MS_logo_KMICROSOFT SDL - DEVELOPER STARTER KIT:

FUZZ TESTING (LEVEL 200)

Guide

Version 1.0

The following documentation provides presenter’s notes for the Microsoft Security Development Lifecycle (SDL) Fuzz Testing (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](#_Toc233607557)

[1.1 Introduction 2](#_Toc233607558)

[1.2 System Requirements 2](#_Toc233607559)

[1.3 Presentation Themes 2](#_Toc233607560)

[2.0 Microsoft SDL Fuzz Testing 2](#_Toc233607561)

[Overview 2](#_Toc233607562)

[Presentation Transcript 2](#_Toc233607563)

[Presentation Voiceover 2](#_Toc233607564)

[Presentation Demonstrations 3](#_Toc233607565)

[Slide 2 – Title Slide 3](#_Toc233607566)

[Slide 3 – Agenda 3](#_Toc233607567)

[Slide 4 – Microsoft Security Development Lifecycle (SDL) 3](#_Toc233607568)

[Slide 5 – Fuzz Testing Overview 4](#_Toc233607569)

[Slide 6 – Fuzz Testing Illustrated 4](#_Toc233607570)

[Slide 7 – Fuzz Testing Pros and Cons 5](#_Toc233607571)

[Slide 8 – Fuzz Testing Techniques 5](#_Toc233607572)

[Slide 9 – Smart versus Dumb Fuzz Testing 6](#_Toc233607573)

[Slide 10 – How to Perform Fuzz Tests 6](#_Toc233607574)

[Slide 11 – Fuzz Testing File Format Parsers 7](#_Toc233607575)

[Slide 12 – File Format Parser Fuzz Testing Demonstration 8](#_Toc233607576)

[Slide 13 – Fuzz Testing Network Parsers 8](#_Toc233607577)

[Slide 14 – Fuzz Testing - Other Types of Parsers 9](#_Toc233607578)

[Slide 15 – Fuzz Testing Tools 9](#_Toc233607579)

[Slide 16 – Microsoft SDL Fuzz Testing Requirements 9](#_Toc233607580)

[Slide 17 – Microsoft SDL Fuzz Testing Recommendations 10](#_Toc233607581)

[Slide 18 – Conclusion 11](#_Toc233607582)

[Slide 19 - Appendix 12](#_Toc233607583)

[Slide 20 – Microsoft Security Development Lifecycle (SDL) 12](#_Toc233607584)

[Slide 21 – Microsoft Writing Secure Code Book Series 12](#_Toc233607585)

[Slide 22 – Microsoft Developer Network (MSDN) Security Developer Center 12](#_Toc233607586)

[Slide 23 – Secure Development Blogs 13](#_Toc233607587)

[Slide 24 – Hunting Security Bugs 13](#_Toc233607588)

[Slide 25 – Additional SDL Training 13](#_Toc233607589)

[Slide 25 – Additional Fuzz Testing Resources 13](#_Toc233607590)

# 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:

* Fuzz Testing (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 Fuzz Testing

## Overview

A majority of all application vulnerabilities that exist today are due to developers failing to validate un-trusted input into an application. Fuzz testing is the process of creating and inputting malformed data into an application and observing the application’s reaction to such malformed data.

This presentation provides an overview of fuzz testing and the related requirements and recommendations for applications developed with the Microsoft SDL.

The insights gleaned by Microsoft, which are incorporated in its SDL, and more specifically, in this presentation focusing on fuzz testing, 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 Fuzz Testing (Level 200) presentation. The precise transcript text provided herein is also incorporated into the notes section of each slide in the Microsoft PowerPoint Fuzz Testing (Level 200) presentation for ease of reference.

## Presentation Voiceover

A voiceover of the Compiler Defenses (Level 300) presentation transcript below, approximately 41 minutes in length, is also available to assist the presenter in becoming sufficiently acclimated with the subject matter addressed in the Fuzz Testing (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: Fuzz Testing](http://go.microsoft.com/?linkid=9672757)

### Slide 2 – Title Slide

The Fuzz Testing (Level 200) presentation introduces the role that the Microsoft Security Development Lifecycle (SDL) fulfills in trusted application development. It also provides an overview of fuzz testing, which is a key verification technique required 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 and cover the topic of fuzz testing. Different approaches to fuzz testing, such as dumb fuzz testing and smart fuzz testing, along with strategies for different application types will also be discussed. Finally, the Microsoft SDL requirements and recommendations for fuzz testing will be presented.

