

# A Black-Box Tracing Technique to Identify Causes of Least-Privilege Incompatibilities

Shuo Chen<sup>‡</sup>, John Dunagan<sup>†</sup>, Chad Verbowski<sup>†</sup>, and Yi-Min Wang<sup>†</sup>

<sup>‡</sup> University of Illinois at Urbana-Champaign  
shuochen@uiuc.edu

<sup>†</sup> Microsoft Research  
{jdunagan, chadv, ymwang}@microsoft.com

**Abstract:** Most Windows users run all the time with Administrator privileges, equivalent to root privileges on a UNIX system. The possession of Administrator privileges by every user significantly increases the vulnerability of Windows systems. For example, simply compromising a user network service, such as an instant messaging client, provides an attacker complete control of the system. We address this problem by making it easier to develop applications that do not require Administrator privileges, thereby decreasing the inconvenience of running without Administrator privileges. To this end, we present a novel tracing technique for identifying the reasons applications require Administrator privileges (which we refer to as *least-privilege incompatibilities*). Our evaluation on a number of real-world applications shows that our tracing technique significantly helps developers fix least-privilege incompatibilities and can also help system administrators mitigate the impact of least-privilege incompatibilities in the near term through local system policy changes.

## 1. Introduction

The principle of least-privilege is well-accepted within software security circles as a method for reducing the attack surface of running systems. To put this principle simply, software should run only with the privileges necessary to accomplish the task at hand. This principle is both very simple and very appealing; much research has sought to develop techniques for decreasing the amount of privileged code on the system [C93, P03].

Conformance to the least-privilege principle on Windows systems is frighteningly low. Most users run all the time as members of the Administrators group, a practice commonly referred to as “being an Admin” or “having Admin privileges.” This increases the severity of security threats faced by Windows users, because the compromise of any user application becomes a system compromise. This threat is both acute and widespread; attacks against user level network services are common, and include spyware [S04, W04], self-propagating email [C04c], web browser exploits [C04a, C04b], and instant messaging (IM) client exploits [C02].

Our own investigation documented that the need to run applications with Admin privileges is common and includes: children that want to play *Bob the Builder*; anyone desiring to file their taxes with *TurboTax*; corporate employees that desire to connect to their corporate network using *Remote Access Service*; and developers that use *Razzle* to setup their build environment. We also found a Microsoft Knowledge Base article listing 188 such least-privilege incompatible applications [MSKB].

Least-privilege incompatibilities increase the attack surface of Windows systems directly by requiring that individual applications run with Admin privileges; any compromise of such an application is a system compromise. Least-privilege incompatibilities also increase the attack surface indirectly, by causing many users to run as Admin all the time, for at least two reasons. First, least-privilege incompatible applications often fail with misleading error messages, so non-Admin users spend significantly more time troubleshooting. Second, the number of least-privilege incompatible applications is sufficiently great that starting each one from a separate account with Admin privileges, or setting up scripts to do this semi-automatically, is a significant inconvenience.

In this paper, we describe a black-box tracing technique to identify the reasons for these *least privilege incompatibilities*, making it easier to fix them. Our tracing technique implements logging and noise filtering by modifying the Windows OS security subsystem (described in Section 3). We believe this technique could be applied to other operating systems, but we have not investigated this. The use of tracing implies a standard set of tradeoffs (particularly as compared to static analysis), and we discuss how these tradeoffs relate to our system in more detail in Section 5. Importantly for our research, not requiring source code meant we were able to validate our tracer on third party applications for which we only have binaries.

Identifying the causes of least privilege incompatibilities enables two important scenarios:

- **Developers understand least privilege incompatibilities faster.** Our evaluation (described in Section 4) suggests that this significantly reduces the total amount of time required to fix least privilege incompatibilities. Several realities of modern software development make understanding least-privilege incompatibilities difficult. In large industrial software projects, developers must often modify code written by others, and simply identifying the line in the code that causes the failing security check can be a significant help. Additionally, software libraries often encapsulate the system calls being made within a class, hiding the details of the failed security check (and sometimes even that it is a security check failure), and thus making the failure more opaque. Lastly, legacy APIs may expose methods that make security assumptions which are no longer valid. Understanding the invalid assumption can significantly help in understanding how to redesign the application.

- **System administrators can mitigate the impact of some least privilege incompatibilities through local system policy changes.** If a system administrator configures the system (e.g., by modifying the Access Control Lists (ACLs) of registry keys and files) so that the small number of logged security check failures are overcome, the applications in question will no longer require Admin privileges. Making ACL changes so that applications can run with reduced privilege is already a common practice, though it does require careful reasoning about the system-wide effect of such changes [O04] – some ACL changes may be worse than granting certain users Admin privileges. Our contribution is to enable faster identification of the relevant ACLs. We hope that this solution can reduce the need for legacy applications to run with Admin privileges.

Our evaluation (Section 4) on 8 real-world examples demonstrates the completeness, accuracy, and usefulness of our technique. To evaluate completeness, we showed that bypassing the small number of checks in our log was sufficient for the application to run without Admin privileges. To evaluate accuracy, we showed that few logged failures are unrelated to least-privilege incompatibilities. To evaluate usefulness, we first showed that the number of security checks responsible for least-privilege incompatibilities is small. We then contacted developers knowledgeable about the design of the application, and we found general agreement that identifying the reasons for these least-privilege incompatibilities was a significant help in fixing the incompatibilities.

