MS_logo_KMICROSOFT SDL - DEVELOPER STARTER KIT:

BUFFER OVERFLOWS (LEVEL 300)

Presenter's Guide

Version 1.0

The following documentation provides presenter’s notes for the Microsoft Security Development Lifecycle (SDL) Buffer Overflows (Level 300) 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, 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 4](#_Toc233535712)

[1.1 Introduction 4](#_Toc233535713)

[1.2 System Requirements 4](#_Toc233535714)

[1.3 Presentation Themes 4](#_Toc233535715)

[2.0 Microsoft SDL Buffer Overflows 4](#_Toc233535716)

[Overview 4](#_Toc233535717)

[Presentation Transcript 5](#_Toc233535718)

[Presentation Voiceover 5](#_Toc233535719)

[Presentation Demonstrations 5](#_Toc233535720)

[Slide 2 – Title Slide 5](#_Toc233535721)

[Slide 3 – Agenda 5](#_Toc233535722)

[Slide 4 – Microsoft Security Development Lifecycle (SDL) 5](#_Toc233535723)

[Slide 5 – Buffer Overflows Overview 6](#_Toc233535724)

[Slide 6 – Buffer Overflow Illustration 6](#_Toc233535725)

[Slide 7 – Review of Application Stack Frames 7](#_Toc233535726)

[Slide 8 – Stack-Based Buffer Overflows 8](#_Toc233535727)

[Slide 9 – Review of Application Heaps 9](#_Toc233535728)

[Slide 10 – Heap-Based Buffer Overflows 10](#_Toc233535729)

[Slide 11 – Buffer Overflow Demonstration 11](#_Toc233535730)

[Slide 12 – Common Buffer Overflow Myths 11](#_Toc233535731)

[Slide 13 – Reducing Exposure to Buffer Overflows with The Microsoft SDL 12](#_Toc233535732)

[Slide 14 – Microsoft SDL: Use Attack Surface Reduction and Least Privileges Principles 12](#_Toc233535733)

[Slide 15 – Microsoft SDL: Review Source Code for Buffer Overflows 12](#_Toc233535734)

[Slide 16 – Microsoft SDL: Use Safer APIs and Avoid Banned APIs 12](#_Toc233535735)

[Slide 17 – Microsoft SDL: Use Run-Time Protection 12](#_Toc233535736)

[Slide 18 – Microsoft SDL: Use Code Analysis tools 13](#_Toc233535737)

[Slide 19 – Microsoft SDL: Use Fuzz Testing 13](#_Toc233535738)

[Slide 20 – PLatform Protection From Buffer Overflows 13](#_Toc233535739)

[Slide 21 – Conclusion 13](#_Toc233535740)

[Slide 22 - Appendix 14](#_Toc233535741)

[Slide 23 – Microsoft Security Development Lifecycle (SDL) 14](#_Toc233535742)

[Slide 24 – Microsoft Writing Secure Code Book Series 14](#_Toc233535743)

[Slide 25 – Microsoft Developer Network (MSDN) Security Developer Center 14](#_Toc233535744)

[Slide 26 – Secure Development Blogs 14](#_Toc233535745)

[Slide 27 – Hunting Security Bugs 14](#_Toc233535746)

[Slide 28 – Additional SDL Training 14](#_Toc233535747)

# 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:

* Buffer Overflows (Level 300) presentation.

## 1.2 System Requirements

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

## 1.3 Presentation Themes

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

# 2.0 Microsoft SDL Buffer Overflows

## Overview

For many years, malicious users have been exploiting native code (C and C++) applications through vulnerabilities called buffer overflows. A buffer overflow occurs whenever data is written into a fixed-length buffer and care is not taken to ensure that the buffer has sufficient space to hold that data. Writing data beyond the bounds of a fixed-length buffer can enable malicious users to gain execution control of a process and the instructions that it executes.