### 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 fuzz testing and how it can be used to uncover vulnerabilities in applications. With respect to specific phases of the Microsoft SDL, this presentation focuses on the Verification phase.

### Slide 5 – Fuzz Testing Overview

A majority of all application vulnerabilities that exist today are due to developers failing to validate un-trusted input into an application. Fuzz testing is the process of creating and inputting malformed data into an application and observing the application’s reaction to such malformed data. If an application fails due to processing the malformed data, then a potential vulnerability has been discovered.

The types of vulnerabilities discovered through fuzz testing are generally very serious in nature. Buffer overflows, integer overflows and denial of service vectors are examples of vulnerabilities that can be identified through fuzz testing. These vulnerabilities can be leveraged by malicious users to assume control of the application or crash it entirely and deny access to legitimate users (i.e., a denial of service attack). Fuzz testing is a key testing technique of the Microsoft SDL that yields several significant benefits; these benefits will be discussed later in this presentation.

Fuzz testing is targeted towards application code that reads and interprets data structures. This type of application code is often referred to as parsers. In this presentation, we will discuss how to fuzz test file format parsers, network protocol parsers and other types of parsers.

The fuzz testing process may require additional resources and commitment from your application development team. As you formulate your fuzz testing strategy and execution plan, you may want to identify any additional resources that may be required in order to execute your fuzz testing strategy and consider how these additional resources may be obtained. Engaging in this thought process as early as possible (e.g., before impending application development efforts commence) will afford you greater time to request and obtain additional budget for the purposes of executing a fuzz testing strategy should additional budget be required.

Lastly, the insights gleaned by Microsoft, which are incorporated in its SDL, and more specifically, in this presentation focusing on fuzz testing, 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 – Fuzz Testing Illustrated

Each application makes certain assumptions about the data it receives from users, such as the format of the data, the type of the data and the length of the data. Fuzz testing is a process for testing those assumptions, and more importantly, it is a process enabling developers to discern how an application reacts when such assumptions are inapplicable / violated.

(Mouse click)

An application may expect input data to contain all numerals, such as in the case of an application reading financial data or telephone numbers. Users who abide by these rules are called valid users.

(Mouse click)

Fuzz testing, however, systematically simulates a user who does not abide by these rules. The specific type of user fuzz testing is employed to simulate is a malicious user.

### Slide 7 – Fuzz Testing Pros and Cons

As with any security testing process, fuzz testing has its advantages and disadvantages.

(Mouse click)

One of the key advantages of fuzz testing is the types of security vulnerabilities that are identified through this process are usually very severe in nature. Examples of security vulnerabilities that can be effectively identified through employing fuzz testing include, but are not limited to, buffer overflows, integer arithmetic errors, denial of service (DoS) and other vulnerabilities that could allow a malicious user to assume complete control over an application or render an application inaccessible / inoperable by authorized users.

(Mouse click)

Another advantage of fuzz testing is it is a highly automatable testing process. The creation and inputting of malformed data can be automated with tools that will be discussed later in this presentation. In some cases, even the analysis of an application crash due to processing malformed data can be automated.

(Mouse click)

A key disadvantage of fuzz testing is in order to identify security vulnerabilities, the application needs to crash, raise an unhandled exception or error out in some unexpected fashion due to processing malformed data. This means that any vulnerability that does not result in an application crash or error condition typically cannot be identified using fuzz testing. Examples of such vulnerabilities that will not be identified through employing fuzz testing include, but are not limited to cryptographic and information disclosure vulnerabilities.

(Mouse click)

This is one of the reasons why the Microsoft SDL employs several testing techniques instead of relying on a single testing technique in order to better ensure that applications developed with the Microsoft SDL are safer and more trustworthy. Lastly, fuzz testing should not replace other assessment activities, and certainly is not to be considered a “silver bullet.”

### Slide 8 – Fuzz Testing Techniques

There are two main methods for fuzz testing applications.

(Mouse click)

The first method is called *dumb fuzzing*. With dumb fuzzing, completely random data is inputted into an application and the reaction of the application to that random data is observed. This type of fuzz testing may not necessarily provide useful results since most applications expect data to be in a certain format. If the malformed data is not presented in the expected data format, then the malformed data is usually rejected by the application. That is, in this scenario, the rejection of the malformed data by the application does not yield any detrimental impact to the operations of the application itself (e.g., the application does not become inoperable due to a crash condition).

