Hardening Windows Applications

olleB - olle@toolcrypt.org

About this document

This paper is aimed at Windows developers in the hope of explaining how Windows' security features can be used to better secure their applications. It is *not* a complete guide to the Windows security model and may skip details that were deemed unimportant at the time of writing. Please bear that in mind while reading. Thank you.

Introduction to Windows security

In this introductory chapter some basic concepts of the Windows security model are explained in just enough detail to understand the material presented in the following chapters. If you are already familiar with how the Windows model of access control works, feel free to skip ahead and use this chapter for reference only. Another great reference is the MSDN section on Access Control available at http://msdn.microsoft.com/en-us/library/aa374860.aspx.

Security Descriptors and Access Control Lists

A central concept in the Windows security model is the Security Descriptor (defined by the SECURITY_DESCRIPTOR structure) which contains all security related information about an instance of an operating system resource. The Security Descriptor for a particular resource can, assuming the current user has sufficient access to the resource, be queried and set from an application in order to inspect and/or modify its components. These components are:

- A bitfield of control flags that describe properties of the Security Descriptor
- The Security Identifier (or SID) that uniquely identifies the owner of the resource
- The SID of a security group (which is only used for POSIX compatibility)
- A System Access Control List (or SACL) which controls auditing of access to the resource
- A Discretionary Access Control List (or DACL) that controls access to the resource

All components of a Security Descriptor (except for the control flags) are optional. A resource with a Security Descriptor with no DACL would mean that no access control restrictions apply to the resource and all access is granted to everyone, this is commonly called a “Null DACL”.

The SACL and DACL are both lists of Access Control Entries (or ACEs) which contain:

- The type of the ACE (e.g. deny, allow, audit)
- A bitfield that controls the inheritance properties of the ACE (i.e. can or can't be inherited)
- The SID of the “trustee” to which this entry applies
- A bitfiled (called an “access mask”) that defines what types of access are governed by this entry

When the DACL of a resource is checked against the “Access Token” of a requesting thread, the Access Control Entries are checked in order and when an ACE where the “access mask” matches the type of access and where the SID in the ACE matches either a user or group SID in the “Access Token”, access is immediately granted or denied depending on the type of ACE encountered. Should no ACEs in the DACL match the requesting Access Token, access is always denied. One could say the DACLs are checked in a “first match, default deny” mode.

Objects and Handles

The Windows NT kernel is built on a concept of kernel "objects" that represent different resources of different types (file, process, timer, etc.). These are, simply put, reference-counted data structures which share a common header (defined by the OBJECT_HEADER structure). The common object header contains, among other things, information about which type an object is an instance of and a pointer to a Security Descriptor which controls access to the object. All instances of object types that can be accessed by name (and some that can't, like processes and threads) have to have a Security Descriptor defined and are referred to by Microsoft documentation as "Securable Objects". The Security Descriptor of a Securable Object can be read and set through the user-mode accessible APIs such as "(Get|Set)SecurityInfo" and "(Get|Set)KernelObjectSecurity".

To expose kernel objects to user-mode applications, the concept of a "Handle" is used. The handle is defined as a pointer-wide value that the kernel uses to keep track of user-mode references to an instance of an object. When an object access is requested from the application (such as when opening a file by name or creating a new semaphore object) the requested kernel object instance is created (if it doesn't already exist) and an opaque handle value is associated with it and returned to the requesting process. The handles are kept track of by the kernel on a per-process basis (to maintain the inter-process security boundary) and objects are dereferenced when the handle is closed (using CloseHandle or on process termination).

Before the kernel grants the requesting user a handle, it first checks the Security Descriptor of the object instance against the "Access Token" of the requesting thread. It is during this access check that DACLs are used to permit (or explicitly deny) access and SACLs are used to generate audit events.

Tokens and Privileges

We have made reference to the "Access Token" of the thread requesting a handle to a particular resource from the kernel. This Access Token is a kernel object which describes the security context of a process (or thread within a process) and thus defines which operating system resources and functions it can be granted access to. The Access Token contains all the relevant security information needed to make access control decisions, including the SIDs identifying user ID and group membership, the current state of system "Privileges", and on Windows Vista and later information about the "Integrity Level" and "Elevation Status" of the Token.

