A Simple and Practical Approach to Input Validation

David Soldera
[daves@ngssoftware.com]
19th June 2007

An NGSSoftware Insight Security Research (NISR) Publication

http://www.ngssoftware.com

1 Introduction

Input validation is the process of ensuring the input into software conforms to what the internal logic of the software expects. According to NIST (http://nvd.nist.gov/statistics.cfm), in 2006 there were 4765 security vulnerabilities due to poor input validation, approximately 72% of all security vulnerabilities discovered that year. These vulnerabilities ranged from denial of service to gaining remote administrator access. Poor input validation is the leading cause of software security vulnerabilities.

Input validation, in theory, is not a difficult problem to solve; it is however a difficult problem to get developers to prioritise security (with regards to other development pressures) and put the time and effort into following good security practice when validating input. The problem is two-fold; most developers implement a degree of input validation but this is generally not sufficient, and there is a lack of knowledge about the possible security related consequences of poor input validation. It is arguable that the situation is improving but the statistics still make for grim reading.

Clearly there is a lot of education that needs to happen to highlight the dangers of poor input validation, but equally important is making input validation part of the software development lifecycle. This means introducing a process which developers can follow to implement input validation. Ideally this process should make the developers life easier and it is the purpose of this paper to offer a solution that requires less code than usual input validation, will save time, is consistent across the entire application, is programming language independent and simplifies auditing.

The simple and practical process outlined in this paper involves converting all user input into an XML representation and then validating that XML against an XML schema defining all the restrictions on the input data. The process outlined here is generic and using XML and XML schema is one possible way of implementing the process, however based on the simplicity of XML and the numerous tools and APIs available in most programming languages to handle XML, the choice is a practical one.

Some background on XML schema will be given, followed by a generic description of the how the process of input validation can be achieved. Following this the process will be described using XML and XML schema and practical examples given including existing tools available that can aid implementation. The various advantages and limitations of the process will also be discussed.

2 Background

2.1 XML Schema

Extensible Markup Language (XML) is a simple and flexible, textual representation of data that was designed to be easily readable by humans and used for transporting data around the Internet. Common uses of XML include web publishing, web searching, web services, documentation, databases etc. An XML Schema Definition (XSD) describes the types and structure that an XML document must conform too. XML can be validated

against an XSD to confirm it is a valid instance of the XSD. The situation is analogous to object oriented programming where you have a class that describes an object, and an object that is an instance of that class.

An XSD is itself written in XML and conforms to the base schema definition (http://www.w3.org/2001/XMLSchema.xsd) which conforms to a schema definition that is itself. This sounds confusing but the point is we don't have to rely on any other data definition languages to define the syntax of an XSD, it defines itself.

Below is an example of an XSD, which will be referred to as Author.xsd.

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Author">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Name" type="xs:string"></xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Note the type of the name is 'xs:string', this means the type 'string' is defined in the XSD with prefix 'xs', which is defined by 'xmlns:xs=http://www.w3.org/2001/XMLSchema', and represents the base XSD. The 'xmlns' stands for XML namespace, however a discussion of namespaces is beyond the scope of this introduction.

An XML instance of Author.xsd could be:

```
<?xml version="1.0" encoding="utf-8"?>
<Author>
  <Name>David Soldera</Name>
</Author>
```

An excellent tutorial on XML schema can be found at http://www.w3schools.com/schema/default.asp.

2.2 XML Schema Data Types

There are numerous built-in data types that can be used in an XSD, however only ones with obvious analogies to typical programming data types will be presented here (somewhat grouped).

- xs:string
- xs:boolean
- xs:base64Binary, xs:hexBinary
- xs:float, xs:double
- xs:long, xs:int, xs:short, xs:byte
- xs:unsignedLong, xs:unsignedInt, xs:unsignedShort, xs:unsignedByte

All the built-in types have been prefixed by 'xs' to indicate they are defined in the XSD schema definition, http://www.w3.org/2001/XMLSchema.xsd. The types presented here are examples of what are called 'simple types'.

2.3 XML Schema Structures

XSD uses 'complex types' to create data structures to represent information. It is convenient to think of complex types as containing simple types and other complex types, and simple types as containing data. So from our previous example 'Author' is a complex type as it contains a simple type, and 'Name' is a simple type as it contains actual data.

