Mitigating Software Vulnerabilities

How exploit mitigation technologies can help reduce or eliminate risk, prevent attacks and minimize operational disruption due to software vulnerabilities

July 2011

Microsoft

Mitigating Software Vulnerabilities

Authors

Matt Miller – *Microsoft Security Engineering Center*

Tim Burrell – *Microsoft Security Engineering Center*

Michael Howard – *Microsoft Security Engineering Center*

Introduction

Software vulnerabilities are weaknesses in computer programs that provide a capable attacker with opportunities to compromise the integrity, availability, or confidentiality of an affected user's computer or data. Severe vulnerabilities can enable attackers to run custom software of their choice, and in some scenarios take full control of a victim's computer. A common method that is used to address software vulnerabilities is to install a security update provided by the vendor of the affected software. Although updating is a robust and well-established vehicle for addressing vulnerabilities, it is not without limitations.

One limitation of security updating is that it presupposes knowledge of what needs to be updated. In other words, a software vendor must know about a vulnerability to create a security update. After the vendor becomes aware of a vulnerability, they must then develop a robust fix and thoroughly test it to ensure that no regressions have been introduced, which can be a time consuming and costly process. Once the update is completed, the vendor must distribute it to their customers as rapidly as possible. The key point to understand is that customers remain at risk until they install the update on their systems.

Accordingly, it has become increasingly important to enable customers to more effectively manage risk when facing an unknown or unaddressed vulnerability. To facilitate this, Microsoft and other software vendors provide customers with guidance on mitigations and workarounds[1] that can be used to reduce or eliminate the risk posed by a software vulnerability. For example, the guidance for mitigating a vulnerability in a network service might include using a firewall to restrict connectivity, using authentication/authorization technologies to prevent access, disabling the service or vulnerable feature, 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 successfully accomplish this goal help protect customers while a security update is being developed and deployed.

One particularly noteworthy method of keeping customers safe while a security update is being developed focuses on breaking the exploitation techniques that attackers rely on when developing an exploit for a vulnerability. Exploitation

[1] All Microsoft customers are encouraged to regularly view Microsoft Security Bulletins, which are issued to address new vulnerabilities: www.microsoft.com/technet/security/current.aspx

techniques can be thought of as the tools attackers have developed for turning a vulnerability into something that enables them to take control of a user's computer. Breaking or destabilizing these techniques essentially removes a valuable tool from the attacker's toolbox and can make exploitation impossible or increase the time and cost of developing an exploit. This approach has a direct impact on an attacker's economic incentive to exploit a vulnerability and can also extend the window of time a customer is protected until an update is deployed.

Mitigations that take this approach are generally referred to as exploit mitigations and they have some unique traits that make them attractive as a mitigation strategy. In many scenarios, the logic that is needed to break an exploitation technique can be built into an application or the Windows operating system itself. If the logic exists in the application, customers will not need to take extra steps to enable the mitigation. Another benefit is that because exploit mitigations focus on breaking exploitation techniques, they are independent of specific vulnerabilities and generally transparent to the application's functionality. By building exploit mitigations into applications and enabling them by default, it becomes possible to provide generic protection for known or currently unknown vulnerabilities.

Microsoft is aware of these benefits, and we have developed and incorporated a wide array of exploit mitigation technologies into many products. These technologies have long been integrated into development tools such as Visual Studio®, and have also been built into the Windows® operating system itself. Furthermore, the latest version of Windows Internet Explorer® fully takes advantage of these technologies. This level of integration enables Microsoft and third-party software vendors to build applications with mitigations that are built in and enabled by default.

