

Cracking Windows Access Control

Andrey Kolishchak
<http://www.gentlesecurity.com>

Abstract

Windows access control evolves from the original model to address actual security threats. In overall it becomes more and more complicated. Starting from Windows Vista, access control shifts towards mandatory access control (MAC). The shift is used to mitigate weaknesses of discretionary access control (DAC) that relies upon user's settings. This paper discusses the main weaknesses of discretionary access control such as privilege elevation through impersonation, complexity of configuration, ownership specific and uncontrollable objects. Then it describes how the issues partly resolved by new mandatory access control model also known as integrity levels. The main focus is given to the least privileges concept or so called "security boundary" that could be achieved by the combination of access control lists and integrity levels. Security boundary is required to reduce attack surface, damage caused by malware and targeted intrusions. While the recent access control developments help to enforce a real security boundary around untrusted applications, they have a number of described usage limitations. Finally, the paper concludes general inadequacy of discretionary user based access control and advocates for true per-process permissions. As per-process permissions apparently are too complicated for a human administrator to configure properly, a solution is proposed addressing the complexity problem.

1. Introduction

Windows Access Control is based on discretionary access control (DAC). The DAC is a user-oriented security policy, which restricts access to various operation system objects, such as files, processes, etc. In Windows implementation, each controllable object gets a list of permissions or Access Control List (ACL), an access control matrix - USER x OBJECT, e.g.:

	OBJECT_A
USER_A:	READ
USER_B:	READ/WRITE
USER_N:	FULL (READ/WRITE/EXECUTE)

(1) Permissions to the objects are assigned by users themselves or inherited from the objects containers. (2) A creator of object, Owner,

implicitly gets full permissions regardless of actual ACL. From one side, properties (1) and (2) make overall access control model very flexible. From another side, these properties introduce many non-obvious interconnections and as result increased complexity.

In general, it is a challenge to prove that USER_A cannot get access to certain object. Even if current state indicates no permission paths to the object, there is no guaranty that USER_A will not receive permissions due to privilege elevation chain or administrator mistake.

In such model, the whole strength of access control heavily relies on user and inherits all corresponding weaknesses.

Note, the weaknesses discussed in this paper are related to the local privilege elevation, a

process when user receives permissions which were not explicitly assigned to him.

2. Windows DAC Weaknesses

First and most obvious weakness is **dependence on proper user authentication**. A user could provide fake identity via attacking authentication procedure. The attacks include:

- Social engineering;
- Stealing authentication information and keys;
- Passwords brute-forcing and sniffing over the network;
- Key-logging.
- Etc.

The attacks fake user identity and receive user's privileges.

The impersonation weaknesses are also exploited via faked user identification but that is not related to authentication. The impersonation is a procedure that allows a server application to replace (impersonate) its security identity (credentials) by the identity of a client. In general, impersonation assumes a server reduces its privileges but it also imposes a threat of unauthorized privilege elevation.

To exploit impersonation, an attacker terminates or crashes a privileged server application and starts its own one with the same interface. Attacker's server receives requests from privileged client and impersonates him. There was a number of attacks reported which used this approach exploiting named pipes [1, 2, 3]. However, the scope is not limited to named pipes. Any communication channel that supports impersonation can be hijacked for privilege elevation purposes, including LPC, RPC, DDE, COM, etc.

Starting from Windows XP, impersonation requires a separate privilege - "Impersonate a

client after authentication". Only LocalSystem, Administrators and services have this privilege by default [4] and can impersonate to the client's credentials. Regular users are not able to exploit impersonation anymore, but services (special processes managed by the Service Control Manager) still capable.

Even if service run on behalf of unprivileged user it still poses a security risk if it serves privileged clients. For example, RpcSs, RPC subsystem runs by unprivileged user and serves clients running LocalSystem account. [4]

In general, there is no way to mitigate the risk of impersonation on the server side besides reducing the number of code vulnerabilities.

