MS_logo_KMICROSOFT SDL - DEVELOPER STARTER KIT:

SECURE VERIFICATION PRINCIPLES (LEVEL 100)

Guide

Version 1.0

The following documentation provides presenter’s notes for the Microsoft Security Development Lifecycle (SDL) Secure Verification 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](#_Toc233623903)

[1.1 Introduction 2](#_Toc233623904)

[1.2 System Requirements 2](#_Toc233623905)

[1.3 Presentation Themes 2](#_Toc233623906)

[2.0 SDL Secure Verification Principles 2](#_Toc233623907)

[Overview 2](#_Toc233623908)

[Presentation Transcript 2](#_Toc233623909)

[Presentation Voiceover 2](#_Toc233623910)

[Slide 2 – Title Slide 3](#_Toc233623911)

[Slide 3 – Agenda 3](#_Toc233623912)

[Slide 4 – Microsoft Security Development Lifecycle (SDL) 3](#_Toc233623913)

[Slide 5 –SDL Verification Overview 4](#_Toc233623914)

[Slide 6 – Functional Testing vs. Security Testing 4](#_Toc233623915)

[Slide 7 – Functional Testing vs. Security Testing 5](#_Toc233623916)

[Slide 8 – Security Testing Tips 5](#_Toc233623917)

[Slide 9 – SDL Verification Phase Testing Techniques 6](#_Toc233623918)

[Slide 10 – Fuzz Testing 6](#_Toc233623919)

[Slide 11 – How to Fuzz Test 7](#_Toc233623920)

[Slide 12 – Penetration Testing 7](#_Toc233623921)

[Slide 13 – Run-Time Verification 7](#_Toc233623922)

[Slide 14 – Code Review 8](#_Toc233623923)

[Slide 15 – SDL Verification Phase Automation 8](#_Toc233623924)

[Slide 16 – Static Source Code Analysis 8](#_Toc233623925)

[Slide 17 – Binary Analysis Tools 9](#_Toc233623926)

[Slide 18 – Run-Time Analysis Tools 9](#_Toc233623927)

[Slide 19 – Handling Security Vulnerabilities 10](#_Toc233623928)

[Slide 20 – Security Bug Bar Example 10](#_Toc233623929)

[Slide 21 – Conclusion 10](#_Toc233623930)

[Slide 22 - Appendix 11](#_Toc233623931)

[Slide 23 – Microsoft Security Development Lifecycle (SDL) 11](#_Toc233623932)

[Slide 24 – Microsoft Writing Secure Code Book Series 11](#_Toc233623933)

[Slide 25 – Microsoft Developer Network (MSDN) Security Developer Center 11](#_Toc233623934)

[Slide 26 – Secure Development Blogs 11](#_Toc233623935)

[Slide 27 – Hunting Security Bugs 11](#_Toc233623936)

# 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 Verification 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 Verification Principles

## Overview

After software has reached a functional state, test teams begin verifying that desired functionality has indeed been attained. In its continued mission and efforts to deliver more secure and trusted software to its customers, Microsoft, through its SDL, has augmented its own verification (i.e., or testing) processes to require the adoption and execution of industry accepted security and privacy testing techniques, methodologies and tools. This presentation provides an overview of these testing techniques, methodologies and tools.

The insights gleaned by Microsoft, which are incorporated in its SDL, and more specifically, in this presentation focusing on Secure Verification 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 Verification 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 Verification Principles (Level 100) presentation itself for ease of reference.

## Presentation Voiceover

A voiceover of the Secure Verification Principles (Level 100) presentation transcript below, approximately 29 minutes in length is also available to assist the presenter in becoming sufficiently acclimated with the subject matter addressed in the Secure Verification 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 Verification Principles (Level 100) presentation introduces the role that the Microsoft Security Development Lifecycle (SDL) fulfills in trusted application testing and provides an introduction to security testing best practices, techniques and tools of the SDL.

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 testing 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 Verification (i.e., testing) phase, and the security testing techniques and tools used in this phase. These security testing techniques and tools have been pivotal in enabling Microsoft to continue to deliver safer and more privacy aware and 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 verification, or sometimes referred to as testing or quality assurance, phase of the SDL and related tools and techniques.

