MS_logo_K

November 13, 2009

By Windows Live Engineering—Eric Holton, Dan Crevier

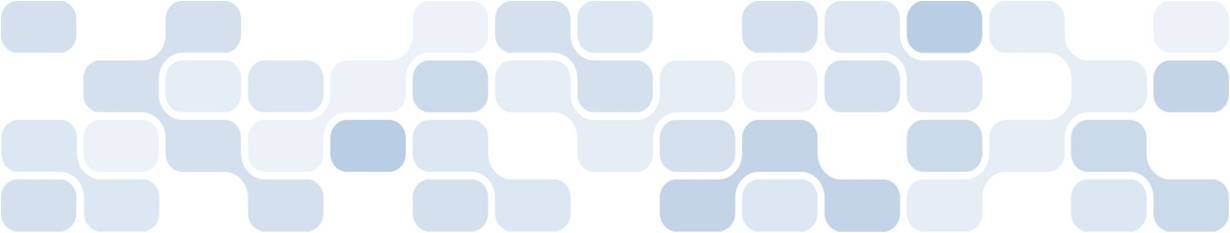
Windows Live Security—Spencer Low, Ali Pezeshk, Deepak Manohar

Business Online Security—Sacha Faust Bourque, John Walton

Casaba—John Hernandez, Chris Weber

ASP.NET MVC—Phil Haack

For the latest information, please see <http://www.microsoft.com/sdl>



The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in, or introduced into a retrieval system, or transmitted in any form, by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

© 2009 Microsoft Corporation. All rights reserved.

Microsoft, Visual Studio, Windows Live, Hotmail, and Windows are trademarks of the Microsoft group of companies.

**Table of Contents**

[Introduction 2](#_Toc244318508)

[Overview of Windows Live 2](#_Toc244318509)

[Features of ASP.NET MVC 3](#_Toc244318510)

[Windows Live and the SDL Process 3](#_Toc244318511)

[Security Threat Landscape 4](#_Toc244318512)

[Solution Details 6](#_Toc244318513)

[Security Testing and Results 10](#_Toc244318514)

[Lessons Learned 11](#_Toc244318515)

[Acknowledgments 12](#_Toc244318516)

# Introduction

Security is an important aspect of any modern software development project. The advent of the Internet has made cybercrime an attractive proposition for many criminal organizations; according to U.S. government agencies, cybercrime is now more profitable than the illegal U.S drug trade. Operating systems have become more resistant to security attacks, so hackers frequently focus on applications, seeking new vulnerabilities and ways to exploit them.

The Windows Live™ team is responsible for a collection of services that enable users to store and manage their personal data, such as e-mail, photographs, documents, and contacts. Each user's data is stored remotely from the user's desktop. Users can sign in to Windows Live services to access their data by using a Web browser or other front-end application. Windows Live also includes a framework that enables developers to build applications that can integrate Windows Live services into their own custom solutions.

On the Microsoft platform, most Web applications are based on ASP.NET and the Microsoft®.NET Framework. ASP.NET MVC is a new framework based on ASP.NET that developers can use to build Web applications that follow the Model-View-Controller (MVC) pattern. The Windows Live team decided to use ASP.NET MVC as the basis for its services and framework. You can find more information about ASP.NET MVC on the Microsoft ASP.NET MVC Web site at <http://www.asp.net/mvc/>.

The Windows Live team used the Security Development Lifecycle (SDL) to analyze the security risks that the Windows Live services would face. In addition, the team used the SDL to mitigate these risks by designing the software carefully, following best practices in implementing these services, and subjecting the software to rigorous security testing prior to public release. The white paper "Applying the Security Development Lifecycle at Windows Live" describes the approach that the Windows Live team took to applying SDL to its services.

This paper discusses in depth the application of one of the principles of the SDL, “security by default.” It describes the approach that the Windows Live team took when it adopted the ASP.NET MVC framework, helping to prevent developers from making security errors while developing the services that Windows Live includes. Organizations building their own solutions based on ASP.NET MVC might consider adopting some of these practices as part of their development effort. In addition, this paper describes the benefits of introducing security mitigations into applications and services when the underlying framework is changed.