Two common ways in which a complex type represents the structure of the data it contains is as a sequence or a choice. A sequence simply means that types listed as part of the complex type must appear in the sequence specified for that complex type.

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Author">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Firstname" type="xs:string"></xs:element>
        <xs:element name="Lastname" type="xs:string"></xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The above XSD restricts any XML document that wishes to conform to it, to provide the name of the author by the sequence of first name and then last name.

A complex type that uses the choice structure simple means that only one of the possible types listed can be part of that complex type. For instance

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Author">
    <xs:complexType>
      <xs:choice>
        <xs:element name="RealName" type="xs:string"></xs:element>
        <xs:element name="Pseudonym" type="xs:string"></xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The above XSD restricts any XML document that wishes to conform to it, to provide either the real name or the pseudonym of the author, but not both.

The power of XSD to define complex data structures comes from the fact that complex types can contain sequences of sequences, or sequences of choices, or choices of sequences etc, to any depth. New types can also be defined and reused, and an XSD can import other XSDs. Combine this with simple types that can occur numerous times and it

is easy to see that XSD provides a very powerful mechanism for describing both simple and very complex data structures.

2.4 XML Schema Restrictions

The simple types presented so far do not give us the type of granularity of restrictions that we might require in order to perform input validation. An XSD allows for simple type to be restricted in order to give this granularity, possible restrictions include:

- Minimum, maximum or specific length
- Enumerations
- Minimum or maximum, inclusive or exclusive values
- Number of digits or number of fractional digits
- Regular expression

Some restrictions only apply to some data types, for instance an xs:int can have a maximum value, but an xs:string cannot. An example of a Filename element that is restricted in length to 260 characters is given below. Note how the restriction is applied to an already existing type, in this case the xs:string type.