This presentation provides an overview of buffer overflows, the internals of buffer overflows and how they are addressed through the guidance, process and tools of the Microsoft SDL.

The insights gleaned by Microsoft, which are incorporated in its SDL, and more specifically, in this presentation focusing on buffer overflows, 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 Buffer Overflows (Level 300) presentation. The precise transcript text provided herein is also incorporated into the notes section of each slide in the Microsoft PowerPoint Buffer Overflows (Level 300) presentation for ease of reference.

## Presentation Voiceover

A voiceover of the Buffer Overflows (Level 300) presentation transcript below, approximately 43 minutes in length is also available to assist the presenter in becoming sufficiently acclimated with the subject matter addressed in the Buffer Overflows (Level 300) 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: Buffer Overflows](http://go.microsoft.com/?linkid=9672753)

### Slide 2 – Title Slide

The Buffer Overflows (Level 300) presentation introduces the role that the Microsoft Security Development Lifecycle (SDL) fulfills in trusted application development. It provides an overview of buffer overflows, the internals of buffer overflows and how they are addressed through the guidance, process and tools of 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 the topic of buffer overflows. We will discuss how they are exploited by malicious users through application stacks and heaps, as well as common myths regarding buffer overflows. The presentation will conclude by providing an overview of the buffer overflow risk reduction measures as prescribed by the Microsoft SDL.

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

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

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

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

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

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

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

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

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

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

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

As was previously indicated, this presentation focuses on buffer overflows and how they can be addressed through the guidance, process and tools of the Microsoft SDL. With respect to specific phases of the Microsoft SDL, this presentation focuses on the Design, Implementation and Verification phases.

### Slide 5 – Buffer Overflows Overview

A buffer overflow is a condition that occurs whenever more data is written into a fixed-length buffer than the buffer can actually hold. This excess data often overwrites and corrupts important control structures which allow malicious users to corrupt other data, crash programs and control process execution. The ability to perform these actions make buffer overflows the most serious and dangerous vulnerabilities known to-date.

Buffer overflows are common in applications developed in native programming languages, such as C and C++, and are less common in managed languages, such as .NET Framework languages and Java. However, you should not assume that your application is immune from buffer overflow attacks simply due to using a managed language. While buffer overflow attacks are not likely to impact applications developed through using managed languages, it is still a possibility and therefore developers should be cognizant of this threat throughout the development process. More about this misconception of buffer overflows will be discussed later in this presentation.

Buffer overflow vulnerabilities are created in code whenever the secure implementation best practice of validating un-trusted input before writing that data into a fixed-length buffer is not followed.

Buffer overflows can occur on both application stacks and heaps, which will be discussed shortly in this presentation.

Lastly, the insights gleaned by Microsoft, which are incorporated in its SDL, and more specifically, in this presentation focusing on buffer overflows, 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 – Buffer Overflow Illustration

So what does a buffer overflow look like? Here is an illustration to help you better visualize the concept of a buffer overflow. Later, we will see how buffer overflows appear at the CPU-level.

Imagine you have a cup that you will use to drink water. The cup represents a fixed-length buffer, and the water that is placed in this cup represents the data that is written into that fixed-length buffer. Let’s now look at different ways that this cup can be filled with water.

(Mouse click)

In this first scenario you fill the cup half way with water. Filling this cup with this much water is safe because we are still within the bounds of the maximum capacity of this cup.

(Mouse click)

Now consider the scenario where you fill the cup of water all the way to the top. Filling the cup to this level is still safe because you have not exceeded the maximum capacity of the cup.

(Mouse click)

The final scenario is where you exceed the capacity of the cup through attempting to contain excessive water. That is, you try to pour more water into this cup than it can actually hold. This results in water being randomly spilled. In the computer world, that spill-over is not so random and can be used to overwrite important parts of memory, which the malicious can then use to control the execution flow of a process.

Let’s now review the internals of how stacks and heaps operate and then see how a malicious user may exploit a buffer overflow vulnerability.

