Getting out of Jail: Escaping Internet Explorer Protected Mode

September, 2007

Skywing
Skywing@valhallalegends.com
http://www.nynaeve.net

Abstract

With the introduction of Windows Vista, Microsoft has added a new form of mandatory access control to the core operating system. Internally known as "integrity levels", this new addition to the security manager allows security controls to be placed on a per-process basis. This is different from the traditional model of per-user security controls used in all prior versions of Windows NT. In this manner, integrity levels are essentially a bolt-on to the existing Windows NT security architecture. While the idea is theoretically sound, there does exist a great possibility for implementation errors with respect to how integrity levels work in practice. Integrity levels are the core of Internet Explorer Protected Mode, a new "low-rights" mode where Internet Explorer runs without permission to modify most files or registry keys. This places both Internet Explorer and integrity levels as a whole at the forefront of the computer security battle with respect to Windows Vista.

1 Introduction

Internet Explorer Protected Mode is a reduced-rights operational mode of Internet Explorer where the security manager itself enforces a policy of not allowing write access to most file system, registry, and other securable objects by default. This mode does provide special sandbox file system and registry space that is permitted to be written to by Internet Explorer when operating in Protected Mode.

While there exist some fundamental shortcomings of Protected Mode as it is currently implemented, such as an inability to protect user data from being read by a compromised browser process, it has been thought to be effective at blocking most write access to the system from a compromised browser. The benefit of this is that if one is using Internet Explorer and a buffer overrun occurs within IExplore.exe, the persistent impact should be lessened. For example, instead of having write access to everything accessible to the user's account, exploit code would instead be limited to being able to write to the low integrity section of the registry and the low integrity temporary files directories. This greatly impacts the ability of malware to persist itself or compromise a computer beyond just IExplore.exe without some sort of user interaction (such as persuading a user to launch a program from an untrusted location with full rights, or other social engineering attacks).

2 Protected Mode and Integrity Levels

Internally, Protected Mode is implemented by running IExplore.exe as a low integrity process. With the default security descriptor that is applied to most securable objects, low integrity processes may not generally request access rights that map to `GENERIC_WRITE` for a particular object. As Internet Explorer does need to be able to persist some files and settings, exceptions can (and are) carved out for low integrity processes in the form of registry keys and directories with special security descriptors that grant the ability for low integrity processes to request write access. Because the IExplore process cannot write files to a location that would be automatically used by a higher integrity process, and it cannot request dangerous access rights to other running processes (such as the ability to inject code via requesting `PROCESS_VM_WRITE` or the like), malware that runs in the context of a compromised IExplore process is (theoretically) fairly contained from the rest of the system.

However, this containment only holds as long as the system happens to be free of implementation errors. Alas, but perhaps not unexpectedly, there are in fact implementation problems in the way the system manages processes running at differing integrity levels that can be leveraged to break out of the Protected Mode (or low integrity) jail. To understand these implementation errors, it is first necessary to gain a basic working understanding of how the new integrity-

based security model works in Windows. The integrity model is key to a number of Windows Vista features, including UAC (User Account Control).

When a user logs on to a computer in Windows Vista with UAC enabled, their shell is normally started as a "medium" integrity process. Integrity levels are integers and symbolic designations such as "low", "medium", "high", or "system" are simply used to indicate certain well-known intermediate values. Medium integrity is the default integrity level even for built-in administrators (except the default "Administrator" account, which is a special case and is exempted from UAC). Most day to day activity is intended to be performed at medium integrity; for instance, a word processor program would be expected to operate at medium integrity, and (theoretically) games would generally run at medium integrity as well. Games tend to be rather poorly written in terms of awareness of the security system, however, so this tends to not really be the case, at least not without added help from the operating system. Medium integrity roughly corresponds to the environment that a limited user would run as under previous versions of Windows. That is to say, the user has read and write access to their own user profile and their own registry hive, but not write access to the system as a whole.

Now, when a user launches Internet Explorer, an IExplore.exe process is launched as low integrity. The default security descriptor for most objects on Windows prevents low integrity processes from gaining write access to medium integrity securable objects, as previously mentioned. In reality, the default security descriptor denies write access to higher integrities, not just to medium integrity, though in this case the effect is similar in terms of Internet Explorer. As a result, the IExplore.exe process cannot write directly to most locations on the system.

