Методы исследования программного обеспечения без использования исходных кодов

Авторы:

Велижанин А.С., Тюменский Государственный Нефтегазовый Университет, [Anatoliy.Velizhanin@gmail.com](mailto:Anatoliy.Velizhanin@gmail.com)

Ревнивых А.В., Тюменский Государственный Нефтегазовый Университет, alexchr@mail.ru

1. **Введение**

Современное программное обеспечение состоит из множества программных модулей, реализованных с применением различных технологий. Рассматривая тенденции развития технологической базы разработки программного обеспечения отметим движение в сторону платформы .NET Framework для Microsoft Windows. В основе функционирования программных решений, разработанных под платформу .NET Framework лежит байт-код. Принцип JIT компиляции в данной технологии несколько схож с подходом, реализованным для языка программирования Java [1].

Для UNIX-подобных операционных систем преимущественно разрабатывается “Native” (скомпилированное под платформу) программное обеспечение. Отдельно следует обратить внимание на Android системы в основе которых лежит ядро Linux. Значительная часть программных приложений для этой операционной системы разрабатывается для работы под Dalvik виртуальной машиной с применением Android SDK [2].

Многие программные решения распространяются без исходных кодов, к тому же настройки компиляторов в значительной степени влияют на результирующий исполняемый файл. Более того, некоторые программные модули разработаны для выполнения под управлением какой-либо виртуальной машины и скомпилированы в байт-код. Тогда дополнительное влияние на результирующие выполняемые машинные инструкции оказывает и версия виртуальной машины. Таким образом возрастает важность анализа программного обеспечения без использования исходных кодов.

В настоящее время ряд средств позволяет проводить реверс-инжиниринг как Native программного обеспечения, так и базирующегося на байт-коде. К таким средствам относятся Hex Rays IDA Pro [3], Red Gate .NET Reflector [4], JetBrains dotPeek [5] и другие.

Важно отметить, что вышеприведенные средства применяются для анализа исполняемых файлов на диске. Однако, учитывая современное разнообразие технологий разработки программного обеспечения, загруженное и инициализировавшее внутренние структуры данных приложение может несколько отличаться от образа на диске. К тому же, совмещение модулей, реализованных в форме байт-кода с Native компонентами программного решения, может привести к значительному усложнению процесса качественного анализа. Причиной тому является, в том числе, JIT-компиляция. Виртуальные машины различных версий могут генерировать несколько различный машинный код. Таким образом возрастает важность анализа фактически сформированного в ходе JIT-компиляции машинного кода, который зачастую совмещен с программным кодом Native модулей.

В качестве одного из вариантов решения данной задачи является анализ дампов памяти, снятых с приостановленного в интересующий момент времени процесса.

Работа посвящена исследованию методов анализа программного обеспечения без использования исходных кодов с учетом современных тенденций развития технологий разработки программных решений.

1. **Современные подходы к анализу программного обеспечения**

Методы анализа современного программного обеспечения в общем случае делят на три типа [6]:

1. Метод белого ящика. Данный метод подразумевает под собой анализ исходного кода программы как вручную, так и применяя различные статические анализаторы и др. утилиты анализа исходного кода.
2. Метод черного ящика. Примером данного метода является Fuzzing. Данный метод больше подходит для поиска уязвимостей и ошибок в программном обеспечении, поскольку методом полноценного анализом программного обеспечения данный подход назвать затруднительно.
3. Метод серого ящика. Традиционно принято считать, что в данном подходе исследователь прибегает к анализу дизассемблированного представления программного обеспечения. Данный метод считается наиболее сложным и требующим специальных навыков.

Данные подходы зачастую комбинируют для решения множества задач, начиная от тестирования программных решений компанией-разработчиком, заканчивая поиском уязвимостей злоумышленниками. Отметим, что компании-разработчики программного обеспечения зачастую ограничиваются различными тестами (Unit-тесты и т.п.). Исследование программного обеспечения методом серого ящика является достаточно редким явлением, однако такой подход позволяет проводить наиболее углубленное изучение поведения как отдельных программных компонентов, так и особенностей их функционирования в контексте крупных программных решений. В данной работе остановим наше внимание на технике анализа программного обеспечения методом серого ящика.