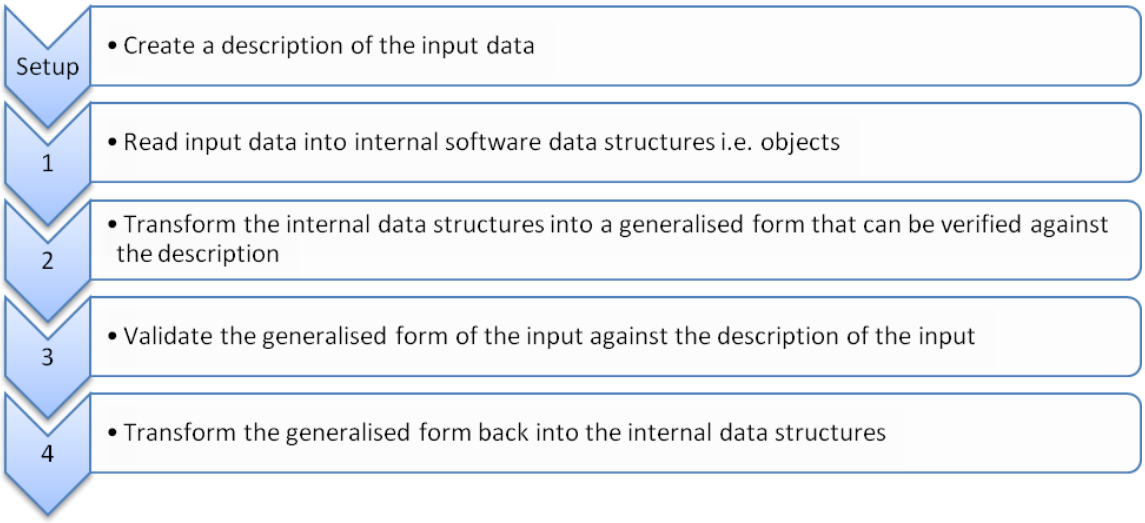
```
<xs:element name="Filename">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:maxLength value="260" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

3 Using XML Schema for Input Validation

We have seen how an XSD can be used to describe complex data types and enable fine grain control over the data that can appear in a conforming XML document. Now we describe the process by which we can leverage this validation to virtually any situation where we need to perform input validation.

3.1 The Process of Input Validation

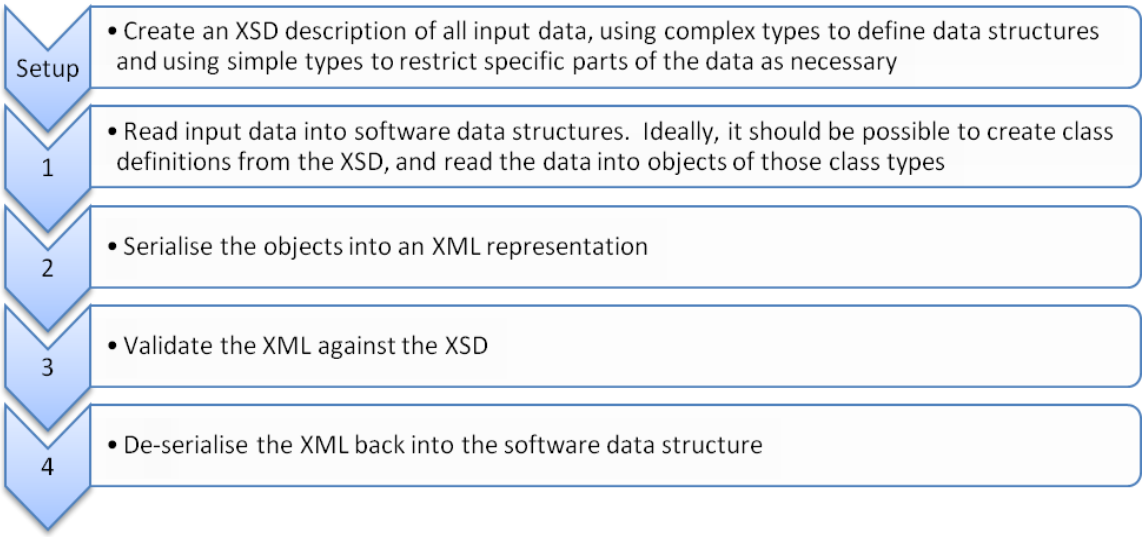
The steps required to achieve a consistent input validation can be informally generalised as follows:



This generalised form of a consistent input validation process is being presented to highlight that it is generic in nature and there are numerous ways in which it can be achieved.

3.2 The Process of Input Validation using XML Schema

Based on the tools available, the widespread programmatic support and the ease of use and understanding, it seems appropriate to use XML Schema to implement a consistent input validation process. An overview of the process is as follows:



The last step in this process is perhaps optional, as if the XML passes validation, then there seems little point in transforming it back into the software data structure, as we already have the data in that format from step 1. However, it is conceivable that the process of transforming the data into XML may change the data slightly, but this is dependent on the implementation of the XML serialisation that is being used. For

consistency's sake it is probably best perform the last step as it ensures the data being used is the data that was validated.

We have already talked about the basics of an XSD and there are numerous tutorials available on how to create one. The following sections will go into some detail about the other steps in this process.

3.3 Internal data representation

The internal representation of the input data is completely arbitrary, and is up to the programmer. They can decide exactly what structures or object hierarchy to use that best describes the data. What is required is the ability to transform that representation into an XML representation, and what will be suggested here is what the author believes to be the most convenient solution.

Ideally we would like to leverage the convenience of any programming language features that enable us to easily convert an object into an XML representation. By way of example, two programming languages have such features, Microsoft .NET and Java. Both provide a tool that takes as input an XSD and outputs class definitions corresponding to that XSD, but more importantly the class definitions are defined in such a way that are already setup to be serialised or de-serialised (marshalled and un-marshalled in Java) to XML.

3.3.1 Creating classes from an XSD in Java

Sun has made available a special Java package to deal with binding to XML called the Java Architecture for XML Binding (JAXB) and is available at https://jaxb.dev.java.net/. Included as part of that package is a tool called xjc, which can take as a parameter an XSD file and create Java class files that correspond to the XSD and can easily be marshaled and un-marshaled to XML.

An example output of xjc can be seen in the Appendix in section 0, where the input was Author.xsd. The command line that was run was simply

