نگاهی اجمالی بر مقابله با مهندسی معکوس کدهای Net.

محمد شمس جاوی
http://www.mshams.ir

همانطور که می‌دانید، ساختار فایلهای اجرایی ویندوز از فرمت Microsoft Portable Executable یا PE تبعیت می‌کند. اما این مسئله در مورد فایلهای کامپایل شده تحت فریمورک داتنت کمی متفاوت است.
برای روش شدن این مسئله، مهمترین طبقه‌بندی در روش کامپایل فایلهای اجرایی سیستم عامل ویندوز را تشریح می‌کنیم:

فایلهای PE در سیستم عامل ویندوز به دو شیوه (معماری) کلی کامپایل می‌شوند:

۱. کد مدیریت شده (Managed Code)
۲. کد مدیریت نشده (Unmanaged Code) یا کد ماشین (Machine Code)

کد مدیریت شده (Managed Code)

به کدهای کامپایل شده در زبانهای برنامه نویسی مانند Visual Studio.Net languages: C#, J#, VB.NET, VC++.Net, Java, ActionScript و غیره اطلاق گشته و مهمترین ویژگیهای ساختاری این کدها به شرح زیر هستند:

- این کدها، تحت مدیریت و محافظت یک ماشین مجازی (Virtual Machine) در سیستم عامل اجرا می‌شوند و در واقع سیستم عامل مسئول اجرای ماشین مجازی بوده و ماشین مجازی نیز مسئولیت اجرای کدهای تحت مدیریت خود در سیستم عامل را به عهده دارد.
منظور از ماشین مجازی می‌تواند Microsoft dotNet Framework که مهمترین و مطرح‌ترین ماشین مجازی موجود در سیستم عامل ویندوز است بوده و یا به فریمورکهای دیگری مانند Java Virtual Machine یا Flashplayer نیز اشاره نمود. در واقع کدهای جاوا و اسکریپتهای زبان ActionScript در محیط فلش، تحت مدیریت فریمورکهای خود یعنی JVM و FP اجرا می‌گردند.

- فایلهای اجرایی آنها حاوی کد ماشین نیستند، چرا که پس از کامپایل برنامه‌های نوشته شده در محیط‌های مدیریت شده، فایل اجرایی (نهایی) به زبانی به نام Intermediate Language (IL) یا به اصطلاح زبان میانی تبدیل می‌شوند.

- به دلیل وجود واسطه‌ای به نام فریمورک، کاملا مستقل از سخت افزار (مستقل از سکو) بوده و قابل حمل (Portable) هستند. چرا که مسئولیت اجرای آنها با فریمورک بوده و به هیچ وجه در تعامل مستقیم با سخت افزار سیستم نیستند. در واقع میزان تعامل و حدود دسترسی آنها به منابع سیستم، منوط به امکانات ارائه شده از طرف بستر آنها (فریمورک) جهت کار با منابع است.

کد مدیریت نشده (Unmanaged Code)

به این کدها اصطلاحا کد ماشین (Machine Code) یا کد محلی (Native Code) گفته شده و در واقع همان کدهای کامپایل شده در زبان‌های غیر فریمورک مانند VC++, Delphi, VB6.0 و غیره هستند. مهمترین ویژگیهای ساختاری این کدها به شرح زیر هستند:

- پس از کامپایل، مستقیما به کد ماشین (زبان ماشین) تبدیل می‌شوند. در سیستم عامل ویندوز کدهای ماشین بر مبنای معماری X86 اینتل طراحی شده‌اند.

- در هنگام اجرا، مستقیما و بدون واسطه، توسط سیستم عامل و بر روی CPU اجرا می‌شوند.