1. **Классический подход к анализу программного обеспечения методом серого ящика**

Классическим подходом к анализу программного обеспечения методом серого ящика является анализ дизассемблированного представления исследуемого программного решения.

Современное программное обеспечение, как уже сообщалось ранее, состоит как из Native-компонентов, так и из модулей, реализованных в виде байт-кода. Миграция приложений для Microsoft Windows под платформу .NET Framework происходит постепенно по причине объемности современных программных решений, что обуславливает смешивание Native и Managed модулей в рамках одного проекта. Примером такого смешивания может быть геологический пакет Schlumberger Petrel.

В настоящее время имеется ряд инструментов анализа как Native, так и основанных на байт-коде программных решений. Разделим существующие средства анализа программного обеспечения методом серого ящика на следующие категории:

1. Дизассемблеры. К таким инструментам относится Hex Rays IDA Pro.
   1. Для Native программных компонентов дизассемблеры формируют код на языке программирования Assembler [7] соответствующей архитектуры процессора. В состав пакета Microsoft Windows SDK так же включена утилита «dumpbin» [8] выполняющая дизассемблирование. Важно отметить, что последующая компиляция результатов работы дизассемблера в общем случае не даст возможности собрать функционирующее программное решение.
   2. Для байт-кода (в качестве примера рассмотрим байт-код платформы .NET Framework) дизассемблеры формируют код на промежуточном языке CIL [9]. Для дизассемблирования байт кода Microsoft так же предоставляет штатное решение «ildasm» [10]. Отметим, что сборка управляемых программных решений по результатам работы дизассемблера в общем случае является возможной.
2. Декомпиляторы. К таким инструментам относится Red Gate .NET Reflector для платформы .NET Framework. Результатом работы данного декомпилятора является код на выбранном пользователем языке программирования для платформы .NET Framework (C#, Visual Basic .NET и т.п.). Многие декомпиляторы предоставляют возможность экспортировать декомпилированный код в проект для Microsoft Visual Studio. В общем случае компиляция декомпилированной программы является возможной. Для Native решений отсутствуют полноценные декомпиляторы.
3. Отладчики. Утилиты данного типа позволяют проводить отладку программного обеспечения и могут ассоциироваться как со встроенными в интегрированную среду разработки решениями, так и с компонентами, запускаемыми отдельно. В данном контексте под отладчиками будем понимать соответствующие инструменты, но применительно к дизассемблированным или декомпилированным формам программного обеспечения.
   1. Для Native программных решений отладка происходит в режиме ассемблера соответствующей архитектуры процессора.
   2. Для байт-кода (в качестве примера рассмотрим байт-код платформы .NET Framework) отладка может происходить как на уровне CIL (или другого промежуточного языка для другой платформы), так и на основе декомпилированных листингов (например, в случае успешного экспорта и компиляции результатов работы декомпилятора)
4. Инструменты мониторинга активности программного обеспечения. К таким инструментам можно отнести целый пакет утилит SysinternalsSuite [11].

Инструменты приведенных категорий обычно используются по отдельности применительно к исследованию функционирования программного обеспечения. В то же время они дают в значительной степени разнородные результаты своей работы. Так, в рамках одного программного решения могут быть как Native модули, так и компоненты разработанные под платформу .NET Framework. Тогда применение дизассемблера для Native модуля позволит получить ассемблерное представление программного решения, а использование декомпилятора для компонента, функционирующего под управлением .NET Framework – высокоуровневый программный код. Средства анализа активности программного обеспечения выявить наиболее интересные модули и участки программного кода. Применение отладчика дает возможность более подробно изучить некоторые элементы программного решения во время выполнения программы.

Приведем участок дизассемблированного представления Native (Рис. 1) и разработанного под платформу .NET Framework (Рис. 2) программного компонента. На рис 3. Изображен участок декомпилированного кода программного компонента, разработанного под платформу .NET Framework.