With this said, there is still considerable value in employing the dumb fuzz testing technique, and therefore this technique should still be employed. It is perfectly reasonable to assume that a malicious user may input completely random data into an application to compromise it accordingly; as such, dumb fuzz testing is aligned with the primary objective of the fuzz testing process, which is to simulate a malicious user. However, it is critical to understand that dumb fuzz testing should not be the only fuzz testing technique used.

(Mouse click)

The other fuzz testing technique is called *smart fuzzing*. Rather than inputting completely random data into an application, smart fuzz testing consists of inputting specific values leveraging knowledge of underlying format and/or expected behavior within an application. Fuzz testing tools that work in this fashion are able to create malformed data that will more likely be accepted and parsed by an application. The smart fuzz testing technique will greatly improve the chances of identifying security vulnerabilities as compared to dumb fuzz testing.

Please note that one fuzz testing strategy does not replace the other. Both dumb and smart fuzzing can be very effective in certain situations. Application development teams should employ both fuzz testing techniques.

### Slide 9 – Smart versus Dumb Fuzz Testing

As indicated in the previous slide, the difference between dumb and smart fuzz testing is that dumb fuzz testing involves randomly malformed data that is sent to an application. No regard to expected data formats is given with dumb fuzzing. In contrast, smart fuzz testing involves expected data formats that are preserved where only certain values are changed to create malformed data.

(Mouse click)

Suppose we have an application that reads telephone numbers in the format shown here. Let’s see the difference in malformed data that these two fuzz testing techniques produce when testing an application.

(Mouse click)

With dumb fuzz testing, random data is used regardless of the expected data format. As shown here, none of the malformed data produced through dumb fuzz testing conforms to the expected data format. Most applications, when it encounters data that is not formatted in an expected format, will reject the data. Dumb fuzz testing tools may be able to randomly create data in the expected format, but the chance of this is rare.

(Mouse click)

With smart fuzz testing, malformed data is created while preserving the expected data format. Changes to specific values are made to create malformed data. For instance, numerals could be replaced with letters, or even different character sets, such as Unicode characters. As shown here, the malformed data created through smart fuzz testing preserves the expected data format where only the data itself is modified.

### Slide 10 – How to Perform Fuzz Tests

Before we discuss common fuzz testing application attack vectors, such as file format parsers, network parsers and other code that reads data, let’s look at the general methodology behind fuzz testing.

(Mouse click)

In order for a malicious user to attack an application, they need to be able to interact with that application in some fashion. Specifically, malicious users need to be able to provide some type of malformed data to an application. Therefore, the first step for fuzz testing is determine all possible vectors, or entry points, that a malicious user can use to interact with an application. Examples of entry points may include files that are read from file systems and data read from network sockets. These will comprise the set of entry points that will be fuzz tested. Architects, developers and testers who are familiar with an application will be able to build such a list of entry points.

(Mouse click)

The next step is to rank each of the entry points enumerated in the previous step by privilege and accessibility. This enables you to effectively prioritize the entry points that need to be fuzz tested by impact and probability of attack. Threat modeling, which is discussed in a different presentation that is part of this series of field-ready content and demonstrations specifically for developer audiences, is a good way to systematically create this prioritization. It is important that you perform this step to better ensure that the entry points at greatest risk of being compromised will be fuzz tested. It is very important to mention that as a general security best practice, any un-trusted entry point into an application should be fuzz tested. While the topic of prioritization of entry points has been discussed, it is not implied that lower priority entry points should not be fuzz tested.

(Mouse click)

Fuzz testing requires the creation of thousands of different malformed data variations. Each of these malformed data instances are inputted into an application via the highest risk entry points and the application’s reaction to these malformed data inputs is observed. To help automate the creation and inputting of malformed data into applications, it is recommended that fuzz testing tools be used whenever possible. Available fuzz testing tools will be discussed later in this presentation.

(Mouse click)

Any unexpected or unhandled crashes to an application caused by processing malformed data should be rigorously analyzed by developers and testers. Error conditions created from processing malformed data often indicate the presence of severe code vulnerabilities.

(Mouse click)

Finally, vulnerabilities should be fixed and then tests should be executed again to better ensure that the vulnerability is fixed. It is also important to re-run tests since a specific crash may be blocking other issues from showing up until after it gets resolved.

