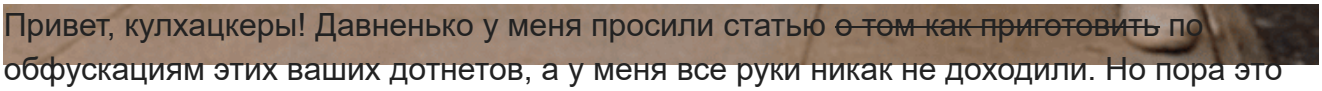
Статья - Дотнетовская кулинария (часть 1)

DildoFagins


ОБФУСЦИРОВАННЫЙ СТИЛЛЕР
АВЕР
МЕНТ
ТВИТТОРНЫЙ
"ЭКСПЕРТ"



Привет, кулхацкеры! Давненько у меня просили статью ~~о том как приготовить~~ по обфускациям этих ваших дотнетов, а у меня все руки никак не доходили. Но пора это исправить, так ведь? Я долго думал о том, как представить эту темку в удобоваримом формате, чтобы осветить всю базовую матчасть, необходимую для понимания того, что, собственно, происходит в обфускаторах дотнетов. Мы не будем рассматривать элитные приваты, но сконцентрируемся на информации, которая доступна в паблике, и на которой можно и нужно учиться. В последствии, после того как мы рассмотрим тот или иной алгоритм, я дам пару подсказок и советов о том, как улучшить алгоритм из паблика. Готовой и неготовой инфы получилось как то многовато, в итоге я решил разбить статью на три части, но надеюсь, у меня всё это выйдет нормально. Ну поехали с первой погружательной частью...

Содержание (часть 1):

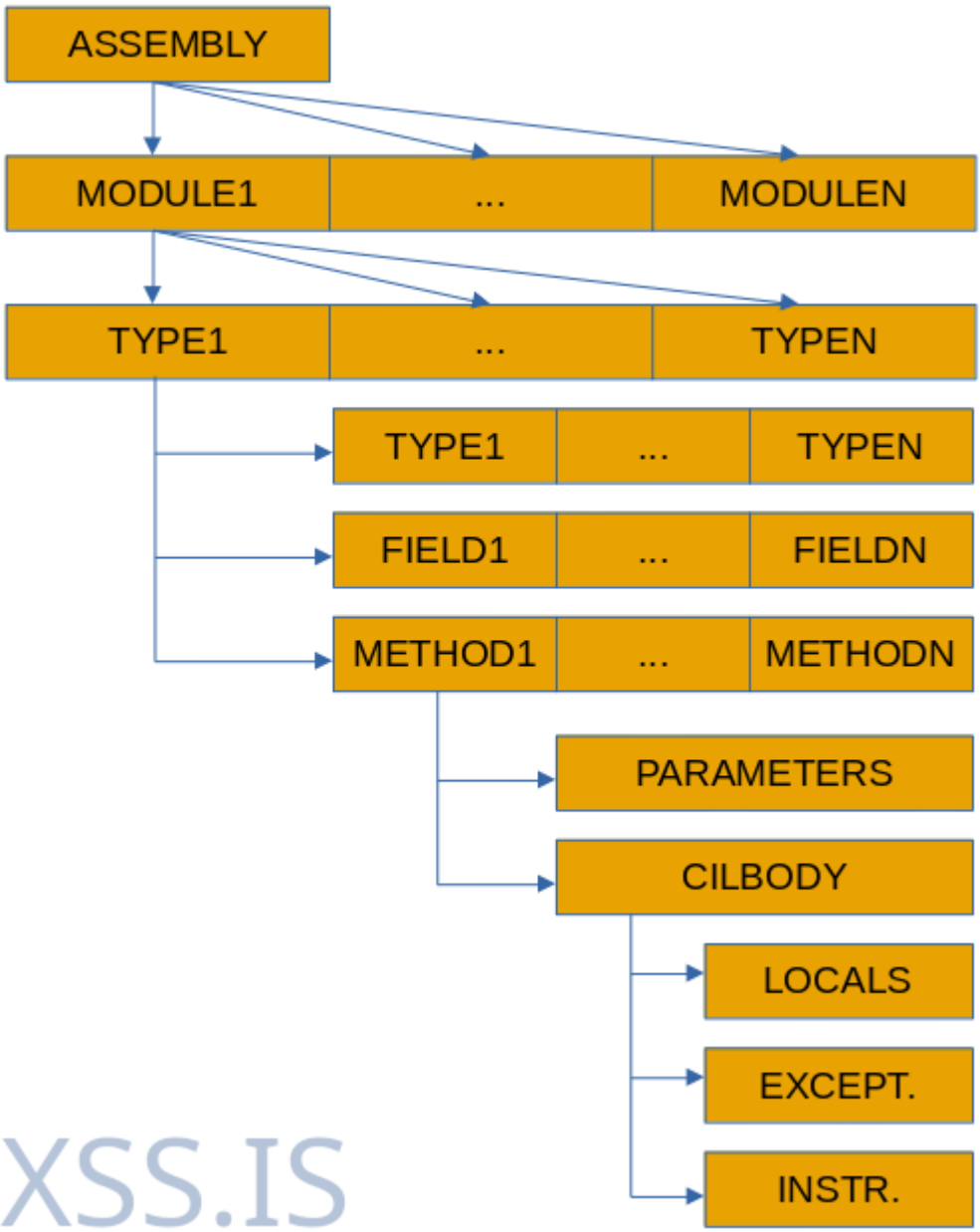
В предвкушении будущих частей:

1. Такие странные дотнет-сборки.

Многие из вас, наверняка, знают, что в Венде исполняемый файлы (программы), динамические библиотеки и тому подобное принято содержать в специальном формате файлов, который носит незамысловатое название «Portable Executable» (он же PE). Но для некоторых людей, как показала практика, не всегда очевидно, что в PE-файлы (помимо исполняемого нативного кода) можно напихать разных странных вещей. Собственно, так и происходит, когда компилируются дотнет-программы из самых разных языков программирования, таких как C#, F#, VB.NET и других. Программа или динамическая библиотека в мире дотнетов носит название «сборка» (она же Assembly) и от традиционных исполняемых файлов в ней остается как говорится «х#й, да ни хуя» (с). Конечно, сборка может иметь прямо внутри себя фрагменты нативного кода, но такую «эзотерику» мы сегодня рассматривать не будем. Обычно в PE-файле сборки присутствует только миниатюрный стаб, который запускает эти ихние дотнеты в текущем процессе и интегрирует в них код, который не имеет к нативному коду никакого отношения чуть более, чем полностью.