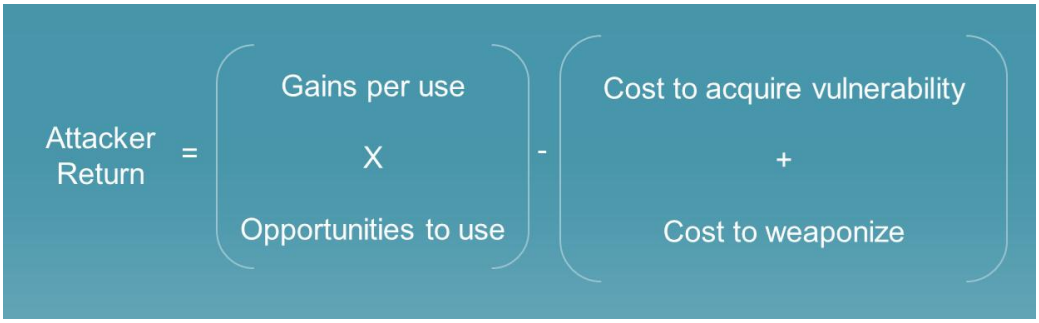
The following sections explore the exploit mitigation technologies provided by Microsoft and also provide a business case for the value of these technologies. The concept of an exploit mitigation is then solidified by introducing the fundamental tactics and technologies that are used to break exploitation techniques. This information forms the basis for providing guidance on how software development teams and IT administrators can use these technologies to protect the applications they develop and deploy.

The economics of exploitation

The primary incentive for an attacker to exploit a vulnerability is to achieve a return on investment. This return need not be strictly monetary—an attacker may be interested in obtaining access to data, identities, or some other commodity that is valuable to them. With these incentives in mind, we can construct a simple formula to help us consider the economics of exploiting vulnerabilities.

The formula in the following figure shows that an attacker must initially invest resources to acquire a vulnerability and develop a weaponized exploit for it. This investment is offset by the gains an attacker receives and the number of opportunities they have to use their exploit.

Figure 1: A formula for modeling the economics of exploitation with respect to an attacker's return on investment.



In recent years, attackers have become very proficient at reducing the cost of acquiring a vulnerability. For example, fuzzing tools and methodologies have become increasingly mature and provide attackers with ways to automate and scale their vulnerability-finding efforts. Software vendors are placed at a noteworthy disadvantage in this respect because they must find and fix all vulnerabilities whereas an attacker only needs to find one. This imbalance is important to recognize—it means that software vendors must invest significantly more than attackers to find vulnerabilities. Even so, it is not possible to guarantee that all vulnerabilities have been eliminated. This issue can be a challenge for vendors whose core business is not software security.

Exploit mitigations can be an important factor in this equation. Put simply, the cost of developing a weaponized exploit for a vulnerability must not exceed an attacker's expected return on investment. Therefore, increasing this cost directly affects an attacker's incentive to develop an exploit. Since exploit mitigations remove generic tools from an attacker's toolbox—an attacker who attempts to exploit a vulnerability must invest significantly more time and resources to develop new exploitation techniques, which may or may not be applicable to other vulnerabilities. For software vendors, the return on investment is also noteworthy because exploit mitigations are relatively cheap to enable and do not require prior knowledge of a particular vulnerability. When combined, these factors suggest that exploit mitigations can be powerful and cost-effective methods for software vendors to use to decrease an attacker's return on investment.

It is important to clarify that the use of exploit mitigations does not excuse a software vendor from finding and fixing vulnerabilities. Vulnerabilities that are difficult or impossible to exploit today have the potential to be exploitable

tomorrow if attackers are able to develop new and improved exploitation methods. In other words, exploit mitigations cannot currently be considered a panacea to the problem posed by vulnerabilities.

Although outside the scope of this document, an attacker's return on investment can also be decreased by reducing the number of opportunities an attacker has to exploit a vulnerability. This reduction can be accomplished by decreasing the amount of time it takes to deploy an update that addresses a vulnerability. However, unlike exploit mitigations, this approach relies on prior knowledge of the vulnerability. In some scenarios, it may also be possible to alter what an attacker can gain from exploiting a vulnerability by employing isolation techniques such as sandboxing to prevent the attacker from gaining access to what they desire. (For more information about sandboxing and other techniques for developing secure software, visit the Microsoft Security Development Lifecycle webpage at www.microsoft.com/sdl.)

Exploit mitigation technology overview

Over the past decade, Microsoft has developed a variety of exploit mitigation technologies that are designed to make it more difficult for attackers to exploit software vulnerabilities such as buffer overruns. This section enumerates each of the mitigation technologies currently available, and provides answers for common questions that relate to how each technology works, how effective they are, and any important performance or compatibility considerations. Availability of each mitigation technology is also provided in terms of which operating system or Visual Studio version supports a given feature.