Each process has a "Primary Token" associated with it from process creation. This token can, assuming required privileges are held, be manipulated in order to restrict or raise the access of a particular process (e.g. by using the user-mode accessible "CreateRestrictedToken" or "SetTokenInformation" APIs). A certain thread within a process can also be temporarily assigned an "Impersonation Token" in order to, for example, act on behalf of a certain user (see the "ImpersonateLoggedOnUser" and "SetThreadToken" APIs).

The primary means of controlling access to resources in Windows is the DACL defined in the Security Descriptor, however, there are additional system "Privileges" that control access to system functions such as working with audit event logging or managing storage volumes. Privileges that override normal security are also defined, such as the privileges to backup and restore files without regard for DACLs, take ownership of files without regard for DACLs, and the big one; "Act as part

of the operating system" also known as the "TCB" Privilege which allows the creation of arbitrary Access Tokens without user authentication (see the "LogonUser" API).
Privileges can be granted to a user or group of users and are usually included in a "disabled" state in the Primary Token. Before using a privilege it is necessary to "enable" it by manipulating the Access Token of the current thread or process. This provides a means of enabling a certain privilege only for a specific thread in a process by assigning an Impersonation Token with the desired Privilege enabled to that thread. All other threads in the process would then be unaffected.

Windows features related to security

This chapter introduces some of the Windows features that are related to security and describes how they are implemented and how they can be used by developers. If no comments are made about availability, features are supported on Windows XP SP3 / Windows Server 2003 SP1 and later.

Restricted Tokens

In order for an application to "drop" privileges and access rights in order to run sensitive code, the Windows security model allows the creation of a "Restricted" version of an Access Token. This Token can delete Privileges so they can't be enabled and used, set attributes of existing user and group SIDs so that they can't be used used to grant access in DACL checks and restrict the SIDs used in DACL checks to a subset of the SIDs defined in the Token. To create a Restricted Token, an application calls the "CreateRestrictedToken" API with a handle to the Token to be restricted, it is then provided with a handle to a copy of that Token that has been restricted.

A Restricted Token can be assigned to a thread or, more interestingly, can be used with the "CreateProcessAsUser" API to start a new process with the Restricted Token as Primary Token. Even though setting the Primary Token is usually a privileged operation that requires the "SeAssignPrimaryTokenPrivilege" system Privilege, if the Token passed to "CreateProcessAsUser" is a restricted token of the calling processes own Primary Token then the Privilege isn't needed.

The Desktop security boundary

All interactive logins (both on a console or via RDP) are assigned a "Session". Prior to Windows Vista, the first interactive logon was assigned a Session ID of 0 (Zero). In Windows Vista and later versions, Session 0 is reserved for Windows Services and the first logon becomes Session 1.

Within a session there are one or more "Windows Stations", these Window Stations are Securable Objects that contain a Clipboard and one or more "Desktops". The only interactive Window Station is the first one, which is named "Winsta0" and is associated with the users' input and display devices. A user cannot directly interact with any other Window Stations in the Session.

A Desktop is also a Securable Object which acts as a container for user interface objects like windows and the messages that create interactions between them. These user interface objects (sometimes called "User Objects" in Microsoft documentation) are *not* Securable Objects but belong to the Win32 environment which is more properly called the "Windows subsystem".

Since the Desktop object *is* a Securable Object it can be used to enforce a security boundary between two Win32 applications running in different security contexts. Since the Win32

environment does *not* enforce any security boundary between windows and since these messages can contain information such as function pointers this means that processes with windows on the same Desktop can directly influence each other in nasty ways. For an example of what can happen when an unsuspecting application receives malicious messages see the 2002 “Shatter Attack”[0].

