Introducing SRE Inspectors.

A tutorial for building plug-ins for the Security Runtime Engine.

Version 0.20 - 30 August, 2010

Abstract

The Security Runtime Engine (SRE) is part of the Microsoft Web Protection Library (WPL); tools Microsoft uses on both internal and external facing web sites to help protect applications from security vulnerabilities.

This tutorial introduces the new SRE design and explains how you can protect your existing or new ASP.NET web sites without touching application source.

This tutorial is Technical Level 200 and assumes you are familiar with developing .NET applications. To take best advantage of this tutorial you must:

* Install Visual Studio 2010 RTM
* Download the latest source of the Security Runtime Engine

For up-to-date versions of this document, the SRE and WPL, including source code visit <http://wpl.codeplex.com>.

This tutorial is based on the September 2010 CTP release of the WPL. This release is an early preview to allow you to learn about the direction of the SRE and offer feedback on the new plug-in model. As such the plug-in model is subject to change.

The WPL is licensed under the Microsoft Public License, an open source license the text of which can be read at <http://www.microsoft.com/opensource/licenses.mspx>.

This CTP is not suitable for running on production servers.

Contents

Introduction to the SRE 3

Prerequisites 3

Computer Configuration 3

Getting Help 4

Getting Started 5

Understanding what and where you can inspect. 5

Request Inspectors 5

Page Inspectors 5

Response Header Inspectors 5

Inspector Return Values 6

Task 1: Compiling the WPL and creating a test project. 6

Compile the SRE Source 6

Create your test project. 7

Task 2: Writing and running an inspector. 8

Create a Request Inspector 8

Loading the inspector into the test web site 10

Task 3: Supporting Exclusions 11

Adding Inspector Specific Path Exclusions 11

Task 4: Understanding return types 13

Task 5: Putting it all together 14

What next? 16

Code Listings

It is often necessary to split code listings over multiple lines in this document when, in fact, they are a single line. The ⮰ symbol indicates where this split takes place. When entering code containing ⮰ symbols please ensure you put the code on a single line.

**Disclaimer:** This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal reference purposes.

© 2010 Microsoft Corporation. All rights reserved.

Microsoft, IntelliSense, Visual Studio, Windows Vista, and Windows are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.

# Changes since the May CTP

The following changes have been introduced since the May CTP.

* A new inspector, ResponseInspector has been added.
* Sample inspectors are now distributed with the SRE – these are documented in “The Microsoft SRE Inspectors” document installed with the SRE.
* Separate .NET 3.5 and .NET 4.0 assemblies are now available.

# Introduction to the SRE

The Security Runtime Engine (SRE) is an HTTP module which wraps itself around the requests to your web application and the response those requests cause. It provides a common framework to examine, log and act on requests and responses without having to access to an ASP.NET application’s source code.

The SRE has a plug-in model which allows you to write your own “inspectors”.

This tutorial introduces the basic features of the SRE plug-in with a series of exercises that show you how to:

* Create an inspector which examines in-bound requests.
* Load your inspector into the SRE.

## Prerequisites

To take advantage of this tutorial, you should be familiar with the following:

* Microsoft® Visual Studio® 2010 RTM
* C# programming language
* .NET Framework
* Basic practices for building, debugging, and testing software
* An understanding of an HTTP request and response and the ASP.NET pipeline.

For developers who are unfamiliar with web security MSDN has articles about security including check lists, videos and tutorials at http://msdn.microsoft.com/security.

## Computer Configuration

These tutorials require that the following software components are installed:

* Windows® 7, Windows Vista® or Windows Server® 2008 operating system
* Visual Studio 2010

You can compile the SRE and write plug-ins using Visual Studio 2008; however you must have Visual Studio 2008 SP1 / .NET 3.5 SP1 and no support is offered for this configuration.

The SRE source can be downloaded from http://wpl.codeplex.com

## Getting Help

If you have a question or comment please post it on the [discussion forum](http://wpl.codeplex.com/Thread/List.aspx) on the WPL CodePlex site.

# Getting Started

## Understanding what and where you can inspect.

Before you start to write an inspector you must understand the types available. The SRE currently has dour types of inspectors;

* Request inspectors
* Page inspectors
* Response Header inspectors
* Response Inspectors

An inspector must, at least, implement one of the four interfaces from the Microsoft.Security.Application.SecurityRuntime.PlugIns namespace.