# Overview of Windows Live

Windows Live is a suite of software and services that keeps your life in sync by enabling seamless communication, providing easy management of your data across devices and services, and enabling you to do more with Windows. It provides communication and sharing services that include e-mail, instant messaging, photo management and sharing services, social networking, and secure online storage. Over 460 million people use Windows Live every day and it is one of the world’s largest online communities.

Windows Live is best known for two flagship services:

* **Windows Live™ Hotmail®.** This is a Web-based e-mail service that serves over 350 million subscribers. To put that in perspective, Hotmail’s user base has grown by an average rate of 42 percent a year since Microsoft acquired it in 1998. Hotmail customers sign in from almost every country connected to the Internet.
* **Windows Live™ Messenger**. This is a client-based instant messaging service. It has also enjoyed phenomenal growth and is the world’s largest, free instant messaging service, with over 320 million active customers.

The Windows Live framework provides an application programming interface (API) that enables developers to build applications that interact with Windows Live services. You can find more information about Windows Live on the Microsoft Windows Live Web site at <http://home.live.com/>.

# Features of ASP.NET MVC

ASP.NET MVC provides a new Model-View-Controller framework on top of the existing ASP.NET 3.5 runtime. This framework enables developers to easily take advantage of the MVC architectural pattern to build Web applications. Using the MVC pattern, developers can divide an interactive application into three functional areas:

* The user interface (the *view*).
* The business logic (the *model*).
* The controlling logic that binds them together (the *controller*).

This architecture maintains a clear separation of the various aspects of an application and helps to facilitate test-driven development (TDD).

The ASP.NET MVC framework provides an implementation of MVC that is specific to ASP.NET; it uses the Web application project format and provides a controller base class to handle and process requests (called actions). Developers can take advantage of the specific Microsoft Visual Studio® 2008 MVC templates provided with this release to create their Web applications.

The MVC framework is fully extensible, enabling developers to implement sophisticated functionality. Common examples include dependency injection (DI) techniques, new view-rendering engines, and specialized controllers.

The ASP.NET MVC framework is built on ASP.NET 3.5, so developers can also take advantage of existing ASP.NET 3.5 features such as localization, membership provider, profile management, and so on.

ASP.NET MVC provides the following benefits for developers seeking to build complex Web applications quickly and easily:

* It provides complete control over the HTML markup for Web pages.
* It enables rich AJAX integration.
* It provides intuitive Web site URLs.
* It maintains a clear separation of concerns (user interface, business logic, and controlling functionality), which results in Web applications that are easier to maintain and extend over time. It supports test-driven development.

These features made ASP.NET MVC the natural choice to use as the development framework for Windows Live.

# Windows Live and the SDL Process

There are more than 50 requirements in the SDL process that apply to the phases in the development process: Design, Implementation, Verification, Release, and Post-release. The requirements and recommendations of SDL are not static; they are added and changed on a regular basis in the light of emerging threats and improvements to supporting infrastructure, tools, and processes. Figure 1 shows the stages in the SDL process.

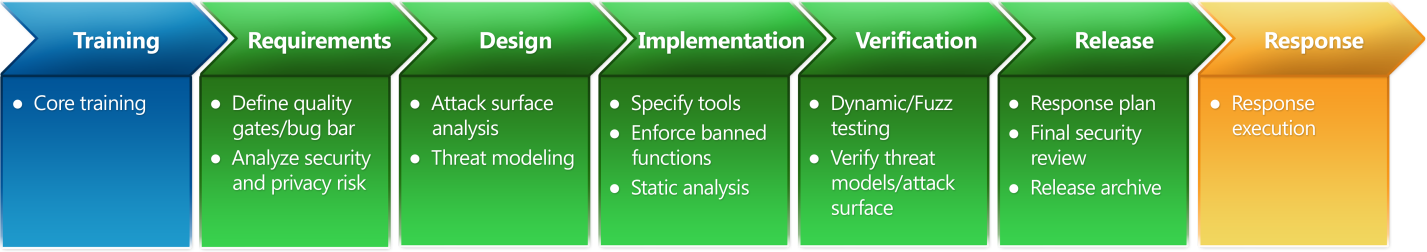


Figure 1: Stages in the SDL Process

As of 2004, meeting the SDL requirements became part of the release policy for all products in Microsoft. Each major product goes through a Final Security Review in the Release stage, to ensure that it complies with security requirements.