An example of where Microsoft uses this feature would be the “WinLogon” Desktop, which is the first Desktop of a session where the user logon interaction takes place and which is protected by a very restrictive DACL. This is also the Desktop which is activated when you press the “Secure Attention Sequence” (i.e. Ctrl-Alt-Del) and on which the UAC “Elevation” dialog is presented. Since these sensitive user interactions are performed on a separate “secured” Desktop, any applications in the users security context will not be able to meddle with them. An example would be a key-logger that uses “Windows Hooks” (see the “SetWindowsHookEx” API) which would not be able to intercept key-presses when a different Desktop is active and cannot gain access to the “secured” Desktop to install its hook because of the restrictive DACL on the Desktop Object. This feature can be used in applications that have similar sensitive interactions with the user, such as credential entry or document signing dialogs to protect them from simple key-loggers and modification by other processes.

Imagine you were creating a new sandbox process with a Restricted Token to run some potentially hostile code. Since the new process inherits its Desktop from the parent process, it can interact with other applications on the Desktop by posting Win32 messages to their windows. To stop this you could create a new (non-interactive) Window Station and a Desktop Object on that Window Station by using “CreateWindowStation“ and “CreateDesktop” with suitable default DACLs[1] and assign the new process to the sandbox Desktop by passing its name in the “lpDesktop” member of the “STARTUPINFO” structure parameter to the “CreateProcess” API. In a real-world example the Chromium browser uses this technique to strengthen the sandboxing of its rendering processes[2].

Job objects

In the release of Windows 2000 and the advent of “Terminal Services” (now referred to as “Remote Desktop Services”) Windows went from supporting only one interactive logon at a time to a model with a potentially very large number of simultaneous users. Microsoft then had to solve the problem of a single user consuming more than a “fair share” of the total system resources and thus degrading service to the other users. The solution implemented was the Job Object. The Job Object is a Securable Object which acts as a container for processes (that belong to the same Session) and enforces a limiting policy on them.

Examples of the limits that can be imposed on processes associated with a Job Object are:

- Maximum number of processes associated with the job (preventing “fork-bombs”).
- Restrictions on committed memory (preventing physical memory starvation).
- Execution time limits to prevent runaway processes (killing off endless loops).
- Prohibit the calling of sensitive system APIs such as “SystemParametersInfo”.
- Prohibit threads from changing the interactive Desktop using the “SwitchDesktop” API.
- Prohibit access to user interface objects such as the Clipboard.

As you can see from this non-exhaustive list, Job Objects can be very useful when running code that can potentially misbehave given unexpected input, for example file format parsers and other complex processing tasks. Job Objects make a good strengthening component of a sandbox for

untrusted code or can just be used to limit the amount of damage a component of your application can do when processing malformed input. The Chromium browsers sandbox uses Job Objects to limit the impact of running untrusted code on the system[2].

Mandatory Integrity Control

Windows Vista introduced a new concept to the Windows security model called Mandatory Integrity Control or “MIC”. A new type of ACE was introduced in the SACL, called a “Mandatory Label” where the SID can specify an “Integrity Level” and the attributes define a policy as follows:

- No Write Up - A process of lower integrity cannot modify a higher integrity object
- No Read Up - A process of lower integrity cannot get read access to a higher integrity object
- No Execute Up - A process of lower integrity cannot execute a higher integrity object

A few Integrity Level values are predefined by Windows, these are Untrusted (0), Low (4096), Medium (8192), Medium+ (8448), High (12288), System (16384) and Protected (20480).
The default Integrity Level and policy for Securable Objects which do not have a Mandatory Label explicitly defined is Medium and No Write Up. For process objects No Read Up is also the default.

When a process is created it inherits the Integrity Level of its parent process, except in the case where the executable file that is used to start the process has an explicit Mandatory Label with Integity Level lower than the parent process. In that case, the new process is started with the lower Integrity Level. For a process to set a Mandatory Label with a higher Integrity Level than its own on a Securable Object, its Acces Token has to have the “SeRelabelPrivilege” Privilege, though any process which has acquired “WRITE_OWNER” access (often called “Take ownership” rights) to an object can set a Mandatory Label on it with an Integrity Level equal or lower to that of its own.