Another extension point is provided for logging. A logging plug-in will be used by the SRE and methods are provided for your inspectors to use logging.

Let’s look at each of the inspector types in turn;

### Request Inspectors

A request inspector allows you to examine an inbound request to your application before it actually reaches your application. The request is passed to the inspector before any processing is done by ASP.NET, and before authentication, authorization or caching. Potential uses for request inspectors include validation of all requests, banning of particular user agents and examining request parameters for SQL Injection strings.

A request inspector implements the IRequestInspector interface.

### Page Inspectors

A page inspector allows you to examine the contents of the Page object created when responding to a request. Page objects are normally created by an ASP.NET WebForms application. Potential uses for page inspectors include ensuring the properties within a control are encoded appropriately.

A page inspector implements the IPageInspector namespace.

### Response Header Inspectors

A response header inspector allows you to add or change HTTP headers to a response before it is sent back to the client. The functions you can perform on HTTP headers depends on your hosting environment –in IIS7.x in an integrated pipeline application pool can examine, add, change and remove headers; in IIS6, or a classic application pool an inspector can only add headers, which will replace any existing headers of the same name. Potential uses for this type of inspector would be to insert mitigation headers such as the new click-jack header, X-FRAME-OPTIONS.

A response header inspector implements the IResponseHeaderInspector interface.

### Response Inspectors

A response inspector is the last inspector that runs during processing, acting as a last defense. The response is delivered to the inspector as a byte array you can examine and change. Potential uses for this type of inspector include response filtering, such as examining responses for credit card numbers and stopping them reaching the client.

A response inspector implements the IResponseInspector interface.

### Inspector Return Values

Each inspector must return an object which implements the IInspectionResult interface from the Microsoft.Security.Application.SecurityRuntime namespace. Three pre-built result classes are available for you to use; RequestInspectionResult, ResponseInspectionResult and PageInspectionResult. You can of course write your own. The value of the Severity property from the IInspectionResult interface controls if further processing is halted or continues.

Task 1: Compiling the WPL and creating a test project.

Before you can create an inspector you must install the WPL, ensuring you select either or both of the SRE installations appropriate to your .NET Framework version. You will also need to create an assembly for your inspectors and a web site to test them with.

### The SRE Assemblies

By default the SRE installs into C:\Program Files\Microsoft Information Security\Web Protection Library v4.0\SecurityRuntimeEngine, or C:\Program Files (x86)\Microsoft Information Security\Web Protection Library v4.0\SecurityRuntimeEngine if you are on a 64bit operating system. Within the install location there separate directories for .NET 3.5 and .NET 4.0 versions of the SRE.

The .NET 4.0 version consists of a single assembly;

* Microsoft.Security.Application.SecurityRuntimeEngine.dll

The .NET 3.5 version consists of two assemblies;

* Microsoft.Security.Application.SecurityRuntimeEngine.dll and
* System.ComponentModel.Composition.dll

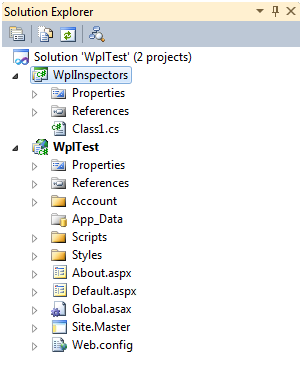
These are the assemblies you reference when creating a new inspector or install the SRE around your applications.

### Create your test project.

Start Visual Studio and create a new ASP.NET Web Application (File ⇨New ⇨Project then choose ASP.NET Web Application from the Web templates). Name the solution WplTest and click OK.

Next create a new Class Library project in the solution (File ⇨Add ⇨New Project, then choose Class Library from the Windows templates). Name the new project WplInspectors.

Your solution should look something like the solution in the following image:



Before you can write an inspector you need to configure the test web site to load the Security Runtime Engine. Right click on the references folder in the WplTest project and choose “Add Reference …” from the context menu. Select the Browse tab, navigate to the SRE binary directory for your version of .NET and add references to the assemblies contained within it.

Repeat the process and add references to the SRE assemblies in the WplInspectors project. In the WplInspectors project you will also need to add references to System.Configuration, System.Web and System.Web.Abstractions which you will find on the .NET tab of the Add Reference dialog.