### Slide 5 –SDL Verification Overview

The remainder of this presentation will focus on key SDL Verification phase topics, such as the differences between functional testing and security, techniques, tools and handling security bugs.

Lastly, the insights gleaned by Microsoft, which are incorporated in its SDL, and more specifically, in this presentation focusing on Secure Verification 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 – Functional Testing vs. Security Testing

An important tenet of the SDL Verification phase is security testing is different and distinct from functional testing.

Functional testers evaluate to what extent an implemented application adheres to its original design; i.e., they compare how closely the two match. Essentially, they ask the question, “Does the application do what it was intended to do?” and they develop and execute test cases to verify whether or not the application does indeed do what it was intended to do. If the functionality is not specified in an application design, functional testers will not test for that particular functionality.

Security testers, however, fulfill a different and often more difficult and complex role. Security testers also evaluate to what extent an implemented application adheres to its original design. However, they focus on the security features of an application and ask the question, “How well were the security features (if any) implemented, and which ones (if any) are missing?” In addition to their emphasis on evaluating the completeness of security features, security testers also evaluate the unintended functionality of an application. Applications operating in the presence of unintended functionality could lead towards misuse. Therefore, security testers evaluate the ability of a malicious user to exploit such extra / unintentional functionality. Often, security testing is referred to as “negative testing” because of this orthogonal goal.

**Why functional testing does not include security testing:** Some have argued that functional testing should incorporate security testing considerations, especially in cases where security is a feature of an application. The operative word there is ‘*should’*. In an ideal world, test plans would include both positive (functional) and negative (security) test cases. In the real world, this has been found to be rarely the case. Given established deadlines, results-oriented application development approaches due to business pressures, and already limited resources, application development teams tend to allocate resources to proving the correctness of an application rather than identifying the incorrectness an application. In this presentation, functionality refers strictly to the productivity aspects of the application being developed.

### Slide 7 – Functional Testing vs. Security Testing

To better conceptualize the goals of functional testing as compared to security testing, let’s focus our attention to the illustration on this slide. The circle on the left corresponds to the intended functionality of an application. This is the functionality envisioned by the designers of an application. The circle on the right, on the other hand, corresponds to the actual implemented functionality of an application. This is the functionality that is implemented through the source code and compiled binary of an application. Let’s now walk through each of these sections and observe the roles fulfilled by both functional and security testers.

(Mouse click)

Traditional functional testing efforts tend to focus on this first section, i.e. placing their emphasis solely on intended functionality without regard to evaluating any extent of actual functionality achieved. This section corresponds to application functionality that has been specified in the application designs, but has not been implemented in the actual application. Here, functional testing is useful because it can answer the question “Does the application do what it was designed (or intended) to do?” If the answer is “no,” then an application bug is noted and addressed by the developers.

Functional testing conducted in this section can also address a subset of security testing, which focuses on the insufficiency of security controls. An example might be insufficient authentication and authorization controls. However, specifying security controls in application design has been found to be rare in practice, so these sorts of issues are rarely identified during the functional testing process.

(Mouse click)

The center section corresponds to application functionality that has been specified in application designs and has also been implemented in the application. Functional testing conducted in this section can also address a subset of security testing which focuses on the quality of security controls. An example might be an evaluation of how well authentication controls were implemented. Again, since specifying security controls in application design has been found to be rare in practice, functional testing performed in this section rarely identifies these types of issues.

(Mouse click)

This final section corresponds to testing application functionality that was not intended by the application designers, but was inadvertently implemented in the actual application code. This section can include security implementation issues, such as buffer overflows and integer arithmetic errors. Implementation issues such as these can have very serious security ramifications. Since this added functionality was never specified by the application designers, these types of issues would not be accounted for in functional tests. Security testing, however, would be able to identify and assess this unintended functionality.

### Slide 8 – Security Testing Tips

In order to be an effective security tester, you need to be able to think like a malicious user who is attacking your application. The first and only rule of security testing is that *there are no rules*. Malicious users are unique in that they are not constrained by any rules, deadlines, best practices, conscience, etc.

