

Decompiler Round-Up

Regenerating your code

Precently, there has been much debate over open source software. But what most developers overlook is that even though they don't release their source, it may still be available. In the general scheme of things, there is always someone out there who will, by some means, be able to regenerate your code if they so desire. Whether it is by analyzing how your application works and replicating its behavior, or by reverse engineering with a tool, it will always be possible for someone to produce the same output with an appropriate amount of time in which to do it.

With .NET, assemblies compile to an Intermediate Language (IL), which allows them to be executed on any system supporting Microsoft's Common Language Runtime (CLR). There are many benefits to this solution that are outside the scope of this article, but a major drawback is the extent to which your code is compiled. It is first compiled into the assembly, then, upon runtime, it is compiled again. The IL to which your assembly is compiled is not dependent on any machine or software aspects excluding the Just In Time (JIT) compiler. It compiles your code the second time into native machine code, then executes it. Your assemblies also include metadata, which describes the types in your code as well as a few other things. For these reasons, .NET assemblies are relatively insecure to decompilers.

The .NET framework also ships with a tool to convert your compiled assembly into plain text MSIL, which looks like a high-level assembly language. This alone could be enough for most skilled

programmers to see what is going on in your program.

Converting this IL back into its original language is another story. Currently, there are few tools suited for this task. I will be focusing on the three major professional solutions I have found, but in addition to these, there is a free tool as well as an open source tool, which has been discontinued (references at the end of the article). I chose to leave these two tools out of this article. Why? Well, for the same reason I don't do my own electrical work: if I hire someone, I can hold him/her liable for faulty work, whereas if I burn my house down I've got no one to blame but myself. The same goes for software: when it doesn't work, the publishers should fix it; if not, demand your money back. With

ment, as they do not offer the needed level of support. I included a code sample of Reflector.NET to demonstrate this. The tools I did evaluate include Remotesoft's Salamander, 9Rays' Spices.NET, and Jungle Creature's Decompiler.NET.

To evaluate each piece of software, I first decided to look at the basics, such as keywords, modifiers, and so on. Then, I created some complex loops, structures, and inheritance examples. Next, I wanted to see how each could handle pointers, because they are my favorite aspect of programming. And finally, I picked a random control that I did not create or have access to the source. I evaluated solely on runtime behavior. That control was HyperCoder's FileSystemControls, which allows you to mimic Windows Explorer.



BY ROBERT STANTON

“Think your code is safe?
Think again.”
—Remotesoft

free or open source software, you don't have that kind of leverage. Second, in most cases, the authors create their free software in their spare time, which isn't much time at all. The now-discontinued open source software was closed due to lack of time. The other tool (Reflector for .NET), however, is doing an extremely good job, but is unable to reproduce some complex pointer statements. Don't get me wrong, these are fantastic tools that are worth checking out, but personally, I would not rely on them in a professional environ-

Salamander features the familiar class browser interface and can examine managed C++ code as well. I was impressed by its robustness; because it uses an explorer with plug-in support you can perform many tasks in a single program. It also has the ability to generate Visual Studio Project files, which is a nice plus that none of the other decompilers feature. I was disappointed when I tried to decompile my complex pointer examples. In my pointer arithmetic example, the offset was not divided by the type that would lead to a



HOME



CLIENT



SERVER





	Spices.Net	Decompiler.NET	Salamander
Supports All Modifiers	Most	X	X
Supports All Keywords	Most	X	X
Complex Nested Statements	X	X	Some
Pointers	Does not mark as unsafe	X	X
Pointer Member Access		X	
PDB		X	X
Optimizes Code		X	All but pointers
COM Interop		X	X
FileSystemControls	Very Little	No compiler errors, Demonstrated correct run time behavior	Few compiler errors, but way off run time behavior

Table 1: Features at a glance

nice runtime bug to squash. Complex nested statements gave the decompiler some trouble as well. Switches decompile to if-then-elses with correct runtime behavior. Overall, there were only a few goto statements, but they popped up here and there.

9Rays' Spices.NET offers a slick class browser interface and many options. Spices.NET also uses its own reflector for decompiling, giving it the most freedom to get the job done. Sadly, some very crucial elements are missing. For instance, pointers resolve but the methods are not marked as unsafe. In order to compile, you must manually fix these methods. Also, there is no support for stackallocate or the volatile keyword. Switches left lots of goto statements in code, which reminds me of good old spaghetti coding.

Jungle Creature's Decompiler.NET features a simple and easy-to-use interface. It was by far the easiest out of which to get source files. The most impressive feature was its ability to handle pointers. It handled all complex pointer examples, as well as pointer member access (->), which no other decompilers could handle. Switch statements decompiled flaw-

lessly. Overall, there were virtually no goto statements.

Both Salamander and Decompiler.NET featured the ability to look at the programmer database (*.pdb file when compiled in debug mode) and resolve exact variable names. This is extremely useful if you happen to lose your source yet still have a debug lying around. This has actually happened to me once. I had a hard drive failure but had already sent the debug build to the host. If I had known of these products, I would have saved a lot of time.