However, Internet Explorer does, in certain cases, need to gain write to locations outside of the low integrity (Protected Mode) sandbox. For this task, Internet Explorer relies on a helper process, known as ieuser.exe, which runs at medium integrity level. There is a tightly controlled RPC interface between ieuser.exe and IExplore.exe that allows Internet Explorer, running at low integrity, to request that ieuser.exe display a dialog box asking the user to, say, choose a save location for a file and then save said file to disk. This is the mechanism by which one can save files in their home directory even under Protected Mode. Because the RPC interface only allows IExplore.exe to use the RPC interface to request that a file to be saved, a program cannot directly abuse the RPC interface to write to arbitrary locations, at least not without user interaction.

Part of the reason why the RPC interface cannot be trivially abused is that there also exists some protection baked into the window manager that prevents a thread at a lower integrity level from sending certain, potentially dangerous, messages to threads at a higher integrity level. This allows ieuser.exe to safely display user interface on the same desktop as the IExplore.exe process without malicious code in the Internet Explorer process simply being able to simulate fake keystrokes in order to cause it to save a dangerous file to a dangerous location without user interaction.

Most programs that are integrity-level aware operate with the same sort of paradigm that Internet Explorer does. In such programs, there is typically a higher integrity broker process that provides a tightly controlled interface to request that certain actions be taken, with the consent of the user. For example, UAC has a broker process (a privileged service) that is responsible for displaying the consent user interface when the user tries to perform an administrative task. This operates similar in principal to how Internet Explorer can provide a security barrier through Protected Mode because the lower privileged process (the user program) cannot magically elevate itself to full administrative rights in the UAC case (which runs a program at high integrity level, as opposed to the default medium integrity level). Instead, it could only ask the service to display the consent UI, which is protected from interference by the program requesting elevation due to the window manager restrictions on sending dangerous messages to a higher integrity level window.

3 Breaking the Broker

If one has been using Windows Vista for some time, none of the behavior that has just been described should come across as new. However, there are some cases that have not yet been discussed which one might have observed from time to time with Windows Vista. For example, although programs are typically restricted from being able to synthesize input across integrity levels, there are some limited circumstances where this is permitted. One easy to see instance of this is the on-screen keyboard program (osk.exe) which, despite running without a UAC prompt, can generate keyboard input messages that are transmitted to other processes, even elevated administrative processes. This would at first appear to be a break in the security system; questions along the lines of "If one program can magically send keystrokes to higher integrity processes, why can't another?" come to mind. However, there are in fact some carefully-designed restrictions that are intended to prevent a user (or a program) from arbitrarily being able to execute custom code with this ability.

First of all, in order to request special access to send unrestricted keyboard input, a program's main executable must resolve to a path within the Program Files or Windows directory[1]. Additionally, any such program must also be signed with a valid digital signature from any trusted code signing root. This is a fairly useless check from a security perspective, in the author's opinion, as anybody can pay a code signing authority to get a code signing certificate in their own name; code signing certificates are not a guarantee of malware-free (or even bug-free) code. Although it would be easy to bypass the second check with a payment to a certificate issuing authority, a plain user cannot so easily bypass the first check relating to the restriction on where the program main executable may be located.

[1]Although the author feels that such a check is essentially a giant hack at best, it does effectively prevent a "plain user" running at medium integrity from being able to run custom code that can synthesize keystrokes to high integrity processes, as a plain user would not be able to write to any of these directories

Even if a user cannot launch custom code directly as a program with access to simulate keystrokes to higher integrity processes (known as "uiaccess" internally), one would tend to get the impression that it would be possible to simply inject code into a running osk.exe instance (or other process with uiaccess). This fails as well, however; the process that is responsible for launching osk.exe (the same broken service that is responsible for launching the UAC consent user interface, the "Application Information" (appinfo) service) creates osk.exe with a higher than normal integrity level in order to use the integrity level security mechanism to block users from being able to inject code into a process with access to simulate keystrokes.

When the appinfo service receives a request to launch a program that may require elevation, which occurs when ShellExecute is called to start a program, it will inspect the user's token and the application's manifest to determine what to do. The application manifest can specify that a program runs with the user's integrity level, that it needs to be elevated (in which case a consent user interface is launched), that it should be elevated if and only if the current user is a non-elevated administrator (otherwise the program is to be launched without elevation), or that the program requests the ability to perform keystroke simulation to high integrity processes.