ماشین مجازی Microsoft .NET Framework

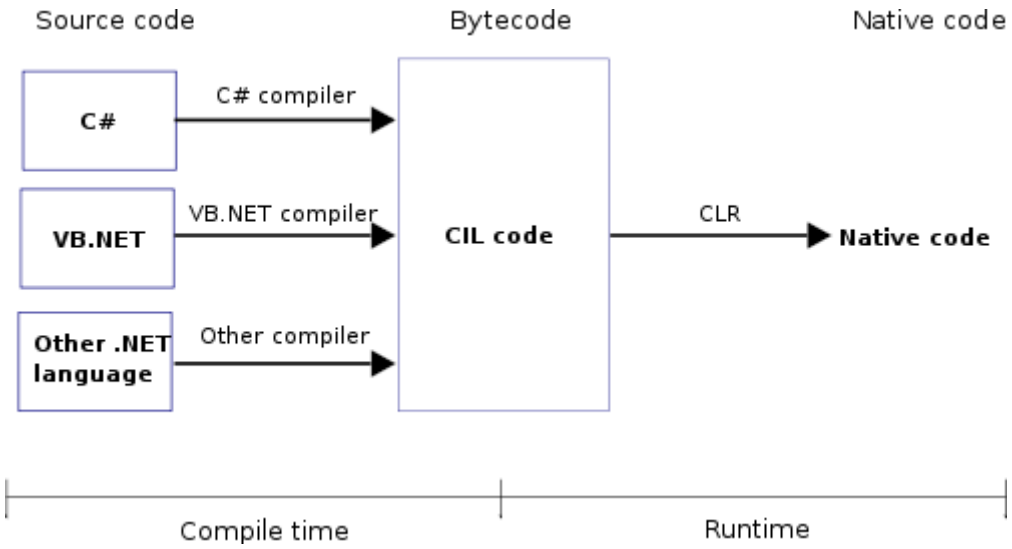
ماشین مجازی Net. حاوی کتابخانه بسیار بزرگی از کدهای از پیش تهیه شده و آماده داتنت است که به منظور تسریع روند برنامه نویسی، به صورت اجزایی به نام Namespace در اختیار برنامه نویسان قرار می‌گیرند. مهمترین بخشهای معماری این ماشین مجازی بزرگ CLR، MSIL و JIT هستند که مختصرا تشریح می‌گردند:

Common Language Runtime (CLR)

در واقع هسته اصلی فریمورک داتنت محسوب می‌شود و تمامی اعمال مهم مدیریتی مانند:

- مدیریت حافظه (Memory management)
- مدیریت نخها (Thread management)
- مدیریت استثناها (Exception handling)
- جمع آوری زباله‌ها (Garbage collection) یا به عبارتی همان آزادسازی حافظه‌های اشغال شده
- امنیت (Security)
- ...

را به عهده دارد. با توجه به شکل ۱ مشاهده می‌کنید که پس از تبدیل کدهای داتنت به کد میانی، این CLR است که وظیفه مدیریت کد میانی جهت اجرا را بر عهده می‌گیرد.



(شکل ۱) CLR در معماری داتنت

لازم به ذکر است که در حال حاضر پروژه‌ای موسوم به Mono به شبیه سازی CLR و یا به عبارتی شبیه سازی بستر داتنت در سیستم عاملهای مبتنی بر Unix و Linux می‌پردازد. با استفاده از امکانات Mono میتوان فایلهای مدیریت شده dotNet را بر روی سکوهای لینوکس نیز اجرا نمود. (البته محدودیتهایی نیز وجود دارد)

Common Intermediate Language (CIL, IL, MSIL)

کد زبان میانی یا IL پایین ترین سطح کدهای موجود در معماری داتنت است. به نوعی همان کد ماشین، اما در فریمورک داتنت محسوب می‌شود.
مهمترین نقطه ضعف کدهای این زبان، قابل فهم بودن آنها توسط انسان است. در واقع یک برنامه‌نویس با تجربه با مطالعه کدهای کامپایل شده به IL و استفاده از Metadata ها، تا حدودی میتواند منطق و روند اجرای برنامه را کشف نماید. به دلیل همین نقطه ضعف عمده، به منظور محافظت از کدهای داتنت در برابر مهندسی معکوس، از انواع ابزارهای محافظ مانند Obfuscator ها استفاده می‌گردد.

