MS_logo_KMICROSOFT SDL - DEVELOPER STARTER KIT:

SECURE IMPLEMENTATION PRINCIPLES (LEVEL 100)

Guide

Version 1.0

The following documentation provides presenter’s notes for the Microsoft Security Development Lifecycle (SDL) Secure Implementation Principles (Level 100) 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](#_Toc233622115)

[1.1 Introduction 2](#_Toc233622116)

[1.2 System Requirements 2](#_Toc233622117)

[1.3 Presentation Themes 2](#_Toc233622118)

[2.0 SDL Secure Implementation Principles 2](#_Toc233622119)

[Overview 2](#_Toc233622120)

[Presentation Transcript 2](#_Toc233622121)

[Presentation Voiceover 2](#_Toc233622122)

[Slide 2 – Title Slide 3](#_Toc233622123)

[Slide 3 – Agenda 3](#_Toc233622124)

[Slide 4 – Microsoft Security Development Lifecycle (SDL) 3](#_Toc233622125)

[Slide 5 – SDL Secure Implementation Overview 4](#_Toc233622126)

[Slide 6 – Input Validation Tips 4](#_Toc233622127)

[Slide 7 – Buffer Overflows 5](#_Toc233622128)

[Slide 8 – Stack Overflows at Work 5](#_Toc233622129)

[Slide 9 – The C/C++ N-Functions Are Safe Right? 6](#_Toc233622130)

[Slide 10 – Buffer Overflow Remedies 6](#_Toc233622131)

[Slide 11 – Integer Arithmetic Errors 7](#_Toc233622132)

[Slide 12 – Integer Arithmetic Error Example 7](#_Toc233622133)

[Slide 13 – Integer Arithmetic Error Remedies 7](#_Toc233622134)

[Slide 14 – Canonicalization Issues 7](#_Toc233622135)

[Slide 15 – Canonicalization Example 8](#_Toc233622136)

[Slide 16 – Canonicalization Issue Remedies 8](#_Toc233622137)

[Slide 17 – Cross-Site Scripting (XSS) Issues 8](#_Toc233622138)

[Slide 18 – Cross-Site Scripting Example: Cookie Stealing 9](#_Toc233622139)

[Slide 19 – Cross-Site Scripting Remedies 9](#_Toc233622140)

[Slide 20 – SQL Injection Issues 10](#_Toc233622141)

[Slide 21 – SQL Injection Example 10](#_Toc233622142)

[Slide 22 – SQL Injection Example 10](#_Toc233622143)

[Slide 23 – SQL Injection Remedies 11](#_Toc233622144)

[Slide 24 – Cryptographic Weaknesses 11](#_Toc233622145)

[Slide 25 – Top Common Cryptographic Mistakes 11](#_Toc233622146)

[Slide 26 – Remedies for Top Common Cryptographic Mistakes 12](#_Toc233622147)

[Slide 27 – Beware, Not All Cryptographic Standards Are Safe! 12](#_Toc233622148)

[Slide 28 – Conclusion 12](#_Toc233622149)

[Slide 29 - Appendix 13](#_Toc233622150)

[Slide 30 – Microsoft Security Development Lifecycle (SDL) 13](#_Toc233622151)

[Slide 31 – Microsoft Writing Secure Code Book Series 13](#_Toc233622152)

[Slide 32 – Microsoft Developer Network (MSDN) Security Developer Center 13](#_Toc233622153)

[Slide 33 – Secure Development Blogs 13](#_Toc233622154)

# 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:

* Microsoft SDL Training – Secure Implementation Principles (Level 100)

## 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 SDL Secure Implementation Principles

## Overview

Safer and more trusted applications begin with designs that are sensitive to security and privacy concerns. Developers then take those designs and implement (or develop) them into actual code. Those security and privacy design considerations can be easily lost and allow vulnerabilities to manifest in code if developers do not follow certain industry implementation best-practices.

This presentation provides an overview of common secure implementation issues and how the guidance, tools and requirements of the Microsoft SDL addresses each of these issues.

The insights gleaned by Microsoft, which are incorporated in its SDL, and more specifically, in this presentation focusing on Secure Implementation Principles, 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 of this document provides a transcript for each slide in the Secure Implementation Principles (Level 100) presentation. The precise transcript text provided herein is also incorporated into the notes section of each slide in the Microsoft PowerPoint Secure Implementation Principles (Level 100) presentation itself for ease of reference.