In the case of a launch request for a program requesting uiaccess, `appinfo!RAiLaunchAdminProcess` is called to service the request. The process is then verified to be within the (hardcoded) set of allowed directories by `appinfo!AiCheckSecureApplicationDirectory`. After validating that the program is being launched from within an allowed directory, control is eventually passed to `appinfo!AiLaunchProcess` which performs the remaining work necessary to service the launch request. At this point, due to the "secure" application directory requirement, it is not possible for a limited user (or a user running with low integrity, for that matter) to place a custom executable in any of the "secure" application directories.

Now, the appinfo service is capable of servicing re-

quests from processes of all integrity levels. Due to this fact, it needs to be capable of determining the correct integrity level to create a new process from at this point. Because the new process is not being launched as a full administrator in the case of a process requesting uiaccess, no consent user interface is displayed for elevation. However, the appinfo service does still need a way to protect the new process from any other processes running as that user (as access to synthesize keystrokes is considered sensitive). For this task, the `appinfo!LUASetUIAToken` function is called by appinfo to protect the new process from other plain user processes running as the calling user. This is accomplished by adjusting the token that will be used to create the new process to run at a higher integrity level than the caller, unless the caller is already at high integrity level (0x3000). The way `LUASetUIAToken` does this is to first try to query the linked token associated with the caller's token. A linked token is a second, shadow token that is assigned when a computer administrator logs in with UAC enabled; in the UAC case, the user normally runs as a restricted version of themselves, without their administrative privileges (or Administrators group membership), and at medium integrity level.

If the calling user does indeed have a linked token, `LUASetUIAToken` retrieves the integrity level of the linked token for use with the new process. However, if the user doesn't have a linked token (i.e. they are logged on as a true plain user and not an administrator running without administrative privileges), then `LUASetUIAToken` uses the integrity level of the caller's token instead of the token linked with the caller's token (in other words, the elevation token). In the case of a computer administrator this approach would normally provide sufficient protection, however, for a limited user, there exists a small snag. Specifically, the integrity level that `LUASetUIAToken` has retrieved matches the integrity level of the caller, so the caller would still have free reign over the process.

To counteract this issue, there is an additional check baked into `LUASetUIAToken` to determine if the integrity level that was selected is at (or above) high integrity. If the integrity level is lower than high integrity, `LUASetUIAToken` adds 16 to the integrity level (although integrity levels are commonly thought of as just having four values, that is, low, medium, high, and system, there are 0x1000 unnamed integrity levels in between each named integrity level). So long as the numeric value of the integrity level chosen is greater than the caller's integrity level, the new process will be protected from the caller. In the case of the caller already being a full, elevated administrator, there's nothing to protect against, so `LUASetUIAccess` doesn't attempt to raise the integrity level above high integrity.

After determining a final integrity level, `LUASetUIAToken` changes the integrity level in the token that will be used to launch the new process to match the desired integrity level. At this point, appinfo is ready to create the process. If needed, the user profile block is loaded and an environment block is created, following which `advapi32!CreateProcessAsUser` is called to launch the uiaccess-enabled application for the caller with a raised integrity level. After the process is created, the output parameters of `CreateProcessAsUser` are marshalled back into the caller's process, and `AiLaunchProcess` signals successful completion to the caller.

If one has been following along so far, the question of "How does all of this relate to Internet Explorer Protected Mode" has probably crossed one's mind. It turns out that there's a slight deficiency in the protocol outlined above with respect to creating uiaccess processes. The problem lies in the fact that `AiLaunchProcess` returns the output parameters of `CreateProcessAsUser` back to the caller's process. This is dangerous, because in the Windows security model, security checks are done when one attempts to open a handle; after a handle is opened, the access rights requested are forever more associated with that handle, regardless of who uses the handle. In the case of appinfo, this turns out to be a real problem because appinfo, being the creator of the new process, is handed back a thread and process handle that grant full access to the new thread and process, respectively. Appinfo then marshals these handles

back to the caller (which may be running at low integrity level). At this point, a privilege escalation problem has occured; the caller has been essentially handed the keys to a higher integrity process. While the caller would never normally be able to open a handle to the new process on its own, in this case, it doesn't have to, as the appinfo service does so on its behalf and returns the handles back to it.