Often times you will learn of application development teams who have analyzed a security issue and conclude that “malicious users would not try to do that.” These are famous last words of application security. It is impossible to accurately predict what a malicious user will or will not do. The only valid assumption that can be made about malicious users is that they *will* *attack your application*.

The third important point regarding the use of an application’s client component is difficult for most developers and testers to understand at first. Malicious users do not have to play the game by *your* rules, such as using provided client applications to access server components. Malicious users commonly bypass client components entirely and use raw protocol commands, scripts, and attack tools to communicate directly with the server. In these scenarios, any protection implemented within client components will be bypassed. This is why it is important to be cognizant of scenarios where a malicious user has built their own client to communicate with your server components while executing your security testing plans.

Lastly, security testers must be aware of the capabilities of more sophisticated malicious users. Any external dependencies that your application may require to function correctly (i.e., Web services, firewalls, and other external applications) are viable attack vectors. Your application must be able to continue to operate securely when its dependencies fail or have been compromised. Building your own malicious version of those external dependencies is the best way to test this scenario.

### Slide 9 – SDL Verification Phase Testing Techniques

Now that we have addressed the differences between functional testing and security testing and the mindset of malicious users, it is now time to cover several prevalent security testing techniques employed during the SDL Verification phase.

The Verification phase of the SDL includes, but is not limited to, the following prevalent security testing techniques:

* Fuzz Testing
* Penetration Testing
* Run-Time Verification
* Code Review

It is important to note that each of these security testing techniques have their own pros and cons. Any one security testing technique is insufficient to yield complete security verification. This is the reason why the SDL employs several security testing techniques, instead of relying on a single security testing technique, to better ensure that applications delivered to customers are more secure and trustworthy.

### Slide 10 – Fuzz Testing

A majority of application vulnerabilities exist today due to developers failing to validate inputs. Fuzzing is a testing methodology that can help identify security issues that manifest due to inputs not being properly validated. Fuzzing is a core testing technique in the SDL Verification phase.

There are several approaches to fuzz testing, however, the two most common are called “smart fuzzing” and “dumb fuzzing”. Smart fuzzing examines valid inputs into an application and then varies those inputs to create *invalid* inputs. Each of those invalid inputs is input into the application being tested and the application’s behavior to those inputs is observed by the security tester. For example, if a valid input into an application is a credit card number (16 digits) then smart fuzzing would create invalid variants of that input, such as 16 characters, a mixture of 16 characters and numbers, 16 symbols, and so on. These invalid inputs are input into the application and the application’s reaction to those inputs is observed. Typically, if an invalid input causes an application to raise an exception, crash, or emit some unhandled error, a vulnerability, such as a buffer overflow, a format string attack vector, or a denial of service vector has been identified.

Dumb fuzzing is similar to smart fuzzing except that the invalid input variants using this approach are created randomly rather than in a targeted and deliberate fashion. Going back to our credit card example, dumb fuzzing would use invalid inputs, such as entering no data, 1,000 characters, and a single character. These again are random values that do not consider actual valid inputs or how they are processed by the application before creating invalid input variations.

As with any security testing technique, fuzz testing has its pros and cons. One of the key strengths of fuzz testing is that the types of security code weaknesses that it identifies are usually very severe in nature. Examples include code weaknesses, such as buffer overflows, integer arithmetic errors, and SQL injection, which are all vulnerabilities that could allow a malicious user to assume complete control of an application. One key detriment of fuzz testing is in order to identify security code weaknesses, the application needs to crash or error out due to an invalid input. This means that any vulnerability or application weakness that does result in a crash or error condition typically cannot be found using fuzz testing. Examples of such code weaknesses include cryptographic weaknesses and information disclosures. This again is one reason why the SDL employs several security testing techniques instead of relying on a single one to better ensure that applications delivered to customers are more secure and trustworthy.