Another weakness is **a complexity of ACLs configuration**. Due to inherited complexity, in many cases applications and installation programs fail to set proper ACLs on the files, devices, registry and other controllable objects [5]. Typically weak ACLs allow full access to Everyone, Users and Authenticated Users accounts. Exploiting these weak permissions an attacker executes its code in privileged context. For example, in case of full permission for Everyone to executable files, a standard user may over-write those executables. If later an administrator runs them, standard user receives his privileges. Particularly it is dangerous for executables run by system - services.

Weak permissions might be audited by means of various tools, such as: cacls.exe, icaccls.exe, SubInAcl.exe, accesschk.exe. The target for search is objects having general write permissions assigned to Everyone, User and Authenticated Users.

While weak permissions for files and registry are routinely discovered and fixed by software vendors [5], weak permissions on non-storage objects such as memory sections, processes, devices and synchronization objects are generally overlooked and quite common.

Next weakness comes from the property that **creator (owner) of object implicitly receive full permissions**. It means if USER_A creates a file he may set any permission on that file as an owner. One of the consequences of this property is difficulty of revoking permissions for users. For example USER_A member of Sales group is moved from the sales department to marketing one. In that scenario USER_A retains full access for all documents he created even if administrator removes him from the Sales group.

The problem is particularly important for members of Administrators group. That is a reason why all files and other objects created by members of Administrators have Administrators group (not user who created it) as an owner. That is valid for all Windows versions except Windows XP. Windows XP by default set particular administrator user as an owner. The setting can be configured in Security Settings\Security Options:

System cryptography: Use FIPS compliant algorithms for encryption,
System objects: Default owner for objects created by Administrators
System objects: Require case insensitivity for non-Windows subsystem

Additionally, administrator may set a new owner for the object in order to fix security policy violations. Setting of new owner requires explicit changing owner permission or SeTakeOwnershipPrivilege privilege, which is assigned to Administrators by default.

Another aspect of ownership weakness is processes. A user who creates the process becomes owner of process object. That is particularly important in case of services as most of them run on behalf of the same user account, such as LocalSystem, NetworkService and LocalService. A successful attack on one of these services, leads to compromising of all others running as the same account [4].

The issue with services is partly addressed in Windows Vista where process objects might be owned by unique service SID, symbolic: NT Service\ServiceName. However, that is not

enabled for all services by default. Not even all services coming with Vista support unique service SIDs. Additionally, in Windows Vista, there is a way to control owners permissions via Owner Rights SID [6].

Finally, one of the most sufficient weaknesses is the fact that **permissions cannot be assigned to all objects**, e.g.: network access, windows subsystem objects. It means these objects are uncontrollable and can be accessed by any user. This weakness is not directly related to DAC but rather to its Windows implementation.

Controlling network access is resolved by supplementary solutions – firewalls and routers that may control traffic on per-user basis. Also Windows Vista has the interface to control network access for individual services.

Uncontrolled access to windows subsystem objects may lead to privilege elevations when specially crafted messages (commands) sent to windows of privileged processes (this is also known as “Shatter Attacks” [7]). Microsoft noted about the issue as early as in 1994 [8] and probably the first info about such attacks was published in Microsoft System Journal by Matt Pietrek in 1997 [9]. Anyway the problem received a public attention only in 2002 [10]. There were several vulnerabilities in services which used windows subsystem to display pop-up dialogs. Since then services were fixed and the main target became privileged applications that run on the same desktop. For example, a logged standard user may launch runas in order to execute some applications with administrative privileges (program installation, administrative routines, etc.). This privileged application could be attacked through windows subsystem objects by malware running as standard user and on the same desktop.

Beside uncontrolled windows subsystem objects there is a closely related problem – uncontrolled installation of desktop wide hooks (SetWindowsHookEx) that injects dynamic link libraries (DLLs) into all processes on given

windows desktop including privileged processes.

Preventing these privilege elevations requires running privileged applications on a different windows desktop associated with different workstation, which could be achieved via starting processes as jobs [11]. But a real solution for this problem appears only in Windows Vista and it is Integrity Levels.