A more detailed discussion of these technologies and how they work is available in Mitigations Unplugged, a Microsoft presentation given at the Microsoft Blue Hat security conference in 2008: http://technet.microsoft.com/en-us/security/dd285253.

Tactics

Every exploit mitigation technology used in Microsoft products that has been developed to date has employed at least one fundamental tactic that is designed to make it more difficult to exploit a vulnerability. Such tactics provide a helpful high-level illustration of how exploit mitigations are designed to work in practice.

Enforce invariants

One tactic that can be used to break exploitation techniques is to enforce new invariants that invalidate an attacker's implicit assumptions. Oftentimes, attackers

depend on a specific set of conditions being true for an exploit to be successful. A simple analogy for this tactic is to add bars to a window—if an attacker could previously just open the window and climb in, they now need to find a way to get past bars that prevent entry through the window. This simple idea has been embodied in the form of multiple mitigation technologies, and two of the most noteworthy examples are Data Execution Prevention (DEP) and Structured Exception Handler Overwrite Protection (SEHOP).

Create artificial diversity

The existence of diversity within a population helps minimize the number of universal assumptions that can be made about its members. This principle is relevant in the digital world because attackers often assume that the configuration of one computer mirrors that of another computer. Introducing artificial diversity into computer systems can invalidate these assumptions, thereby preventing an attacker from reliably exploiting a vulnerability. A good example of creating artificial diversity can be seen in the context of the exploit mitigation known as Address Space Layout Randomization (ASLR).

Leverage knowledge deficits

In some scenarios, exploitation techniques can be broken by taking advantage of secrets an attacker does not know or cannot easily predict. A simple analogy for this tactic is a door with a combination lock. The extensive number of possible combinations prevents the attacker from being able to easily open the door simply because it is impractical or impossible to guess the combination in a timely fashion. The use of this tactic in practice is most clearly demonstrated by the code Generation Security (/GS) support included in the Visual C++ compiler developed by Microsoft.

Technologies

The tactics described in the previous section are general methods by which exploitation techniques can be prevented. These methods are made concrete through the actual technologies that are designed to mitigate exploitation techniques. This section provides a technical description of each of the exploit mitigation technologies that are currently available, and discusses potential performance or compatibility concerns from their use.

Stack buffer overrun detection

The most well-known example of a software vulnerability is a stack-based buffer overrun. This type of vulnerability is typically exploited by overwriting critical

data used to execute code after a function has completed. Since the release of Microsoft Visual C++ 2002, the Visual C++ compiler has included support for the /GS compiler switch which, when enabled, introduces an additional security check designed to help mitigate this exploitation technique. This mitigation 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 checked when the function completes to ensure it is equal to the expected value. If a mismatch exists, it is assumed that corruption occurred and the program is safely terminated. This simple concept demonstrates how a secret value (in this example, 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.

Visual C++ 2005 improved the effectiveness of /GS by reordering local variables and parameters on the stack, which is the storage area for variables referenced in a function. This approach is designed to prevent an attacker from corrupting local variables or parameters that may be used before the cookie check occurs.

Visual C++ 2010 includes enhancements to /GS that expand the heuristics used to determine when stack overrun protection should be enabled for a function, and when it can safely be optimized away. This approach helps maximize coverage while also minimizing overhead.

To enable this mitigation technology

Stack buffer overrun detection is enabled by compiling with the /GS switch, which has been enabled by default in the Visual Studio IDE since it was first introduced in Visual Studio 2002. The command line compiler enabled this by default in Visual C++ 2005.

Proof point

The vulnerability addressed by Microsoft bulletin MS09-0532 was caused by a stack-based buffer overrun. A functional exploit was developed and released before an update was available that targeted Windows 2000, which was not built with/GS protection. The exploit did not work on Windows XP, Windows Vista®, Windows Server® 2003, and Windows Server 2008, which were all built with /GS protection.

Performance considerations

The performance impact of /GS is difficult to measure because it is highly dependent on coding style. Code with large numbers of stack-based string buffers and arguments might see a small impact; code without them will see no impact.