С этого момента давайте поподробнее. Дотнет сборка представляет собой набор таблиц мета-данных, элементы которых ссылаются друг на друга с помощью токенов, которые на самом деле ни что иное, как идентификатор таблицы и номер строки в этой таблице, забитые в байты одного 32-битного числа. Такой набор таблиц стандартизирован и всегда имеет примерно один и тот же вид. Обычно он заботливо запихан внутрь секции «.text» формата PE-файлов, но теоретически он может лежать в любой из секций PE-файла, лишь бы она обладала правами на чтение (такое никогда не встречается, но возможно). Технические детали того, как организованы таблицы мета-данных, для понимания дотнет обфускации не так критичны, важно только знать, что все дотнетовские вещи так или иначе будут находиться в этих таблицах.

Логически же дотнет-сборка имеет вид древовидной структуры, а вот понимать ее как раз может оказаться довольно полезным. Во главе (корнем дерева) стоит сама сборка (Assembly). У сборки может быть один или несколько модулей (Module), хотя чаще всего в одной сборке один модуль. Каждый модуль может содержать один или несколько типов (Type), тип же в свою очередь может быть классом (Class), интерфейсом (Interface), перечислением (Enum) и так далее. Чуть ниже, например, классы могут иметь методы (Method) и поля (Field), которые могут быть статическими (Static) и объектными (Instance). Забавно, но, например, на низком уровне классы не имеют свойств (Properties), как отдельных сущностей, они физически реализованы через поля и методы с префиксами «get_» и «set_» для геттеров и сеттеров соответственно. А тип делегата (Delegate) на самом деле является классом с определенным набором методов. Создание таких обвязок является задачей конкретного компилятора (в мире дотнетов это - чаще всего компилятор языка C#, но не стоит забывать, что есть еще и VB.NET, F#). В свою очередь методы могут иметь или не иметь «тела» (CILBody) и аргументов вызова (Parameter). Тело метода может иметь локальные переменные (LocalVariable), обработчики исключений (ExceptionHandler) и набор инструкций MSIL-кода (Instructions). А каждая инструкция имеет размер, тип и операнд в том случае, когда он необходим этой инструкции. Операндом инструкции на низком уровне может быть константа или тот самый токен, о котором мы говорили чуть ранее (за исключением инструкции «switch», которая операндом принимает список смещений, и она, если честно, довольно ебанутая, как по мне).



Об MSIL/CIL-коде мы поговорим чуточку позже, а пока поподробнее рассмотрим логическую структуру дотнет-сборок. Каждый элемент в такой древовидной структуре имеет набор различных свойств. Например, у класса есть имя (Name) и пространство имен (Namespace), причем пространство имен не является древовидной структурой, это просто строка, в которую можно напихать точек, чтобы предать этой строке какую-то иерархию. Класс может быть публичным (Public) или внутренним (Internal), статическим (Static), абстрактным (Abstract), вложенным в другой класс и так далее. Все эти аспекты сущности класса так или иначе описаны в его свойствах в таблицах метаданных. Описывать все аспекты вряд ли является целесообразным, в статье мы коснемся только тех из них, которые необходимо знать для базового понимания того, как работают алгоритмы обфускации дотнетов.

Говоря о странностях дотнет сборок, нельзя не упомянуть о такой забавной и иногда полезной особенности… Дотнет сборки могут иметь один из четырех флагов, назовем их «флагами архитектуры»: x64, x86, AnyCpu и AnyCpu 32-bit preffered. Если установлен флаг x64, то сборка будет запускаться как 64-битный процесс всегда (и не будет запускаться на 32-битных системах). Если установлен флаг x86, то сборка всегда будет запускаться как 32-битный процесс, как на 64-битной операционной системе, так и на 32-битной. Если установлен флаг AnyCpu, то сборка запускается как 64-битный процесс на 64-битной системе, и как 32-битный процесс на 32-битной системе. Этот флаг уже дает нам (не очевидное изначально) преимущество в том плане, что не нужно изъебываться всякими хевенсгейтами, чтобы оказаться в 64-битном режиме на 64-битной системе, при этом прекрасно работая на 32-битной системе. Механизм того, как это реализовано мелкомягкими довольно забавен, почитать о нем подробнее можно тут: https://debugandconquer.blogspot.com/2015/04/the-relationship-between-net-and.html?m=1 — кто бы знал, что загрузчик PE-файлов операционной системы вполне себе может «переобуть» процесс из 32-битного в 64-битный. Кастую призыв моих «ядерных-бро» varwar и atavism, возможно им будет это интересно (я юзер-модный хомячок, за ядро не то чтобы шарю, поэтому от меня не спрашивайте глубоких подробностей об этом)…

И еще один забавный факт про дотнетовские сборки. При старте нового процесса исполняемый файл загружается в виртуальную память по всем привычным правилам, то есть настраиваются секции (включая атрибуты доступа к ним), таблица импорта и так далее, как это происходит с нативными PE-файлами. Но когда мы, допустим, с помощью Assembly.Load загружаем из памяти другие дотнет-сборки, то никаких обычных для PE-файлов настроек не происходит. Загруженные таким образом сборки лежат в памяти так же в виде массива, идентичного тому, что был передан в Assembly.Load. Более того, настройка атрибутов доступа к памяти тоже не происходит, и этот байтовый массив лежит на страницах с доступом на чтение/запись (без доступа на исполнение). Флаг исполнения оказывается «не нужон» (с), если в сборке отсутствует нативный код, а есть только MSIL. Такое поведение дает нам еще одно (не очевидное) преимущество с точки зрения сканеров памяти. Так как, если затереть ряд заголовков и типичных для PE-формата вещей (при этом ничего не поломав), то загруженная сборка будет внешне выглядеть в памяти, как данные, а не как исполняемый файл. Конечно, сканеры памяти тоже успели поесть говна с этим и умеют до определенной степени успеха такие вещи определять, а также ETW может дать им посказочку о том, что этот буфер - сборка, но сам факт, что такая возможность есть, немножко да греет душонку малварщика.

