Empirical Privilege Profiling

Carla Marceau and Rob Joyce
ATC-NY
33 Thornwood Dr. Suite 500
Ithaca, NY 14850
+1 607.257.1975

carla@atc-nycorp.com

ABSTRACT

The well-known Principle of Least Privilege states that a program should run with the minimal authority that it requires to get the job done, and no more. However, application of the principle has been left to software developers, developers of installation procedures, and system administrators with few tools to assist them. How much privilege does a given program need? How do you know if you write a program that uses too much privilege or install a program with too little? Empirical privilege profiling provides a partial answer to this question by tracking a program's actual use of resources, which can be used as a guide during program development and installation, as well as for detecting intrusions and providing assurance for mobile code. In this paper, we introduce the concept of dealing with privilege as a measurable quantity, rather than in terms of a "rule of thumb."

1 INTRODUCTION

The Principle of Least Privilege [1] states that a program should run with the minimal authority that it requires to get the job done, and no more. Programs, sites, and organizations that observe this principle help to minimize the amount of damage that can be caused by errors in a program or attacks that subvert it. For thirty years, this principle has served as a guide to program and system developers. However, it is difficult to translate this principle into practice. In general, a program will always have more privilege than it needs: if it has less, it will fail, people will be alerted, and the program privilege will be increased (perhaps more than necessary). By contrast, it typically goes unnoticed when a program has too much privilege. Worse, developers often take shortcuts in order to ease implementation, debugging, and testing, sometimes resulting in poor default installation parameters and hidden backdoors in programs. These can be exploited by either insider or external attackers.

NSPW 2005 Lake Arrowhead CA USA

Many years ago, one of the authors worked on the Multics project at MIT. As code was developed, team members specified access requirements for modules of the operating system, reasoning carefully about what access was needed by whom. When the system was finally tested, it failed repeatedly, because few modules had sufficient privilege to do their work. In order to get the system running, the programmers changed all access control lists to give sweeping access rights to files—including even "execute" permission on data files! Although this occurred before the formulation of the principle of least privilege—and may have contributed to it—it illustrates the difficulty of setting privilege appropriately without tools.

We have begun an effort to create a system that can empirically approximate the least privilege required by a program. We assume that the program is a black box in the sense that we cannot examine and instrument the source code. (Some recent papers [2, 3] distinguish between white-box, gray-box, and black-box based on whether the analysis delves into program source or binary, examines artifacts such as the program stack, or can look only at kernel calls. By "black-box," we simply mean that source code is not available and that analysis focuses on execution, not on program artifacts.) The general approach is to run many instances of the program, typically with multiple users, multiple hosts, and multiple sites, and to record the privileges actually exercised by all these instances of the program. This information is then collected to create an abstract composite *privilege profile* for the program. Any single program instance may well use less privilege than the composite profile; however, the composite profile specifies a reasonable minimum, and quite possibly less privilege than that granted to the program by the default installation.

Empirical privilege profiling is potentially of use in several areas. First, groups of users can profile the privileges that a given application actually exercises when they use it; the profile for a group that uses the application for a limited purpose or in a restricted way could be quite different from the profile of a group of, say, undergraduate students. System administrators could use that information to set up the program's access to resources. Second, program developers could profile the program's use of resources and look for anomalies that indicate poor or excessive use of resources. We have noticed such anomalies in programs that we analyzed. Tracking resources used by a program could provide a rich new data stream for intrusion detection.

A naïve approach to profiling program privilege would be to list the raw privileges exercised by the program, such as "read file `C:\foo\bar\baz`". However, raw privileges used in specific instances of a program are a poor indicator of future use. It is necessary to generalize over a wide range of individual computers, file systems, and sites; raw file names (and other

site-specific privilege information) are not acceptable. To collect *abstract program privileges* it is necessary to correlate privileges exercised by the same program on different computers at different sites and create an abstraction of each privilege.

Further, a naïve approach to profiling program privilege would collect a *set* of privileges exercised by the program. However, a simple set does not provide enough information for correlating privileges across distinct program instances. A program may write to file A and file B, for example, where file A is a file specified by a user, while file B is a file private to the program, whose integrity is crucial to correct operation. Or file A might be a file that is only read by the program, while file B is one that it writes. In correlating privilege exercised by two programs, it is important to correlate the "same" privileges: A_1 with A_2 (instances of A written by programs P_1 and P_2, respectively) and B_1 with B_2 (similar instances of B).

