Kurapica .Net Reversing TIPS

Introductionto.NETcracking

Reflector

WDSM32 was one of the most popular tools for cracking in the past, after you decide to crack something you disassembled its code in WDSM32 to search for strings like"Registration successful" or "Invalid serial number !" and then you try to nop or inverse the right bytes to get the job done..

在过去，WDSM32是很流行的破解工具.当你开始破解一个软件时，你打开wdsam32,搜索象Registration succesfull或 invalid serial number 之类的字符串，然后你尝试改变流程跳转或是倒转字节以使软件能顺利破解 。

Unfortunately this tool is now history, since all the .net programs are interpreted and not compiled, you can no longer use the old techniques to or tools to crack something written with this new technology.

不幸的是，这个工具已经成为过去了。现在，所有的.net 程序是解释型的而不是编译型的，你无法使用这个古老的技术和工具来破解用这项新技术创作的软件。

All .net applications require the .net runtime [.NETFramework] to be installed on your machine in order to be able to run them, unlike for instance Delphi executables which you can run without any runtime needed to be installed on your pc, the .net runtime works like java virtual machine which is needed to run java executables on your PC.

所有的.net 应用程序需要计算机上安装.net 框架时以便于运行.net 程序。而不是象delphi执行程序，你不需要安装任何运行时支持到你的电脑中。.net 框架工作起来就象java虚拟机，运行java程序时也必须安装java虚拟机到你的电脑中.

Is this good or bad?

这是好还是不好?

This is a good question, as we all know that native code executables run really much faster than interpreted executables, this is a very notable fact when you start ado-nothing .net executable you will see that It takes a little bit more time to show the main form compared to a C++ or Delphi executables which will show the main form faster.

这是个有趣的问题,我们知道，本地代码运行比解释型代码要快很多。这是个明显的事实，当你开始打开.net程序时，它打开主窗口花费的时间和c++或delphi的程序比起来要慢一些。

If speed was the bad point about .net programs so what is good about them when it comes to cracking?

The good point is that a typical .net program source code is compiled into something called IL-code, which stands for"Intermediate Language", something like Java's byte code sand later in run-time the IL-code is compiled into native code by the [JIT compiler], this Just In Time compiler turns the IL bytes into native code that can be run by your CPU, what is important here is a simple fact that IL-code is a higher level code than machine code that we usually dealt with when we targeted native code

programs, meaning ?

当开始破解.net 程序时，速度是一个不好的方面，但是什么是好的一面。优势是一个典型的.net 程序被编译为一种叫做中间语言的代码。就象java字节码。在运行时，由即时编译器编译成本地代码，即时编译器把字节码编译成本地代码，一个重要的事实是，中间语言代码是比机器码更高级的一种语言，不是吗？

This means that you can understand the IL instructions and analyze it more easily than Assembly language that WDSM32 provided us with when we used to disassemble native code executables.

这意味着，你需要理解中间语言代码，相对于过去的汇编语言，你能够更容易地分析它，

Compiler JIT Compiler

.NET Source Code ➔ IL Bytes ➔ Native code

Compiler JIT Compiler

.NET Source Code ➔ IL Bytes ➔ Native code

When we crack a .net executable we will be targeting it in the IL bytes level before it's compiled to native code.

当我们开始破解一个.net执行程序时，我们的目标时在它被编译为本地代码之前的中间语言

Now we will talk about the tool that should replace WDSM32,Actually there are many tools today and the most popular one called "Reflector", This utility is the best available tool that can help us disassemble .net applications and understand what's going in the code, What makes this utility useful is that it can disassemble the application and show us the IL instructions of the code which is more than enough to do a crack, and it can also decompile the IL instructions into other higher level languages like VB.net or C# or Delphi .net !

现在，我开始谈到取代wdsam32的工具，实际上，现在有很多工具，reflector是其中一个很流行的工具。这个实用工具很容易得到，能够很容易地帮助我们理解代码里面是什么。使这个工具很有用的一个原因是，它级反编译程序，并能显示给我们中间语言，这些对破解已经足够了，它同时也能反编译中间语言为其它的高级语言，如vb.net,c#,或delphi.net

This means that you can see the IL instructions or the source code itself with this tool, but when it comes to cracking .net programs you must basically be dependent up on the IL instructions and not on the higher level de compilation of the IL instructions, in most cases "Reflector" will be able to disassemble the program and show you the IL instructions, but it will fail in many cases to decompile the IL instructions into higher level languages and this is simply because the commercial applications industry started to understand that "Reflector" is simply the new WDSM32 for crackers and so they developed many tricks to confuse it and make it unable to decompile the IL instructions into your favorite language like C# or VB.net.