Just-in-time compiler (JIT)

کامپایلر بلادرنگ یا Jit کامپایلر، بخشی از CLR محسوب شده که وظیفه کامپایل کدهای زبان میانی (Intermediate Language) به کدهای زبان ماشین (Machine Code) را بر عهده دارد.

برای درک بهتر این موضوع به روند و پروسه اجرای کدها و برنامه‌های فریمورک داتنت توجه نمایید:

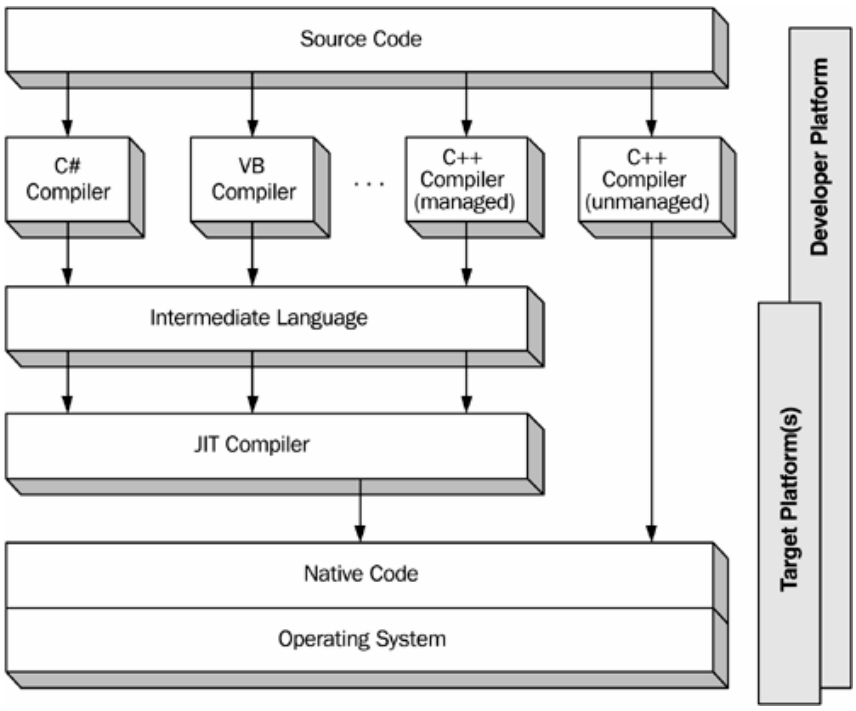
۱. برنامه نویس، برنامه خود را در محیط Visual Studio.Net و در یکی از زبانهای آن مانند VB.Net، C# یا ... می‌نویسد.

۲. برنامه مذکور توسط کامپایلر زبان مربوطه، به کد زبان میانی (IL) کامپایل میشود. به عنوان مثال اگر برنامه با C# نوشته شده باشد، توسط کامپایلر آن به نام CSC.EXE و اگر با VB.Net نوشته شده باشد توسط کامپایلر VBC.EXE به کد میانی کامپایل می‌شود.

۳. کاربر اقدام به اجرای برنامه کامپایل شده (فایل exe) خود می‌نماید.

۴. در این مرحله CLR کار را به دست گرفته و پس از خواندن محتوای فایل مربوطه از دیسک سخت و بارگذاری آن در حافظه، ابتدا تمام پیش‌نیازهای مربوط به آن مانند کتابخانه‌های اشتراکی و غیره را در حافظه بارگذاری نموده و سپس توسط JIT Compiler کد زبان میانی را به کد ماشین (Machine Code) تبدیل و ترجمه می‌نماید.

این عمل ترجمه معمولا تنها یکبار صورت گرفته و پس از آن به دلیل اینکه کل کد میانی به یکباره به کد ماشین تبدیل شده است، در مراحل بعدی اجرای برنامه با سرعت بیشتری انجام می‌گردد.