Our approach to privilege correlation is to associate each exercise of privilege with a point in the program. A point in the program that reads a preferences file, for example, will read a preferences file in every instantiation of the program at every site. Therefore, we construct a model of the program's *behavior*; we then associate each exercise of privilege with a program point in the model. The intuition is that the set of program points that access a given resource captures the semantics of the resource *from the point of view of the program.*

To investigate the feasibility of program privilege profiling, we have instrumented Windows programs to collect data about their use of resources, implemented our approach in a simple test bed, and conducted an experiment to create an abstract profile based on several instances of Microsoft Notepad. In this paper, we report on the results of this preliminary investigation of empirical privilege profiling.

In the next section of this paper, we present a model of program privilege, with particular application to Microsoft Windows. Section 3 describes an experiment we performed to validate our concept of privilege abstraction. In Section 4, we describe how the privacy of collaborators can be protected and how the system can protect itself from malicious inputs. We close with a discussion of the results obtained to date.

2 A MODEL OF PROGRAM PRIVILEGE

A *privilege profile* for a program consists of a set of *privileges* that the program must have in order to function properly. Intuitively, a *privilege* is a *resource* together with a set of access rights on that resource.

An individual exercise of privilege is a pair <resource, {access right}>. For example, the right to read file Foo in Windows XP may be represented as `<Foo,{FILE_READ_ACCESS}>`. The heart of the privilege profile is a set of privileges.

2.1 Abstract privileges

We distinguish between *abstract* and *concrete privileges*; more precisely, the distinction is between abstract and concrete *resources*. A concrete resource is a file, registry key, or other resource that the program accesses by a specific name such as a pathname. An abstract privilege is the use of an abstract resource—that is, an abstraction of concrete resources. Intuitively, an example of an abstract resource might be "the log file for the application" or "a new document that the application creates at the request of the user."

In order to create abstractions for concrete privileges used in many instances of an application, we must correlate instances of the "same" concrete privileges. The central hypothesis of this effort is that such correlation is possible using the *program points*—that is, places in the code—at which privilege is exercised. Specifically, we reason as follows:

> A concrete resource that is an instance of a given abstract resource will be accessed at the same program points as other instances of that abstract resource
>
> Therefore, by comparing the program points at which concrete resources are used, we can abstract from those instances to an abstract resource

Figure 1 shows an abstract resource, identified by the program points that access it.

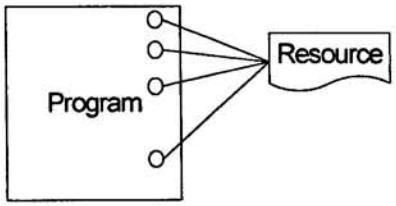

Figure 1. An abstract resource, identified only by the way it is used in the program

Because program points are central to this line of reasoning, a privilege profile also includes the program points at which each (abstract) privilege is exercised.

In the following sections, we further explore the concepts of program points, resources, and the privilege profile. A precise definition of privilege depends on the operating system, which defines the resources of a computing system as well as the applicable set of access rights. In Section 2.5, we describe how these concepts are implemented in Windows.

2.2 Program points

It remains to make precise our intuitive notion of "program point," which should be tied both to the application program to be profiled and to observations of program behavior at run-time (enabling us to peek inside the "black box"). We capture these by defining a program point to be a pair <pc, op> consisting of the application program counter (adjusted for the program's location in memory) and the kernel operation that requires privilege. Intuitively, the application program counter captures *what the program thinks it is doing with the resource.* The kernel operation captures its *use* of the resource and the *privileges* required for that use. By monitoring a process's kernel calls and associating them with the program counter, we can capture program points.

Note that because modern programs in general make extensive use of dynamically loaded libraries (DLLs, also called shared objects in UNIX™), a single application program counter typically causes many kernel operations to be invoked. We have observed dozens of different kernel operations associated with a

single application program counter. Thus, the program counter alone does not adequately characterize a program point.