Note that, since the Integrity Level of a process is defined by a Mandatory Label ACE in the SACL of the process' Primary Token, there can be no separation between threads within the process, even though these threads can have differing Integrity Levels defined in their Security Descriptors. This means that if you would like to run some code at Integrity Level “Low”, you probably want to spawn a new process to contain it. This can be done by passing a Token that has a Mandatory Label ACE in its SACL to the “CreateProcessAsUser” API.

Mandatory Labels are “Mandatory” in that they are checked before the DACL is checked in every access check performed by the system, however they are not defined by Microsoft as a security boundary in that bypasses of the “Mandatory” controls are not treated as security bugs. In effect what Microsoft is telling us is that we should not rely on a timely fix to such a bug and therefore not build software that depends on *only* this function for security.

An example of an application that uses MIC for additional security is Microsoft's own Internet Explorer 7 (on Vista and later). IE7 introduced a new feature called Protected-Mode[3], which is basically running the whole browser in a “Low” Integrity Level process. This prevents attackers who have gained control of the browser to modify the system, as all user applications and resources are at Integrity Level “Medium” (and Windows system resources and services can be even higher) though it does nothing to prevent the browser from stealing the users' information as only the “No Write Up” policy is used. In an application with less dependancies (such as a separate process with clearly defined interfaces to the parent application) it would probably be easier to implement the full policy set, though “No Write Up” is a pretty good start.

User Account Control

User Account Control or "UAC" was introduced in Windows Vista along with many other security improvements and is probably the most misunderstood security feature Microsoft has ever released.

The point of UAC is to make logged in Administrators run in the security context of the "standard user" most of the time and only enabling their Administrative rights when explicitly needed (through a process called "Elevation"). This is referred to as running in "Admin Approval Mode". With this restriction, Microsoft hope to "persuade" developers of third-party applications *not* to assume that their application would run with Admin privileges and, as an example, not attempt to store its configuration in the "Program Files" directory or the HKEY_LOCAL_MACHINE hive.

This lowering of privileges is implemented by an extension to the Access Token, where a Token can include a "Link" to another Token and information about which type it is ("Elevated" or not). When Administrators log in, their initial process receives a Primary Token with the same group membership and system Privileges as a standard user would. This Token also has a link to the "full" Administrator Access Token and a flag that indicates that the current Token is "limited".

To determine if an Access Token is a "limited" Token with a "full" Linked Token attached, call the "GetTokenInformation" API with the "TokenElevationType" information class and check for the "TokenElevationTypeLimited" value. Then, to retrieve a handle to the "full" Linked Token call "GetTokenInformation" with the "TokenLinkedToken" information class.

To enable the "full" Access Token on (or "Elevate") a process, a new service was created called the Application Information Service (AIS or "AppInfo") which displays the user with a Consent dialog before switching the Primary Token of the target process to the "full" Token. Operating system functions have been altered to call into the AIS and request elevation when it is deemed necessary, for example when copying a file to a DACL-protected directory.

To allow applications that legitimately require Admin privileges to continue doing so, a way of requesting Admin privileges on startup was introduced through the application "Manifest"[4]. When the "requestedElevationLevel" key is set to either "highestAvailable" or "requireAdministrator" in the Manifest, the "CreateProcess" API will call AIS to display the Elevation prompt.

There are actually two different kinds of Elevation prompts, the normal Consent dialog that users in Admin Approval Mode see when Elevating to perform an action which requires Admin rights and the Credential entry dialog which other users will see if, for example, attempting to install a software package or run an application with a Manifest which specifies it requires Admin rights. Note that in this case there is no Linked Token as the Elevated process will run with the security context of the user that entered the credentials as Primary Token (this is commonly called Over-The-Shoulder or "OTS" Elevation after the intended usage scenario).

Microsoft has stated in documentation and through communications that UAC does *not* implement a security boundary and, thus, bypasses for UAC Elevation will *not* be considered security flaws.

As an application developer, it is important to follow the Microsoft guidelines on UAC so as not to annoy your users with unnecessary Elevation prompts or put them in danger by always requiring Admin rights to run and nullifying the benefit of running as a standard user.

