The SDL Progress Report

Progress reducing software vulnerabilities and developing threat mitigations at Microsoft

2004 - 2010

Microsoft

The SDL Progress Report

Authors

David Ladd – *Microsoft Security Engineering Center*
Frank Simorjay – *Microsoft Trustworthy Computing*
Georgeo Pulikkathara – *Microsoft Trustworthy Computing*
Jeff Jones – *Microsoft Trustworthy Computing*
Matt Miller – *Microsoft Security Engineering Center*
Steve Lipner – *Microsoft Security Engineering Center*
Tim Rains – *Microsoft Trustworthy Computing*

Foreword

This year I reached a fairly major career milestone – the 40th anniversary of the first report I ever wrote on the security of software. As I reflected on that milestone, three things occurred to me: First, I've been in the business a long time! Second, a lot of approaches to building secure software just haven't worked. Third, I believe the industry has now developed some effective approaches to building more secure software and I'm cautiously optimistic about the future.

I have spent most of the last eleven years working in the Trustworthy Computing group at Microsoft, baking security and privacy principles into the culture and software development processes of the company. A key part of Trustworthy Computing is the Security Development Lifecycle (SDL). The SDL is a security assurance process that focuses on software development and introduces security and privacy throughout all phases of the development process. The SDL has been a company-wide mandatory policy since 2004. It combines a holistic and practical approach to reducing the number and severity of vulnerabilities in Microsoft products and services, and thus limits the opportunities for attackers to compromise computers. We freely share the SDL with the software industry and development organizations, and we're delighted to see that it has been adopted (sometimes in adapted form) by a variety of ISVs and IHVs, government agencies, and end users' development organizations.

Even before the SDL was formalized, we created a small team to research security vulnerabilities, their causes, and systematic ways of removing them or mitigating their effects. We have come to refer to this team and its activities as "security science." Security science is the "science inside the SDL". We study how computer systems are attacked and how such attacks can be defeated, and then develop cutting-edge tools and techniques that help make it harder to successfully attack software. When we're convinced that those tools and techniques are reliable and effective, we require their application as part of the SDL. And we release them to the public so that our customers, partners, and even competitors can build more secure software.

In this report you'll learn about the evolution of the SDL and the progress we have made in using the SDL and security science to reduce vulnerabilities and mitigate

threats to Microsoft software and services. We believe that the SDL has helped us to protect Microsoft customers, and because of its broad adoption, we believe it has also helped to protect the wider community of Internet users.

If you are an independent software vendor or other software developer and you're already using the SDL, this report will provide you with some of the background of the SDL and how it has matured over the last six years. If you're not yet using the SDL, we hope this report will help you understand why we believe it's an effective and efficient process and encourage you to try the SDL in your own organization.

Steve Lipner
Senior Director of Security Engineering Strategy
Trustworthy Computing Security, Microsoft

Introduction

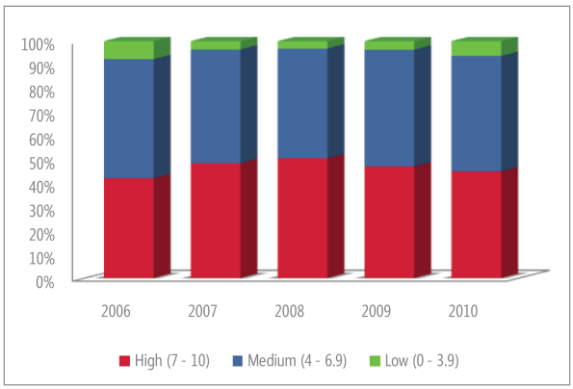
Vulnerabilities are weaknesses in software that enable an attacker to compromise the integrity, availability, or confidentiality of that software or the data that it processes. Some of the most severe vulnerabilities enable attackers to run software code of their choice, potentially compromising the computer, its software, and the data that resides on the computer. The disclosure of a vulnerability can come from various sources, including software vendors, security software vendors, independent security researchers, and those who create malicious software (also known as "malware").

It is impossible to completely prevent vulnerabilities from being introduced during the development of large-scale software projects. As long as human beings write software code, mistakes that lead to imperfections in software will be made – no software is perfect. Some imperfections ("bugs") simply prevent the software from functioning exactly as intended, but other bugs may present vulnerabilities. Not all vulnerabilities are equal; some vulnerabilities won't be exploitable because specific mitigations prevent attackers from using them. Nevertheless, some percentage of the vulnerabilities that exist in a given piece of software will be potentially exploitable.

Data from the National Vulnerability Database[1] shows that the long-term trend for industry-wide vulnerability disclosures is characterized by thousands of vulnerability disclosures each year, most of which are high severity and low complexity. In addition, most vulnerability disclosures are in applications as opposed to operating systems or web browsers. This trend is concerning because it essentially means that there are thousands of high severity vulnerability disclosures in applications, and most of them are relatively easy to exploit.

[1] The information in this section is compiled from vulnerability disclosure data that is published in the National Vulnerability Database (http://nvd.nist.gov), the U.S. government repository of standards–based vulnerability management data represented using the Security Content Automation Protocol (SCAP).

Figure 1: *left*: Industry-wide vulnerability disclosures by severity, 2006 – 2010; *right*: Industry-wide vulnerabilities by access complexity, 2006 – 2010



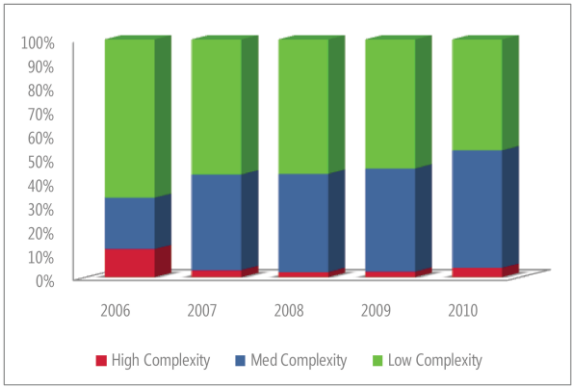
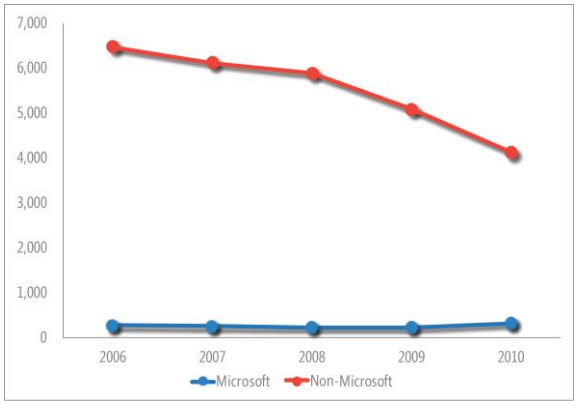
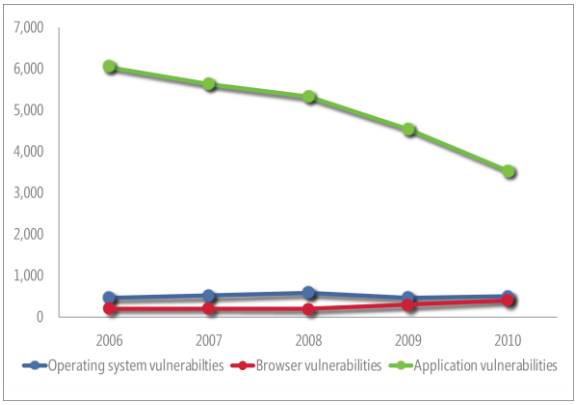