The final step in getting ASP.NET to load the SRE is to place it in the ASP.NET pipeline. Open up the web.config file in the WplTest project and make the following changes.

1. In the httpmodules section of system.web add a new module:

<httpModules>

**<add name="SecurityRuntimeEngine" ⮰  
type="Microsoft.Security.Application.SecurityRuntimeEngine.⮰  
SecurityRuntimeEngine, ⮰  
Microsoft.Security.Application.SecurityRuntimeEngine"/>**

…

</httpModules>

1. In the modules section of system.webServer make the following changes:

<system.webServer>

**<validation validateIntegratedModeConfiguration="false"/>**

<modules>

**<remove name="SecurityRuntimeEngine" />**

**<add name="SecurityRuntimeEngine"**

**preCondition="managedHandler"**

**type="Microsoft.Security.Application.⮰  
SecurityRuntimeEngine.SecurityRuntimeEngine, ⮰  
Microsoft.Security.Application.SecurityRuntimeEngine"/>**

…

</modules>

</system.webServer>

Depending on your version of Visual Studio and the framework version you target your web.config file may be missing some of the sections mentioned above – simply add them if they don’t already exist.

Task 2: Writing and running an inspector.

If you run the test web site at this point you will see that nothing appears to have changed – there aren’t any inspectors to load, so nothing will happen. Let’s start off with the simplest inspector, a Request Inspector. Request Inspectors run at the first possible point in the ASP.NET pipeline, BeginRequest. This event runs just after a request is received, before any authentication or authorisation takes place.

### Create a Request Inspector

Right click on the WplInspectors project in Solution Explorer and select Add ⇨Class from the context menu. Edit the name field to be RequestInspector.cs. Replace the empty class declaration with the following:

using System;

using System.ComponentModel.Composition;

using System.Web;

using Microsoft.Security.Application.SecurityRuntimeEngine;

using ⮰

Microsoft.Security.Application.SecurityRuntimeEngine.PlugIns;

namespace WplInspectors

{

[Export(typeof(IRequestInspector))]

public class RequestInspector : IRequestInspector

{

public IInspectionResult Inspect(

HttpRequestBase request)

{

return new RequestInspectionResult(

InspectionResultSeverity.Halt);

}

public ExcludedPathCollection ExcludedPaths

{

get

{

return new ExcludedPathCollection();

}

}

}

}

The Request Inspector implements the IRequestInspectorInterface which requires a single method,

public IInspectionResult Inspect(HttpRequestBase request)

and a single read-only property,

ExcludedPathCollection ExcludedPaths

The SRE comes with three prebuilt classes which implement IInspectionResult, one for each type of inspector;

* RequestInspectionResult
* ResponseInspectionResult and
* PageInspectionResult

The ExcludedPaths property is common to all inspectors and allows you to return a list of request virtual paths which will be excluded from inspection by the inspector. The SRE also has a way of globally excluding paths from all inspectors via configuration and a way of programmatically excluding web form pages/classes from processing by Page Inspectors via attributes applied to the page.

### Loading the inspector into the test web site

The sample inspector above halts every request – which makes it very easy to see in action. In a real deployment of the SRE default location for inspectors will be in a wplPlugins directory under your application’s bin directory. This is rather hard to mimic in a development environment, but what you can easily do is make sure your inspector assembly is copied to your web site’s bin directory. Right click on the WplTest project and choose “Add Reference…”, then select the Projects tab and select the WplInspectors project.

To change the directory where the SRE looks for inspectors you needed to change the SRE setting. Open the web.config file and add a section declaration to the configSections section as follows (if you don’t have a configSections section create it):

<configSections>

<section name="sreSettings"