这意味着，你可以用这个工个看到中间语言，和它的源代码，但是，当你破解.net程序时，你还是需要依赖中间语言而不是更高级的语言。大多数的情况是，reflector能够显示给你程序的中间语言，也常常在尝试把中间语言反编译为高级语言时失败，因为商业应用程序开发商理解了reflector就象wdasm32一样，能够反编译执行程序，因此，它们设计了一些诡计而使它不能成功反编译中间语言为你喜欢的语言，如c#, vb.net

As a good cracker you must expect that you will be depending on IL instructions disassembly to analyze the code and crack the program you are targeting.

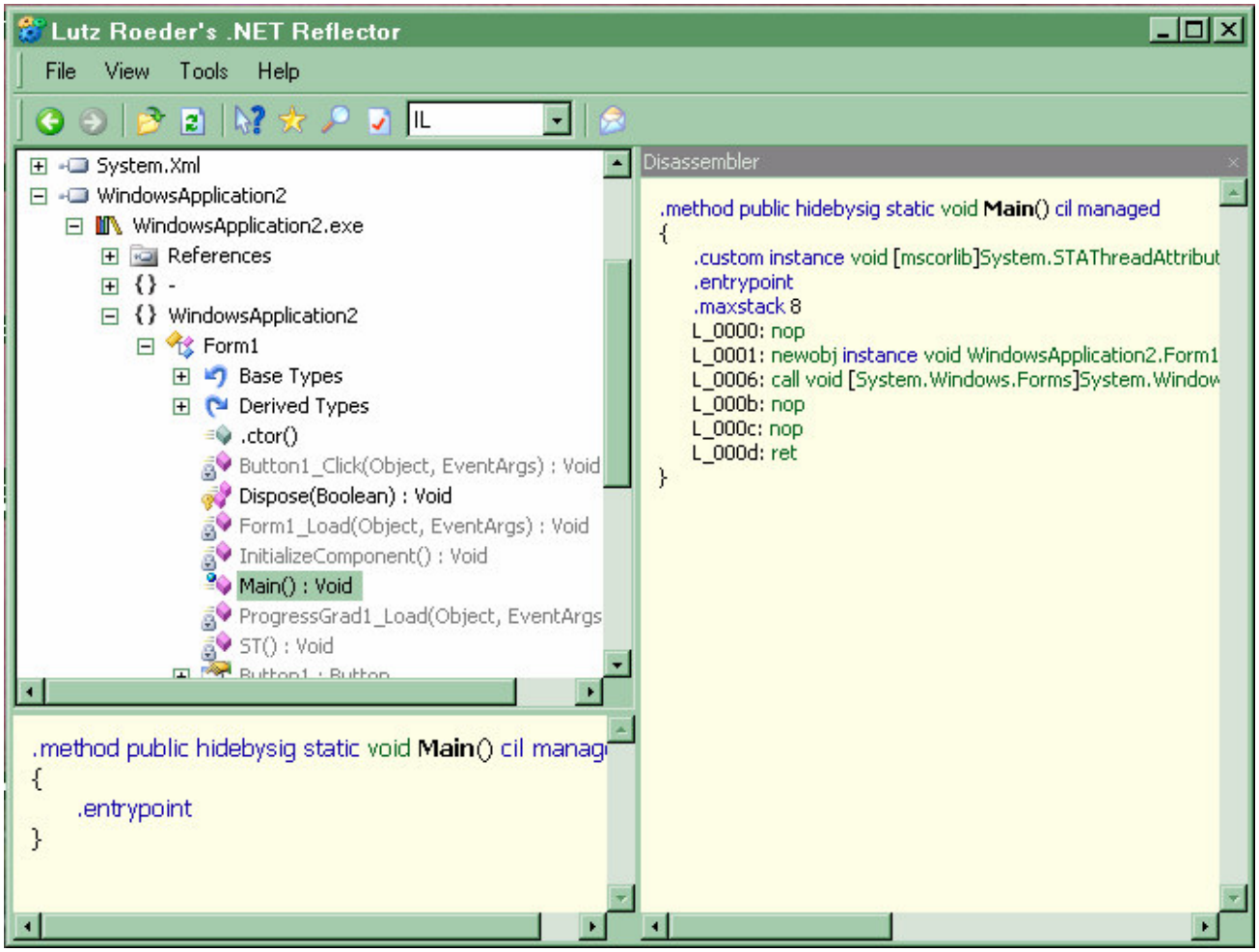
作为一个优秀的cracker,你必须意识到需要依赖中间语言，分析然后破解目标。

Another tool to mentions here is "MSIL Disassembler" which is developed by Microsoft and installed with Visual Studio as a part of the SDK tools, it's an excellent tool to disassemble the .net application and you will be using it side by side with "Reflector" because it does something that "Reflector" can't do.