---

[2] Information on Microsoft security updates is available at: http://www.microsoft.com/technet/security/current.aspx

Compatibility considerations

Well-written software should have no compatibility impact when making use of /GS. The only time /GS will affect how a program performs is when there is an existing stack-based buffer overrun vulnerability. In these scenarios, /GS makes it easier to identify the vulnerability and fix it because the application stops at the point of failure.

Data Execution Prevention (DEP)

One assumption attackers often make is that data can be executed as code. The origin of this assumption stems from the common exploit practice of injecting custom machine code (often referred to as shellcode) and then later executing it—from which the term arbitrary code execution is derived. In most cases, exploits will store this custom machine code in portions of a program's memory, such as the stack or the heap, which are traditionally meant to contain only data. This exploitation technique has historically been quite reliable because older Intel and AMD processors and versions of Windows prior to Windows XP Service Pack 2 did not support making memory non-executable. The introduction of Data Execution Prevention (DEP) in Windows XP Service Pack 2 established a new invariant that made it possible to prevent data from being executed as code by leveraging processors that support the hardware NX bit (No eXecute). When DEP is enabled, it is not possible for an exploit to directly inject and execute custom machine code from regions of memory that are comprised strictly of data. Most modern processors at the time of this writing support the hardware NX bit.

To enable this mitigation technology

DEP is enabled by linking with /NXCOMPAT or by calling SetProcessDEPPolicy at runtime.

Proof point

Although Internet Explorer 8 was vulnerable to MS10-002 and MS10-018, the fact that DEP is enabled by default successfully mitigated the public exploits that were released. In addition, enabling DEP for Microsoft Office broke 100% of the exploits that were tested in a lab environment. DEP is enabled by default in Microsoft Office 2010.

Performance considerations

There are no performance considerations for the use of DEP.

Compatibility considerations

Applications written without taking DEP into consideration may encounter problems when enabling DEP. For example, if a program generates code at runtime but does not properly allocate the memory as executable, the program will crash when the code is executed. Applications that use the Active Template Library (ATL) must use version 8.0 or greater to ensure compatibility with DEP.

Address Space Layout Randomization (ASLR)

Attackers often assume that certain objects (such as executable .DLL and .EXE files) will be located at the same address in memory every time a program runs, and on every computer the program runs on. Assumptions such as these are convenient for an attacker and are often fundamentally required for the exploit to succeed. The inability to hardcode such addresses can make it difficult or impossible to write a reliable exploit that will work against every computer. This insight is what drives the motivation for Address Space Layout Randomization (ASLR), which can 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 reliably assuming their location. This mitigation makes the address space layout of a program different across multiple computers, which ultimately prevents an attacker from developing a successful exploit by assuming the location of objects in memory.

Although Windows XP Service Pack 2 was the first version of Windows to randomize the location of internal data structures such as Process Environment Blocks (PEBs) and Thread Environment Blocks (TEBs), true support for ASLR was not available until Windows Vista. With Windows Vista it became possible to randomize the location of stacks, heaps, and executable files. For compatibility reasons, executable files are required to indicate that they support being randomized by ASLR.

The randomization of stacks is conditional on the process EXE indicating that it supports ASLR. Heap randomization is always enabled and cannot be disabled.

To enable this mitigation technology

ASLR is enabled by linking executable files with /DYNAMICBASE. EXEs must link with /DYNAMICBASE to enable stack randomization.

Proof point

MS08-067 was actively exploited in the wild on Windows 2000, Windows XP, and Windows Server 2003 prior to an update being released. Some exploits were even able to bypass DEP on Windows XP and Windows Server 2003. There are no known exploits that target Windows Vista even though it is also vulnerable, primarily because of the presence of ASLR and DEP.

Performance considerations

ASLR introduces negligible performance overhead the first time an executable file is loaded into memory. Enabling ASLR can result in performance gains by compacting the address space and by reducing the chance that two executable files will try to map to the same address. Executable files that attempt to map to the same address can result in wasteful memory overhead because one must be

relocated to a new address and private copies of certain memory pages may be necessary. There is no performance overhead associated with stack and heap randomization.

Compatibility considerations

In rare cases, an application may assume that the location of stacks, heaps, or executable files will not change each time the application is run. Enabling ASLR for an application that makes these assumptions may cause the application to crash.