```
xjc.bat Author.xsd
```

3.3.2 Creating classes from an XSD in .NET

Microsoft has made available a tool as part of the .NET Framework SDK called the XML Schema Definition Tool (xsd.exe). xsd.exe can take as a parameter an XSD file and create C# (or (managed) C++, VB, Visual J# or JScript) class files that correspond to the XSD and can be easily serialised or de-serialised to XML.

An example output of xsd.exe can be seen in the Appendix in section 9.2, where the input was Author.xsd. The command line that was run was simply

```
xsd.exe Author.xsd /classes
```

3.4 XML Serialisation

It is beyond the scope of this paper to discuss the various APIs available in the various programming languages that are capable of XML serialisation. It suffices to say that a quick search on the internet reveals XML serialisers for languages such as C++, Python, Delphi, JavaScript, Ruby, Perl, PHP and of course C# and Java. The authors preferred solution is to use the method of automatic class creation as outlined in the previous section, thus bypassing the need to be familiar with the intricacies of a particular languages XML serialisation APIs.

3.5 Validating the XML

Once the input data has been transformed into XML it can then be validated against the XSD to determine if the data should continue to be processed by the application. Again, a description of the various APIs available in the various programming languages that are capable of doing this is beyond the scope of this paper. Instead we shall briefly concern ourselves with what the appropriate process is should the validation fail.

Obviously should validation fail the data should not be processed any further by the application and some sort of error information should be generated. It is often necessary to be able return meaningful error text describing the nature of the problem that caused the input data to fail validation, one way that this might be achieved is by adding an attribute to each element in the XSD that corresponds to error text (or an index into an error text resource file) should that element fail validation, in this way the error text could be retrieved, and be passed back to the user, or logged, or whatever is appropriate for the application.

4 Other considerations

Using this process of input validation elevates the importance of the XSD as it is responsible for defining the validity of all the input. From a malicious user's perspective it then becomes a target for modification in order to provide malicious data to the application. This scenario is essentially a local user running a local application so they have complete control anyway, but it would be prudent to make it as least as difficult as what is currently required, which is modifying the application binary. To that end it is sensible to embed the XSD as a resource inside the application binary and sign the XSD itself, with the signature being validated on application start-up. As an extra layer of security the XSD could also be encrypted to deter malicious users from examining it for overly relaxed validation criteria.

It may also be prudent to add versioning information to the XSD as part of the process. Although normal source control is enough for versioning for historic reasons, depending on if the XSD is in a stand-alone file or not, it is very important from a security point of view that an old version of the XSD could not be used to replace a more current version.

This process is not an alternative to traditional software testing, but should be used in conjunction with, and indeed should compliment, such testing. Having the input to the application well defined in XSD files should aid in more focused testing, and indeed

there exist software testing tools that take a specification of input in XSD e.g. www.fuzzware.net.

5 Advantages

To understand the advantages this solution offers we must understand how current solutions attempt to solve the problem of input validation. At best there are clearly specified guidelines on how input validation should be achieved and possible common libraries to perform some of the validation. However, in the author's experience, after reviewing and being involved in numerous development projects, this kind of organised approach is very rare. More typically it is up to each developer to write their own input validation code which naturally leads to a spectrum of success, but more importantly most tend to focus their efforts on functionality and do not have the required security background to appreciate the importance of input validation.

This solution offers a simple and consistent process of input validation that actually means a developer needs to write less code. Using an XSD for input validation is simple because the XSD is written in XML which lends itself to be read easily, at least more so than auditing someone else's code. Having an easy to read syntax for describing input validation means it's easy to write and it can be reused, this should encourage developers to use more rigorous definitions for input validation as it takes less effort to implement. Just having the process itself forces developers to focus on input validation as a separate entity in the software design.

