MS_logo_KMICROSOFT SDL - DEVELOPER STARTER KIT:

SOURCE CODE ANNOTATION LANGUAGE (LEVEL 200)

Guide

Version 1.0

The following documentation provides presenter’s notes for the Microsoft Security Development Lifecycle (SDL) Source Code Annotation Language (Level 200) presentation.

For the latest information, please see [http://www.microsoft.com/sdl](http://go.microsoft.com/?linkid=9672761).

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, SQL Server, Visio, Visual Basic, Visual C#, Visual C++, Visual Studio, Windows, Windows Server, and Windows Vista are trademarks of the Microsoft group of companies.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Contents

[1.0 Security Development Lifecycle Content 2](#_Toc233628754)

[1.1 Introduction 2](#_Toc233628755)

[1.2 System Requirements 2](#_Toc233628756)

[1.3 Presentation Themes 2](#_Toc233628757)

[2.0 Microsoft SDL Buffer Overflows 2](#_Toc233628758)

[Overview 2](#_Toc233628759)

[Presentation Transcript 2](#_Toc233628760)

[Presentation Voiceover 3](#_Toc233628761)

[Presentation Demonstrations 3](#_Toc233628762)

[Slide 2 – Title Slide 3](#_Toc233628763)

[Slide 3 – Agenda 3](#_Toc233628764)

[Slide 4 – Microsoft Security Development Lifecycle (SDL) 3](#_Toc233628765)

[Slide 5 – Microsoft Source Code Annotation Language Overview 4](#_Toc233628766)

[Slide 6 – Key Security Benefits of Source Code Annotation Language 5](#_Toc233628767)

[Slide 7 – Buffer Annotations Overview 5](#_Toc233628768)

[Slide 8 – Buffer Annotations: Optional 5](#_Toc233628769)

[Slide 9 – Buffer Annotations: Indirection 6](#_Toc233628770)

[Slide 10 – Buffer Annotations: Direction 6](#_Toc233628771)

[Slide 11 – Buffer Annotations: Size 6](#_Toc233628772)

[Slide 12 – Buffer Annotations: Size Units 6](#_Toc233628773)

[Slide 13 – Buffer Annotations: Initialization 7](#_Toc233628774)

[Slide 14 – Buffer Annotations: Null-Termination 7](#_Toc233628775)

[Slide 15 – Advanced Annotations Overview 7](#_Toc233628776)

[Slide 16 – Advanced Annotations 7](#_Toc233628777)

[Slide 17 – Using Source Code Annotation Language In Your Applications 8](#_Toc233628778)

[Slide 18 –Source Code Annotation Language Demonstration 8](#_Toc233628779)

[Slide 19 –Microsoft SDL Annotation Recommendations 8](#_Toc233628780)

[Slide 20 – Conclusion 8](#_Toc233628781)

[Slide 21 - Appendix 9](#_Toc233628782)

[Slide 22 – Microsoft Security Development Lifecycle (SDL) 9](#_Toc233628783)

[Slide 23 – Microsoft Writing Secure Code Book Series 9](#_Toc233628784)

[Slide 24 – Microsoft Developer Network (MSDN) Security Developer Center 9](#_Toc233628785)

[Slide 25 – Secure Development Blogs 10](#_Toc233628786)

[Slide 26 – Hunting Security Bugs 10](#_Toc233628787)

[Slide 27 – Additional SDL Training 10](#_Toc233628788)

[Slide 28 – Source Code Annotation Language Resources 10](#_Toc233628789)

# 1.0 Security Development Lifecycle Content

## 1.1 Introduction

“The Microsoft Security Development Lifecycle (SDL) is an industry-leading software security assurance process. A Microsoft-wide initiative and a mandatory policy since 2004, the SDL has played a critical role in embedding security and privacy in Microsoft software and culture. Combining a holistic and practical approach, the SDL introduces security and privacy early and throughout all phases of the development process. It has led Microsoft to measurable and widely-recognized security improvements in flagship products, such as Windows Vista, Windows Server (2003 and 2008) and SQL Server. Microsoft is publishing the detailed SDL process guidance as part of its commitment to enable a more secure and trustworthy computing ecosystem.” -- [The Microsoft SDL 3.2 Whitepaper](http://go.microsoft.com/?linkid=9672762)

To help promote the adoption and awareness of the Microsoft SDL, Microsoft is developing content and demonstrations specifically for external developer audiences. The remainder of this document provides individuals who will present this content internally within their respective organizations with a transcript for the Microsoft SDL Training:

* Source Code Annotation Language (Level 200) presentation.

