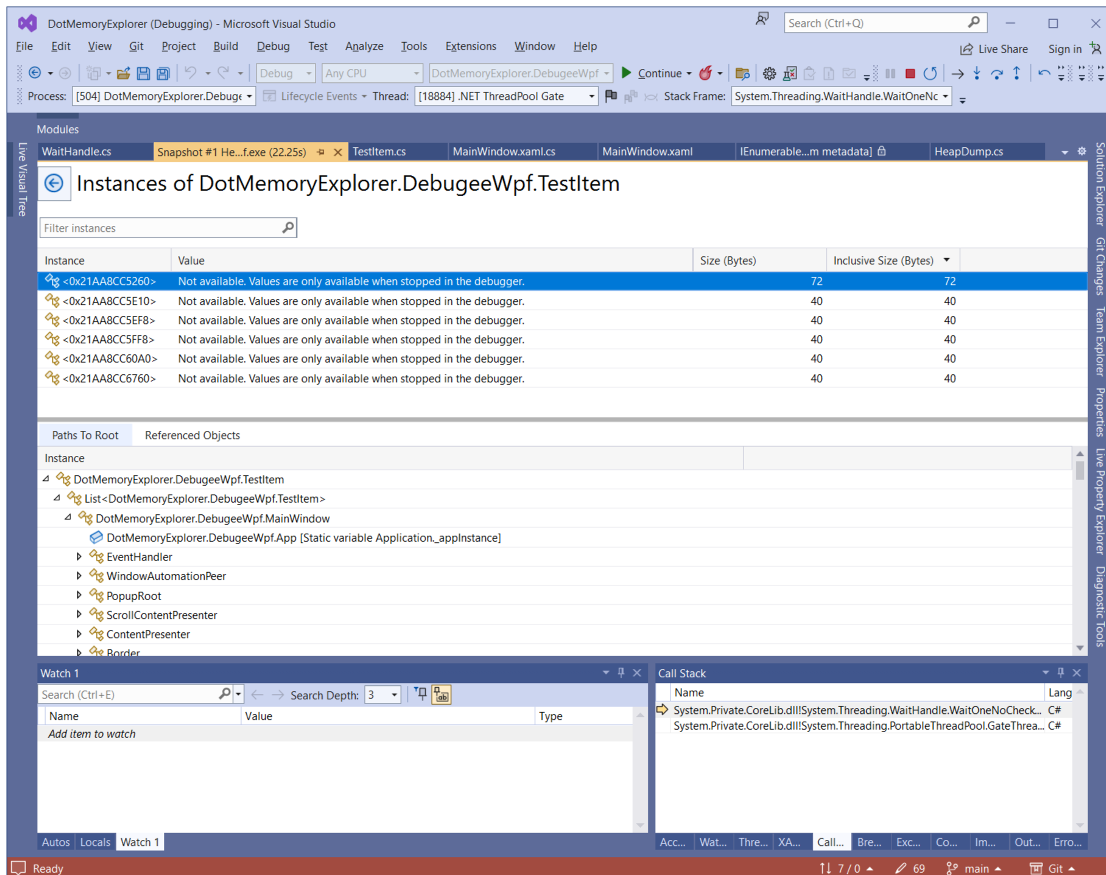


Figure 2.4 Listing of Instances of TestItem class in Visual Studio 2022

memory leaks. For this reason, the window has features like insights showing duplicate strings in memory, which can be beneficial in development time. Still, it is not very helpful in the case of security-related analysis and reverse engineering. One of the main features of this tool is that it can compare two snapshots and view statistics of allocated objects and compare them between snapshots. The tool does not allow almost any memory operation known from other native debuggers like x64dbg; for example, it has no option to search objects by their content. The capabilities of this tool are very limited for reverse engineering and security analysis. In the case of obfuscated applications, it also does not allow any detailed inspection since most labels shown in this window are useless, and there is no way to at least temporarily modify and track them.

2.3 Dotnet CLI Diagnostics Utilities

The second tool described in this chapter is not the only tool, but instead, it is a set of tools. Especially two of them are interesting in the context of this thesis. All of the tools mentioned in this section are open-source. Their source codes are not in the main dotnet/runtime [16] repository but instead, they are available as part of dotnet/diagnostics repository [17]. Sources of each tool are available in `src/Tools/\<tool name\>` folders. This set of tools works on diagnostics data output of .NET Core runtime implementation. Each tool is designed for different diagnostics tasks. Each tool internally connects to the diagnostics API of the .NET program.

Set contains following tools [18]:

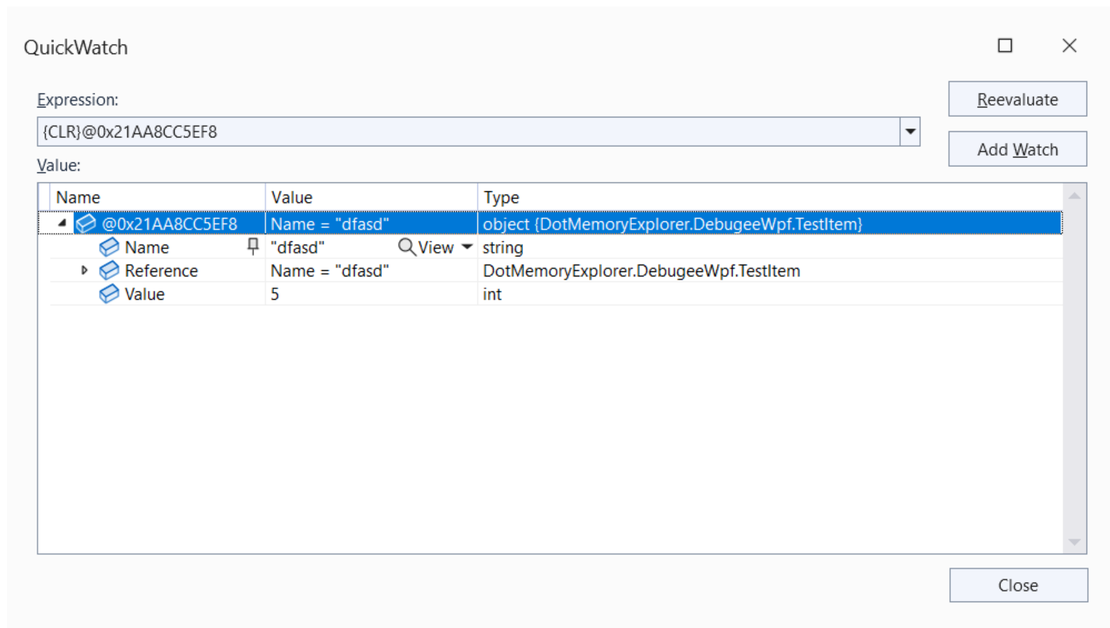


Figure 2.5 Quick Watch of instance content in Visual Studio 2022

1. **dotnet-counters**: This tool allows the user to export selected counters from the internal counters of .NET runtime. These counters are updated internally, and the user selects the frequency of readings. Dotnet runtime implements many counters ranging from system-level counters like CPU and Memory utilisation to .NET-specific counters like the number of garbage collections executed within the defined time slot. These tools connect to the diagnostic channel of .NET Core runtime and export retrieved information to standard output in a human-readable format.

2. **dotnet-dump**: This tool allows the user to make a memory dump. The dump from this utility can be later loaded in WinDBG or Visual Studio, and in the case of the Visual Studio approach and tool described in section Watching .NET process heap using Visual Studio 2022 can be applied to the dump created by this tool.

3. **dotnet-gcdump**: dotnet-gcdump creates a memory dump of the process but does it entirely differently, and the output dump has a different format. Dump generated using this tool contains a lot of metadata generated by Garbage Colletor at collection time.

4. **dotnet-monitor**: This tool allows attaching to the process and collecting data from diagnostics channels on demand or based on defined rules. It can export data that most of the tools mentioned in this section process. It is designed using in production environment and is designed for adding low overhead to monitored processes.

5. **dotnet-trace**: dotnet-trace command is used for printing events sent from .NET runtime. Users can select which part of the .NET runtime they want to monitor. He can limit traces, for example, only to events from the JIT engine of .NET Runtime. Traces generated by the runtime and processed by this tool usually contains information that some action occurred. This is usually used for debugging performance issues.

6. **dotnet-stack**: dotnet-stack tool prints stack trace for every running thread in the specified process.

7. **dotnet-symbol**: dotnet-symbol is a utility which does not connect to any process. It is used for downloading debugging symbols for .NET runtime and other system DLLs, which are later used by other tools (not limited to tools listed in this section). Symbols allow them to translate addresses to human-readable names.

8. **dotnet-sos**: Similarly to dotnet-symbol, this tool does not connect to any process; instead, it just downloads the SOS extension. The SOS debugger extension will be described later in this thesis.

9. **dotnet-dsrouter**: The last tool is used for routing diagnostics events from sandboxed environments. Tools mentioned in this listing mostly directly connect to the process, but on mobile and some other platforms, this is impossible due to sandboxing. dsrouter receives events from sandboxed environments using a TCP connection and reexports diagnostics data as it they were generated directly by the .NET app running in this process.

In terms of this thesis, the essential tools are `dotnet-dump` and `dotnet-gcdump`. Mainly it is a case of `gcdump`, which processes a lot of metadata exported by Garbage Collector, which holds information about objects in the process memory. As mentioned above tool is fully Open Source. Its implementation is not very long, mostly because the tool only processes events from the diagnostics stream and generates output based on this data. Tool work in a way that connects to the event stream of .NET runtime with flags which enable transmission of several GC-related events as well as a flag which, in a remote process, immediately start (full) GC collection. In this collection, GC emits information about processed memory. Because it is a full collection, it processes the whole heap and reports information and metadata about all objects on the managed heap. This tool processes this information and saves them to the output `.gcdump` file. All the concepts used in this utility will be described in more detail in chapter .NET Memory Explorer Implementation because the implementation of the .NET Memory Explorer tool was created as part of this thesis using the same concept for getting information about objects living in the memory of target .NET process.

2.4 ProcDump

Another tool shown in this chapter is ProcDump [19]. This tool from the Sysinternals tool set is a utility which is not dependent on .NET and instead makes a memory dump of complete (or partial if selected using command line arguments) process memory. Its output is similar to the output of `dotnet-dump` command, but it works differently since it is not dependent on the .NET runtime running inside the target process. It also has more options describing which memory sections will be included in the dump. Dumps generated using this command can be processed using WinDBG or Visual Studio like it was shown in section Watching .NET process heap using Visual Studio 2022.

2.5 WinDBG and SOS

WinDBG is a debugger historically used for debugging native applications. WinDBG does not have support for analysing .NET applications directly. Still, there is a plugin named Son of Strike (SOS) [20], which can be loaded into WinDBG. Then this plugin adds commands related to dotnet runtime, for example, for allowing viewing runtime state, listing managed heaps, managed threads and many others.

There are two versions of WinDBG. They differ mainly in GUI. The old (but still used and supported) has legacy GUI based on menus, windows and command line. The new version has ribbon control, and windows were replaced by tabs similar to tabs used in Visual Studio. The latest version has some new features, like time travel debugging, but generally, both versions

support similar features. While the especially new version allows basic usage using buttons in the control panel, most control in WinDBG is done using the command line. The following screenshot in figure 2.6 shows an older version of WinDBG. This thesis will use a new version of WinDBG in all cases.

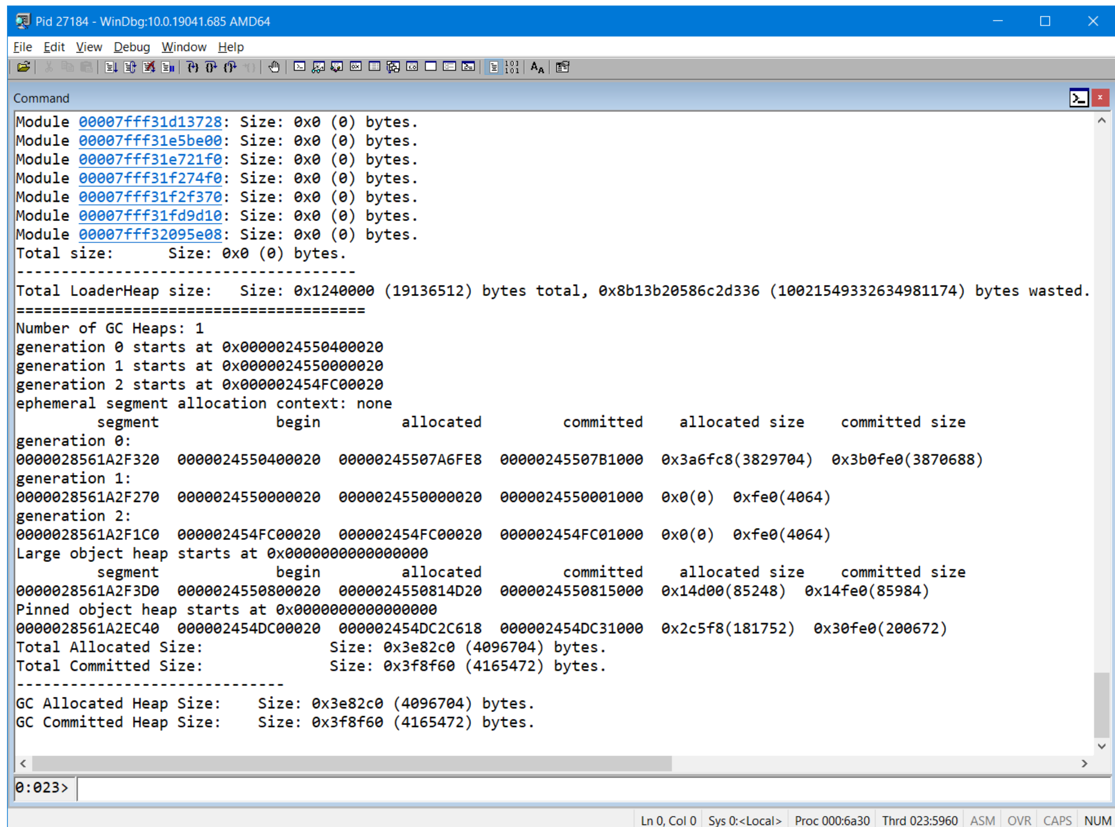


Figure 2.6 Legacy version of WinDBG attached to the testing program

For adding support for commands related to the .NET, we need to install and load the SOS plugin. As mentioned in section .NET Implementations there are multiple implementations of .NET runtime and plugins analysing runtime, which have to be implementation-aware. For this reason, several implementations of SOS exist. All of them target different .NET runtimes. SOS can be loaded using .load \<path to SOS.dll\> command or using .load sos <runtime type> command, which will try to find DLL in proper folders on disk-based. <runtime type> parameter of .loadby in this case have to be one of following value: mscorsvr, mscorwks, clr, and coreclr [21]. The first two options are used when attaching a debugger to process running a very old version of the .NET Framework. The CLR option is useful for programs running .NET Framework 4 and above, and finally, CoreCLR is useful when running processes with UWP, SilverLight or .NET Core runtime. In the case of the .NET Core situation is easier because there is dotnet-sos command mentioned in the previous section Dotnet CLI Diagnostics Utilities. This command installs the latest version of SOS usable with .NET Core runtime. Its outputs hint how to load the newly installed plugin in WinDBG using .load command with a fixed path to the newly installed SOS.dll. The output of this command is highlighted on the following code listing 2.2.

Code listing 2.2 Installing SOS by

```
C:\\Users\\michal\>dotnet-sos install
Installing SOS to C:\\Users\\michal\.dotnet\\sos
```

```
Installing over existing installation...
Creating installation directory...
Copying files from C:\\Users\\michal\\.dotnet\\tools\\.store\\dotnet-sos\\
   6.0.351802\\dotnet-sos\\6.0.351802\\tools\\netcoreapp3.1\\any\\win-x64
Copying files from C:\\Users\\michal\\.dotnet\\tools\\.store\\dotnet-sos\\
   6.0.351802\\dotnet-sos\\6.0.351802\\tools\\netcoreapp3.1\\any\\lib
Execute '.load C:\\Users\\michal\\.dotnet\\sos\\sos.dll' to load SOS in your
   Windows debugger.
Cleaning up...
SOS install succeeded
```

After installation and loading the SOS, new commands can be used in the WinDBG command line. The following screenshot on figure 2.7 shows a new version of WinDBG after loading and executing `!EEHeap` command, which prints basic information about managed heap controlled by Garbage Collector. Most information from this listing will be in more detail explained in the next chapter Garbage Collector.

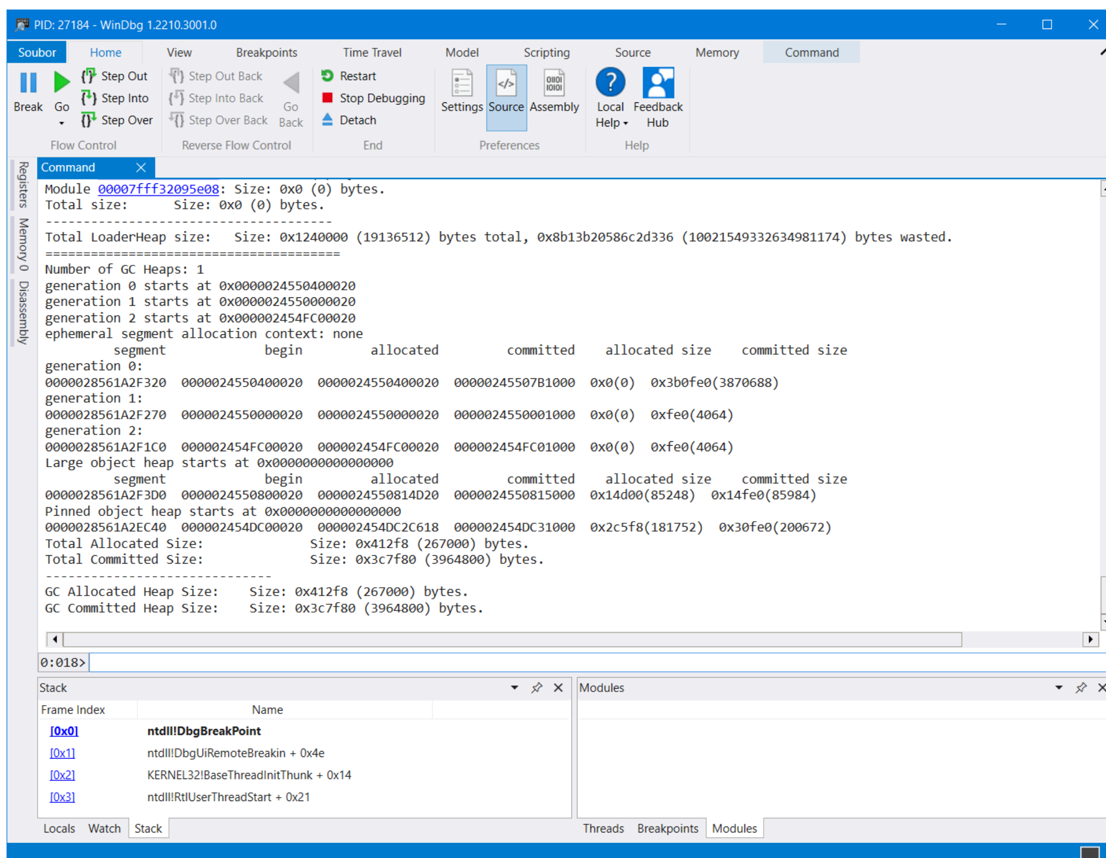


Figure 2.7 New version of WinDBG attached to the testing program

2.6 Listing .NET objects using WinDBG and SOS

WinDBG (with SOS extension) is another tool that shows objects running in process memory. In comparison with `dotnet-dump` and `dotnet-gcdump` mentioned in section Dotnet CLI Diagnostics Utilities it does it differently. WinDBG, compared with mentioned tools, does not rely on

diagnostics API and runtime. Instead, it can analyse the memory of the fully suspended process. WinDBG works in a way that parses internal structures directly from the process's memory. As mentioned, WinDBG can scan process memory and output information about objects in almost any process state. Still, one exception is mentioned in Scenario 9-1 in [1]. Even WinDBG does not work when the process is paused when the Plan phase of Garbage Collection is in progress. In the Plan phase, GC uses object memory to create internal structures (binary trees) on objects for their easier processing in the following (Compact/Sweep) phase. Some objects may be invalid in this state, and analysis tools like WinDBG with SOS can't rely on them. For this reason, SOS detect barriers indicating that GC is in that critical phase, and when the user wants to analyse anything about the heap using this tool at this time, then it fails with an error instead. But it is very rare to hit GC in this phase since it usually takes only a fraction of the garbage collection execution time and the program running overall.

For listing all objects on the .NET process heap, there is `!dumpheap` command. When executed in list pointers to all objects (which is not the beginning of the object because there is some object-related metadata before this pointer as will be described in next section Garbage Collector). It prints a pointer to MethodTable, containing metadata about the object type and size. Example execution of `!dumpheap` is shown on screenshot in figure 2.8.

