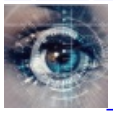
数据结构在.net加密解密反流程混淆中的应用[看雪学院2006金秋读书季]

转载

zfrong 于 2009-01-09 10:48:00 发布 1223 收藏
分类专栏： 防火墙、安全 Win32/MASM/C++ OpenSource开源/分享 C#.Net/ASP.Net/ADO.Net 文章标签： 数据结构 .net 读书 解密 加密 编译器


防火墙、安全 同时被 3 个专栏收录
7 篇文章 0 订阅
订阅专栏


Win32/MASM/C++
14 篇文章 0 订阅
订阅专栏


OpenSource开源/分享
1 篇文章 0 订阅
订阅专栏

标 题: 数据结构在.net反流程混淆中的应用[看雪学院2006金秋读书季]
作 者: tankaiha
时 间: 2006-11-07 12:52
链 接: http://bbs.pediy.com/showthread.php?threadid=34505
详细信息:

数据结构是计算机专业的必修课，但抽象的概念有时让人觉得它难以运用，也有人认为它太基础而不去重视。下面就来看看数据结构在解决实际问题中的作用。

附件下载。

问题描述：

.net平台下的一种软件保护方式叫流程混淆，类似win32下的花指令，主要功能是改变程序流程，添加垃圾代码，增大分析难度。.net中的许 多反编译软件可以直接将源代码解码成高级语言（如C#、VB.net）格式，而流程混淆可以使解码失败，或者解码出的结果错误百出。

先看一个被混淆的crackme的代码片段，Reflector直接解码为C#，输出如下：

```
private void x85601834555fb7d5()
{
    this.x3d9c937c1f3cf311 = new TextBox();
    if (-1 != 0)
    {
    Label_0652:
        this.x6020c4e7a1cc0f6b = new TextBox();
        this.xf5622c25220a6c23 = new Label();
        if (0 != 0)
        {
            goto Label_0490;
        }
```

```
        }
        this.x9001f8afc870fc4c = new Label();
        if (0 != 0)
        {
            goto Label_0342;
        }
        if (0xff != 0)
        {
            if (0 == 0)
            {
                do
                {
                    if (0 != 0)
                    {
                        goto Label_0113;
                    }
                    if (0 != 0)
                    {
                        goto Label_02BF;
                    }
                    if (-2147483648 == 0)
                    {
                        goto Label_05D3;
                    }
                    this.xde320a064856d64c = new Button();
                    this.x1ae679ea6e03596b = new Button();
                }
                while (0 != 0);
            }
        }
        else
        {
            goto Label_0652;
        }
        if (0x7fffffff != 0)
        {
            goto Label_05F2;
        }
        goto Label_0578;
    }
```

问题分析：

看一下被混淆方法的IL代码会更清楚原理，IL代码片段如下：

```
L_0668: br.s L_0634
L_066a: ldc.i4.0
L_066b: brfalse.s L_0624
L_066d: ldc.i4 2147483647   //恒成立的跳转
L_0672: brtrue L_05f2
L_0677: br L_0578   //直接跳转
L_067c: br L_0015   //直接跳转
L_0681: ret
```

代码中被添加了很多br（跳转），以及恒成立的判断跳转。这样，程序就被拆分成很多小块，执行时跳来跳去，造成静态分析的困难。试想一个上万行的函数，被拆分成数百个小块，执行时前后跳转，静态分析还如何进行？

解决方法：

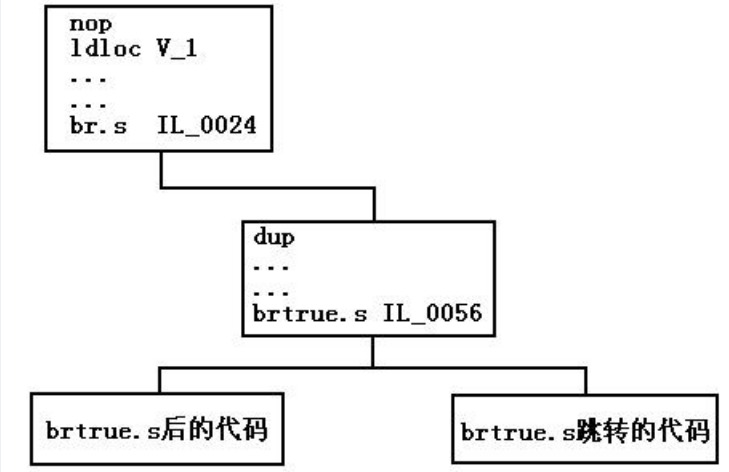
怎么反流程混淆，很自然地，我们想到将br进行连接，将恒成立的跳转去除。（不同的流程混淆软件有不同的特征，注意收集。）对于代码很多的方法， 手工进行处理是不现实的，因此必须编程实现。一个简单的反混淆器，无须用到编译原理的知识，我们需要的是数据结构，下面就介绍下怎么将数据结构的知识用在 反混淆器的编写上。

树的构造：

分析一下IL代码，我们可以将其分为三大类（暂不考虑switch指令）：不跳转（继续运行），判断跳转（如brtrue,brfalse）和直接跳转（br和br.s）。

```
public enum JumpType
{
    NoJump,
    BooleanJump,
    DirectJump
}
```

很自然地，我们想到了二叉树。每个结点代表一条（和多条）指令，对于直接跳转和非跳转指令，该结点只有一个子结点；对于判断跳转，每个结点有两个子结点，分别代表跳转目标和下一条指令。基本构造如下：