type="Microsoft.Security.Application.SecurityRuntimeEngine.⮰

SecurityRuntimeSettings, Microsoft.Security.Application.⮰

SecurityRuntimeEngine" />

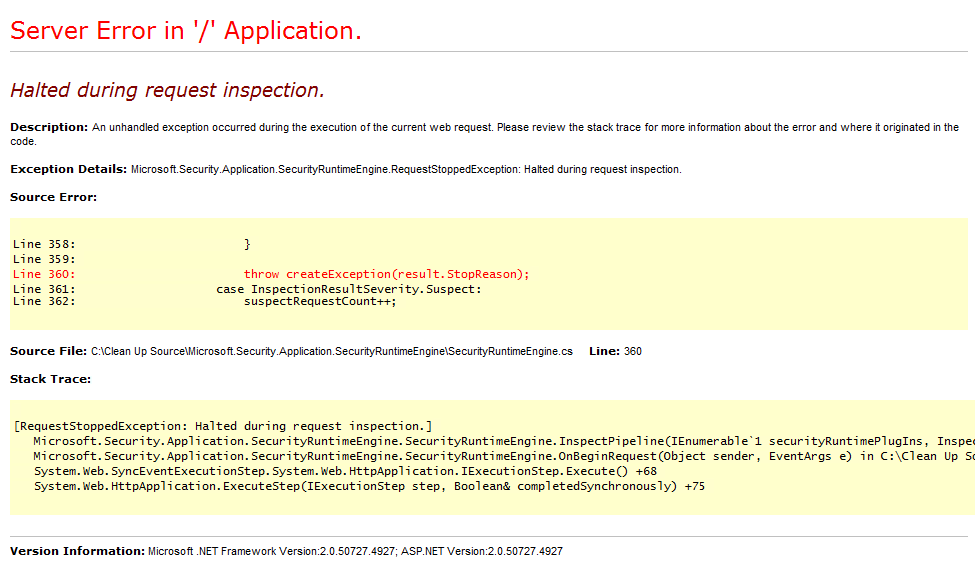
</configSections>

Now create a new section in the configuration file as follows:

<sreSettings plugInDirectory=".">

</sreSettings>

Now run your test web site and you should see the yellow exception screen:



Of course in production web sites you would never leave the standard error page turned on under any circumstances. You now have your first inspector.

Task 3: Supporting Exclusions

Obviously this isn’t a very useful inspector, so now might be a good time to talk about excluding inspections and excluding them from particular paths.

First let’s look at excluding an inspector from the SRE. Open up web.config and edit the sreSettings section you added in Task 2. Add the following lines

<sreSettings plugInDirectory=".">

**<disabledPlugIns>**

**<add name="RequestInspector"**

**type="WplInspectors.RequestInspector" />**

**</disabledPlugIns>**

</sreSettings>

Run the sample web site again and you’ll the request is successful. Globally disabling an inspector is useful if you have an assembly which contains multiple inspectors. You can also exclude by virtual path. Replace the disabledPlugIns section with an excludedPaths section as shown below:

<sreSettings plugInDirectory=".">

**<excludedPaths>**

**<add path="/default.aspx" />**

**</excludedPaths>**

</sreSettings>

Again if you try to load the example web site then the default page will load, however any other request paths will be rejected. Of course if you are using request rerouting or any other remapping technology you will need to be very careful to cover all your paths if you use path based exclusion.

### Adding Inspector Specific Path Exclusions

Global options are useful, but what if you want to support exclusions on a per-inspector basis? If you remember when you created the inspector there was a method and a property, ExcludedPaths. By returning a collection of ExcludedPath objects from the ExcludedPaths property you can support inspector based exclusions.

A simple way would be to hard code an exclusion list thus;

public ExcludedPathCollection ExcludedPaths

{

get

{

ExcludedPathCollection excludedPaths =

new ExcludedPathCollection();

excludedPaths.Add("/default.aspx");

return excludedPaths;

}

}

However obviously this is not sustainable. You could source the exclusions for a database, or any other data source you want to, or you could use web.config. The SRE contains an abstract class, BasePlugInConfiguration, from which you can create your own configuration sections.

Add a new class to your inspector project called RequestInspectorSettings. Replace the contents of the class with the following:

using ⮰

Microsoft.Security.Application.SecurityRuntimeEngine.PlugIns;

namespace WplInspectors

{

internal class RequestInspectorSettings :

BasePlugInConfiguration

{

}

}

This is almost enough to support an <excludedPath> section, and provides you with a starting point with which to add your own ConfigurationProperty elements. However the SRE doesn’t know about this new class and neither does your inspector.