Figure 2: *left*: Vulnerability disclosures for Microsoft and non-Microsoft products, 2006 – 2010; *right*: Industry-wide operating system, browser, and application vulnerabilities, 2006 – 2010





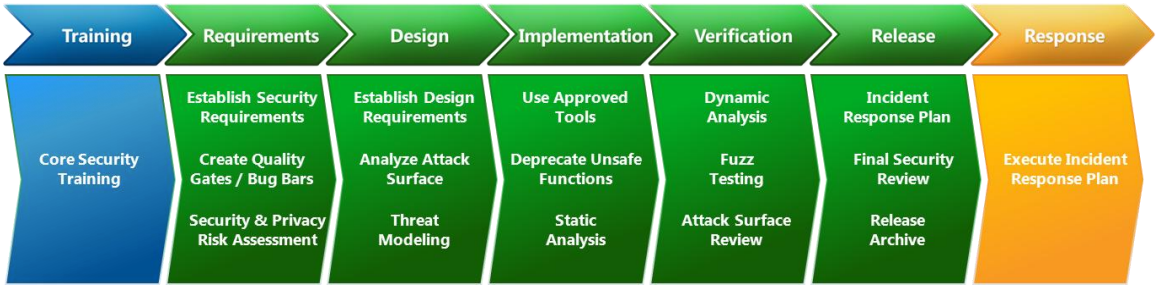
Microsoft believes that sensitive data and personal information must be protected, that software businesses must adhere to business practices that promote trust, and that the technology industry must focus on solid engineering and best practices to provide reliable, secure products and services. Our approach to meeting these challenges is called Trustworthy Computing: a long-term, collaborative effort to create and deliver secure, private, and reliable computing experiences for everyone.

A key part of Trustworthy Computing is the Security Development Lifecycle (SDL). The SDL is a security assurance process that focuses on software development and introduces security and privacy throughout all phases of the development process. As a company-wide mandatory policy since 2004, the SDL has played a critical role in embedding security and privacy in software and

culture at Microsoft. The SDL combines a holistic and practical approach to reduce the number and severity of vulnerabilities in Microsoft

products and services, thus limiting the opportunities for attackers to compromise computers. Microsoft freely shares the SDL with the software industry and customers' development organizations, where it has been used to develop more secure software.

Figure 3: The practices that define the Security Development Lifecycle (SDL) process



Security science is the science inside the SDL, and it builds on an innovative foundation of research to understand how computer systems are attacked and how such attacks can be prevented or mitigated. Security science then incubates and develops cutting-edge tools and techniques that help make it harder to successfully attack software. Security science makes the SDL better three ways:

- Helping to find software vulnerabilities.
- Developing exploit mitigation techniques and tools that developers should adopt.
- Constantly monitoring threat trends and activity in the threat landscape and improving tools and processes based on these observations. If monitoring efforts determine that a new threat has entered the ecosystem, Microsoft security response processes are engaged.

Benefits of Secure Development

Organizations that are beginning software development programs typically focus on functionality first and add on security testing at the end of the development process. This approach, however, has significant drawbacks compared to a structured approach that integrates security efforts throughout the development process.

The National Institute of Standards and Technology (NIST) estimates[2] that code fixes performed after release can result in 30 times the cost of fixes performed during the design phase. Recent studies by Forrester Research, Inc. and Aberdeen Group have found that when companies adopt a structured process like the Microsoft SDL, which systematically addresses software security during the appropriate lifecycle phase, vulnerabilities are more likely to be found and fixed earlier in the development cycle, thereby reducing total cost of software development.

In January of 2011, Forrester published[3] the results of a Microsoft-sponsored survey of top software development influencers across North America. Forrester found that, though application security was not a mature practice for many, those employing a coordinated, prescriptive approach experienced a stronger return on investment.

In August 2010, Aberdeen Group published[4] research confirming that the total annual cost of application security initiatives is far outweighed by the accrued benefits. In December 2010, Aberdeen Group published more detailed research findings from organizations implementing structured programs for security development and "… found that they realized a very strong 4.0-times return on their annual investments in applications security."

With this background information in mind, the rest of this report will focus on the progress Microsoft has made using the SDL and security science to reduce vulnerabilities and develop threat mitigations in Microsoft software and services in ways that help protect Microsoft customers and Internet users. The findings of new research on some of the world's most popular applications are included to provide insight into how many of these applications actually take advantage of the security mitigations that are built into Windows® operating systems.

---

[2] 2002 NIST Report; *The Economic Impacts of Inadequate Infrastructure for Software Testing*
[3] 2011 Forrester Consulting; *State of Application Security: Immature Practices Fuel Inefficiencies, But Positive ROI is Attainable*
[4] 2010 Aberdeen Group; *Securing Your Applications: Three Ways to Play*

Security Development Lifecycle

Microsoft Security Pre-History (1990 - 2004)

Microsoft has a long history of implementing software security – with some efforts, such as Common Criteria compliance, spanning decades. However, prior to the adoption of the SDL at Microsoft, security processes were uneven – product teams implemented security and privacy protections largely at their own discretion. Some development teams subjected their applications to considerable scrutiny, while others prioritized new features and functionality over security and privacy.

Emerging Threats and the Microsoft Response

A series of high profile malware incidents in the mid to late 1990s (Melissa) and early 2000s (Code Red, Nimda, UPnP, etc.) led Microsoft to rethink developer security process and strategy. Early elements of what became the ***threat modeling process*** ***(for example, STRIDE)*** were introduced during this period, along with the concept of a universal ***bug bar*** for establishing vulnerability severity. Formalized ***root cause analysis*** was implemented to understand the cause and impact of various vulnerability types across the spectrum of Microsoft products. Other technology driven changes were introduced – including ***static code analysis*** – when Microsoft acquired Intrinsa and their PREfix static analysis tool. Many of the early security efforts were applied in a limited way to Windows 2000 and eventually became foundational underpinnings of the SDL as practiced today.

STRIDE is the taxonomy used at Microsoft to classify threats found during threat modeling activities.

- ***S**poofing*
- ***T**ampering*
- ***R**epudiation*
- ***I**nformation disclosure*
- ***D**enial of Service*
- ***E**levation of privilege*

Although these changes did have a positive impact on Microsoft software, they weren't mandatory or comprehensive enough to address the challenges facing Microsoft's products. The disruption caused by the release of malware that exploited vulnerabilities in Microsoft software affected many Microsoft customers

and Internet users, and resulted in an erosion of trust. As a result, Bill Gates published his Trustworthy Computing memo in January 2002. The initiation of Trustworthy Computing fundamentally changed the company's priorities regarding software security. This executive mandate positioned security as a top priority for Microsoft and provided the necessary impetus for a sustained campaign of engineering culture change.

.NET and Windows Security Pushes

Executive support at the highest level provided Microsoft security experts with a "sanctioned" opportunity in early 2002 to test the application of a more holistic approach to secure development. Microsoft temporarily halted development of the .NET Framework Common Language Runtime (CLR) to shift development team focus from features and functionality to writing secure code. Over a period of about ten weeks, dozens of security bugs were found and fixed – and the collective emphasis on security introduced new methods of protecting software, with ***attack surface reduction*** being a notable example.

Attack surface analysis and reduction is a security analysis activity performed during the design phase of the Microsoft SDL; its purpose is to reduce the amount of code and data exposed to untrusted users.