(شکل ۲) موقعیت کامپایلر JIT در معماری داتنت

تفاوت کدهای مدیریت شده و مدیریت نشده از نظر ساختار PE

حال که مفاهیم CLR و IL تشریح شدند، به وجه تمایز کدهای مدیریت شده و مدیریت نشده از نظر ساختار PE می‌پردازیم.

مهمترین تفاوت فایلهای PE که به صورت کدهای مدیریت شده کامپایل و ترجمه شده‌اند، وجود بخشی اضافه به نام "Net Directory." در سکشنهای آنها است. سکشن "Net Directory" دارای سه بخش اصلی و بسیار مهم با نامهای زیر است:

۱. CLR Header که هدر تشریح کننده عملیات CLR است.

۲. IL code که شامل کد اجرایی برنامه به زبان میانی است.

۳. Metadata که اطلاعاتی اضافه در باره چگونگی تفسیر و اجرای کدهای اجرایی زبان میانی موجود در بخش قبلی (IL code) می‌باشد. در واقع مهمترین نقطه ضعف این معماری از همین قسمت ناشی می‌گردد.

وجود Metadata در فایلهای کامپایل شده تحت فریمورک برای اجرای کدهای داتنت الزامی است، اما مشکل این است که با بررسی Metadata به همراه کدهای IL به راحتی می‌توان عملیات انجام شده در IL را کشف نمود.

معنی این کار اینست که با استفاده از دانش مهندسی معکوس و اتکا بر Metadata های موجود در فایلهای داتنت، می‌توان سورس کد اولیه برنامه را محاسبه و تعیین نمود.

به عنوان مثال در شکل زیر، برنامه نوشته شده در زبان C# و کد معادل (تبدیل شده) آن در زبان IL را مشاهده می‌کنید. با توجه به قابل خواندن بودن کد IL و با استفاده از اطلاعات موجود در Metadata میتوان سورس کد اولیه C# را از آن استخراج نمود.

```
public void Stop()
{ this.flag = false; }
```