### Slide 11 – How to Fuzz Test

The general approach to fuzz testing is to first identify all entry points into an application. The second step is to determine the valid inputs that each application entry point is expecting and create a set of valid inputs. For instance, if a text box in an application is expecting a username, then valid user names might be Steve, Lenny, and Kevin. The third step is to modify the valid inputs created in step 2 into *invalid* inputs. Going back to our example from before, invalid inputs might be Steve, Lenny, and Kevin, but with each letter “e” replaced with the number “3”, i.e., St3v3, L3nny, and K3vin. The final step is to feed these invalid inputs into the application and observe how it responds. If the application crashes, raises an exception, or causes some unexpected error, then a potential security code issue has been identified. If dumb fuzzing is being employed, then steps 2 and 3 may be skipped and random data may be used for step 4.

The amount of invalid inputs used in fuzz testing (smart or dumb) can grow to be very large, sometimes exceeding hundreds of thousands of different invalid inputs. Also, the act of entering those invalid inputs into the application and then observing the application’s reaction to the inputs can be very repetitive. Therefore, it is recommended in the SDL that fuzzing tools be used or developed to conduct fuzz testing. Automation and tools you can use will be discussed later in this presentation.

### Slide 12 – Penetration Testing

The next security testing technique discussed in the SDL is penetration testing. Penetration testing simulates an actual attack against an application and measures the application’s ability to withstand those attacks. Penetration testers will use a series of manual techniques and automated tools to test and exploit potential vulnerabilities in the application.

While penetration testing can be useful to understand the security posture of an application, it is important to also understand that the results of the penetration test are only relevant to the state of the application *at the time of testing*. This means that if an application changes or is re-engineered in anyway, the results from that penetration test will most likely no longer be applicable. Therefore, if a penetration test is to be conducted for an application, then it should be performed when that application is nearing its final state.

At this point you might be asking yourself what is the difference between penetration testing and more tool-centric security testing techniques, such as fuzz testing. The goals of each are very similar, if not overlapping, and both may be used to analyze and improve the security posture of an application. However, it is how each of these techniques arrive at the results is where the difference may be best realized. Automated tools use set checks and hard-coded methodologies to discover vulnerabilities, whereas penetration testing uses the ingenuity of the penetration tester. The human element allows the penetration tester to go beyond the simple discovery of vulnerabilities and yields the ability to connect vulnerabilities together to illustrate more potentially damaging scenarios. This ability to connect different vulnerabilities and application behaviors together to simulate more complex attacks is something that automated tools lack or do not perform as well as humans.

### Slide 13 – Run-Time Verification

Run-time verification testing techniques are used to simulate error conditions while an application is operational. The goal is to observe how an application behaves under these conditions and detect any security code issues that may exist. Tools are often used in run-time verification. One of the key tools used in the SDL is Microsoft’s AppVerifier. AppVerifier will be discussed later in this presentation, but briefly, it is a tool used to simulate run-time error conditions that would be difficult to create under normal testing situations.

### Slide 14 – Code Review

Code review is the last SDL Verification phase security testing technique that will be discussed in this presentation. Code review involves the manual line-by-line review of an application’s source code to identify common code weaknesses that could allow a malicious user attack the application.

Code reviews involve a considerable amount of effort and should ideally be conducted for high-priority code.

Code review is a very effective security testing technique. Having direct access to the source code of an application yields security testers a distinct advantage. With code review, security testers can clearly see what the application does, and they can correctly focus their efforts on areas of code that are highly likely to be attacked. However, for all the benefits of code review, there is one serious negative aspect of the code review process: it can be a very labor-intensive and time consuming process. Applications today can easily exceed hundreds of thousands of lines of code. As such, the likelihood of a code reviewer missing certain vulnerabilities can be high. This likelihood increases even more after long and arduous code review sessions are conducted by code reviewers.

Fortunately, as we will see in the next section, Microsoft has developed several code review tools used in the SDL that are available for download to assist its customers.

### Slide 15 – SDL Verification Phase Automation

Microsoft has developed several tools its internal teams use to identify common application security weaknesses when developing applications with the SDL. To help Microsoft’s customers achieve similar levels of success, several of these tools have been made freely available for download. These tools can be broadly categorized as source code analysis, binary analysis, and run-time analysis tools. The next section of this presentation provides a high-level overview of each tool category.