Для чтения, модификации и записи таблиц метаданных на высоком уровне существует несколько библиотек: Mono.Cecil, dnlib, AsmResolver (все три библиотеки написаны на C# и предоставляются в виде дотнет-библиотеки классов) и DotNetPeLib (библиотека

на C++, но я ей никогда не пользовался, ничего не могу сказать). Из всех библиотек в своих проектах я предпочитаю использовать AsmResolver, поскольку помимо в некоторых аспектах более качественной поддержки метаданных дотнетов, она еще имеет хороший набор функционала для манипуляции непосредственно PE-форматом. Как мы в дальнейшем увидим, многие из публичных проектов обфускаторов будут использовать dnlib. API этих библиотек довольно похожи друг на друга и пересесть с одной библиотеки на другую не составит большого труда (я начинал с использования Mono.Cecil, потом без проблем пересел на dnlib, а уже потом моё сердечько целиком и полностью завоевал AsmResolver, который я стал использовать для многих вещей, даже иногда не связанных с дотнетами). Указанные библиотеки можно найти по следующим ссылкам:

GitHub - jbevain/cecil: Cecil is a library to inspect, modify and create .NET programs and libraries.

Cecil is a library to inspect, modify and create .NET programs and libraries. - GitHub - jbevain/cecil: Cecil is a library to inspect, modify and create .NET programs and libraries.

github.com

GitHub - 0xd4d/dnlib: Reads and writes .NET assemblies and modules

Reads and writes .NET assemblies and modules. Contribute to 0xd4d/dnlib development by creating an account on GitHub.

github.com

GitHub - Washi1337/AsmResolver: A library for creating, reading and editing PE files and .NET modules.

A library for creating, reading and editing PE files and .NET modules. - GitHub - Washi1337/AsmResolver: A library for creating, reading and editing PE files and .NET modules.

github.com

GitHub - LADSoft/DotNetPELib: A C++11 library used to create a managed

program (CIL) and dump to either .IL, .EXE, or .DLL format

A C++11 library used to create a managed program (CIL) and dump to either .IL, .EXE, or .DLL format - GitHub - LADSoft/DotNetPELib: A C++11 library used to create a managed program (CIL) and dump t...

github.com

Мы с вами будем рассматривать несколько доступных в паблике обфускаторов, которые так или иначе реализуют один или несколько типичных аспектов обфускации дотнетов. Хорошо каждый конкретный аспект сделан или нет тут скорее не важно, поскольку я предполагаю, что погружаясь в тему обфускации вы будете делать свою собственную реализацию аспектов «с преферансом и куртизанками» (с). Помните, что все, что доступно в каком-то виде в паблике, уже палится большинством аверов, или начнет палиться, когда скрипткиддисы его окончательно задрочат. Поэтому я — своего рода приверженец кастомных приватных тулзов, однако учиться на публичных проектах на мой взгляд совершенно не грешно. Для вас и этой статьи я отобрал следующие проекты, в исходный код которых мы будем заглядывать в процессе всех частей:

GitHub - yck1509/ConfuserEx: An open-source, free protector for .NET applications

An open-source, free protector for .NET applications - GitHub - yck1509/ConfuserEx: An open-source, free protector for .NET applications

github.com

GitHub - obfuscar/obfuscar: Open source obfuscation tool for .NET assemblies

Open source obfuscation tool for .NET assemblies. Contribute to obfuscar/obfuscar development by creating an account on GitHub.

github.com

GitHub - Elliesaur/TinyJitHook: Example JIT Hook for .NET FW/Core.

Example JIT Hook for .NET FW/Core. Contribute to Elliesaur/TinyJitHook development by creating an account on GitHub.

github.com

GitHub - hexck/Hex-Virtualization: .NET Virtualization made in C#

:guardsman: .NET Virtualization made in C#. Contribute to hexck/Hex-Virtualization development by creating an account on GitHub.

github.com

GitHub - TobitoFatitoRE/MemeVM: A small virtualizer for .NET which works together with ConfuserEx

A small virtualizer for .NET which works together with ConfuserEx - GitHub - TobitoFatitoRE/MemeVM: A small virtualizer for .NET which works together with ConfuserEx

github.com

GitHub - TheWover/DInvoke: Dynamically invoke arbitrary unmanaged code from managed code without PInvoke.

Dynamically invoke arbitrary unmanaged code from managed code without PInvoke. - GitHub - TheWover/DInvoke: Dynamically invoke arbitrary unmanaged code from managed code without PInvoke.

github.com

<утекшие_исходники_VMProtect>

2. Кто вставил Forth внутрь дотнета?

Ранее я сказал, что дотнет-код и нативный код — это две довольно разные вещи. Нативный код, который компилируется из этих ваших Сишечек, Плюсов, Дэ, Ржавого, Нимов и других нативных языков программирования исполняются на реальном железе (читай «на процессоре»). Шарпы (будь то C# или F#) и VB.NET компилируются в специальный код, который называют CIL (Common Intermediate Language) или же MSIL (который по сути является синонимом первого, я буду использовать оба термина, не запариваясь о деталях). CIL — это байткод для стековой виртуальной машины и среды исполнения CLR (Common Language Runtime), которая живет в глубине дотнетов. На

вендовых операционных системах часто предустановлена CLR версии 2.0 (на которой исполняется код для фреймворка 2.0-3.5, идет в базовой пачке Windows 7) и/или версии 4.0 (на которой исполняется код для фреймворка 4.0-4.8, идет в базовой пачке Windows 10). Тут важно заметить, что фреймворк 3.5 на самом деле — фреймворк 2.0 с набором дополнительных библиотек, а с фреймворка 4.0 версионность уже идет более адекватно в том плане, что куда логичнее и понятнее, что фреймворк 4.5 — это фреймворк 4.0 плюс библиотеки. Современные дотнеты 5.0 и выше имеют немного другую среду исполнения, рассматривать их мы не будем потому, что малвари на дотнетах в подавляющем большинстве случаев используют именно фреймворк 2.0-4.8 (хотя сейчас уже, наверное, можно смело сказать 4.0+). А на компьютерах обычных рядовых пользователей крайне редко можно встретить дотнеты 5.0 или того выше (которые не предустановлены и их нужно устанавливать отдельно).