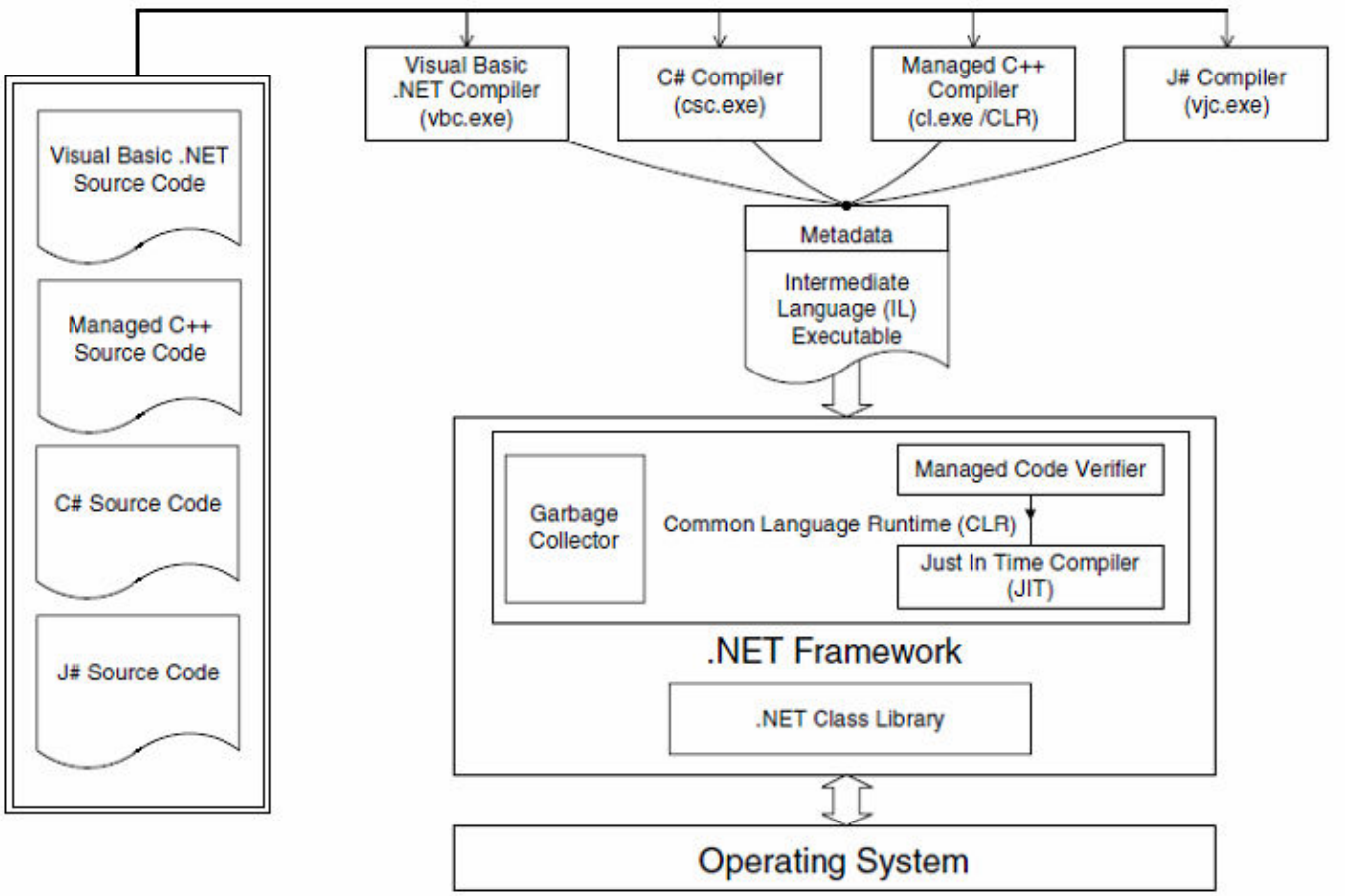
C# Source

```
.method public hidebysig instance void Stop() cil managed
{   .maxstack 8
    L_0000: ldarg.0
    L_0001: ldc.i4.0
    L_0002: volatile.
    L_0004: stfld bool modreq([mscorlib]System.Runtime.Compiler_
            Services.IsVolatile) GTS.TSA::flag
    L_0009: ret   }
```

IL Code

(شکل ۳) برنامه نوشته شده در C# و کد معادل آن در IL

شکل بعدی، معماری dotNet و نحوه و چگونگی برخورد آن با فایلها در زمان اجرا (Runtime) را به خوبی نشان می‌دهد.



(شکل ۴) معماری dotNet Framework و طریقه تعامل در آن

ایمن سازی کدهای dotNet

در این بخش به تشریح سه ایده کلی مورد استفاده برای مقابله با مهندسی معکوس کدهای داتنت می‌پردازیم:

۱. روشهای مبتنی بر Obfuscator ها
۲. روشهای مبتنی بر Compiler ها
۳. روشهای مبتنی بر Linker ها

Obfuscator (پیچیده‌سازها)

مبنای کار آنها، تبدیل کد IL برنامه (Net Assembly.) به یک کد IL معادل ولی پیچیده و غیر قابل فهم است. در واقع عملیات انجام شده در این ابزارها، در ساده‌ترین شکل ممکن به صورت زیر است:

(شکل ۵) مراحل کار Obfuscator

الگوریتم این روال را به این شکل می‌توان تعریف نمود:

[۱]. ابتدا کد زبان میانی (IL) را با استفاده از ابزار ILDASM به کد اسمبلی تبدیل می‌نماییم. این برنامه جزء ابزارهای استاندارد ویژوال استودیو داتنت بوده و در تمامی نسخه‌ها و نگارشهای آن موجود است.