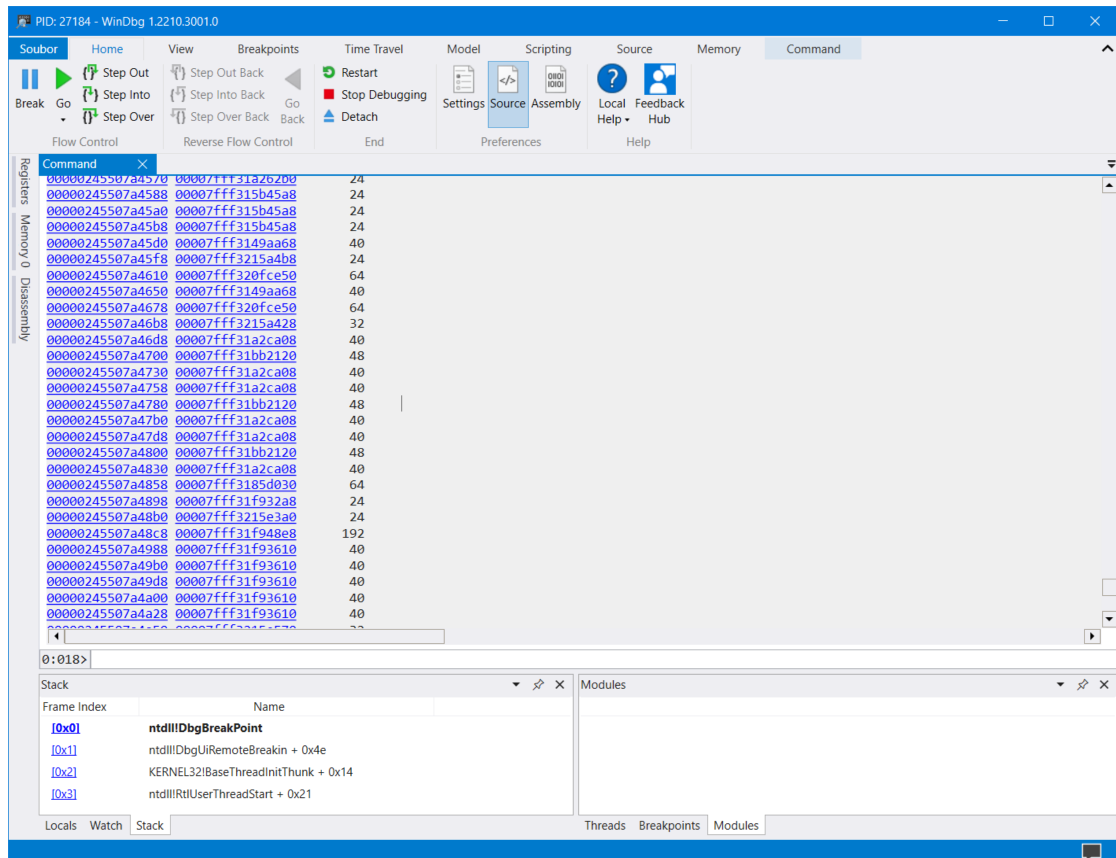


Figure 2.8 Objects on managed heaps listed by WinDBG with SOS loaded

After finding the proper object, there is an option to dump its content using `dumobj` command as shown in the following screenshot on figure 2.9.

Description of development and debugging tools with highlights of techniques for dumping and the analysing process is complete now. In the next session, the thesis will describe tools used for reverse engineering. Two tools will be described: ILSpy and dnSpy.

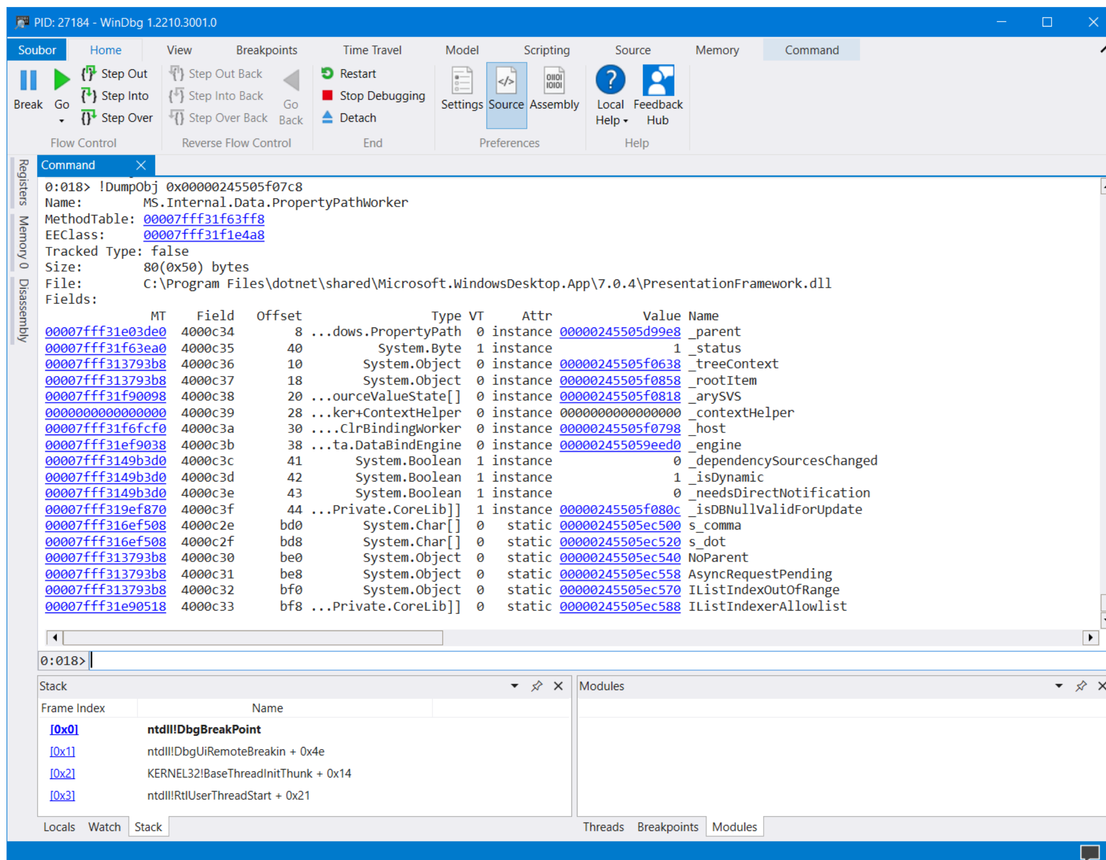


Figure 2.9 Objects on managed heap dumped using `dumobj` command

2.7 ILSpy

The first discussed tool is ILSpy, a disassembler and decompiler for .NET binaries. As mentioned in the previous chapter .NET Overview .NET programs are not compiled to instructions directly processed by the processor, but instead, binaries hold IL instructions. ILSpy is a good tool for viewing this IL code inside of generated binary. ILSpy is an Open Source tool, and its source codes are available at Github [22]. The following screenshot on figure 2.10 shows disassembled IL code of GetMagicNumber method from a simple debugge program created as part of thises with source code shown in listing 2.1. Even if the disassembly is pure listing without any possibility of a graphical representation of code and jumps, it still basically splits basic blocks of the code using empty lines.

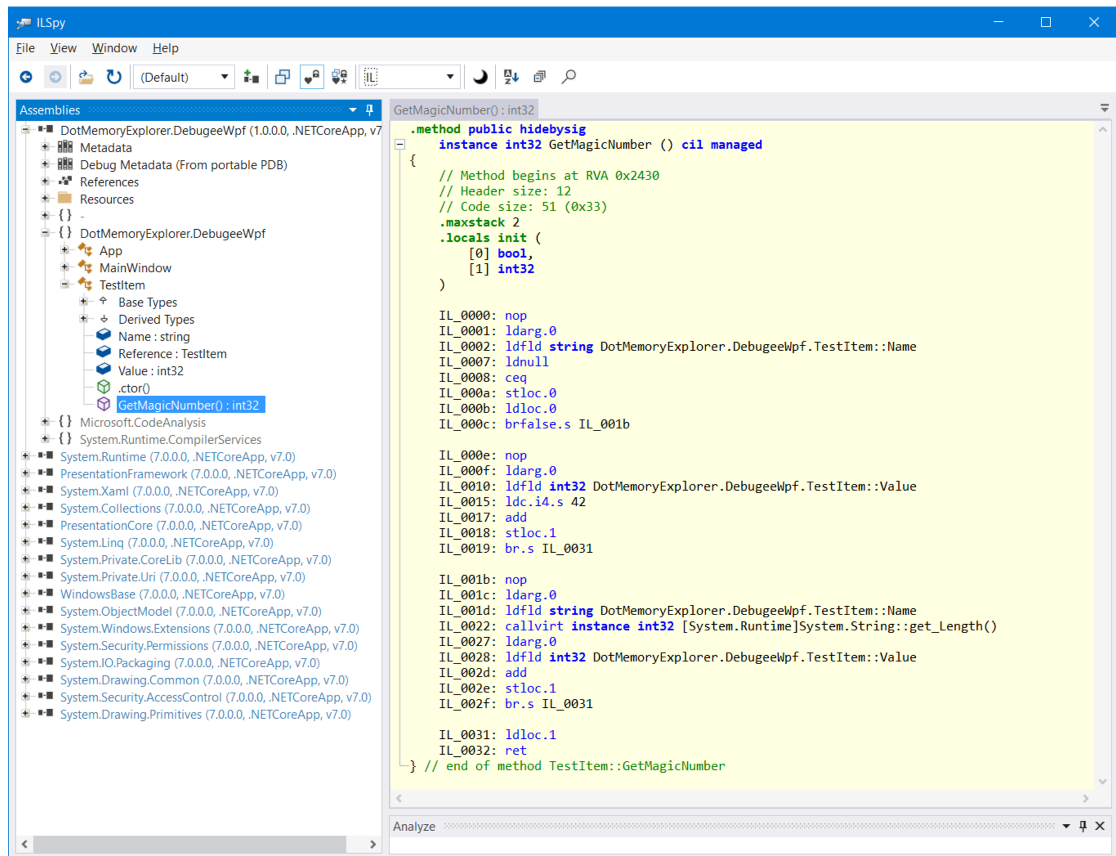
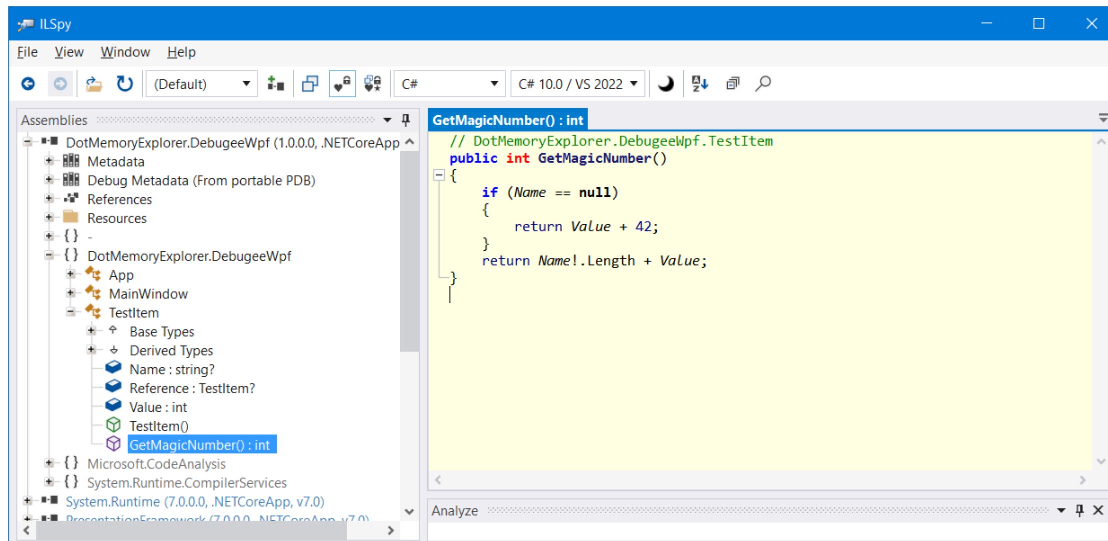


Figure 2.10 IL Code generated of GetMagicNumber method shown using ILSpy

Except for disassembly, ILSpy can decompile programs also. The language and its dialect are selectable in the menu. Decompilation is enabled by selecting C# in the menu. The latest version of ILSpy (version 7.2 was the latest available version at the time of writing this thesis) allows selecting C# standard version. Using this option user can enable (or disable) formatting of new language constructs added in the latest C# standards. In the following screenshot on figure 2.11 there is decompiled method GetMagicNumber to C# code. As you can see code is not the same as it was shown in listing 2.1. The difference is that code is completely missing the else clause, which is possible because the positive side has returned at the end. Decompiler also handled the null check of the else clause differently and recovered code using `!.` operator, which was not present in the source code and even it is not present in compiled IL. It is because both the programmer and compiler know that null was handled by a positive branch of the original

if, and it does not need to check it again. Still, the decompiler indicated that the null value was handled again using `!.` oeprator explicitely (`!.` operator is an operator which prevents crashes on accessing the field of a null value. If the expression on the left side is not null, it evaluates express as with standard `.` operator, but if the left side is null, it returns the default value in case of value type fields or propagates null in case reference types).



■ **Figure 2.11** C# Decompiled code of GetMagicNumber method shown using ILSpy

2.8 dnSpy and dnSpyEx

dnSpy is a debugger with an integrated disassembler, decompiler and compiler. The project is also Open Source and available on Github [23], but the original project is not maintained anymore. The successor of dnSpy is fork referred under the name dnSpyEx [24]. dnSpy development continues under this project. In this section, I will refer dnSpy using it's original name, but all information in this section applies to dnSpyEx also. dnSpy is internally based on several tools and libraries, including, for example, ILSpy, which was mentioned in the previous section. The dnSpy support modifying code in runtime. For this reason, dnSpy includes an open-source Roslyn compiler, which is the same engine that powers the compiler used by csc.exe (which is used by Visual Studio and MSBuild for compiling C# programs to IL). The dnSpy is a debugger and allows placing breakpoints inside decompiled C# code, stepping it, switching between threads and many other features known from debuggers. Interesting features are possible modifications of code in the runtime as well as modification of data types. dnSpy, for example, allows changing class names in the runtime. Changing flags of the class, visibility, adding members to the class and many others can simplify debugging of obfuscated programs. The dnSPy also has options to deobfuscated dnSpy is currently best in class tool for reverse engineering .NET apps, but its capabilities for memory content analysis, as discussed in this thesis, are minimal.

The dnSpy is shown on the screenshot in the following figure 2.12.

2.9 PerfView

PerfView is the last tool discussed in this section. Its main purpose is to connect to the diagnostics API of the .NET runtime and Windows and collect events and traces generated by the target

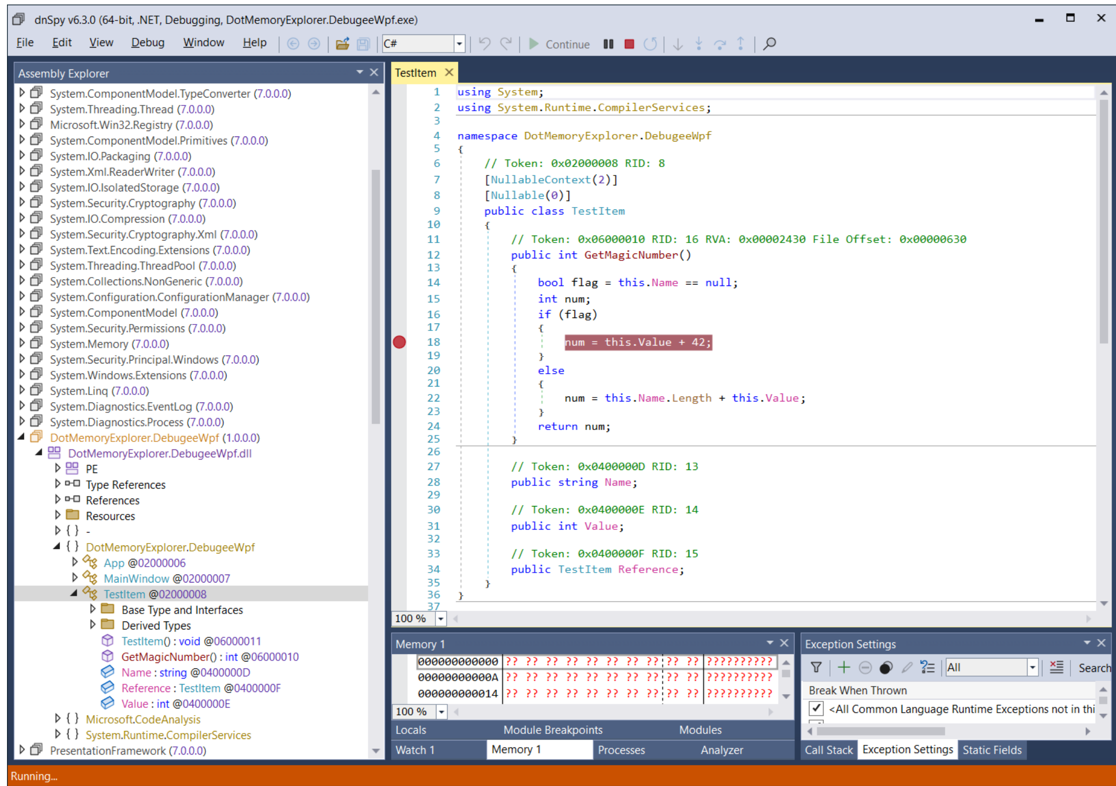


Figure 2.12 dnSpyEx attached to the simple testing program

program. The tool mainly targets performance-related events and can generate reports based on them. It uses the same API as tools mentioned in the previous section Dotnet CLI Diagnostics Utilities and also uses API for getting performance counters from the Windows operating system. While performance is not a very important topic of the application, there are still useful use cases of this tool for reverse engineering tasks. Events generated, collected and processed by this tool are side effects of the target program and thus can leak much information about the program. The tool also supports collecting stack trace information for every collected event, simplifying several analysis tasks. But since the tool target performance analysis, many details get hidden inside statistics.

The tool can generate a heap exactly as the already mentioned tool `dotnet-dump` described in section Dotnet CLI Diagnostics Utilities do. Even the output format is the same. PerfView can introduce a different point of view to heap dump generated in this way, but since it targets performance, most details about every object are hidden in statistics. On the following screenshot 2.13.

Besides collecting heap dump, PerfView can track all allocations of the .NET program. This is disabled by default because of performance impact on the target process but can be enabled by checking `.NET Alloc` option in `Advanced Options` when starting a collection. In this case, PerfView will collect metadata about object allocation and a stack trace indicating the place in the code that triggered the allocation. Outputs from this analysis is later grouped by stack traces, and there is in the current version of PerfView (version 3.1.0 was the latest version of PerfView at the time of writing this thesis) no way to access allocated objects information except statistics of allocations grouped per stack trace.

This was the last tool described in this section. The following section describes the Garbe Collector in detail.

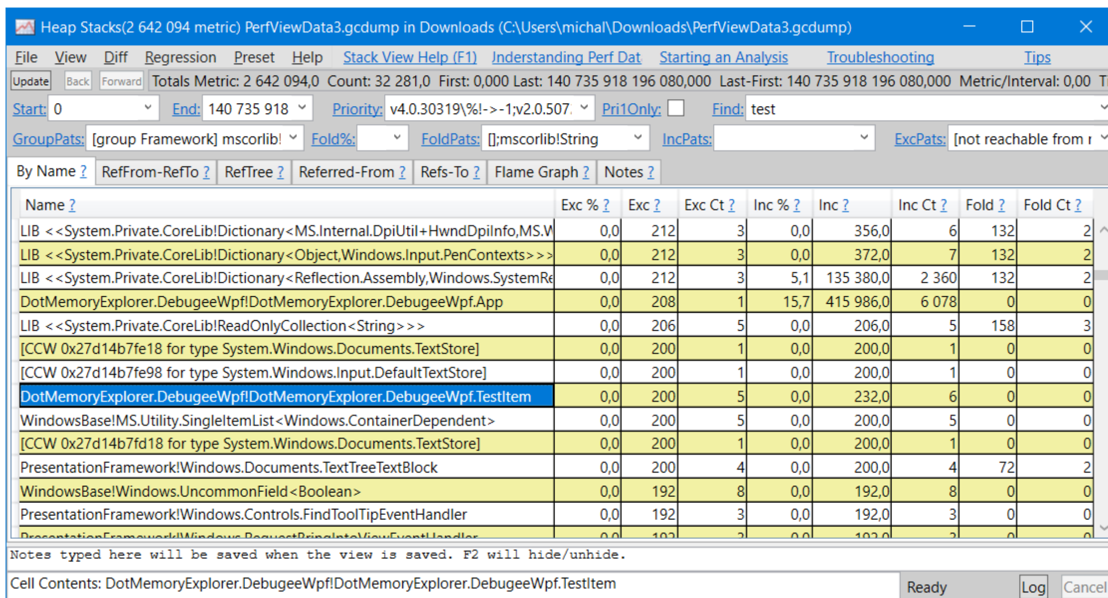


Figure 2.13 gcdump object visualisation in PerfView

Chapter 3
Garbage Collector

This section describes the theory of operation of one of the most important parts of .NET runtime, which most significantly affects the memory footprint of the .NET program.