The work on the .NET CLR was deemed a success, and as a result executive management agreed that a similar approach should be used on what was then the upcoming release of Windows Server 2003. This effort became known as the "Windows Security Push." Applying these processes to Windows development efforts presented daunting logistical and technology challenges; for example, the size of the Windows code base (at that time) was roughly ten times that of the CLR. Organizational size and complexity of the Windows Division also dictated an increased reliance on technology, yet process innovations continued to surface. ***Tool driven threat modeling*** was in its infancy, and in addition to the existing PREfix solution, ***lightweight static analysis*** (PREfast) was introduced to developer desktops as a means to find and fix bugs prior to code check-in. In addition, the requirement for ***"security audit"*** was added – this audit was the precursor to the ***Final Security Review (FSR)***, which has since become a critical element of the SDL at Microsoft.

The .NET CLR, Windows Server 2003, SQL Server 2000 SP3, and other security pushes yielded dramatic results. But Microsoft security leaders understood that a "push" before product release could not be as effective as the integration of security into product design and development. With confidence borne of the security push experience, a proposal for a mandatory process was presented in early 2004 to Microsoft's Senior Leadership Team: The proposal was approved as

Microsoft policy with the result that all products and online services that are exposed to meaningful security risk or that process sensitive data must conform to SDL requirements.

Microsoft SDL (2004 – Present)

With the SDL, Microsoft has chosen an incremental approach to security process improvement that adds security and privacy process and technology improvements as they mature. As might be expected of a process that matures over time, many security techniques and processes were added early in evolution of the SDL. As time has passed, cumulative focus has shifted to the efficiency and productivity of SDL tools and processes, and improvements have been added as either requirements or recommendations. Some additions to the SDL address new threats while others are improvements to existing requirements.

Many security techniques and tools were in use by Microsoft product groups, prior to their inclusion as Microsoft SDL security requirements or recommendations

The following items that are emphasized in **bold type** are ***not*** an exhaustive year-by-year accounting of all the additions to the SDL; instead they are major additions to the SDL. Detailed discussion of Microsoft SDL requirements (since version 3.2) can be found on the MSDN Developer Center. In addition, these items reflect the points in time when processes or technologies became SDL ***requirements***. However, it is important to note that many techniques were in use by Microsoft product groups, or were included in the SDL as recommendations, long before they became SDL requirements. For brevity and clarity, timelines for SDL recommendations have been omitted.

2004 – SDL 2.0

The first official version of the Microsoft SDL, created in 2004, was essentially a codified list of refinements to techniques and processes that had been used previously during the .NET and Windows Security Pushes discussed above. Some elements were listed as requirements and others were included

The concept of "**Meaningful Security Risk**" is used to determine which applications are subject to the constraints of the Microsoft SDL. In short, any application that exhibits one or more of the following characteristics must conform to the SDL:

- Is widely deployed in an enterprise or organization
- Processes sensitive data; including personally identifiable information (PII)
- "Listens" or interacts on a network

as recommendations. The initial version of the SDL ensured that actions such as threat modeling, static analysis, and the Final Security Review were required for Microsoft software that was exposed to meaningful security risk.

2005 – SDL 2.1 & 2.2

Bug bar. The bug bar is a quality criterion that applies to an entire software development project. It was added to the SDL as a means to set release quality criteria based on bug criticality. It is defined at the start of a project to improve understanding of risks associated with security issues and enables teams to identify and fix security bugs during development. The project team negotiates a bug bar with their assigned security advisor with project-specific clarifications and (as appropriate) more stringent security requirements specified by the security advisor. The bug bar, once defined, is never relaxed.

Fuzzing (file and RPC). Deliberate introduction of random or invalid data to a program (referred to as fuzzing or fuzz testing) was introduced to help find assertion failures, memory leaks and crashes. This was a successful technique used by the attacker ecosystem; due to its effectiveness, it was adopted by Microsoft as a mandatory testing practice. Fuzzing techniques focused initially on file parsers and RPC interfaces.

Cryptographic standards. A series of requirements around the use of cryptography in Microsoft applications was instituted to require product teams to use established cryptographic standards instead of attempting to develop their own non-standard algorithms. Other requirements stipulate the use of cryptographic libraries (such as CryptoAPI), discourage the use of hard-coded crypto solutions (Crypto agility), and specify the use of strong cryptographic algorithms with deliberate notification to users if a fallback to a weaker algorithm was required for compatibility.

Runtime verification testing. In addition to fuzzing, the SDL required additional testing of programs at run time. Using AppVerifier and other tools, teams detected faults in a target application by passively monitoring and reporting on program behavior while it was running in its planned operational environment.

2006 – SDL 3.0 & 3.1

Fuzzing (ActiveX). Fuzz testing was extended to ActiveX® controls that were included in Microsoft applications. ActiveX controls can be used as a conduit for injecting untrusted data from websites and other locations onto a host computer.

A buffer overrun in a scripted ActiveX control can lead to the execution of code from the Internet zone in the context of the logged-in user.

Banned APIs (Banned.h). The C runtime library was first created more than 25 years ago, when network connectivity and the threat environment was much less of a problem. Microsoft decided to deprecate a subset of the C runtime library to remove functions that were known to present security threats, with a focus on buffer overruns. At first this requirement was simply articulated in the form of a list of bad APIs, but it changed over time to become a header file (Banned.h) that could be used in conjunction with a compiler to help provide an automated method of sanitizing source code.

Privacy standards for development. Microsoft established extensive internal guidelines for developers that focus on the protection of customer and user privacy. These guidelines helped developers understand user expectations, global privacy laws, and approved procedures for ensuring privacy across Microsoft products and services.

Online services requirements. Prior to SDL 3.1, there were two distinct sets of security requirements: the SDL for client/server application development (applied to traditional product teams) and a separate set of security requirements applied to MSN and other online services development. The Online Services security requirement was added to unify the elements of the two processes into one coherent SDL.

2007 – SDL 3.2

Cross-site scripting defenses. The SDL mandated procedures to mitigate or prevent the use of cross site scripting attacks (XSS), including input validation, output encoding, and black box vulnerability scanning. This requirement also prescribed the use of the Anti-XSS library for output encoding, which uses the principle of inclusion (also known as White Lists).

SQL injection defenses. Microsoft incorporated specific SDL guidance on SQL injection attacks, including the use of stored procedures and parameterized queries as well as black box security scans.

XML parsing defenses. A requirement was added to address XML parsing attacks. Because a web service will generally attempt to parse any valid XML that is passed to it, fuzzing all interfaces to ensure proper input validation is critical. In addition to parser fuzzing, requirements for specific secure versions of XML parsers were included.

First public release of Microsoft SDL process. In response to queries from customers, users, and other interested parties, Microsoft published the SDL process to increase transparency and allow a greater understanding of the processes and technologies used to secure Microsoft software.

2008 – SDL 4.0 & 4.1

Address space layout randomization (ASLR). Address space layout randomization was moved from a recommendation to a requirement. This requirement specified that ASLR be enabled on all native code (C/C++) binaries to protect against *return-to-libc* attacks. ASLR is described in depth in the "Security Science Mitigations" section of this paper.

CAT.NET for managed code/online services. The use of CAT.NET was specified to ensure that static code analysis for managed code (.NET/C#) is performed prior to code check-in.