2.3 Constant and variable resources

Programs access some resources for their own use and others on the user's behalf. A *constant* or *program-specified resource* is one that the program "knows" about, such as a configuration file, a home directory, or a DLL that the program needs to execute. The exact pathname of a configuration file may vary between sites, for example based on the value of an environment variable. Nevertheless, the program typically expects and knows how to find this resource. At one or more points in the program, for example, the program will read, execute, or write a "constant" file resource; when the program accesses the file, it is always from these points. A (resource type-specific) invariant property is associated with a constant resource; for example, every instance of a constant file resource will have the same filename (last component of the pathname). By using additional information, such as environment variables or strings embedded in the executable, we can specify invariant properties more accurately and completely.

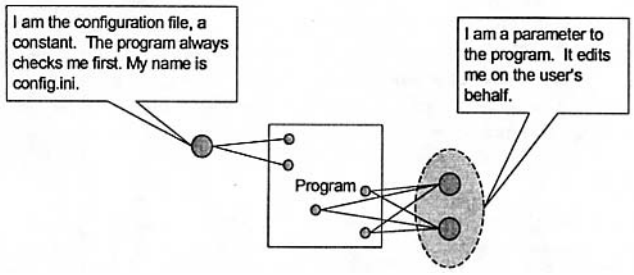

Figure 2. Constant and variable file resources, accessed at various points in the program

By contrast, a *variable* or *parametric resource* is one that the program has no a priori notion of. For example, the user may ask the program to append its output to a given file. In this case, the pathname and content of the file have no meaning to the program; any file supplied by the user will do. We call this a variable resource because it typically takes on different values for different executions of the program. It can even take on multiple values within a single execution. This situation is shown in Figure 2, which depicts the program's use of a configuration file at two program points and of two variable resources, both used in the same way. Other variable resources may be determined by environment variables or the system configuration. For example, a file dialog box may always access the user's home directory.

Distinguishing between constant and variable resources is important in order to accurately portray the program's use of resources. Variable resources are like parameters to a function: a variable can take on many distinct values (concrete resources), even within the same instantiation of the program. If Notepad is used to edit three files, each file corresponds to the same abstract variable resource. Advice to a system administrator about the access required for these resources must be couched in general terms: "Any file edited in Notepad requires ...." By contrast, each constant resource can be named more or less accurately: "The initialization file, which is in the application's home directory and is named '`foo.ini`' requires ...."

The distinction between constant and variable resources can also be exploited for correlating privileges between different executions of a program. A program's initialization file, for example, always has the same filename (local name in the directory) and is always in the "same" place in the file system (where "same" is typically relative to a home directory or other parameter); correlation can exploit these invariant properties, which can typically be found in standard formats in the program executable. By contrast, variable data files have no *a priori* similarity, other than the way in which the program accesses them.

2.4 The privilege profile

An individual exercise of privilege is a pair <resource, {access right}>. For example, the right to read file Foo is represented as `<Foo,{FILE_READ_ACCESS}>`. The heart of the privilege profile is a set of privileges.

In order to facilitate correlation of concrete resources, a privilege profile records privileges together with the program points at which they are exercised. For each resource, the profile lists the program points that access the resource; there may be one or more of these. For each program point, the profile documents the access rights required for the access.

More precisely, let us denote a map (partial function) from range R to domain D by {R => D}. Then the heart of the program privilege profile is a map:

```
{Resource => {ProgramPoint =>
{AccessRight}}}
```

In particular, note the difference between this and two maps

```
{Resource => ProgramPoint}
{ProgramPoint => {AccessRight}}.
```

A single map distinguishes between access rights required on two different resources, even when they are accessed at the same program point. Using two maps does not.

We can construct a profile by first running the program several times on test data to obtain a preliminary profile. The preliminary profile will contain some constants, some variables, and some concrete resources that have not occurred often enough to be categorized. As more data is collected, the EPP profiler incrementally learns more about these resources—and new ones that occur during the collection—and will be able to abstract them as either constant or variable abstract resources. We believe that we can in fact do better than that by using environment variables and strings embedded in the binary executable to differentiate between constant and variable resources.