[۲]. با انجام عملیات پیچیده سازی، کد اسمبلی بدست آمده را به یک کد اسمبلی معادل ولی غیر قابل فهم تبدیل می‌نماییم. لازم به ذکر است که در اینجا منظور از کد معادل، کدی است که دقیقا همان کار را انجام می‌دهد، اما از نظر اندازه یا محتوا می‌تواند کمی متفاوت باشد.
و در واقع همینطور بوده و معمولا پس از پایان فرایند پیچیده سازی، حجم کد معادل بدست آمده کمی بیشتر از حجم کد اولیه است. (به دلیل اضافه شدن برخی روتینهای پیچیده سازی).

[۳]. با استفاده از ابزار ILASM کد اسمبلی معادل را به کد زبان میانی (IL) تبدیل می‌نماییم. این برنامه نیز جزء ابزارهای استاندارد ویژوال استودیو داتنت بوده و در تمامی نسخه‌ها و نگارشهای آن موجود است.

طی این فرآیند برنامه کامپایل شده داتنت (فایل اجرایی)، به یک فایل اجرایی معادل (از نظر عملیاتی و کاری که انجام خواهد داد) ولی با محتوایی متفاوت تبدیل می‌گردد.

ابزارهای مورد استفاده

از معروفترین ابزارهای پیچیده‌سازی کدهای داتنت به موارد زیر می‌توان اشاره نمود:

شرکت Xenocode Inc:
ابزار Xenocode Postbuild for .NET

شرکت Remotesoft Inc:
ابزار Salamander .NET Obfuscator

ابزار Salamander .NET Protector

ابزار Salamander .NET Linker and Mini-Deployment Tool

شرکت Wise Owl:

ابزار Demeanor for .NET, Enterprise Edition

شرکت Jungle Creatures, Inc:

ابزار Decompiler.NET

ابزار Deploy.NET

شرکت PreEmptive Solutions:

ابزار Dotfuscator for .NET Professional Edition

شرکت PV Logiciels:

ابزار dotNet Protector

Compiler (مبدلهای IL به Native)

مبنای این روش، صرف نظر کردن از تمام ویژگیها و قابلیتهای مدیریتی CLR در زمان اجرا، و نهایتا تبدیل کد مدیریت شده (Managed Code) به کد مدیریت نشده (Unmanaged Code) است.

در این روش با استفاده از کامپایلرهای خاص تبدیل IL به Native، فایل اجرایی برنامه داتنت به یک فایل اجرایی حاوی کد ماشین تبدیل می‌گردد. مهمترین نکته‌ای که در اینجا می‌بایست مورد توجه قرار بگیرد، خارج شدن برنامه حاصل، از حالت مستقل از سکو (platform-independent) به وابسته به سکو است، چرا که پس از تبدیل فایل اجرایی به Native، دیگر وجود فریمورک در سیستم عامل اهمیتی نداشته و برنامه جهت اجرا، مستقیما توسط سیستم عامل، بر روی CPU اجرا خواهد شد.

Linker (برنامه‌های چسباننده فریمورک به فایل برنامه)

وظیفه این لینکرها، اتصال یا لینک ایستای تمامی بخشهای مورد نیاز از CLR جهت اجرای برنامه، به فایل اجرایی است. با اینکار بخشهایی از کتابخانه CLR که در زمان اجرای برنامه مورد استفاده قرار خواهند گرفت، به صورت ایستا به فایل اجرایی برنامه متصل شده و یک فایل اجرایی مستقل (Stand alone) را تشکیل می‌دهند.

پس از این عمل، می‌توان بر روی کل فایل اجرایی مستقل بدست آمده، به یکباره عملیات Obfuscation یا پیچیده سازی را انجام داد.

البته باید توجه داشت که پس از تشکیل فایل اجرایی مستقل، حجم آن در حدود ۱۵ تا ۳۰ مگابایت افزایش یافته (به دلیل الصاق CLR به آن) و از سرعت اجرای آن نیز مقداری کاسته خواهد شد.

