Least Privilege and More[1]

Fred B. Schneider
Department of Computer Science
Cornell University
Ithaca, New York 14853

Introduction

Operating system *access control mechanisms* are intended to protect programs and data from corruption, yet still allow sharing of these resources. The goal is to support a broad range of *authorization policies*. But at least in commercial operating systems of the last few decades, we find support only for authorization policies concerning operations implemented by the operating system itself; policies concerning operations or resources that applications implement are not supported. This made sense

- when operations provided by the operating system were the sole means by which programs communicated and synchronized, and
- when the operating system was small, since positioning the access control mechanism inside the operating system then resulted in a trusted computing base that was small.

Today's applications, however, are increasingly being structured in terms of a *base* and a set of *extensions* which augment the functionality of that base[2] and which do not use the operating system for communication and synchronization. In addition, today's operating systems are no longer small. So associating the access control mechanism with an operating system interface has become less sensible.

A misbehaving extension has the potential to compromise the base system it extends because, for performance reasons, extensions are typically executed in the same address space and with the same privileges as the base and, therefore, have access to resources on which the base depends. Moreover, once compromised, a base system might then wreck havoc by abusing its privileges. Examples abound: email containing viruses as executable attachments, Microsoft Word documents bearing hostile macros, and new browser "helper apps" that are a far cry from being helpful.

The situation could be improved if we posit some sort of reference monitor [1] that intercepts all program actions and, based on privileges held by the issuer of the action, blocks those that cause

[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.

[2] Examples include mass-market PC software, where we see new hardware being accommodated in Microsoft Windows platforms through "plug and play" and we see web browsers—hence, the web itself—supporting new data formats by use of downloaded "helper apps" that extend a browser's functionality.

compromise. (See sidebar: Reference Monitor Architectures.) To make this vision a reality, two technical questions must be solved:

(1) Implementation of such a reference monitor.

(2) Determination of a suitable authorization policy for that reference monitor to enforce.

Regarding (1), one promising approach is to use a program rewriter that modifies an object program before execution, adding tests that effectively in-line a fine-grained reference monitor [3]. This paper sketches some recent thinking on (2).

What Authorization Policy to Enforce?

Much is gained by allowing a program's privileges to change as execution progresses, with the granularity of privileges rather fine-grained. The benefits were first articulated in 1972 by Roger Needham, writing [4]:

> "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."

Three years later, in 1975, Saltzer and Schroeder's formulation of these ideas, today known as the *Principle of Least Privilege*, was published [5]:

> "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 Saltzer-Schroeder formulation does not speak explicitly about the granularity of privileges, but actions speak louder than words and their contemporaneous Multics operating system offered a fine-grained access control mechanism as part of its virtual memory system. So, in that sense (and others), Multics can be seen as superior to today's operating systems, with their relatively coarse-grained targets of access control (*viz* programs and files). The Saltzer-Schroeder formulation also does not explicitly call out the benefits of allowing the set of privileges associated with a program to change as execution progresses, but Multics did provide some support (with its rings of protection) for dynamic authorization policies.

Least Privilege. Policies consistent with the Principle of Least Privilege depend not only on the code to be executed but also on what that code is intended to do. For an extension *Ext* and some specification Σ_{Ext} describing what a user expects of *Ext*, define $\mu Priv(Ext,\Sigma_{Ext})$ to be the policy[3] that grants the minimum privileges for execution of base system *B* when augmented by *Ext* to satisfy Σ_{Ext}. As an example, a specification Σ_{Ext} for a word processor's spell-checker extension *Ext* might stipulate 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 gives the spell-checker read (but not write) access to *F*, read (but not write) access a file containing a spelling dictionary, and read/write access to a file containing user-added spellings for local jargon terms.

Interposition of a reference monitor between a base *B* and extension *Ext*, along with knowledge of $\mu Priv(Ext, \Sigma_{Ext})$, allows *Ext* to be executed even if its provider is not trusted; the reference monitor simply enforces $\mu Priv(Ext, \Sigma_{Ext})$. The crucial question then becomes how $\mu Priv(Ext, \Sigma_{Ext})$ might be obtained? Here are two possible answers.

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

(2) The base system could fetch $\mu Priv(Ext, \Sigma_{Ext})$ from elsewhere.

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* indeed satisfies Σ_{Ext}, and we know that question is undecidable for general-purpose programming and specification languages. There might exist specialized languages for which $\mu Priv(Ext, \Sigma_{Ext})$ could be computed; this is a research question that bears closer scrutiny.