Stack randomization has a minor effect on the amount of stack space that can be used by a thread. Applications that use significant amounts of stack space may encounter stack exhaustion issues when stack randomization is enabled.

Deployment considerations

All executable images should be built with /DYNAMICBASE to maximize the effectiveness of ASLR.

SAFESEH and Structured Exception Handler Overwrite Protection (SEHOP)

Certain types of stack-based buffer overrun 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 allows the attacker to execute code from anywhere in memory. Because exceptions can occur before a function returns, it is not possible for /GS to fully mitigate this technique. Instead, two mitigations designed to break this exploitation technique have been developed.

The first mitigation technique is known as SAFESEH (image has safe exception handlers), which first shipped with Visual C++ 2003. This mitigation works by building a table of safe exception handlers when a program is being compiled. The table of safe exception handlers is then consulted at runtime when an exceptional condition occurs to ensure that a matching exception handler is in the table. If a match is not found, the application is terminated. Although this technique has the potential to be effective, it does have some noteworthy limitations such as the requirement that all code must be rebuilt with SAFESEH enabled.

The second mitigation technique is known as Structured Exception Handler Overwrite Protection (SEHOP), and was first shipped with Windows Vista Service Pack 1 and the original release-to-market (RTM) version of Windows Server 2008. SEHOP differs from SAFESEH in that it does not require code to be built with any

special flags. Instead, SEHOP is able to mitigate SEH overwrites by verifying the integrity of the chain of registered exception handlers at the time that an exceptional condition occurs. Typically, an SEH overwrite will break the integrity of this chain, which is what enables SEHOP to mitigate it.

SEHOP support was extended in Windows 7 and Windows Server 2008 R2 by permitting applications to opt-in on a per-application basis, as opposed to enabling or disabling SEHOP for the entire system. By default, SEHOP is disabled in Windows Vista and Windows 7 for compatibility reasons, and is enabled by default on Windows Server 2008 and Windows Server 2008 R2.

SEHOP and SAFESEH are both only relevant to 32-bit x86 applications running on 32-bit or 64-bit versions of Windows. The SEH overwrite exploitation technique is not relevant to native x64 or Itanium-based versions.

To enable SAFESEH

SAFESEH is enabled by linking x86 executable images with /SAFESEH.

To enable SEHOP

SEHOP can be enabled by setting the DisableExceptionChainValidation Image File Execution Option (IFEO) to zero on Windows 7 and Windows Server 2008 R2.

Proof point

There are no known exploits for stack-based vulnerabilities that have been capable of bypassing the combination of /GS, SEHOP, DEP, and ASLR.

Performance considerations

SAFESEH and SEHOP introduce negligible performance overhead into the exception handling path. Because this path is rarely used, there is generally no observable performance overhead.

Compatibility considerations

SAFESEH and SEHOP should both be transparent to applications because they interact with internal and undocumented data structures. However, a small number of applications have been found to be incompatible with SEHOP because they modify these internal data structures in an unsupported way.

Deployment considerations

All executable files should be built with SAFESEH to maximize effectiveness of SAFESEH.

Heap metadata protection

The Windows heap uses metadata to manage allocations made by an application. Sometimes a heap-based buffer overrun may result in this metadata becoming corrupted. Failing to detect this corruption could enable an attacker to direct the

heap to perform an action that is to the attacker's advantage, and could ultimately enable them to execute arbitrary code. To prevent these types of attacks, various checks designed to detect metadata corruption and prevent it from being abused have been added to the Windows heap. In some cases, features more prone to these types of attacks have been removed from the heap, such as lookaside lists and free lists.

Windows XP Service Pack 2 was the first version of Windows to introduce support for a metadata protection technique known as safe unlinking. This protection is designed to detect when a linked list data structure has been corrupted and terminate the use of any corrupt data structures. Windows Vista introduced many additional integrity checks, such as block header encryption and an expanded role for block header cookies, which are designed to protect against other types of heap metadata corruption.

Applications can instruct the Windows heap to terminate when metadata corruption has been detected. The default configuration of this functionality is application-dependent and platform-dependent.