روتینهای مورد استفاده در پیچیده‌سازها (Obfuscators)

Obfuscators ها به عنوان بهترین و کاراترین ابزارهای محافظت از کدهای داتنت، دارای روشهای متنوع و متفاوتی برای پیچیده‌سازی کد برنامه هستند که مهمترین انواع آنها را به پنج دسته اصلی می‌توان تقسیم نمود:

اضافه کردن کدهای خنثی (Dead-Code Insertion)

در این روش با افزودن کدها و دستورات بی تاثیری مانند دستور Nop، یا استفاده متوالی از دستورات Push و Pop، یا اعمال شرطها و حلقه‌های خنثی به کد اولیه، سعی در پیچیده سازی کد اولیه می‌نماییم. بدیهی است که

تشخیص بی اثر بودن برخی کدها در میان انبوه کدهای فایل اولیه، عمل مهندسی معکوس را دشوار می‌سازد.

تغییر و پیچیده‌سازی روند اجرا (Control Flow Obfuscation)

با استفاده از انواع دستورات پرش مثل Jump و Goto روند اجرای برنامه را پیچیده کرده تا دنبال کردن سیر خطی روند اجرا (Trace) دشوار گردد.

تغییر نام موجودیتها (Entity Renaming)

از آنجا که نام تمامی موجودیتهایی مانند Namespace ها، انواع داده‌ای (Data Types)، روتینها، متدها و صفات (Properties) در فایل اجرایی اولیه قابل تشخیص هستند، نام آنها را تغییر می‌دهند.

به عنوان مثال دو تصویر زیر، کد برنامه قبل و بعد از تغییر نام موجودیتها را نشان می‌دهند:

Decompiled executable:

```
private static void Converse(string name1, string name2)
{
   Friendly friendly = new Friendly(name1);
   Friendly friendly2 = new Friendly(name2);
   friendly.SayHello();
   friendly2.SayHello();
   friendly.SayGoodbye(friendly2.Name);
   friendly2.SayGoodbye(friendly.Name);
}
```

```
public void SayGoodbye(string othername)
{
   Console.WriteLine("Goodbye {0}", othername);
}
```

(شکل ۶) کد Decompile شده برنامه قبل از تغییر نام موجودیتها

Decompiled obfuscated executable

```
private static void a(string A_0, string A_1)
{
   a a = new a(A_0);
   a a2 = new a(A_1);
   a.b();
   a2.b();
   a.b(a2.a());
   a2.b(a.a());
}
```

```
public void b(string A_0)
{
   Console.WriteLine("Goodbye {0}", A_0);
}
```

(شکل ۷) کد Decompile شده،برنامه بعد از تغییر نام موجودیتها

کدگذاری رشته‌ها (String Encryption)

با استفاده از انواع متدهای کدگذاری، رشته‌های موجود در برنامه را به رمز در آورده تا در برنامه نهایی به صورت plain قابل دیدن نباشند.

حقه‌های مقابله با دیکامپایلرها (Decompiler Cracher Tricks)

شامل انواع روشهای مورد استفاده جهت ایجاد خلل در کار Decompiler ها هستند. برخی از این روشها حالت عمومی داشته و علیه اکثر Decompiler ها مورد استفاده قرار می‌گیرند، برخی دیگر نیز فقط برای نسخه‌های خاص و معروف نوشته می‌شوند. از جمله این روشها به موراد زیر می‌توان اشاره نمود:

- خاتمه دادن به پروسس Decompiler با جستجوی نام پروسس آن
- تزریق کد به حافظه پروسس
- بستن Handle های حیاتی و مهم پروسسها

منابع:

[1]. The .NET File Format -- Daniel Pistelli 2006

[2]. How-To-Select an Obfuscation Tool for .NET -- Mike Gunderloy 2005

[3]. Compiling MSIL to Native Code -- MSDN Library 2008

[4]. .NET Internals and Code Injection -- Daniel Pistelli 2006

[5]. Rebel.NET Official Guide -- Daniel Pistelli 2006

[6]. NGen Revs Up Your Performance with Powerful Features -- Reid Wilkes 2006

[7]. Native Image Generator (Ngen.exe) -- MSDN Library 2008