## 1.2 System Requirements

In order to use this content, a system that is capable of running [Microsoft PowerPoint 2003](http://www.microsoft.com/powerpoint) or later is required.

## 1.3 Presentation Themes

The Microsoft PowerPoint deck that accompanies this Presenter’s Guide has been intentionally provided with very limited graphics and formatting. The Microsoft PowerPoint presentation materials have been designed in this fashion to enable individuals who will present this content internally within their respective organizations to incorporate the content into custom PowerPoint themes, styles, and templates with minimal required effort.

# 2.0 Microsoft SDL Buffer Overflows

## Overview

Current static code analysis tools must make certain assumptions regarding the code they are analyzing. Whenever these assumptions are incorrect, false positives and false negatives are produced. Both of these types of findings degrade the trust a developer may have in an analysis tool, and may eventually lead to the developer not using the tool at all. Annotating source code reduces the amount of assumptions a static code analysis tool must make by explicitly stating certain function behaviors and conditions, and thus improves the overall accuracy of code analysis results.

This presentation provides an overview of the Microsoft Source Code Annotation Language, different classes of annotations and how they are used through the guidance, process and tools of the Microsoft SDL to produce safer and more trustworthy applications.

The insights gleaned by Microsoft, which are incorporated in its SDL, and more specifically, in this presentation focusing on Source Code Annotation Language, are provided as a way for external developer communities to enhance its application development practices and the security of its applications.

## Presentation Transcript

This Presentation Transcript section provides a transcript for each slide contained in the Source Code Annotation Language (Level 200) presentation. The precise transcript text provided herein is also incorporated into the notes section of each slide in the Microsoft PowerPoint Source Code Annotation Language (Level 200) presentation for ease of reference.

## Presentation Voiceover

A voiceover of the Source Code Annotation Language (Level 200) presentation transcript below, approximately 38 minutes in length is also available to assist the presenter in becoming sufficiently acclimated with the subject matter addressed in the Source Code Annotation Language (Level 200) presentation, as well as to better understand the author’s perspective behind each slide in the presentation.

## 

## Presentation Demonstrations

This presentation uses the Microsoft Virtual Labs environment to facilitate demonstrations in this presentation. Please refer to the following link for further instructions:

[MSDN Virtual Lab: Microsoft SDL Developer Starter Kit: Source Code Annotation Language](http://go.microsoft.com/?linkid=9672759)

### Slide 2 – Title Slide

The Source Code Annotation Language (Level 200) presentation introduces the role that the Microsoft Security Development Lifecycle (SDL) fulfills in trusted application development. It also provides an overview of the Microsoft Source Code Annotation Language (SAL), and how it can be used to augment the accuracy of reported vulnerabilities by static code analysis tools. A demonstration of using the Source Code Annotation Language with Microsoft Visual Studio and Microsoft PREFast will be given. This presentation will then conclude with a discussion of the annotation recommendations, as prescribed by the Microsoft SDL.

Addressing this subject matter will enable our organization to enhance our application development practices and the security of our applications.

### Slide 3 – Agenda

In this presentation, we will complete an overview of the Microsoft SDL, the Microsoft Source Code Annotation Language, annotation classes, and annotation recommendations as prescribed by the Microsoft SDL.

### Slide 4 – Microsoft Security Development Lifecycle (SDL)

The Microsoft SDL is a holistic and comprehensive approach that leverages education, process, technology and executive commitment to consistently create more secure software internally within and external of Microsoft. Since 2004, all internal Microsoft developers have been required to adhere to the SDL, and Microsoft has updated the SDL every six (6) months to address any emerging threats since its inception.

True to its name, the SDL was created to complement (rather than disrupt) the software development life cycle. The core phases and principles of the SDL include:

**Training phase:** Every Microsoft developer must complete mandatory security training focusing on secure application development practices. Training session topics include topics, such as threat modeling, secure development and testing practices, and security for application development managers.