User Interface Privilege Isolation

Since MIC was introduced in Windows Vista, we now have applications running on the same Desktop but with different Integrity Levels. Remembering the Desktop security boundary, we recognize that these applications can mess with each other since they share the same user interface

objects and can pass Win32 messages to each other. To help mitigate this issue, Microsoft has implemented a new feature called User Interface Privilege Isolation or “UIPI” which provides a “Message Filter”, a list of Win32 message types allowed to be passed between windows of different Integrity Levels. The allowed message types in this Message Filter can be altered using the “ChangeWindowMessageFilter” API, however processes at or below “Low” Integrity Level are not allowed to change the Message Filter.

Just as with MIC and UAC, UIPI is *not* treated as a security boundary by Microsoft, in fact they even provide ways around it for “Accessibility” functions such as the built in on-screen keyboard.

Memory protection and exploit mitigation

Historically, a great number of critical security vulnerabilities in Microsoft software have been caused by unsafe memory accesses, such as out-of-bounds array indexing and pointer arithmetic. It is a well established fact that such bugs are hard to avoid when programming in C or C++ and since most of the Windows code-base is C (with C++ a close second), Microsoft have worked hard to create operating system technologies, programming libraries and compiler features that make it harder to exploit security bugs and easier to write secure code. All of these are available to third-party application developers that wish to benefit from the hard work Microsoft has put in since Bill Gates' famous “Trustworthy computing” memo[5]. We won't go into a lot of detail as to how these different technologies work, just briefly describe them and how an application developer can make use of them.

The oldest of these technologies is the stack overflow protection [6], otherwise known as /GS after the compiler flag that enables it. /GS detects memory writes to stack-allocated buffers that exceed the buffers bounds and corrupt the stack by inserting a “cookie” value in each stack frame that is checked after every copy/move operation to make sure it hasn't been overwritten. /GS has been a default-enabled compiler option since Visual Studio 2003 and if you haven't disabled it, it is already protecting you from many common unbounded write operations to stack allocated buffers, although you shouldn't be using it as a “cheap” way around ridding your code of bounds-checking errors. To help avoid such bugs (and more), the Microsoft Security Development Lifecycle or SDL[11] project maintains a list of “banned” API calls[12] that have historically been hard to use in a safe way. This document also provides safer alternatives to these APIs that should be considered instead.

A popular technique[13] that is often used to simplify exploitation of stack-corrupting bugs is to overwrite the saved pointer to a Structured Exception Handling[14] exception handler routine and trigger an exception, thereby redirecting execution to the known address of an exploit payload. Microsoft has implemented two features in Windows to make this technique harder to use, SafeSEH and SEHOP. The SafeSEH linker feature provides Windows with a table of known exception handler routines so that it can verify that the saved function pointer taken from the stack when an exception occurs is in fact a valid exception handler. To verify that your executables and libraries are using SafeSEH, use the “/SAFESEH” linker option[15] which will abort the linker with an error if it cannot produce a SafeSEH table for an image. SEHOP is an operating system feature in Windows Vista and later that, for compatibility reasons, isn't enabled by default on client systems and needs to be enabled by a registry change[16].

Windows has also implemented security features protecting heap structures from corruption, for example by an unbounded copy operation to a dynamically allocated buffer. The most simple of these protections (safe unlinking) is available in Windows SP2 and later, while the stronger protections are available on Windows Vista and later, where applications can enable them by calling

the “HeapSetInformation” API with the “HeapEnableTerminationOnCorruption“ information class.

Another two Windows features designed to make exploitation of security bugs harder are Data Execution Prevention and Address Space Layout Randomization features or DEP and ASLR. These features act together to make the life of exploit developers a pain, DEP[7] by using the CPUs' memory page permission feature to ensure that instructions can only be executed from memory pages marked as “code”, which are also write-protected. This has the benefit of preventing the CPU from executing code from the stack or other data areas and makes writing exploit payloads harder. Because of compatibility issues, DEP can be enabled in a few different modes and many clients systems run in the “OptIn” mode where only applications that indicate DEP-compatibility[8] will be protected. To enable DEP for a process on a system with an “OptIn” policy, while at the same time making sure DEP is permanently enabled and cannot be switched off in an attack, developers can call the “SetProcessDEPPolicy“ API in an application initialization routine.