Now, in the `ShellExecute` case, the client stub for the appinfo `AiLaunchAdminProcess` routine doesn't want (or need) the process or thread handles, and closes them immediately after. However, this is obviously not a security barrier, as this code is running in the untrusted process and could be patched out. As such, there exists a privilege escalation hole of sorts with the appinfo service. It can be abused to, without user interaction, leak a handle to a higher integrity process to a low integrity process (such as Internet Explorer when operating in Protected Mode). Furthermore, even Internet Explorer in Protected Mode, running at low integrity, can request to launch an already-existing uiaccess-flagged executable, such as osk.exe (which is conveniently already in a "secure" application directory, the Windows system directory). With a process and thread handle as returned by appinfo, it is possible to inject code into the new process, and from there, as they say, the rest is history.

4 Caveats

Although the problem outlined in this article is indeed a privilege escalation hole, there are some limitations to it. First of all, if the caller is running as a plain user instead of a non-elevated administrator, appinfo creates the uiaccess process with integrity level 0x1010 (low integrity + 16). This is still less than medium integrity (0x2000), and thus in the true limited user case, the new process, while protected from other low integrity processes, is still unable to interfere with medium integrity processes directly.

In the case where a user is running as an administrator but is not elevated (which happens to be the default case for most Windows Vista users), it is true that appinfo.exe returns a handle to a process running at high integrity level. However, only the integrity level is changed; the process is most certainly not an administrator (and in fact has BUILTIN Administrators as a deny only SID). This does mean that the new process is quite capable of injecting code into any processes the user has started though (with zero user interaction). If the user happens to already have a high integrity process running on the desktop as a full administrator, the new process could be used to attack it as the process would be running at the same integrity level and it would additionally be running as the same user. This means that in the default configuration, this issue can be used to escape from Protected Mode, but one is still not given full-blown administrative access to the system. However, any location in the user profile directory could be written to. This effectively eliminates the security benefit of Protected Mode for a non-elevated administrator (with respect to treating the user as a plain user).

Source code to a simple program to demonstrate the appinfo service issue is included with the article. The problem is at this point expected to be fixed by Windows Vista Service Pack 1 and Windows Server 2008 RTM. The sample code launches osk.exe with ShellExecute, patches out the CloseHandle calls in ShellExecute to retain the process and thread handles, and then injects a thread into osk.exe that launches cmd.exe. The sample program also includes a facility to create a low integrity process to verify correct function; the intended use is to launch a low integrity command shell, verify that directories such as the user profile directory cannot be written to, and then use the sample program from the low integrity process to launch a medium integrity cmd.exe instance without user interaction, which does indeed have free reign of the user profile directory. The same code will operate in the context of Internet Explorer in Protected Mode, although in the interest of keeping the example clear and concise, the author has not included code to inject the sample program in some form into Internet Explorer (which would simulate an attack on the browser).

Note that while the uiaccess process is launched as a high integrity process, it is configured such that unless a token is explicitly provided that requests high integrity, new child processes of the uiaccess process will launch as medium integrity processes. It is possible to work around this issue and retain high integrity with the use of `CreateProcessAsUser` by code injected into the uiaccess process if desired. However, as described above, simply retaining high integrity does not provide administrative access on its own. If there are no other high integrity processes running as the current user on the current desktop, running as high integrity and running as medium integrity with the non-elevated token are functionally equivalent, for all intents and purposes.

5 Conclusion

UAC, Internet Explorer Protected Mode, and the integrity level model represent an entirely new way of thinking about security in the Windows world. Traditionally, Windows security has been a user-based model, where all processes that execute as a user were considered equally trusted. Windows Vista and Windows Server 2008 are the first steps towards changing this model to support the concept of a untrusted *process* (as opposed to an untrusted *user*). While this has the potential to significantly benefit end user security, as is the case with Internet Explorer Protected Mode, there are bound to be bumps along the way. Writing an integrity level broker process is difficult. It is very easy to make simple mistakes that compromise the security of the integrity level mechanism, as the appinfo issue highlights. The author would like to think that by shedding light on this type of programming error, future issues of a similar vein may be prevented before they reach end users.