It is important to note that any one of these tools in their own right is insufficient to ensure that an application is free from vulnerabilities or is “secure.” In fact, no tool from Microsoft or any other company can provide such assurance, and each tool consists of certain strengths and weaknesses. This is again why the SDL does not rely on any one testing technique or tool alone, but rather uses a combination of guidance, tools, and processes to help deliver more secure and more trusted applications to Microsoft’s customers.

### Slide 16 – Static Source Code Analysis

The first category of application security testing tools is static source code analysis tools. These are tools that scan an application’s source code and alert testers to common detectable security vulnerabilities. As was previously indicated, tools should not be used as a replacement for sound security development and testing practices, but rather as an addition to those practices.

**Microsoft PREFast:** PREFast is a static source code analysis tool that detects certain classes of C/C++ coding errors. It analyzes C/C++ source code by stepping through all possible execution paths in each function and simulates execution to evaluate each path for potential problems, such as buffer overruns. PREFast is provided with the Microsoft Windows Driver Kit (WDK), and more information regarding PREFast is located at <http://www.microsoft.com/whdc/DevTools/tools/PREfast.mspx>.

**Microsoft Visual Studio Code Analysis Feature:** Microsoft has integrated the abilities of PREFast into certain versions of Microsoft Visual Studio (Team Edition and higher). Developers and testers can use this feature through the “/analyze” switch to help identify common security coding weaknesses in their applications. More information regarding this feature is located at <http://msdn.microsoft.com/en-us/library/ms173498.aspx>.

*Note:* The PREFast tool is available as a separate download for developers who do not require the use of this tool in an integrated environment.

**Microsoft Cross-Site Scripting (XSS) Detect Tool:** Microsoft has also developed a Visual Studio plug-in that may be used to analyze .NET framework Web applications for a common Web-based application vulnerability called cross-site scripting (XSS). This tool can be downloaded from <http://www.microsoft.com/Downloads/details.aspx?FamilyID=19a9e348-bdb9-45b3-a1b7-44ccdcb7cfbe&displaylang=en>.

**Microsoft Source Code Analyzer for SQL Injection (in ASP code):** Microsoft released a tool to help developers identify certain SQL injection weaknesses in their ASP code. The tool can be downloaded from <http://www.microsoft.com/downloads/details.aspx?FamilyID=58a7c46e-a599-4fcb-9ab4-a4334146b6ba&displaylang=en>.

### Slide 17 – Binary Analysis Tools

The next category of application security testing tools is binary analysis tools. Binary tools, like static source code analysis tools, can evaluate an application for common security implementation weaknesses. However, instead of examining source code to identify application security issues, binary analysis tools examine the compiled binary of an application.

**Microsoft FxCop:** Microsoft FxCop is a binary analysis tool that examines compiled .NET Framework assemblies for common coding weaknesses. In addition to security checks, FxCop also provides checks for domains, such as performance, localization, and others. Microsoft FxCop can be downloaded at <http://code.msdn.microsoft.com/Release/ProjectReleases.aspx?ProjectName=codeanalysis&ReleaseId=553>.

**Microsoft Visual Studio Code Analysis Feature:** Microsoft has integrated the abilities of FxCop into certain versions of Microsoft Visual Studio (Team edition and higher). Developers and testers can use this feature through the “/analyze” switch to help identify common security coding weaknesses in their application implementations. More information regarding this feature is located at <http://msdn.microsoft.com/en-us/library/ms173498.aspx>.

*Note:* The FxCop tool is available as a separate download for developers who do not require the use of this tool in an integrated environment.

### Slide 18 – Run-Time Analysis Tools

The final category of application security testing tools is the run-time analysis tools category. Run-time analysis tools may be used to assist testers in evaluating an application while it is operational.

**Microsoft HTTP Fiddler:** The Microsoft HTTP Fiddler tool is a HTTP proxy which testers can use to analyze request and response data to a Web-based application. More information about this tool can be found at <http://www.fiddlertool.com/fiddler>.

**Microsoft Windows SysInternals Tools:** The Microsoft Windows SysInternals tools are a collection of tools that can be used to examine applications while running on the Windows operating systems. Testers can use these tools to closely examine an application’s running-state behavior, monitor the resources it accesses, and much more. More information regarding the Microsoft Windows SysInternals tools are located at <http://technet.microsoft.com/en-us/sysinternals/default.aspx>.