To enable this mitigation technology

Metadata integrity checks are automatically enabled and cannot be disabled. Heap termination on corruption is enabled by default for x64 and Itanium-based applications, and disabled by default for 32-bit x86 applications. To enable this feature in an application, use the HeapSetInformation API.

Visual C++ 2010 enables HeapTerminateOnCorruption by default for the C-runtime—that is, any corruption of a C++ object allocated via the typical 'new' operator will lead to termination.

Proof point

No exploits have been observed in the wild that rely on corrupting heap metadata and target Windows Vista and beyond.

Performance considerations

Because heap metadata protection is enabled by default, there is no additional performance overhead concern. Heap termination on corruption adds no additional overhead because this path is only executed when corruption is detected.

Compatibility considerations

Enabling heap termination on corruption can cause applications with latent heap corruption issues to terminate when previously they may have silently executed without a problem.

Enhanced Mitigation Experience Toolkit (EMET)

In 2009, Microsoft released a stand-alone tool called the Enhanced Mitigation Experience Toolkit (EMET). This toolkit is designed to make it easier to enable and disable exploit mitigation features for a computer as well as for individual applications. EMET centralizes the management of these settings and includes support for additional mitigations that are not currently supported by some versions of Windows. EMET can enable software vendors to test their products with various mitigations in place, and can also enable both large organizations and home users to better protect applications that may not currently take advantage of certain mitigations.

The latest version of EMET can be downloaded here: http://go.microsoft.com/fwlink/?LinkID=200220&clcid=0x409.

Proof point

EMET was able to successfully break exploits that targeted unpatched vulnerabilities in Internet Explorer (MS10-090) and Adobe Reader (CVE-2010-2883).

Availability of Mitigation Technologies

The availability and default settings for the mitigation technologies described previously vary based on operating system and compiler version. The following two figures provide the availability and default settings for mitigation technologies that are supported by the Windows operating system. The third figure provides the availability and default settings for mitigation technologies that are supported by the Visual C++ compiler. The data in these figures show that the latest versions of Windows and the Visual C++ compiler have the most complete support for the exploit mitigation technologies described earlier.

Figure 2 and Figure 3 provide mitigation availability details by Windows operating system version. For software developers, Figure 4 provides mitigation availability details by Visual C++ compiler version.

The following key is provided to help interpret the data presented in the figures.

Variable	Description
n	The feature is not supported.
y	The feature is supported and enabled.
OptIn	The feature is supported but is not enabled by default; applications must explicitly enable the feature.
OptOut	The feature is supported and is enabled by default; applications must explicitly disable the feature if they do not support it.
AlwaysOn	The feature is supported, enabled, and cannot be disabled.

Figure 2: Availability and default settings of platform mitigation features by Windows operating system version (client).

	XP RTM, SP1	XP SP2	XP SP3	Vista RTM	Vista SP1, SP2	Win7 RTM, SP1
SEH						
SafeSEH	n	y	y	y	y	y
SEHOP	n	n	n	n	OptIn	OptIn
SEHOP per-process OptIn support	n	n	n	n	n	y
Heap						
Safe unlinking	n	y	y	y	y	y
block header cookies	n	y	y	y	y	y
lookaside/ freelist removal	n	n	n	y	y	y

	XP RTM, SP1	XP SP2	XP SP3	Vista RTM	Vista SP1, SP2	Win7 RTM, SP1
metadata encryption	n	n	n	y	y	y
terminate on corruption (32-bit app)	n	n	n	OptIn	OptIn	OptIn
terminate on corruption (64-bit app)	n	n	n	OptOut	OptOut	OptOut
DEP						
NX support (i386)	n	OptIn	OptIn	OptIn	OptIn	OptIn
NX support (amd64, 32-bit app)	n	OptIn	OptIn	OptIn	OptIn	OptIn
NX support (amd64, 64-bit app)	n	Always On	Always On	Always On	Always On	Always On
ASLR						
randomization support						
images	n	n	n	OptIn	OptIn	OptIn
stacks	n	n	n	OptIn	OptIn	OptIn
heaps	n	n	n	y	y	y
PEBs/TEBs	n	y	y	y	y	y
entropy (bits)						
images	0	0	0	8	8	8
stacks	0	0	0	14	14	14
heaps	0	0	0	5	5	5
PEBs/TEBs	0	4	4	4	4	4
APIs						

	XP RTM, SP1	XP SP2	XP SP3	Vista RTM	Vista SP1, SP2	Win7 RTM, SP1
Set process DEP policy support	n	n	y	n	y	y

Figure 3: Availability and default settings of platform mitigation features by Windows operating system version (server).