In summary, the main contributions of this paper are:

- A black-box tracing technique for identifying security checks that could result in least-privilege incompatibilities. On exercised code paths, this technique has perfect completeness (no checks are missed; no false negatives), and acceptable soundness (few checks are logged that do not result in least-privilege incompatibilities; some false positives). Our evaluation suggests that this technique effectively limits the number of logged security checks to a human-manageable size.
- A black-box verification technique to verify that the logged checks indeed record every source of least-privilege incompatibility.

## 2. Background on the Windows Security Model

The Windows security model provides several abstractions and mechanisms, which we

describe by comparison to the UNIX security model. A Windows *token* represents the security context of a user. Tokens are inherited by processes created by the user. A token contains multiple Security IDs (*SIDs*), one expressing the user's identity, and the rest for groups that the user belongs to, such as the Administrators group, or the Backup Operators group. UNIX similarly attaches both a user ID and a set of group IDs to a process. In order to implement the *setuid* mechanism, UNIX adds another two user IDs, so that at any point there is a real user ID, an effective user ID, and a saved user ID [C02a].

Windows does not support the notion of a *setuid* bit, and Windows developers typically follow a different convention in implementing privileged functionality. For example, in UNIX, *sendmail* was historically installed with the *setuid* bit so that an unprivileged user could invoke it, and the process could then read and write to the mail spool, a protected OS file. In Windows, a developer would typically write *sendmail* as a service (equivalent to a UNIX daemon), and a user would interact with *sendmail* using Local Procedure Call (LPC). One would implement the *sendmail* command-line interface as a simple executable that sends the command line arguments to the service via LPC. The Windows service model allows services to be started on demand, so dormant services occupy no memory, just as in the UNIX *sendmail* case.

A Windows token also contains a set of *privileges* (which can be enabled or disabled), such as the SystemTime or Shutdown privilege. These two privileges grant the abilities, respectively, to change the system clock and to shutdown the system. Conceptually, privileges are used to grant abilities that do not apply to a particular object, while accesses to individual objects are regulated using Access Control Lists (*ACLs*). In contrast, UNIX typically uses groups to implement named privileges. For example, membership in the floppy group grants access to the floppy drive. To create a SystemTime privilege in UNIX, one might create a SystemTime group, create a ChangeSystemTime *setuid* executable, set its group to SystemTime, and give it group-execute permission.

Windows and UNIX both support *ACLs*, but again, their implementations are slightly different. UNIX file systems typically associate each file with an owner and a group, and store access rights for the owner, members of the group, and all others. Windows *ACLs* can contain many <SID, access> pairs, as in AFS (the Andrew File System). These <SID, access> pairs are used to grant one user the ability to read and write the object, another user the ability only to read the object, all members of another group the ability to read the object, etc. *ACLs* in Windows can be attached not only to files, but to any object accessible through a handle, such as registry entries and semaphores. In UNIX, and more so in Plan 9, access control is made uniform across resources by exporting most resources through the file system (e.g., /dev/audio).

## 2.1 Security Checking Functions

One of the difficulties we faced was deriving a small but complete set of functions to instrument for our security check tracing technique. Windows consists of a large amount of source code, and understanding all paths through the security subsystem proved challenging. We addressed this through a combination of reading the source code and setting breakpoints at observed application failures, followed by working back through the kernel call stack to determine the functions involved. Based on our evaluation, and later discussions with a senior Windows architect, we believe we were successful.

The five functions we identified, and their role in the security subsystem, are presented in Figure 1 – the functions themselves are circled, and the arrows denote function inputs and outputs. For the purpose of discussion, we have changed the function names to make them more intelligible. *Privilege-Check* is used to check that privileges are held and enabled in the token. *Adjust-Privilege* is used to enable or disable privileges. *Access-Check* is used to check whether a user has access to a particular object, as determined by its *ACL*. *Reference-Object* also performs access checks; requests to write or read from an object flow through this function, which checks

the *Handle Table* to see whether the ability to perform the operation was previously granted by *Access-Check* when the handle to the object was created.

*SID-Compare* is used both internally by the security subsystem and directly by applications. In particular, least-privilege incompatible applications often use *SID-Compare* to fail early. The application checks if the user holds a SID granting membership in the Administrators group, and fails if not. Intercepting this direct application check was necessary for us to determine the later (and more interesting) set of checks causing least-privilege incompatibilities. Of course, a developer attempting to fix a least-privilege incompatible application would find removing this *SID-Compare* check to be an obvious modification.