Listing 1 is a sample of the code I tested. The goal of this code is to run through an array with a pointer. The output should be all zeros because there are no values.

Salamander had trouble with the pointer arithmetic and declared the pointer as a reference variable. Spices.NET was along the same lines. Neither compiled. Reflector.NET has some interesting output. It did compile, but produced way off output. This is where using a free tool can get frustrating. The code compiles, but can leave a nasty bug that is hard to trace down, especially because it may not be your code.

Salamander has stated that they are working on a new version to support better pointer operations. Spices.NET also has plans, but they are currently working on a few projects. Decompiler.NET was the only application to consistently produce compilable correct code, even in this case. Table 1 compares these features.

► How to Prevent

These decompilers demonstrate amazing ability, which might leave you afraid to develop in .NET. But with .NET, there is always another solution. The best way, in this case, is to obfuscate your code. When you do this, you make it unreadable from a person's perspective. For example, getXML(string path) could obfuscate to aaa(string aaa), which gives no meaning to a person. Because the computer sees no symbolic difference between getXML and aaa, both compile and run correctly. Salamander has the most powerful obfuscator, as it will take your compiled assembly and output an exe, which in some cases, would not need the .NET framework to run. It had trouble decompiling some examples and it was a



AUTHOR BIO:

Robert Stanton is a developer for Realized Solutions (www.realizedsolutions.com) in Bristol, CT. He has been developing for 8 years.

▶ rstanton@myrsi.com

bit buggy, but it features enough options to toggle to always get an obfuscated release. I then tried to decompile this release with Jungle Creature's Decompiler.NET. It was successful when the fail-safe options were set, but the code was very difficult to follow. Jungle Creatures built-in obfuscator was a bit disappointing at

▶ Conclusion

All products offer a free trial of some sort, and they are all worth the time to evaluate on your own to see which best fits your needs. For me, Jungle Creature's Decompiler.NET offers tremendous abilities, especially with pointers. Its GUI is simple and easy to use, but it lets you do what needs to be done. More

obfuscator. Spices.NET is the cheapest of the bunch, at \$292.95, which reflects the code it generates. If you only need the simplest of decompilations, Reflector.NET or Spices.NET can do the job. But the results from a commercial product that you can send your requests to for an immediate fix make the investment worthwhile. Trying to track down a bug in

“Obfuscation optimizes code while protecting it from prying eyes.”

first, mainly because it takes an assembly and decompiles it into obfuscated source code. Upon first glance, I saw this as a major vulnerability, especially after the power of Salamander's obfuscator. But the code was extremely difficult to follow. I later liked the ability to modify the source and verify what my code would look like, then modify if necessary and recompile. It would have been nice if the obfuscator also outputted an assembly similar to Salamander.

importantly, it can handle whatever you throw at it. Salamander is close, but still needs some optimizations. Overall, I was a bit disappointed by Spices.NET's performance, but its future looks promising. It has a strong foundation on which to build. Jungle Creature's Decompiler.NET with obfuscator is available for \$500 per CPU, giving it leverage over its toughest competitor, Salamander, which sells for \$1099, plus an additional \$799 if you want the

decompiled source code that you didn't write will end up costing you even more than Salamander with the obfuscator.

▶ Resources

- *Decompiler.NET*: www.junglecreatures.com
- *Salamander*: www.remotesoft.com
- *Spices.NET*: www.9rays.net
- *Reflector.NET*: www.aisto.com/roeder/dotnet
- *Open Source Decompiler*: www.saurik.com/net/exemplar ●

Listing 1: Running through an array with a pointer

```
Original
[STAThread]
public unsafe static void Main() {
    int [] a = new int[10];
    fixed(int* p = a)
    for (int i= 0; i< 10; i++) {
        Console.WriteLine(*(p + i) );
    }
}

9Rays
[STAThreadAttribute()]
public static void Main() {
    ref int p;
    fixed (p = &new int[10][0]) {
        for (int i = 0; i < 10; i++) {
            Console.WriteLine(*(IntPtr)p + (IntPtr)i * 4));
        }
    }
}

Lutz Reflector.NET
[STAThread]
public static void Main()
{
    int num1;
    int[] numArray1 = new int[10];
    fixed (int* local1 = numArray1)
    {
        for (num1 = 0; (num1 < 10); num1 += 1)
        {
            Console.WriteLine(*(local1 +
(num1 * 4)));
        }
    }
}

Output
0,0,0,2042847508,65536,0,77937960,65793,65001

Remotesoft
[STAThreadAttribute()]
public static void Main()
{
    ref int objRef;
    fixed (objRef = &new int[10][0])
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine((int)(IntPtr)objRef + i * 4);
        }
    }
}

Decompiler.NET
[System.STAThreadAttribute()]
public static unsafe void Main ()
{
    int[] a = new int[10];
    fixed (int* p = a)
    {
        for (int i = 0; (i < 10); i++)
        {
            Console.WriteLine (p[i]);
        }
    }
}
```



HOME



CLIENT



SERVER