Стековую виртуальную машину мы уже однажды с вами писали тут: https://xss.is/threads/64508/ - по сути все подобные ВМ имеют примерно похожий вид (как, например, у Java или CPython). Смысл в том, что вместо привычных глазу реверсера регистров для подавляющего большинства операций используется стек. Я не знаю, почему исторически сложилось, что стековые виртуальные машины куда более распространены, чем регистровые, скорее всего в стековое представление проще компилить код из абстрактного синтаксического дерева (не нужно писать специальный аллокатор регистров). Но факт остается фактом, среди реализаций языков программирования вы куда чаще встретите именно стековые виртуальные машины (передаю привет бразильцам, которые Lua пилят, быть другим — нормально, не стесняйтесь этого).

Реализация CLR использует так называемый «JIT-компилятор» (от выражения «just in time»). Смысл в том, что при первой попытке вызова метода его CIL-код компилируется в нативный код для текущей архитектуры (x86 — для 32-битного процесса, x64 — для 64-битного). JIT-компилятор выделяет исполняемую память на специальной куче, считывает CIL-код метода, с применением ряда оптимизаций компилирует его в нативный код, записывает его в выделенный исполняемый буфер, и запускает на исполнение. Компиляцию метода можно вызвать принудительно еще до его непосредственного вызова с помощью RuntimeHelpers.PrepareMethod, это часто используется при взаимодействии с внешними нативными библиотеками (например, когда в библиотеку нужно передать callback). С JIT-компилятором тоже можно своего рода «поиграться», но об этом мы поговорим позже, когда познакомимся с пиратом по прозвищу Джит-Крюк, пока не будем забегать вперед.

Для того, чтобы было удобно исследовать механизмы работы CIL-кода и JIT-компилятора, есть замечательный сайт: https://sharplab.io/ - на нем можно писать C#-код, а затем смотреть, в какой CIL-код он скомпилировался, и в какой нативный код CIL-код будет скомпилирован JIT-компилятором. Конечно, в рамках одной статьи

невозможно коснуться всех интересных аспектов работы CIL-кода, и тем более нельзя научить вас программировать на чистом MSIL'е (или думать в рамках стекового языка программирования). Поэтому я рекомендую вам самим потыкать сайт sharplab.io и посмотреть, какие конструкции высокоуровневого языка C# компилируются в какие низкоуровневые конструкции CIL-кода. Это будет полезно. Я же в статье приведу только небольшой ряд примеров, которого на мой взгляд будет достаточно для понимания остального контента статей цикла.

Как я уже говорил, в стековых языках программирование все происходит через стек (очевидно, к гадалке можно не ходить). Давайте увидим это на примере одной простой операции сложения двух 32-битных чисел. На сайте sharplab.io вобьем следующий Цэ-шарповый код:
C#:

```
using System;

public class C {
    public int M(int a, int b) {
        return a + b;
    }
}
```

Окей, теперь давайте посмотрим, какой же CIL-код мы получили из этого. Инструкция ldarg.N кладет на стек аргумент вызова метода с номером N, то есть при исполнении инструкций IL_0000 и IL_0001 на стеке последовательно окажутся два аргумента вызова метода. Кстати, нулевым аргументом в instance-методы класса неявно передается ссылка на this (при необходимости кладется на стек инструкцией ldarg.0). Далее будет исполнена инструкция add. Она забирает два элемента с верхушки стека, складывает их вместе и кладет результат сложения на верх стека. Инструкция ret выходит из метода или же завершает его. При этом, если метод не является void, а возвращает какое-либо значение, то инструкция ret получает это значение с верхушки стека и кладет его на стек вызывающего метода (таким образом происходит возврат значения).
Code:

```
IL_0000: ldarg.1
IL_0001: ldarg.2
IL_0002: add
IL_0003: ret
```

Забавно, что CLR (видимо, для ускорения своей работы) проводит очень мало валидаций того, что находится на стеке. Она предполагает, что CIL-код всегда

корректно сгенерирован компилятором. Поэтому, изменяя CIL-код можно как все неимоверно испортить (что обычно приводит к жестким падениями или к BadImageFormatException), так и получить разные забавные эффекты (например, получить внутренний адрес объекта из его ссылки на стеке, с чем в обычном не-unsafe коде C# компилятор не захочет вам помогать). В контексте этой статьи важно понимать, что CIL-код довольно хрупок, и нужно быть внимательным, чтобы ничего не поломать. Ну и ради интереса давайте посмотрим в какой нативный код будет скомпилирован наш пример JIT-компилятором (уффф, настолько оптимизированно, что мне пришлось секунд 10 разглядывать asm-код, чтобы понять, куда сложение делось):
Code:

```
L0000: lea eax, [rdx+r8]
L0004: ret
```

Вызов методов происходит с помощью инструкций call, callvirt и довольно изотерической инструкции calli. Инструкция call используется для вызова статических и невиртуальных методов. Если метод так или иначе задействован в цепочке наследования, то для вызова используется инструкция callvirt (которая уже осуществляет вызов через vtable). Vtable дотнетов на уровне нативного кода реализован очень похоже на тоже самое в плюсах (скорее всего, это было сделано для интероп в том числе и с COM-классами). Физически у объекта в памяти есть указатель на таблицу указателей на виртуальные методы. При вызове виртуального метода сначала происходит разименование указателя на таблицу с указателями на методы, затем из этой таблицы получается нужный указатель, затем метод вызывается по этому указателю. Мы с вами делали бы тоже самое руками, если бы пытались вызвать такой метод из православной Сишечки, в Плюсах же и Шарпах эти вещи реализует компилятор (в первом случае) и рантайм (во втором). Про calli не спрашивайте, ебал я в рот в этой залупе разбираться . Аргументы для вызова метода кладутся на стек в прямом (а не обратном) порядке. То есть вызов статического метода TestMethod(1, 2, 3) будет выглядеть так:
Code:

```
IL_0000: ldc.i4.1
IL_0001: ldc.i4.2
IL_0002: ldc.i4.3
IL_0003: call void C::TestMethod(int32, int32, int32)
```