程序中，我们可以为结点构造如下的类：

```
public class ILNode
{
    public ILNode();

    public List<ILLine> ilblk;   //代码块
    public ILNode left;   //指向左结点
    public ILNode right;   //指向右结点
}
```

程序的实现：

附件中给出我编写的简单反混淆器，可以用Reflector直接查看源码。程序工作流程：

1.产生指令表：既产生一张包含所有IL指令的表，用于读入源代码时进行比较。表中每个IL指令的格式用一个结构表示。

```
public struct OpCodeTypeInfo
```

```
public struct OpCodeTypeInfo
{
    public string name;      //指令名称
    public JumpType jumptype;   //跳转类型
    public bool hasoprand;      //是否有操作数
    public OpCodeTypeInfo(string n, JumpType jtp, bool oprand);//初始化方法
}
```

2.读入所有的源代码。（由于没有实现完整的词法，语法和语义分析，因此只读入标准的ildasm反编译代码，且必须包含行号。）

```
private static bool ReadSource(string src)
```

每一行源代码，用一个结构来表示：

```
public struct ILLine
{
    public int lable;      //行号
    public string opcode;   //指令
    public string oprand;   //操作数
    public int size;      //大小（用于计算行号，未使用）
    public JumpType jumpType;   //（代码的类型）
}
```

3.构建二叉树：将有读入的源码进行树构造，结果是产生如上图的二叉树，所有的代码都在树结点中。

```
private static bool MakeBTree()
private static ILNode ReadSrcIntoNode(int idx)
```

在这一步中已经完成了对恒成立判断跳转的处理。

4.对树结点进行优化：主要是边接br块。

```
private static bool SecondOptimise(ILNode ilnode)
```

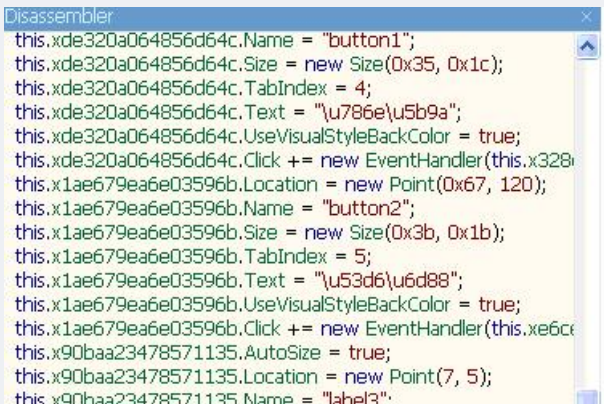
5.输出反混淆结果：按 一定顺序遍历二叉树，生成代码。这是这种反混淆方法的通用之处所在，也是其弱点。编译器将高级语言编译成IL时，通常有一定的顺序。如果要完全还原源程 序，必须对不同编译器生成的不同代码块有所了解（比如if、while的编码方法），而我们直接按照先右子树，后左子树的顺序进行输出，既无法完全还原源 代码，但在一定程度上又做到了通用，因为无论是MS还是别的编译器生成的IL代码，都可以解决。

```
private static string OutputTree(ILNode ilnode)
```

反混淆结果：

使用时，将某个方法的IL代码全部复制到DeFlowOb.exe中，输出后将反混淆代码覆盖源程序。在SDK中用ilasm编译。

用Reflector载入附件中的crackme，分别对比反混淆前后的x85601834555fb7d5方法，可以看到效果。



```
Label_0113:
    base.Controls.Add(this.x7e52e8715f121f0d);
Label_01D3:
    if (0 == 0)
    {
        goto Label_0283;
    }
Label_025F:
    if (0xff != 0)
    {
        this.x90baa23478571135.Location = new Point(7, 5);
        this.x90baa23478571135.Name = "label3";
        this.x90baa23478571135.Size = new Size(0x9b, 12);
        if (1 == 0)
        {
            goto Label_03CA;
        }
        this.x90baa23478571135.TabIndex = 6;
```

```
this.x90baa23478571135.Size = new Size(0x9b, 12);
this.x90baa23478571135.TabIndex = 6;
this.x90baa23478571135.Text = "Crackme No.1 By Inraining";
this.x7e52e8715f121f0d.Location = new Point(0xb7, 0x77);
this.x7e52e8715f121f0d.Name = "button3";
this.x7e52e8715f121f0d.Size = new Size(0x68, 0x1b);
this.x7e52e8715f121f0d.TabIndex = 7;
this.x7e52e8715f121f0d.Text = "\u5220\u9664\u6ce8\u518c\
this.x7e52e8715f121f0d.UseVisualStyleBackColor = true;
this.x7e52e8715f121f0d.Click += new EventHandler(this.x1a51
base.AutoScaleDimensions = new SizeF(6f, 12f);
base.AutoScaleMode = AutoScaleMode.Font;
base.ClientSize = new Size(0x124, 0x9d);
base.Controls.Add(this.x7e52e8715f121f0d);
base.Controls.Add(this.x90baa23478571135);
base.Controls.Add(this.x1ae679ea6e03596b);
base.Controls.Add(this.xde320a064856d64c);
```

小结：

这个小程序还有很多可扩展的地方，比如加入try-catch-finally块的处理（同样可以利用树，不过不是二叉树），加入特定编译块的处理（比如判断while(){}块）。

数据结构在程序开发中的运用很广泛，在逆向工程中也是一样。只要多想，便可以学有所用。抛砖引玉，希望更多人将自己的知识灵活运用，编写出更好的工具。