A blog posting from Scott Lambert from the Microsoft Security Engineering Tools team on fuzz testing can be found at <http://blogs.msdn.com/sdl/archive/2007/09/20/fuzz-testing-at-microsoft-and-the-triage-process.aspx>.

### Slide 11 – Fuzz Testing File Format Parsers

File format fuzz testing targets application parsers that read files and the data contained within those files. Files are a common source of input into applications and are frequently targeted by malicious users. Therefore, if an application supports any file types, each of those file types should be fuzz tested.

(Mouse click)

In order to effectively fuzz test files, the first step is to identify all the file formats supported by an application. Application architects, developers and testers are optimal resources to leverage in order to produce a list of all supported file formats.

(Mouse click)

For each supported file format, a library of valid files should be created. These valid files will be used as templates for creating malformed data variations. The Microsoft SDL recommends that at least 100 different valid files be collected for each supported file type.

(Mouse click)

The next step is to create malformed variations of those valid files. Due to the large number of malformed files that need to be created throughout the fuzz testing process, it is recommended that application development teams build tools to automate the creation of the malformed files.

(Mouse click)

The final step involved in fuzz testing file format parsers is to input each of the malformed files into the application being fuzz tested and observe how the application reacts. Again, using or building a tool that automates the inputting of the malformed data into the application is recommended.

A blog posting from Scott Lambert from the Microsoft Security Engineering Tools team on fuzz testing can be found at <http://blogs.msdn.com/sdl/archive/2007/09/20/fuzz-testing-at-microsoft-and-the-triage-process.aspx>.

### Slide 12 – File Format Parser Fuzz Testing Demonstration

Let’s now see a demonstration of the file format fuzz testing process.

(Start file format fuzz testing demonstration)

### Slide 13 – Fuzz Testing Network Parsers

Fuzz testing is a good way for application development teams to systematically identify vulnerabilities in their application implementations. Fuzz testing can be highly automated and typically the vulnerabilities identified through fuzz testing are very serious in nature. If an application reads data from the network, then the network parsers for that application should be fuzz tested. For instance, an application that reads application-layer configuration data sent from a network should have that network parser fuzz tested. Fuzz testing network parsers can be performed in three different ways.

(Mouse click)

The first method of network parser fuzz testing is to create malformed packets that conform to the data format expected by an application, and then send those malformed packets over a network to the application.

(Mouse click)

As shown here, with this technique a fuzz testing tool creates some malformed data and sends that malformed data directly to the application over a network.

(Mouse click)

The second method is to record valid network packets, malform the captured data and then resend (or “replay”) the malformed network packets to an application and observe how the application reacts.

(Mouse click)

With this method, testers first record valid network packets sent to an application using a tool, such as a network packet sniffer.

(Mouse click)

The captured data is then malformed in some way and re-sent, or “replayed” back to the application.

(Mouse click)

The last method of fuzz testing network parsers is to create malformed network packets on the fly.

(Mouse click)

With this method, valid network packets are intercepted by a network device, such as a HTTP proxy, malformed in some fashion and then forwarded to the application.

### Slide 14 – Fuzz Testing - Other Types of Parsers

As I mentioned earlier in this presentation, as a general security best practice, any un-trusted entry point into an application should be considered a candidate for fuzz testing. Each of these entry points should be prioritized in terms of impact from attack and probability of attack. Again, a good way to determine this is through threat modeling which is discussed in another presentation. All entry points with high priorities and risk potential should be fuzz tested first before ones with lower priorities.

Applications may implement custom code that reads, consumes and analyzes data from un-trusted sources. Since reading data from un-trusted sources introduces risk to applications these other types of parsers should be fuzz tested.

Examples of such code include, but are not limited to, ActiveX controls, mobile code and code hosted in a browser. Command-line arguments and user interfaces are other good examples of code that may be at risk and that should be fuzz tested accordingly.

### Slide 15 – Fuzz Testing Tools

Due to the large number of malformed data variations required for fuzz testing, it is important to use tools. Several tools are available to help application teams perform fuzz testing more effectively and efficiently.