另一个需要提到的工具是msil disassembler，由微软开发，并作为visual studio的一部分安装在SDK工具中，它也是个很优秀的反编译.net程序的工具，你可以让它与reflector同时运行，因为有些功能是reflector是不能做到的.

"MSIL Disassembler" can disassemble the .net executable just like reflector and it also shows you the "Actualbytes" of these instructions.

Msil disassembler 能反编译.net 执行程序，就象reflector一样，它给显示指令的实际字节

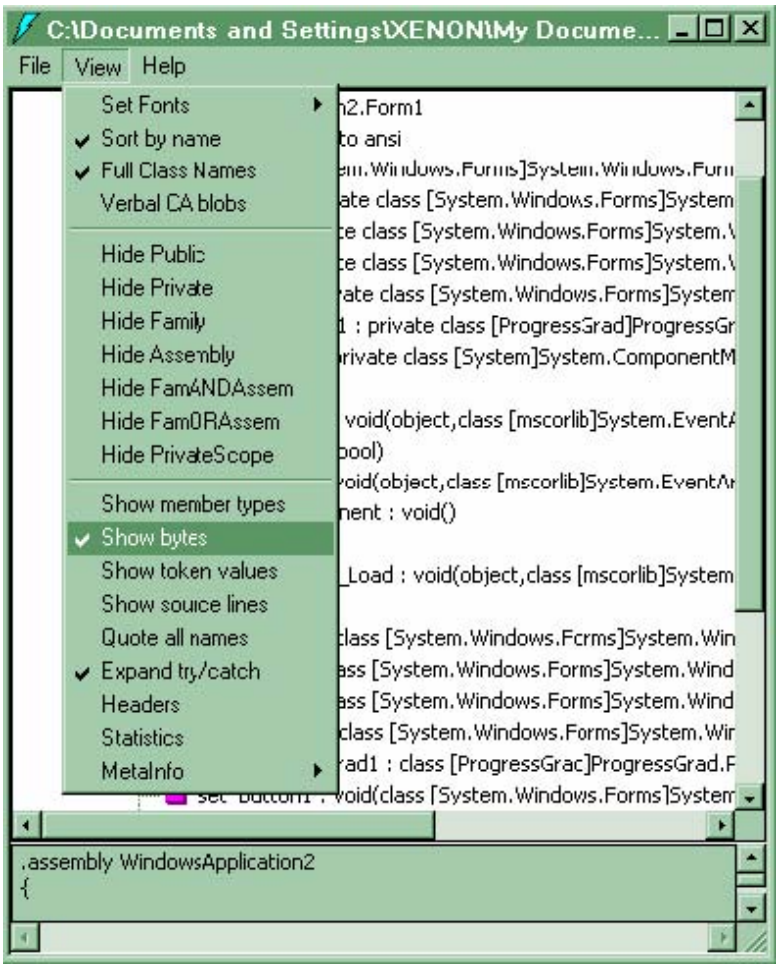


What you see here is reflector and on the right the disassembly of some procedure called Main():Void, You can see the IL instructions in the disassembly here:

当你在reflector中看到的反编译到的Main方法的代码，

Down here you will see "MSIL Disassembler" which can show us the actual bytes of this procedure after we click the"Show bytes" menu.