**Requirements phase:** Requirements for security and privacy must accompany functional requirements of the software that is being created. Such requirements may include the use of encryption, authentication, and other security measures based on the business requirements, exposure and sensitive data. To that end, a security and privacy risk analysis is performed at this stage. In addition, the threshold for security and privacy (or “bug-bar”) is defined during this phase to ensure that vulnerabilities with certain severity are addressed and resolve before the software is officially released.

**Design phase:** Eradicating coding issues with security implications is not sufficient. Design vulnerabilities can have a substantial detrimental impact on security and are much more difficult to address during the verification phase. To that end, threat modeling is a critical SDL requirement and a Microsoft security innovation that is recognized by analysts as the next evolution in creating more secure software. Through threat modeling, architects and developers at Microsoft are able to approach security in a structured and methodical way from an attacker’s perspective. This allows Microsoft to identify and reduce the attack surface and mitigate the risk of potential security design issues.

**Implementation phase:** This is the application code development phase where code is written by developers using industry best practices and analyzed with both internal and externals tools (such as static code analyzers and special security debuggers) to help ensure that those best practices are being followed. Requirements are also specified by the SDL in this phase to ensure that applications are built using the latest compilers versions and built-in compiler protection features.

**Verification phase:** This is the quality assurance phase within which rigorous security testing is conducted in addition to typical functional testing procedures.

**Release phase:** The final security review is the major milestone that a Microsoft product team must pass in order to release a product under the SDL. During this meeting, security experts and the development team review all of the activities, mitigations and security artifacts that are relevant to the project in order to ensure that the security quality requirements are satisfied. During this phase, the product team defines a response plan describing procedures, accountabilities and contact information in case security vulnerabilities are discovered after the product is operational and used by customers.

**Response phase:** After an application is released, the Microsoft Security Response Center (MSRC) handles any security issues that are uncovered “in the wild” and mobilize product teams within Microsoft to provide timely fixes for security issues.

In summary, secure software development requires executive commitment, ongoing process improvement, education and training (from VPs to product managers to developers to testers), tools to aid in detecting security vulnerabilities, and incentives and consequences to ensure everyone adheres to the Microsoft SDL process.

As was previously indicated, this presentation focuses on the Microsoft Source Code Annotation Language and how annotations can be used to improve the results reported by static code analysis tools. With respect to specific phases of the Microsoft SDL, this presentation focuses on the Implementation and Verification phases.

### Slide 5 – Microsoft Source Code Annotation Language Overview

The Microsoft Source Code Annotation Language (SAL) is a set of annotations to help describe how a C/C++ function uses its arguments, the assumptions made and the execution guarantees that the function provides.

There are two implementations of the Source Code Annotation Language. The first is *\_\_declspec* syntax, which is supported in both Microsoft Visual Studio 2005 and Microsoft Visual Studio 2008. The second is *Attribute* syntax, which is supported only in Microsoft Visual Studio 2008. The examples shown in this presentation use \_\_declspec. More information regarding attribute annotation syntax can be found at <http://msdn.microsoft.com/en-us/library/ms182032.aspx>.

There are two classes of annotations: *Buffer* annotations and *Advanced* annotations. Buffer annotations provide information on how functions will use pointer arguments, whereas Advanced annotations provide additional information about functions that are not expressible by Buffer annotations. We will discuss each of these types of annotations in more detail and see how annotations can be used to improve the security of applications built using the guidance, processes and tools of the Microsoft SDL.

This presentation will provide a good overview of the Source Code Annotation Language. Additional information regarding this annotation language can be found at <http://msdn.microsoft.com/en-us/library/ms235402(VS.80).aspx>.

Lastly, the insights gleaned by Microsoft, which are incorporated in its SDL, and more specifically, in this presentation focusing on Source Code Annotation Language, are being shared with each of you as a way for our organization to enhance our application development practices and the security of our applications.

### Slide 6 – Key Security Benefits of Source Code Annotation Language

When analyzing code, static code analysis tools must make certain assumptions regarding the code being analyzed. Whenever those assumptions are incorrect, any vulnerabilities reported based on those assumptions may result in false positives. Furthermore, in the absence of detailed information about how a function uses its arguments, static code analysis tools often miss vulnerabilities; this results in the generation of false negatives. Both false positives and false negatives require extra effort from developers to investigate erroneous findings and may degrade the confidence developers have in the static code analysis tool used. With enough false positives or false negatives, developers may stop using a static code analysis tool altogether.