### Slide 7 – Review of Application Stack Frames

The first type of buffer overflow that will be discussed is the stack-based buffer overflow. In order to understand the internals of stack-based buffer overflows, it is important to first understand how an application stack frame functions.

Here is a sample C program that calls a function called *FunctionOne* and then calls a second function called *FunctionTwo*. Let’s see what occurs on an application’s stack when this program executes.

(Mouse click)

The first action is the execution of FunctionOne. When FunctionOne is called, the computer CPU will create a *stack frame* for FunctionOne. Then, the CPU will perform a series of steps on that stack frame in order for FunctionOne to execute correctly.

(Mouse click)

The first step is to reserve space to store any arguments that are passed to FunctionOne.

(Mouse click)

The second step is to establish the address of the next instruction to execute after FunctionOne has completed. This address is simply called the “saved return address”. In this case, the value of this saved return address is the address of FunctionTwo since it is supposed to be executed immediately after FunctionOne. When FunctionOne has completed, this address is loaded into the Extended Instruction Pointer (EIP) register. The CPU will then execute whatever instruction it finds at the address specified by this register.

It should be noted that the saved return address is extremely critical since it controls the execution flow of the program. Malicious users exploiting stack-based buffer overflows will often try to target this particular part of memory. Other regions of memory can be used to influence execution flow; however, this particular one is most commonly targeted.

(Mouse click)

Next, the CPU saves a fixed frame pointer which it uses to reference local function variables and parameters. On Intel processors, this is known as the Extended Base Pointer (EBP).

(Mouse click)

Finally, the CPU reserves space for any local function variables, such as buffers and integers. In this example, space is allocated for the integer variable called *LocalInt* and the 32-byte buffer called *LocalBuffer*.

(Mouse click)

The function FunctionOne executes and then the CPU loads the saved return address, which points to FunctionTwo, and executes the next instruction. The stack frame setup and tear-down process that was just illustrated is then repeated for the FunctionTwo function.

### Slide 8 – Stack-Based Buffer Overflows

Now that we have gained a basic understanding of stack frames, it is now time to discuss the internals of stack-based buffer overflows.

The primary risk from stack-based buffer overflow vulnerabilities is that malicious users may be able to overwrite key memory control structures. If malicious users are able to successfully overwrite key memory control structures, they may be able to control the execution flow of a process. Having this control over a process allows malicious users to control the instructions and commands that a CPU executes. These commands and instructions can be nefarious actions, such as adding unauthorized users and turning off security controls, such as firewalls and antivirus products.

(Mouse click)

Here is an example of a function called *UnsafeFunction* that has a classic stack-based buffer overflow vulnerability using the *strcpy* function. If you inspect the implementation of UnsafeFunction, you will see that it allocates a fixed-length buffer of 32 bytes on the stack called *Buffer*, and then it copies the input parameter *str* into Buffer using a call to strcpy. The assumption that this code incorrectly makes is that the str parameter will never exceed 32 bytes and will never exceed the capacity of the destination buffer. Whenever assumptions like these are made, such assumptions result in the presence of buffer overflow vulnerabilities in the code.

(Mouse click)

Recall from the previous slide that before UnsafeFunction actually executes, the CPU will create a stack frame for UnsafeFunction. Also recall that the return address in the stack frame specifies that next instruction to execute after UnsafeFunction has completed, and that this saved address is frequently the target of buffer overflow attacks.

(Mouse click)

Here are some sample inputs that will be inputted into UnsafeFunction. The first input is simply the string “Kevin”, and the second input will be the letter “A” repeated 40 times. Let’s start with the first input.

(Mouse click)

The first input is the string “Kevin”. This string is only 6 bytes long (5 bytes from the string “Kevin” plus a trailing null terminator byte) and fits within the bounds of Buffer, which was declared as 32 bytes. The function completes, the CPU restores the saved return address value back into the EIP register, and execution continues normally. There is no buffer overflow condition here.