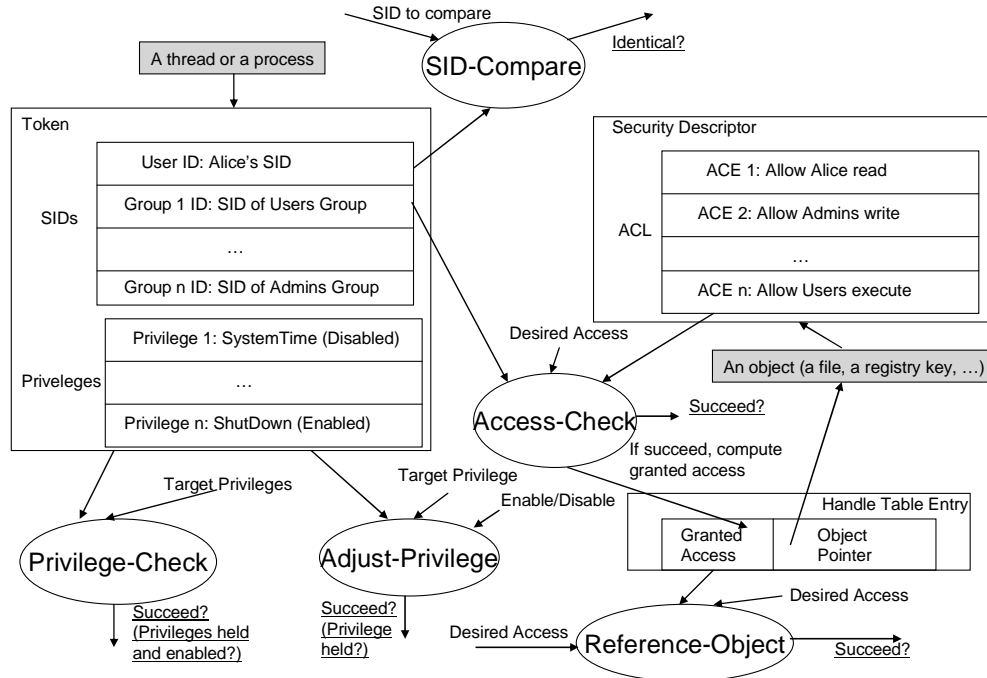


Figure 1: Windows Security Checking Functions

### 3. Identifying Least Privilege Incompatibilities

We have implemented our security check tracing technique as a modified Windows XP Service Pack 1 kernel. To use it, a developer or system administrator starts the tracer, runs the least-privilege incompatible application, and then stops the tracer. The Security Check Monitor and Noise Filter component applies a conservative noise filtering algorithm to log only security checks that might be responsible for least privilege incompatibilities. The actual logging of these checks is done by a separate component, the Security Check Event Logger. The logger records the security check in a log file, as well as some additional information obtained using stack walking. The resulting log contains all the checks that might be responsible for the least privilege incompatibility. To verify the log, we have also implemented a log verification technique; we postpone the discussion of log verification to Section 3.3. Figure 2 shows this workflow.

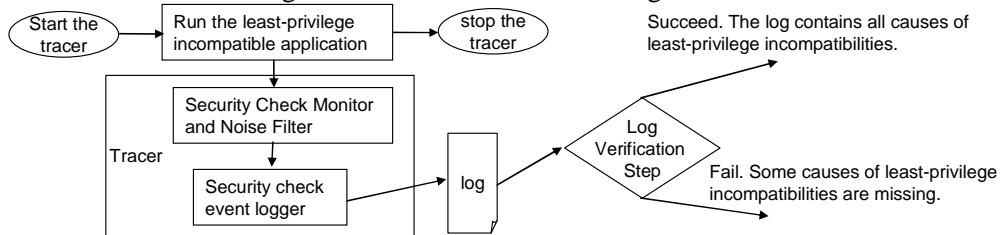


Figure 2: Workflow of the Tracing and Verification Techniques

### 3.1 Security Check Monitoring and Noise Filtering

Failed security checks are very common on Windows systems. Perhaps surprisingly, they rarely have any observable end user impact. We speculate that these failed checks do not result in application failures because the applications and libraries responsible for these failed checks are designed to handle several combinations of security settings, and they do so by attempting to acquire objects with as many rights as possible, falling back to acquiring the objects with fewer rights, and only failing if later calls require rights they were unable to grab in the first place. Regardless of the reason, this reality motivated the development of a noise filtering strategy specific to our goal, identifying least-privilege incompatibilities.

Our noise filtering algorithm assumes the user is running the application with Admin privileges. In the security subsystem, we intercept all security checks, and initially allow the check to pass through unmodified. If the check is successful, and the token contained membership in the Administrators group, the noise filter temporarily removes this membership from the token, and performs a second check. If this second check fails, the Security Check Event Logger is called. Although this algorithm only differentiates between membership and non-membership in the Administrators group, it would be straightforward to apply the same technique on other groups (e.g., the Backup Operators group). To evaluate the likely success of this noise filtering strategy, we performed a quick study, collecting three 2-hour traces during regular office hours on one of our primary machines. The results of these traces are summarized in Table 1.

**Table 1: Two-Hour Traces of Security Checks**

	Security checks	Security checks with user token	Failures when user is Admin	Failures when user is non-admin	Difference
Trace 1	1,756,000	417,257	79,317	81,597	2,280
Trace 2	1,124,000	315,014	64,336	66,385	2,049
Trace 3	913,000	422,783	94,453	97,170	2,717

In each of these traces, the set of security checks that would be logged after applying our noise filtering algorithm (the column labeled “Difference”) is much smaller than the total number of failed checks. The 2K-3K remaining failed checks still constitute a conservative superset of the checks corresponding to least-privilege incompatibilities. Though 2K-3K checks is probably too many to examine by hand, in practice we expect the tracer to be run in much shorter intervals – identifying the least-privilege incompatibilities described in Section 4 required trace lengths of less than 20 seconds. Manually inspecting the logs also yielded two other unsurprising observations. First, security checks tend to occur in bursts right after new processes are started. Second, the potential causes of least-privilege incompatibilities appear to cover the entire range of security checks: access check failures on semaphores and registry keys, privilege check failures, and many others.

The noise filtering algorithm we have implemented depends on the fact that the underlying Windows security subsystem is stateless. This might have been a difficult invariant to maintain if we had attempted to implement the noise filtering algorithm at another layer. For example, if the monitor and the noise filter were built on the system call level, then handling a *File-Open* call would have required closing the file, attempting to reopen it with a different set of permissions, and doing the appropriate fixup. Appropriately handling calls to arbitrary objects would have been even more challenging (or impossible). Thus, this noise filtering algorithm strongly argued for monitoring security checks at the lowest possible level.