Garbage Collector is part of .NET Runtime. As mentioned in .NET Implementations section, there are multiple implementations of .NET runtime. Unless otherwise specified main implementation of .NET Core runtime is used. This implementation is open source, and the source code of the garbage collector can be found in `src/coreclr/gc/` folder. The main part of the implementation is in `gc.cpp` file [25]. The file is very long, and at the time of writing this thesis, it has 49 301 lines of code which indicate the large complexity of the algorithm described in this section.

Garbage Collector is an algorithm which provides .NET runtime possibility to allocate objects on the heap, which this GC algorithm maintains. Under some conditions, the .NET runtime triggers Garbage Collection. In .NET, this means that GC stops all threads in the program, cleans unused objects from the heap and then resumes all threads and returns back control to the program.

3.1 Virtual Memory and Cooperation with Operating System

.NET process, like any other process, allocates memory from the Operating System (OS). The operating systems manage memory in blocks of 4096 bytes which are referred to as pages. Currently, all operating systems used on x86 computer implements the concept of virtual memory. Every process in the system has its own memory address space independent (except shared pages) on any other process. The operating system allocates a page from physical memory and maps them to the virtual address space of the process. Internally it is implemented using a translations table, in which the user bits from the memory address to several chained tables, and the last table contains the target physical address. Since pages are 4096 bytes long, a translated address consists of upper 54 (64 - 10) bits. Lower 10 bits are used as an offset to this block of memory.

Even though virtual address space is dedicated to every process, there are some requirements on addressing within virtual address space defined by the operating system. It is mainly restricted that addresses above specified thresholds are reserved for OS use. In practice, OS use this address for mapping pages containing binaries of programs and libraries as well as for pages containing its own data used related to the target process. This boundary efficiently limits the amount of usable memory by process, and this limit is OS-dependent since every OS use different requirements. There are also large differences between 32-bit and 64-bit OS and processes. In the case of a 32-bit system standard split was that 2 GiB of address space was reserved for the operating

system, and the remaining 2 GiB of address space was available for memory managed by user-space code. On 64-bit systems currently, only 48-bit address space is available, and the memory split on Windows is set to 128 TiB for OS and other 128 TiB for the user-space program.

One page can be mapped to the address space of several processes. This page is referred to as shared. Most memory important from GC's point of view is private and is mapped to the only one process which owns it. The process can request memory from the operating system using a system call. In the case of Windows, it is done using `VirtualAlloc` function, and in the case of `linux`, it is done using `mmap` system call. The process can later release pages if it does not need them anymore.

The Garbage Collector algorithm is the place of .NET runtime which internally calls `VirtualAlloc` (or `mmap` on Linux) for getting raw memory from the operating system. Garbage Collector supports releasing unused memory also, but this depends on configuration and happens only when the large enough block of memory is completely freed, and no object/data remains here. GC does not allocate a single region of memory. Instead, it uses several regions. A number of regions depend on several things and mostly on the type of GC used, which will be described later in this chapter.

3.2 Types of Heap

Current GC implementation uses two types of heaps for allocating objects. The implementation handles both heaps differently and, for example, runs different garbage collection processes on them. Many descriptions in this chapter will later split because implementation for handling heaps is different.

1. Small Object Heap (SOH)
2. Large Object Heap (LOH)

Small-Object Heap is used for most locations in the .NET program. A large object heap is used for allocating large objects. The boundary for deciding whether an object is small or large is hardcoded in `gc.h` [26] file on line 105 by `LARGE_OBJECT_SIZE` constant, which is in current implementation hardcoded to a value of 85 000 (bytes). Simply all objects larger than 85 000 bytes are allocated to Large Object Heap and objects with a size less than 85 000 bytes are allocated to Small Object Heap. Note that the number is not a multiple of 1024 or any other power of 2. There are no official statements why 85 000 was chosen, but most probably, it came from some internal research in Microsoft, which resulted in the best boundary of 85 000 bytes [1]. In the case of the array, the size limits naturally apply to the size of the whole array and not the single object inside the array (size of data type).

There is one exception when an object(s) with a size smaller than 85 000 bytes are allocated on Large Object Heap [1]. It is an array of more than 1000 doubles. This exception has a historical source. It comes from the 32-bit era. The reason for this exception is that double is the 64-bit wide (8-byte) data type, and its efficient processing requires an 8-byte alignment. But on a 32-bit system, the .NET implementation of Small Object Heap guarantees only a 4-byte alignment. This resulted in performance degradation when allocating doubles to the unaligned addresses, which happened randomly with a probability of 50

3.3 GC Variants and Configuration

Currently, there is a single implementation of GC in .NET Core, but this single implementation can run in different modes, which significantly impacts its behaviour. Currently, it can run in modes referred to as workstation and server. The other divide is done using running it in concurrent and non-current modes. Both decisions result in 4 different variants of the Garbage

Collection process. [1]. Modes differ in several parameters, which affect the frequency of running garbage collections and affect the type of garbage collection executed.

Workstation mode is used for programs running on client computers and is optimised for GUI programs and other programs frequently running on computers. Server mode is optimised for server application which targets high throughput (measured as, for example, processed HTTP requests per second). In the case of workstation mode, garbage collection runs more frequently. Because it runs more frequently, it always collects fewer objects, and collection is faster. In opposition, server mode triggers collections less frequently, collects more objects, and collections take more time. In server mode, collections take more time but occur less frequently, which in sum results in less program time spend by garbage collection (which pauses program execution on all threads) and increase overall program processing throughput [1]. Modes also differ in the number of heaps used. Workstation mode uses one heap set (heap set consists of one Small Object Heap and one Large Object Heap) for the whole program. Server mode, in opposition to multiple heap sets. The number of heap sets in server mode depends on available processor cores. Runtime will create one heap set for every processor available in the system. The listing of differences in this paragram is not exhaustive, and there are minor parameter differences, but these are not important in terms of this. While the workstation and server mode behaves differently, they are both implemented by the same `gc.cpp` file which is included twice with the `SERVER_GC` macro set and unset [1]. In `gc.cpp` file, this macro is then used in several `\#ifdef` blocks.

The second dividing of the .NET GC mode of operation is concurrent and non-concurrent. The non-concurrent mode works in a way that when GC happens, all thread gets suspended except one thread which runs the GC. Concurrent mode, in opposition, lets other threads run while GC is in progress. This has performance benefits but has severe impacts on implementation because user code can modify the references inside objects while GC is processing them. Even in concurrent-mode, there is a situation when GC need to stop executing all threads and do its work using a single thread. The purpose of concurrent GC is to minimise this time and do GC work as much as possible in the background thread, which runs when the program is running.

Users can select GC mode in the configuration file stored next to the program executable. On startup, the .NET runtime reads this file and uses the proper implementation of GC. In the case of .NET Core, the configuration is done using `\<program name\>.config.json` file loced in the same folder where the `\<program name\>.dll` is.

3.4 Object Layout and Metadata

In .NET, every object has three parts.

1. object header
2. pointer to MethodTable
3. object content

References to the object do not point to the location of its object header, but instead, they point to the location of the pointer to MethodTable. For this reason, the object header is located at the negative index to the base address of the object. The first two parts of the object have a size of one pointer. On a 32-bit system, both the object header and a pointer to MethodTable are 32-bit (4-byte) wide. On 64-bit systems, their size is 8-byte. This also applies to the object header. The content of the object header is the on both 32-bit and 64-bit systems the same, but on the 64-bit system object, the header uses only 4 bytes, and the remaining 4 bytes are in the current implementation zeroed. The structure has no information about the exact object size. This information is stored in MethodTable. There are requirements for object content length. It must be multiple of pointer size and cannot be zero. The empty object does not exist. Even

empty objects will have 4 or 8 bytes (depending on system bitness) long (unused) content. This restriction comes from GC because in plan phase (which will be described later in this chapter) reuses 4 or 8 bytes of object content for storing internal metadata used in this phase and later in the compact/sweep phase. For this reason, every object needs to have at least space for one pointer-size content.

3.5 Allocations

For allocating objects, GC uses a simple strategy. It holds a pointer to the free space of memory, and when runtime requests memory, then it returns the pointer and increments it by the size of the allocated object. Metadata about memory (for example, the size of free memory region) from which allocation is made is referred to as allocation context. GC triggers garbage collection when memory handled by a location context gets exhausted (in fact, it can happen sooner than it gets exhausted due to internally computing a location budget). This collection makes some free space. If, even after garbage collection, there is no sufficient space for newly allocated objects, then the GC extends the heap by allocating more memory from OS.

3.6 Generations in GC

The garbage collector implemented in .NET is generational. Generations mean that objects are grouped to predefined generations. Every object which survives the collection is moved to the older generation. Currently, .NET uses three generations, Generation 0, Generation 1 and Generation 2, for objects in the Small Object Heap. Large Object Heap do not use the concept of the generation. In some tools (and also in .NET internals as well as in GC API provided to programs), Generation 3 refers to objects on a large object heap. Generations allow running collection only against a subset of objects in memory. The idea of generations is based on the hypothesis that many objects in the program are temporal and many objects live in program memory for very long. For this reason, it makes sense to run garbage collection more frequently on a recently created object which most probably died before the collection was triggered, and it makes sense to run garbage collection against long-living objects less frequently because, in most cases, they will still survive. Every object is allocated in generation 0, and generation 0 is most frequently garbage collected. Garbage collection runs against some generation and always include all lower generation. When GC decide to collect garbage, for example, generation 1, then it automatically collects generation 0 also. If a recently allocated (in generation 0) object survived collection, then it became part of generation 1. Generation 1 is collected only sometimes. Not all garbage collections check objects in generation 1, but at some point, GC decided to check even objects in generation 1. In this case, objects in generation 1 are checked, and if they survive, then they are moved to final generation 2. GC collections of generation 2 are rare, and objects that survive this collection stay there until they get collected.

Implementation of generation is done using pointers to the Small Object Heap. Pointers split generations. Moving objects between generations does not require modifying objects but means that pointer splitting the generation will change its target location. Generations are ordered in reverse order on the heap. Since the allocation occurs by incrementing the pointer at the end of the heap objects, objects are automatically allocated in generation 0. The layout of the generation split of Small Object Heap is highlighted in the following figure 3.1.

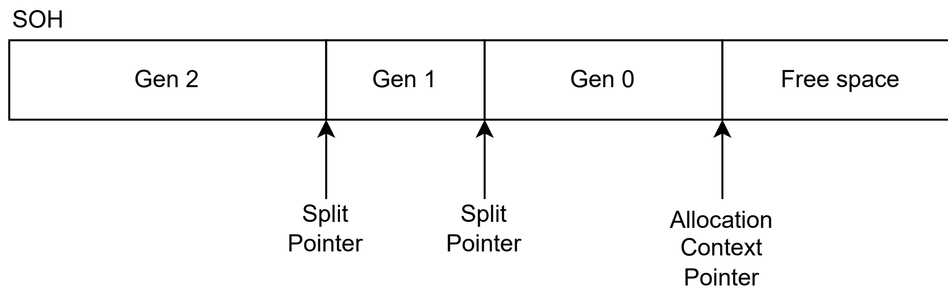


Figure 3.1 The layout of objects grouped into generations

During garbage collection time, GC will analyse which objects are used and which are unused at the end of this phase. In the end, regions of memory are marked or unmarked. For example, it may look like what is visualised in the following figure 3.2.

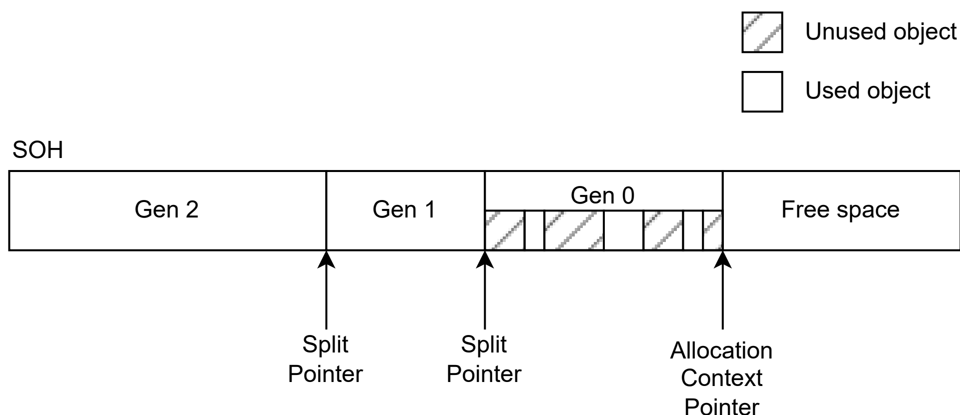


Figure 3.2 The layout of objects after marking objects in Generation 0

At this point, GC can select one of two strategies for collecting unused objects.

1. Sweep
2. Compact

The first one is a simple method which stores metadata about free regions in the list and later uses these lists and free spaces in new allocation contexts. This method is easier to implement and do not require moving object in memory and fixing all places where the object was referenced. After this collection, pointer generation, 0 and 1 split pointer will point after the last valid object in the previous original generation 0. Generation 0 now becomes empty, and a location pointer will be adjusted to start allocating objects in new generation 0. This is indicated in the following image in figure 3.3.

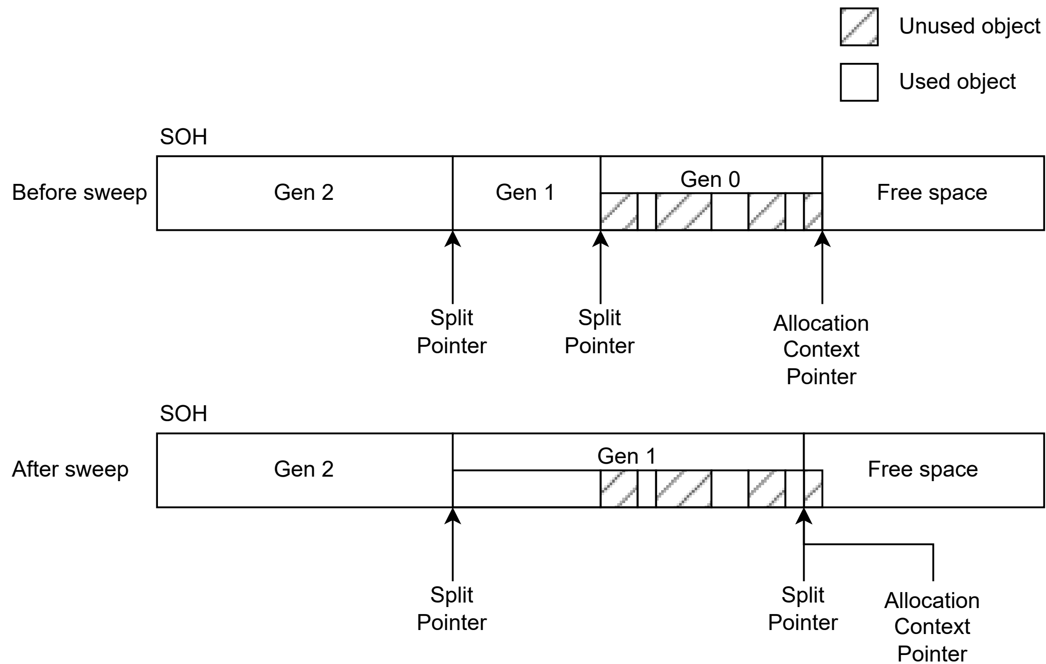


Figure 3.3 The layout of objects and their assignment to generations after sweep

The other strategy is compacting, which will move objects to reduce the fragmentation of the heap. It requires a lot of work and cannot be done in concurrent mode when the user threads are executing code. In concurrent mode, it will just result in non-concurrent behaviour if GC decides to go this way. The benefit of this mode is that it significantly reduces the fragmentation of the heap. Impact on objects and their assignment to generation is highlighted in the following image in figure 3.4.

The examples above show behaviour for collecting generation 0, but the same algorithm applies for compacting generations 1 and 2 as well. While user objects are always allocated in generation 0, GC internally use allocators in generations 1 and 2 also. They are used, for example, when moving objects to fill gaps caused by already collected objects.

3.7 Garbage Collection Triggers

As mentioned in the section above, one situation when GC triggers allocation is when there is not enough memory in the current allocation context. GC may trigger even if there is enough memory. This happens due to a location budget [1]. The allocation budget is the space of memory which is allocated before GC is triggered. It is not equal to free space which can be allocated. The allocation budget is used for triggering GC earlier, which results that this collection will collect fewer objects and consume less processor time. The allocation budget is computed statically only the first time, and all following allocation budget updates are based on statistics from the last garbage collection. GC adjust this allocation budget to achieve the best trade-off between collection frequency and processing time. Since GC allows allocating memory from non-zero generations internally, there are allocation budgets for generations 1 and 2 also.

The other way to trigger garbage collection is directly from the running program. .NET runtime provides GC class with a static Collect method, which can be used for explicitly triggering

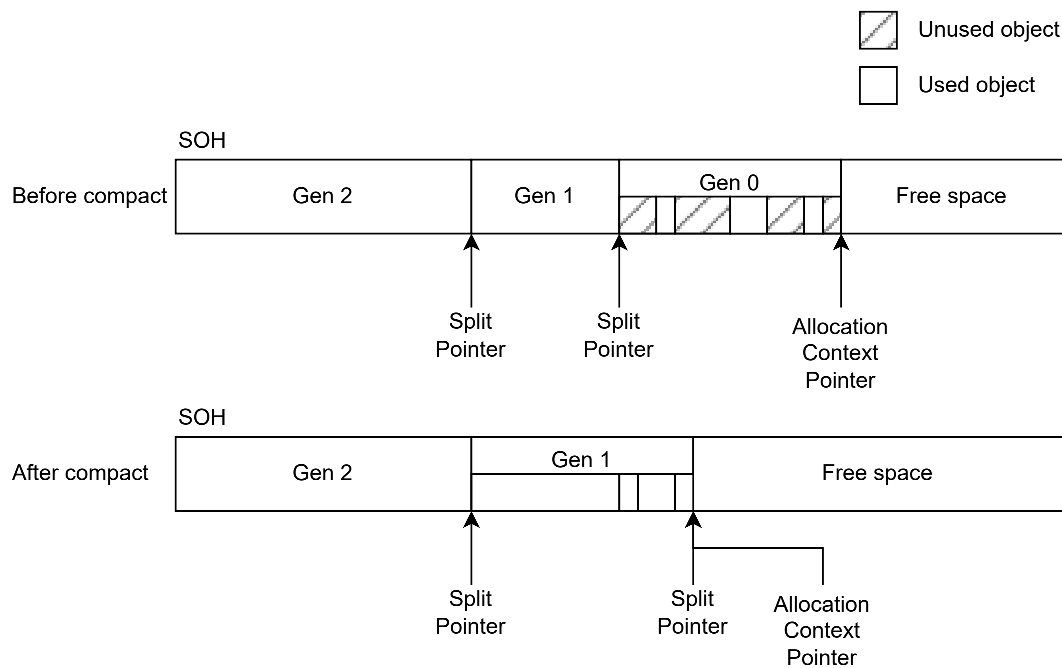


Figure 3.4 The layout of objects and their assignment to generations after compact

garbage collection. According to [1], it is not recommended to do that because it, for example, makes the calculation of the next allocation budget less precise since statistics are calculated based on data from the shorter terms.

The other way which explicitly triggers garbage collection is low memory notification from the operating system. The operating system sends this signal to all processes. The .NET processes to handle this notification by starting garbage collection, which may result in clearing some pages and returning them to the operating system, decreasing memory pressure on the overall system.

At last, there are some .NET runtime internal triggers, for example, when some thread dies. In this situation, GC is internally triggered even though allocation budgets did not exhaust yet [1].

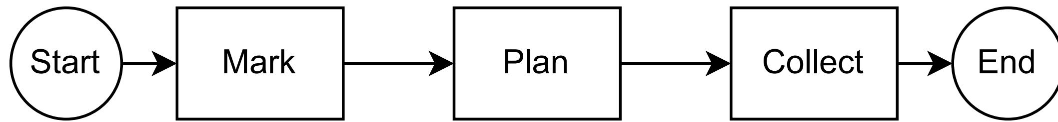
3.8 Garbage Collection Steps

Garbage collection on the highest abstraction level consists of three steps:

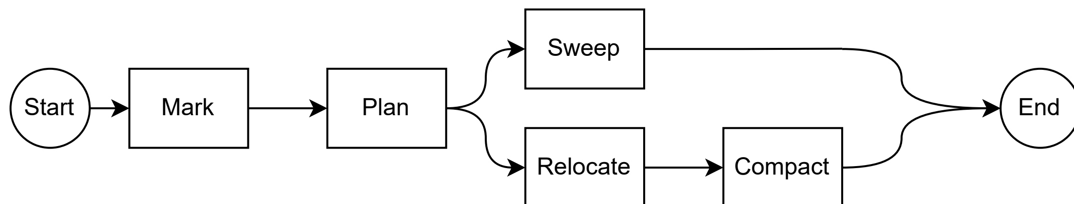
1. Mark
2. Plan
3. Collect

The most abstract view of their execution is highlighted in the following figure 3.5.