Approach (1) also presumes that Σ_{Ext} is known, and this too is a questionable supposition. Extensions are generally downloaded with some expectation of the job they are intended to do, so one might expect that a known, high-level task-oriented specification Σ_{Ext} was the impetus for downloading *Ext.* But any specification Σ_{Ext} that is high-level will likely lack the low-level information needed for determining whether *Ext* accesses only those resources needed for accomplishing its task. For example, recall the spell-checker extension introduced above. A high-level task-oriented specification for *Ext* would likely only discuss the single file *F* where misspellings are to be found. Yet, this extension actually accesses two other files (a spelling dictionary and a jargon dictionary) and might, in addition, access a backing-store file perhaps even remotely over a local network. These lower-level implementation details—the two other files and the remote backing store—are not necessarily going to be known to the initiator of the *Ext* download and, therefore, would not be included in high-level task-oriented specification Σ_{Ext}. Yet, clearly, $\mu Priv(Ext, \Sigma_{Ext})$ would need to include privileges for accessing the spelling dictionary, the jargon dictionary, the network, and the backing-store. So high-level specification Σ_{Ext} lacks information about how *Ext* works that would be needed for deducing $\mu Priv(Ext, \Sigma_{Ext})$.

If $\mu Priv(Ext, \Sigma_{Ext})$ cannot be deduced locally, then perhaps it could be obtained elsewhere. One would need a basis to trust a policy *LP* (say) gotten that way. Automatically checking *LP* has the same undecidability problems as automatically computing $\mu Priv(Ext, \Sigma_{Ext})$; manual inspection of *LP* requires a human to analyze a complicated policy and a program (*Ext*) that might only be available in binary form. So, for all intents and purposes, to obtain $\mu Priv(Ext, \Sigma_{Ext})$ from elsewhere

[3] A *policy* here is defined as a mapping from system histories to sets of privileges.

is tantamount to trusting the provider of that policy; and an obvious question, then, is whether trusting a provider of *LP* is materially different from trusting some provider to offer a secure version of *Ext*.

And More. So at least for the time being, it seems as though obtaining $\mu Priv(Ext, \Sigma_{Ext})$ for use by a reference monitor interposed between a base and its extensions is infeasible, and alternatives must be sought. One such alternative, which might be called a *Principle of Most Privilege*, is to enforce a policy *vPriv(B)* that merely prevent extensions from subverting the base system *B* or, equivalently, to prevent extensions from destroying the guarantees on which the correct operation of *B* depends. Such guarantees include:

- Properties implied by the programming model employed for building the base. 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 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.

Notice that *vPriv(B)* is independent of any extension *Ext*. The problem of deciding what specification Σ_{Ext} to use with a given extension *Ext* is thus eliminated. Moreover, if the guarantees being defended can be expressed as safety[4] properties—and most can—then the guarantees of interest can be enforced by in-line reference monitoring.

Many policies are stronger than *vPriv(B)*.[5] Selection of a single one of these for use with 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 some, if not all, execution by certain extensions). So instead of enforcing a single policy, we might postulate a small set of categories and associate a separate policy with each. A user or some third party would then classify the intent of each extension in terms of those categories—such as, an editor, a game, or a data viewer; the associated policies would be enforced whenever that extension executes. (Extensions that fall in no category are executed, as before, under a policy that does not take into account the purpose of the extension and thus is probably more restrictive than the other polices.) Note that, although individual users might be the ones to define categories and their associated policies, *a priori* wide-spread agreement on a small set of categories as being standard would allow code producers to ensure that their extensions satisfy expected policies.

[4] A *safety property* stipulates that some set of finite-length "bad" execution prefixes never occur.
[5] Policies weaker than *vPriv(B)*are uninteresting because they are not strong enough to prevent attacks on the base systems's enforcement mechanism.

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. However, some abstractions are useful even though they cannot be implemented. Belady's optimal page replacement policy [2], 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 incorrectly restricts the functionality of *Ext*.

This paper not only revisits a classic security principle but also revisits a classic abstraction, the reference monitor. 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. The weak policies entailed by our Principle of Most Privilege 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. This is a revised version of a paper written in honour of Roger Needham (1935 - 2003) for the conference "Roger Needham: 50 and 5" at Microsoft Research, Cambridge, England (Feb 17, 2003) held shortly before his death. The proceedings will appear in the Springer-Verlag's Texts and Monographs in Computer Science. Helpful comments on drafts of that paper were provided by Lorenzo Alvisi, Ulfar Erlingsson, Butler Lampson, Greg Morrisett, Andrew Myers, E. Gun Sirer, and Mike Schroeder.

References

[1] J. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Electronic Systems Divison, Hanscom Air Force Base, Hanscom, MA, 1974.

[2] L. A. Belady. A study of replacement algorithms in a virtual storage computer. *IBM Systems Journal 5*, No. 2 (1966), pp. 78-101.

[3] U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. *Proceedings of the New Security Paradigms Workshop* (Caledon Hills, Ontario, Canada, September 1999), Association for Computing Machinery, pp. 87-95.

[4] R. Needham. Protection systems and protection implementations. *Proceedings 1972 Fall Joint Computer Conference*, AFIPS Conf. Proc., vol. 41, pt. 1, pp. 571-578.