The first tool is Microsoft AppVerifier. Microsoft AppVerifier is not a fuzz testing tool, but rather a tool that helps developers analyze code at runtime for unmanaged Microsoft Win32 applications. Microsoft AppVerifier can be used to identify runtime conditions that can lead to vulnerabilities, such as heap-based overflows, memory leaks and access violations. All of these conditions can create security risks for applications. Applications should be monitored by Microsoft AppVerifier prior to fuzz testing. Microsoft AppVerifier is also capable of recording any anomalous conditions observed during the fuzz testing process.

Refer to <http://technet.microsoft.com/en-us/library/bb457063.aspx> for more information regarding Microsoft AppVerifier.

The Microsoft SDL book companion disk contains a simple file fuzzer called MiniFuzz. This program provides an example of how to fuzz test a file parser through dumb and smart fuzzing.

Refer to [http://www.microsoft.com/mspress/books/8753.aspx](http://go.microsoft.com/?linkid=9672766)for more information regarding the Microsoft SDL book.

Finally, several freely available tools from third-party security companies can be downloaded. A search for “fuzzing tools” using a search engine will provide helpful information regarding available tools.

### Slide 16 – Microsoft SDL Fuzz Testing Requirements

Applications developed with the Microsoft SDL must adhere to the following fuzz testing requirements.

(Mouse click)

The first requirement is that fuzz testing must be performed on retail builds of an application; fuzz testing is not conducted on debug builds of applications. Since fuzz testing is a systematic way of simulating a large number of actual attacks against an operable application, it is important that fuzz testing be performed on applications as they would be run by customers. This means that fuzz testing should be tested on the same binaries that will be delivered to customers, i.e., once again, testing retail builds and not debug builds. Retail builds are sometimes referred to as “release” builds.

(Mouse click)

While all entry points into an application can potentially be attacked by malicious users and should be fuzz tested, there are certain types of entry points in Microsoft’s experience that are particularly at-risk. They are files that are read by applications, remote procedure call (RPC) interfaces, and ActiveX controls. Any application that contains these types of entry points must have these entry points fuzz tested. Once again, as previously mentioned, it is security best practice to fuzz test all entry points.

(Mouse click)

In addition to fuzz testing retail builds of an application and at-risk entry points, any time an application is updated or parsers are re-engineered, that application must be re-fuzz tested. The purpose of this requirement is to help ensure that new vulnerabilities have not been introduced into the application implementation.

(Mouse click)

The final Microsoft SDL fuzz testing requirement is that for each file format supported by your application, the parsers for each of those file formats must be fuzz tested with at minimum 100,000 different malformed files. Whenever a vulnerability is found, the count must be reset back to zero, and another pass involving 100,000 different malformed files must be completed.

### Slide 17 – Microsoft SDL Fuzz Testing Recommendations

The following tips can be used to help you perform more effective fuzz testing exercises.

(Mouse click)

The first recommendation is to keep all malformed files that cause an application to fail. These files will be useful in helping developers reproduce any vulnerabilities found through fuzz testing malformed data instances and remediate the vulnerabilities accordingly. Furthermore, these malformed files can be integrated in regression tests to ensure that specific vulnerabilities do not manifest in future versions of the application.

(Mouse click)

The next recommendation is to perform network fuzz testing on private sub-networks. Network-based fuzz testing can often produce large amounts of traffic or can cause network service outages. Production networks and other critical systems should therefore be isolated from network-based fuzz testing exercises.

(Mouse click)

As effective as fuzz testing is, it should always be used in conjunction with other assessment techniques, such as security code review and code analysis. Remember, no one security assessment technique can identify all possible vulnerabilities and therefore should always be used in conjunction with other security assessment techniques.

(Mouse click)

Fuzz testing applications, due to the large number of tests required, can become very time consuming. Iterating through 100,000 or more files, for example, may require significant effort, resources and time. To help reduce the chances of fuzz testing efforts blocking or delaying other verification processes, separate machines should be dedicated for fuzz testing efforts. Additionally, as previously mentioned, the fuzz testing process may require additional resources and commitment from your application development team. As you formulate your fuzz testing strategy and execution plan, you may want to identify any additional resources that may be required in order to execute your fuzz testing strategy and consider how these additional resources may be obtained. Engaging in this thought process as early as possible (e.g., before impending application development efforts commence) will afford you greater time to request and obtain additional budget for the purposes of executing a fuzz testing strategy should additional budget be required.

(Mouse click)

Finally, all application errors generated by fuzz testing efforts should be investigated thoroughly. Errors produced through fuzz testing are often very serious in nature since these errors commonly can enable severe attacks from malicious users. Application development teams should take care to carefully investigate all errors and vulnerabilities identified via performing fuzz testing.