Many resources like [27] split the description of the last phase into two or three sub-phases, usually referred to as Relocate, Compact and Sweep. In this case, the diagram is not as straightforward as presented in figure 3.5, and instead, it looks as follows in figure 3.6.



■ **Figure 3.5** Sequence of GC phases execution



■ **Figure 3.6** Detail of sequence of GC phases execution

Both diagrams and descriptions are correct. The only difference is that the second one is more detailed.

Some of the steps were basically already mentioned in this chapter. The next section contains more detailed descriptions and more implementation details about every GC step mentioned in this section.

3.9 Garbage Collection Mark Phase

The Mark phase is the first phase of the Garbage Collection process. Its purpose is to mark objects that are used by the program and not mark objects which are not used anymore. The used object is an object which is accessible by the program or is referenced by some other marked (used) object and thus is accessible. References between objects can be visualized using a graph. For every accessible object, there are one or more paths from some GC root which is not on the heap. One of the possible GC roots is, for example, a variable on the execution stack of some managed thread. When marking, GC needs to start at this point and recursively (but implementation is done using a loop instead of recursion [1]) travel over all references in all objects and mark all object. An example of a graph needed to traverse is shown in the following figure 3.7. In the illustrated example, after the marking phase, objects A, B, and C will be marked (used), and object D remain unmarked (unused).

The program stack is not the only GC root. Other GC roots are, for example, static variables. Also, since JIT use optimizations, the variable can exist only in CPU registers, and valid reference to an object will never store in the RAM. A description of how the GC determine which registers and stack locations contains references will be described later in this chapter in section Manual parsing.

As mentioned before, for performance reasons, GC usually (most of the time) operates only a subset of objects and processes only a subset of generations. The marking phase also works in this way. If there is a reference (for example, on the program stack) to an object in the generation which is not processed, then this object is not marked and visited by the algorithm. This improves overall performance, but the issue is that this skipped object can reference other objects, including objects in a generation which are currently processed. When not handled, these objects remain

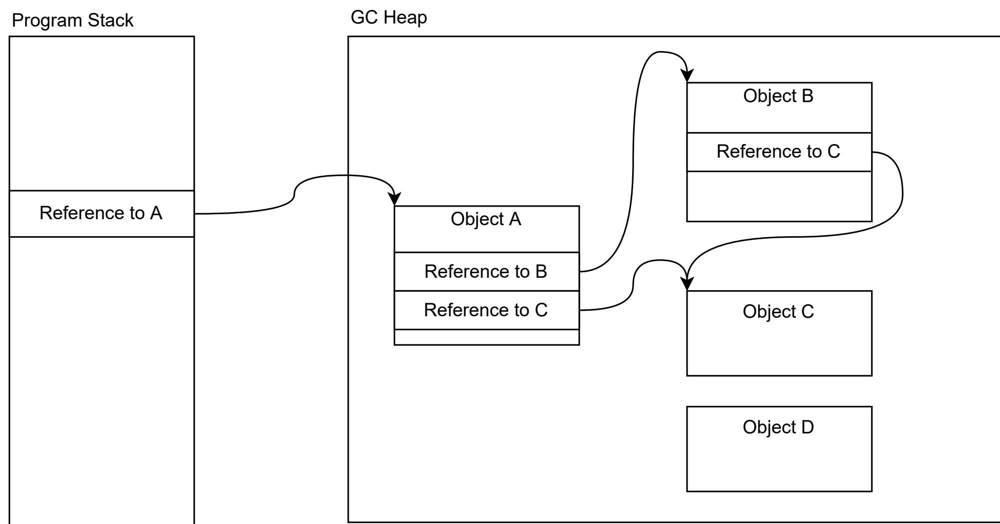


Figure 3.7 Example topology with one GC root and four objects

unmarked since their reference holder object is also unmarked. In a later phase, these objects get collected, but they are in use! For this reason, GC implements the concept of remembered sets, which are used to store information about references from older-to-younger references. Because there are more than two generations (the current .NET implementation uses three generations), there are more remembered sets. Because programs usually have a lot of references, the current implementations do not implement remembered set as a list, but instead, the concept referred to as a card table [1] is used. A card table is implemented by a dedicated bitmask, which has one bit per memory region of defined size (256 bytes in the current implementation), and the bit for this region is set (in implementations nomenclature, it is referred to as a dirty card) when the region contains an object containing reference to the object in the younger generation. This bitmask is managed when the program assigns a new reference to the object field. Reference assignment is not a straightforward task (like, for example, in C is), and every write operation use barrier which internally updates the target bit in the card table. Now we can go back to the marking algorithm and better explain which objects are processed in this phase. GC algorithm in mark phase processes not only objects in generations intended to collect but also processes objects which are in older generations on memory locations which are in card table marked as dirty.

At the end of the phase, the algorithm knows which objects are reachable from any GC root and which are not. The mark information is not stored in any dedicated group, but instead, a lower bit of the MethodTable address table is set. Since MethodTable addresses are always aligned, there are always two or three 64-bit system bits unused. For this reason .NET developers decided to use one of these bits for storing marks indicating object status for later garbage collection phases.

3.10 Garbage Collection Plan Phase

The planning phase is the most complex phase of the overall GC cycle, but in runtime, it is not the most time-consuming. The most time-consuming is usually the compact phase which copies large blocks of memory but otherwise is trivial [1]. The compact phase is not mandatory,

and GC can decide that it will use less time-consuming operation for clearing unused objects. This decision is made in the Plan phase. This is not the only task of the planning phase. In the planning phase, GC also prepares information which significantly simplifies the processing of the later phase. Note that in the Plan phase, are Small Object Heap and Large Object Heap processed differently? Generally, LOH uses a subset of algorithms and features used for SOH. LOH does not have a generation concept and has a simpler structure, so its processing is easier.

In the planning phase, the GC starts grouping objects and tries to process consecutive objects in the batches. In this phase, instead of objects, it starts working on groups referred to as *gaps*, which indicates a group of unused objects prepared for cleanup, and *plugs*, which are groups of consecutive live objects [1]. After this rearranging, the plugs and gaps are alternating in the memory. To simplify the implementation, the heap is designed in a way that every heap segment allocated from the operating system has metadata and one free object at the beginning. Since there is always one free object at the beginning, every plug has to process no matter whether the first real object is or is not marked in the Mark phase. This transforms, converting a sequence of objects to a sequence of alternating gaps, and plugs are highlighted in the following figure 3.8. This conversion is, of course, done only against objects in generations which are intended to collect.

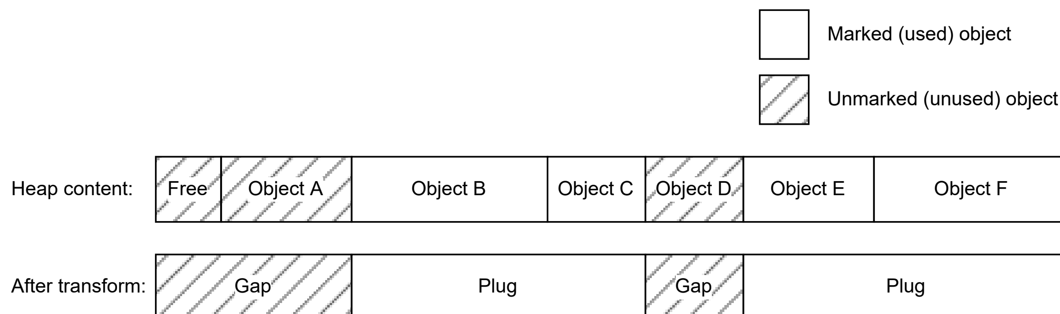


Figure 3.8 Heap conversion to plgus and gaps

Metadata about gaps and plugs are referred to as `gap_reloc_pair`. They are not stored in any dedicated collection. Instead, they are stored at the end of each gap and replace the content of some original object; since this object is unreachable anymore, it is not a problem that its content is overwritten.

Metadata computed for every gap and plug pair contains information about gap size and relocation offset, which can be later used for computing the new address of the object after possible moving and two offsets pointing to the metadata of other gap/plug pairs [1]. There are two offsets pointing to two-gap/plug pairs. The algorithm works in a way that the gap/plug pairs are organised into binary trees, which later allows a quick search for plug or gap location from any arbitrary address which can point into the region. This is crucial because, as we have stated, plugs can group multiple objects, and in later phases, we will need to find new addresses of these objects, but non-first objects have an original address somewhere in the middle of the plug, and there is no other way to find related gap/plug metadata except searching in the already build binary search tree.

The concept itself is quite easy to understand, but practical implementation is more complicated because GC needs to support pinned objects. Pinned objects are objects which are marked with pinned flags, and these objects will never get relocated. Pinning is used, for example, for objects which are passed to some unmanaged (for example, C) code. In managed (.NET) code, we know all locations of references to an object, and in the case of moving object, we can fix

them, but in the case of unmanaged (for example, C) code, we do not know places where the code promotes the pointer and how does it deal with it. For this reason, the pinning concept was added to .NET, and when the object is pinned, GC must ensure that the object never changes its location [1]. GC algorithm needs to take these objects into account, and when computing relocation offsets, it needs to handle these objects. The algorithm treats the pinned objects as special pinned plugs, but there is an issue with storing metadata for plugs in two cases. Normally metadata about plugs is stored in a gap preceding the plug, but pinned plug can be preceded by a normal plug and not a gap. This case is, in the current implementation, resolved in a way that metadata are stored inside the preceding plug, and its destroyed content is backed up to the special GC internal queue for restoration in a later phase. The other issue is with the plug following pinned plug. In this case, for storing metadata of following normal plug, we also do not have space, and in this case, we can't back up and overwrite part of a pinned object because it can be accessed by unmanaged (for example, C) code at the same time. In this case, pinned and normal plugs are merged, and the originally normal plug will become part of the pinned plug, which reduces the need to store its metadata but also means that this object will not relocate (which consecutively makes fragmentations of the heap). There is one more caveat related to pinned objects. They are not taken into account in the case of computing generation boundaries. Due to this, pinned objects can change their generation from younger to older (desired behaviour), stay in the same generation or even travel from older to younger. The former two naturally should never happen based on generational design, but in the case of pinned objects, they can happen, and in the case of merging pinned plug with the following normal plug, this can happen even to originally non-pinned (normal) object [1].

At the end of the plan, the phase algorithm makes a decision about which phase will follow. It has two options, both described in detail in later sections. This decision is made mostly on statistics related to the fragmentation of the heap. These statistics are computed on the fly when processing gaps and plugs.

3.11 Garbage Collection Sweep Phase

Sweep collection is based on making free-space objects from gaps and adding them to the free list. Free lists are used when selecting a new location context in the case when the actual became exhausted. Sweep operations do not resolve heap fragmentation but do not require copying any memory (with the exception of restoring the content of objects whose content was overwritten by gap/plug metadata) and fixing references. In this phase process use gap/plug metadata created in the previous Plan phase and use them for processing gaps in batches. Every gap can consist of several consecutive dead objects, and sweep operations create a single free object with the size of all destroyed objects which created the previous gap.

3.12 Garbage Collection Relocate and Compact Phase

In some cases (mostly depending on heap fragmentation), GC can decide that it will go by relocating way instead of sweeping way. This way consists of two steps. At first, it needs to fix (relocate) all references to this object. References are updated before the actual moving of objects happens. It needs to apply an offset to all pointers which point to this object. The second step is moving the objects. This model is not expensive in terms of computational power because all relocation offsets were already calculated in the Plan phase. But it still takes most of GC time because it makes memory-intensive moves of large memory segments.

3.13 Interactions with Runtime Interactions

GC is not the only part of the program in the .NET process, and when the collection is needed, it needs to communicate with other runtime parts, especially its Execution Engine (EE). The most important part of GC and EE cooperation happens when stopping all threads of the program (except one running GC). This is not a trivial task because program memory may be in an inconsistent state since most operations are implemented using multiple native processor instructions, and stop can happen anytime, including in the middle of the execution of single IL instruction. Remember, for example, write the barrier described in section Garbage Collection Mark Phase. This barrier, after reference, writes updates dirty flags in the card table. If GC triggered by a different thread attempts to pause the program after the reference was written but before the dirty flag in the card table is set, then it naturally processes an inconsistent heap, which can result in severe faults. [1]

There is a safe point concept to ensure that the program can be interrupted safely. Safe points are locations in the code where it is guaranteed that the heap is in a consistent state and GC can operate on it. At the beginning of GC, it pauses all threads and checks which threads are and which are not at a safe point. Threads which are not in the safe point are temporarily resumed, and they run as long as they reach the first safe point. The current implementation of the JIT compiler can generate safe points in two modes:

1. Fully interruptible. Every IL instruction is surrounded by safe points.
2. Partialy Interruptible. In this mode, only part of the construction is surrounded by a safe point. In case of stopping on a non-surrounded instruction thread, the suspension will take because the program will need to wait longer until the interrupted thread reaches the first nearest safe point.

JIT compiler cheese is one of the methods above. In the case of code, for example, containing short static iteration count cycles or consecutive short computational operations, it usually generates as partially interruptible. A safe point is always added to every function call because, in many cases, it is not clear how long the function will run. In opposition, fully interruptible code JIT generates, for example, when code contains loops with an unknown number of iterations. In this case of omitting a safe point, it may take a long time to reach the first safe point by the program. For this reason, this code is generated as fully interruptible, and the code contains a safe point around every instruction.

When all threads stop, GC can find a structure named `gc_info`, which, for every safe point of the method, contains information about stack locations and CPU registers containing references to living objects. These act as GC roots. There are several optimizations done, and for example, the object can be stacked unmarked as live right after the program executes the last instruction of the method using that object. If it is not used anymore, it is *removed* from `gc_info`, and GC can collect this object even before the method ends. For performance reasons, `gc_info` is very compressed and contains only differential changes between safe points. [1]

This was the last part of the description of the .NET Garbage Collector. In this chapter, the thesis describes its architecture, operation, phases, data format and many implementation details of current GC implemented in .NET Core and .NET Framework. In the next chapter, the thesis will describe options for obfuscating .NET programs.

Chapter 4

Obfuscation

This chapter thesis describes standard techniques used for obfuscating .NET programs and shows the impact of some freely available obfuscators on the testing program.

4.1 Motivation

Like in the case of any other programs, .NET programs need to be sometimes obfuscated. The purpose of obfuscation is to protect the program against reverse engineering, which, in general, is the process of converting compiled program back to the source code. Some needs for obfuscating are legitime. Liegitime's need for obfuscation is, for example, the need for hiding high-value business logic or, for example, protecting the algorithms responsible for checking licence keys from reverse engineering (cracking). In many cases, obfuscators are *abused* for protecting harmful from reverse engineering and related security analysis, which are attempted against this dangerous program. Obfuscators never make analysis completely impossible but can significantly increase its complexity. Using obfuscators have some disadvantages, like a decreased performance of the program. In the worst case, obfuscation can change the program's behaviour, and in the worst case, obfuscation can cause program crashes.

The situation in the .NET is slightly more complicated than in natively compiled languages because the .NET needs to store much information from the source code directly in the final binary metadata, even when building a project in Release mode without debugging capabilities enabled. It is because .NET offers much more reflection APIs for accessing code information. In the .NET, programmers can use reflection access by naming all classes, methods, properties, namespaces and assemblies. .NET binary also holds information about data types used in all method parameters and return value data types. Since this information is available at runtime, they need to be encoded somehow in the binary. Disassembler and decompiler tools like ILSpy mentioned in section .NET Overview use this information encoded in Binnary, and all this information is present to the user. Metadata in the binaries makes these tools extremely powerful when analysing non-obfuscated binaries, and in some cases, they are robust even against obfuscated ones. Since the IL is a highly object-oriented assembly language and object metadata is available, it makes even .NET decompilers extremely powerful.

4.2 Obfuscation techniques

The following subsection of the thesis will describe standard obfuscating techniques used by available obfuscators, described later in this chapter. The list of this obfuscation is not exhaustive

and contains examples of obfuscation which are implemented in any of the obfuscators mentioned later in this chapter. The description in this chapter targets the .NET-related aspects of obfuscation and contains obfuscations related to .NET programs.

4.2.1 Symbols protection

The most fundamental obfuscation is renaming all metadata symbols in the binary. These tools are used for destroying the best property simplifying reverse engineering of .NET programs [28]. This is the most popular obfuscation in .NET, and almost all obfuscators do it. There are two common approaches. The first approach use mapping to random strings and the second approach uses encryption. Mapping to random string is stronger obfuscation from a security point of view, but this approach completely destroys the option to use reflection in the program anymore. In opposition, encryption possibly allows to use of reflection, but the obfuscator has to search for its usage in the code and replace it with logic encrypting and decrypting the inputs and output from the reflection API accordingly. Obfuscators are quite powerful because .NET allows Unicode encoding in all names, and they heavily utilize as well as white characters, which are prohibited by most specifications of languages with available .NET compilers (all three C#, VB.NET, and F# prohibit white characters in the name) but strings inside final assembly can contain them, and programs using classes, methods, properties with these names work in current runtime. Analysis with the tool without support for deobfuscating or at least escaping white characters becomes significantly more complicated because all identifiers are invisible.

4.2.2 String Protection

Another very powerful protections are string protection. Since they need to be restored in runtime mapping technique is impossible to use, and most obfuscators which support this obfuscation encrypt the string and implement proper decryption routine in obfuscated code [28]. String protection targets strings which are hardcoded in the program. Disassembled and decompiled code will not contain human-readable strings in the code anymore. Instead, there will be a routine which decrypts the string in the runtime. Obfuscation is powerful against static code analysis but is less powerful against dynamic analysis (which is the analysis targetted by this thesis). In .NET, every string is instantiated, and in runtime, it is located inside the instance of the `System.String` class. .NET Memory Explorer tool created as part of this thesis can find these instances on the managed heap in runtime and can search and show identifiers as soon as they are decrypted in the runtime. The little complication is object lifetime because, in the normal program, string instances with hardcoded strings live for the whole time of running the program (even if they are not actively used), but in the case of decrypted instances usually lives only when they are needed and they get collected. Some strings can be missed just because they are very short living.

4.2.3 Codeflow Change

While compiler and JIT do their best to make code short and fast as much as possible, obfuscators do the opposite. The purpose of this obfuscation is to make some parts of code flow more complicated [28]. Obfuscation works by adding dead code to the program, adding always true or false conditions and many others.

4.2.4 Operation and Constant Substitution

This obfuscation is similar to the previous one. But it works on the command level. Note that obfuscations are frequently combined. An example of this obfuscation is making arithmetic

operations more complex and replacing constants with complex expressions.

4.2.5 Virtual Machine

Virtual Machine implementation heavily affects program execution and splits program blocks to virtual instructions, which are encoded in an array, and they are later executed by the interpreter [28] [**nirev-obfuscations**]. The interpreter executes these virtual instructions. Obfuscation significantly changes the code flow graph of the program because it contains the code flow graph of the interpreter instead of the code flow graph of the program. The original program becomes flattened in the array. Obfuscation can be nested. Obfuscation is more powerful in native code with fewer decompilers available. .NET decompilers can (without combining with other obfuscations) decompile it to readable code, and the reverse engineer can see the implementation of the virtual machine as well as the implementation of virtual instruction. Obfuscation is extremely strong when nested and combined with other obfuscations.

4.2.6 Metadata Destroy

This obfuscation attempts to break reversing tool by modifying internal metadata. It targets differences between processing metadata in runtime and by tool. Currently, there is an implementation which adds the MethodDescriptor structure, which contains a specific sequence of IL instructions which causes ILSpy to crash.

4.2.7 Anti-Design Time

Anti-design time obfuscations prevent reusing Windows Forms and Windows Presentation Foundation (WPF) components from the libraries. .NET libraries allow instantiating of components from libraries in different programs, including programs created by reverse engineers. A reverse engineer can create his own Graphical User Interface (GUI) program, reference reversed library in his/her project and add components from the analysed library to his/her window to access features of the component which are otherwise hard to access in the original program (for example accessible only after entering licence key). Obfuscation makes this usage complicated by crashing the graphical designer in Visual Studio after adding a component from the library to the window [29].

4.3 Available Obfuscators

Several freeware, open, source, and closed-source obfuscators are available on the market. Open-source libraries dnlib (also used by the dnSpy tool) and Mono.Cecil allows editing .NET binaries, and many, especially open source, obfuscators using these libraries were created for learning purposes, and they are not maintained anymore. In this section, the thesis describes several open-source and paid obfuscators which are continuously maintained and developed. In GitHub project `NotPrab/.NET-Obfuscator` [30], there is a much longer list containing links to many other obfuscators, including archived projects.