Cross-site request forgery (CSRF) defenses. Added a requirement (ViewStateUserKey) to ensure that random, per-session unique tokens are used to prevent CSRF attacks on web applications implemented in .NET languages.

2009 – SDL 5.0

Fuzzing (network). Enhanced fuzzing requirements for all network interfaces and parsers. Increased acceptability thresholds for fuzz tests (100,000 successful iterations).

Operational security reviews. All applications intended to run in Microsoft data centers must pass an additional operational security review prior to deployment as well as meet all SDL security development criteria. While the requirement for operational security reviews had been in place previously, it was integrated into the SDL to ensure that online service teams were presented with a unified framework of security requirements.

Third-party licensing security requirements. Expansion of existing SDL development practices and security servicing criteria to *all* third-party code licensed for use in Microsoft products and services.

External tool releases. First public release of tools from the Microsoft SDL team.

- SDL Threat Modeling Tool enables non-security subject matter experts to create and analyze threat models.

- SDL Template for Visual Studio® Team System (Classic), a template that automatically integrates the policy, process, and tools associated with Microsoft SDL Process Guidance version 4.1 directly into Visual Studio Team System software development environment.

- SDL Binscope Binary Analyzer, a verification tool that analyzes binaries to ensure that they have been built in compliance with the SDL requirements and recommendations.

- SDL Minifuzz File Fuzzer, a basic fuzz testing tool that was designed to help detect issues that may expose security vulnerabilities in file-handling code.

2010 – SDL 5.1

"Sample Code" compliance with SDL. Sample code is used as a template for many development projects; security bugs in sample code often mean security bugs in code developed by third parties that leverage the sample code. All sample code shipped with Microsoft products must meet the same security bar as our products.

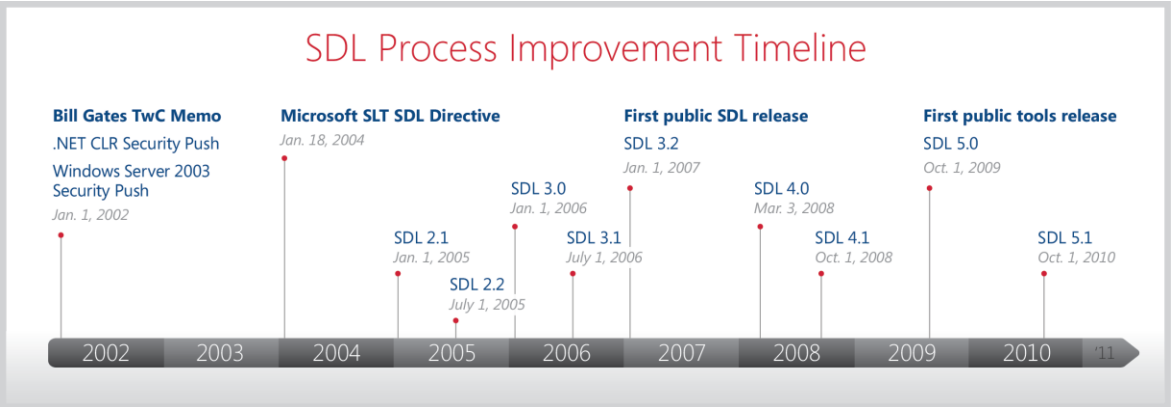
Public release of *Simplified Implementation of the Microsoft SDL*. The Microsoft SDL is based on proven security concepts tailored to meet specific Microsoft business and technical needs. The *Simplified Implementation of the Microsoft SDL* is a non-proprietary, simplified adaptation of the Microsoft SDL suitable for use in other development environments and scenarios and other software platforms.

External tool releases. Second public release of tools from the Microsoft SDL team.

- SDL MSF+Agile Template for Visual Studio Team System, a Team Foundation Server template that automatically incorporates the policy, processes, and tools associated with the SDL for Agile development guidance into the familiar Microsoft Solutions Framework (MSF) for Agile software development (MSF-Agile) process template that ships with Visual Studio Team System.

- Regular Expressions (Regex) Fuzzer, a verification tool to help test regular expressions for potential denial-of-service vulnerabilities.

Figure 4: Timeline of major milestones in the evolution of the SDL at Microsoft



Microsoft has been using the SDL or its precursors for nearly a decade to inject proven security practices into the development of our software. Although it might be tempting to consider the SDL and security development methodologies "completed," the reality is that threats to software are not, and will never be, static. As a result, we will continue to monitor the threat environment, make the investments necessary in people and technology to refine and improve the Microsoft SDL, and aggressively share best practices with third-party software developers to help create a safer computing experience for everyone.

Security Science

Mitigations Help Manage Risk

It's an unfortunate but realistic scenario: a vulnerability in one of your products has been publicly disclosed and a patch is not yet available. How can you help your customers protect themselves? As a software developer, it is critical that the answer to this question be understood well before such a situation ever arises. Thinking about this problem ahead of time makes it possible to build defense-in-depth features into your software that can later enable customers to more effectively manage risk when facing an unpatched or unknown vulnerability. Such defense-in-depth features are generally referred to as *mitigations* because they can partially or completely mitigate the risk associated with a vulnerability.

Oftentimes different approaches can be taken to mitigate a vulnerability. For example, a vulnerability in a network service could be mitigated by preventing connectivity (firewall), preventing access (authentication/authorization), disabling the service or vulnerable feature (configuration), limiting attacker capabilities (containment), and so on. In each case the goal is the same: to make it impossible or very costly for an attacker to successfully exploit a vulnerability. Mitigations that are able to successfully accomplish this goal are therefore able to keep customers protected while a security update is being developed and deployed.

One particularly noteworthy approach to accomplishing this goal is to focus on breaking the *exploitation techniques* that attackers rely on when developing an exploit for a vulnerability. Exploitation techniques can be thought of as the tools that attackers have developed for turning a vulnerability into something that enables the attacker to execute arbitrary code and take control of a user's computer. Breaking or destabilizing these techniques essentially removes a tool from the attacker's toolbox and can (in the best case) make exploitation impossible or increase the time and cost of developing an exploit. Such efforts have a direct impact on an attacker's economic incentive to exploit a vulnerability, and at the same time protect customers while a patch is being developed and deployed. Mitigations that take this approach are generally referred to as *exploit mitigations*.

Exploit mitigations have some additional traits that make them quite attractive as a mitigation strategy. In many cases, the logic that is needed to break an exploitation technique can be incorporated in an application. Using such an approach means customers are not burdened with having to retrofit a mitigation into their existing environment when a new vulnerability is discovered. Another benefit is that because exploit mitigations focus on breaking exploitation techniques, they are generally transparent to typical application functionality. If they are transparent, exploit mitigations can be enabled by default without hindering application functionality. Including exploit mitigations in applications and enabling them by default makes it possible to provide generic protection for vulnerabilities that are known or may currently be unknown.

Given these benefits, it should be no surprise that Microsoft has used security science to develop a wide array of exploit mitigation technologies over the past decade. These technologies are now integrated into development tools such as Visual Studio and are also built into Windows operating systems themselves. This level of integration enables Microsoft and third-party software developers to build applications with mitigations that are baked in and enabled by default.

Tactics

Until now, this paper has only discussed exploit mitigations at a high level. This section explores some of the specific technologies and fundamental tactics that are used to break exploitation techniques, so that you might better understand how exploit mitigations work. The set of tactics used by exploit mitigations can be lumped into a few distinct categories, which include enforcing invariants, adding artificial diversity, and leveraging knowledge deficits. The following subsections will illustrate how these tactics work by highlighting specific examples of exploit mitigation technologies that employ them. It is often common for exploit mitigations to blend one or more of these tactics to maximize their effectiveness.