它能显示实际的字节，当你选中“show bytes”菜单时



Now I will list the disassembly for the same procedure from"MSIL Disassembler"

现在，我列举用msil disassembler反编译的同样的Main过程

```
      .method public hidebysig static void Main() cil managed// SIG: 00 00 01{
  .entrypoint

  .custom instance void [mscorlib]System.STAThreadAttribute::.ctor() =( 01 00 00 00 )
  // Method begins at RVA 0x3a14

  // Code size 14 (0xe)

  .maxstack 8

  IL_0000: /* 00 | */ nop
  IL_0001: /* 73 | (06)000002 */ newobj instance voidWindowsApplication2.Form1::.ctor()
  IL_0006: /* 28 | (0A)000001 */ call
void[System.Windows.Forms]System.Windows.Forms.Application::Run(class[System.Windows.Forms
]System.Windows.Forms.Form)
  IL_000b: /* 00 | */ nop
  IL_000c: /* 00 | */ nop
  IL_000d: /* 2A | */ ret} // end of method Form1::Main
```

How to read this?
如何开始阅读这些指令?

The IL instructions will begin right after the "maxstack #"line, the first line is IL_0000 which contains the instructions "NOP" and its actual bytes are "00" and that's what we will find if we open the program's file in a hexeditor.

IL指令从maxstack后面的一行开始执行，第一行含有指令nop,它实际字节是00,就是当我们用hexeditor打开时看到的

```
IL Line Actual bytes Instructions


IL_0000 00 1 byte nop

IL_0001 7302000006 5 bytes newobj

IL_0006 280100000A 5 bytes all

IL_000b 00 nop

IL_000c 00 nop

IL_000d 2A ret
```

If you open the program in a hex editor then you will find a series of these bytes from first line to last line and that's what I mean by "Actual bytes" of instructions.

当你用hex editor打开一个程序时，你会看到一系列的字节，从第一行到最后一行都是我指出的实际字节指令.

The advantage of these actual bytes is that it tells you what to replace them with when you want to invert some jump or modify any part of the code, to nop a call in the past times we used to do that by replacing all its actual bytes of that call by 90s in a hex editor, inverting a jump was done by replacing the JNE instruction byte with JE instruction byte. 75 to 74 or 85 to 84 and so on.

实际字节码的好处是,当你需要传换流程跳转或是修改代码的一部分时,它很容易地告诉你那些需要替换,用nop指令(空指令)替换所有的实际字节。

Another advantage is that it helps you locate the actual offset for these bytes in a hex editor, for example if you decide to nop the call in line IL_0006 then you will have to locate the file offset for these bytes, All you have to do is to open the file in a hex editor and locate the series of bytes you are looking for which are

另一个优势是，在hex editor中它能帮助你很容易地这位实际的字节位移。例如，你想用nop指令替换IL_006的字节位移，你所要做的就是在hex editor中打开它，然后定位到一系列诸如下面的字节：

Values 00 7302000006 280100000A
Bytes 15 5

And usually about 8 bytes are enough to locate these bytes in your hex editor, after you find the right offset you will have to replace 280100000A with 0000000000

Every IL instruction has a specific actual byte and they are very important in cracking too, for example in native code the nop instruction is expresses by one byte of value0x90 but in .net Assembly the nop instruction is 0x00 byte,I will list the IL instructions here with their function and the actual bytes too.

通常，关于8字节以经足够定位到那里。当你找到正确的偏移位置时，用00000000取代280100000A就可以

每一条中间指令都有一个特殊的字节，它们对于顺利破解程序很重要。例如，本地代码中，nop指令表示为值0x90,但是在.net程序集中，nop指令表示为0xoo. 这里，我列出中间语言和它们的功能以及实际

的字节.

THE END

I hope this was a good tutorial and that you enjoyed reading it...In next tutorials we will see how to replace bytes to invert jump or nop calls.That's all for now....

Kurapica Wednesday, December 27, 2006
我希望这是一篇很好的教程，您也非常喜欢阅读.在接下来的的学习中，我们将看到怎样跳转流程，和空指令调用.

名词
Assembly 程序集 nop 空指令