## Presentation Voiceover

A voiceover of the Secure Implementation Principles (Level 100) presentation transcript below, approximately 36 minutes in length is also available to assist the presenter in becoming sufficiently acclimated with the subject matter addressed in the Secure Implementation Principles (Level 100) presentation, as well as to better understand the author’s perspective behind each slide in the presentation.

### Slide 2 – Title Slide

The Secure Implementation Principles (Level 100) presentation introduces the role that the Microsoft Security Development Lifecycle (SDL) fulfills in trusted application implementation and provides an overview of some of the common secure implementation issues addressed by the SDL, including a) buffer overflows; b) integer arithmetic errors; c) canonicalization issues; d) cross-site scripting issues; e) SQL injection issues; and f) cryptographic weaknesses.

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

*Note:* This is a level 100 presentation meant to familiarize you with security implementation fundamentals and principles. These fundamentals and principles will be built upon in later SDL presentations.

### Slide 3 – Agenda

In this presentation we will complete a high-level overview of the SDL, an overview of secure implementation and an introduction of common secure implementation issues. We will also briefly look at how each of these issues is addressed by the SDL to help Microsoft better deliver safer and more trusted applications to its customers since the inception of the SDL in 2004.

### 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 bugs with certain severity are addressed and resolve before the software is officially released.

**Design phase:** Eradicating coding bugs 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 SDL process.

As was previously indicated, this presentation focuses on the secure implementation principles of the SDL. Specifically, this presentation focuses on principles and common issues related to the SDL Implementation phase.

### Slide 5 – SDL Secure Implementation Overview

Developers have the role of taking an application design and implementing it into functional code. Whenever certain secure implementation best practices are not followed, vulnerabilities in the application code implementation are created and can allow a malicious user to compromise an application and users.

Secure implementation issues can be categorized broadly into two categories. The first category includes issues that stem from failing to validate input, and the second category is everything else. Trusting the state of input and using it in an application without proper validation is the root cause of *many* vulnerabilities. For example, buffer overflows, integer arithmetic errors, cross-site scripting issues, SQL injection issues, all which will be discussed in this presentation, are all caused by some form of malicious input. Other secure implementation issues that do not stem from failing to sufficiently validate input include, but not limited to, issues such as improper access controls and improper use of, or weak, cryptography algorithms.

Lastly, the insights gleaned by Microsoft, which are incorporated in its SDL, and more specifically, in this presentation focusing on Secure Implementation Principles, 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 – Input Validation Tips

The best approach to input validation is to take the defensive approach and assume that all input contains malicious attack data until proven otherwise. At Microsoft, the motto amongst developers and secure implementation is “all input is evil, until proven otherwise” (with input validation).

Any data that crosses a trust boundary and is used within an application you are developing should be validated. Examples of data that cross trust boundaries include data from users (e.g., malicious or legitimate users) and even other systems (e.g., your competitor’s systems). Remember that an attack can originate from a human or a computer adversary. A good way to determine trust boundaries is through a technique used at Microsoft and mandated by the SDL called threat modeling. Threat modeling is the topic of another presentation, but briefly threat modeling lets you decompose your application into separate components and lets you analyze the trust boundaries between separate components and the data flowing between them.

Finally when performing input validation, validate against expected data formats, lengths, types and ranges. Do not look for specific attack data which is called the black-listing technique. Black-listing requires that you know all possible attack patterns which is extremely difficult to perform correctly. Rather, use the white-listing approach which compares input data against expected data formats, lengths, types and ranges. Any data that does not match the expected data patterns is considered malicious and should be safely rejected by the application.

### Slide 7 – Buffer Overflows

One of the best and most widely recognized examples of secure implementation issues that manifest whenever inputs are not validated are buffer overflows. A buffer overflow results when you allocate a fixed amount of memory and then write more data into that buffer than it can actually hold. Overflowing a buffer can modify sensitive in-memory constructs that control execution flow. A malicious user can craft malicious inputs to overflow a buffer in such a way that the application executes the code supplied by the malicious user or crashes the application to make it unavailable to other users. Internet worms such as the SQL Slammer and Zotob worm are examples of buffer overflow attacks.