3. Mandatory Access Control – Integrity Levels

Mandatory Access Control (MAC) assumes that access permissions are defined by system itself and are not controlled by user. An example of Windows mandatory access control is Windows File Protection which protects critical system files regardless of assigned user's permissions, so even administrator cannot modify the protected files.

Windows Vista goes farther and extends traditional access control by Integrity Levels (ILs). The ILs are designed to address trustworthiness of running application and objects such as files, registry, etc. The integrity level is a label that assigned to every process and object. A label is a number in the range from 0 up to 16384. A higher number represents higher integrity level. Practically Windows Vista uses only four ILs defined in this range: Low Integrity, Medium Integrity, High Integrity and System Integrity [12].

A process with lower integrity gets number of restrictions on what it could do with higher integrity objects. Windows Vista implements three restriction policies.

No-Write-Up policy restricts generic write access. The policy is used to prevent modification of trusted objects, such as files and registry. So, untrusted process cannot write to trusted locations regardless of user's permissions.

No-Read-Up restricts generic read access and used to prevent reading memory of trusted processes.

No-Exec-Up limits generic execution access. By default system uses the policy to prevent execution of higher integrity COM methods.

Integrity level of process correspond privileges of user who launched it. LocalSystem, LocalService and NetworkService receive System Integrity; Administrators – High Integrity; Users – Medium Integrity and Everyone – Low Integrity.

By default all present and new objects created by Medium, High and System Integrity processes have Medium Integrity level assigned. The exceptions from this rule are: process, thread, access token and job objects which get the same integrity level as parent process. It means Medium Integrity process may write to files and registry of higher integrity processes. Additionally, the system maintains the locations with Low Integrity (LI) files and registry. All objects including files and registry created by LI processes receive LI level. As a result No-Write-Up policy for files and registry is effective only for LI processes. Above LI, files and registry still protected by user permissions (DAC).

Beside traditionally controllable objects, ILs control access to windows subsystem objects. That is called User Interface Privilege Isolation (UIPI). With UIPI, lower integrity process cannot send messages to windows of higher integrity processes. Also lower integrity process cannot install desktop wide hooks and inject DLLs.

Internet Explorer's Protected Mode uses full power of ILs by performing its main functions at Low Integrity level [13, 14]. Protected Mode assumes two processes.

- `ieplcore.exe` – main browser processes responsible for rendering of html. The process is restricted by Low Integrity level.

- ieuser.exe is a broker that runs at Medium Integrity and exposes RPC interface for iexplore.exe in order to perform certain privileged operations, such as "Save File" dialog box.

Thus, browser works in confined environment, and may not get out of it (elevate privileges) unless there are implementation bugs in ieuser.exe or in other LI processes. Adding elements of mandatory access control brought essential security boundary that maintained by system itself and does not depend on user configuration. At the same time browser retains high level of usability.

So, ILs complement Windows discretionary access control by mitigating described DAC weaknesses. Starting from Windows Vista ILs become a new target for attacks.

4. Exploiting Windows ILs

One of the main tasks of ILs is enabling a security boundary among processes started by the same user. So circumvention mechanisms suppose running code at elevated integrity level.

While by design ILs may provide a strict security boundary, certain defaults are weakened for the sake of applications compatibility. Among such weaknesses is **assigning Medium Integrity level to all objects by default** (except process, thread, token and job objects). As result files and registry written by Medium Integrity processes are used by High and System Integrity processes. It matches the same security level as provided by DAC. For example: administrator running at High Integrity may install a SCM service, which is executed at System Integrity level. Due to this, the main gain from ILs policies, in case of Medium Integrity and above, is UIPIs restrictions. The rest is still controlled by means of standard ACLs. A strict security

boundary is used only around Low Integrity processes.