Source code annotations help static code analysis tools, such as the Microsoft Visual Studio */analyze* feature and Microsoft PREFast, find more vulnerabilities with less false positives and less false negatives. The additional detail provided by source code annotations enable static code analysis tools to draw more accurate conclusions regarding the source code being analyzed.

Annotating code yields benefits that extend beyond security. By annotating code, developers must document assumptions and behaviors of functions they are implementing. In some cases, annotations can help developers discover assumptions they may have incorrectly made. For developers using annotated functions, annotations can help them better understand the expected behavior of those functions and reduce any confusion.

Annotating source code requires a relatively small amount of effort and can help developers and testers find more vulnerabilities earlier in the software development lifecycle. This is beneficial because the cost of fixing a vulnerability earlier in the software development lifecycle is much less than fixing a vulnerability later in the software development lifecycle.

It should be noted that while annotating source code can significantly improve the results returned by static code analysis tools, annotations, like any other defensive measure, should not be regarded as a silver bullet. Annotations will help to find a large number of vulnerabilities, but it cannot be used to find all possible vulnerabilities.

### Slide 7 – Buffer Annotations Overview

The first class of annotations that will be discussed is Buffer annotations. This type of annotation describes pertinent information about a particular buffer argument, such as the location of the buffer, its size, and how the function uses that buffer. Annotations in this class are useful for detecting buffer overruns, which are among the most serious vulnerabilities known to date.

Annotations found within this class are divided into seven main categories. The categories are Optional, Indirection, Direction, Size, Size Units, Initialization, and Null-Termination. Each of these categories will be discussed in more detail later in this presentation.

### Slide 8 – Buffer Annotations: Optional

The first Buffer annotation category is the *Optional* category. This category indicates whether or not a buffer argument can be NULL or not. The annotation *\_opt* is used to indicate if a buffer argument is allowed to be NULL. If no annotation is specified, then it is assumed that the pointer to the buffer must not be NULL.

(Mouse click)

Here is sample code that illustrates the use of the \_opt annotation. In this sample code, a function called *OptionalSample* is declared, which takes one argument, called *buf*, which is a pointer to a *TCHAR* type. The annotation \_\_opt here indicates that buf can be null.

### Slide 9 – Buffer Annotations: Indirection

The annotations found in the Indirection category are used to describe the level of indirection of an argument or return value. If no annotation is specified, then it is assumed that the argument *p* is the buffer pointer. When the annotation *\_deref* is specified, this means that the pointer argument p is a pointer to a buffer, and that it must not be NULL. The last annotation in this category is \_deref\_opt which indicates that the pointer argument p is a pointer to a buffer, and that the pointer may be NULL.

(Mouse click)

Here is sample code that illustrates the use of the \_deref annotation found in this category. In this sample code a function called *IndirectionSample* is declared. IndirectionSample has two arguments: *cb*, which is a size\_*t* type, and *ppv*, which is a pointer whose dereference will be a buffer of type *T*. The \_deref annotation declared on the ppv argument means that ppv must not be NULL.

### Slide 10 – Buffer Annotations: Direction

The annotations found in the Direction category indicate in what direction a function will use a buffer argument. For instance, if the *\_in* annotation is specified, this will indicate that the function will only read from the given buffer and not write into the buffer. Conversely, if the *\_out* annotation is specified, this will indicate that the function intends to write into the buffer, but not read from it. If the *\_inout* annotation is specified, this will indicate that the function will read and/or write to the buffer.

(Mouse click)

Here is sample code that illustrates the use of the \_in annotation found in this category. In this sample code a function called *DirectionSample* is declared, which takes one argument called *pszString*. This annotation simply specifies that the pszString argument will only be read from and not written to by the DirectionSample function.