Buffer overflows are a big problem in certain programming languages, such as C and C++ and much less of a problem in managed languages like .NET and Java. However, you cannot assume that using a managed language provides immunity from buffer overflow attacks. The runtime environment itself (e.g., JRE or .NET runtime) is written in native code and may contain a buffer overflow that could be exploited via your code. This is not something that occurs often, but is still something to be aware of. More problematic are buffer overflows in unsafe managed code. For instance calling into native code through interoperability services, or using .NET code wrapped in UNSAFE blocks. These situations can definitely be dangerous and require lots of care. They are basically, just as susceptible to buffer overflow attack as a normal native application is.

Finally, buffer overflows can occur on application stacks and heaps which will be discuss next.

### Slide 8 – Stack Overflows at Work

The code shown here contains a buffer overflow on an application’s stack. It obtains 128 characters worth of memory, but does not check the length of the character string before writing to that memory. On many computer architectures, this buffer of memory will be allocated on a stack with the return address to the calling function immediately following the arguments to the function.

(Mouse click)

Therefore after copying this string, the computer uses the “address” following the function arguments, which could now be data supplied by a malicious user as the point from which to continue executing instructions.

(Mouse click)

The malicious user merely writes data in this “address” to point to the rest of the malicious user’s input, making the malicious user’s code be what the computer executes when the function returns. The malicious in other words now has complete control of the application. This is very bad!

(Mouse click)

Buffer overflows can also occur on application heaps. Heap overflows work in a similar fashion as stack overflows, but instead of overwriting execution flow control data on the stack, data used to manage the heap structure itself is overwritten. This can also lead to a malicious user being able to control an application’s execution flow. Heap overflows will not be discussed in more detail in this presentation; however both are addressed by the SDL which will be discussed later.

### Slide 9 – The C/C++ N-Functions Are Safe Right?

The C/C++ n-functions are often used by developers to control the length of data being written from one buffer to another. A common misconception amongst developers is that the use of these C/C++ n-functions renders an application immune from buffer overflows. This is incorrect. Here are several examples where the use of certain C/C++ n-functions can still result in a buffer overflow:

(Mouse click)

**Example 1:**

This example illustrates a common mistake using the C/C++ n-functions and that is using a constant (either from a header file, or as in this case defined in the code) that is not the same as the actual size of the data type. The variable pszSrc is a char \* defined elsewhere. In this example malloc creates a buffer of size 4; however strncpy copies at most 50 bytes into this 4 byte buffer resulting in a potential buffer overflow condition.

(Mouse click)

**Example 2:**

This example illustrates another common mistake using the C/C++ n-functions. The function strncpy does not null terminate the szDest buffer because the source buffer is as long as the destination buffer – this behavior is defined in the C99 or ISO/IEC 9899:1999 specification. Since the buffer is not null terminated it can cause unexpected application behavior and could also cause a subsequent buffer overflow condition if the buffer is copied into another buffer without restricting the size of that copy operation.

(Mouse click)

The C/C++ n-functions are useful to restrict the size of data being operated on; however without extra care from the developers these functions can just as easily facilitate buffer overflow attacks as their non-n counterparts. The n-functions require that developers have a deep understanding of the functions which makes using them correctly difficult. Applications development teams at Microsoft are prohibited from using the C/C++ n-functions by the SDL and are required to use safer alternatives such as the StrSafe and SafeCRT libraries.

### Slide 10 – Buffer Overflow Remedies

How do you find and fix buffer overflow vulnerabilities? Here are some of the tools, techniques and libraries developers can use to address the problem of buffer overflows as prescribed by the SDL:

* Reduce Attack Surface – This technique helps prevent buffer overflow attacks by reducing the possible application vectors that can be exploited by a malicious user.
* Search for Risky Functions – Reviewing your application code for weaknesses that could facilitate buffer attacks before releasing it to customer can save you significant development costs fixing buffer overflows after release. Code review is not limited to buffer overflows only. It is useful for identifying other types of secure implementation issues.
* Use Safer Libraries – Using safer libraries gives developers less opportunity to make coding mistakes that could lead to buffer overflows.
* /GS, NX and Heap Checking – Current Microsoft compilers provide protection and features such as /GS and heap checking that can detect certain buffer overruns during application run time. These protection features give errors instead of allowing an exploit to run successfully. The Microsoft SDL requires that certain flags such as the /GS flag are enabled for all applications developed with the SDL.
* PREFast and SAL – Code scanning tools and source code marking techniques such as Microsoft PREFast and the Source Code Annotation Language (SAL) featured in higher versions of Visual Studio can help detect vulnerabilities through static code analysis
* Fuzz Testing – Fuzz tests can run in an automated fashion and find large numbers of vulnerabilities with very little engineering effort. This testing technique is introduced in the secure testing principles presentation.