2.5 Privilege in Windows

Although the general concept of privilege profile is applicable to any platform, the specifics depend on the operating system, which defines the *resources* for which privilege is required and the *access rights* that apply to them. In this research, we have been concerned with the Windows XP operating system. In this section, we briefly describe the Windows security model, which we used in this effort. We believe that a model that is adequate

to capture privilege in Windows will suffice for more perspicuous systems, such as Linux.

The Windows security model is more complicated than that of UNIX. Windows distinguishes between access rights, which apply to *securable objects* and are documented in access control lists, and "privileges," such as `SeDebugPrivilege` (the ability to modify the memory of another process) or `SeBackupPrivilege` (the ability to copy files for backup), which control access to system resources and system-related tasks and are therefore needed to perform *privileged operations*. System privileges are assigned by administrators to user and group accounts. In our work, we create a uniform model of privilege by treating such Windows privileges as access rights on a generic SYSTEM resource. In this paper, we will not be further concerned with such special privileges, but in general they must be considered when creating privilege profiles for programs.

Windows defines access rights for many kinds of resources. Persistent resources include files, directories, registry keys, and system services (such as a service that provides access to the Internet). Ephemeral resources include pipes, threads, inter-process synchronization objects, access tokens (which capture salient information about the process's owner), and many others. We are concerned only with privileges on persistent objects.

Each type of resource is associated with a set of type-specific access rights; for example, FILE_READ_ACCESS is an access right on files and directories. Type-specific rights are meaningful to objects of a specific class; for example, KEY_QUERY_VALUE applies to registry keys and FILE_APPEND_DATA to files and directories. Windows also defines generic access rights (GENERIC_READ, GENERIC_WRITE, and GENERIC_EXECUTE), which apply to all classes of objects and are mapped by the kernel into type-specific access rights for each class of securable object. Users can also define new resource types and new types of access rights on them. The work described in this paper focuses on file and directory resources. However, it applies equally to registry keys and we believe can be extended to all persistent resources.

Each kernel operation requires a certain set of rights to succeed. We instrument calls to the kernel at the interface to `ntdll.dll` [sic], which is the relatively stable and documented interface to the Windows kernel. In some cases, the required set of rights depends on arguments to the operation. For example, the operation to open a file specifies requested rights on the file; if the owner of the process does not enjoy those rights, the operation fails.[1]

In Windows, we define a privilege as *a set of access rights associated with a securable object.* For example, the right to read file Foo can be represented as <Foo,{FILE_READ_ACCESS}>.[2] Access rights for a given securable object are defined by the object's owner in a Discretionary Access Control List (DACL) associated with the object.

3 AN EXPERIMENT IN PRIVILEGE ABSTRACTION

We performed an experiment to test our ideas of privilege abstraction. Three different users each exercised Microsoft Notepad, performing various functions for a short time, for a total of eight "runs."

Some parameters of the experiment are shown in Table 1. Notepad's use of resources was captured and a log of exercises of privilege was prepared for each run. The logs were fed into a correlation engine that attempted to correlate the runs' use of resources by comparing program points. We required three different concrete instances of an abstract resource to categorize the resource as a variable or constant. In addition to program points we used limited additional information for correlation; specifically, we identified DLL files to the correlation procedure as constant resources. In a practical system, invariant properties of all constant resources could be extracted from the binary code of the program, the DLLs it invokes, and possibly from other sources, such as environment variables of the platform. The invariant properties would be used to ensure that (a) no variable resource was incorrectly categorized as a constant, and (b) no constant (from that run) was interpreted as a variable.

Table 1. Use of file resources in eight invocations of Notepad

User	Log size (KB)[3]	# uses of privilege	# res. used	Actions
Rob	3588	951	281	Create new file on desktop
Rob	4042	1049	64	Create 2 files on desktop
Test	1966	518	56	Open (for reading) one file from command line and one from GUI
Test	1523	406	51	Open file for reading, save as new file (create file)
Test	2148	578	54	Open 2 files for reading and create 2 new files
Carla	2476	715	65	Create new file and print it
Test	51	20	5	Create and write file from command line
Carla	3539	963	57	Read, edit, and save file

[1] Many `ntdll.dll` calls to open an object also accept MAXIMUM_ALLOWED as a desired access; in such cases, the rights actually required depend on the operations (e.g., NtWriteFile) performed on that object's handle.