The process is consistent, it can be used everywhere in the code where input validation needs to occur and because it is programming language independent developers or testers needn't be fluent in that language to audit their own or someone else's XSD. Once a framework is established for validating input there is very little overhead if more input needs to be added or input restrictions changed and less code means less bugs and more time can be spent on areas of the code that the developer actually wants to work on.

Having a process in place is crucial. Since there is inevitable a range of abilities in any development team, a simple process for input validation means developers don't have to concern themselves with how to do input validation, there is set process they can follow that is easier than writing their own code to do it. Thus input validation can be achieved throughout the software to a consistent standard regardless of variations in ability.

The set of restrictions available via the XSD is finite, practical and simple. This means a developer defining restrictions is able to assess the relevance of all the possible restrictions. Having the set of available restrictions to hand means restrictions are like a checklist and it is less likely that a restriction will be missed. For instance a common mistake in length checking is checking against the upper bound but not the implicit lower bound of 0, however if the developer has a list of possible restrictions to work from they are less likely to miss a restriction. Being able to easily see the sort of restrictions that other developers use (since all restrictions are in a common XSD format as opposed to

buried in code) means a developer can gain insight and knowledge into better input validation.

Different internationalisations of the software can have different XSDs. For instance the date format varies from one country to another and there is a significant code overhead in handling the possible formats. The character alphabet varies from language to language and so a regular expression in an XSD could be updated to include/exclude language specific characters. If a different XSD is used per internationalised version then there is very little overhead in supporting new versions.

6 Limitations

It should be stressed that the type of input validation offered by this solution is only in regards to data type restrictions, and possibly some structure validation. There may still be significant application or business logic that needs to be applied to the data before it should be stored or processed further. Logic restrictions are by their nature application dependent and so generally require customised validation. For example, XSD can validate the format of an account or ID number, but not that the account or ID number actually exists in the system.

Using an XSD for input validation means there is an exact description of what is being verified, and this description is likely to be available to a malicious user. If there were an error in the XSD where an input was not being properly validated it would be potentially easier for an attacker to determine this. This is akin to the security arguments around open-source software; in this sense the XSD represents a sort of open-source input validation.

Having a single process for input validation creates a single point of failure, and any errors in the XML/XSD parsing and validation libraries could lead to security issues in the application relying on them. Arguably, whenever third-party code is used there is an acceptance of this risk.

The process of serialising to XML, validating against a schema and de-serialising is unlikely to be particularly efficient. There is always a balance when it comes to efficiency, the benefits of security and ease of design must be weighed against performance, however the final decision depends on the needs of the product.

7 Conclusions

Poor input validation is the leading cause of software security vulnerabilities. This is arguably due to a lack of education and prioritisation given to developers with regards to the security consequences.

More can be done to rectify this problem than education alone, developments teams can implement a consistent approach to input validation that can be used across an entire application. Implementing a process for input validation will only be successful if it

makes a developer's job easier, and it has been the aim of this paper to introduce a process that requires less code than usual input validation, will save time, is consistent across the entire application, is programming language independent and simplifies auditing.

8 References

i. http://nvd.nist.gov/statistics.cfm - National Vulnerability Database CVE statistics
ii. http://www.w3.org/XML/ - The XML working group
iii. http://www.w3.org/TR/xml11 - The XML specification
iv. http://www.w3.org/TR/xmlschema-0/ - The XML Schema specification, part one
v. http://www.w3.org/TR/xmlschema-1/ - The XML Schema specification, part two
vi. http://www.w3.org/TR/xmlschema-2/ - The XML Schema specification, part three
vii. http://www.w3.org/2001/XMLSchema.xsd - The XML Schema schema definition file
viii. http://www.w3schools.com/schema/default.asp - An excellent tutorial on XML Schema

9 Appendix

9.1 Output of Java's xjc.bat on Author.xsd

9.1.1 com\ngssoftware\Author\package-info.java