Another weakness related to the defaults is allowance of **UIPI bypassing for UI automation applications**, formerly known as Microsoft Active Accessibility. Such applications normally require unrestricted access to windows subsystem objects and Windows Vista allows that for any application which comply with following requirements:

1. Presence of UIAccess="true" clause in the application manifest;
2. Digital signature verifiable via machine's Trusted Root Common Authorities;
3. Application binary location should be in %ProgramFiles% or %WinDir% (in case of WinDir certain directories are excluded due to weak permissions).

UI automation applications started by AppInfo service which also assigns elevated integrity level: High Integrity in case of protected administrator and Medium+16 for standard users. This prevents code injection attacks. Normally AppInfo is used by User Account Control (UAC) subsystem and before elevation receives a confirmation from user via pop-up dialog. In case of UIAccess=true, no dialogs are displayed and elevation fully transparent for user.

Complying the first requirement is trivial. The second introduces some cost but still achievable. The main obstacle for exploiting is the third as attacker must have permissions to %ProgramFiles% and %WinDir%, which are normally assigned only to administrators. Moreover, if attacker is able to circumvent this requirement then it is sufficient for him to provide the rest. Due to complexity of ACLs configuration there is a chance to find such location in certain configurations.

The simplest technique would be using side-by-side DLL injection for one of the "UIAccess=true" applications that comes with Vista by default: Windows Guided Help

(acw.exe), INF Default Install (infdefaultinstall.exe), Windows Remote Assistance (msra.exe), Narrator (narrator.exe), On-Screen Keyboard (osk.exe), Program Compatibility Assistant (pcaelv.exe), Microsoft Tablet PC Input Component (wisptis.exe). The attack sequence could be following:

1. Finding writable location within %ProgramFiles% - WritebleDirectory;
2. Copying one of existing UI automation application into the WritebleDirectory;
3. In WritebleDirectory creating a custom DLL statically linked by copied application;
4. Starting copied application;
5. During the start system will load custom DLL, hence code of custom DLL would run at elevated integrity level and without UIPI restrictions.

Additionally, there is a trick that allows injecting code into UI automation application run as a standard user. In this case, system increments integrity level by 16 (Medium+16). Hence it is possible to start UI automation application at Medium-16 and as result it will have Medium Integrity level assigned: Medium-16+16.

The described attack scenarios work only for elevation at Medium Integrity level, but cannot be used at Low Integrity. Indeed, by default LI processes may write only to LI location (which is not %ProgramFiles% or %WinDir%) ; and AppInfo fails if called below LI (-16 trick). However, LI might be successfully due to AppInfo implementation bug [15].

AppInfo's RPC, which is used to elevate processes, leaks access handles on newly created processes [15]. These handles allow unrestricted access to created process including code injection. In case of protected administrator, injected code is executed at High Integrity level; hence there is immediate elevation from Low up to High. For the standard user elevated IL is LI+16, but subsequent attacks could be conducted to elevate farther: LI+16+16, etc.

The issue with AppInfo, an implementation bug may happen with any other service or application that (1) expose RPC interface; (2) has COM interface with permissions for execution at LI. Note **any RPC interface might be affected** not only RPC of processes designed to work with LI, such as ieuser.exe.

Another possible exploitation vector is related to the fact that **ILs are enforced only locally**. And network shares are accessed at Medium Integrity level. But a LI process cannot exploit this because access to network shares is prohibited for LI.

Finally, it worth to note, that **No-Read-Up policy is not used for files and registry** in default configuration. As result LI processes may read any files user has permissions to.

5. Per-Application Access Control

ILs provide strict security boundary around LI processes. In spite of certain weaknesses the boundary may effectively isolate damage from successful attacks. However, usage of LI is limited due to following problems.

1. In many cases it would not be possible to configure application to run at LI. Thus, most of applications have to be re-designed in order to properly operate at LI. This requires moving application files and registry into LI accessible location and creating of privileged broker. A broker is necessary as administrator cannot configure what privileged operations are permitted for individual applications.