[2] In general, the second element of the pair is a set of access rights. Reading a file in Windows typically exercises five distinct access rights.

[3] Log sizes are large because logs are written in XML and expand the often-complex data structures used as arguments to kernel calls.

In total, 633 concrete resources were accessed at thirteen program counter values of Notepad.[4] Correlation identified 53 constant abstract resources (most of them DLLs) and 6 variable abstract resources. After all eight instantiations had been analyzed, 44 concrete resources remained unassigned.

The constant resources used by Notepad in our experiment include DLLs and system services. (Some files, directories, and non-DLL executables were identified as constants by our early software, which does not make use of environment variables or the program binary to help identify program constants. Our preliminary analysis identified three constant executables accessed—for reading—by one user's execution of Notepad: explorer, tweakui, and xemacs. The last two are certainly not Notepad program constants.)

System services, such as access to files stored on a server, are handled by special services in Windows; services are treated as resources, access to which is mediated by the kernel. Notepad uses three system services: "dav rpc service," wkssvc, and srvsvc. To access the services, the process exercises FILE_APPEND_DATA, FILE_READ_ACCESS, and FILE_WRITE_ACCESS privileges on named pipes in order to communicate with each system service.

The DLLs used by Notepad are shown in Table 2. DLLs were called from many different points in the program. A single program point can also trigger loading of a large number of DLLs; one program point started 20 DLLs.

Table 2. DLLs called by Notepad in eight invocations

xpsp2res.dll	mydocs.dll
ole32.dll	rpcrt4.dll
uxtheme.dll	audiodev.dll
wintrust.dll	drprov.dll
setupapi.dll	ntlanman.dll
riched20.dll	oleaut32.dll
shell32.dll	browseui.dll
comctl32.dll	mslbui.dll
ntshrui.dll	davclnt.dll
mpr.dll	shdocvw.dll
msctf.dll	apphelp.dll
netapi32	advapi32.dll
wininet.dll	kernel32.dll
userenv.dll	clbcatq.dll
cscui.dll	nview.dll

[4] This underestimates the number of resources actually accessed. The Notepad execution included one or more additional threads whose execution began in a DLL. At least one thread managed the GUI; it required many more DLLs and other privileged operations. In the interest of focusing on the most important aspects of our problem, we omitted those threads from our analysis.

The variable resources—all files and directories—used by Notepad are summarized in Table 3. Six abstract resources are identified by the values in the first two columns of Table 3.

The "PC values" in Table 3 are the last four digits of the hex representation of the program counter at the point of return from the call. The table records the number of concrete files that contribute to the abstract resource. For example, four directories from four different invocations of Notepad contributed to the fourth resource of Table 3:

```
C:\documents and settings\rob\desktop
C:\documents and settings\rob\desktop
C:\some_dir\new_files_here
C:\documents and settings\carla\My documents
```

In Table 3, we estimated how Notepad used each resource, based on what users were doing in the experiment.

Table 3. Variable resource use by Notepad

PC values	Access rights	# files	How resource is used by Notepad
2CC6 4A61 4EDE 4C30	DELETE FILE_APPEND_DATA FILE_READ_ATTR FILE_WRITE_ACCESS FILE_WRITE_ATTR FILE_WRITE_EA READ_CONTROL SYNCHRONIZE	8	Files created using Notepad
2683 2659 2DB0	FILE_READ_ACCESS FILE_READ_ATTR FILE_READ_EA READ_CONTROL SYNCHRONIZE	5	Files opened and read using Notepad (but not saved)
2CC6 2D89	FILE_READ_ACCESS SYNCHRONIZE	4	Files in same directory as file traversed by Notepad
2CC6 2AC3 2D89 4EDE	FILE_READ_ACCESS FILE_READ_ATTR FILE_READ_EA FILE_WRITE_ACCESS READ_CONTROL SYNCHRONIZE	4	Directory in which Notepad opened and saved files
2CC6 2D89	FILE_READ_ACCESS FILE_READ_ATTR FILE_READ_EA READ_CONTROL SYNCHRONIZE	8	Directory in which Notepad opened file (but did not save)
2D89	FILE_READ_ACCESS SYNCHRONIZE	3	Directory traversed in file chooser (?)