```
//
// This file was generated by the JavaTM Architecture for XML
Binding(JAXB) Reference Implementation, v2.1.2-b01-fcs
// See <a
href="http://java.sun.com/xml/jaxb">http://java.sun.com/xml/jaxb</a>
// Any modifications to this file will be lost upon recompilation of
the source schema.
//

@javax.xml.bind.annotation.XmlSchema(namespace =
"http://www.ngssoftware.com/Author")
package com.ngssoftware.author;
```

9.1.2 com\ngssoftware\Author\ObjectFactory.java

```
//
// This file was generated by the JavaTM Architecture for XML
Binding(JAXB) Reference Implementation, v2.1.2-b01-fcs
// See <a
href="http://java.sun.com/xml/jaxb">http://java.sun.com/xml/jaxb</a>
// Any modifications to this file will be lost upon recompilation of
the source schema.
//


package com.ngssoftware.author;

import javax.xml.bind.annotation.XmlRegistry;


/**
 * This object contains factory methods for each
 * Java content interface and Java element interface
 * generated in the com.ngssoftware.author package.
 * <p>An ObjectFactory allows you to programatically
 * construct new instances of the Java representation
 * for XML content. The Java representation of XML
 * content can consist of schema derived interfaces
 * and classes representing the binding of schema
 * type definitions, element declarations and model
 * groups.  Factory methods for each of these are
 * provided in this class.
 *
 */
@XmlRegistry
public class ObjectFactory {


    /**
```

```
     * Create a new ObjectFactory that can be used to create new
instances of schema derived classes for package: com.ngssoftware.author
     *
     */
    public ObjectFactory() {
    }

    /**
     * Create an instance of {@link Author }
     *
     */
    public Author createAuthor() {
        return new Author();
    }

}
```

9.1.3 com\ngssoftware\Author\Author.java

```
//
// This file was generated by the JavaTM Architecture for XML
Binding(JAXB) Reference Implementation, v2.1.2-b01-fcs
// See <a
href="http://java.sun.com/xml/jaxb">http://java.sun.com/xml/jaxb</a>
// Any modifications to this file will be lost upon recompilation of
the source schema.
//


package com.ngssoftware.author;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;


/**
 * <p>Java class for anonymous complex type.
 *
 * <p>The following schema fragment specifies the expected content
contained within this class.
 *
 * <pre>
 * &lt;complexType>
 *   &lt;complexContent>
 *     &lt;restriction
base="{http://www.w3.org/2001/XMLSchema}anyType">
 *       &lt;sequence>
 *         &lt;element name="Name"
type="{http://www.w3.org/2001/XMLSchema}string"/>
 *         &lt;element name="Filename">
 *           &lt;simpleType>
 *             &lt;restriction
base="{http://www.w3.org/2001/XMLSchema}string">
 *               &lt;maxLength value="260"/>
 *             &lt;/restriction>
```

```
 *           &lt;/simpleType>
 *         &lt;/element>
 *       &lt;/sequence>
 *     &lt;/restriction>
 *   &lt;/complexContent>
 * &lt;/complexType>
 * </pre>
 *
 *
 */
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "name",
    "filename"
})
@XmlRootElement(name = "Author")
public class Author {

    @XmlElement(name = "Name", required = true)
    protected String name;
    @XmlElement(name = "Filename", required = true)
    protected String filename;

    /**
     * Gets the value of the name property.
     *
     * @return
     *     possible object is
     *     {@link String }
     *
     */
    public String getName() {
        return name;
    }

    /**
     * Sets the value of the name property.
     *
     * @param value
     *     allowed object is
     *     {@link String }
     *
     */
    public void setName(String value) {
        this.name = value;
    }

    /**
     * Gets the value of the filename property.
     *
     * @return
     *     possible object is
     *     {@link String }
     *
     */
    public String getFilename() {
        return filename;
```

```
    }

    /**
     * Sets the value of the filename property.
     *
     * @param value
     *     allowed object is
     *     {@link String }
     *
     */
    public void setFilename(String value) {
        this.filename = value;
    }

}
```

9.2 Output of .NET's xsd.exe on Author.xsd

9.2.2 Author.cs

```
//------------------------------------------------------------------------
---------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:2.0.50727.42
//
//     Changes to this file may cause incorrect behavior and will be
lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------
---------

using System.Xml.Serialization;

//
// This source code was auto-generated by xsd, Version=2.0.50727.42.
//


/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
[System.Xml.Serialization.XmlTypeAttribute(AnonymousType=true)]
[System.Xml.Serialization.XmlRootAttribute(Namespace="http://www.ngssof
tware.com/Author", IsNullable=false)]
public partial class Author {

    private string nameField;

    private string filenameField;

    /// <remarks/>
```

```
[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.Xm
lSchemaForm.Unqualified)]
    public string Name {
        get {
            return this.nameField;
        }
        set {
            this.nameField = value;
        }
    }

    /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.Xm
lSchemaForm.Unqualified)]
    public string Filename {
        get {
            return this.filenameField;
        }
        set {
            this.filenameField = value;
        }
    }
}
```