4.3.1 Obfuscar

Obfuscator is an open-source obfuscator [31] which supports Symbols and string obfuscations. Nowadays, Obfuscar is available using Nuget [**obfuscar-nuget**] packaging system (Nuget is a packaging system for distributing official and community .NET libraries and tools). Obfuscar is distributed as a command line tool integrating with `dotnet` command. Internally it is based on

Mono.Cecil library. Recommended use case mentions triggering this tool in a post-build event setting in the Visual Studio project, but the tool can be triggered manually.

4.3.2 MindLated

MindLated is another open-source obfuscator [32]. In comparison with Obfuscar, it is less documented but supports more obfuscations. GitHub main page [32] of this project mentions that this tool was created for learning purposes, but it is still actively maintained. MindLated supports more obfuscations than previously mentioned Obfuscar. It is also based on the different libraries for modifying binaries instead of Mono.Cecil, used by Obfuscar, uses lib. It supports obfuscations which are selectable in the obfuscator GUI including symbol renaming, string encryption, metadata destruction, control flow obfuscations and anti-debugging.

4.3.3 Z00bfuscator

Z00bfuscator is another open-source tool based. It is very simple, and the only obfuscation it can do is symbol obfuscation. It is easily identified because instead of an entirely random identifier, it uses an identifier prefixed with `[SECURED-by-Z00bfuscator]-`, which are easily visible in ILSpy.

4.3.4 Eazfuscator.NET

Eazfuscator.NET is one of the paid obfuscators which supports most of the techniques described in section Obfuscation techniques. It supports symbol renaming, string encryption, code flow obfuscation, anti-design time [29], automatic conversion of code for using a virtual machine and many other small obfuscations [28].

4.3.5 CSharpObfuscator

The last mentioned tool is CSharpObfuscator which is one of the few tools which do not process compiled binary but instead obfuscate original source codes. It is available in commercial editions, and a free edition with limited options is available as an open-source project on GitHub [33]. It supports symbol and string obfuscations.

4.4 Deobfuscators

Since obfuscators exist, reverse engineers naturally are interested in tools which will reverse the obfuscator operation. The tools responsible for doing this task are named deobfuscators. Similarly to the analysis of .NET memory, deobfuscation is implementation specific, and in the case of deobfuscators, it means that deobfuscation tools target obfuscations created by some specific obfuscator. A lot of available deobfuscators are mentioned in Github project `NotPrab/.NET-Deobfuscator` [34], which is maintained by the same author as a list of obfuscators in project `NotPrab/.NET-Obfuscator` [30] mentioned in previous section Available Obfuscators.

4.5 Impact of Obfuscation on Program Memory Footprint

The impact of mentioned obfuscations and obfuscators on the memory footprint differs for every obfuscation.

1. **Symbol obfuscation** does not directly impact the data structure in the memory but removes all identifiers simplifying the determination of data meaning.

2. **String obfuscation** affects memory footprint significantly because original strings are not in the memory anymore, and as mentioned in String Protection section, dynamic decryption of strings brings them to the heap, but their lifetime is different, and many strings get collected by the garbage collector after they are used while in original program strings remained in the memory for a whole program execution time due to string interning which is the concept of deduplication strings used in the program code [1].

3. **Codeflow obfuscation** does not impact the layout of the objects in the memory as well as their metadata but can change object lifetime. It may also cause additional allocation and boxing, which will bring new objects to the help which was originally allocated only on the stack (this is a case of value types like `int` or structs). Since code flow is changed and more complicated objects may live on the heap longer, which can be for reverse engineering beneficial, but most probably, it does not compensate for the impact of the original obfuscation.

4. **Operation and code substitution** behave similarly to code flow obfuscations. They can introduce new objects on the heap, but their lifetime is very short. At the end of the execution of the substituted operation, there is a fully evaluated value available in the memory like it was in the case of the original program.

5. **Metadata Destroy obfuscation** targets breaking reversing tools and modifying metadata (which are located in the memory). It naturally directly impacts the memory footprint of the program (at least it introduces *broken* structure to the memory), but at least in the case of currently implemented InvalidMD obfuscation [32], it does not change the layout of any object and does not impact the lifetime of the object.

4.6 Obfuscator demonstration

For testing obfuscators there, I chose open source obfuscator MindLated. Like many other obfuscation tools, it has a black background and is decoded by pictures. It is a GUI tool which allows the selection of up to 20 obfuscation which the tool will apply to the passed .NET program assembly. It supports passing only one assembly. Currently, the .NET Core compiler supports merging assemblies automatically, but in the case of older .NET Framework user needs to merge them manually, for example, using the ILMerge tool [35]. Obfuscator was tested against a simple testing program, and their behaviour will be shown on method `GetMagicNumber` from `TestItem` class which source code was shown on listing 2.1 and decompiled non-obfuscated class was shown in ILSpy on figure 2.11. For the first test, the tool was used with configuration related to the content of `GetMagicNumber`. The following obfuscation options were enabled:

1. String Encryption
2. Strong Online Decryption
3. Int confusion
4. Arithmetic
5. Renamer

After obfuscation size of the binary grew from 12 800 bytes to 16 896 bytes which is 1.32 times more. After decompiling the new assembly with ILSpy, we cannot see most names of Namespaces, Classes, Methods and Properties anymore. Some names remained. It is because they rely on an underlying framework. Since the application is WPF, any references to WPF classes are visible in decompiled tool. A number of classes in the main namespace used by application logic are

unchanged and based on class inheritance (one class inherits from the Window class, another class inherits from the Application class, and TestItem class does not inherit from any other class) we can simply assign obfuscated class names to original one. Decompiled test item class is shown in the following figure 4.1

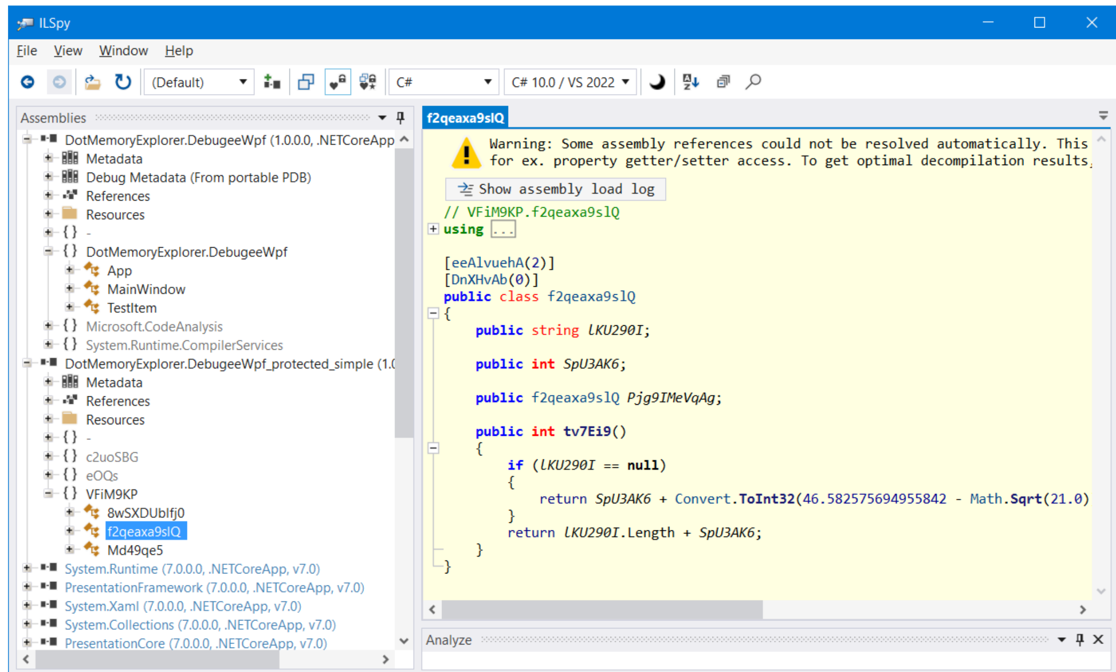


Figure 4.1 Decompiled obfuscated `TestItem` class

As we can see, the content of method `GetMagicNumber`, which is now named `tv7Ei9`, is similar to the content shown in figure 2.11, but because Int confusion obfuscation was enabled, constant 42 is now replaced by expression `Convert.ToInt32(46.582575694955842 - Math.Sqrt(21.0))`.

In the next experiment, all obfuscations are offered by MindLated. In this case, the obfuscated class now contains an explicit constructor, but their decompilation emits several warnings and ends with failure, as shown in the following figure 4.2.

But when switched from decompilation to disassembly view (by changing language choice from C# to IL in ILSpy), we can see that it is just an explicit call of `System.Object` class constructor as shown in the following figure 4.3. The last IL possible view containing IL and decompiled C# code indicates that the decompiler was most probably messed with a lot of dead code here.

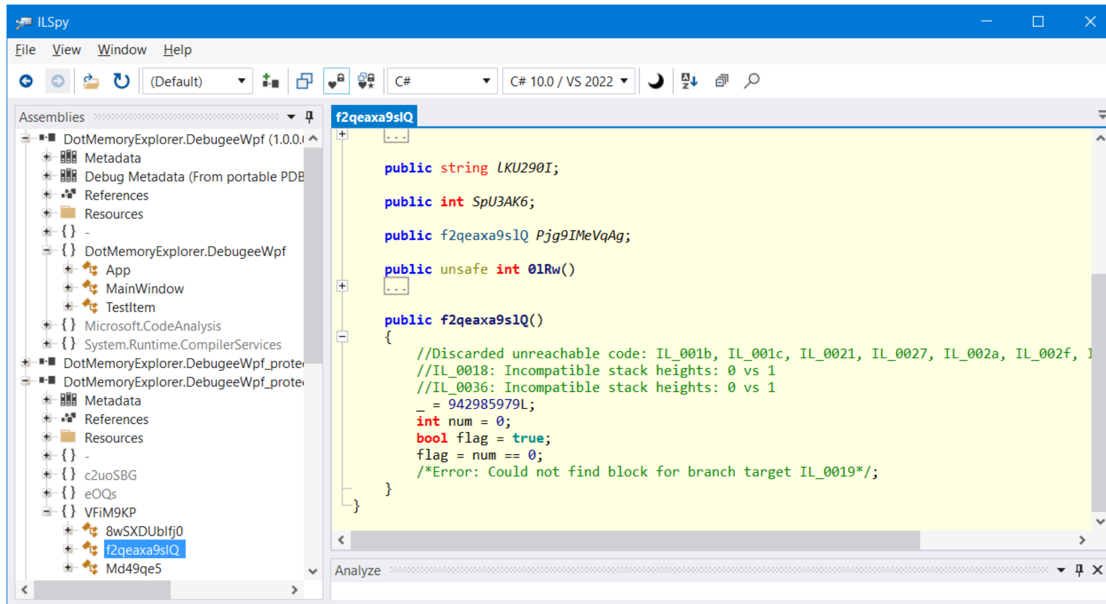
Except for the newly added constructor, there is still `GetMagicNumber` method, but now it is much more obfuscated. Decompiler emits 266 warnings when decompiling it, and decompiled executable code contains 154 lines of code, including several comments referencing that `calli` instruction with passed arguments is not currently supported by ILSpy. The beginning of this code is shown in the following figure 4.4

Unluckily, programs obfuscated in this way do not work anymore. When I attempted to start, it crashed with `System.TypeLoadException` and following message from listing 4.1:

Code listing 4.1 `System.TypeLoadException` exception generated by the faulty program

```
A ByRef or ByRef-like type cannot be used as the type for a static field.
```

The issue is .NET runtime implementation specific. Some obfuscations provided by this tool were designed for use with older .NET Framework runtime implementation, and they do not



■ **Figure 4.2** Decompiled MindLated obfuscated class `TestItem` with all possible obfuscations

```
.method public hidebysig specialname rtspecialname
    instance void .ctor () cil managed
{
    // Method begins at RVA 0x246f
    // Header size: 1
    // Code size: 8 (0x8)
    .maxstack 8

    IL_0000: ldarg.0
    IL_0001: call instance void [System.Runtime]System.Object::.ctor()
    IL_0006: nop
    IL_0007: ret
} // end of method TestItem::.ctor
```

■ **Figure 4.3** Decompiled MindLated obfuscated explicit constructor `TestItem`

work with the .NET Core (version 7.0) runtime. The problematic obfuscations which are needed to disable for making the program work are the following:

1. Local to Field
2. Local to Field V2
3. Calli
4. Proxy Meth
5. Renamer
6. Anti Tamper
7. Invalid MD

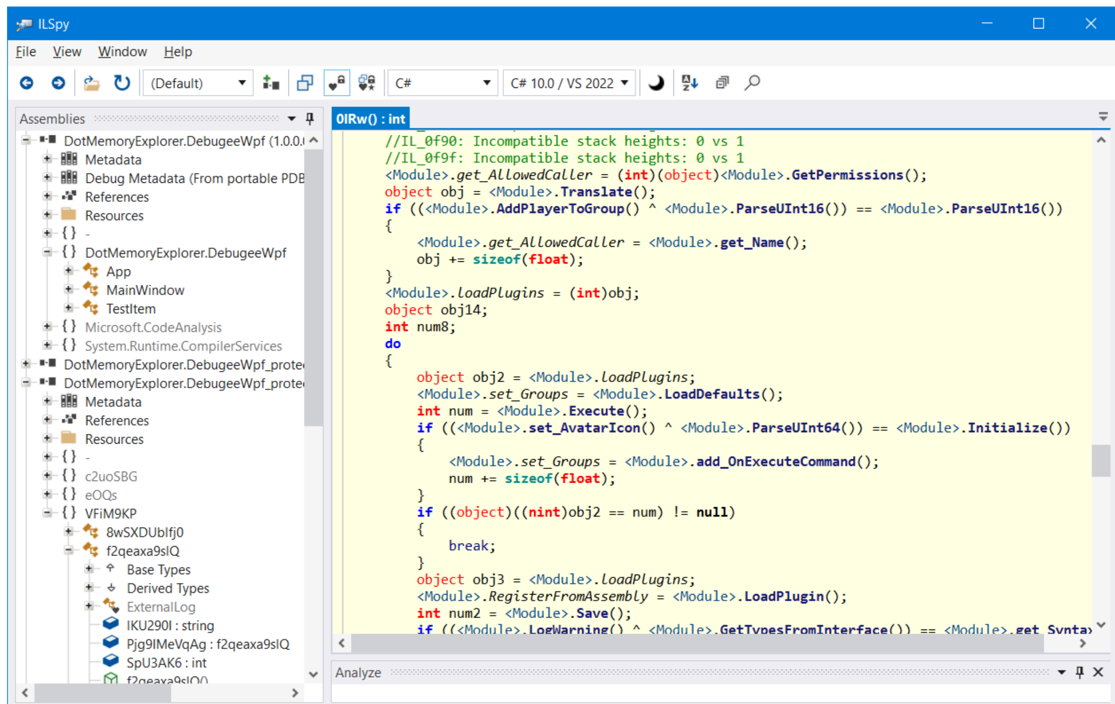


Figure 4.4 Decompiled MindLated obfuscated `GetMagicNumber` method

Need to disable even Renamer obfuscation make this obfuscator less usable because it is the flagship optimisation in .NET applications, but the implementation of GetMagicNumber is still messed with 685 decompilation warnings (which is even more than it was in the case when all obfuscations were allowed. This is most probably the result of some obfuscations that make some parts of code visible to ILSpy as dead) and working obfuscated GetMagicNumber is now decompiled to 165 lines of code. Remember, obfuscation does not cause fatal exception at startup, but instead, it results in the exception in runtime, indicating that obfuscation renamed classes related to XAML files also, and the WPF framework is now unable to find these classes and compiled resources since they have different than expected (`mainwindow`) name.

Even though Anti Debug obfuscation was enabled, it is possible to attach a debugger (Visual Studio) to the obfuscated program and make a memory dump of its memory. Anti-debug check debugger presence is most probably only at startup. TestItem class (which now has the original name) is in the memory dump visible, including unencrypted data, which indicates that even using several obfuscations did not affect the layout of the obfuscated process, and its memory content remained searchable and analysable.

This is all from the obfuscation chapter. In this chapter, the thesis describes some obfuscation techniques used for obfuscating .NET programs, lists some obfuscators, describes the impact of obfuscation on the memory footprint of the obfuscated .NET program and shows an example of one obfuscated program which will be later used in testing chapter Testing. The following section describes considerations for implementation and details about the final implementation of the .NET Memory Explorer tool.

Chapter 5

.NET Memory Explorer Implementation

This chapter thesis describes the implementation of the .NET Memory Explorer tool, which was created as part of this thesis, including describing the design decision that influenced the tool's implementation. The chapter describes possible methods for retrieving information about objects from memory and later describe the implementation of one concept in more detail. As mentioned in .NET Implementations section .NET Explorer Tool is naturally runtime implementation specific, and .NET Memory Explorer implemented as part of this thesis can scan objects in memory of processes running .NET Core runtimes.

5.1 Methods of searching objects in .NET memory

Several possible methods exist for retrieving information about .NET objects from the target process memory and their content. In this chapter, the thesis will describe and compare three possible methods. The last described was implemented in the .NET Memory Explorer tool. Its implementation is described later in this section.

5.1.1 Manual parsing

One of the most simple methods to invent is just reading memory from the target process (which is possible on all nowadays operating systems using proper system calls) and parsing all the structures manually. Since the implementation of the .NET core is open source, all structures used by runtime are available, including source codes. Even if source codes are available, it needs to consider that tools do not work against structures in source code but rather work against their compiled variant. Most C standards specify requirements on struct standard fields ordering [36], but this is not the case, for example, for bit fields (which are used in .NET Core runtime. For example in `src/coreclr/vm/field.h` struct at line 50 [16]). In this case, even with available sources layout of the internal structure is not precisely defined. In the case of implementing .NET Memory Explorer in this way, it requires an additional technique for checking the implemented layout by the compiler, for example, by decompiling code accessing this field or just experimentally watching the memory layout.

The most challenging part of the manual parsing approach is ensuring the integrity of analysed data. In the case of simple implementation working in a way that it pauses all threads, copies process memory and then resumes threads, we need to handle several possible inconsistencies. As mentioned in section , virtual instructions are after JIT compiling implemented using a different

number of native instructions. Our simple implementation needs to properly handle the situation when the program was stopped in the middle of the execution of some IL instruction. Remember, for example, the barrier mentioned in section Garbage Collection Mark Phase, which is updated after every reference update written to the object field. Suppose we stop the program after writing new references but not updating the card table (and other structures which are possibly managed by barriers). In that case, we need to handle this situation somehow. Possible handling is that we can analyse the program counter of all stopped threads and properly evaluate partially executed instructions and attempt to evaluate the state of memory in the time before instruction started (try to reverse all partially done instructions) or evaluate the impact of remaining instructions and complete partially processed instruction. Alternatively, we can choose a different approach; for example, we can ignore this situation. The case of ignoring possible inconsistent card tables is not a big issue. Since we want to dump the full heap, there is no need to process the card table for us because the card table holds information about cross-generational references, which do not impact the reachability of objects in case of dumping them all. Another more sophisticated approach is using the safe point feature of Execution Engine (EE) described in section . We can implement the usage of this feature to allow the program to complete its task and stop at a nearby safe point. Unluckily the most beneficial option is the hardest to implement. Except for inconsistencies caused by program execution, there are even more significant inconsistencies in the cases when we attempt to dump memory at the time when garbage collection is in progress. The severity of inconsistencies depends on the GC phase we stop the program in. Handling inconsistencies caused by stopping the program in the Mark phase is quite easy. The Mark phase modifies objects only a little by adding a mark (which is done by setting one bit in object content), and this bit can be easily filtered. In this case, we can handle this situation just by filtering the mark bit when processing object metadata. A much more complicated problem is with handling the integrity of the heap in case of stopping the program when running the Plan phase. As described in section Garbage Collection Plan Phase in the Plan phase, GC modifies the heap by creating plugs and gaps, overwriting parts of objects (which get very soon cleaned) by metadata about gaps and plugs, and in some cases, it even overwrites valid live objects and backups overwritten content to structures allocated internally by GC elsewhere (in completely different memory segment outside the managed heap). Handling this situation is tricky because we need to restore possibly overwritten content objects and detect which objects were processed and which were not processed yet. If in the planning phase, GC chose sweep collection, then the third phase of GC is almost without issue in terms of the inconsistency of the heap because all gap and plug information is now valid, and GC updates free_list structures, which have no impact on (live) objects on the heap. The situation is entirely different if GC chooses the compacting way. In this case, it is in Relocate phase, updating references in objects to new addresses of objects which will be moved in the compact phase. Issues are similar to those already mentioned. In the case of stopping in relocate phase, it is not trivial to determine references already updated (and now pointing to the new object position, but since the object will physically move in the next phase, they are now pointing to invalid memory) and references not updated yet. In the compact phase, all references point to addresses of objects after moving, but since objects are moving, it is unclear which objects were moved and which are pending to move.