### Slide 18 – Conclusion

This concludes the discussion on fuzz testing. A majority of today’s security vulnerabilities are caused by applications not validating un-trusted inputs. Failing to validate un-trusted inputs can lead to serious vulnerabilities, such as buffer overflows, integer overflows and denial of service. Fuzz testing provides a systematic process for identifying certain types of vulnerabilities in applications during runtime.

Fuzz testing can be performed in two different ways: dumb and smart. Dumb fuzz testing is an iterative process that takes a set of valid inputs into an application and mutates that data by randomly changing values within it. That malformed data is then inputted into an operational application and the application’s reaction to that malformed data is observed. If the application crashes or another unhandled error condition is observed then a potential security vulnerability has been discovered. Developers need to rigorously analyze each crash to determine if actual security vulnerabilities exist.

Oftentimes the malformed data produced by dumb fuzz testing will be automatically rejected by applications since that data will most likely not conform to expected application data formats. This means that a large amount of effort is commonly expended to create malformed data instances that yield minimal potential of identifying security vulnerabilities. This characteristic of dumb fuzz testing is a major criticism of this fuzz testing technique. However, sending random data to an application in the same fashion as is performed via dumb fuzz testing has merit and therefore should not be excluded from fuzz testing plans.

Typically the more effective form of fuzz testing is smart fuzz testing. Smart fuzz testing applies the same approach as dumb fuzz testing except the format of the expected data format is preserved. With smart fuzz testing, only specific values within the data are changed. This improves the chances that the malformed data will be accepted by the application and processed to create an error condition. While smart fuzz testing will generally produce better results, it is important not to discount the usefulness of dumb fuzz testing, or any other security assessment technique for that matter. Fuzz testing should not be used to replace other security assessment techniques, and certainly is not to be considered a “silver bullet.”

One major benefit of fuzz testing is that the types of vulnerabilities that are identified through this security assessment technique are often very severe in nature. Buffer overflows, integer overflows and denial of service vulnerabilities are examples of the types of vulnerabilities that can be identified through fuzz testing. Another benefit is that fuzz testing is highly automatable. Developers will still need to analyze crashes and other error conditions; however, the effort of creating thousands and thousands of different malformed data instances, and then inputting that malformed data into running applications can be automated. One disadvantage of fuzz testing is that it can only uncover security vulnerabilities that cause an application to crash. Information disclosure and cryptographic vulnerabilities are examples of vulnerabilities that are generally not detectable through fuzz testing.

Additionally, applications developed using the Microsoft SDL must adhere to certain fuzz testing requirements. Fuzz testing requirements, such as testing retail builds of applications and performing fuzz testing on certain types of application components, are put in place to help application development teams develop safer and more trustworthy applications through the Microsoft SDL.

The fuzz testing process may require additional resources and commitment from your application development team. As you formulate your fuzz testing strategy and execution plan, you may want to identify any additional resources that may be required in order to execute your fuzz testing strategy and consider how these additional resources may be obtained. Engaging in this thought process as early as possible (e.g., before impending application development efforts commence) will afford you greater time to request and obtain additional budget for the purposes of executing a fuzz testing strategy should additional budget be required.

Lastly, the insights gleaned by Microsoft, which are incorporated in its SDL, and more specifically, in this presentation which focused on fuzz testing, 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 19 - Appendix

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

### Slide 20 – 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 21 – 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 22 – 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 23 – 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 24 – 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://go.microsoft.com/?linkid=9672768>.

### Slide 25 – 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 25 – Additional Fuzz Testing Resources

See the following resources for more information regarding fuzz testing:

* **Fuzz Testing at Microsoft (Scott Lambert):**

<http://blogs.msdn.com/sdl/archive/2007/09/20/fuzz-testing-at-microsoft-and-the-triage-process.aspx>

* **Triaging crashes:**

<http://msdn.microsoft.com/en-us/magazine/cc163311.aspx>

<http://download.microsoft.com/download/7/2/8/728FE40F-93B6-47BD-B67D-78D04B63E27D/Automated%20Security%20Crash%20Dump%20Analysis.pptx>

<http://www.microsoft.com/security/msec/default.mspx>

* **Automated Penetration Testing With White-Box Fuzzing**

<http://msdn.microsoft.com/en-us/library/cc162782.aspx>