2. Beside Internet Explorer there are many candidates for running at LI. All applications using the internet including web browsers, email clients, messengers, p2p clients, multimedia applications should run at LI. As result, their objects, such as files and registry, should be labeled as LI and accessible by any LI process. That creates a security breach, which

reduces the whole value of IL. For example, a database of email client would be accessible by browser. Obviously, capacity of LI applications pool is very limited.

So, ILs do not provide usable security boundary. A solution would be a true per-application access control by introducing an additional dimension in access control matrix:

USER x APPLICATION x OBJECT.

That is apparently too much for a human administrator to configure correctly. Even actual two-dimensional ACL matrix is complicated enough. A way to resolve this complexity is forming proper default settings by two approaches.

1. Hiding settings behind a security model, meaning the access control settings are generated automatically in accordance with some policy.
2. Maintaining the applications permissions database that contains knowledge on resources required by various applications. That could be a centralized database or an integrated into applications themselves, e.g. application privileges specification in .NET manifest files.

In terms of implementation of enforcement mechanisms, a first step towards per-application access control is done. Each

Windows Vista service may get a unique service SID and administrator may configure permissions for the individual service even if it runs on behalf generic group such as LocalSystem.

The next steps suppose providing the same unique SID for regular applications (not only services); and integrating applications permissions with user ones.

6. Conclusion

Windows access control has a number of weaknesses which could not be resolved over discretionary access control. Thus Windows Vista goes farther and extends traditional access control by mandatory integrity levels that enforce a security boundary between groups of applications with different trustworthiness. However, usage of integrity levels is limited due to certain weaknesses and inherited design. The next step is true least privilege concept based on per-application permissions. The main obstacle towards this goal is complexity of three dimensional access control matrix. The complexity could be effectively addressed by default security policy, which opens a new direction where security of settings is over to be replaced by security of policies.

References

- [1] @Stake. Named Pipe Filename Local Privilege Escalation.
Available: <http://www.securiteam.com/windowsntfocus/5BP012KAKI.html>
- [2] Maceo. Named Pipe Filename Local Privilege Escalation Exploit.
Available: <http://www.securityfocus.com/archive/1/329197>
- [3] Georgi Guninski. Elevation of privileges with debug registers on Win2K.
Available: <http://www.guninski.com/dr07.html>
- [4] Andrey Kolishchak. The Weakness of Windows Impersonation Model.
Available: <http://www.gentlesecurity.com/adv04302006.html>

- [5] Mark Russinovich. The Case of the Insecure Security Software. Personal blog, June 2007. Available: <http://blogs.technet.com/markrussinovich/archive/2007/06/19/1256677.aspx>
- [6] New ACLs Improve Security in Windows Vista. TechNet Magazine, June 2007.
- [7] Shatter attack. Wikipedia. Available: http://en.wikipedia.org/wiki/Shatter_attack
- [8] Accessing the Application Desktop from a Service. Microsoft KB115825. Available: <http://support.microsoft.com/kb/115825>
- [9] Matt Pietrek. Under The Hook. Microsoft Systems Journal, March 1997. Available: <http://www.microsoft.com/msj/0397/hood/hood0397.aspx>
- [10] Chris Paget. White paper: Exploiting the Win32 API. Bugtraq mail list, August 2002.
- [11] David LeBlanc. Practical Windows Sandboxing, Part 2. Personal blog, July 2007. Available: http://blogs.msdn.com/david_leblanc/archive/2007/07/30/practical-windows-sandboxing-part-2.aspx
- [12] Windows Vista Integrity Mechanism Technical Reference. MSDN Library. Available: <http://msdn2.microsoft.com/en-us/library/bb625964.aspx>
- [13] Understanding and Working in Protected Mode Internet Explorer. MSDN Library. Available: <http://msdn2.microsoft.com/en-us/library/Bb250462.aspx>
- [14] Michael Howard and David LeBlanc. Writing Secure Code for Windows Vista. 2007.
- [15] Skywing . Getting out of Jail: Escaping Internet Explorer Protected Mode. Uninformed. Volume 8, September 2007. Available: <http://www.uninformed.org>