Let’s now see what happens when the second input, 40 “A” characters, is provided to UnsafeFunction.

(Mouse click)

The function strcpy will copy the first 32 “As” into the space allocated for Buffer. What about the last 8 “As”? What happens to these? Strcpy will continue copying the remaining 8 “As” over the saved frame pointer, as well as over the saved return address!

(Mouse click)

The saved frame pointer and the saved return address are now overwritten with the byte values for the letter A (which is 0x41). When UnsafeFunction completes its execution, the CPU will read the value currently in the saved return address (which is 0x41414141) into the EIP register and try to execute the instruction located at that address. The instruction at 0x41414141 will probably cause the process to crash; however, a malicious user could have specified a different address to continue execution.

(Mouse click)

Now let’s consider another scenario. What would happen if the data stored in Buffer contained malicious code or payload, and the address stored in the saved return address pointed to the address of Buffer? The CPU would actually blindly execute whatever instructions were stored in Buffer. This is exactly what malicious users attempt to do when exploiting stack-based buffer overflows!

### Slide 9 – Review of Application Heaps

The next type of buffer overflow I would like to address is the heap-based buffer overflow. As the name suggests, heap-based buffer overflow attacks exploit buffers that have been allocated on an application’s heap rather than on its stack. Heap-based overflows are slightly more complicated to understand and exploit than their stack-based counterparts, so let’s take some time to review specifically how heaps work.

An application’s heap is a dynamic section of memory. Both heaps and stacks may be used to store data; however, they differ in that heaps are allocated at run-time whereas stacks are allocated at compile-time. Heaps are useful in scenarios where the size of the incoming data is not known to the developer at compile-time but is known at run-time.

(Mouse click)

Heaps are broken into chunks. Within each of these chunks is a section that stores information about each chunk, such as the size of the chunk data and pointers to neighboring heap chunks in memory, if any. For our discussion on heap-based overflows, we will only be looking at heap chunk back-pointers (BP) and the forward-pointers (FP). The BP is used to point to the previous heap chunk, while the FP is used to point to the next heap chunk. As you might have guessed from these forward and backward pointers, the heap in essence is just a large doubly-linked list.

(Mouse click)

Let’s now see what happens to the heap when memory is dynamically allocated during normal application execution.

(Mouse click)

Here is a C function called *SampleFunction* that simply allocates space on the heap with a call to the function *malloc*. It then performs some operations and then de-allocates the allocated heap space with a call to the function *free*. Let’s see what happens to the heap when malloc is called by SampleFunction.

(Mouse click)

Chunk #2 is the newly created heap chunk. Chunk #3’s BP is changed to point to Chunk #2, and Chunk #1’s FP is also changed to point to Chunk #2. Chunk #2’s FP and BP are changed to Chunk #3 and Chunk #1 respectively. Note that this entire pointer management process is handled automatically in the background by the operating system heap manager. Also note that the location of any allocated heap chunks will differ depending on the specific platform and heap manager implementation that the platform is using. Chunk #2 did not necessarily have to be placed between Chunk #1 and Chunk #3; however, to help illustrate backward and forward pointers of heap chunks, this was the most convenient placement.

After the initial heap allocation, some operations are performed on the chunk and then the chunk is de-allocated with a call to the function free.

(Mouse click)

When a heap chunk is freed, the FP and BP of adjacent heap chunks (if any) need to be updated. The pseudo-code that is used to free chunks is shown here in the box on the right-hand side. Pay special attention to this pseudo-code, because it is during chunk de-allocation that heap-based overflows are exploited by malicious users.

(Mouse click)

Chunk #2 is finally removed, and the FP and BP of adjacent heaps chunks are updated.

Now that we have a basic understanding of application heaps, let’s examine how a malicious user can exploit heap-based buffer overflow vulnerabilities.