Open up your request inspector class and make the following changes:

[Export(typeof(IRequestInspector))]

public class RequestInspector : IRequestInspector,

**IConfigurablePlugIn**

{

**private RequestInspectorSettings internalSettings =**

**new RequestInspectorSettings();**

public IInspectionResult Inspect(HttpRequestBase request)

{

return new

RequestInspectionResult(InspectionResultSeverity.Halt);

}

public ExcludedPathCollection ExcludedPaths

{

get

{

**return this.internalSettings.ExcludedPaths;**

}

}

**public string ConfigurationSectionName**

**{**

**get**

**{**

**return "requestInspectorSettings";**

**}**

**}**

**public BasePlugInConfiguration Settings**

**{**

**get**

**{**

**return this.internalSettings;**

**}**

**set**

**{**

**this.internalSettings =**

**(RequestInspectorSettings)value;**

**}**

**}**

}

You can see that the inspector has implemented a second interface, IConfigurablePlugIn. This requires two properties, ConfigurationSectionName and Settings. When the SRE starts and loads any inspectors it examines each inspector to see if it needs configuration. If the inspector implements IConfigurablePlugIn the SRE will attempt to load a configuration section with the name returned from the ConfigurationSectionName property and then set the Settings property with the configuration section, if one exists.

So now if you add a new section declaration to the configSections section of web.config;

<section name="requestInspectorSettings"

type="WplInspectors.RequestInspectorSettings,⮰

WplInspectors"/>

and a matching section in your web.config, for example;

<requestInspectorSettings>

<excludedPaths>

<add path="/default.aspx" />

</excludedPaths>

</requestInspectorSettings>

you can have an excludedPath collection for your inspector.

Task 4: Understanding return types

Each Inspect method is expected to return a class which implements IInspectionResult. This interface requires two properties, Severity and StopReason. The Severity property is an enum, InspectionResultSeverity; which has three possible values:

* Halt – This halts all processing of the request or response.
* Continue – This allows further processing of the request or response.
* Suspect – This marks the request or response as suspect.

Suspect results have a cumulative effect – if enough suspect results are returned during the processing of the request and its response then any further processing will be halted. By default all suspect results are ignored, however you can set the maximum number of allowed suspect results by adding an allowedSuspectResults attribute to the SRE settings element in web.config. You can also control if the suspect count is reset at each inspection stage by adding a resetSuspectCountBetweenStages attribute; this defaults to false. For example;

<sreSettings allowedSuspectResults="2"

resetSuspectCountBetweenStages=false>

</sreSettings>

The StopReason property is logged when a request or response is stopped using any logging plug-ins that are available and as the message in the exception thrown at that point.

The SRE contains three built-in implementations of IInspectionResult you can use;

* RequestInspectionResult
* ResponseInspectionResult and
* PageInspectionResult

Each of these implementations allows you to add more information returned in the StopReason, for example the constructor for RequestInspectionResult can take an IEnumerable<RequestProblemParameter> which allows you to pass in a collection of the name, value and type of parameter which your inspector will not let past.

Task 5: Putting it all together

So let’s imagine you have an existing web application where sending a request to it with a query string parameter named crash and value of true would cause it to crash. You don’t have access to the source code right now but want to quickly stop any request like this reaching the application while letting all other requests through – an inspector for this may look something like:

[Export(typeof(IRequestInspector))]

public class CrashRequestInspector : IRequestInspector

{

public IInspectionResult Inspect(HttpRequestBase request)

{

if (!string.IsNullOrEmpty(

request.QueryString["crash"]) &&

string.Compare(request.QueryString["crash"],

"true", true) == 0)

{

List<RequestProblemParameter> problemParameters =

new List<RequestProblemParameter>();

problemParameters.Add(

new RequestProblemParameter(

RequestParameterType.QueryString,

"crash",

request.QueryString["crash"]));

return new RequestInspectionResult(

InspectionResultSeverity.Halt, problemParameters);

}

return new RequestInspectionResult(

InspectionResultSeverity.Continue);

}

public ExcludedPathCollection ExcludedPaths

{

get

{

return new ExcludedPathCollection();

}

}

}

# What next?

We’ve released the September CTP source drop in order to gather your feedback about where we are taking the SRE. We would like you to look at the code, play with it, write inspectors, write loggers and post your feedback, and any bugs you may find, on the [WPL CodePlex site](http://wpl.codeplex.com/Thread/List.aspx). We don’t promise to change the WPL in the exact way you want, but we do promise to read every piece of feedback we get.

Thanks for taking the time to download the source, this tutorial and for any feedback you can give us. We look forward to helping you protect your ASP.NET applications!