**Intricacies of Access-Check Called with MAXIMUM\_ALLOWED Access.** Implementing our noise filtering algorithm for the function Reference-Object required some additional complexity to handle a particular usage pattern, opening an object with MAXIMUM\_ALLOWED access. This parameter is only used by the Windows object management subsystem, but this subsystem uses it quite frequently. In such situations, rather than

determining whether a specific set of accesses are granted, Access-Check computes the maximum set of accesses allowed by a given token and a given security descriptor, and stores it the Handle Table. When operations are later attempted on the object, this cached set of accesses is used to decide whether the operation should be allowed. Though the maximum set of accesses may be quite different for an Admin and a non-Admin, the difference is unimportant unless a later call to Reference-Object makes use of accesses that are only granted to an Admin.

This complicated situation occurs frequently. For example, when we started the TurboTax application (one of the examples in our evaluation), Access-Check was called 303 times with MAXIMUM\_ALLOWED access, among which 189 calls would have granted different access if the token had been a regular user's token. Therefore, treating each such access check as a potential cause of least-privilege incompatibility would quickly have led to an unacceptable amount of noise.

To implement our noise filtering algorithm in this case, we added an extra field to the Handle Table data structure, AssumedGrantedAccess. This field records the access that would have been granted if the token had not contained the Administrators group SID. This allowed the decision to log the security check to be appropriately made when Reference-Object was called.

### 3.2 Security Check Event Logger

The Security Check Event Logger needs the capability to log kernel events. We implemented this by modifying ETW (Event Tracing for Windows), a kernel component that already allows logging events such as registry accesses, page faults and disk I/O. The focus of our work has not been performance, but we did not notice any overhead from enabling tracing. Each security check log entry indicates the current process name, the monitored security checking function, target privileges, the desired access and granted access, a stack dump (the return addresses on the kernel stack), and the object name.

Obtaining the object name of each Access-Check call is more difficult than obtaining other information. Access-Check is performed on a security descriptor and a token. There is no backward pointer from the security descriptor to the object, and indeed, a security descriptor can be created by a programmer without reference to an object, though this practice is rare. To obtain object names when they exist, the logger walks back along the kernel stack, traversing frame pointer frames. The traversal stops at any function frame that is known to contain object name information, which is then written to the log. This technique requires a kernel compiled with frame pointers, which is the case for Windows XP Service Pack 1. We have currently implemented retrieving object names from five functions that we know to be particularly common parents of the access checks, such as *Create-File*. This has been sufficient to give us very good coverage. It allowed us to debug all the least-privilege incompatibilities given in the evaluation section, and it returned object names for 98.3% of the access checks in one of our 2-hour traces (8490 out of 8639 checks).

### 3.3 Log Verification

Developing a tracing technique that worked without access to source code allowed us to identify least-privilege incompatibilities in third party applications (3 of the 8 examples in our evaluation). To avoid having to wait for responses from developers familiar with these applications in order to know whether we were successful, we also developed a black-box verification technique that is the inverse of the tracing technique. In the verification step, the user runs the application as a non-Admin, and the previously logged checks are made to succeed where they normally would have failed. The non-Admin user will be able to use the application if and only if the logged checks cover all causes of least-privilege incompatibilities.

To simplify our implementation, we used regedit and explorer to change ACLs so that file and registry checks would succeed, and we only used the Security Check Monitor and Noise Filter component to change the return value on the remaining security checks. If the verification

technique proves to be as useful a development tool as the tracing technique, one could easily extend our implementation to remove the need for manual editing of registry and file system ACLs. Our evaluation in Section 4 shows that the log verification technique worked on all the examples we have studied for all the code paths we managed to exercise.

## 4 Evaluation

We evaluated the effectiveness of our tracing technique on eight least privilege failure scenarios drawn from real applications. These applications include small utility programs, video games, document processing applications, and software development tools, and span the spectrum of users, including pre-school children, teenagers, professionals and home users. The purpose of the evaluation is both to understand the effectiveness of the technique in producing a small set of security checks responsible for the least-privilege incompatibility, and to understand how helpful this would be to a developer seeking to fix the incompatibility, or a system administrator seeking to mitigate it.

For our experiments we installed and traced applications using the built-in “Administrator” account. We found that most least-privilege incompatibilities are encountered as soon as the application starts up, so we took only short tracers. We verified our logs using a separate account that is not a member of the Administrators group. We found that the causes of least-privilege incompatibilities in our evaluation fall into three categories: overly-restrictive ACL settings, insufficient granularity of privilege in the application design, and programmatic enforcement of unnecessary privilege requirements.

### 4.1 Overly Restrictive ACLs

The three applications in this section required elevated privileges because they either stored their settings in a more secure location than necessary, or they did not correctly configure ACLs to allow access by the appropriate users. These problems are all fixable with small code changes, and it appears to be possible to work around them by manually reconfiguring the relevant ACLs.

#### 4.1.1 Bob The Builder Game

*Bob The Builder, Can We Fix It* is a video game designed for children as young as 3. If a regular user starts the game, the error message shown in Appendix A.1 appears. To identify the least-privilege incompatibilities, we started the tracer, launched the game, and then stopped the tracer. The tracer intercepted 4002 checks, of which 899 would have failed if the user was not an Admin. Only 15 checks survived noise filtering. The 5 unique entries among these 15 checks are shown in Table 2.