	Srv03 RTM	Srv03 SP1, SP2	Srv08 RTM	Srv08 R2 RTM, SP1
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				

	Srv03 RTM	Srv03 SP1, SP2	Srv08 RTM	Srv08 R2 RTM, SP1
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

Figure 4: Availability and default settings for Visual C++ compiler mitigation features and flags.

Visual C++ Compiler Tools	VC6	VC7 (VS2002)	VC7.1 (VS2003)	VC8 (VS2005)	VC8.1 (VS2005 SP1)	VC9 (VS2008)	VC10 (VS2010)
GS							
stack cookies	n	OptOut	OptOut	OptOut	OptOut	OptOut	OptOut
string buffers	n	OptOut	OptOut	OptOut	OptOut	OptOut	OptOut
strict_gs_check pragma	n	n	n	n	OptIn	OptIn	OptIn
non-pointer arrays	n	n	n	n	n	n	OptOut
structs (pure data)	n	n	n	n	n	n	OptOut
variable reordering	n	n	OptOut	OptOut	OptOut	OptOut	OptOut
shadow parameter copying	n	n	n	OptOut	OptOut	OptOut	OptOut
operator new[] integer overflow check	n	n	n	AlwaysOn	AlwaysOn	AlwaysOn	AlwaysOn
Linker flags							
/DYNAMICBASE	n	n	n	OptIn	OptIn	OptIn	OptOut
/SAFESEH	n	n	OptOut	OptOut	OptOut	OptOut	OptOut
/NXCOMPAT	n	OptIn	OptIn	OptIn	OptIn	OptIn	OptOut

The tables in Figure 5 and Figure 6 provide a breakdown of the mitigation technologies that are enabled by default for major versions of Microsoft Internet Explorer and Microsoft Office when running on different versions of Windows. The key point to observe is that the latest versions of these products benefit the most in terms of exploit mitigation technologies when running on the latest versions of Windows. It should be noted that since Windows XP Service Pack 2 all versions of Internet Explorer have been built with /GS and SAFESEH enabled. Microsoft Office has been built with /GS since Microsoft Office 2003 and SAFESEH since Microsoft Office 2007.

In Figure 5 and Figure 6 green indicates that the feature is enabled.

Figure 5: Default settings for exploit mitigation technologies for Microsoft Internet Explorer by version of Windows.

	Internet Explorer 6	Internet Explorer 7	Internet Explorer 8	Internet Explorer 9[3]
XP SP2	SEHOP Heap terminate DEP ASLR (images & stacks)	SEHOP Heap terminate DEP ASLR (images & stacks)	SEHOP Heap terminate DEP ASLR (images & stacks)	
XP SP3	SEHOP Heap terminate DEP ASLR (images & stacks)	SEHOP Heap terminate DEP ASLR (images & stacks)	SEHOP Heap terminate DEP ASLR (images & stacks)	
Vista RTM		SEHOP Heap terminate DEP ASLR (images & stacks)	SEHOP Heap terminate DEP ASLR (images & stacks)	
Vista SP1, SP2		SEHOP Heap terminate DEP ASLR (images & stacks)	SEHOP Heap terminate DEP ASLR (images & stacks)	SEHOP Heap terminate DEP ASLR (images & stacks)
Win7			SEHOP Heap terminate DEP ASLR (images & stacks)	SEHOP Heap terminate DEP ASLR (images & stacks)

[3] Internet Explorer 9 requires Windows Vista Service Pack 2, Windows 7 RTM, or above.

Figure 6: Default settings for exploit mitigation technologies for Microsoft Office by version of Windows.