Correct assignments. Of the 53 constants identified by correlation, 49 were correctly identified. All six variable resources were correctly identified, representing from three to eight distinct files or directories each.

Variables identified as constants. Four resources that clearly are instances of variables were identified as constants by our algorithm. Two or three others (.exe files) are probably variables, yet they happened to occur in many of our runs, due to

the small number of test sites. Mis-assignments occurred, for example, when common directories (e.g., "`c:\Documents and Settings\All Users`") occurred in almost every run.[5] These mis-assignments could be avoided by identifying the invariant properties of constants in advance; static analysis of the program together with environment variables should be sufficient to identify almost all constants and effectively eliminate this type of mis-assignment.

Constants identified as variables. None.

Resources that the algorithm was unable to classify. The algorithm was unable to identify 44 concrete resources of the 633 that it encountered, due to insufficient evidence. For example, only one run used the printing function, which invokes DLLs not otherwise used; the algorithm therefore lacked enough evidence to classify them. In practice, we would expect that at the beginning of privilege profiling, there would be many unclassified resources, but that after a while very few would remain.

4 PROTECTING COLLABORATOR PRIVACY AND DETECTING INCORRECT INPUTS

We now consider how to protect the privacy of collaborators that provide input to the profile and to protect the profile from malicious inputs.

4.1 Protecting collaborator privacy

The privilege profile contains a list of resources, with the privileges that are required for each resource. The potential threat to privacy is that the names of resources could reveal information about the collaborator's site that is not intended to be public. For example, the path name of a file exposes information about the structure of the local file system. To a lesser extent, the full path name of a registry key can provide the same information. We would argue that other resources are much less likely to betray sensitive information. Of the various Windows resource classes, only file names and registry key names contain potentially sensitive information.

We argue that first, little or no information need be collected for constant resources, since the resource can be located using invariant properties that can already be present in the profile. Second, little information is needed for variable resources, since particulars about a resource (such as pathname) are not relevant to the program and can be ignored. Third, it will often be possible to express invariant properties in terms of environment variables or the values of registry keys.

However, in some cases it may be desirable to collect information about some resources, for example to establish a relation between resources (such as several files located in the "home" directory of an application). We argue that this can be done without jeopardizing collaborator privacy, since our only concern during correlation is to check for the appearance of the same strings in path names. The key idea is to hash each element in a path name. The hashes can easily be compared for equality, while the original path names cannot be re-computed from the hashes.

Consider the following approach.

In the profile, each constant element is represented by a hash value. The profile might contain partial path names for constant file and registry key resources, as described above (see Section 2.3). This information is distributed to each collaborator site. The partial path names identify the constant elements of the pathnames and elide the variable elements. At collaborator sites, local file path names are hashed element-wise and compared with the (hashed) names in the profile. If a match is found, the file corresponds to a constant resource in the profile; there is no need to send any path name to the EPP central site.

Information about variable resources is also sent to collaborator sites. This information does not include path names, since such names are not meaningful. Instead it includes, for each variable resource, the set of program points (which implies a set of access rights) at which the variable is accessed. If a program resource is accessed at the same set of program points, it is considered an instance of the variable resource. Again, there is no need to send information about the resource to the EPP central site.

If the file access pattern does not match any constant or variable resource in the profile, the path name can be sent to the EPP central site, hashed element-wise, together with information about the program points at which the file was accessed. The element-wise hashing protects the path name from possible attempts to extract information about the local system.

4.2 Malicious and incorrect inputs

There is a danger that malicious collaborators will provide incorrect information, either to vandalize the system or to trick it into attributing excessive privilege to a target program.

Any defense against malicious collaborators must rely on the assumption that there are very few of them compared with the total number of collaborators. By requiring corroboration for new privilege requirements from alternative sites and users, we can avoid premature acceptance of spurious inputs.

What kind of incorrect inputs can occur? The malicious user can attempt to introduce new program points and new constant resources. He can also introduce new access rights on constant or variable resources. We treat each of these threats in turn.

First, the user can attempt to introduce a new program point that is not in the program. Introduction of new program counter values can be checked against the program binary. Introduction of new ntdll operations for a given program counter value is discussed below.

