Least Privilege and More[1]

Fred B. Schneider
Cornell University, Ithaca, New York, USA

Introduction

What today is known as the *Principle of Least Privilege* was described as a design principle in a paper by Jerry Saltzer and Mike Schroeder [4] first submitted for publication roughly 30 years ago:

> "f) Least privilege: Every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily, this principle limits the damage that can result from an accident or error. It also reduces the number of potential interactions among privileged programs to the minimum for correct operation, so that unintentional, unwanted, or improper uses of privilege are less likely to occur. Thus, if a question arises related to misuse of a privilege, the number of programs that must be audited is minimized. Put another way, if a mechanism can provide 'firewalls,' the principle of least privilege provides a rationale for where to install the firewalls. The military security rule of 'need-to-know' is an example of this principle."

The power of this principle comes from leaving unspecified how frequently privileges might change and their granularity. Back in 1972, Roger Needham certainly understood the value of support for dynamic assignments of privileges, writing [3]:

> "Protection regimes are not constant during the life of a process. They may change as the work proceeds, and in a fully general discussion they should be allowed to change arbitrarily. Statements would be allowed, for example, to the effect that certain segments were only accessible if the value standing in a system microsecond clock were prime. In practice one departs from full generality, and limits those circumstances which may give rise to a change of protection regime."

My own interest in the Principle of Least Privilege developed in connection with devising security enforcement mechanisms for systems structured in terms of a base and a set of extensions which augment the functionality of that base. Such extensible systems are prevalent today in mass-market PC software, where we see new hardware being accommodated in Microsoft Windows platforms through "plug and play" and we see web

1 Supported in part by AFOSR grant F49620-00-1-0198, Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory Air Force Material Command USAF under Agreement number F30602-99-1-0533, National Science Foundation Grant 9703470, and ONR Grant N00014-01-1-0968. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

browsers—hence, the web itself—supporting new data formats by use of downloaded "helper apps" that extend a browser's functionality.

A misbehaving extension *Ext* has the potential to compromise the base system *B* it extends. Examples abound: email containing executable attachments, Microsoft Word documents bearing hostile macros, and new browser "helper apps" that are a far cry from being helpful. This situation could be improved if we posit some sort of reference monitor that intercepts all program actions and, based on privileges held by the issuer of the action, blocks those that would be disruptive. However, to make this vision a reality, two technical questions must be solved:

(1) Implementing the reference monitor.

(2) Determining a policy for it to enforce.

Regarding (1), my collaborators and I have elsewhere reported success with program rewriters to modify an object program before execution, adding tests that effectively in-line a fine-grained reference monitor [2]. This paper sketches my current thinking on (2).

What policy to enforce?

Least privilege. Policies consistent with the Principle of Least Privilege depend not only on the code to be executed but also on what job that code is intended to do. For an extension *Ext* and some specification Σ_{Ext} of a job to be done, we define $\mu Priv(Ext, \Sigma_{Ext})$ to be the policy that grants the minimum privileges needed for execution of *Ext* to satisfy Σ_{Ext}. (A policy here is a mapping from system histories to sets of privileges.) As an example, specification Σ_{Ext} of a spell-checker extension *Ext* for a word processor might specify that misspelled words be flagged in the word processor's open file *F*; we would then expect $\mu Priv(Ext, \Sigma_{Ext})$ to be a policy that permits the spell-checker read (but not write) access to *F*, read (but not write) access to a file containing a spelling dictionary, and read/write access to a file containing user-added spellings for local jargon terms.

It is clear how the base system comes to get an extension *Ext*, but how does it get $\mu Priv(Ext, \Sigma_{Ext})$ for use by its reference monitor? Here are two possible approaches.

(1) The base system could itself compute $\mu Priv(Ext, \Sigma_{Ext})$.

(2) The base system could fetch $\mu Priv(Ext, \Sigma_{Ext})$ from some site *S*.

Approach (1) presumes that $\mu Priv(Ext, \Sigma_{Ext})$ can be computed—a questionable supposition. Implicit in computing $\mu Priv(Ext, \Sigma_{Ext})$ is establishing that extension *Ext* satisfies specification Σ_{Ext}, and we know that question cannot be decided for general-purpose programming and specification languages. There might exist specialized languages, however, for which $\mu Priv(Ext, \Sigma_{Ext})$ could be computed; this is a research question that bears closer scrutiny. One might start by restricting consideration to specifications Σ_{Ext} that are safety properties, because the language of specifications now can be restricted to state predicates that hold throughout system execution. The weakest precondition (*wp*) predicate transformer might then provide a starting point for defining $\mu Priv$ by structural induction on *Ext*.

Approach (1) also presumes that Σ_{Ext} is known. This, too, is a supposition of dubious practicality. Since extensions are generally downloaded with some expectation of the job they are intended to do, one might suspect that a high-level, task-oriented specification

Σ_{Ext} would be known to the initiator and serve as the impetus for the *Ext* download. But employing such a high-level task-oriented specification does not suffice if *Ext* involves implementation details that are not obvious for the task and thus have been omitted from Σ_{Ext}. For example, recall the spell-checker extension introduced above, which is specified in terms of a single file *F*. This spell-checker actually also involves accessing two other files (a spelling dictionary and a jargon dictionary) and might in addition even access a backing-store file perhaps over a local network. Such knowledge of implementation details is not going to be available to the initiator of an *Ext* download and, therefore, would not be included in high-level task-oriented specification Σ_{Ext}, though clearly $\mu Priv(Ext, \Sigma_{Ext})$ would need to include privileges for accessing the spelling dictionary, the jargon dictionary, and the backing-store.