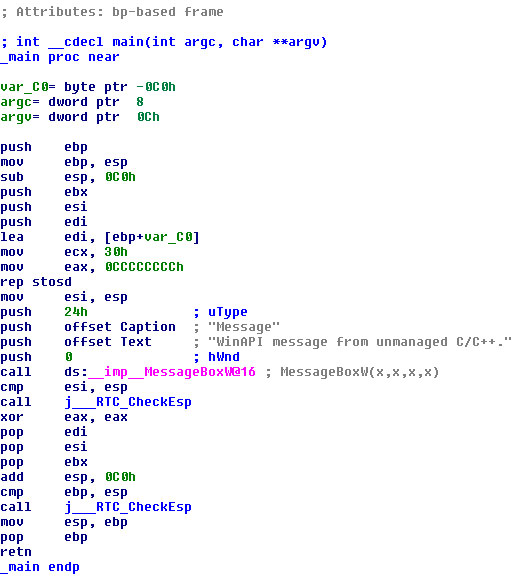


Рис 1. Дизассемблированное представление участка программного кода Native программного компонента.

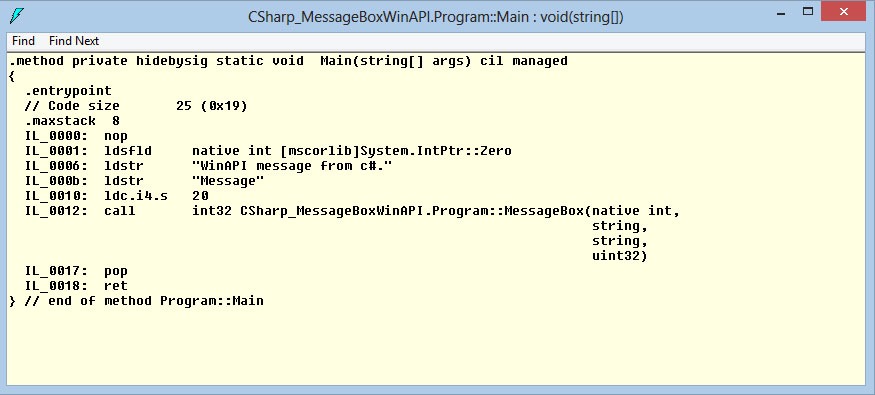


Рис 2. Дизассемблированное представление участка программного кода компонента, разработанного под платформу .NET Framework.



Рис 3. Участок декомпилированного кода программного компонента, разработанного под платформу .NET Framework.

Приведенные на рис. 1-3 примеры реверс-инжиниринга являются наиболее простыми. Во всех случаях был рассмотрен программный код, вызывающий функцию MessageBox из библиотеки user32.dll. Данные примеры приведены для визуализации различий в результатах работы высокоуровневых декомпиляторов и дизассемблеров для Native и байт-кода платформы .NET Framework. Несмотря на наличие вызовов Windows API из программного модуля, реализованного на языке программирования C# под платформу .NET Framework, это не приводит к значительному усложнению декомпилированного и дизассемблированного листингов.

Таким образом, в ходе классического анализа программного обеспечения методом серого ящика применяются самые различные инструменты. Наиболее частыми из них являются дизассемблеры, декомпиляторы, отладчики, мониторы активности программных продуктов и многое другое.

1. **Более сложное комбинирование управляемого и неуправляемого программного кода**

В настоящее время принято делить программный код на управляемый – выполняющийся под контролем среды исполнения (такой, как CLR - Common Language Runtime в .NET Framework) и неуправляемый (Native программный код). Это формирует в общем случае и деление языков программирования на соответствующие категории. Однако, реализация языка программирования С/С++ в Microsoft Visual C++ компиляторе, позволяет формировать как Native, так и байт код, в том числе в рамках одного проекта. На рис. 4 приведен дизассемблированный отрывок кода на языке программирования C/C++ в режиме смешанного (Native и Managed кода в рамках одного исполняемого программного модуля).