### Slide 10 – Heap-Based Buffer Overflows

The primary risk that manifests from heap-based buffer overflows is when a corrupted heap chunk is de-allocated. When this occurs, a malicious user can gain the ability to write an arbitrary 4 byte DWORD anywhere in an application’s memory space. With this ability, malicious users can modify saved return addresses and pointers elsewhere in memory and take control of the executing process in the same fashion as stack-based buffer overflow attacks.

Due to the dynamic nature of heaps, heap-based buffer overflow exploitation is not as straight forward or reliable as its stack-based counterparts. While exploiting these types of vulnerabilities is much more difficult, it should be noted that is not impossible. In fact, there are several publicly available attack tools that automate the process of exploiting heap-based buffer overflow vulnerabilities. Developers should never assume that because a buffer is allocated on the heap it is not susceptible to heap-based buffer overflow attacks.

Let’s now take a look at how a heap-based buffer overflow might be successfully exploited.

(Mouse click)

Here is the same vulnerable function called *UnsafeFunction* that was used when demonstrating stack-based buffer overflows. In this version, however, the allocated buffer is created on the heap instead of the stack.

(Mouse click)

Here is the current heap layout. As illustrated, there are already two heap chunks (Chunk #1 and Chunk #2) present in the heap. These particular two chunks together are not necessarily required to exploit heap-based overflows, but they are shown here to illustrate the concept more easily.

Let’s go ahead and see what happens when UnsafeFunction is executed. Let’s also assume that the parameter passed to this function consists of several letter “As”, where the number of “As” is much greater than 32 bytes (for example, 200 to 500 consecutive “A” characters.)

(Mouse click)

The first instruction that gets executed is the call to the *malloc* function. This particular function call will allocate a heap chunk onto the heap with 32 bytes of usable data space. In our heap diagram, the heap chunk that was just created corresponds to Chunk #3.

Let’s now see what happens when the call to the *strcpy* function is executed.

(Mouse click)

When the call to the strcpy function is performed, the first 32 “As” are written into the data section of Chunk #3.

(Mouse click)

The rest of the “As” will spill over to the adjacent heap chunk and overwrite the FP and BP fields! Now if Chunk #2 is de-allocated or freed, then a malicious user will be able to write an arbitrary 4 byte DWORD anywhere in memory. Let’s follow the de-allocation process and watch the results of this heap-based buffer overflow.

(Mouse click)

Here on the right-hand side is the pseudo-code for de-allocating a heap chunk. The chunk that is being de-allocated is Chunk #2, so let’s follow the math. The first set of calculations is to determine the value of NextChunk and PreviousChunk. NextChunk is specified as the value of the FP pointer of the chunk being de-allocated (Chunk #2), which at the moment is AAAA. PreviousChunk is specified as the value of the BP pointer of the chunk being de-allocated (Chunk #2), which is also AAAA.

(Mouse click)

The third step is then to set the value of NextChunk plus the offset of the BP field to PreviousChunk. We can now clearly see that a malicious user can write an arbitrary 4 byte DWORD anywhere in memory because they control the values of Chunk #2’s FP and BP fields through the heap-based buffer overflow of Chunk #3.

(Mouse click)

A malicious user could modify this heap-based overflow so that NextChunk pointed to the saved return address in some stack frame and set PreviousChunk to the address of the malicious code they wanted the CPU execute!

As shown, exploitation of heap-based buffer overflows can be difficult but not impossible. Developers need to remain vigilant in following secure implementation best-practices in order to avoid stack-based or heap-based buffer overflows. How developers can do this and reduce their exposure to buffer overflow attacks will be discussed shortly.

### Slide 11 – Buffer Overflow Demonstration

Let’s now see a demonstration of buffer overflow attack.

(Start buffer overflow demonstration)

### Slide 12 – Common Buffer Overflow Myths

Before we look at steps that developers can take to reduce their exposure to buffer overflow attacks through the Microsoft SDL, let’s take some time to dispel some of the more common myths surrounding buffer overflows.