If *Ext* cannot be deduced locally, then perhaps it could be downloaded and checked? Unfortunately, this architecture also has problems. The local checking is really a form of policy review, and policy review is a hard problem whenever the policy being checked is complicated. A specification Σ_{Ext} that involves internal details is going to be complicated and thus difficult for a human to understand. The alternative to policy review is simply to trust the source of Σ_{Ext}. But, then, why not simply trust the source of *Ext* to provide a safe extension and dispense with reference monitoring altogether?

For approach (2) to be workable, either *S* must be trusted or the base system must itself have some means to check whether what it has fetched equals $\mu Priv(Ext, \Sigma_{Ext})$. The latter is unworkable for the reasons argued above. Regarding the former, an obvious question is whether trusting *S* to provide $\mu Priv(Ext, \Sigma_{Ext})$ could be materially different from trusting *S* to provide a safe implementation of *Ext*.

And more. At least for the time being, then, it seems as though obtaining $\mu Priv(Ext, \Sigma_{Ext})$ for use by a reference monitor associated with the base of an extensible system is infeasible, and an alternative must be sought. So the policies we are now investigating seek to prevent extensions from subverting a base system or, equivalently, seek to prevent any extension from violating the assumptions underlying the design and implementation of that base. Such assumptions include:

- Characteristics of the programming model employed for building the base, such as properties of underlying system abstractions and language-level abstractions. For example, the separate address spaces usually accorded to process abstractions bring guarantees about integrity of storage; and type systems in modern programming languages, like Java and C#, bring guarantees about how certain variables can be used.
- Invariants that the base maintains about state. For example, a complicated linked-list data structure might be characterized by an invariant stating which nodes are reachable from each other; each routine to manipulate the data structure is then designed to (i) work correctly if that invariant holds prior to execution and (ii) upon termination, leave the data structure in a state satisfying the invariant.

Provided these assumptions can be expressed as safety properties—and most can—then they can be enforced by use of in-lined reference monitoring. Prior to execution, each extension is rewritten by adding checks that ensure no action the extension performs will violate any assumption required by the base system.

Notice that in this alternative to $\mu Priv(Ext, \Sigma_{Ext})$, a single policy is being employed, independent of extension *Ext*. The problems of deciding what specification Σ_{Ext} to use with a given extension *Ext* is thus eliminated. But the use of a single policy for all extensions implies that the policy being enforced might not be as restrictive as it could be (thereby admitting attacks) or might be too restrictive (thereby ruling-out execution of certain extensions). And there is thus some flexibility in formulating a policy for a given base.

Some final comments

The articulation of abstractions and principles is an important facet of doing research in computing systems. An implementation is certainly one way to demonstrate the utility of a new systems abstraction or principle, with system performance a sensible figure of merit. However, some abstractions are useful even though they cannot be implemented. Belady's optimal page replacement policy [1], which involves predicting future memory references and therefore is unrealizable in practice, is one example. The Principle of Least Privilege might be another, offering value primarily as a benchmark against which to compare policies that are being enforced—when compared with $\mu Priv(Ext, \Sigma_{Ext})$, a deployed policy would be considered inferior if it either admits additional attacks or it excludes certain classes of extensions.

The classical approach to computer security—address space isolation associated with processes—would seem a good place to start in a comparison of security policies for extensible systems. It isn't. The context switches required on modern processors for communication and synchronization between separate processes make it impractical to have fine-grained interaction between a base implemented as one process and an extension as another. Without the possibility of such fine-grained interaction, the set of functions that can be implemented as extensions becomes quite limited.

But with in-lined reference monitors, different programs can be isolated from each other without incurring the high cost of context switches. In fact, many forms of fine-grained access control that are not practical with traditional reference monitors become practical with in-lined reference monitors. Another concern now confronts us, though: How best to exploit the flexibility. To make progress here, not only must we learn the art of writing policies but we must also develop the mathematical tools for analyzing them. Collections of weak policies are likely to provide workable defenses for broad sets of extensions, for example. Weak policies might well be easier for humans to understand, too. Exactly how these advantages trade with the "security" $\mu Priv(Ext, \Sigma_{Ext})$ provides is the ultimate question. For the present, however, it seems that practical protection for extensible systems is most easily obtained using policies that grant more privileges than would $\mu Priv(Ext, \Sigma_{Ext})$—the least privilege and more.

Acknowledgments. Helpful comments on a preliminary draft of this paper were provided by Lorenzo Alvisi, Butler Lampson, Greg Morrisett, Andrew Myers, and Mike Schroeder.

References

1. BELADY, L.A., 'A study of replacement algorithms in a virtual storage computer,' *IBM Systems Journal* vol. 5, no. 2,1966, pp. 78-101.

2. Erlingsson, U. and Schneider, F.B., ‘SASI enforcement of security policies: a retrospective,’ *Proceedings of the New Security Paradigms Workshop*, Caledon Hills, Ontario, Canada, September 1999, ACM, pp. 87-95.
3. Needham, R.M., ‘Protection systems and protection implementations,’ *Proc. 1972 Fall Joint Computer Conference*, AFIPS Conf. Proc., vol. 41, pt. 1, pp. 571-578.
4. Saltzer, J.H. and Schroeder, M.D., ‘The Protection of information in computer systems,’ *Proceedings of the IEEE,* vol. 63, no. 9 (Sept 1975), pp. 1278-1308.