**Table 2: Unique Log Entries for Bob The Builder**

Checking Function	Process Image	Object Name
Reference-Object	Automenu.exe	\REGISTRY\HKEY_LOCAL_MACHINE\SOFTWARE\BBC Multimedia\Bob the Builder\1.0.0
Access-Check	explorer.exe	\Program Files\THQ\Bob the Builder\StartBTB.exe
Access-Check	explorer.exe	\WINDOWS\explorer.exe
Access-Check	explorer.exe	\WINDOWS\system32\mydocs.dll
Access-Check	explorer.exe	\WINDOWS\system32\shell32.dll

Two things point to the first entry in the log being the likely cause of the least-privilege incompatibility. First, the error message (Appendix A.1) mentions the *AutoMenu* process. Second, HKEY\_LOCAL\_MACHINE (HKLM) is a portion of the Registry used for storing machine-wide settings. We used our verification technique to confirm this hypothesis. Although we have not heard from the application developers directly, anecdotal evidence points to this being a common mistake leading to least-privilege incompatibility, easily fixed by using a per-user store. We have not yet deduced why explorer also generates entries in the log.

### 4.1.2 RAZZLE

Several Microsoft products use the *razzle* build environment configuration tool. Developers must have Admin privileges to use the current version of this tool. The figure in Appendix A.6 shows the error messages when user *t-shuoc* started razzle and then attempted to change directory to *c:\sysman*, the root directory of our project source code.

Tracing razzle from start to finish yielded 7 log entries (shown in Table 3) out of 8660 security checks. Our first hypothesis was that the ACL on *c:\sysman* was responsible for the least-privilege incompatibility, and we change the ACL manually. However, when we ran razzle a second time, the ACLs reverted to requiring Admin privileges. Our second hypothesis was that *razacl* was changing the ACLs. We confirmed this by changing the ACL manually and removing the *razacl* executable; this allowed a non-Admin to use Razzle. We learned from consulting razzle developers that *razacl* removes user accounts from ACLs in the build tree to produce a consistent build environment across user accounts. The next version of razzle was already slated to address this by configuring the ACLs consistently, yet such that non-Admins can use the tool.

**Table 3: Log Entries for RAZZLE**

Checking Function	Process Image	Object Name or Checked Privilege/Membership
Access-Check	explorer.exe	\WINDOWS\system32\cmd.exe
Adjust-Privilege	razacl.exe	Enable SECURITY privilege
Privilege-Check	razacl.exe	Check if SECURITY privilege is enabled
Access-Check	cmd.exe	\sysman
Access-Check	findstr.exe	\sysman
Access-Check	perl.exe	\sysman
SID-Compare	tfindcer.exe	To determine if the current user is an admin

### 4.1.3 Microsoft Greetings 2001

Microsoft Greetings 2001 is a document processing application. Our 12-second trace of Microsoft Greetings' startup recorded 37 potential causes of least-privilege incompatibilities, (summarized in Table 4), out of the 12,618 total security checks.

**Table 4: Log Entries for Microsoft Greetings 2001**

One entry for SID-Compare to determine if the current user is an admin
One entry for access-check on the file <i>\Program Files\Microsoft Picture It! PhotoPub\pidocob.dll</i>
3 entries for access-check on the following registry key and its subkeys <i>\REGISTRY\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Picture It!</i>
22 entries for access-check on the following registry key and its subkeys <i>\REGISTRY\HKEY_LOCAL_MACHINE\SOFTWARE\Classes</i>
11 other entries

In verifying the logs, we found that the 1<sup>st</sup> three classes of logged security checks must succeed for the application to be usable by a non-Admin. We encountered a series of error messages (shown in Appendix A.4) as we began to force these checks to succeed. We feel that the increasing obscurity of the error messages is not surprising given the application developer's decision to add an early check for membership in the Admins group (the 1<sup>st</sup> entry in the log).

Despite the many least privilege incompatibilities, our technique required only one trace to identify all of them. We found that a later version of this software, *Microsoft Picture It! 2002*, does not have any least-privilege incompatibilities, and therefore we assume that these issues have been addressed.

## 4.2 Insufficient Privilege Granularity in Application Design

The three applications in this section all have some functionality that is appropriate for regular users, and some functionality that should only be usable by Admins. However, they all fail to accommodate both modes of operation in their design, and consequently are not usable at all by regular users. We thank [B01] for bringing many of these examples to our attention.

#### 4.2.1 Remote Access Service (RAS)

RAS is a program for Microsoft employees to remotely connect to the corporate network. Running RAS as a non-Admin leads to an error message (shown in Appendix A.3) roughly one minute after the program starts. Reproducing this problem was technically challenging because our lab did not allow us to easily fake the remote environment that RAS assumes. Our workaround was to trace a small script [MSDN] that replicated the core RAS behavior, and then to verify the results with the real RAS program from a remote location.

Tracing the small script generated 7 log entries out of 2566 security checks. Six of the seven checks were related to files, registry keys and TCP/IP devices that we eliminated as causes of least-privilege incompatibilities using our verification technique. Causing just the last check to succeed allowed both the script and the real RAS program to be run by a non-Admin.

To analyze this one security check in more detail, we set a kernel breakpoint on the check, ran the script, and dumped the call stack when the script hit the breakpoint. This showed the Windows script interpreter *wscript.exe* attempting to enumerate all network connections by calling the function *get\_EnumEveryConnection* in class *CNetSharingManager* defined in *HNETCFG.dll* (Home Networking Configuration Manager). This function checks that the user is an Admin using the function *CheckTokenMembership* exported by *ADVAPI32.dll*, which internally calls into the kernel function *Access-Check*.

From discussions with the developers of this tool we learned that the RAS security model is to switch every network connection to run over the newly created Virtual Private Network (VPN) connection. To accomplish this they were forced to use the only known API for enumerating all connections on the system, and the API was designed to allow only Admins to call it. One way to address this problem in the future is to provide a service that can do this work on behalf of RAS.