(Mouse click)

**Myth #1: Buffer overflows only affect Microsoft Platforms**

The first myth about buffer overflows is that they only affect Microsoft technologies and platforms. Native languages (C and C++) are supported on many platforms; not just Microsoft Windows. Any application that is written in a native language on any platform is potentially susceptible to having buffer overflow vulnerabilities. The inaccuracy of this myth can be seen if you go to any search engine and type in the name of a platform other than Windows, such as Linux, along with the phrase “buffer overflow”. For instance, if you type “Linux buffer overflows” in any search engine, a list of known buffer overflows on the Linux platform will be returned. All operating systems, applications, and even mobile devices are somehow susceptible to buffer overflow attacks.

(Mouse click)

**Myth #2: Buffer overflows cannot exist in applications developed in a managed language**

The second common myth regarding buffer overflow vulnerabilities is that they cannot exist in managed languages, such as .NET Framework languages and Java. Since these languages provide automated memory management, and for the most part deny direct memory access, buffer overflow vulnerabilities are rarely encountered with these languages.

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 of which to be aware. More problematic are buffer overflows in unsafe managed code, such as the scenario of calling into native code through interoperability services or using .NET code wrapped in UNSAFE blocks. These situations can be very dangerous and require extreme care.

(Mouse click)

**Myth #3: Buffer overflows cannot be exploited on application heaps**

The last common myth that will be discussed is the belief that buffers allocated on heaps cannot be exploited. As we just saw earlier in this presentation, this is just not true. Various modern day operating systems, such as Windows, Linux and others have built-in mechanisms to make exploiting heap-based overflows very difficult; however, none of these mechanisms can guarantee or warrant their respective operating systems to be immune to heap-based overflow attacks.

### Slide 13 – Reducing Exposure to Buffer Overflows with The Microsoft SDL

Now that you have seen the internals of buffer overflows, understand their impact and understand some of the common myths surrounding these vulnerabilities, the question you now might have is, “What can developers do to reduce the number of buffer overflows in code?” Here are some of the guidance, processes and tools developers can use to address the problem of buffer overflows as prescribed by the Microsoft SDL:

* Reduce Attack Surface and Implement Least Privilege
* Search for Risky Functions
* Use Safer Libraries
* Use Compiler Protection
* Use Code Analysis Tools
* Use Fuzz Testing

Please note that presentation content is available for all of these topics.

Please note that none of these risk-reducing measures in their own right are silver bullets. While each of these references, processes and tools may be able to prevent buffer overflow attacks for specific scenarios, none of these in isolation are able to provide comprehensive protection. Developers should therefore try to integrate as many of these measures into their software development lifecycle (SDLC) as possible.

### Slide 14 – Microsoft SDL: Use Attack Surface Reduction and Least Privileges Principles

Two key design principles prescribed by the Microsoft SDL can be used to reduce the risk from buffer overflow attacks. These two design principles are attack surface reduction and least privilege implementation.

Reduce Attack Surface – This technique helps prevent buffer overflow attacks by reducing the possible application vectors that can be exploited by a malicious user.

Least Privilege – This technique takes the approach of conceding that buffer overflows attacks will happen, but isolates the potential damaged caused by those attacks by limiting the privilege level of the overall process being attacked.

For more information see the Secure Design Principles (Level 100) presentation.

### Slide 15 – Microsoft SDL: Review Source Code for Buffer Overflows

Reviewing application code for weaknesses that could facilitate buffer attacks before releasing it to customers requires a financial investment much less than development costs that would be required to resolve buffer overflow issues after applications have been released. The code review process should not focus solely on preventing buffer overflows; it should be employed holistically throughout all code in order to identify other present vulnerabilities.

For more information see the Secure Code Review – Buffer Overflows (Level 200) presentation.