Enforce invariants

One tactic that can be used to break exploitation techniques is to enforce new invariants that invalidate the implicit assumptions of one or more exploitation techniques. A simple analogy for this can be seen in terms of adding bars to a window – whereas previously an attacker could simply open the window and climb in, they now need to deal with bars that prevent entry through the window. This simple idea has been embodied in the form of multiple mitigation

technologies; two of the most noteworthy are *Data Execution Prevention* (DEP) and *Structured Exception Handler Overwrite Protection* (SEHOP).

Data Execution Prevention (DEP)

One of the assumptions that exploits often make is that data can be executed as code. The origin of this assumption stems from the common practice in which exploits inject custom machine code (often referred to as *shellcode*) and then later execute it – the process known as arbitrary code execution. In most cases, exploits will store such custom machine code in portions of a program's memory, such as the stack or the heap, that are traditionally meant to contain only data. This exploitation technique has historically been quite reliable because older Intel processors and versions of Windows prior to Windows XP Service Pack 2 did not support non-executable memory. The introduction of DEP in Windows XP Service Pack 2 established a new invariant that made it possible to prevent data from being executed as code. When DEP is enabled, it is no longer possible for an exploit to directly inject and execute custom machine code from regions of memory that are intended for data.

Structured Exception Handler Overwrite Protection (SEHOP)

Certain types of vulnerabilities can allow an attacker to make use of an exploitation technique known as a *Structured Exception Handler Overwrite* (SEH overwrite). This technique involves corrupting a data structure that is used when handling exceptional conditions that may occur while a program is running. The act of corrupting this data structure can enable the attacker to execute code from anywhere in memory. This technique is mitigated by SEHOP, which checks to ensure that the integrity of the data structures used for handling exceptions is intact. This new invariant makes it possible to detect the corruption that occurs when an exploit uses the SEH overwrite technique and is ultimately what makes it possible to break exploits that make use of it. SEHOP is a relatively new mitigation technology and is expected to become a requirement in future versions of the SDL.

Add artificial diversity

The existence of diversity within a population helps to minimize the number of universal assumptions that can be made about members of the population. A fitting analogy for this is often given in terms of biodiversity – if a new virus arises, the presence of biodiversity can help to ensure that the entire population is not affected. This principle is also relevant in the digital world, where attackers will often assume that the configuration of one PC will mirror that of another PC.

Introducing artificial diversity into PCs can invalidate these assumptions and thereby prevent an attacker from reliably exploiting a vulnerability. A good example of adding artificial diversity can be seen in the context of an exploit mitigation known as address space layout randomization (ASLR).

Address space layout randomization (ASLR)

Attackers often assume that certain objects (such as DLLs) will be located at the same address in memory every time a program runs (and on every PC that the program runs on). Assumptions such as these are convenient for an attacker because they are often fundamentally required for the exploit to succeed. The inability for an attacker to hard-code these addresses can make it difficult or impossible to write a reliable exploit that will work against every PC. This insight is what drives the motivation for ASLR. ASLR is able to break numerous exploitation techniques by introducing diversity into the address space layout of a program. In other words, ASLR randomizes the location of objects in memory to prevent an attacker from being able to reliably assume their location. This tactic has the effect of making the address space layout of a program different on all PCs and is what ultimately prevents an attacker from making universal assumptions about the location of objects in memory.

Leverage knowledge deficits

In some scenarios, exploitation techniques can be broken by taking advantage of secrets that the attacker does not know or cannot easily predict. A simple analogy for this tactic can be seen in the form of a door with a combination lock. The extensive number of possible combinations prevents the attacker from being able to trivially open the door simply because it is impractical for the attacker to guess the combination in a timely fashion. The same concept applies equally well in the context of breaking exploitation techniques. The use of this tactic in practice is most clearly demonstrated by the Code Generation Security (GS) support that exists in Microsoft's Visual C++ compiler.

GS

The most well-known example of a software vulnerability is the stack-based buffer overflow. This type of vulnerability is traditionally exploited by overwriting critical data that is used to execute code after a function has completed. Since the release of Visual Studio 2002 the Microsoft Visual C++ compiler has included support for the /GS compiler switch which, when enabled, introduces an additional security check that is designed to mitigate this exploitation technique. This tactic works by placing a random value (known as a cookie) prior to the critical data that an

attacker would want to overwrite. This cookie is then checked when the function completes to make sure that it is equal to the expected value. If there is a mismatch, it is assumed that corruption occurred and the program can be safely terminated. This simple concept demonstrates how a secret value (in this case the cookie) can be used to break certain exploitation techniques by detecting corruption at critical points in the program. The fact that the value is secret introduces a knowledge deficit that is generally difficult for the attacker to overcome.

Availability

The diagrams in Figure 5 and Figure 6 summarize the availability of exploit mitigation technology by Windows operating system versions starting with Windows XP RTM[5] and Windows Server® 2003 RTM. Exploit mitigations that are marked as OptIn indicate that the feature is disabled by default and individual applications must explicitly enable the feature (with OptOut being the inverse).

[5] RTM is an acronym for "released to manufacturing" that essentially means no service pack has been installed

Figure 5: Availability of exploit mitigations on Windows client SKUs.

	XP RTM SP1	XP SP2	XP SP3	Vista RTM	Vista SP1	Vista SP2	Win7 RTM
SEH							
SafeSEH	N	Y	Y	Y	Y	Y	Y
SEHOP	N	N	N	N	OptIn	OptIn	OptIn
SEHOP per-process OptIn support	N	N	N	N	N	N	Y
Heap							
Safe unlinking	N	Y	Y	Y	Y	Y	Y
Block header cookies	N	Y	Y	Y	Y	Y	Y
Lookaside/freelist removal	N	N	N	Y	Y	Y	Y
Metadata encryption	N	N	N	Y	Y	Y	Y
Terminate on corruption (32-bit app)	N	N	N	Opt In	Opt In	Opt In	Opt In
Terminate on corruption (64-bit app)	N	N	N	Opt Out	Opt Out	Opt Out	Opt Out
DEP							
NX support (i386)	N	OptIn	OptIn	OptIn	OptIn	OptIn	OptIn
NX support (amd64, 32-bit app)	N	OptIn	OptIn	OptIn	OptIn	OptIn	OptIn
NX support (amd64, 64-bit app)	N	AlwaysOn	AlwaysOn	AlwaysOn	AlwaysOn	AlwaysOn	AlwaysOn
ASLR							
Randomization support							
Images	N	N	N	OptIn	OptIn	OptIn	OptIn
Stacks	N	N	N	OptIn	OptIn	OptIn	OptIn
Heaps	N	N	N	Y	Y	Y	Y
PEBs/TEBs	N	Y	Y	Y	Y	Y	Y
Entropy (bits)							
Images	0	0	0	8	8	8	8
Stacks	0	0	0	14	14	14	14
Heaps	0	0	0	5	5	5	5
PEBs/TEBs	0	4	4	4	4	4	4
APIs							
SetProcessDEPPolicy support	N	N	Y	N	Y	Y	Y

Figure 6: Availability of exploit mitigations on Windows server SKUs.