This example illustrates that least-privilege incompatibilities may occur as a result of non-intuitive security checks made as a result of a library that indirectly checks access in its lower level implementation. Our tracing and verification techniques allowed us to quickly identify the failing security check without using the source code. However, because we had access to the application source code in this case, we were additionally able to construct the entire call sequence and identify the specific code responsible for the critical security check failure.

#### 4.2.2 Windows Power Configuration

Windows power options are configured per user and stored in the user's profile. However, only Admins have sufficient privileges to change power options, and the application only allows them to change their own. When a non-Admin attempts to change their power options, they receive an error message showing access denied (shown in Appendix A.5). Tracing this action led to 5 logged checks out of 1364 total. Two of these logged checks were for *HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Controls Folder\PowerCfg*, and we verified that this one ACL was the cause of the least-privilege incompatibility.

From discussing this with internal Microsoft developers we have surmised that it is difficult to identify what the correct configuration should be in scenarios with settings that affect the entire machine, and when there can be multiple simultaneous users of a machine. Conflicting power options, such as time to turn off hard disks, may interfere with applications running for other logged in users. Our recommended solution to this issue is opposite to that of what Bob The Builder and RAS need to do – that is, make the power options a system-wide configuration rather than per user.

#### 4.2.3 Windows Clock/Calendar

Double clicking the numeric clock on the right-bottom corner of the Windows desktop presents a pictorial clock and a calendar. Users find this a handy tool to use when they want to answer questions like “what is the date of the last Monday of May?” Unfortunately a regular user attempting to launch the clock is presented with an error message (shown in Appendix A.2)

indicating insufficient privilege. This application may even be difficult for a regular user to launch under a separate privileged account, because the command line is not readily available. We traced this application for 2 seconds, and logged 3 checks (shown in Table 5) out of 455 total.

**Table 5: Log Entries for Windows Clock/Calendar**

Checking Function	Process Image	Object Name or Checked Privilege
Access-Check	explorer.exe	\WINDOWS\system32\rundll32.exe
Access-Check	rundll32.exe	\BaseNamedObjects\shell.{A48F1A32-A340-11D1-BC6B-00A0C90312E1}
Adjust-Privilege	rundll32.exe	Enable SystemTime privilege

We verified that the system time privilege check is the cause of the least-privilege incompatibility. From discussing this case with internal Microsoft developers, we surmised that the original clock was not designed to be used in a read-only manner, but that this privilege check would provide a good place to branch, displaying a read-only UI if the privilege was missing.

### 4.3 Programmatic Enforcement of Unnecessary Privilege Requirements

The two applications in this section programmatically enforce that the user is an Admin, but they appear to function perfectly well if this check is bypassed. We discuss the reasons for these requirements in more detail in the context of each application.

#### 4.3.1 Diablo II Game – Failure opening CDROM device.

Diablo II is an action game that ships on three CDs: an install disc, a cinematics disc, and a play disc that must be in the drive for the game to work. When a non-Admin attempts to play the game, a misleading error message (Figure 3) pops up claiming the CD drive is empty. Tracing this action took 10 seconds, and generated 3 log entries out of 1573 total checks, 440 of which fail for a non-Admin. The 3 log entries are shown in Table 6.



**Figure 3: Error Message When a Regular User Starts Diablo II Game**

Because the error message mentions the CD-ROM drive, we hypothesized that the third log entry was responsible for the least-privilege incompatibility. We verified that passing this check alone allowed a non-Admin to play the game.

**Table 6: Log Entries for Diablo II**

Checking Function	Process Image	Object Name
Access-Check	explorer.exe	\Program Files\Diablo II\Diablo II.exe
Access-Check	Game.exe	\REGISTRY\MACHINE\SYSTEM\ControlSet001\Control\MediaProperties\PrivateProperties\Joystick\Winmm
Access-Check	Game.exe	\Device\CdRom0

This example illustrates how least-privilege incompatibilities can manifest as errors unrelated to user privileges. We have not received any response from the Diablo II developers, but the misleading error message leads us to believe that the failure mode was not anticipated by the developers. We speculate that this may be a simple programming oversight where the program attempts to acquire certain unnecessary CD-ROM accesses that Admins are granted by default.

#### 4.3.2 TurboTax 2003

TurboTax is tax calculation software released by Intuit. Running TurboTax as a non-Admin generates the error message in Appendix A.7. Tracing the application startup generated 11 log entries out of 12503 total security checks. The 11 logged entries break down to one entry for SID-

Compare, three for access checking semaphores, four for access checking HKLM registry keys and three others. Our verification seems to suggest that just causing the SID-Compare call to succeed is sufficient for a non-Admin to use the application extensively; we succeeded in running TurboTax, completing a 1040A tax form and printing it to a PDF file. We have not yet received a response from the developers of TurboTax, but we have two different reasons that we believe might have caused the TurboTax developers to insert this check. First, a publicly available transcript of a discussion with an Intuit customer service representative suggests that requiring Admin privileges was a quick fix solution to data privacy concerns. Because Admins already have complete control of the system, leaking information about other users through the application does not represent an increased exposure of private data if the user viewing the information is already an Admin [A01]. In contrast to the Intuit customer representative, we do not believe this is a reasonable design decision. The second reason we considered is that some code path we did not execute generates a failure when the user is a non-Admin. On balance, we believe the evidence points to the Admin check being unnecessary.