For more information about SDL, consult the Microsoft Security Development Lifecycle page at <http://www.microsoft.com/sdl>. The following sections describe specific steps that the Windows Live team took during the Design and Implementation phases. These steps were intended to prevent common security errors from entering the SDL.

# Security Threat Landscape

The Design stage of SDL includes performing a threat-modeling exercise to help understand the likely threats that an application will face. The development team can then consider how to mitigate these threats and incorporate these mitigations into the design.

The security threat landscape of Windows Live is very similar to that of other Web applications. The following table summarizes the Web-specific threats and attacks that the Windows Live team considered, together with strategies for countering them.

| Possible attack | Mitigations |
| --- | --- |
| Cross-Site Scripting  *Cross-site scripting, also known as XSS, is a specialized version of an HTML injection attack. XSS vulnerability enables an attacker to execute script in the user's browser. By running with the identity of the user on the Web site, this script can hijack the user's session, attempt to deface Web sites, insert hostile content, conduct phishing attacks, or even take control of the user's browser.* | * Encode output by using the Anti-XSS library. This library is available from the CodePlex Web site at <http://www.codeplex.com/AntiXSS>. * Perform input validation by using the **ValidateRequest** attribute. * In addition, the SDL also recommends the following two defense-in-depth practices:   1. Do not use the JavaScript **eval()** function of any similar functions that can evaluate and execute a string as though it was script code.   2. Do not set the **domain** property of a document to the top-level domain of the Web site. |
| Open Redirects  *An open redirect, also known as a cross-site redirect or XSR, enables an attacker to redirect a user request to an arbitrary site through a URL on your Web site. Attackers could use this for phishing attacks, spam, and serving malicious software (also called malware).* | * Verify all redirections against a list of allowed URLs. |
| Cross-Site Request Forgery  *A cross-site request forgery attack, or XSRF, forces the browser for a logged-on user to send a request to a vulnerable Web application. The vulnerable Web application performs the request with the identity of the user.* | * Set the **ViewStateUserKey** property on pages accessed by an authenticated user. * Alternatively, use a custom-built dynamic canary library (described below). |
| JS/JSON Hijacking  *JavaScript (JS) or JavaScript object notation (JSON) hijacking enables an attacker to read a JavaScript or JSON response from another site. Web sites typically return personally identifiable information (PII) about users in JSON responses, so this attack could allow an attacker to steal users’ PII just by visiting the attacker’s Web site. When the vulnerable Web site receives the request, it runs the script with the identity of the user.* | * Use a custom-built dynamic canary library. * As a defense-in-depth option, consider returning PII in JSON responses only to POST requests. |
| Note: The option of using ViewStateUserKey or a custom-built dynamic canary library warrants a little more explanation. Some teams at Microsoft do not use the ViewState property of a page for several reasons. This means that the corresponding pages cannot set the ViewStateUserKey property either. Consequently, these teams have built a library with application-level logic that incorporates anti-forgery validation tokens, referred to as *canaries*. A canary is an unpredictable, unguessable string of characters that the Windows Live Web applications can validate. A canary has a short period of time during which it is valid. A method in the library generates a canary, inserts the canary into the page, and then verifies that subsequent HTTP requests include the same canary. This challenge response sequence is similar to the ViewStateUserKey functionality. | |

The next section describes the methodology that the Windows Live team followed to implement mitigations in the ASP.NET MVC framework to counter XSRF attacks, open redirects, and JS/JSON hijacking threats.