In opposition to the complications mentioned in the previous paragraph most straightforward (but not simple to implement) approach has several significant advantages. At first, it is transparent to the process and does not require any interaction with the process. It is very hard to detect process dumping, and it is almost impossible to implement anti-debugging techniques in the process. The process can’t directly detect the external dumping but can detect it using side channel detections. Generally, this detection must tolerate program pauses because even GC similarly stops the program. Detection must accept program stops but still can, for example, make statistics of stop duration and dect atypical stop (for example, based on the duration of the program stop) caused by memory dumping instead of running GC. Another benefit of this approach implemented in a simple way is that it can be used against memory dump and not the

live process.

This approach is implemented in the SOS plugin. Since it is a debugger plugin, there is no need to explicitly copy (dump) large blocks of memory, and there is no need to handle program stopping since the underlying debugger assures these tasks. The SOS handle most possible inconsistencies in the program, but exception still exists. An example of a situation when the SOS cannot process a heap is Plan Phase as mentioned in Listing .NET objects using WinDBG and SOS section. The SOS implementation is huge, which indicates the complexity of this kind of implementation. Current SOS.dll targeting .NET Core runtime has 901 KiB, and its `.text` containing compiled code sections spans over 574 KiB of virtual address space, which indicates that code used for retrieving metadata from the .NET process is quite a large and complex thing.

5.1.2 SOS Hosting

The other option to implement .NET Memory Explorer is to use an already implemented solution - SOS. SOS is distributed as a dynamically linked library which WinDBG or LLDB debuggers can load (LLDB loads wrapped SOS). SOS and debugger are interfacing over using function calls. The SOS provides commands to the debugger, and the debugger provides functionalities to access debugged process memory.

One possible way to implement the .NET Memory Explorer tool is in a way that .NET Memory Explorer will act as a debugger and will load SOS as a plugin. In this way, .NET Memory Explorer does not need to understand almost any internal structure inside the target memory process because this stuff is handled by the SOS. In comparison with the previously described approach in Manual parsing section, it needs to interact much more with the target progress, and it needs to implement a lot of debugger features. In this case, implementation needs to not only access the memory of the remote process but, for example, need to place breakpoints in the program and properly handle them because SOS needs to use them for its operation.

In opposition, after implementing all low-level requirements of SOS implementation can benefit from comprehensive features implemented in the SOS library and process .NET heap in almost any state (except the state mentioned in section Listing .NET objects using WinDBG and SOS when even SOS is unable to process it). An additional benefit of using SOS is that SOS currently exists in several variants targetting .NET runtime. The implementation approach described in the previous section Manual parsing works against the single .NET runtime implementation, which it was developed for. Support for additional .NET runtime requires implementing the support from scratch (unless newly added .NET runtime inherits at least a subset of internal metadata structures used by previously implemented and supported runtime. This is, in fact, a case of .NET Framework and .NET Core). In the case of implementation using SOS hosting, support for additional runtime implementation can be added just by using the new SOS.dll. This, of course, works in the case where SOS.dll for target runtime implementation exists, which is still not the case of, for example, Mono runtime [37]. SOS Hosting approach shares most benefits of a fully manual processing approach, and the difference is that complexity of implementation moves from manual handling process states to the complexity of implementing a debugger.

5.1.3 GC Dumps using Diagnostic API

The last mentioned idea on dumping and parsing objects from the .NET process heap is using diagnostics API. This approach is implemented in `dotnet-gcdump` tool described in section Dotnet CLI Diagnostics Utilities. This approach utilizes EventPipe API, which is the successor of ETW API, which was previously used in .NET Framework [38] and replaces LTTng, which was used in the early days of .NET Core at Linux platform [1]. This API allows transmitting diagnostics events over platform-specific Inter-Process Communication (IPC) channels. Several .NET Core runtime parts, including GC, expose runtime events using this API. Using this API GC allows to trigger full GC collection and, as part of this collection, emits events which will enable identifying

of all objects on the heap. API works in a way that there are sources and consumers of the events. Events sourced by the application can be consumed using multiple consumers. API provides the option to filter events to only events interesting for the consumer. Filtering is done using specifying flags when connecting to the API. APi transmits formatted events in XML format, but there is currently no need to parse them manually since `Microsoft.Diagnostics.NETCore.Client` library available using Nuget package manager [39] exists. This library contains parsers for all events generated by all dotnet runtime parts.

Since this dump is driven directly by GC running in the target process, consistency of the data is assured. No matter when the client connects to the process, GC assures that it will trigger garbage collection at the right time, but the consumer (.NET Memory Explorer) needs to handle the situation where it connects at the time when another collection is already running, and it misses some events from the begging collection. In this situation, it must wait until new garbage collection is triggered by connecting to EventPipe with a flag indicating a memory dumping request was set. After garbage collection starts (which is indicated by sending event), the consumer listens to events which contain information about all objects on the heap; all detected GC roots and all detected references (edges) between the objects on the heap.

EventPipe's approach offers very simple implementation and has the benefit of providing valid data in all situations. The eventPipe approach also offers quite a simple implementation in comparison with previously mentioned approaches in sections Manual parsing and SOS Hosting. It also has some disadvantages. One of the disadvantages is that while metadata provided over this channel contains a lot of information, it is still a subset of the information which are stored inside internal structures. In case when the application is interested in some data which are not available in events passed over the channel, then it needs to read them in another way. This is a case of some type of related information needed by the .NET Memory Explorer tool, as described later in this chapter. Another security-related disadvantage is that in the case when GC start failing (for example, due to malicious activity inside the process), then it, of course, stops working, and there is no other way how to receive information about objects on the .NET process heap anymore. Similarly, since the approach internally uses IPC channels which are detectable from the process application can utilize even very simple techniques to detect memory-dumping activity. In opposition, this API is not only used for dumping memory, but a more common use case of this API is for reporting performance counters. In case when applications start actively suppressing this API, then they, of course, also stop working. While API is defined and documented, it is still implementation specific, and the only implementations supporting it are .NET Core and .NET Framework, which use it using older Windows-defined Event Tracing for Windows (ETW) technology. The benefit of this approach is that it does not act as a debugger attached to the process, which allows using the tool at the time when a different debugger is attached to the process. This makes it a tool suitable for use even if developers or reverse engineers use different debuggers attached to the process. The API does not work at the time when the program is stopped because it is handled by code running in that program, but after resuming the program, it starts working again.

After comparing the advantages and disadvantages of the three approaches described in this section, the EventPipe API approach was selected, and the .NET Memory Explorer tool created as part of this thesis is based on this technology.

5.2 .NET Memory Explorer Implementation

.NET Memory Explore is implemented in C# in Visual Studio 2022. The main project is split into two projects:

1. `DotMemoryExplorer.Core`
2. `DotMemoryExplorer.Gui`

Except for these projects, Visual Studio solution contains projects of small testing applications also written in C#, which were used for debugging purposes of the main application in the development time. The first mentioned project `DotMemoryExplorer.Core` is a class library which holds the core of the logic needed for making, managing and also analysis of .NET memory and heap dumps. The later project `DotMemoryExplorer.Gui` depends on `DotMemoryExplorer.Core` project and contains definition and logic related to Graphical User Interface (GUI). GUI was created using Windows Presentation Foundation (WPF). The first mentioned `DotMemoryExplorer.Core` project designed as a library can be used in other (for example, console) projects used for user-specific tasks and analysis. GUI is the front end for the core library. GUI adds to the tool several features which make this tool suitable for reverse engineering. It especially supports temporary renaming of the identifiers and support for searching objects by data. The first mentioned is used when reversing obfuscated applications with symbol renaming obfuscation applied and enabling the reverse engineer to add custom labels to data type names and field names for each class. These renaming mappings are exportable and importable to the text file, which later allows them to reuse them between sessions as well as simplifies the manual movement of translations between different tools. The other mentioned reverse engineering-related tool is support for searching objects by their content. All tools mentioned in section Tools allow going from objects to their content, but in case of reverse engineering, we frequently do not know what data are managed by what class, but instead, we know data (for example, data which we entered) and want to learn which class (and it's related) code is responsible for their management.

In the next section, the most important part and classes of the .NET Memory Explore implementation are described.

5.2.1 .NET Diagnostic Interface Client

The most important part in terms of this thesis is class `EtwHeapDumper` implements the base for connecting to the remote process and triggering garbage collection while listening for events related to objects, types and references (edges) from the remote process. The class have two public methods:

1. `MakeDump`: These functions trigger the whole process of making a heap dump.
2. `Dispose`: This method cleans unmanaged resources after the heap dump is completed. The class should be used in C# `using` block for automatic calling `Dispose` method.

Class is designed for single making single dump. Attempting to make a second dump using a single instance will result in `InvalidOperationException`.

In the whole class (and the program in general), two terms related to dumps are distinguished.

1. **Memory Dump**: This is a dump of raw memory without any metadata. It contains byte arrays of all private memory mapped to the process.
2. **Heap Dump**: This is a dump of metadata about objects in the .NET Heap but does not contain the content of any object. The content of the object has to be retrieved from the related Memory Dump, which is collected `EtwHeapDumper` right after starting the dumping process.

The dumping process consists of five steps which are executed in the following order.

1. Creating Memory Dump
2. Preparing EventPipe client
3. Processing events in the background thread

4. Waiting for garbage collection to complete

5. Generating a Heap map based on collected events

The memory dump collection is managed by class `WindowsProcessMemoryManger`, which using Windows kernel32.dll function `VirtualQueryEx`, enumerates memory regions of pages allocated in the target process. After listing all memory pages, private committed pages are collected and loaded using kernel32.dll function `ReadProcessMemory`.

EventClient preparation is done in `SetupClient` private method. This method instantiates `DiagnosticsClient`, `EventPipeSession`, and `EventPipeEventSource` classes from the `Microsoft.Diagnostics.NETCore.Client` library. The first two mentioned are used for establishing the EventPipe channel, and the third is used for parsing events from this channel. After instating classes, the method attaches events to an instance of `EventPipeEventSource` class, generating an event when a specific event is received from the EventPipe channel. The following five events are attached:

1. **GCStart**: This event is used to determine that garbage collection started. This event is used to identify that full GC triggered by connecting to EventPipe with `GCHeapSnapshot` keyword (flag) started. Unless this event is received, all other events are ignored.

2. **GCStop**: This event is used to determine that garbage collection is completed and there will be no more events related to monitored GC collection.

3. **GCBulkNode**: This event holds information about objects detected on the .NET process heap. Information are sent in bulk, and one event contains information about multiple objects.

4. **TypeBulkType**: This event holds information about binding data type names to data type id, which is used for identifying data types of objects received using `GCBulkNode` event.

5. **GCBulkEdge** This even holds information about objects referenced by objects which were received in `GCBulkNode` event.

The reception of events is asynchronous, and there is a circular buffer of default size 256 MiB used. Since the memory for storing events is limited, there is pressure to process events as fast as possible. For this reason, processing of `GCBulkNode`, `TypeBulkType`, and `GCBulkEdge` is implemented in a way that event handlers just clone and save the received item. Saved items are later processed in the last phase.

In the last phase, after `GCStop` event is received, the saved information about objects, data type names and references are processed, and a heap dump is created. For building the final heap dump, there is `EtwHeapBuilder` class which processes data collected by `EtwHeapDumper` class. This class unpacks received bulk events and process data for creating instances of `DotnetObjectMetadata`, `DotnetReferenceMetadata`, and `DotnetTypeMetadata` structs, which hold enhanced information from the original events and are later used inside the whole application. The final heap dump is represented using `HeapDump`, which except references to lists of `DotnetObjectMetadata`, and `DotnetTypeMetadata` classes contains information about the timing of heap dump creating and basic statistics which are later shown in tool GUI.

5.2.2 .NET Type Information

While metadata received from EventPipe contains a basic mapping of type id to data type name, they do not contain any other information about the data type of the object, for example, information about object field locations. As mentioned in section GC Dumps using Diagnostic API, the API is statically defined, and there is no other way to get any additional data except information sent. In this case, implementation needs to manually get this information elsewhere

and preferably in the memory of the target process (the other option is to examine loaded process modules and try to find the required information in binaries on the disk). In this case, loading type information from process memory does not cause any trouble with consistency because most parts of type information are embedded directly in module images, and they are at static positions in address space, do not change and do never relocate. For supplying information which is not provided using EventPipe API, implementation contains `ObjectDetailProvider` class. This class instantiated per object can retrieve additional information about the object and its data type directly from the process memory (and partially from the memory dump).

The structures which are needed to reach basic data type information, like locations where fields of the class are located in their content, are publicly described by several sources like [1] and [40]. Since .NET Core runtime source codes are open source, we can also see the headers containing these structures directly. The most important structures containing information about data types used in runtime are `MethodTable` and `EEClass`. Both are designed for the same purpose but were split to separate classes for making `MethodTable` memory footprint as small as possible for better utilization of cache lines [40]. The remaining less frequently used data are stored in `EEClass`.

5.2.3 MethodTable

The Method Table is the most fundamental class containing basic information about the data type. It includes, for example, the size of the object. As mentioned in section Object Layout and Metadata, every object contains at the beginning pointer to MethodTable. MethodTable class is defined in `dotnet/runtime` project available at GitHub in file `src/coreclr/vm/method table.h`. It contains 12 fields, but its definition spans over 3254 lines of code, starting at line 516. The first field is declared on line 3556. MethodTable directly does not contain any information about fields or class hierarchy but has object size in `m_BaseSize` field, a pointer to parent MethodTable, which is used in case of inheritance (note that all objects in .NET inherit from `System.Object`) and a pointer to EEClass.

The Type Id, which contains every object received from the Event Pipe, is, in fact a pointer to the Method Table. Another way to get a pointer to the method table is to dereference the object address because the pointer to the method table is stored at the beginning of every object. `ObjectDetailProvider` class do not have any method for getting the pointer MethodTable because it is just TypeId which is already present in `DotnetObjectMetadata`. `ObjectDetailProvider` offers a way for getting a pointer to the parent MethodTable and a pointer to EEClass. Both are implemented in a way that takes the method table address, adds an offset to the pointer field and dereferences it. In the case of `GetEEClassAddress`, the address to EEClass is in MethodTable after 2 DWORD, 4 WORD and 3 pointer fields. WORD and DWORD values make a 16-byte offset, and 3-pointers make a 12 or 24-byte offset depending on the target platform. This offset is added to the method table base and dereference, which results in getting the address of the EEClass structure. Pointer to parent MethodTable is right after 2 DWORD and 4 WORD fields.

5.2.4 EEClass

EEClass contains much more information about data type, including a pointer to a list of fields which the class has. It is a structure defined in file `src/coreclr/vm/class.h` in `dotnet/runtime` at the GitHub. The main EEClass definition starts at line 710 and spans over 1178 lines of code. The list of fields which .NET Memory Explorer needs is stored in an array elsewhere. EEClass contains a pointer to this array. The pointer to this array follows three other pointers in EEClass. It is an array containing instances of FieldDesc class defined in file `src/coreclr/vm/field.h` in the same project as EEClass. It is a simple structure containing a pointer to the method table of the class woning the field and some data about the field, including the offset of its data in the object.

Class Hierarchy showing bindings between MethodTable, EEClass and an array of FieldDesc is shown in the following figure 5.1. Offsets are expecting running on a 64-bit platform. In the case of a 32-bit platform, offsets are slightly different (and since structures contain non-pointer fields, they are not half values).

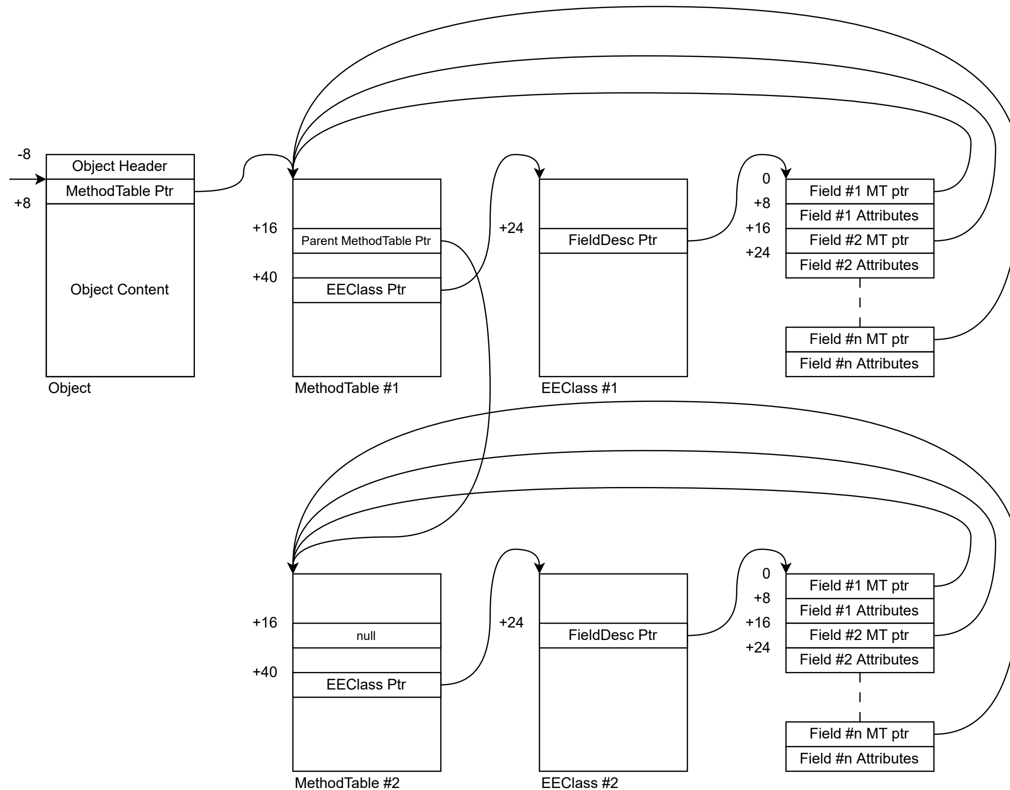


Figure 5.1 The class hierarchy of .NET runtime structures describing data types

Note that there is no direct encoded number of fields in the array. This can be determined from packed fields inside EEClass, but implementation in .NET Memory Explorer differs. It instead checks pointers to MethodTable, and unless the array contains a valid Method table for the same owning class, it supposes FieldDesc. After the first occurrence of the wrong MethodTable pointer implementation, suppose that the end of the array is reached.

The only missing information is the name of the field. Unluckily MethodTable, EEClass, and FieldDesc have no name string (or pointer to a string) encoded inside except in cases when .NET runtime is compiled in debug mode (which is not the case of normally used .NET runtimes which are always release build). For accessing field name, there is quite a complicated way to get metadata from module information which are encoded in a binary file (which is loaded to the process memory loaded also), but their access is quite complicated and proper implementation require travels over several classes and implementing own enumeration of .NET assembly metadata and searching for the entry with the same field token as it is the token in the EEClass. The current proof of concept created as part of this thesis is not implemented, and the current .NET Memory Explorer just does not show field names.

Chapter 6
Testing

This section of the thesis describes the testing process of the .NET Memory Explorer tool, whose implementation was described in the previous chapter .NET Memory Explorer Implementation. The .NET Memory Explorer tool created as part of this thesis was tested against three types of programs which are later described in this section, and example usages of .NET Memory Explorer against these programs are shown. The program was tested again:

1. The custom simple testing program
2. The Paint.NET is an example of a real-life .NET program
3. An obfuscated version of a simple program

6.1 Testing program

For this thesis, a simple testing program was developed. It is avalaible in the `DotMemoryExplorer.DebugeeWpf` project, which is assigned to the same Visual Studio solution as the main projects of .NET Memory Explorer. It is a simple program with GUI developed using WPF, and its GUI is shown in the screenshot in the following figure 6.1.

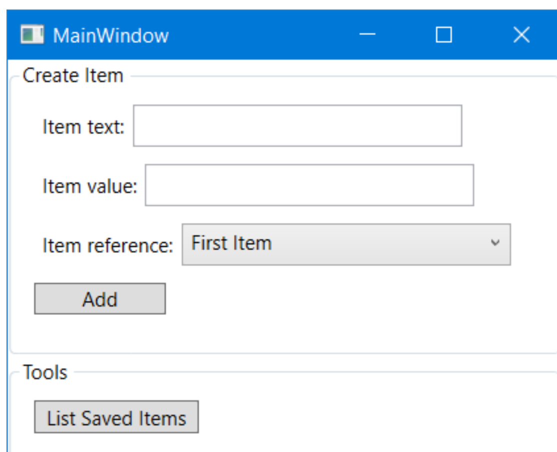


Figure 6.1 The simple testing program used for testing .NET Memory Explorer

The program has the internal object of class `TestItem`. Its source code was shown in chapter Tools at listing 2.1. This class allow storing the information entered by the user. One of them is an integer field, the second file is a string field, and the third is a reference to some other item or null. In the GUI user can directly enter specified integer values and text, which will be stored in the object. The reference field is filled by the program, which gives the user for options: load fill with a null value, load field with reference to the item which was first added by the user, item which was added as second and reference to the last item added. Items are added to an instance of `List\<TestItem\>` class, which is stored in an instance of `MainWindow` class. The GUI program can print information stored in the objects after clicking the button. This is shown in the following figure 6.2.