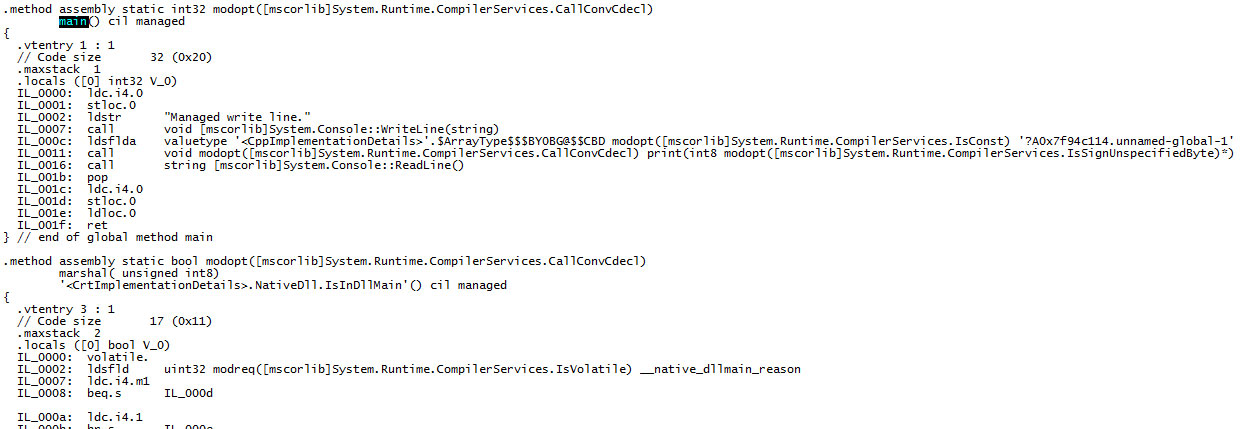


Рис. 4. Дизассемблированное представление смешанного кода.

Смешивание Native и Managed программного кода дает возможность удобной интеграции платформозависимого кода в управляемые сборки. Возможность такого смешивания реализована в компиляторе Microsoft Visual C++. Компиляторы Microsoft Visual C# и т.п. не предоставляют такой возможности. Декомпилированный с применением Red Gate .NET Reflector и JetBrains dotPeek программный код так же является крайне запутанным. Более того, после экспорта в проект зачастую не удается обнаружить искомый код по ключевым элементам.

Таким образом, если в случае с Microsoft Visual C# мы имели довольно удобочитаемый программный код после обработки декомпилятором, то в других случаях (например, компилятор Microsoft Visual C++) код на выходе декомпиляторов является запутанным и по уровню сложности понимания сопоставимым с дизассемблерным представлением, содержащем CIL код (рис. 4.).

1. **Анализ современного программного обеспечения**

Современное программное обеспечение сочетает в себе как Native, так и Managed программный код. Из вышеуказанного видно, что удобный анализ на основе декомпилированного кода не всегда возможен, на основе дизассемблированных в код СIL листингов так же затруднителен. Можно заметить, что в общем случае, анализ высокоуровневого программного кода гораздо удобнее анализа CIL инструкций и Native ассемблера. Однако, было так же показано, что не во всех случаях высокоуровневый код является удобным для понимания.

Управляемые сборки хранят в своем составе множество метаинформации. Такая метаинформация используется различными декомпиляторами для восстановления структуры кода. Генерация управляемых средой CLR объектов происходит во время выполнения кода. К таким объектам относятся как структуры данных различных типов, так и различные функциональные объекты.

Распространенные расширения SOS и SOSEX для отладчика WinDBG, как показали исследования, имеют ряд недостатков. Задача анализа смешанного (Native и Managed) кода значительно усложняется в случае отсутствия доступа к файлам символов.