	Srv03 RTM	Srv03 SP1, SP2	SRV08 RTM	Srv08 R2 RTM
SEH				
SafeSEH	N	Y	Y	Y
SEHOP	N	N	OptOut	OptOut
SEHOP per-process OptIn support	N	N	N	Y
Heap				
Safe unlinking	N	Y	Y	Y
Block header cookies	N	Y	Y	Y
Lookaside/freelist removal	N	N	Y	Y
Metadata encryption	N	N	Y	Y
Terminate on corruption (32-bit app)	N	N	OptIn	OptIn
Terminate on corruption (64-bit app)	N	N	OptOut	OptOut
DEP				
NX support (i386)	N	OptOut	OptOut	OptOut
NX support (amd64, 32-bit app)	N	OptOut	OptOut	OptOut
NX support (amd64, 64-bit app)	N	AlwaysOn	AlwaysOn	AlwaysOn
NX support (ia64)	N/A	AlwaysOn	AlwaysOn	AlwaysOn
ASLR				
Randomization support				
Images	N	N	OptIn	OptIn
Stacks	N	N	OptIn	OptIn
Heaps	N	N	Y	Y
PEBs/TEBs	N	Y	Y	Y
Entropy (bits)				
Images	0	0	8	8
Stacks	0	0	14	14
Heaps	0	0	5	5
PEBs/TEBs	0	4	4	4
APIs				
SetProcessDEPPolicy support	N	N	Y	Y

Mitigation Adoption

The effectiveness of the exploit mitigation technologies previously described is heavily dependent upon adoption by applications that run on Windows. This means that applications must be built with compiler mitigations such as GS enabled and must also properly enable support for platform mitigations such as DEP and ASLR, which currently require applications to explicitly opt-in. Failing to correctly enable one or more of these mitigations makes it easier for attackers to exploit vulnerabilities using tools and techniques that would otherwise not be available to them.

These implications raise a question: if effectiveness depends on adoption, how many applications currently adopt these mitigation technologies? To better understand the answer to this question, we surveyed the DEP and ASLR settings for the latest versions of 41 popular consumer applications that are used by millions of users worldwide. Through this process we found that 71% of the applications surveyed fully enabled support for DEP but only 34% of the applications fully enabled support for ASLR. A detailed analysis of this data is provided in the following sections.

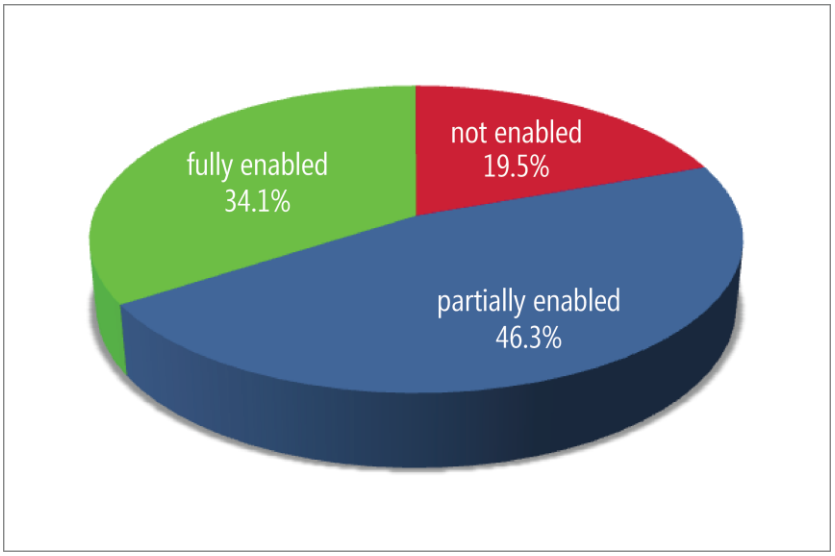
ASLR adoption

Before reviewing the ASLR adoption data it is important to understand how applications actually enable support for ASLR. To enable support for ASLR, an application must link its executable images (EXEs or DLLs) with the /DYNAMICBASE flag. This flag tells the applicable versions of the Windows operating system that an image is ASLR-aware. Images that lack this flag are not randomized by Windows when an application runs and are therefore likely to load at the same address. This lack of randomization significantly reduces the effectiveness of ASLR because it can provide an attacker with a predictable foothold in the program's address space that they can use to develop an exploit. As such, *every* executable image that an application includes must be linked with this flag for ASLR to be most effective. Although the /DYNAMICBASE flag is enabled by default in Visual Studio 2010, previous versions of Visual Studio require developers to explicitly enable this flag in their project settings. These defaults were designed to better ensure compatibility for applications that make assumptions about a program's address space layout.

In practice, of the 41 applications surveyed, 34% fully enabled support for ASLR, 46% partially enabled support, and 20% did not enable support for any of their

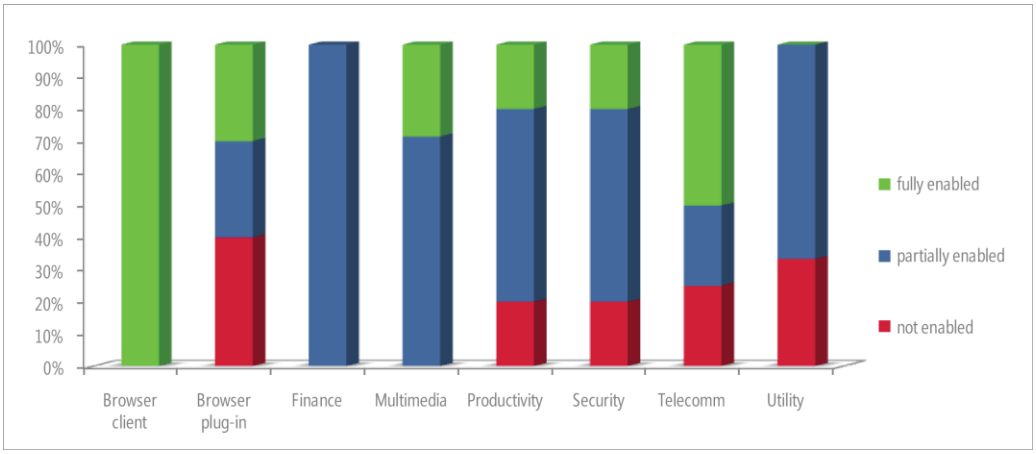
images (Figure 7). This data indicates that many of the popular consumer applications have not fully enabled support for ASLR at this time.

Figure 7: Percentage of applications that fully enable, partially enable, or do not enable ASLR.