## 5. Related Work

A common approach to increasing system security is to sandbox applications or users, so that the scope of individual compromises is decreased. Common sandboxing techniques include virtual machines [G03], system call interposition [A00, G03a, G04], and restricted file systems [C00]. Our work differs from this prior art in that we are not inventing a new sandbox or developing a new technology to better implement an existing sandbox. Instead, our tracing technique is designed to help developers and system administrators make use of an existing and well-understood sandbox: the unprivileged user.

Other previous work has investigated technologies for building or re-building systems so that they better conform to the principle of least privilege [E, P03a, V]. Provos et al show how separating OpenSSH into privileged and unprivileged parts (privilege separation) would have reduced its vulnerability to several security holes that were later discovered [P03]. Brumley and Song describe the *Privtrans* tool which significantly automates this process using static analysis and annotations on privileged operations [B04]. Our technique is complementary to Privtrans, and our tracing technique could potentially be used to automatically produce the annotations required by Privtrans.

A common assumption in much of this earlier work is that some part of the program under investigation (e.g., OpenSSH) legitimately requires the ability to perform a privileged operation. In contrast, our investigation into Windows applications suggests that in many cases, the requirement that the application run in a privileged context is a trivial bug. In other cases, the requirement that the application run in a privileged context reflects a larger design flaw. There was only an argument for the application requiring Admin privileges in two of the eight cases we evaluated, RAS and Power Config.

We now consider previous work that has used static analysis [A02, Y03], a commonly cited alternative to dynamic tracing techniques such as our own. A commonly cited strength of static analysis is that it can achieve code-coverage trivially, while dynamic techniques often require sophisticated test-case generation strategies to exercise all code paths, if exercising all code paths is possible at all. In our context, the stronger completeness guarantees possible from static analysis are unnecessary because our goal is only to reduce, not eliminate, the effort required to fix least-privilege incompatibilities. Also, we believe static analysis would be difficult or impossible to apply in our context for three reasons. First, the underlying property being checked is a function of all ACLs on the system, and these ACLs are meant to be configurable. Second, our investigation shows that privilege failures commonly occur after the flow of control passes through a third party library, and static analysis is known to be difficult in the absence of source code. Lastly, because static analysis typically requires source code, and sometimes also annotations, this rules out analyzing many legacy and third party applications.

The most closely related previous work is the current developer practice of identifying privilege failures by tracing the file system or registry and grepping for ACCESS\_DENIED [G01]. Our technique goes beyond this by monitoring a complete set of functions within the Windows security subsystem and implementing a more sophisticated noise filtering strategy. Our evaluation in Section 4 justifies the importance of both of these advances for identifying least-privilege incompatibilities. On UNIX systems, system call tracing is sometimes similarly used to debug access failures. Our tracing technique differs from system call tracing in its more sophisticated noise filtering, and the significantly smaller code base that must be correctly understood in order to correctly capture all access failures. System call tracing must monitor all functions that have security implications and are exposed by the OS API, while we only need to monitor 5 functions in the Windows security subsystem.

## 6. Future Directions

Although we believe our techniques are already useful, we can think of at least three features that would make them more useful. First, we have currently implemented our tracer as a modified Windows XP Service Pack 1 kernel, but a driver version would be more easily deployed. Second, we would like to integrate the tracer with existing debugging technologies, such as the ability to break into a debugging environment when a certain security check is reached. Third, we believe that our verification technique highlights the need for a utility to configure security permissions not associated with files or registry keys. Even if most system administrators were not comfortable reasoning about the implications of such changes, this would be a convenient utility for developers seeking to debug least privilege incompatibilities.

## 7. Conclusion

Application least-privilege incompatibilities are a severe security concern. These incompatibilities force most Windows users to run with Administrator privileges. This significantly increases the vulnerability of the system, because compromises of user network services become system compromises. This threat is widespread, as a large number of network attacks, ranging from spyware to instant messaging client exploits, target user network services.

To help reduce least-privilege incompatibilities, we introduce a black-box tracing technique that identifies the security checks causing these incompatibilities. Our technique catches all least-privilege incompatibilities on the examined code paths, and implements a noise filtering algorithm that effectively limits the number of logged security checks to a human-manageable size. We also introduce a black-box verification technique to test if the logged checks indeed record every source of least-privilege incompatibility. We evaluated our techniques using eight least-privilege incompatible applications. Based on these evaluations and subsequent discussions with developers, we conclude that these techniques make fixing least-privilege incompatibilities substantially easier.

**Acknowledgements:** We thank our colleagues at Microsoft and Microsoft Research for their assistance and insightful comments, and in particular Doug Beck, Bill Bolosky, Brad Daniels, Jon Howell, Dan Simon, and Rich Ward.