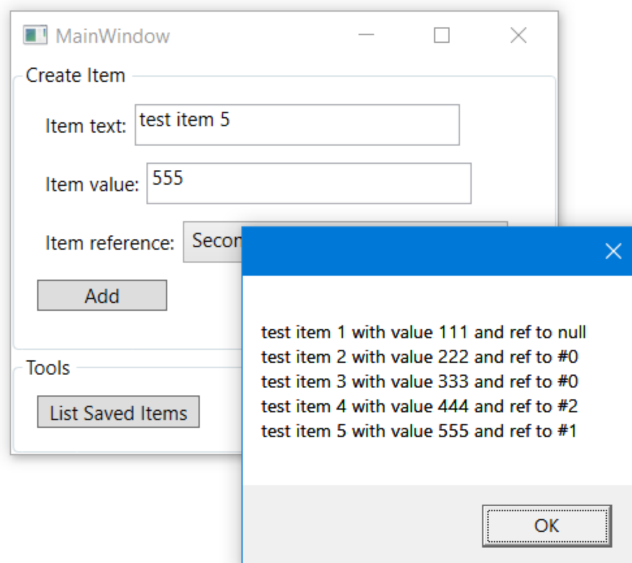


Figure 6.2 Listing of saved objects in testing program

6.2 Browsing objects

The most natural feature of .NET Memory Explorer is to browse objects in the process memory. After attaching to the testing program, the .NET Explorer lists all regions of pages allocated in the target process memory. Some interesting regions from this listing are later dumped. The program support generating and making multiple dumps, but in comparison with other tools like Visual Studio and Perfiew, it currently does not support for comparison of dumps, but the user can manually check for differences in monitored objects. After creating dumps by clicking the `Dump Heap` button on the process tab tool connects to the diagnostics API of the remote project and starts dumping its memory. After the dump is complete, the program lists basic statistics about dumped memory, like object count and the number of references between objects as well as timing information about dump generation. In the case of a simple testing program, we dump is done almost immediately and contains over 30000 objects which can be quite interesting the first time, but it is mostly caused by internal objects of .NET runtime, and also, there are a lot of objects related to WPF GUI. The Heap Dump view showing this information is shown in the following screenshot 6.3.

The main part of the window is occupied by a listing of data types with information on how many instances of that data type exist in the program memory on the heap. Since we know

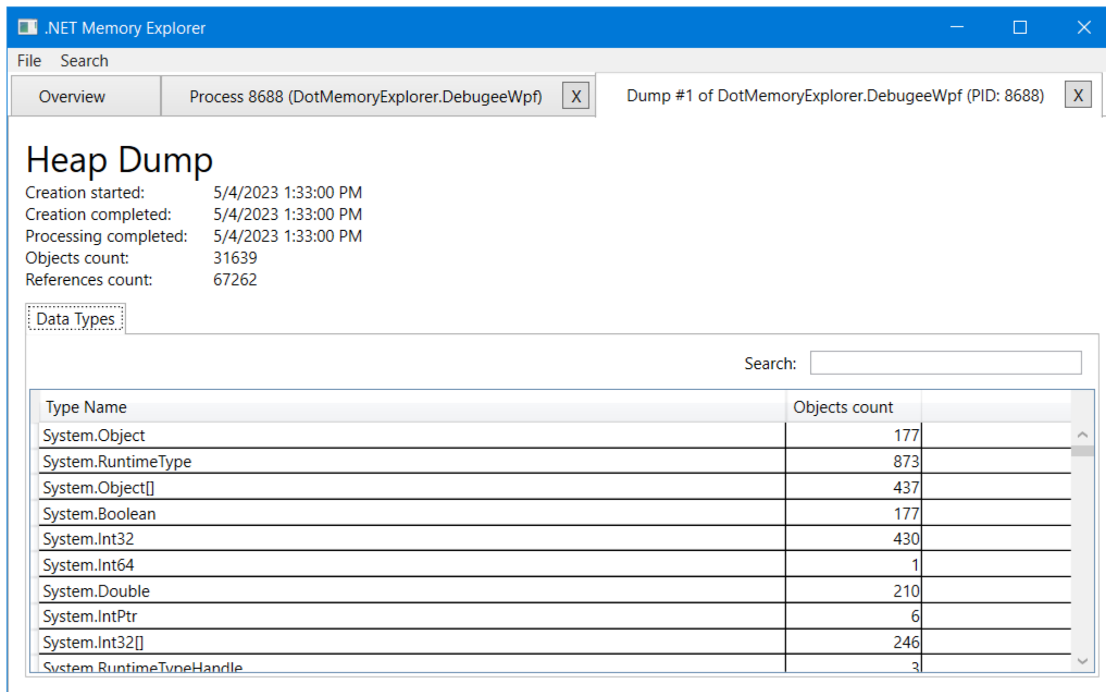


Figure 6.3 Heap Dump Details in .NET Memory Explorer

the structure of this program, we can search for the most interesting type in the application: `TestItem`. Search will filter listing to four data types which contain searched term in the name as shown in the following figure 6.4.

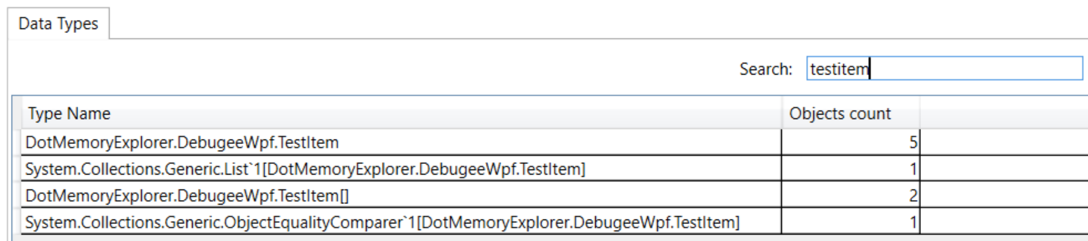


Figure 6.4 `TestItem` classes listing in .NET Memory Explorer

Now we can see that five instances of `TestItem` class are allocated in the program. Except for the main classes, we see that the program has allocated two instances of arrays of this class, one `List\<TestItem\>` and one `ObjectEqualityComparer\<TestItem\>`. After double-clicking the data type, we can see a listing of instances of that data type. In the case of TestItem, we can see the memory address of all five instances of objects. Listing is shown in the following figure 6.5. Except, for instance, address and data type information, we see the Label field. Labelling is a feature of the program which allows users to add custom text labels to instances, data types and object fields. Program in all places where the labelled object (or type or field) occurs sho this label instead of the original information (which in the case of obfuscated programs are usually meaningless).

After double-clicking the instance, its detail tab is opened. Detail is shown in the following figure 6.6.

Object detail shows basic information about the object, internal information and addresses.

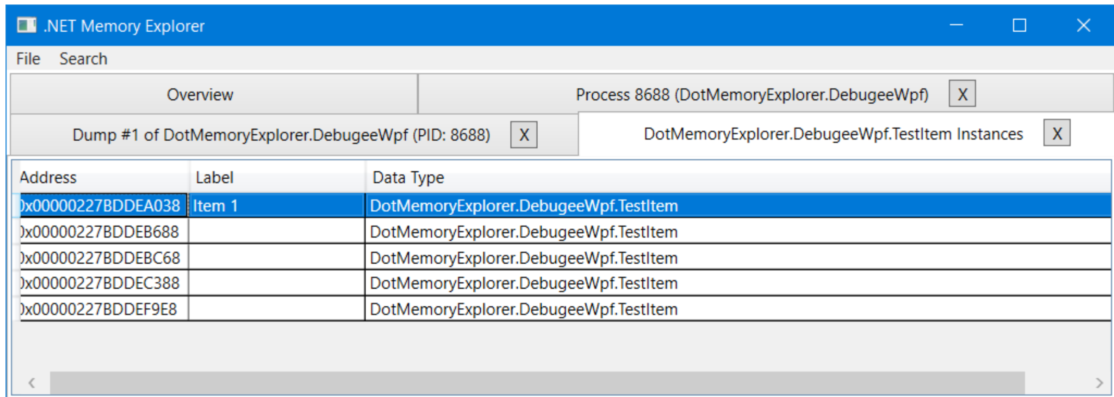


Figure 6.5 `TestItem` instances listing in .NET Memory Explorer

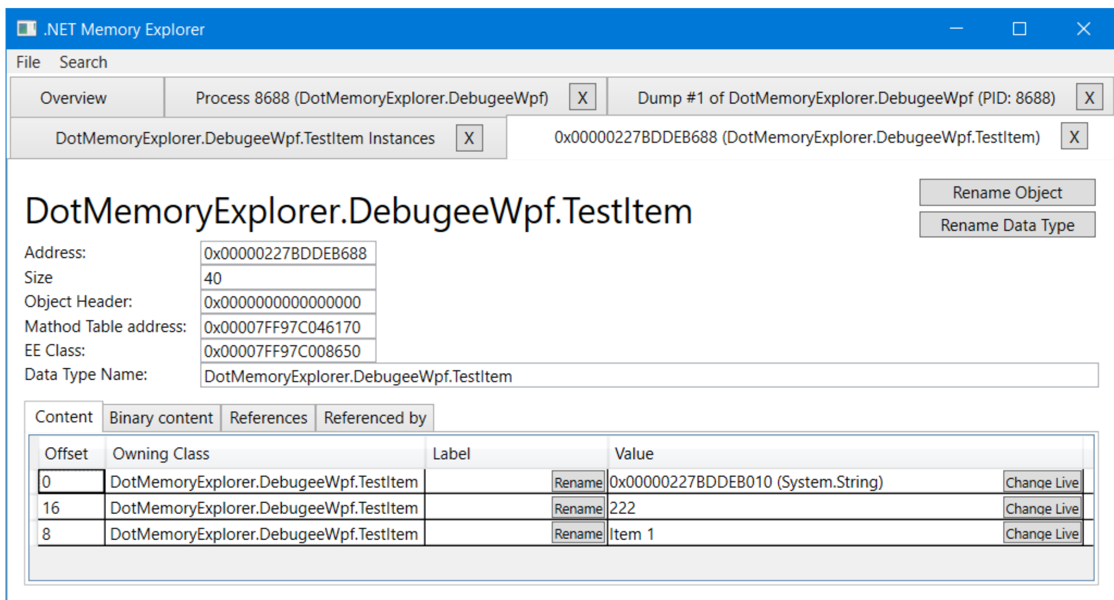


Figure 6.6 Detail of selected `TestItem` instance

The meaning of the fields described in the upper half of the pane is the following:

1. **Address**: Memory address where the object is located in the memory. Note that as described in section Object Layout and Metadata, this is the address which does not point to ObjectHeader, but it points to the MethodTable field of the object.

2. **Size**: Size of the object as allocated on the heap. This includes the size of tailing padding, which is added for making the heap address aligned to 8 bytes on 64-bit systems.

3. **Object Header**: This is the value of the Object Header field, which is located before the object in the memory. Its structure is complicated, and most bits are overloaded and, in different contexts, have different meanings. For most users value of this field is not important and can be ignored.

4. **Method Table address**: Contains address to method table, which holds basic information about the data type of the object. For most users, this field is also not very relevant but can

be used for identifying types with obfuscated names, and the value of this field can be reused in other debugging tools like WinDBG with SOS plugin.

5. **EE Class**: This field contains the address of the EE Class related to the data type of the object. Similarly to the method table, it can be used for identifying data types with obfuscating names and addresses and can be reused for use with other tools like WInDBG and SOS.
6. **Data Type Name**: Contains the name of the data type or label if the user renames it.

.

The tool can parse object content which is the default view. Alternatively, the tool can show the raw content of the object memory. Binary dumping starts at the pointer object, which means that pointer to MethodTable is included, but the Object Header field is not. The binary dump of object content of selected TestItem instance is shown in the following figure 6.7.

Content | Binary content | References | Referenced by

```
0000    70 61 04 7C F9 7F 00 00  10 B0 DE BD 27 02 00 00    pa.|ù....°Þ½'...
0010    38 A0 DE BD 27 02 00 00  DE 00 00 00 00 00 00 00    8.Þ½'...Þ.......
0020    00 00 00 00 00 00 00 00                             ........
```

Figure 6.7 Binary content of selected `TestItem` instance

At the main `Content` tab, object fields are parsed and listed. In the case of referencing, a labelled object label is used in the value field instead of object address and data type. This is visible in the previous figure 6.6. Shown instance contains in the third field (which is memory located as a second field) reference to an instance of other TestItem classes, which were labelled by label `Item 1`. Labelling of objects is possible using `Rename Object` button in the right corner of the object detail view. Similarly, the user can rename the data type of the object here. The labelling object is useful for reverse engineers who want to add some notes to specific instances of the analysed objects. He or she can, for example, remember some of his/her notes determined after analysis of the object content. Labelling objects later simplify searching for that instance and also allows easier analysis of the object referencing the labelled object. Renaming data types is useful in the case of data types with obfuscated names.

The references tab of object detail contains objects referenced by this object as indicated in metadata provided by the garbage collector over the diagnostics channel. In most cases, the listing contains the same references as listed on the main tab. Differences occur in the case of the arrays, which do not have a properly defined field for every item in the array, but in the References tab, all referenced objects in the array are listed.

Referenced by contains a listing of objects which contains a reference to analysed objects. In the case of shown test item, there are two references, as shown in the following screenshot in figure 6.8.

Content | Binary content | References | Referenced by

Address	Label	Data Type
0x00000227BDDEF9E8		DotMemoryExplorer.DebugeeWpf.TestItem
0x00000227BDDEFA10		DotMemoryExplorer.DebugeeWpf.TestItem[]

Figure 6.8 Object referencing selected `TestItem` instance

One of the references is caused by references created by the program. It is a reference from different TestItem instances. Slightly unexpected is reference caused by array because, as we know, the source code of debuggee program, the instances are not stored in array but in List.

The reason for having an array instead of a List is that List internally stores objects in the array, and the array we see is the internal array of the List, which we can easily check. After double-clicking the reference of the array, its detail is opened. The tool indicates empty content of the object, but pointers to objects are visible in the binary content of the object, and references to TestItems stored in the array are visible at the References tab, including the item with an address ending `688`, which we came from (it's tab is still open and user can switch between views at any time). References of the array are shown in the following figure 6.9.

Content | Binary content | References | Referenced by

Address	Label	Data Type
0x00000227BDDEA038	Item 1	DotMemoryExplorer.DebugeeWpf.TestItem
0x00000227BDDEB688		DotMemoryExplorer.DebugeeWpf.TestItem
0x00000227BDDEBC68		DotMemoryExplorer.DebugeeWpf.TestItem
0x00000227BDDEC388		DotMemoryExplorer.DebugeeWpf.TestItem
0x00000227BDDEF9E8		DotMemoryExplorer.DebugeeWpf.TestItem

Figure 6.9 Objects referenced by the array of `TestItem`

We can see that array contains all five instances referenced in this array, including the already labelled item. Finally, we can see an object referencing this array which displays the only referencing object of the List<Testitem> data type, which confirms the theory with an internal array of the list.

Content | Binary content | References | Referenced by

Address	Label	Data Type
0x00000227BDC68DA0		System.Collections.Generic.List`1[DotMemoryExplorer.DebugeeWpf.TestItem]

Figure 6.10 Objects referencing array of `TestItem`

The same we can do for finding objects referencing the list, and we can see that the only object referencing this list is an instance of MainWindow class which is expected. In the case of MainWindow, we will see a lot of objects from WPF referencing MainWindow. The only non-WPF objects referencing MainWindow are an array of systems.Object and App class which is defined in the testing program and is responsible for starting the application and creating the window.

6.3 Modifying objects

The tool supports modifying field values. It supports the most common field values and supports changing references. Note that some time assignment fails. This can happen when the object is relocated by some garbage collection triggered after the memory dump heap is collected. The object also may get collected in the meantime. In this case, the tool tries to detect and attempt to value modification fails.

Modification of value is triggered by clicking the `Change Live` button of the target field on the `Content` tab in the object detail view. After clicking, one of two possible dialogue windows is opened. For editing value types like int, char, bool or other windows with textbox is opened. The entered value is later parsed and written to the object body in the target process memory. In case of an editing field containing reference to some object, a window containing a list of instances is shown, and the user can be selected.

For testing, a test program was configured with objects as shown in figure 6.2. Using the .NET Memory Explorer tool reference in the last object was set to the third object (second if counted from zero). In normal cases, this should never happen because the tool, when creating

objects, allows selecting references to the first, second and last create (which is fourth in the case of creating the fifth instance). But after modifying the reference internally using the tool, it is possible to assign a reference to any object. After modifying, the object listing was triggered, and the program indicated that the fifth object referenced the third one (second when counted from zero). Also, the program does not allow modifying value after an item is added, but in case of external modifications using .NET Memory Explorer, this is possible. For test value of the first object was changed from 111 to -99999. After modifying objects, the object listing in the program was triggered with output containing reflected reference and value changes as shown in the following figure 6.11.

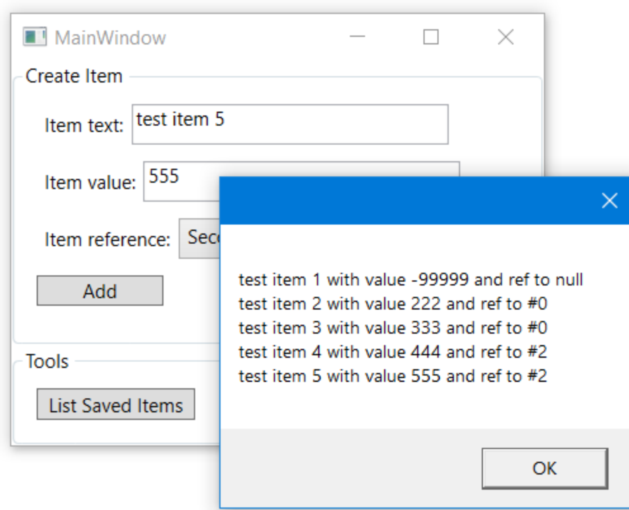


Figure 6.11 Externaly modfied value and reference in testing program

6.4 paint.net

The second program which was used for testing was used paint.net tool, which is a competitor of the Paint program available in Windows. It offers more features. The program is currently available for free when downloaded from the official website, and the program is paid when bought using Windows Store. The program is currently freeware, but its sources are not available. .NET Memory Explorer can be helpful when reversing software to find places containing known data processed by the program or, for example, for searching sensitive artefacts (for example, encryption keys) in the program memory. These searches can be done using arbitrary debuggers like x64dbg and WinDBG, but .NET Memory Explorer can map found occurrences to objects if the searched data belong to some.

For testing the image with some lines, one text and one imported image (an image from the previous section of this thesis) were created. The scene created in Paint.NET is highlighted in the following figure 6.12.

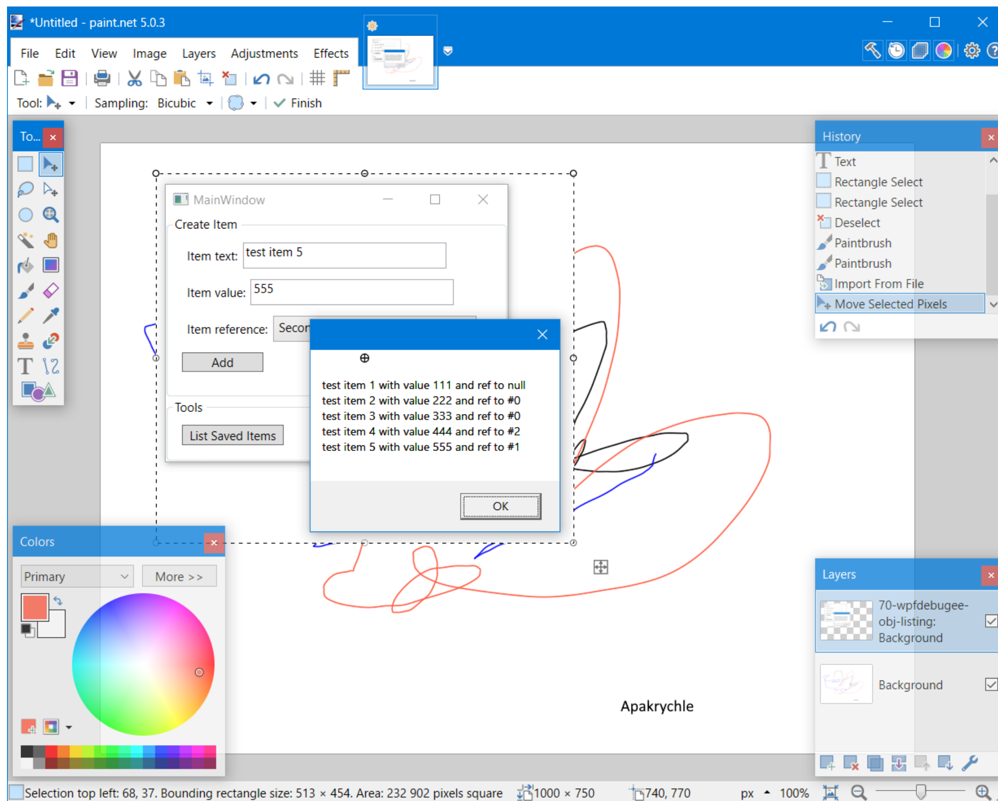


Figure 6.12 paint.net and testing scene

After attaching .NET Memory Explorer to process, the statistics and data types used in the program are listed. The reverse engineer may quickly see what types of resources are in the analysed program. In the case of paint.net, we can see, for example, references for data types indicating the use of Direct2D API as shown in the following figure 6.13.