ASLR is complementary to DEP in that it prevents the generic DEP bypass method of re-using snippets of code from legitimate executable images in memory[9] by making the memory locations of loaded executable images (as well as the location of the stack) hard for the attacker to guess. As with DEP, an application that wishes to take advantage of this protection needs to indicate compatibility for it using a linker option[10]. Please note that this flag needs to be used when linking all DLLs in a process, not just on the main executable. ASLR is supported on Windows Vista and later and the linker option is available in Visual Studio 2005 SP1 and later.

Strategies for hardening Windows applications

Standing on Microsoft's shoulders

The exploit mitigations in both Windows and the Visual Studio toolchain are always evolving and, as you have seen, most of them are extremely “cheap” to implement in an existing project. To make sure you receive the full benefit of these technologies, make sure to build your software with the latest possible version of the compiler, linker, libraries and templates. This of course applies equally to third-party source code, libraries and components you might be using in your project.

To gain additional wisdom from the Microsoft security process, read the Microsoft simplified implementation of the SDL document[17] that covers the security specific development practices mandated within Microsoft. Remember that the SDL is an iterative process so make sure you go back and check for new information when a revision is made to the documentation!

Microsoft has developed a number of software libraries aimed at helping developers write more secure code, one such library is SafeInt[18] which allows you to perform safer integer calculations in C++ by using templates to do runtime checking for integer overflows. A version of SafeInt has even been integrated into Visual Studio 2010[19]. If you are forced to use C (i.e. cannot for some reason compile the code as C++ with Visual Studio) then the Windows SDK has included similar functionality for strict C in the “intsafe.h” header[20] since the Vista release. Another example is the "Secure Template Overload” functionality[21] in the Microsoft C runtime library (CRT) that can automatically convert some unsafe API calls into their safer counterparts.

Securing your application boundaries

To write secure code you have to know where your application boundaries are. Where does input data come from and how is the execution flow impacted by things outside the application? Examples of such boundaries are:

- Data format parsers - Where does the application read data from files or network pipes?
- IPC - Where does the application interact with other processes in the system?
- COM/RPC/Networking - What interfaces are exposed to other applications, remote or local?
- Windows - Messages posted to the message queue aren't always due to user input!
- Driver interfaces - Does the application have kernel/usermode communications?

Some of these boundaries can be secured by careful application of DACLs. Go over your code and every time your application creates a resource (like a shared memory segment or semaphore for communications between threads or across processes), make sure the resource has an appropriate default Security Descriptor set. If it isn't supposed to be opened after it is created and initialized, securing it with a restrictive DACL that denies all access might be a good idea.

Other boundaries are harder to deal with, implementing complex interfaces to other software components can be hard to wrap your head around and even harder to do securely. The Microsoft SDL project has done a lot of work developing a “Threat Modeling”[22] process that is intended to help developers and software architects think about these things in a “destructive” way. If you don't like their models or tools you should at least check out the “Elevation of Privilege” card game[23] developed by the SDL folks in an attempt to make the process of “Threat Modeling” a bit more fun!

When defining your applications' boundaries it helps to have a good modular design internally. If you can map out the different parts of your application and draw their internal interactions and data-flows on a big whiteboard, it should be easy for you to identify the external interactions. If your application doesn't fit on a whiteboard, then each component should be able to fit on a whiteboard and account for all interactions with other components as well as all external interactions.

Partitioning your application code

Untrusted input is probably the largest problem your application faces. If it's a standard desktop application that probably means opening files and parsing file formats. Microsoft Office is one of the most prevalent desktop applications in the world and has been seen as a high value target for attackers because an e-mailed Office document that exploits a file parsing bug would be a very simple way of compromising a lot of computers. Microsoft has tried to mitigate this threat with the “Microsoft Office Isolated Conversion Environment” or MOICE[24] for Office 2003 and 2007. Recently a more mature solution has emerged as the Office File Validation feature in Office 2010. These features use the Desktop isolation and Restricted Token security features with a separate process that is split from the main Office process just to do the file format parsing. This helps isolate any malware that the document might contain so that it cannot modify system resources.