Теперь, когда вы уже это всё увидели (идеальное планирование статьи в действии), давайте я расскажу, как на стеке должны оказываться различного рода константы. Для этого есть ряд специальных инструкций. Так, например, чтобы на стек положить 32-

битное число, нужно воспользоваться инструкцией ldc.i4 (или одной из ее коротких форм), для 64-битного числа есть инструкция ldc.i8, а для чисел с плавающей точкой инструкции ldc.r4 и ldc.r8 для float и double соответственно. Значение константы вшивается в код в след за опкодом инструкции. Особняком в этой группе стоит инструкция ldstr, которая загружает на стек константную строку. Дело в том, что вместо непосредственного значения (как в случае ldc.i4 и других) за опкодом в CIL-код вшивается тот самый токен, который указывает на строку в таблицах мета-данных. То есть сами данные строки хранятся отдельно, на данные ссылается элемент таблицы мета-данных, а на элемент таблицы через токен ссылается код. Это делается для того, чтобы не раздувать объем кода (за счет дублирования данных строки) в том случае, если в нескольких местах код будет пытаться положить на стек одну и ту же строку. Сложно, но понять и простить можно.

Но часто на форумах я вижу отчаянный вопрос: «так а как на блядский стек положить массив, чего-то я не вдупляю ёбанарот». Да, брат, понимаю твою боль, массивы в CIL-коде — это совершенно отдельный способ садомазохизма (когда пытаешься их реализовывать на уровне CIL-кода). Для начала давайте посмотрим, как на стеке оказывается массив объектов (в данном случае массив строк):
C#:

```
using System;

public class C {
    public string[] M() {
        return new string[] {
           "wtf is that, bro?",
           "arrays sucks so much"
        };
    }
}
```

Для начала на стеке нужно создать объект массива с пустым набором элементов. Для этого на стек сначала помещается длина массива (ldc.i4.2 — загружает на стек двойку), а затем с помощью инструкции newarr создается массив нужного нам типа (в качестве аргумента инструкция принимает токен, указывающий на описание типа в таблице мета-данных). Затем внутрь массива запихиваются элементы друг за другом. Для этого для каждого элемента сначала дублируется ссылка на массив с помощью инструкции dup (получает с вершины стека ссылку и добавляет ее копию выше исходной в стеке). Помимо инструкции dup может быть использована инструкция ldloc, если массив хранится в локальной переменной. Затем на стек кладется порядковый номер элемента и само значение для элемента (в нашем случае это строки, которые загружаются на стек с помощью инструкции ldstr). Далее идет инструкция stelem.ref, которая забирает со стека значение элемента, его индекс (порядковый номер) и

ссылку на массив, и записывает в массив элемент по индексу. С одной стороны это выглядит неоправданно сложно, но с другой стороны, а как вы еще сделаете массивы в стековом языке программирования:
Code:

```
IL_0000: ldc.i4.2
IL_0001: newarr [mscorlib]System.String
IL_0006: dup
IL_0007: ldc.i4.0
IL_0008: ldstr "wtf is that, bro?"
IL_000d: stelem.ref
IL_000e: dup
IL_000f: ldc.i4.1
IL_0010: ldstr "arrays sucks so much"
IL_0015: stelem.ref
IL_0016: ret
```

Теперь давайте посмотрим, как дела обстоят с массивами примитивных типов, например, с массивами 32-битных целых чисел. И предвкушая ваше удивление, тут все еще неоправданно сложнее…
C#:

```
using System;

public class C {
    public int[] M() {
        return new int[] {
           1, 2, 3, 4, 5
        };
    }
}
```

Как и в прошлый раз (в случае с объектами) сначала на стеке создается массив на этот раз уже примитивного типа (инструкциями ldc.i4 и newarr). Но затем происходит с первого взгляда «неведомая х#йня» и «страшно, очень страшно, если бы мы знали, что это, мы не знаем что это» (с). Ну давайте я поясню. Непосредственно данные массива примитивных элементов хранятся прямо внутри CIL-кода. Для этого компилятор создает специальный класс с незамысловатым названием «<PrivateImplementationDetails>», этому классу создается поле, которое указывает на данные массива примитивных элементов. Токен этого поля загружается на стек инструкцией ldtoken, а затем вызывается метод InitializeArray (инструкцией call). Этот метод получает первым аргументом ссылку на массив, вторым — токен поля (все со стека) и копирует данные примитивных типов изнутри поля в выделенный для массива буфер. На уровне CIL-кода такую вещь реализовывать та еще морока.

Code:

```
IL_0000: ldc.i4.5
IL_0001: newarr [mscorlib]System.Int32
IL_0006: dup
IL_0007: ldtoken field valuetype
'<PrivateImplementationDetails>'/'__StaticArrayInitTypeSize=20'
'<PrivateImplementationDetails>'::'4F6ADDC9659D6FB90FE94B6688A79F2A1FA8D36EC43F8F3E1D9B6528C4
48A384'
IL_000c: call void
[mscorlib]System.Runtime.CompilerServices.RuntimeHelpers::InitializeArray(class
[mscorlib]System.Array, valuetype [mscorlib]System.RuntimeFieldHandle)
IL_0011: ret
```

Безусловно, чтобы разобраться во всех тонкостях того, что происходит в CIL-коде, таких примеров далеко не достаточно. Но я надеюсь, что я научил вас главному: как при необходимости удобно и просто смотреть, какие высокоуровневые конструкции во что компилируются. Обычно я так и делал сам. Нужно что-то сделать на уровне MSIL — сделал на Шарпах, посмотрел CIL-код, понял его, повторил на уровне CIL-кода, получил профит. При необходимости можно почитать описание всех инструкций CIL-кода на MSDN’е тут: https://learn.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes?view=netframework-4.0 — знаю, что мелкомягкие любят перехерачить ссылки MSDN’а, поэтому, если эта ссылка протухнет, просто гуглите конкретную инструкцию, типа «opcode ldstr».