### Slide 11 – Buffer Annotations: Size

The annotations found in the Size category indicate the explicit counts for the size and length of a buffer argument. Annotations in the Size category are used in conjunction with other Buffer annotation categories, such as the Size Units and Initialization categories. The first annotation, (Size), indicates the total size of the buffer. The second annotation, (Size, Length), is very similar to the first annotation in this category; however, it also has a length component which indicates how much of the buffer is initialized by the function.

Examples of how to use the Size annotations will be shown when the Size Units and Initialization categories are discussed.

### Slide 12 – Buffer Annotations: Size Units

The annotations found in the Size Units category are used to describe the total size of a buffer argument. Annotations found in this category indicate whether the specified buffer size is referring to element count (the number of logical elements in a buffer) versus byte count (the actual size of the buffer). The \_ecount annotation is used to indicate element count and the \_bcount annotation is used to specify byte count. Whenever annotations from this category are used, they must also be accompanied by the \_in, \_out, or \_inout annotations, as described in the Direction category.

(Mouse click)

Here is sample code that illustrates the use of the *\_ecount* annotation from this category. In this sample code, a function called *SizeUnitsSample* is declared, which takes two arguments. The first argument consists of a pointer to a *TCHAR* type called *buf*, and the second is an integer called *cchBuf*, which is used to describe the character count of buf. To explicitly indicate this to static code analysis tools, the buf argument has been annotated with *\_\_out\_ecount(cchBuf)* annotation. Recall that annotations from different categories can be combined together, so \_\_out\_ecount(cchBuf) is actually the annotation *\_out* from the Direction category and then followed by \_ecount from the category discussed here. The cchBuf component in this annotation is taken from the Size category and indicates that the cchBuf argument holds the size of the argument buf.

### Slide 13 – Buffer Annotations: Initialization

The annotations found in the Initialization category are used to describe how much of a buffer argument will be initialized by a function. The *\_full* annotation indicates that the entire buffer will be initialized, whereas the *\_part* annotation indicates that only part of the buffer will be initialized by the function. Note that the \_part annotation can only be used with output buffers.

(Mouse click)

Here is sample code that illustrates the use of the \_part annotation from this category. In this sample code, a function named *InitializationSample* is declared, which takes three arguments. The first argument, *buf*, is annotated with *\_\_out\_ecount\_part(count, \*countOut)*. What this annotation indicates is that buf is a buffer with count elements and it will be partially initialized by the function. The length of that initialization will be specified in \*countOut.

### Slide 14 – Buffer Annotations: Null-Termination

The annotations found in the Null-Termination category are used to indicate whether the presence of a“\0” (null-terminating) character should be used to indicate the end of valid elements in a buffer. The first annotation *\_z* specifies that a null-terminating character indicates the end of the buffer. The second annotation *\_nz* does the opposite by indicating that the buffer may not be null-terminated.

(Mouse click)

Here is sample code that illustrates the use of the \_z annotation from this category. In this sample code a function called *NullTerminationSample* is declared, which takes a single pointer argument called *buf*. The variable buf is annotated with *\_\_in\_z*, which indicates that the function will only read from buf and that buf will be null-terminated.

### Slide 15 – Advanced Annotations Overview

There may be some function behaviors that are not expressible by Buffer annotations. Advanced annotations provide an additional set of annotations that can be used to enrich existing annotations or describe conditional or complex behaviors.

(Mouse click)

Advanced annotations are declared independently of Buffer annotations, as shown with this sample function called *FreeStrings*. This function requires a single argument called *strblock* which is a *LPTCH* type. The argument strblock has been marked with two annotations. The first is a Buffer annotation which is shown here in green on the left, and the second is an Advanced annotation which is shown here in yellow on the right.

### Slide 16 – Advanced Annotations

Again, advanced annotations are used to describe function behaviors that may not be expressible by buffer annotations. While these annotations may not be specifically focused towards finding buffer overruns, developers are still encouraged to use these annotations as they can assist in finding other types of code vulnerabilities.

(Mouse click)

The *\_\_checkReturn* annotation, for example, specifies that the return values from the annotated function must not be ignored by callers. Return values can provide important information about the success or failure of a function. It is important to always check return values for functions that perform write and read operations from buffers. The \_\_checkReturn annotation can help enforce this practice.