Using the security features detailed in this document you should be able to develop a similar approach that can be used in your own applications. Any highly complex processing, that doesn't depend on a lot of operating system resources, can be partitioned off into a separate process contained in a highly isolated sandbox-environment that only allows it access to the expected input and output resources. This approach would also work for network protocol parsers that could take untrusted input in the format required by the network protocol and deliver structured data with a

strictly defined schema to the main process (which would naturally verify that data according to the schema before accepting it), thus introducing separation between the network and the application.

Once again, if your application has a modular design internally with well-defined interfaces between components then partitioning off some of the more exposed and/or complex modules shouldn't be that hard. Remember the old security maxim of only giving someone just enough access to be able to perform their task; in a perfect world only the user interface component would need the full security context of the user, every other component could probably be constrained in some way, if only by running with a Restricted Token and perhaps a lower Integrity Level.

Wrapping the onion

None of the security features mentioned in this document are any kind of “silver bullet” and the age old “defense in depth” model is as relevant as ever to security. Think of your security defenses as an onion with several layers, should an attacker be able to peel the first layer there would be another layer left for him to penetrate. Even though a security feature may not seem to bring you any additional *coverage* with regards to security (e.g. “that data is trusted, we already validated it!”) or may seem less than bullet-proof (e.g “there's ways around that!”), any and all extra roadblocks you can put in front of an attacker will help your users. Of course there must be some sanity involved, we can't use every feature every time, but when it makes sense; try to put in that little extra effort!

Conclusion

In the last decade, Microsoft has made some huge improvements in the way they develop more trustworthy code and has added many new security features to the Windows operating system. Third-party application developers can take advantage of these advances to improve the security of their own software, often as easily as enabling a compiler or linker option. It is high time the Windows development community follows Microsoft's lead and starts making it harder for attackers by using exploit mitigation techniques and building security in instead of bolting it on afterwards.

References

[0] http://web.archive.org/web/20040216200736/http://security.tombom.co.uk/shatter.html
[1] http://blogs.msdn.com/b/david_leblanc/archive/2007/07/31/practical-windows-sandboxing-part-3.aspx
[2] http://dev.chromium.org/developers/design-documents/sandbox
[3] http://msdn.microsoft.com/en-us/library/bb250462.aspx
[4] http://msdn.microsoft.com/en-us/library/bb756929.aspx
[5] http://www.microsoft.com/about/companyinformation/timeline/timeline/docs/bp_Trustworthy.rtf
[6] http://msdn.microsoft.com/library/aa290051.aspx
[7] http://msdn.microsoft.com/en-us/library/aa366553.aspx
[8] http://msdn.microsoft.com/en-us/library/ms235442.aspx
[9] http://cseweb.ucsd.edu/~hovav/talks/blackhat08.html
[10] http://msdn.microsoft.com/en-us/library/bb384887.aspx
[11] http://www.microsoft.com/security/sdl/
[12] http://msdn.microsoft.com/en-us/library/bb288454.aspx
[13] http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf
[14] http://msdn.microsoft.com/en-us/library/ms680657.aspx
[15] http://msdn.microsoft.com/en-us/library/9a89h429.aspx
[16] http://support.microsoft.com/kb/956607
[17] http://go.microsoft.com/?linkid=9708425
[18] http://safeint.codeplex.com/
[19] http://msdn.microsoft.com/en-us/library/dd570023.aspx
[20] http://blogs.msdn.com/b/michael_howard/archive/2006/02/02/523392.aspx
[21] http://msdn.microsoft.com/en-us/library/ms175759.aspx
[22] http://www.microsoft.com/security/sdl/getstarted/threatmodeling.aspx
[23] http://www.microsoft.com/security/sdl/eop.aspx
[24] http://www.microsoft.com/technet/security/advisory/937696.mspx