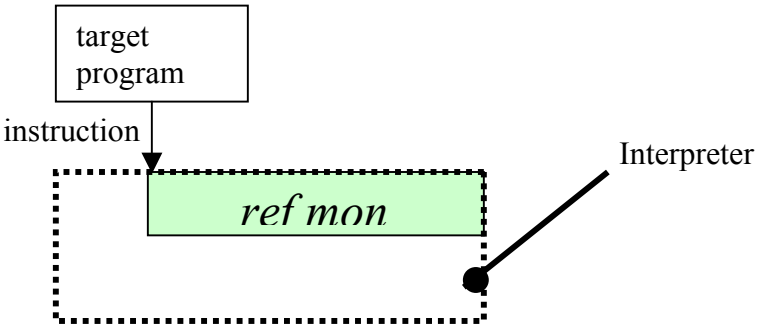
[5] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEE, 63*, 9 (Sept 1975), pp. 1278-1308.

Sidebar: Reference Monitor Architectures

A *reference monitor* [1] must be:

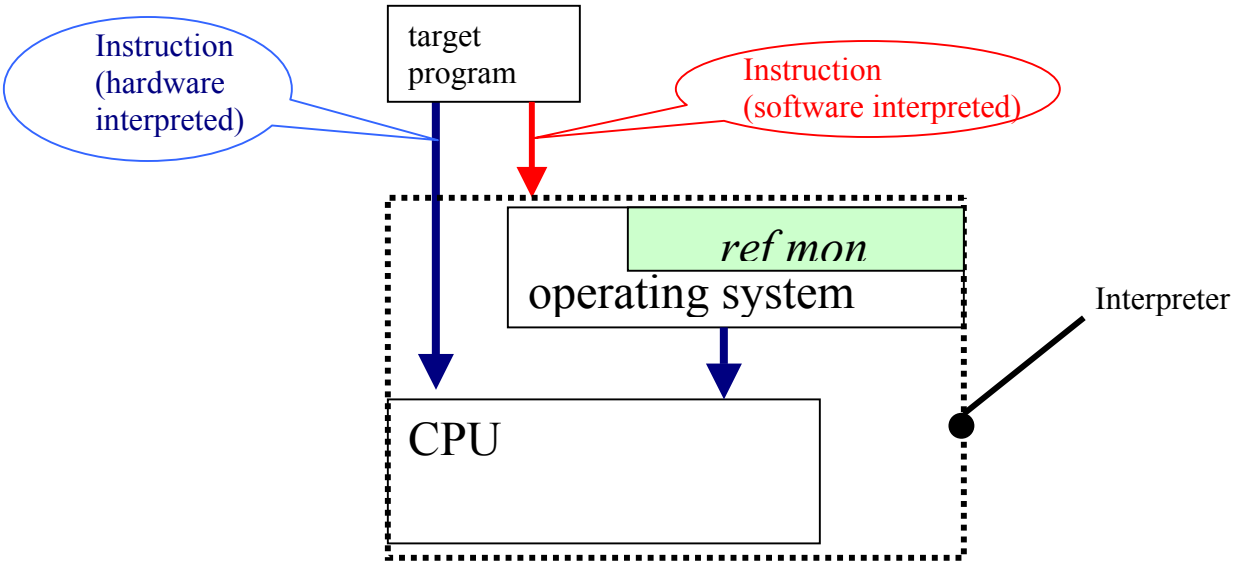
- tamper-proof,
- invoked whenever an event occurs that is relevant to the policy being enforced, and
- small enough to be trusted (through testing or analysis).

A reference monitor can be understood in terms of an interpreter that is trusted not only to implement the semantics of an instruction set but also to perform checks so that executions forbidden by a policy are prevented.



Implementing this interpreter in software leads to significantly slower execution of target programs than would be possible were the raw hardware used as the interpreter. Nevertheless, in some settings, this performance penalty has not been a problem. For example, LISP programs are frequently implemented by software interpreters.

Historically, concerns about raw execution speed have dominated, and an interpreter was implemented by a combination of hardware and software. The CPU was the sole interpreter for most instructions; the lowest levels of the operating system interpreted the remaining instructions.



Checks made by the reference monitor with this architecture are located in the operating system. Information about only certain program events is thus available to the reference monitor, and a somewhat impoverished vocabulary of events must suffice for formulating security policies. Exactly which events are in that vocabulary depends on what causes control to enter the operating system. Two approaches have been prevalent:

- The effect of executing certain instruction opcodes causes a trap, and in handling that trap the processor transfers control to the operating system.
- The effect of referencing certain addresses causes the memory management hardware to signal a trap, and in handling that trap the processor transfers control to the operating system.

Use of memory reference traps can support finer-grained security policies (depending on the virtual memory architecture) but at a cost of more frequent context switches into the operating system; use of opcodes is most natural for security policies that concern abstractions manipulated by routines provided by the operating system.

With an *in-lined reference monitor*, security checks are added to a machine-language target program some time before that program starts executing [3]. Effectively, the reference monitor is in-lined into the target program. The security checks are designed so that they cannot be circumvented by the target. Since the security checks have access to the target's internals, policies concerning application-specific abstractions can be enforced—and quite efficiently, too, because only those checks needed for the given policy and target are in-lined by the program rewriter. Moreover, with this enforcement scheme, context switches into the operating system are not required each time a security check is made.