(Mouse click)

Another example is the *\_\_format\_string* annotation, which indicates that the argument is a string that contains “*%”* markers in the style of *printf*. Whenever un-trusted data is read that may contain % markers and then provided as an argument into the *printf* family of functions, a vulnerability known as a “format string vulnerability” can occur. Using this annotation can assist static code analysis tools to identify this type of vulnerability that may exist in code.

### Slide 17 – Using Source Code Annotation Language In Your Applications

To use the Source Code Annotation Language in your C or C++ applications, the first requirement is to include the *sal.h* header file. This header file contains all the macros required to use Source Code Annotation Language.

The second requirement is applications must be compiled with versions of Visual Studio that support the */analyze* compile-time flag.

The Source Code Annotation Language can also be used with Microsoft PREFast, which is a standalone static code analysis tool distributed with the Microsoft Device Driver Kit (DDK). Additional information about PREFast can be found at <http://www.microsoft.com/whdc/DevTools/tools/PREfast.mspx>.

### Slide 18 –Source Code Annotation Language Demonstration

Let’s now take a look at a demonstration of the Source Code Annotation Language, and how it can be used to help static code analysis tools better detect vulnerabilities.

(Start Source Code Annotation Language demonstration)

### Slide 19 –Microsoft SDL Annotation Recommendations

The use of Source Code Annotation Language, *\_\_declspec* syntax or *attribute* syntax, is not formally required for applications developed in alignment with the Microsoft SDL. The following are annotation recommendations as prescribed by the Microsoft SDL to help produce safer and more trusted applications.

The first is to start annotating new code. Any new code developed should be annotated to improve results from static code analysis tools. Existing code should also be annotated over time.

The second recommendation is to annotate the prototypes of functions that write to buffers. Functions that write to buffers are highly susceptible to buffer overflow vulnerabilities and should be annotated whenever possible.

The third recommendation is to annotate the prototypes of functions that read from buffers. Reading from buffers is not as prone to buffer overflow vulnerabilities as writing to buffers; however, reading from buffers can significantly affect the security posture of an application. For example, an application function that tries to dereference a pointer to a buffer without checking first to ensure that the pointer is non-null can crash. This crash may cause the application to be non-responsive to legitimate users, therefore resulting in what is referred to as a “denial of service (DoS)” attack.

Finally, use annotated headers whenever possible. Any code that references an annotated function will also gain the benefits of annotation. Microsoft has annotated the C runtime functions found in Visual Studio 2005, Visual Studio 2008 and in the latest Windows Software Development Kit to help developers developing on Microsoft platforms produce safer and more trustworthy code.

### Slide 20 – Conclusion

This concludes the discussion on the Microsoft Source Code Annotation Language (SAL). Current static code analysis tools must make certain assumptions about the code that they are analyzing. Whenever those assumptions are incorrect, the results reported by the analysis tools may be inaccurate or missing. In either case, these erroneous results degrade the confidence developers have in their static analysis tools, and may eventually result in those tools not being used at all.

The Microsoft Source Code Annotation Language (SAL) provides a set of annotations to describe how a function uses its arguments, the assumptions made about those arguments, and the guarantees provided by the function. This information can be leveraged by static code analysis tools to provide more accurate findings. Annotating code requires minimal upfront effort, and can help developers find vulnerabilities earlier in the software development lifecycle where the cost of addressing such vulnerabilities is much lower. While the Source Code Annotation Language will not enable static code analysis tools to uncover all possible vulnerabilities in C/C++ applications, it can be used as an effective way to enhance those tools.

There are currently two classes of annotations found in the Source Code Annotation Language: buffer annotations and advanced annotations. While the Buffer annotations are designed to help detect buffer overflow vulnerabilities, advanced annotations should still be used as they can describe complex information about functions not expressible by Buffer annotations. Furthermore, advanced annotations can help enforce certain best-practices when using high-risk functions and can assist in detecting other vulnerability types.

Applications that are developed using the Microsoft SDL are currently not required to use the Source Code Annotation Language. Microsoft, however, still encourages developers to use annotations along with other defensive measures to help them develop safer and more trusted applications on Microsoft platforms and technologies.

Lastly, the insights gleaned by Microsoft, which are incorporated in its SDL, and more specifically, in this presentation which focused on Source Code Annotation Language, have been shared with each of you as a way for our organization to enhance our application development practices and the security of our applications.