### Slide 11 – Integer Arithmetic Errors

Buffer overflows manifest whenever buffer data is not first validated before being used by applications. Similar conditions can be reproduced if developers are not careful when performing integer arithmetic. This category of coding errors is known at Integer Arithmetic Errors and can lead to vulnerabilities such as integer overflows/underflows, signed versus signed errors and truncation.

### Slide 12 – Integer Arithmetic Error Example

*Note that this example was adapted from Microsoft Writing Secure Code, 2nd Edition* ([*http://www.microsoft.com/mspress/books/5957.aspx*](http://www.microsoft.com/mspress/books/5957.aspx))

Here is an example of an integer arithmetic error that can lead to a stack-based buffer overflow condition. The code on this slide is a C language function called Example that takes two arguments. The first is a char \* argument called str, which is the data to copy, and the second is an integer argument called size, which represents the size or length of str. The code allocates a fixed buffer of 80 bytes and checks to ensure that the size of the intended source buffer (str) is less than the size of the intended destination buffer (buf). If it is then it assumes that it is safe to copy the data from str into buf and proceeds with the call to strcpy.

There is a significant problem with this code from a secure implementation perspective. The problem is that the code incorrectly assumes that the size argument will never be a negative value. Look at what happens if for example size is passed in as -10. The check done by the if statement will evaluate to true because -10 (an integer) is less than sizeof(buf) which returns 80. Technically, the function sizeof(buf) will return an unsigned integer which most compilers will cast back to an signed integer because it is being compared with a signed integer. Strcpy then copies the data from str into buf and if the length of str exceeds the capacity of buf then a buffer overflow condition will occur.

Again, this is an example of where a simple integer arithmetic error can lead to a buffer overflow condition.

### Slide 13 – Integer Arithmetic Error Remedies

Fortunately there are several actions developers can take to address the threat of attacks based on integer arithmetic errors.

The first is to closely examine any calculation used to determine an array offset or memory location. Code review and code scanning tools can provide help in this effort.

The second is to use unsigned variables for all array indexing and buffer sizes. Logically indexed data should not have negative indexes and so the use of integers when indexing into arrays should be avoided.

Lastly, code compilers can provide indication of possible integer arithmetic errors. The Microsoft Visual C++ compiler for example emits warning messages such as the following whenever it detects potential integer arithmetic errors in code:

* C4018 and C4389. These two warnings are generated whenever there is a sign/unsigned mismatch and the compiler needs to perform some signed/unsigned conversion.
* C4288. This warning is generated whenever one type needs to be converted into another type and that conversion could result in a loss of data
* #pragma casts. #pragma casts can be used to silence warnings from the Visual C/C++. Silencing the compiler could mask useful warning messages that could provide clues to potential secure coding issues.

### Slide 14 – Canonicalization Issues

Data can be represented in many different forms but still hold the same meaning. For instance, the word dog can be represented in French, English, and Japanese and so on, but each of those translations has the same meaning.

In application security, a canonicalization issue occurs whenever a security check is done on data that is represented in one form, and then used by an application in another form. A few examples include data that is escaped, doubly escaped or even triply escaped.

Malicious users can often use canonicalization issues in code to bypass weak security checks and in the next slide you will see an example of this.

### Slide 15 – Canonicalization Example

Here is an example of a common canonicalization issue and how a malicious user can use this issue to bypass security checks. On Windows file systems, these 4 names can all refer to the exact same file.