Data Types

Search:

Type Name	Objects count
Internal.PaintDotNet_Direct2D1_IDeviceContext_Factory	1
Internal.PaintDotNet_Direct2D1_IDeviceEffect_Factory	1
Internal.PaintDotNet_Direct2D1_IDeviceEffectContext_Factory	1
Internal.PaintDotNet_Direct2D1_IDeviceImage_Factory	1
Internal.PaintDotNet_Direct2D1_IDirect2DFactory_Factory	1
Internal.PaintDotNet_Direct2D1_IDrawInfo_Factory	1

Figure 6.13 Direct2D related Data Types in paint.net

After scrolling to the bottom, we most probably notice many data types in PaintNET namespace, as shown in the following figure 6.14, which indicates that the program is most probably not obfuscated or is obfuscated by obfuscations which do not rename symbols.

Data Types

Search:

Type Name	Objects count
PaintDotNet.Shapes.Arrows.PentagonArrowShape	1
PaintDotNet.Shapes.Basic.DiamondShape	1
PaintDotNet.Shapes.Basic.EllipseShape	1
PaintDotNet.Shapes.Basic.ParallelogramShape	1
PaintDotNet.Shapes.Basic.RectangleShape	1
PaintDotNet.Shapes.Basic.RightTriangleShape	1

Figure 6.14 PaintDotNet related Data Types in paint.net

6.5 Discovering Classes by Data

Except direct method described in the previous section, the tool support for finding objects by the data they hold. For this use case, tools offer two search tools: string search and binary search. String search support searching for UTF-8, UTF-16 and ASCII encoded strings. Since all strings in .NET are encoded using UTF-16 other two will very rarely find any string except strings in unmanaged memory and in arrays containing strings encoded to different coding. This knowledge can be, of course, used by reverse engineering since knowing storage for encoded data can be, in some cases, useful too. The following figure 6.15 shows the output from string `Apakrychle`, which was added and drawn on the paint.net scene.

.NET Memory Explorer

File Search

Overview | Process 23700 (paintdotnet) X

Dump #1 of paintdotnet (PID: 23700) X | 'Apakrychle' (len 10) Search Results X

Occurence Address	Object
0x0000018875790F40	Occurence is in non-heap segment or containing object is collected
0x0000018875791C90	Occurence is in non-heap segment or containing object is collected

Figure 6.15 Search string in paint.net memory

Results indicate that there are two occurrences in the process of memory found, but there is no living object exists. This is because paint.net, after completion of typing, immediately transforms text to bitmap. Two occurrences shown here are most probably addresses where the string object was located before it got collected.

The second test was with searching for PNG ASCII text. PNG is a string which is used in PNG images in the header for identifying this header. Since it was searched as an ASCII string tool did not find any System.String reference, but another tool for searching in the memory is the binary search tool. This search was used to find three instances, as shown in the following figure 6.16.

Two occurrences were in arrays of bytes, and one in an array of DecoderPattern instances. After searching objects referencing these objects, we can deduce that one occurrence is related to some icons which were loaded from objects referencing the objects that are the usage of the PNG header come from icons which were loaded and are used by `PaintDotNet.Dialogues.MainForm` class. The second occurrence is after analysis of referencing object related to `PhotoSauce.MagicScaler.DecoderInfo` class, which is referenced by an array of `PhotoSauce.MagicScaler.IImageCodecInfo`, which indicates that it is not related to any real PNG image, but instead, this header is used when parsing

Overview | Process 23700 (paintdotnet) X | Dump #1 of paintdotnet (PID: 23700) X | 'PNG' (len 3) Search Results X

Occurence Address	Object
0x00000147CB2DB281	Occurence is in non-heap segment or containing object is collected
0x00000147CB2E8371	Occurence is in non-heap segment or containing object is collected
0x00000147CDE04E3F	0x00000147CDE04E3F (System.Byte[])
0x00000147CDE74159	0x00000147CDE74159 (System.Byte[])
0x00000147CDE77599	0x00000147CDE77599 (DecoderPattern[])
0x00000188601E85F1	Occurence is in non-heap segment or containing object is collected
0x00000188601EE821	Occurence is in non-heap segment or containing object is collected

Figure 6.16 `PNG` string search in paint.net memory

PNG images in codec implementation. The last search result entry points to the codec info also.

Another test was done using searching the binary of the image imported to the scene. Random six consecutive bytes from the image used were searched in the memory, but there were no occurrences of that pattern found in the program, which can indicate that the program does not originally store it in the memory and free it after loading and converting to bitmap.

But part of the original file name searched as UTF-16 string was found in the process memory (note that this string is visible in program GUI in layers window). It is referenced by `LayerProperties`, whose content is shown in the following figure 6.17.

LayerProperties

Rename Object | Rename Data Type

Address:	0x00000147D0909550
Size	40
Object Header:	0x00007FF960CC1E08
Mathod Table address:	0x00007FF961A2C3B8
EE Class:	0x00007FF961A0F5B0
Data Type Name:	LayerProperties

Content | Binary content | References | Referenced by

Offset	Owning Class	Label		Value	
0	LayerProperties		Rename	layernamefilename	Change Live
8	LayerProperties		Rename	0x00000147D09095C8 (PaintDotNet.Metadata)	Change Live
20	LayerProperties		Rename	True	Change Live
21	LayerProperties		Rename	False	Change Live
22	LayerProperties		Rename	255	Change Live
16	LayerProperties		Rename	0	Change Live

Figure 6.17 LayerProperties object in paint.net memory

This object reference to Metadata, and this contains a List of instances of ExifPropertyItem, which can, for example, hold information for reverse engineers.

6.6 Obfsucated application

After testing the program by searching for commonly used programs, tests against obfuscated programs were done. For testing obfuscation, the Mindlated tool is described in section MindLated. This tool has issues with obfuscating symbols in WPF GUI programs as described in Obfuscator demonstration section, but with the console program, it works well. For this reason, the console variant of that debuggee was developed and obfuscated with Control Flow, Int Confusion, Arithmetic, Reneamer, and JumpCflow obfuscations enabled.

.NET Memory Explorer was attached to this obfuscated process as shown in the following figure 6.18.

Console application use in the memory much less object than WPF variant of this debuggee. At this time, there are only 361 objects in the memory, and it contains 536 references in total. In the case of an obfuscated program, we do not see class names anymore; instead, there are

Type Name	Objects count
System.WeakReference`1[System.Diagnostics.Tracing.EventSource][]	1
System.Diagnostics.Tracing.NativeRuntimeEventSource	1
0F4.XNJbDVN73J	2
INPUT_RECORD	1
System.Reflection.RuntimeAssembly	1

Figure 6.18 LayerProperties object in paint.net memory

identifiers like `0F4.XNJbDVN73J`, which are replacements for original identifiers. Note that some obfuscators use replacement identifiers containing only white characters, but Mindlated uses randomly generated strings of letters and numbers.

Search for the string entered in the program directly finds the string and character array instances which are shown in the following figure 6.19.

Dump #1 of DotMemoryExplorer.DebugeeConsole (PID: 5188) X | 'test' (len 4) Search Results X

Occurence Address	Object
0x0000020E8901430C	0x0000020E8901430C (System.String)
0x0000020E89016B50	0x0000020E89016B50 (System.Char[])
0x0000020E890171A4	Occurence is in non-heap segment or containing object is collected

Figure 6.19 Search for data in obfuscated program memory

After analysis of this object, we can see that it is a standard string object with a content of `test string 1`. In obfuscated applications, it is a good idea to label the analysed objects. For this reason found string was labelled as `test string`, and in all further screenshots, it will be referenced under this name instead of the address. It is referenced by an instance of type `0F4.XNJbDVN73J`, which we now know most probably represents the item because it holds a reference to the string which we entered in the application. This is confirmed by opening its detail. Except for the reference to a known string, it contains a known value which was entered into the application. Class detail is shown in the following screenshot in figure 6.20.

test string

Rename Object | Rename Data Type

Address:	0x0000020E89014300
Size	48
Object Header:	0x00007FF8BD78FD10
Mathod Table address:	0x00007FF8BD78FD10
EE Class:	0x00007FF8BD77A710
Data Type Name:	System.String

Content | Binary content | References | Referenced by

Address	Label	Data Type
0x0000020E89016408		0F4.XNJbDVN73J

Figure 6.20 Search for data in obfuscated program memory

Exactly in the case of analysed string instance, it is good to place labels. Data type `0F4.XNJbDVN73J` was labelled as `ItemClass` and analysed object as `Item 1`. This time the analysed first item is not referenced by any array or List like it was in the case of the non-obfuscated version, but this time, it is referenced only by a different item as shown in the following figure 6.21.

After opening the item which references the specified item, we can see that the second item

Content | Binary content | References | Referenced by

Address	Label	Data Type
0x0000020E89016470		ItemClass

Figure 6.21 Search for data in obfuscated program memory

was created in the program, and it is not referenced by any other object. This time it is because items are not added to the list, but instead, they are forming manually created linked-list. The second item is not referenced anywhere because it is the first item in the list, and since it is located only on the process stack, there are no references from other objects to it. But its address can be searched using the binary search tool. The program has three occurrences, as shown in the following figure 6.22.

.NET Memory Explorer

File Search

Overview | Dump #1 of DotMemoryExplorer.DebugeeConsole (PID: 5188) X | 'test' (len 4) Search Results X

test string X | Item 1 X | Item 2 X | '70 64 01 ... 02 00 00' Binary Search X

Occurence Address	Object
0x0000007FC4D7C1F0	Occurence is in non-heap segment or containing object is collected
0x0000007FC4D7E4D0	Occurence is in non-heap segment or containing object is collected
0x0000024F1CE22126	Occurence is in non-heap segment or containing object is collected

Figure 6.22 Search for object address in obfuscated program memory

One of the occurrences is most probably located on a stack of the executing program. For determining this address can e searched and analysed in other tools like WinDBG with SOS or dnSpy.

Chapter 7

Conclusion

The outcome of the thesis is a .NET Memory Explorer tool with a graphical user interface for analysing memory which allows browsing and searching in the memory of the .NET process and shows information about objects located in the process memory. The created tool communicates with the diagnostics API of the target process, processes events generated by the process and access process memory for information not present in the events. Tools contain a renaming feature, which simplifies working with obfuscated applications.

At development time, analysis of the garbage collector and other .NET runtime parts behaviour was done, resulting in an enumeration of techniques which can be implemented for accessing information and structures related to objects in the process memory. This analysis's results were used to decide the approach used in the .NET Memory Explorer implementation.

The tool was tested against elementary, real-world and obfuscated applications. Analysis of available obfuscators for the .NET was done, and one obfuscator was shown in detail with examples of its operation and analysis of their impact on the program. The thesis shows the approach of analysing the program's memory with and without source code.

The .NET Memory Explorer tool is shown in the following figure 7.1.

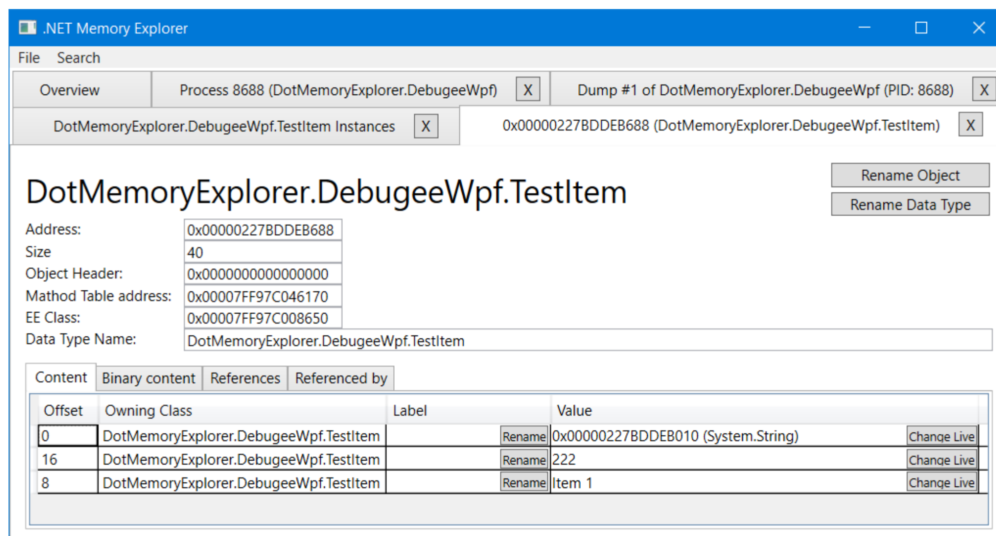


Figure 7.1 .NET Memory Explorer

Bibliography

1. KOKOSA, Konrad. *Pro .NET Memory Management*. Warsaw, Poland: Apress Berkeley, CA, 2018. ISBN 978-1-4842-4026-7. Available from DOI: 10.1007/978-1-4842-4027-4.
2. WIKIPEDIA. *.NET Framework* [online]. 2015 [visited on 2023-03-05]. Available from: https://en.wikipedia.org/wiki/.NET_Framework.
3. RUMP, Arthur. *.NET Framework - Versions of .NET* [online]. 2023 [visited on 2023-03-05]. Available from: https://versionsof.net/framework/.
4. MICROSOFT. *Version compatibility* [online]. 2023 [visited on 2023-03-05]. Available from: https://learn.microsoft.com/en-us/dotnet/framework/migration-guide/version-compatibility.
5. RUMP, Arthur. *.NET Core - Versions of .NET* [online]. 2023 [visited on 2023-03-05]. Available from: https://versionsof.net/core/.
6. MICROSOFT. *.NET Standard* [online]. 2023 [visited on 2023-03-05]. Available from: https://learn.microsoft.com/en-us/dotnet/standard/net-standard.
7. MICROSOFT. *Ecma standards* [online]. 2023 [visited on 2023-03-05]. Available from: https://learn.microsoft.com/en-us/dotnet/fundamentals/standards.
8. ECMA. ECMA-334: C# language specification. *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,* 6th edition. 2022.
9. ECMA. ECMA-335: Common Language Infrastructure (CLI). *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,* 6th edition. 2012.
10. MICROSOFT. *Visual Basic language specification* [online]. 2023 [visited on 2023-03-06]. Available from: https://learn.microsoft.com/en-us/dotnet/visual-basic/reference/language-specification/.
11. F# SOFTWARE FOUNDATION AND INDIVIDUAL CONTRIBUTORS. *The F# Language Specification* [online]. 2020 [visited on 2023-03-06]. Available from: https://fsharp.org/specs/language-spec/.
12. IOLEVEL. *PeachPie | PHP compiler to .NET* [online]. 2023 [visited on 2023-03-06]. Available from: https://www.peachpie.io/.
13. WIKIPEDIA. *Category:.NET programming languages* [online]. 2021 [visited on 2023-03-06]. Available from: https://en.wikipedia.org/wiki/Category:.NET_programming_languages.
14. WIKIPEDIA. *List of CLI languages* [online]. 2021 [visited on 2023-03-06]. Available from: https://en.wikipedia.org/wiki/List_of_CLI_languages.

15. MICROSOFT. *.NET implementations* [online]. 2023 [visited on 2023-03-10]. Available from: https://learn.microsoft.com/en-us/dotnet/fundamentals/implementations.

16. .NET PLATFORM. *.NET Runtime* [https://github.com/dotnet/runtime]. GitHub, 2023.

17. .NET PLATFORM. *.NET Core Diagnostics Repo* [https://github.com/dotnet/diagnostics]. GitHub, 2023.

18. MICROSOFT. *What diagnostic tools are available in .NET Core?* [Online]. 2023 [visited on 2023-03-10]. Available from: https://learn.microsoft.com/en-us/dotnet/core/diagnostics/.

19. MICROSOFT. *ProcDump v11.0* [online]. 2023 [visited on 2023-03-10]. Available from: https://learn.microsoft.com/en-us/sysinternals/downloads/procdump.

20. SCHMICH, Chris. *What's the story behind the name of the SOS (Son of Strike) debugger extension?* [Online]. 2010 [visited on 2023-03-22]. Available from: https://stackoverflow.com/questions/3572990/whats-the-story-behind-the-name-of-the-sos-son-of-strike-debugger-extension.

21. WELLER, Thomas. *Cannot .loadby sos mscorwks or .loadby sos clr* [online]. 2018 [visited on 2023-03-25]. Available from: https://stackoverflow.com/questions/52258291/cannot-loadby-sos-mscorwks-or-loadby-sos-clr.

22. ICSHARPCODE. *ILSpy* [https://github.com/icsharpcode/ILSpy]. GitHub, 2023.

23. DNSPY. *dnSpy* [https://github.com/dnSpy/dnSpy]. GitHub, 2020.

24. DNSPYEX. *dnSpy* [https://github.com/dnSpyEx/dnSpy]. GitHub, 2023.

25. MICROSOFT. *gc.cpp* [online]. 2023 [visited on 2023-04-08]. Available from: https://github.com/dotnet/runtime/blob/340508f5c750d190090be93f0c731b1a8055a354/src/coreclr/gc/gc.cpp.

26. MICROSOFT. *gc.h* [online]. 2023 [visited on 2023-04-08]. Available from: https://github.com/dotnet/runtime/blob/340508f5c750d190090be93f0c731b1a8055a354/src/coreclr/gc/gc.h.

27. MICROSOFT. *Garbage Collection Design* [online]. 2023 [visited on 2023-04-09]. Available from: https://github.com/dotnet/runtime/blob/main/docs/design/coreclr/botr/garbage-collection.md.

28. GAPOTCHENKO. *Eazfuscator.NET - Features* [online]. 2023 [visited on 2023-04-16]. Available from: https://www.gapotchenko.com/eazfuscator.net/features.

29. GAPOTCHENKO, LLC. *Design-Time Usage Protection* [online]. 2023 [visited on 2023-04-15]. Available from: https://learn.gapotchenko.com/eazfuscator.net/docs/sensei-features/design-time-usage-protection.

30. NOTPRAB. *.NET-Obfuscator* [online]. 2023 [visited on 2023-04-16]. Available from: https://github.com/NotPrab/.NET-Obfuscator.

31. OBFUSCAR. *Obfuscar* [online]. 2023 [visited on 2023-04-16]. Available from: https://github.com/obfuscar/obfuscar.

32. SATO-ISOLATED. *MindLated* [online]. 2023 [visited on 2023-04-16]. Available from: https://github.com/Sato-Isolated/MindLated.

33. NWTSOLUTION. *CSharp Obfuscator* [online]. 2023 [visited on 2023-04-16]. Available from: https://github.com/nwtsolution/CSharpObfuscator.

34. NWTSOLUTION. *CSharp Obfuscator* [online]. 2023 [visited on 2023-04-16]. Available from: https://github.com/NotPrab/.NET-Deobfuscator.

35. MICROSOFT. *ILMerge* [online]. 2023 [visited on 2023-04-16]. Available from: https://github.com/dotnet/ILMerge.
36. DAN04. *Struct memory layout in C* [online]. 2010 [visited on 2023-04-28]. Available from: https://stackoverflow.com/questions/2748995/struct-memory-layout-in-c.
37. ALEX. *[Mono-dev] SOS.dll for Mono ?* [Online]. 2011 [visited on 2023-04-28]. Available from: https://mono.github.io/mail-archives/mono-devel-list/2011-August/037937.html.
38. MICROSOFT. *EventPipe* [online]. 2022 [visited on 2023-04-29]. Available from: https://learn.microsoft.com/en-us/dotnet/core/diagnostics/eventpipe.
39. MICROSOFT. *Microsoft.Diagnostics.NETCore.Client* [online]. 2023 [visited on 2023-04-29]. Available from: https://www.nuget.org/packages/Microsoft.Diagnostics.NETCore.Client.
40. PROSEK, Ladi. *Type Loader Design* [online]. 2007 [visited on 2023-04-29]. Available from: https://github.com/dotnet/runtime/blob/main/docs/design/coreclr/botr/type-loader.md.

Contents of the Attached Medium

- README.md ... Overview of the medium content
- bin/ ... Build binaries
- src/ ... Folder containing source codes
  - DotMemoryExplorer.Core/ ... Sources of .NET Memory Explorer Core Library
  - DotMemoryExplorer.Gui/ ... Sources of .NET Memory Explorer GUI Application
  - DotMemoryExplorer.DebugeeWpf/ ... Sources of Testing Program
  - DotMemoryExplorer.sln ... Visual Studio Solution containing C# projects
- thesis/ ... Source files of this thesis
- thesis.pdf ... Text of this thesis in PDF