### Slide 16 – Microsoft SDL: Use Safer APIs and Avoid Banned APIs

Using safer libraries helps reduce the likelihood of developers performing coding mistakes that could result in the presence of buffer overflow vulnerabilities. For applications being developed using the Microsoft SDL, Microsoft has banned the use of certain APIs to further reduce the risk of buffer overflow attacks.

For more information see the Banned APIs (Level 200) presentation.

### Slide 17 – Microsoft SDL: Use Run-Time Protection

Current Microsoft compilers provide protection and features, such as limited stack protection and heap checking, that can detect certain buffer overruns during application run-time. These protection features emit errors instead of allowing an exploit to run successfully. The Microsoft SDL requires that certain flags, such as the /GS flag, be enabled for all applications developed with the Microsoft SDL.

For more information see the Compiler Protection (Level 200) presentation.

### Slide 18 – Microsoft SDL: Use Code Analysis tools

Code scanning tools and source code marking techniques, such as Microsoft PREFast and the Standard Annotation Language (SAL) featured in higher versions of Visual Studio, can help detect vulnerabilities through static code analysis.

For more information see the Code Analysis (Level 200) and Source Code Annotation Language (Level 200) presentations.

### Slide 19 – Microsoft SDL: Use Fuzz Testing

Fuzz testing can run in an automated fashion and find large numbers of vulnerabilities with very little engineering effort.

Please see the Secure Verification Principles (Level 100) and Fuzz Testing (Level 300) presentations for more information.

### Slide 20 – PLatform Protection From Buffer Overflows

Modern day operating systems, such as Windows Vista, Windows 2008, and Linux inherently provide built-in protection against buffer overflows. These types of platforms can be leveraged to help reduce the risk from buffer overflow attacks in your applications.

For example, Windows Vista, Windows Server 2008 and higher includes a feature called *Address Space Layout Randomization (ASLR)* which randomizes the address space bases of certain libraries and memory regions. This feature makes it much more difficult for malicious users to successfully exploit buffer overflows.

Processors also have features that protect users from buffer overflows. An example of such a feature is the *Data Execution Protection (DEP)*, sometimes referred to as the *No-Execute (NX)* feature, which is included in most processors.

It is important to mention that none of these types of protection should be regarded as “silver bullets.” These protection features can help reduce the exposure to buffer overflow attacks, but none of them in isolation can completely eliminate such exposure. This reinforces the need for developers to always conform to application development best practices and to always apply the Microsoft SDL.

### Slide 21 – Conclusion

This concludes the discussion on buffer overflows. When developing applications in native languages, such as C and C++, developers need to ensure that any un-trusted data being written into a fixed-length buffer is first validated. Specifically, they need to ensure that the size of data being copied does not exceed the maximum capacity of a fixed-length buffer. If this extra step is not taken, applications may be susceptible to buffer overflow attacks.

Buffer overflows are a major threat for applications developed in native languages on all platforms, and not just the Windows platform. Buffer overflows can be exploited on both application stacks and heaps, although it was long believed that heap-based buffer overflow exploitation was not possible. Although it is rare to find buffer overflow vulnerabilities in managed languages, such as Java and languages used by the .NET Framework, developers coding in these languages still need to remain vigilant in continuing to follow implementation security best-practices, such as input validation. Managed languages are highly resistant to buffer overflow vulnerabilities, however they are not immune.

When a malicious user is able to exploit a buffer overflow vulnerability, they gain tremendous control over the exploited process. Buffer overflows essentially grant malicious users complete control over the instructions that get executed by the CPU and are considered the most serious currently known vulnerability.

Fortunately there are several actions developers can take to reduce their application’s exposure to buffer overflow attacks. The Microsoft SDL has documented guidance, process and tools that developers can use find and fix buffer overflows in their code. Detailed explanations of each of these preventive measures are available as separate presentations.

Lastly, the insights gleaned by Microsoft, which are incorporated in its SDL, and more specifically, in this presentation which focused on buffer overflows, 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).

### Slide 28 – 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.