### Slide 21 - Appendix

This section provides additional slides, materials, and information to supplement the main contents of the presentation.

### Slide 22 – Microsoft Security Development Lifecycle (SDL)

This diagram compares the security engineering steps of the SDL to the software engineering steps of the classic SDLC (software development lifecycle). The blue outer ring represents traditional software development and the orange inner circle represents the SDL. Notice that the security engineering steps are incorporated into the existing software engineering steps and that any engineering task can be supplemented with a security engineering task.

Both of these development lifecycles, or collections of engineering steps, apply to the software development lifecycle regardless of the particular development model you use (for example waterfall, Agile, etc.) The small pewter colored circles represent the various milestones in your model and are an excellent time for ensuring that the steps in both the security and software development lifecycles have been adequately addressed.

The SDL process has been documented and published in *The Security Development Lifecycle* book (Microsoft Press 2006, ISBN: 9780735622142), and the official Web site can be accessed at [http://www.microsoft.com/sdl](http://go.microsoft.com/?linkid=9672761).

### Slide 23 – Microsoft Writing Secure Code Book Series

Microsoft has several publications on secure implementation including the industry leading Writing Secure Code series. Writing Secure Code is mandatory reading for software engineering teams at Microsoft and provides an in-depth discussion of common software weaknesses and effective remedies.

It also provides information with which testers can use to better ensure that the applications they are testing meet security quality assurance requirements.

### Slide 24 – Microsoft Developer Network (MSDN) Security Developer Center

Microsoft also has a security developer center located at [http://msdn.microsoft.com/security](http://go.microsoft.com/?linkid=9672763) where development teams (architects, developers and testers) can find a wealth of resources, including guidance and tools, to help them build safer applications using Microsoft technologies and platforms.

### Slide 25 – Secure Development Blogs

Visit the [SDL Blog](http://go.microsoft.com/?linkid=9672765) to get the most current ideas and thoughts from Microsoft SDL team members.

Visit [Michael Howard’s Blog](http://go.microsoft.com/?linkid=9672764) to read all about how security can be effectively incorporated into the software development process from the author of the popular book, *Writing Secure Code* (Howard, Michael and David LeBlanc, Microsoft Press, Redmond, Washington, 2003).

### Slide 26 – Hunting Security Bugs

Members of the Microsoft Office Security team have written a book that covers common application security issues and how to test for them. More information about this book can be found at [http://www.microsoft.com/mspress/books/8485.aspx](http://go.microsoft.com/?linkid=9672768).

### Slide 27 – Additional SDL Training

Additional SDL training content, such as the following is currently or will be available soon:

**Secure Design Principles:** This content provides application designers with the fundamentals and principles they require to design more secure applications. Other content related to secure design builds upon the knowledge established in this content.

**Secure Implementation Principles:** This content provides developers with the fundamentals and principles they require to develop more secure applications. Other content related to secure implementation builds upon the knowledge established in this content.

**Secure Verification Principles:** This content provides testers and quality assurance personnel with the fundamentals and principles they require to test secure applications. Other content related to secure testing builds upon the knowledge established in this content.

**SQL Injection Vulnerabilities:** SQL injection vulnerabilities are commonly encountered vulnerabilities in applications using a database. As more applications move towards the Web paradigm and are driven by databases, this vulnerability is expected to become even more prolific than is currently being realized. This content provides an overview of SQL injection vulnerabilities and how the SDL can be used to significantly reduce the risk of a SQL injection attack.

**Cross-Site Scripting Vulnerabilities:** Cross-site scripting vulnerabilities are the most commonly encountered Web-based vulnerabilities today. These types of vulnerabilities continue to plague the Web-application world and a user’s ability to trust the applications they are using. This content provides an overview of cross-site scripting vulnerabilities, and how the SDL can be applied to significantly reduce the risk of a cross-site scripting attack.

**Buffer Overflow Vulnerabilities:** Buffer overflows are considered the most dangerous application-level vulnerability. This content provides an overview of buffer overflows, and how the SDL can be used to significantly reduce the risk of a buffer overflow attack.

### Slide 28 – Source Code Annotation Language Resources

Here are additional resources and links on the Microsoft Source Code Annotation Language.