	Microsoft Office 2003	Microsoft Office 20074	Microsoft Office 2010
XP SP2	SEHOP DEP ASLR (images & stacks)	SEHOP DEP ASLR (images & stacks)	SEHOP DEP ASLR (images & stacks)
XP SP3	SEHOP DEP ASLR (images & stacks)	SEHOP DEP ASLR (images & stacks)	SEHOP DEP ASLR (images & stacks)
Vista RTM	SEHOP DEP ASLR (images & stacks)	SEHOP DEP ASLR (images)	SEHOP DEP ASLR (images & stacks)
Vista SP1, SP2	SEHOP DEP ASLR (images & stacks)	SEHOP DEP ASLR (images)	SEHOP DEP ASLR (images & stacks)
Win7	SEHOP DEP ASLR (images & stacks)	SEHOP DEP ASLR (images)	SEHOP DEP ASLR (images & stacks)

[4] Microsoft Office 2007 Service Pack 3 enabled ASLR support for stacks.

Call to action

Maximizing the effectiveness of the exploit mitigation technologies described in this document requires action on the part of software vendors, enterprise administrators, and home and business users. These actions are itemized in the following subsections.

Software vendors

- Build your software with exploit mitigation technologies such as DEP, ASLR, SEHOP, and /GS enabled by default. Detailed instructions on how this can be accomplished are available at: http://msdn.microsoft.com/en-us/library/bb430720.aspx.
- Verify that your software has been built with DEP, ASLR, SEHOP, and /GS enabled by taking advantage of the SDL BinScope tool developed by Microsoft, which is available at: www.microsoft.com/downloads/en/details.aspx?displaylang=en&FamilyID=90e6181c-5905-4799-826a-772eafd4440a.

Enterprise IT departments

- Require ISVs and application suppliers to opt in to exploit mitigations as part of the acceptance criteria when procuring an application.
- Use EMET to enable exploit mitigation technologies for critical applications that may be at risk of being attacked. EMET can be downloaded from: http://go.microsoft.com/fwlink/?LinkID=200220&clcid=0x409.
- Enable exploit mitigations such as SEHOP (provided by the client/server Windows platform) on a system-wide basis whenever possible.

Home and business users

- Demand that software vendors enable exploit mitigation technologies.

- Use EMET to enable exploit mitigation technologies for critical applications that may be at risk of being attacked. EMET can be downloaded from: http://go.microsoft.com/fwlink/?LinkID=200220&clcid=0x409.

References

/GS Stack Buffer Overrun Detection

- Compiler Security Checks In Depth: http://msdn.microsoft.com/en-us/library/aa290051.aspx
- /GS cookie protection—effectiveness and limitations: http://blogs.technet.com/b/srd/archive/2009/03/16/gs-cookie-protection-effectiveness-and-limitations.aspx

  Enhanced /GS in Visual Studio 2010: http://blogs.technet.com/b/srd/archive/2009/03/20/enhanced-gs-in-visual-studio-2010.aspx

Data Execution Prevention (DEP)

- Understanding DEP as a mitigation technology:
  - Part 1: http://blogs.technet.com/b/srd/archive/2009/06/12/understanding-dep-as-a-mitigation-technology-part-1.aspx
  - Part 2: http://blogs.technet.com/b/srd/archive/2009/06/12/understanding-dep-as-a-mitigation-technology-part-2.aspx
- On the effectiveness of DEP and ASLR: http://blogs.technet.com/b/srd/archive/2010/12/08/on-the-effectiveness-of-dep-and-aslr.aspx

Address Space Layout Randomization (ASLR)

- On the effectiveness of DEP and ASLR: http://blogs.technet.com/b/srd/archive/2010/12/08/on-the-effectiveness-of-dep-and-aslr.aspx

SAFESEH and Structured Exception Handler Overwrite Protection (SEHOP)

- /SAFESEH: http://msdn.microsoft.com/en-us/library/9a89h429(VS.80).aspx
- How to enable SEHOP: http://support.microsoft.com/kb/956607
- Preventing the exploitation of SEH overwrites: http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx

Heap Metadata Protection

- Windows Vista Heap Management Enhancements: www.blackhat.com/presentations/bh-usa-06/BH-US-06-Marinescu.pdf
- Preventing the exploitation of user mode heap corruption vulnerabilities: http://blogs.technet.com/b/srd/archive/2009/08/04/preventing-the-exploitation-of-user-mode-heap-corruption-vulnerabilities.aspx

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

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