В [12] изложено введение в функционирование программного модуля создания Rut-Time объектов в платформе .NET Framework. Генерируемый JIT-компилятором код так же является динамически создаваемым объектом и располагается в куче. The just-in-time (JIT) compiler generates x86 instructions and stores them on the JIT Code Heap [12]. На рис. 5 изображен снимок распределения памяти, полученный утилитой VMMap пакета SysinternalsSuite [13]

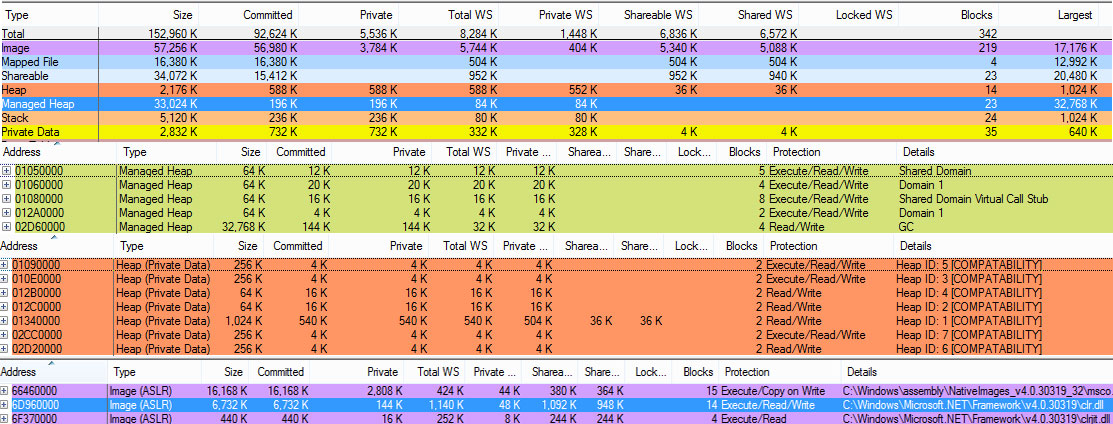


Рис. 5. Снимок распределения памяти управляемого процесса

Из рис. 5 мы видим, что управляемая куча (Managed Heap) имеет установленные флаги, разрешающие выполнение (Execute), запись (Write) и чтение (Read) процессом. Отметим, что переход на 64-х битную архитектуру частично обосновывался и необходимостью в защите страниц памяти с применением NX-бита (XD-бита в терминологии Intel), запрещающего выполнение кода на страницах с данными [14].

Важно отметить, что генерируемый в режиме Run-Time код и данные не фиксированы по определенным участкам памяти и могут быть перемещены исполняющей средой в процессе работы. С другой стороны, CIL код, можно обнаружить с помощью отладчика отступив от базового адреса загрузки исполняемого модуля в память смещение секции кода и смещение функции. Это проиллюстрировано на рис. 6.

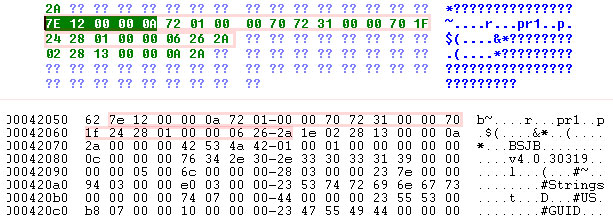


Рис 6. IL код функции main в Ida Pro (сверху) и снятый с дампа в WinDBG (внизу)

1. **Заключение**

Современное программное обеспечение совмещает в себе как Native, так и Managed программные модули. Кроме того, компилятор Microsoft Visual C++ позволяет формировать Mixed код, хранящий в одном исполняемом модуле и платформозависимый и байт-код. Существующие инструменты анализа смешанного (Native и Managed) кода являются в значительной степени разрозненными и не совершенными. С другой стороны, программное обеспечение, несмотря на наличие байт-кода в некоторых программных модулях, с помощью ряда методик организуется в однообразный (с точки зрения процессора) исполняемый машинный код. .NET сборки содержат в себе метаданные и код на языке CIL.

В ряде случаев доступ к исходным кодам исследуемого проекта затруднителен или невозможен. Кроме того, важно учитывать именно особенности фактически выполняемого кода, полученного, в том числе, после JIT компиляции. В связи с этим видится важным развитие средств анализа программного обеспечения, способных обрабатывать не только Native или Managed, но и Mixed сборки.