**Microsoft AppVerifier:** AppVerifier can help quickly identify security issues related to heap overruns in unmanaged code. The Microsoft SDL provides specific details regarding the AppVerifier tests that applications must pass. More information about AppVerifier can be found at <http://msdn.microsoft.com/en-us/library/dd371695(VS.85).aspx>.

**Microsoft Visual Studio Analysis Tools:** Several run-time analysis tools, including AppVerifier, are bundled with Microsoft Visual Studio. More information about Microsoft Visual Studio integrated security tools are located at <http://msdn.microsoft.com/en-us/library/ms364073(VS.80).aspx>.

**Microsoft IE Developer Toolbar:** The Microsoft Internet Explorer team has published a tool bar that is useful for debugging and analyzing Web-based applications. The toolbar can be downloaded at <http://www.microsoft.com/downloads/details.aspx?FamilyId=E59C3964-672D-4511-BB3E-2D5E1DB91038&displaylang=en>.

**Microsoft SDL Book Fuzzing Tools:** The Microsoft SDL book companion CD contains simple fuzzing tools that readers can use to fuzz test applications. More information regarding the SDL book is located at [http://www.microsoft.com/mspress/books/8753.aspx](http://go.microsoft.com/?linkid=9672766).

### Slide 19 – Handling Security Vulnerabilities

In addition to guidance on identifying application security risks, the SDL also provides guidance regarding how to respond to such risks. Specifically, the SDL Verification phase defines how application security issues should be handled through the creation of a “security bug bar”. A security bug bar defines beforehand how application security vulnerabilities will be rated, filed, and fixed. The SDL security bug bar accounts for the following threats, which are created by application security vulnerabilities (in order of most to least severity):

**Elevation of Privilege:** This refers to the ability of a malicious user to execute arbitrary code or to obtain more access privileges than intended.

**Denial of Service:** This refers to the ability of a malicious user to deny legitimate users access to the application.

**Targeted Information Disclosure:** This refers to the ability of a malicious user to locate and read information from anywhere on the system that the application is running on, including system information, that was not intended or designed to be exposed.

**Spoofing:** This refers to the ability of a malicious user to impersonate another user or resource of the application or operating system.

**Tampering:** This is the ability of the malicious user to permanently modify any user data or data used to make trust decisions in a common or default scenario that persists after restarting the operating system or application.

### Slide 20 – Security Bug Bar Example

Here is a sample security bug bar. In the first column the bug bar defines the severity of a security bug and the second column describes the threat type and criteria required to be considered in that severity category. A severity rating of 1 represents those bugs that have the highest potential for inflicting damage, whereas a severity rating of 4 represents those bugs that have the lowest potential for inflicting damage. The Defense in Depth row represents bugs with security ramifications without any known methods for exploit.

In this example, a security bug is considered to be severity 1 if it allows a malicious user to elevate their privilege as a remote user. Another example might be that a security bug is considered to be severity 3 if it allows a malicious user to cause a denial of service condition where the damage is temporary. As indicated before, this bug bar shown is only an example of a bug bar.

When using a security bug bar, application development teams agree upon how a bug in each severity category is to be addressed. For instance, a response to a severity 1 category bug might be that is must be fixed immediately, whereas a severity 3 category bug might not need to be addressed until the next major release of an application is scheduled.

A sample security bug bar can be accessed at <http://msdn.microsoft.com/en-us/library/cc307404.aspx>.

### Slide 21 – Conclusion

This concludes the discussion on the SDL Secure Verification Principles. In this presentation, a high-level overview of the SDL and the important role it fulfills in the verification (i.e., testing) stage of an application’s software development lifecycle was provided. Several security testing tips, techniques, and tools were also addressed. The presentation concluded by providing an overview of how the SDL categorizes application security issues through the use of a security bug bar.

Lastly, the insights gleaned by Microsoft, which are incorporated in its SDL, and more specifically, in this presentation which focused on Secure Verification 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 22 - Appendix

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

### Slide 23 – 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 24 – 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 25 – 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 26 – 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 27 – 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)