# Solution Details

The SDL process mandates that developers adhere to specific best practices for coding software. The best practices that are pertinent to an application depend on the environment in which the application runs, and the threats posed to applications running in this environment. However, regardless of the type of application being developed, a fundamental tenet of SDL is the principle of "security by default." This means that software should be designed and implemented in such a way that a user or developer has to explicitly enable functionality that could possibly increase the attack surface, rather than disable functionality to make the application more secure.

***Defending Against XSRF Attacks***

*To defend a Web site against XSRF attacks, ASP.NET MVC provides* ***AntiForgeryToken*** *helpers. These consist of a* ***ValidateAntiForgeryToken*** *attribute, which the developer can attach to controller classes or methods, and the* ***Html.AntiForgeryToken()*** *method. The Windows Live team chose to implement a modified version of this approach for the following reasons:*

* *The* ***ValidateAntiForgeryToken*** *attribute expects the token to be passed in the HTTP POST body from an HTML FORM post request, but the Windows Live Web sites sometimes pass tokens in querystring parameters or HTTP request headers.*
* *The* ***AntiForgeryToken*** *helpers write a cookie named* ***\_\_RequestVerificationToken\_<AppPath Base64 encoded>****, whereas the Windows Live Web sites already have existing session cookies that they use to token values from. Therefore, Windows Live services can help protect against XSRF attacks without using an additional cookie. This reduces bandwidth and removes the need for cookie management, such as deleting a cookie when the user logs out.*
* *It was important to Windows Live that its XSRF defense mechanism be tied to the user session and ASP.NET MVC did not allow this.*

The following sections describe the techniques used by the Windows Live team to implement security by default by using ASP.NET MVC, and summarize how these strategies mitigate developer errors that might make Windows Live services vulnerable to XSRF, XSR, and JSON hijacking attacks.

## Using ASP.NET MVC Action Filters

To help counter the possible threats identified for the Windows Live sites, the Windows Live team required a strategy that guaranteed that all sites performed the checks necessary to guard against open redirects and XSRF and JSON hijacking attacks when running each action (an action is a method that runs as the result of a Web request from a user). These checks ensured that each action explicitly stated the following items:

* The HTTP verbs that it would accept.
* Whether each action required the use of a canary.

To implement these checks and provide security by default, the Windows Live team decided to take advantage of action filters in ASP.NET MVC. An action filter enables a developer to implement code that runs before and after each action is executed. The Windows Live team created a class called **AsyncBaseController** that all Windows Live sites inherit from when building their controller classes. A custom filter attribute called **WLXSecurityAttribute** was defined and added to this class. This attribute provides functionality that performs the security checks for all actions that the class implements (the implementation is discussed in the next section).

***Understanding ASP.NET Action Filters***

*As part of the normal flow of ASP.NET MVC, reflection is used to check for filters on an action or the action’s controller. ASP.NET MVC implements multiple types of filters (authorization filters, error filters, and so on), although action filters are the type most commonly used. If an action has action filters, the methods are called in the following order:*

1. ***ActionExecuting*** *(filter). Runs before the action method itself.*
2. *Action method (returns an* ***ActionResult****).*
3. ***ActionExecuted*** *(filter). Runs directly after the action method is executed.*
4. ***ResultExecuting*** *(filter). Runs before the* ***ActionResult*** *returned by the action method is executed.*
5. *Result (****ActionResult.Execute****).*
6. ***ResultExecuted****. Runs after the* ***ActionResult*** *is executed.*

*The filter methods provide interception points that can check and change the flow for an action, such as by raising an exception or by changing the action result.*

[WLXSecurity]  
public abstract class AsyncBaseController : System.Web.Mvc.Controller,   
 IAsyncController, IAsyncManagerContainer, IAsyncBaseController  
{  
 ...  
}

The **AsyncBaseController** class is itself an action filter, inherited from the **Mvc.Controller** class, which in turn implements the **IActionFilter** interface as virtual methods. The Windows Live team could have simply overridden these methods. However, had they followed this approach, it would have been too easy for descendent classes to override these methods and thus defeat the built-in security. Instead, they decided to implement the security filter as an attribute. In this way, developers inherit the restrictions of the security filter, maintaining the principle of security by default.