## Bibliography

- [A00] Anurag Acharya and Mandar Raje. MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications. USENIX Security 2000.
- [A02] Ken Ashcraft and Dawson Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. IEEE Security and Privacy 2002.
- [B01] Keith Brown. Keith's Security Hall of Shame.  
<http://www.pluralsight.com/keith/hallofshame/default.htm>
- [B04] David Brumley and Dawn Song. Privtrans: Automatically partitioning Programs for Privilege Separation. USENIX Security 2004.
- [C93] Mark E. Carson. Sendmail without the Superuser. USENIX Security 1993.
- [C00] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle and Virgil Gligor. SubDomain: Parsimonious Server Security. LISA 2000.
- [C02] CERT. AOL Instant Messenger client for Windows contains a buffer overflow while parsing TLV 0x2711 packets. <http://www.kb.cert.org/vuls/id/907819>
- [C02a] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. USENIX Security 2002.
- [C04a] CERT. Critical Vulnerabilities in Microsoft Windows.  
<http://www.us-cert.gov/cas/techalerts/TA04-212A.html>
- [C04b] CERT. Internet Explorer Update to Disable ADODB.Stream ActiveX Control.  
<http://www.us-cert.gov/cas/techalerts/TA04-184A.html>
- [C04c] CERT. Advisory CA-2004-02 Email-borne Viruses.  
<http://www.cert.org/advisories/CA-2004-02.html>
- [E] Chris Evans. Very secure FTP daemon. <http://vsftpd.beasts.org>
- [G01] Dr. GUI. Debugging Permissions Problems. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaskdr/html/askgui03272001.asp>
- [G03] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. SOSP 2003.
- [G03a] Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. NDSS 2003
- [G04] Tal Garfinkel, Ben Pfaff, Mendel Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. NDSS 2004.
- [MSKB] Microsoft. Certain Programs Do Not Work Correctly If You Log On Using a Limited User Account. <http://support.microsoft.com/default.aspx?scid=kb;en-us;307091>
- [O04] Tobias Oetiker. MSI Packaging How-to. <http://isg.ee.ethz.ch/tools/realmen/det/msi.en.html>
- [P03] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. USENIX Security 2003.
- [P03a] Niels Provos. Improving Host Security with System Call Policies. USENIX Security 2003.

[S04] Stefan Saroiu, Steven D. Gribble, and Henry M. Levy. Measurement and Analysis of Spyware in a University Environment. NSDI 2004.

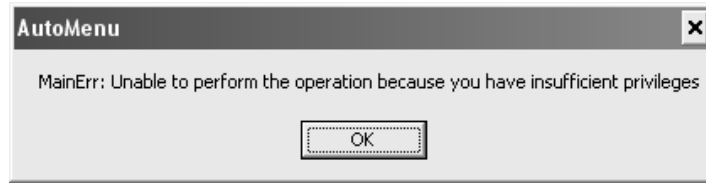
[V] Wietse Venema. Postfix Overview. <http://www.postfix.org/motivation.html>

[W04] Yi-Min Wang, Roussi Roussev, Chad Verbowski, Aaron Johnson, and David Ladd. AskStrider: What has changed on my machine lately? Microsoft Research Technical Report, January 2004. MSR-TR-2004-03. Ro appear in USENIX LISA 2004.

[Y03] Junfeng Yang, Ted Kremenek, Yichen Xie, and Dawson Engler. MECA: an Extensible, Expressive System and Language for Statically Checking Security Properties ACM CCS 2003.

## Appendix: Error Messages of Least-Privilege Incompatible Applications

### A.1. Error Message of Bob The Builder



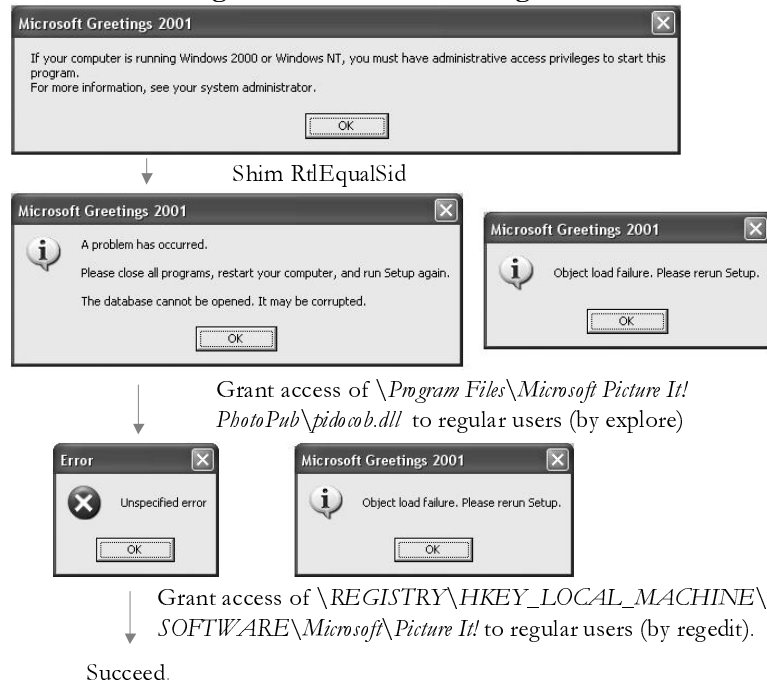
### A.2. Error Message of Windows Clock/Calendar



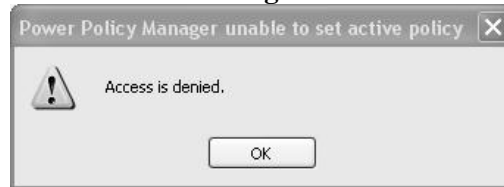
### A.3. Error Message of Remote Access Service (RAS)



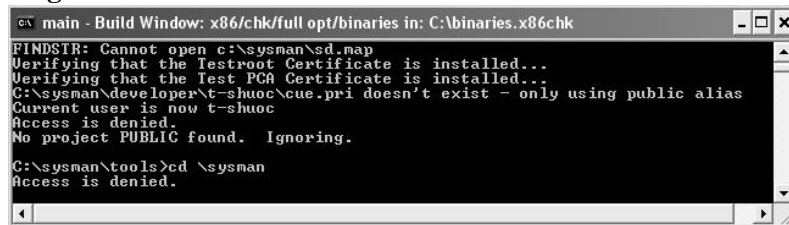
#### A.4. Successive Error Messages of Microsoft Greetings 2001



#### A.5. Error Messages of Windows Power Configuration



#### A.6. Error Messages of RAZZLE



#### A.7. Error Message of TurboTax 2003