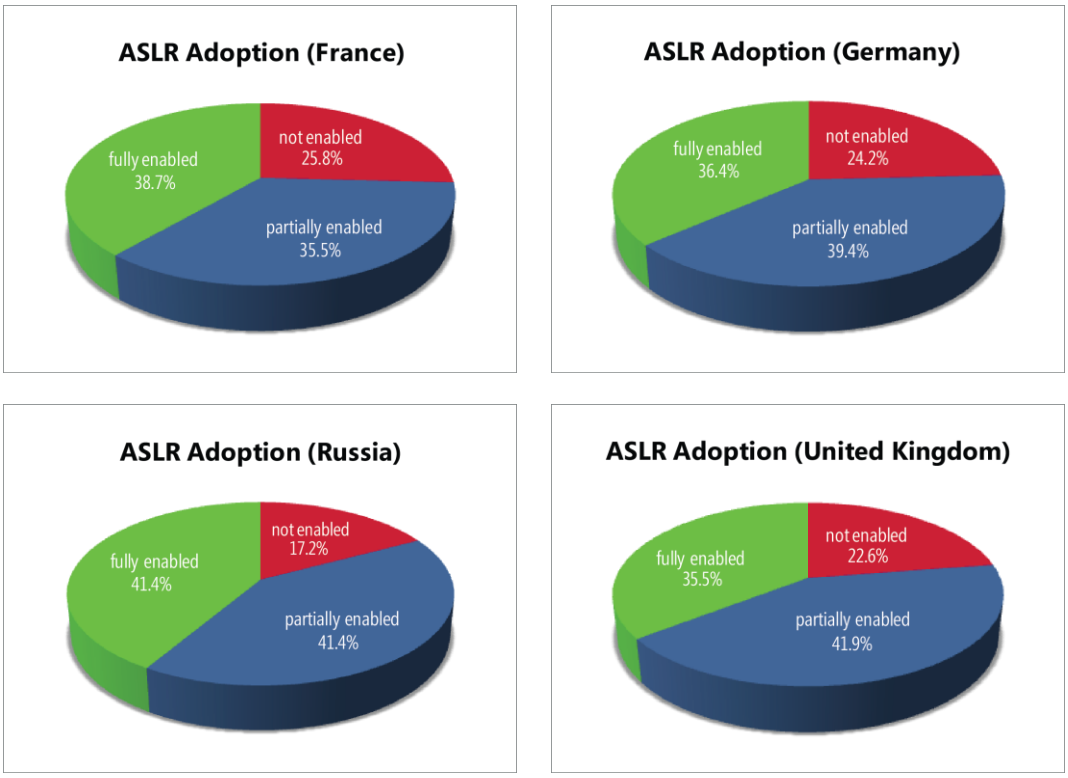
The diagram in Figure 8 provides additional insight about ASLR adoption according to the market segments that each application belongs to. A few noteworthy observations can be made from this data. The first observation is that all of the web browser clients (such as Windows Internet Explorer®) that were surveyed fully enable support for ASLR. Unfortunately, 70% of the surveyed browser plug-ins did not, which means that although ASLR should be effective in default browser installations, the presence of browser plug-ins is likely to weaken ASLR. A second observation is that only one of the five security products included in this analysis fully enabled support for ASLR. This is noteworthy given that security products are inherently exposed to untrusted data and the limited adoption of ASLR might therefore make it easier for attackers to exploit vulnerabilities in security products.

Figure 8: Percentage of applications that fully enable, partially enable, or do not enable ASLR by market segment.



The question of geographic relevance of this data was also explored by considering the subset of surveyed applications that are known to be popular in France, Germany, Russia, and the UK (totaling 31, 33, 29, and 31 applications, respectively). These findings are displayed in Figure 9.

Figure 9: Percentage of surveyed applications that fully enable, partially enable, or do not enable ASLR (France, Germany, Russia, and the UK).



This data clearly shows that ASLR adoption by applications in most market segments has been very slow, despite the technology being available for more than four years. It suggests that many applications have a security posture that is weaker than necessary, which could allow attackers to more easily exploit vulnerabilities. Web browser clients and, to a lesser degree, telecommunication software (such as IM clients) currently appear to be exceptions to this rule. The fact that these applications have adopted ASLR more quickly than others should not be surprising given the amount of direct exposure these types of products have to untrusted data on the Internet.

Call to action:

- Software developers should ensure that they have configured their applications to build with the /DYNAMICBASE linker flag.

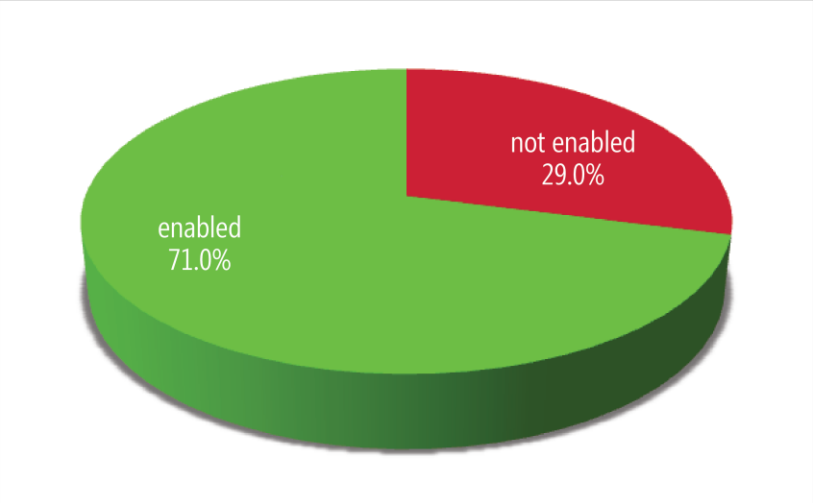
- Software users should demand that software vendors build their applications with the /DYNAMICBASE linker flag.
- Verify the /DYNAMICBASE settings for applications of concern by using the Microsoft SDL BinScope tool.

DEP adoption

Unlike ASLR, DEP is enabled on a per-process basis rather than on a per-image file basis. DEP is automatically enabled for all 64-bit applications but must be explicitly enabled for 32-bit applications that run on client versions of the Windows operating system. The reason 32-bit applications need to explicitly enable support for DEP is to ensure application compatibility with software that was written without taking DEP into consideration. There are multiple ways that an application can choose to enable DEP, including by linking EXEs with the /NXCOMPAT flag and by calling the SetProcessDEPPolicy API at runtime. After DEP is enabled through either of these mechanisms it is permanently enabled. In other words, an application cannot later disable DEP while the application is running. Because DEP is enabled on a per-process basis, there are no special flags that need to be set for every executable image.

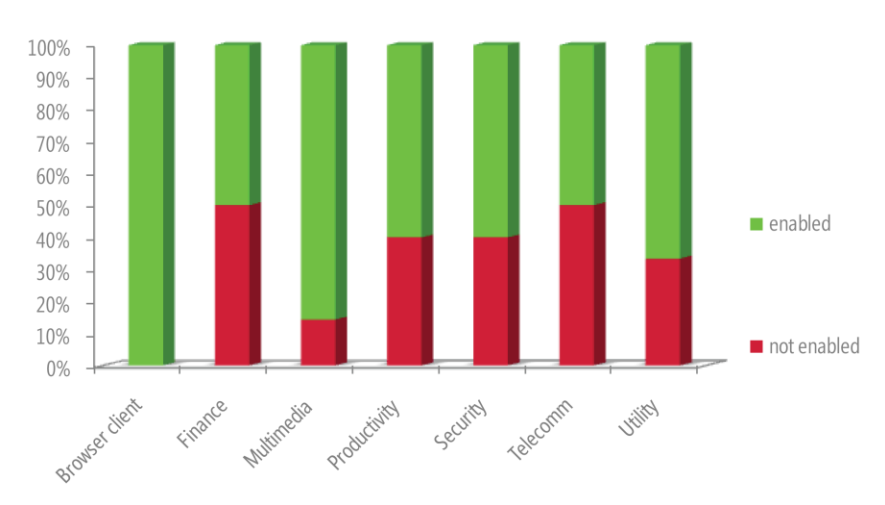
The simplicity of enabling DEP for an application has likely contributed to the rate at which it has been adopted by software vendors. The diagram in Figure 10 shows that of the applications surveyed, 71% enabled DEP and 29% did not. There are two factors that have likely contributed to the majority of surveyed applications having DEP enabled. The first factor is that DEP has been available since Windows XP Service Pack 2, whereas ASLR has only been available since Windows Vista®. Software developers have had more time to adopt DEP as a technology and to overcome any obstacles in doing so. The second factor is that enabling DEP for an application is a relatively simple process in most cases. It should be noted that browser plug-ins are excluded from this data because their DEP settings are dependent on the configuration of the browser that hosts them.