Ну и да, возвращаясь к вопросу о том, кто вставил Forth внутрь дотнетов? Я не знаю, наверное, мистер Андерс Хейлсберг. Но он скорее всего спиздел вдохновлялся реализацией Джавы, а не Фортом в чистом виде. Если кому интересно, посмотрите на стековые языки программирования типа Форта и сразу поймете отсылку...

3. Что в имени тебе моем?

Как писал классик: «Что в имени тебе моем? Оно заобфусцируется, как шум печальный...» (с). Мы с вами начнем с самого простого алгоритма обфускации этих ваших дотнетов — переименования идентификаторов. Дело в том, что в отличии от нативных языков программирования (Цэ, Плюсы, Дэ, Ним и так далее), компиляторы которых при сборке нативного кода пренебрегают сохранением названий классов, методов, аргументов и переменных (за исключением RTTI и отладочной информации, конечно), в мире дотнетов почти все имена сохраняются в таблицах метаданных. Это сделано для удобства отладки и использования скомпилированных библиотек. Сами посудите, если в нативных динамических библиотеках мы официально можем вызывать только функции из таблицы экспорта (в которой имена сохраняются), то из дотнетовских динамических библиотек (и даже экзешников) мы можем использовать

любой публичный метод публичного класса (и даже приватные методы внутренних классов через Reflection). Мелкомягкие давно пытались сделать нечто подобное для своих COM'ов через эти ихние TypeLib'ы, но согласитесь, с дотнетами вышло куда лучше.

Смысл этого алгоритма обфускации в том, чтобы рекурсивно обойти дерево сборки (которое мы обсуждали в самом начале) и переименовать всё, что можно было переименовать (при этом ничего не поломав). Что я подразумеваю под «поломать», вы спросите? Дело в том, что дотнет фреймворк предполагает, что определенные вещи называются определенными именами. Так, например, глобальный класс «<Module>», который есть во всех модулях должен так и называться. Тот самый удивительный класс «<PrivateImplementationDetail>» можно аккуратно попереименовывать, но лучше в это не влезать, так как особого практического смысла в этом нет. Виртуальные методы переопределенные в классе наследника должны называться также, как исходные виртуальные методы в классе родителе. Методы геттеры и сеттеры должны соответствовать имени свойства с префиксами «get_» и «set_» соответственно. И так далее. Если этот набор правил не соблюдать, то есть существенная вероятность того, что сборка будет поломана и перестанет нормально работать.

Теперь давайте подумаем, во что мы можем переименовывать элементы метаданных. Если нам хочется минимизировать размер сборки, то мы можем переименовывать элементы в короткие названия, вплоть до одной буквы, типа класс «A», класс «B» и так далее. Если мы хотим, чтобы у реверсера ломались глаза, то можем переименовывать во что-то типа «lI1lll1lll1» (где название состоит из зрительно очень похожих друг на друга символов «l», «I» и «1»). Если мы хотим, чтобы сборка неотсвечивала после переименования, то можно брать псевдослучайные слова из какого-то word-листа, который можно найти в интернетах. Кроме того, дотнеты поддерживают юникод символы в идентификаторах, поэтому в принципе помимо букв и цифр из Latin-1 мы можем использовать некоторый набор локализованных символов, или даже непечатаемых символов. Но тут нужно быть осторожнее, на старых версиях дотнет фреймворка могут возникать странные баги с этим связанные, в общем, если решите пойти по этому путь, то уделите больше внимания тестированию.

В некоторых случаях при переименовании вещей имеет смысл сохранить некий текстовый файл, в котором будет записаны соответствия исходных имен новым именам для того, чтобы в будущем облегчить себе отладку и тестирование. После того, как вы переименовали исходные идентификаторы в новые все стек-трейсы и другие привычные вещи будут выводиться вам с новыми идентификаторами. Из-за этого иногда бывает очень сложно и муторно искать, где что-то пошло не по вашему плану.

Теперь давайте рассмотрим пример того, как это реализовано в одном из реальных публичных обфускаторов. Для этого я решил взять фрагменты из обфускатора Obfuscar для разнообразия (поскольку он под капотом использует Mono.Cecil) и для простоты (поскольку его реализация выглядит чуть проще, чем у ConfuserEx, который старается рассматривать кучу частных случаев, в большинстве ситуаций ненужных по моему мнению). Собственно переименования происходит в методе «RunRules» класса «Obfuscator»:
C#:

```
public void RunRules()
{
    // The SemanticAttributes of MethodDefinitions have to be loaded before any
fields,properties or events are removed
    LoadMethodSemantics();

    LogOutput("Hiding strings...\n");
    HideStrings();

    LogOutput("Renaming:  fields...");
    RenameFields();

    LogOutput("Parameters...");
    RenameParams();

    LogOutput("Properties...");
    RenameProperties();

    LogOutput("Events...");
    RenameEvents();

    LogOutput("Methods...");
    RenameMethods();

    LogOutput("Types...");
    RenameTypes();

    PostProcessing();

    LogOutput("Done.\n");

    LogOutput("Saving assemblies...");
    SaveAssemblies();
    LogOutput("Done.\n");

    LogOutput("Writing log file...");
    SaveMapping();
    LogOutput("Done.\n");
}
```