Second, the malicious user can introduce a new constant resource (for example, access to a protected system file or a confidential personnel file belonging to a user). If we exploit the fact that constants must appear in the program binary, then this threat is extremely limited.

---

[5] It can be argued that `c:\Documents and Settings\All Users` should be considered a constant resource, as it is the value of the environment variable `%ALLUSERSPROFILE%`.

Third, the user can attempt to introduce new access rights on a constant or variable resource. Because we identify resources by the set of program points that access them and required access rights are determined by the ntdll operation (part of the program point), this requires introduction of a new program point for the resource. To defend against this threat, we can require that each program point in the profile be corroborated by N collaborators, where N can be a small absolute number or a percentage of the number of collaborators. The former is probably preferable, since unusual exceptions may result in rare program points being reached.

In the limit, it may prove difficult to distinguish in the privilege profile between extremely rare—but correct and benign—uses of privilege and the effect of concerted attacks on the profile. It remains to be seen what type of challenges arise in practice.

5 RELATED WORK

Program behavior profiles have been an important topic in anomaly-based intrusion detection, since they can be used to detect buffer overflows and other exploits that cause a program to behave in novel and usually undesirable ways. An accurate and parsimonious representation of program behavior is based on traces of its calls to the operating system kernel [4-6]. Our program profiles are coarser, in the sense that we are concerned only with their use of resources; many changes to the program code, including changing the order of calls, will be reflected in kernel call traces but be invisible to our method. Our profiles also require instrumentation of the program itself (to obtain program points), not just the interface to the kernel. On the other hand, profiles that focus on resource usage have the potential to be exceptionally parsimonious while still capturing the essence of the program's security implications.

Koved et al. [7] have collected information about a Java program's **exercise of privilege**. [7] is based on static analysis, but they are also working on empirical analysis, which is applicable to a wider range of languages. Inoue ([8], Ch. 4) uses an empirical approach in support of dynamic sandboxing for Java programs. Both [7] and [8] express privileges in terms of Java's convenient Permission class; our concept of privilege is useful when dealing with privileges defined by other platforms.

Anita Jones's thesis [9] provides a model for the distinction between **constant and variable resources** (or "program" vs. "user resources); Jones identified program-related resources in calls to the operating system. More recently, Ford [10] has used resource "ownership" as a criterion in intrusion detection.

6 DISCUSSION

We have shown that it is possible to collect information about a program's exercise of privilege from a variety of users and to produce an abstract profile of the program's use of resources. Such a profile could be used for program testing (avoiding unintended use of resources), program installation (establishing correct privileges for privileged programs), and intrusion detection (detecting anomalous use of resources).

Using empirical privilege profiling for intrusion detection is subject to the usual caveats that apply to any anomaly detection technique: detecting anomalies requires training, training takes time, and during training the system is unguarded. By using a large number of sites for training—which abstract resources support—this time can be minimized. Further, administrators can observe that the program exercises excessive privilege and investigate further. Another frequent objection to empirical approaches is that empirical profiles are typically incomplete, since rare but harmless behaviors are excluded even after extensive training. On the other hand, such profiles also exclude rare but possibly harmful behavior, such as the exercise of back doors. We believe that rare false positives against a mature profile do not pose a serious problem.

Distinguishing between constant and variable resources makes it possible to monitor the resources that the program accesses on the user's behalf and detect insider attacks. If the set of those resources can be characterized for any one user, it might be possible to detect the anomalous—and possibly unauthorized—use of privilege. In this context, one might wish to use *optimistic access control* [11], in which questionable attempts to access resources (in this case, novel accesses) are permitted but logged for later analysis.

Yet another possibility is for a software provider to provide clients with a privilege profile for the software, with the guarantee that it includes all privileges required by the program. At run-time, the client can perform privilege enforcement in a manner analogous to model carrying code [12, 13], that is, prohibit any access to the resources that exceeds the profile.

In performing this work, we encountered instances of surprising exercise of privilege by Windows programs. For example, the commonplace Microsoft Calculator writes to the protected file `C:\WINDOWS\win.ini` (to record use of Scientific mode) *only if* the user is an Administrator (the write silently fails for users not in the Administrator group). The standard Windows file chooser, used by Notepad as well as many other applications, opens files with wild abandon for reasons of user convenience—for example, to obtain a custom icon to display next to the file name. While such practices thwart traditional attempts at reducing program privileges, the use of tools such as we are working on would enable system designers and programmers to find ways to avoid them.