Actions in controller classes that inherit from the **AsyncBaseController** class must specify the **AcceptVerbs** attribute to indicate the HTTP verbs that the action will accept. In addition, depending on the HTTP verb, certain attributes are assumed as follows:

* Actions that accept HTTP verbs other than GET expect canaries by default. If developers want to disable checking for canaries, they can do so by specifying the **NoCanary**attribute.
* For actions that specify the HTTP GET verb, the **NoCanary** attribute is assumed. Although, from a security standpoint, it is ideal to enforce all requests to require a canary, it is not appropriate for most GET requests, such as those associated with performing navigation between Web pages. Therefore, the Windows Live team decided to make **NoCanary** checking the default mechanism for processing HTTP GET requests. If developers want to enable canary checking, they can do so by specifying the **RequireCanary** attribute.

As the names of the attributes indicate, the **RequireCanary** attribute requires all such requests to include canaries and the **NoCanary** attribute means that such requests do not need to include canaries. The developer can override these requirements if necessary, as shown in the following code examples.

[AcceptVerbs(HttpVerbs.Get)]  
[RequireCanary]  
public *returnType GetActionMethodName()*  
{  
 ...  
}  
  
[AcceptVerbs(HttpVerbs.Post)]  
[NoCanary]  
public *returnType PostActionMethodName()*  
{  
 ...  
}

The strategy of enforcement guarantees that the developer needs to make a conscious decision about:

* The HTTP verbs that are allowed for each action.
* Whether to turn on or turn off canary checking for each action.

Finally, using the **NoCanary** attribute (turning off canary checking) triggers an SDL security code review. This is a good mechanism to easily identify portions of the code that need a closer look from a security perspective.

If an action requires canary checking and a canary is not provided with a request, the action throws an exception that the Web site handles, usually by displaying an error page.

## Implementing the WLXSecurity Attribute

The **WLXSecurity** attribute is a class that implements the **IActionFilter** interface. This interface defines two methods called **OnActionExecuting** and **OnActionExecuted**. The **OnActionExecuting** method runs just before an action executes, and the **OnActionExecuted** method runs when the action has completed.

In the **WLXSecurity** attribute class, the **OnActionExecuting** method checks for the **AcceptVerbs** and canary attributes as described previously. The following pseudo code shows the typical logic implemented by the **OnActionExecuting** method.

public override void OnActionExecuting(ActionExecutingContext filterContext)  
{   
 *if the filterContext.Controller is not the AsyncBaseController class  
 throw an ArgumentException and exit the method  
  
 if the HTTP verb for the request is not a valid verb for the action  
 throw an ActionVerbRequired exception and exit the method  
  
 if the action requires the canary to be validated  
 verify that the canary in the HTTP request is identical to the canary set for the request  
 if canary verification is successful  
 indicate that the validation was successful  
 else  
 throw an exception and exit the method*  
}

The **OnActionExecuted** method performs some checks based on the type of the action result.

* If the action result is of type **RedirectResult**, the URL of the result is checked against a list of allowed Web sites. If the URL is not found in this list, the **OnActionExecuted** method throws an exception and the redirect does not occur.

***The SafeRedirectResult Class***

The Windows Live team could have added a new **SafeRedirectResult** class that checked the list of allowed Web sites and required all developers to use this instead. However, it is possible that a developer who was new to the team would not know about this rule and might use the **RedirectResult** object manually. If this code was not caught by performing a code review and testing, it could result in a security bug. This was why the team did not create a new class and instead changed the behavior of the default class.

* If the action result is of type **JSONResult**, **JavaScriptResult**, or **ContentResult**, with the content type set to **JSON** or **JavaScript**, the result is checked to ensure that canary validation has been applied. The canary validation is typically included as part of the form post when using POST actions, but it can also be part of the query string. When using GET actions, the query string must include a canary.