После сокрытия строк (вызов метод HideStrings) обфускатор переименовывает все поля, параметры, свойства, события, методы и типы. Возможно, у автора проекта была «какая-то тактика», почему переименовывать имена нужно в этом порядке, «и он ее придерживался». Я же в своих обфускаторах переименование реализую, последовательно проходя древовидную структуру: сборка, затем модуль, затем тип, затем все, что вложено в этот тип, затем перехожу к следующему типу. Мне кажется, такой алгоритм более оптимальным в том плане, что не нужно по несколько раз перечислять типы для каждого отдельного алгоритма. Давайте рассмотрим для примера, как происходит переименование типов в методе RenameTypes.
C#:

```
public void RenameTypes()
{
    //var typerenamemap = new Dictionary<string, string> (); // For patching the parameters
of typeof(xx) attribute constructors
    foreach (AssemblyInfo info in Project.AssemblyList)
    {
        AssemblyDefinition library = info.Definition;

        // make a list of the resources that can be renamed
        List<Resource> resources = new List<Resource>(library.MainModule.Resources.Count);
        resources.AddRange(library.MainModule.Resources);

        var xamlFiles = GetXamlDocuments(library, Project.Settings.AnalyzeXaml);
        var namesInXaml = NamesInXaml(xamlFiles);

        // Save the original names of all types because parent (declaring) types of nested
types may be already renamed.
        // The names are used for the mappings file.
        Dictionary<TypeDefinition, TypeKey> unrenamedTypeKeys =
            info.GetAllTypeDefinitions().ToDictionary(type => type, type => new
TypeKey(type));

        // loop through the types
        int typeIndex = 0;
        foreach (TypeDefinition type in info.GetAllTypeDefinitions())
        {
            if (type.FullName == "<Module>")
                continue;

            if (type.FullName.IndexOf("<PrivateImplementationDetails>{",
StringComparison.Ordinal) >= 0)
                continue;

            TypeKey oldTypeKey = new TypeKey(type);
            TypeKey unrenamedTypeKey = unrenamedTypeKeys[type];
            string fullName = type.FullName;

            string skip;
            if (info.ShouldSkip(unrenamedTypeKey, Project.InheritMap,
Project.Settings.KeepPublicApi,
                Project.Settings.HidePrivateApi, Project.Settings.MarkedOnly, out skip))
            {
                Mapping.UpdateType(oldTypeKey, ObfuscationStatus.Skipped, skip);

                // go through the list of resources, remove ones that would be renamed
                for (int i = 0; i < resources.Count;)
                {
                    Resource res = resources[i];
                    string resName = res.Name;
                    if (Path.GetFileNameWithoutExtension(resName) == fullName)
                    {
```

```
                    resources.RemoveAt(i);
                    Mapping.AddResource(resName, ObfuscationStatus.Skipped, skip);
                }
                else
                {
                    i++;
                }
            }

            continue;
        }

        if (namesInXaml.Contains(type.FullName))
        {
            Mapping.UpdateType(oldTypeKey, ObfuscationStatus.Skipped, "filtered by
BAML");

            // go through the list of resources, remove ones that would be renamed
            for (int i = 0; i < resources.Count;)
            {
                Resource res = resources[i];
                string resName = res.Name;
                if (Path.GetFileNameWithoutExtension(resName) == fullName)
                {
                    resources.RemoveAt(i);
                    Mapping.AddResource(resName, ObfuscationStatus.Skipped, "filtered by
BAML");
                }
                else
                {
                    i++;
                }
            }

            continue;
        }

        string name;
        string ns;
        if (type.IsNested)
        {
            ns = "";
            name =
NameMaker.UniqueNestedTypeName(type.DeclaringType.NestedTypes.IndexOf(type));
        }
        else
        {
            if (Project.Settings.ReuseNames)
            {
                name = NameMaker.UniqueTypeName(typeIndex);
                ns = NameMaker.UniqueNamespace(typeIndex);
```

```
                }
                else
                {
                    name = NameMaker.UniqueName(_uniqueTypeNameIndex);
                    ns = NameMaker.UniqueNamespace(_uniqueTypeNameIndex);
                    _uniqueTypeNameIndex++;
                }
            }

            if (type.GenericParameters.Count > 0)
                name += '`' + type.GenericParameters.Count.ToString();

            if (type.DeclaringType != null)
                ns = ""; // Nested types do not have namespaces

            TypeKey newTypeKey = new TypeKey(info.Name, ns, name);
            typeIndex++;

            FixResouceManager(resources, type, fullName, newTypeKey);

            RenameType(info, type, oldTypeKey, newTypeKey, unrenamedTypeKey);
        }

        foreach (Resource res in resources)
            Mapping.AddResource(res.Name, ObfuscationStatus.Skipped, "no clear new name");

        info.InvalidateCache();
    }
}
```

Дрочь с XAML и ресурсами нам не особо интересна, поскольку это больше свойственно легитимному коду (если хотите, то можете сами в этом покопаться, это не так сложно должно быть). В методе сначала происходит сохранение исходным имен для создания так называемого «mapping»-файла: файла с соответствием оригинальных и сгенерированных имен (чтобы упрощать отладку обфусцированного образца и стек-трейсов исключений). Затем в цикле происходит обработка каждого типа в отдельности. В цикле, как я говорил вам ранее, типы с именами «<Module>» и «<PrivateImplementationDetails>» игнорируются. Дальше происходит проверка каждого типа на то, нужно ли его пропускать или нет. Этот метод реализован в классе AssemblyInfo, он сравнивает характеристики текущего типа с настройками обфускатора, и не переименовывает типы с флагом Runtime и SpecialName. Далее происходит проверка присутствия имени типа в XAML — нахер её. Затем для типа генерируется уникальное имя с помощью класса NameMaker, а переименование типа происходит в методе RenameType. Последняя функция достаточно простая: в Mono.Cecil для изменения имени типа необходимо просто установить значения свойств Name и NameSpace на типах. А вот генерацию уникального имени давайте рассмотрим поподробнее:

C#:

```
static class NameMaker
{
    static string uniqueChars;
    static int numUniqueChars;
    const string defaultChars = "AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz";

    const string unicodeChars = "\u00A0\u1680" +

"\u2000\u2001\u2002\u2003\u2004\u2005\u2006\u2007\u2008\u2009\u200A\u200B\u2010\u2011\u2012\u
2013\u2014\u2015" +

"\u2022\u2024\u2025\u2027\u2028\u2029\u202A\u202B\u202C\u202D\u202E\u202F" +
                                "\u2032\u2035\u2033\u2036\u203E" +

"\u2047\u2048\u2049\u204A\u204B\u204C\u204D\u204E\u204F\u2050\u2051\u2052\u2053\u2054\u2055\u
2056\u2057\u2058\u2059" +
                                "\u205A\u205B\u205C\u205D\u205E\u205F\u2060" +

"\u2061\u2062\u2063\u2064\u206A\u206B\u206C\u206D\u206E\u206F" +
                                "\u3000";

    private static readonly string koreanChars;

    static NameMaker()
    {
        // Fill the char array used for renaming with characters
        // from Hangul (Korean) unicode character set.
        var chars = new List<char>(128);
        var rnd = new Random();
        var startPoint = rnd.Next(0xAC00, 0xD5D0);
        for (int i = startPoint; i < startPoint + 128; i++)
            chars.Add((char) i);

        ShuffleArray(chars, rnd);
        koreanChars = new string(chars.ToArray());
    }

    private static void ShuffleArray<T>(IList<T> list, Random rnd)
    {
        int n = list.Count;
        while (n > 1)
        {
            n--;
            int k = rnd.Next(n + 1);
            (list[n], list[k]) = (list[k], list[n]);
        }
    }

    public static string UniqueChars
    {
        get { return uniqueChars; }
```

```
}

public static string KoreanChars
{
    get { return koreanChars; }
}

public static string UniqueName(int index)
{
    return UniqueName(index, null);
}

public static string UniqueName(int index, string sep)
{
    // optimization for simple case
    if (index < numUniqueChars)
        return uniqueChars[index].ToString();

    Stack<char> stack = new Stack<char>();

    do
    {
        stack.Push(uniqueChars[index % numUniqueChars]);
        if (index < numUniqueChars)
            break;
        index /= numUniqueChars;
    } while (true);

    StringBuilder builder = new StringBuilder();
    builder.Append(stack.Pop());
    while (stack.Count > 0)
    {
        if (sep != null)
            builder.Append(sep);
        builder.Append(stack.Pop());
    }

    return builder.ToString();
}

public static string UniqueNestedTypeName(int index)
{
    return UniqueName(index, null);
}

public static string UniqueTypeName(int index)
{
    return UniqueName(index % numUniqueChars, ".");
}

public static string UniqueNamespace(int index)
```

```
    {
        return UniqueName(index / numUniqueChars, ".");
    }

    internal static void DetermineChars(Settings settings)
    {
        if (!string.IsNullOrWhiteSpace(settings.CustomChars))
        {
            uniqueChars = settings.CustomChars;
        }
        else if (settings.UseUnicodeNames)
        {
            uniqueChars = unicodeChars;
        }
        else if (settings.UseKoreanNames)
        {
            uniqueChars = koreanChars;
        }
        else
        {
            uniqueChars = defaultChars;
        }

        numUniqueChars = uniqueChars.Length;
        string lUnicode = uniqueChars;
        for (int i = 0; i < lUnicode.Length; i++)
        {
            for (int j = i + 1; j < lUnicode.Length; j++)
            {
                System.Diagnostics.Debug.Assert(lUnicode[i] != lUnicode[j], "Duplicate
Char");
            }
        }
    }
}
```

Для генерации имени обфускатор использует один из наборов символов: это могут быть символы английского алфавита (без цифр, так как имя типа по-хорошему не может начинаться с цифры, но для второго и последующих символов в имени текущего типа цифры можно использовать, я думаю, что автор просто не стал запариваться этим), странные символы юникода и символы из корейского языка. Наборы английских и юникод символов захардкожены в виде констант, а корейские символы генерируются один раз в статическом конструкторе типа. Кстати, если вы не знали, в дотнетах статические конструкторы типов вызываются при первом обращении к типу, будь то создание инстанса или вызов статического метода типа. Поэтому, если, например, добавить статический конструктор типу «<Module>», он будет вызван неявно при инициализации модуля в процессе загрузки сборки. Это вполне можно использовать в разных целях, как мы увидим в следующих статьях цикла. Но мы отвлеклись… Класс в

зависимости от настроек обфускатора выбирает массив символов, из которого он будет формировать уникальные имена. Затем в зависимости от порядкового номера имени с помощью вспомогательного стека генерирует имя. Таким образом, первые типы будут, например, иметь имена «A», «a», «B» и тому подобное. Когда алфавита не будет хватать (номер имени выйдет за длину алфавита), то имя будет генерироваться из двух символов и так далее.

Этот метод сравнительно неплохой, когда нам хочется уменьшить объем исполняемого файла сборки, так как имена типов будут короткими (от одного до N-символов в зависимости от количества типов). Единственное, если бы я реализовывал такое, то я бы скорее всего обошелся без стека, записывая символы сразу в StringBuilder. Необходимость дополнительной коллекции в данном случае оставим под вопросом. И да, если нам хочется казаться легитимным исполняемым файлом, то куда лучше использовать для этого реально существующие слова из какого-нибудь word-листа. А использование юникода и корейских символов я бы не советовал, в этом случае сразу становится понятно, что файл скорее всего был обфусцирован, не знаю, насколько современные антивирусы способны это установить в автоматическом режиме, но в теории это может добавить отрицательных баллов исполняемому файлу в статике.

Алгоритмы переименования других имен очень похожи, поэтому вооружившись полученными знаниями, я уверен, что у вас получится их разобрать и без моего участия. После того, как обфускатор прошел все свои «правила» в методе RunRules, он сохраняет модифицированную в памяти сборку библиотекой Mono.Cecil в методе SaveAssemblies, на этом «его полномочия всё — окончены» (c). Obfuscar по функционалу достаточно примитивный обфускатор, но на его примере можно показать отдельные алгоритмы. В следующих статьях мы уже будем рассматривать более «навороченные» (хоть и задроченные) обфусктаторы, которые уже реализованы на базе библиотеки dnlib. По моему мнению, использовать Mono.Cecil для обфускации вполне возможно, но dnlib и AsmResolver предоставляют более удобное API для этого (выбирать вам, но я рекомендую AsmResolver).

На этом, я думаю, с первой частью статьи пора заканчивать, а то я опять вышел за 30к символов, такие статьи становится тяжело читать за раз. Спасибо вам, что дочитали мою статью до конца, я надеюсь, что у меня вышло в достаточно понятном виде преподнести вам необходимую базу для понимания того, как работают дотнеты и обфускаторы для них. В следующих статьях этого цикла мы будем рассматривать более сложные алгоритмы, руководствуясь этой базой. И отдельное спасибо тем ребятам, которые периодически тыкали меня со словами «дядь, когда уже статья по обфускации дотнетов, ты заебал морозить». Простите, ребятки, я всегда рад поделиться знаниями и своим особо важным мнением, но времени и сил на всё не хватает. Надеюсь, что со следующими статьями такого «крокодильева ануса» не выйдет, и ждать их не придется слишком долго.

Эта статья написана специально для вас — для моего любимого уютненького комьюнити XSS.is. ❤️ ❤️ ❤️