* **SecretFile.txt**: This is the name of the file that is being protected in this example.
* **SecretFile.txt.**: Notice the trailing period on this second form. The Microsoft Win32 file system determines that the trailing dot (.) should not be there and removes it.
* **Secret~1.txt**: The third form is used by FAT (file allocation table) systems. Legacy FAT systems require that file names to have a maximum of eight characters, followed by three characters. Later file systems, such as FAT32 and NTFS (NT File system), allow longer file names. For backwards compatibility, later systems automatically generate an 8.3 format file names that includes the first six legal characters, followed by a tilde (~) and an incrementing digit.
* **SecretFile.txt::$DATA**: This last form is used by the NTFS file system which supports a feature called file streams to store extra data about a file such as custom properties. The “::$DATA” simply indicates to the file system to return the default data stream or contents for the file.

(Mouse click)

Here is some sample code that reads a filename from the user and grants access to the file only if the filename is not SecretFile.txt. Notice that it only looks for the filename “SecretFile.txt” and none of the variants shown above.

(Mouse click)

Because the code is only checking for one form of the filename, a malicious user could specify the same file using the 2nd, 3rd or 4th representation of the file SecretFile.txt and bypass the security check of this code.

### Slide 16 – Canonicalization Issue Remedies

The only way to truly avoid this type of secure implementation issue is to understand what the final form of the data will be and make sure you are using that form when you perform your check. In other words, canonicalize the data into a standard form and then make a security decision based on that standard form.

Another tip is to avoid making security decisions based on names altogether. Let the operating system do all the work for you and leverage built in authorization and authentication systems.

Finally, if you need to make name-based security decisions, then use regular expressions to restrict the type of data allowed in a valid name and validate data names in canonicalized forms with these regular expressions and other security criteria.

### Slide 17 – Cross-Site Scripting (XSS) Issues

Cross-site scripting (XSS) seems to be one of the most recurrent issues in Web applications. In fact, according to the Open Web Application Security Project (OWASP), cross-site scripting vulnerabilities are the number one Web vulnerability today. Although it is fairly simple issue to comprehend it can be difficult for developers to prevent because of the multiple contexts in which information can be displayed in a web application.

Cross-site scripting vulnerabilities in a Web application can lead to a compromise of the client using the application such as the theft of cookies and temporary Web content modification.

A Web application can contain a cross-site scripting vulnerability if it reads input data into the application and does not validate that input data before echoing it as part of its response. Any input that comes from the client can be used to inject script in an application page such as textboxes, text areas and even hidden fields and drop-downs. All parameters that are sent to the application through GET or POST requests can be modified using various Web proxies and tools used to conduct this type of injection attack.

### Slide 18 – Cross-Site Scripting Example: Cookie Stealing

Here is an example of how a malicious user can use a cross-site scripting vulnerability to steal the cookie of another user using a vulnerable Web application.

(Mouse click)

The vulnerable web application in this example takes input that is not validated and is echoed back in HTML response data. Note that the input can be controlled by a malicious user.

(Mouse click)

In this case the attack can be carried out by modifying GET request parameters in a URL, so a link can be provided that exploits the vulnerability for example by email. All it needs is to be clicked on!

(Mouse click)

When the link is clicked on, the exploit inserts the attacker’s own JavaScript into the document, which is executed in the same domain as the vulnerable Web site! The code has access to the user’s cookie, and using a simple IMG tag, sends the attacker the user’s cookie data and saves that data elsewhere on a server they control.

(Mouse click)

Our victim, Mary, clicks the link in her email inbox to claim her million dollar prize, and unknowingly sends her cookie to the attacker. You can tell something is not quite right, and you can see the URL Encoded attack string in the Address Bar. A more clever attack string could execute the attack with little or no visible indication.

### Slide 19 – Cross-Site Scripting Remedies

As with all the other security implementation issues discussed in this presentation thus far, the key technique to addressing this particular issue is to validate all inputs into a Web application including that which gets echoed back as Web response data. This will help ensure that potential attack data cannot be echoed back in Web responses, and will reduce the likelihood that a cross-site scripting attack will succeed.

In addition to input validation, all input data that gets echoed back as part of Web responses should also be encoded. What encoding does it is takes potentially executable Web code or script and turns it into equivalent, but non-executable forms. The Microsoft developer platform provides numerous encoding methods that can be used to provide protection from cross-site scripting attacks such as those found in the .NET System.Web.HttpUtility class.

The HttpOnly cookie option prevents access to cookie data via the document.cookie property from client-side script in Internet Explorer 6 SP1 and later.