The following pseudo code shows the typical logic implemented by the **OnActionExecuted** method.

public override void OnActionExecuted(ActionExecutedContext filterContext)  
{  
 *identify the type of the action result (RedirectResult, JSONResult, ContentResult, …)*  
  
 *if the action result is a RedirectResult*  
 *verify that the redirect is to an approved list of domains  
  
 if the action result requires a canary  
 verify that the canary was validated by the OnActionExecuting method*  
}

# 

# Security Testing and Results

Security testing is a critically important part of the SDL process. The Windows Live test team conducted extensive security testing. The following sections describe the results of the tests that they performed.

## Testing for XSRF Vulnerabilities

The **WLXSecurity** attribute verifies that the canary is set according to the HTTP verb associated with the action (by default, a canary required for all HTTP verbs except GET). The base controller checks for the existence of a canary on all POSTs, and it prevents developers from accidentally omitting canaries or forgetting to validate them. Figure 2 shows the number of XSRF bugs found between January 1, 2008 and September 15, 2009. The number decreases markedly in 2009.

Figure 2: XSRF bugs found between January 1, 2008 and September 15, 2009

The test team could not find any XSRF bugs in code that was based on ASP.NET MVC; the base controller eliminated the possibility that developers could make this error.

## Testing for Open Redirects

The **WLXSecurity** attribute checks the URL of any actions that have a result of type **RedirectResult** against a list of acceptable domains. The redirect only occurs if the URL maps to an item in this list; otherwise, it is blocked and an exception is thrown. Figure 3 shows the number of open redirect bugs found between January 1, 2008 and September 15, 2009. Again, it is clear that this number has decreased over this time.

Figure 3: Open redirect bugs found between January 1, 2008 and September 15, 2009

Forcing developers to provide an approved list of domains that their code can redirect to reduced the number of open redirect bugs that were recorded in Windows Live during development.

## Testing for JSON Hijacking Vulnerabilities

The **WLXSecurity** attribute verifies that a canary token has been validated for the action, if the action result is of type **JSONResult**, **JavaScriptResult**, or **ContentResult** with the content type of **JSON** or **JavaScript**.

To date, the Windows Live test team has not been able to find any security bugs relating to **JSON** hijacking vulnerabilities in Windows Live.

# Lessons Learned

The experiences that the Windows Live team gained by following an SDL approach have resulted in several practices that have been adopted elsewhere within Microsoft. The following list summarizes these practices:

* **Implement security by default**. Make the default behavior secure and require extra work on the part of developers if they need to override that behavior. It also helps code reviewers to focus on code that deviates from the secure defaults.
* **Centralize security functionality**. Over time, the Windows Live team realized that the security-related code spread through many different layers, sometimes with overlapping functionality that did not always work well together. To rectify the situation, the Windows Live team consolidated all of the security-related code into a central library. This has several benefits:
  + All developers know where to find this code.
  + All developers use the latest, most secure versions of this code.
  + If there are bugs in the security-related code, they only need to be rectified once.
* **Use mandatory attributes**. ASP.NET MVC uses a declarative model with attributes that specify the requirements for methods that implement actions in the controller. The Windows Live team followed the same approach for its security model. The **AcceptVerbs** attribute is a good example of this.

The Windows Live team considered making the **AcceptVerbs** attribute optional with a default value of GET to avoid developers having to always specify it. However, the team found the code much more understandable and less prone to errors if it was required to be always present.

* **Train developers**. The Windows Live team added information about the threats faced by Windows Live services and the countermeasures that they had incorporated to the security training courses attended by all developers. In this way, all developers are aware of the underlying fundamental security issues, they understand the defenses used in the existing source code, and they follow a consistent approach to mitigating threats as they write new code.

# Acknowledgments

The authors would like to thank Andy Glover and Len Zuvela for taking the time to provide feedback on this paper.