Figure10: Percentage of surveyed applications that enabled DEP.



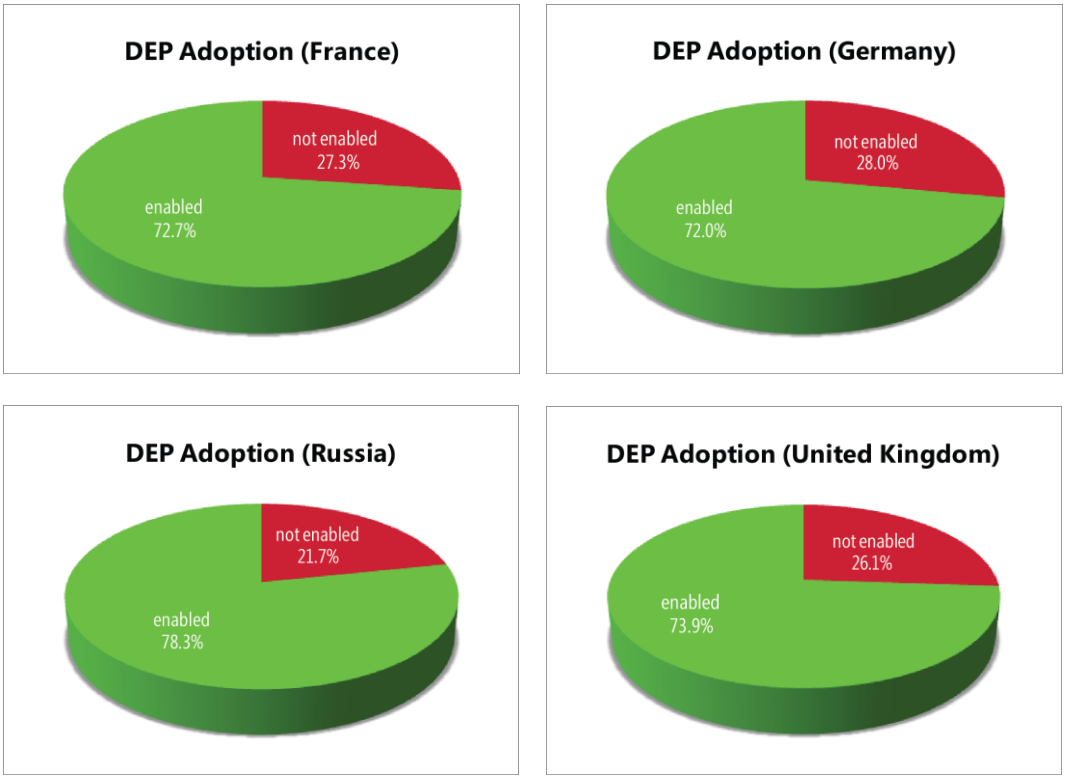
The diagram in Figure 11 provides a breakdown of this data according to the market segment that each application belongs to. As with the ASLR adoption data, the data shows that all surveyed browser clients fully enable support for DEP. Similarly, at least half of the applications in each market segment enable support for DEP.

Figure 11: Percentage of applications that enable or do not enable DEP by market segment.



The question of geographic relevance of this data was explored by considering the subset of applications that are known to be popular in France, Germany, Russia and the UK (totaling 22, 25, 23 and 23 applications, respectively, excluding browser plug-ins). These findings are provided in Figure 12.

Figure 12: Percentage of surveyed applications that enable or do not enable DEP in France, Germany, Russia, and the UK.

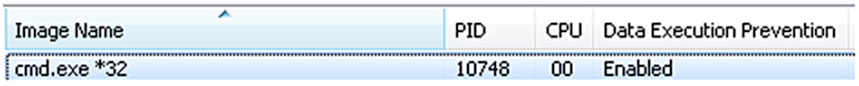


Call to action:

- Software developers should ensure that they have configured their applications to enable DEP either via SetProcessDEPPolicy or via the /NXCOMPAT linker flag.
- Software users should demand that software vendors enable DEP for their applications.

- Verify the DEP settings for applications of concern by using the Enhanced Mitigation Experience Toolkit (EMET), Sysinternals Process Explorer, or the built-in Windows Task Manager as shown below:

Figure 13: Windows Task Manager



Enabling Mitigations in Your Software

This paper shows why exploit mitigations can be a valuable tool in helping to mitigate risks posed by both known and unknown vulnerabilities. Microsoft strongly believes in the value of exploit mitigation technologies and has incorporated mandatory requirements into the SDL that product teams make use of these features when developing software. In practice, the effectiveness of these mitigations is heavily dependent on adoption by applications that run on the Windows operating system. Surveying popular consumer applications has shown that although many applications enabled DEP, the majority did not fully enable ASLR. To improve on this situation, software vendors need to make a concerted effort to enable these and other mitigation technologies in their products. Towards this end, Microsoft provides guidance that is designed to help software vendors enable various exploit mitigation technologies in their products: http://msdn.microsoft.com/en-us/library/bb430720.aspx.

Conclusion

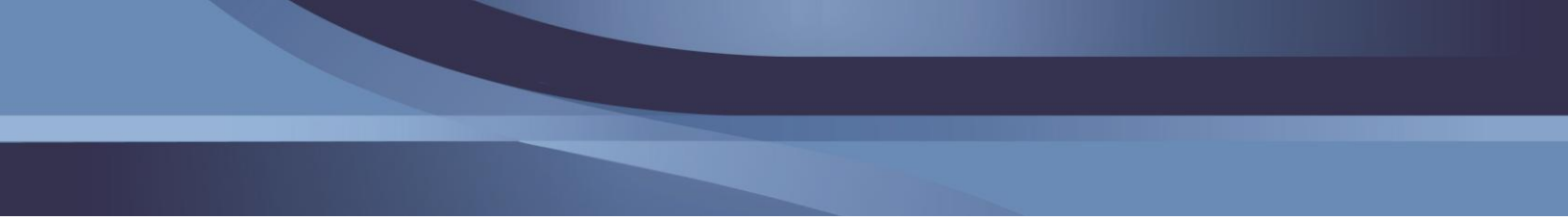
It is impossible to completely prevent vulnerabilities from being introduced during the development of large-scale software projects. The long term trend illustrates that thousands of security vulnerabilities are disclosed across the entire software industry each year; these vulnerabilities are mostly high severity vulnerabilities in applications that are relatively easy to exploit.

Microsoft has made progress by using the SDL and security science to reduce vulnerabilities and introduce threat mitigations in Microsoft software and services in ways that help protect Microsoft customers and Internet users. The research findings in this report show that while many of the world's most popular applications take advantage of the security mitigations that are built into Windows operating systems, there is still opportunity for improvement.

Independent software vendors and software development organizations that are not currently using the SDL should consider applying the SDL in order to realize the benefits it provides.

To learn more about the SDL, how development organizations are using it, and the benefits of secure development, please visit:
http://www.microsoft.com/sdl

To learn more about security science at Microsoft please visit:
http://www.microsoft.com/msec



Microsoft

One Microsoft Way
Redmond, WA 98052-6399
microsoft.com/security