Internet Explorer 6 and later support a new security attribute for the <**frame**> and <**iframe**> elements. You can use the security attribute to apply the user's Restricted Sites Internet Explorer security zone settings to an individual frame or iframe. By default, the Restricted Sites zone does not support script execution.

Finally, ASP.NET provides some limited built-in protection against cross-site scripting attacks through the validateRequest page attribute. This feature will check all inputs to an ASP.NET page for certain cross-site scripting attacks and will throw a page exception if found. It is important to note that this feature provides only limited protection. Developers should still follow security development best practices, such as validating input and encoding Web output that may contain un-trusted input data.

Microsoft has provided developers with guidance on how to prevent cross-site scripting in ASP.NET at <http://msdn.microsoft.com/en-us/library/ms998274.aspx>.

### Slide 20 – SQL Injection Issues

SQL injection like cross-site scripting is another common vulnerability that often plagues Web-based applications. Like all the other security implementation issues discussed so far, SQL Injection attacks are possible whenever input from an un-trusted source is not validated and is inserted into strings that are later passed to a database engine for parsing and execution.

SQL injection attacks allow malicious users to control the SQL commands executed by the database and allows them to do such things as modify table data and execute commands from the database server.

### Slide 21 – SQL Injection Example

Let’s look a very simple example of a Web application where a user can enter a shipping id to track an online order. The user enters their tracking ID and behind the scenes, the following SQL query retrieves the appropriate shipping data.

(Mouse click)

Let’s see how a malicious user could potentially exploit this scenario.

### Slide 22 – SQL Injection Example

Above is the SQL query statement that is running behind the scenes. Let’s look at different input scenarios, the resulting query that gets executed by the database engine and how failing to validate the input can lead to exploitable conditions.

(Mouse click)

In the first scenario, input ID is set to 1000. The database simply executes SELECT \* FROM ShipmentOrders WHERE ID=‘1000’; and retrieves the data for order corresponding to ID 1000 if it exists. This is the expected behavior we want. Now let’s look at some more interesting ID inputs and see how they can be used by malicious users to perform unauthorized actions.

(Mouse click)

In the second scenario, ID is specified by the malicious user as 1000, but some other data is also included. Notice here the malicious user is entering the shipment ID as 1000’; DROP TABLE ShipmentOrders;--. When this gets inserted into the original SELECT statement above, the database server actually sees two different SQL queries to execute. The first is to retrieve the information for the order corresponding to ID 1000, and the second is to drop the table labeled ShipmentOrders. If this were a real-life situation, then all shipment data stored in that table would be deleted. This is bad!

(Mouse click)

In the third and last scenario, ID is specified by the malicious user as 1000’; exec xp\_cmdshell and then some command to execute it. As with the second scenario, the database server sees two different SQL queries to execute. The first is to retrieve information for the order corresponding to ID 1000 and the second is to execute the xp\_cmdshell stored procedure and have it execute any command that the malicious user provides. This is really, really bad!

***Note:*** Double dashes (--) is the equivalent of comment characters in a programming language like C#, C or C++. It indicates to the database engine that anything after the double dashes should be considered as a comment and not executed. In the attacks above, the additional single (‘) quote that the malicious user inserted would have imbalanced the number of quotes in the executing SQL query. The double dashes was used to comment out the final quote and semi-colon that would have imbalanced the SQL query, caused a database error and prevented the attack from succeeding.

### Slide 23 – SQL Injection Remedies

Here are several actions that developers can take to reduce the threat of SQL injection attacks.

As with all the previously discussed secure implementation issues, all input should be validated for type, length, format and range.

The second approach is to use parameterized queries. In parameterized queries, input data is specifically labeled as a parameter which indicates to the executing database server that the input should be treated strictly as data and not as executable statements.

In addition to input validation and parameterized queries, developers should consider allowing access to stored procedures (using parameterized queries) and not directly to the tables those procedures operate on. This will limit the level of data access a malicious user can possible have.

Again, this portion of the presentation was meant to be an introduction to common secure implementation issues such as SQL Injection. A more in-depth discussion on this particular issue is available.

### Slide 24 – Cryptographic Weaknesses

Often applications need to store sensitive information such as passwords and user data. Cryptography, which is the process and the study of hiding secrets, is often used by developers to do this. However, using cryptography correctly can be difficult and when used incorrectly can leave sensitive application data at risk.