**Список литературы**

[1] <http://ru.wikipedia.org/wiki/.NET_Framework> (дата обращения: 01.10.2013)

[2] <http://developer.android.com/sdk/index.html> (дата обращения: 01.10.2013)

[3] <https://www.hex-rays.com/products/ida/index.shtml> (дата обращения: 01.10.2013)

[4] <http://www.red-gate.com/products/dotnet-development/reflector/> (дата обращения: 01.10.2013)

[5] <http://www.jetbrains.com/decompiler/> (дата обращения: 01.10.2013)

[6] Fuzzing. Исследование уязвимостей методом грубой силы, Автор: Майкл Саттон, Адам Грин, Педрам Амини, Издательство: Символ-Плюс, ISBN 978-5-93286-147-9; 2009 г.

[7] <http://en.wikipedia.org/wiki/Assembler_(computing)#Assembler> (дата обращения: 01.10.2013)

[8] <http://msdn.microsoft.com/ru-ru/library/c1h23y6c.aspx> (дата обращения: 02.10.2013)

[9] <http://en.wikipedia.org/wiki/Common_Intermediate_Language> (дата обращения: 02.10.2013)

[10] <http://msdn.microsoft.com/ru-ru/library/f7dy01k1.aspx> (дата обращения: 02.10.2013)

[11] <http://technet.microsoft.com/en-us/sysinternals/bb842062.aspx> (дата обращения: 02.10.2013)

[12] <http://msdn.microsoft.com/en-us/magazine/cc163791.aspx> (дата обращения: 02.10.2013)

[13] <http://technet.microsoft.com/en-us/sysinternals/bb842062.aspx> (дата обращения: 02.10.2013)

[14] <http://ru.wikipedia.org/wiki/NX_bit> (дата обращения: 02.10.2013)

**Приложение 1. Программа из рис.1.**

#include <Windows.h>

int main(int argc, char\*\* argv)

{

MessageBox(0, TEXT("WinAPI message from unmanaged C/C++."), TEXT("Message"), MB\_YESNO | MB\_ICONQUESTION);

return 0;

}

**Приложение 2. Программа из рис.2-3.**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Runtime.InteropServices;

using System.Text;

using System.Threading.Tasks;

namespace CSharp\_MessageBoxWinAPI

{

class Program

{

[DllImport("user32.dll", CharSet = CharSet.Unicode)]

public static extern int MessageBox(IntPtr hWnd, String text, String caption, uint type);

enum MsgType : ulong

{

MB\_ABORTRETRYIGNORE = 0x00000002L,

MB\_CANCELTRYCONTINUE = 0x00000006L,

MB\_HELP = 0x00004000L,

MB\_OK = 0x00000000L,

MB\_OKCANCEL = 0x00000001L,

MB\_RETRYCANCEL = 0x00000005L,

MB\_YESNO = 0x00000004L,

MB\_YESNOCANCEL = 0x00000003L

};

enum IconType : ulong

{

MB\_ICONEXCLAMATION = 0x00000030L,

MB\_ICONWARNING = 0x00000030L,

MB\_ICONINFORMATION = 0x00000040L,

MB\_ICONASTERISK = 0x00000040L,

MB\_ICONQUESTION = 0x00000020L,

MB\_ICONSTOP = 0x00000010L,

MB\_ICONERROR = 0x00000010L,

MB\_ICONHAND = 0x00000010L

};

static void Main(string[] args)

{

MessageBox(IntPtr.Zero, "WinAPI message from c#.", "Message", (uint)((ulong)MsgType.MB\_YESNO | (ulong)IconType.MB\_ICONQUESTION));

}

}

}

**Приложение 3. Программа из рис.4.**

#using <mscorlib.dll>

#include <stdio.h>

using namespace System;

#pragma unmanaged

void print(char \*message)

{

printf("%s\n", message);

}

#pragma managed

int main()

{

Console::WriteLine("Managed write line.");

print("Unmanaged write line.");

Console::ReadLine();

return 0;

}