The basic premise of this work is that by observing a program's actual use of resources, we can find the least privilege that it needs. The profligate use of resources that the program does not in some sense *really* need threatens that premise. We hope to be able in the future to learn ways to distinguish vital from trivial use of resources—for example, by observing program behavior in restricted environments. At the very least, we can identify and expose the problems and contribute to their solution.

7 ACKNOWLEDGEMENTS

We owe the idea of creating an empirical privilege profile to Lee Badger, who suggested tracking a program's exercise of privilege at a large number of user sites, possibly in the manner of SETI@home [14]. We also wish to thank the reviewer who suggested that empirical privilege profiling might be used for privilege enforcement in a manner analogous to model-carrying code. The work described in this paper was supported under DARPA contract W3194Q-04-C-R260.

8 REFERENCES

Information about privileges and ntdll operations is available from the Microsoft on-line library [15]. Microsoft does not publish detailed information about ntdll operations. We relied on a book that we and others have found to be reliable [16]. We also used [17], which is not as detailed as [16], but sometimes has additional information. We are also aware of other hacker resources [18, 19].

References cited in this report:

[1] Saltzer, J.H. and M.D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, 1975. 9(63): p. 1278-1308.

[2] Gao, D., M.K. Reiter, and D. Song. "On Gray-Box Program Tracking for Anomaly Detection," in *Proceedings of the USENIX Security Symposium*. 2004, pp. 103-118.

[3] Gao, D., M.K. Reiter, and D.X. Song. "Gray-box extraction of execution graphs for anomaly detection," in *Proceedings of the ACM Conference on Computer and Communications Security*. 2004, pp. 318-329.

[4] Forrest, S., S.A. Hofmeyr, and A. Somajayi. "A Sense of Self for UNIX Processes," in *Proceedings of the IEEE Symposium on Computer Security and Privacy*. 1996: IEEE Press, pp.

[5] Hofmeyr, S.A., S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, 1998. 6(3): p. 151–180.

[6] Marceau, C. "Characterizing the Behavior of a Program Using Multiple-Length N-grams," in *Proceedings of the New Security Paradigms Workshop*. 2000. Ballycotton, Ireland, pp.

[7] Koved, L., M. Pistoia, and A. Kershenbaum. "Access rights analysis for Java," in *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2002. Seattle WA, pp.

[8] Inoue, H., *Anomaly Intrusion Detection in Dynamic Execution Environments*, in *Computer Science*. 2005, University of New Mexico: Albuquerque NM.

[9] Jones, A.K., *Protection in Programmed Systems*. 1973, Carnegie-Mellon University.

[10] Ford, R., M. Wagner, and J. Michalske. "Gatekeeper II: new approaches to generic virus prevention," in *Proceedings of the Virus Bulletin 2004*. 2004. Chicago IL, pp.

[11] Povey, D. "Optimistic Security: A new access control paradigm," in *Proceedings of the New Security Paradigms Workshop*. 1999, pp.

[12] Sekar, R., et al. "Model-Carrying Code (MCC): a new paradigm for mobile-code security," in *Proceedings of the New Security Paradigms Workshop*. 2001. Cloudcroft, New Mexico, pp.

[13] Sekar, R., et al. "Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'03)*. 2003. Bolton Landing, New York, pp.

[14] SETI@home, http://setiathome.ssl.berkeley.edu/.

[15] Microsoft Corporation, The MSDN Library, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secauthz/security/privileges.asp.

[16] Nebbett, G., *Windows NT/2000 Native API Reference*. 2000, Thousand Oaks, CA: New Riders Publishing.

[17] NTinternals.net, Undocumented Functions for Microsoft Windows NT/2000, http://undocumented.ntinternals.net/.

[18] Holy Father, Hooking Windows API - Technics [sic] of hooking API functions on Windows, www.assembly-journal.com/include/getdoc.php?id=244&article=157&mode=pdf.

[19] Ivanov, I., API hooking revealed, http://www.codeproject.com/system/hooksys.asp.