Let’s take a look at some common application development cryptographic mistakes.

### Slide 25 – Top Common Cryptographic Mistakes

Some common cryptographic usage mistakes include incorrectly storing secrets inside applications, and developers creating their own custom cryptographic algorithms.

(Mouse click)

The first mistake is storing secrets inside an application. Application source code cannot defend itself, and so therefore secrets stored inside that source code is impossible to defend. A malicious user who gets access to the source code or to the memory of a machine that the application is running on (or recently ran on) will be able to easily retrieve the secret data!

(Mouse click)

The second most common mistake is creating and using custom cryptographic algorithms. Standard cryptographic algorithms undergo several years of review and effectiveness testing by academic and industry experts before they are even considered for standard approval. One can certainly create their own cryptographic algorithms, but those self-created algorithms would not have undergone the same rigorous testing and scrutiny that the standard algorithms will have. And in the absence of that rigorous industry expert review it is extremely difficult to be assured of the robustness and safety of that algorithm.

### Slide 26 – Remedies for Top Common Cryptographic Mistakes

The Microsoft SDL requires that any application storing secrets such as keys, passwords and other secret data must do so using the Microsoft Data Protection API (DPAPI). The Data Protection API provides easy access to platform provided cryptographic services and provides services useful for protecting application secrets such as key generation, integrity checking and others.

To address the common mistake of developers creating their own cryptographic algorithms, the SDL requires that application development teams only use approved cryptographic standards implemented on Windows platform such as the libraries found in the .NET Framework namespace System.Security.Crytography and native code libraries like CryptoAPI (CAPI) and CryptoAPI Next Generation (CNG).

### Slide 27 – Beware, Not All Cryptographic Standards Are Safe!

Even if a cryptographic algorithm has been approved as a standard, this does not necessarily mean that the algorithm is safe for use. Several algorithms since their approval as a standard has been shown to be weak or ‘broken’ by researchers. Examples include MD5 (used to hash data) and DES (used for symmetric data encryption). Both have been shown to be weak and have been deprecated for use by the industry and government.

The Microsoft SDL provides guidelines as to the cryptographic algorithms that are no longer considered safe, and the appropriate replacements such as the Advanced Encryption Standard (AES) or sometimes referred to as Rijndael. All new applications developed within Microsoft using the SDL must follow these guidelines.

### Slide 28 – Conclusion

This concludes the discussion on the SDL Secure Implementation Principles. In this presentation we looked at an overview of the SDL and the important role it plays in the implementation stage of an application’s software development lifecycle. We discussed the importance of validating input and how failure to do so can lead to a majority of known application security issues such as buffer overflows and SQL injection.

We then briefly discussed some of the common secure implementation issues addressed by the SDL which were:

* **Buffer Overflows.** This secure implementation issue allows a malicious user to control the execution flow of an application and is responsible for some of the most devastating Internet worm incidents to date.
* **Integer Arithmetic Errors.** This secure implementation issue involving integer arithmetic errors that could allow malicious users to control the execution flow of an application in a similar fashion as buffer overflows.
* **Canonicalization Issues.** Canonicalization issues stem from the fact that the same data can be represented in various forms and that security checks that do not take these various forms in account can be easily bypassed by malicious users.
* **Cross-Site Scripting.** Cross-site scripting attacks rely on un-validated malicious user controlled data being echoed in Web responses. This developer oversight allows attacks to execute code on the client browser and facilitate attacks such as stealing cookie data.
* **SQL Injection.** This common secure implementation issue manifests whenever data controlled by a malicious user is not validated and is used as part of a SQL query. Malicious users can use this attack technique to control the query being executed by the database server and perform unauthorized actions (e.g., modify data and execute commands).
* **Cryptographic Weaknesses.** This last secure implementation issue did not involve the lack of input validation, but rather focused on the common mistakes that developers make when using cryptography to protect secret data in applications.

Lastly, the insights gleaned by Microsoft, which are incorporated in its SDL, and more specifically, in this presentation which focused on Secure Implementation Principles, 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 29 - Appendix

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

### Slide 30 – 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 31 – 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.

### Slide 32 – 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 developers can find a wealth of resources, including guidance and tools, to help them build safer applications using Microsoft technologies and platforms.

### Slide 33 – 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).