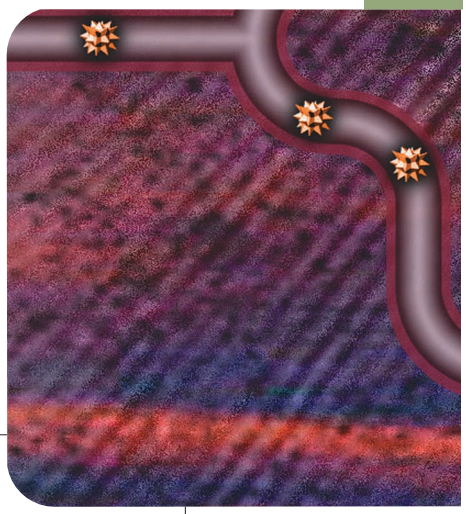
Least Privilege and More



With new methods for enforcing security policies comes the opportunity to formulate application-specific policies. But leveraging that flexibility might prove a difficult problem—not only in practice, but also in theory.

Fred B. Schneider
Cornell University

Access control mechanisms are intended to protect programs and data from corruption, yet still allow sharing of these resources. Their goal is to support a broad range of policies. But at least in recent commercial operating systems (OSs), we find support only for such policies concerning operations implemented by the operating system itself; policies concerning operations or resources that applications implement are not supported. This made sense when operating system operations were the sole means by which programs communicated and when operating systems were small (because positioning the access control mechanism inside the operating system resulted in a small trusted computing base). It makes less sense now.

Today's applications are increasingly structured in terms of a *base* piece of software and a set of *extensions* that augment the base's functionality and that do not use the operating system for communication. For example, mass-market PC software accommodates new hardware in Microsoft Windows platforms through "plug and play," and Web browsers—hence, the Web itself—support new data formats through downloaded "helper apps" that extend the browser's functionality. In addition, today's operating systems are no longer small. Thus, associating the access control mechanism with an operating system interface has become less sensible.

A malevolent extension has the potential to compromise the base system it extends because, for performance reasons, extensions typically execute 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 wreak 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 far cries from being helpful.

The situation could improve if we posit some sort of reference monitor[1] that intercepts all program actions and, based on privileges held by the action's issuer, blocks those that cause compromise (see the "Reference monitor architectures" sidebar). To make this vision a reality, we must answer two technical questions:

- How do we implement such a reference monitor?
- How do we determine a suitable policy for that reference monitor to enforce?

Regarding reference monitor implementation, one promising approach is to use a program rewriter that modifies an object program before execution, interspersing tests with the object program's instructions. This effectively inserts "inline" a fine-grain reference monitor.[2] That leaves the policy question, and I discuss some of my recent thinking on that in this article.

Which policy to enforce?

Much is gained by allowing a program's privileges to change as execution progresses—especially when privileges are fine-grained. Roger Needham first articulated the benefits, when, in 1972, he wrote:

> "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

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 an instruction set's semantics but also to perform checks so that it prevents executions forbidden by a policy. Figure A depicts this structure.

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, software interpreters frequently implement LISP programs.

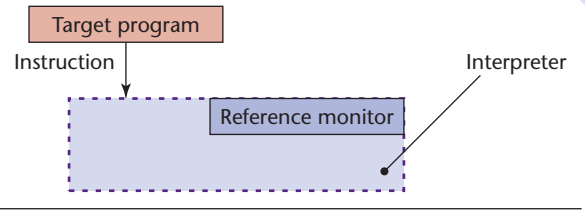


Figure A. A reference monitor as an interpreter.

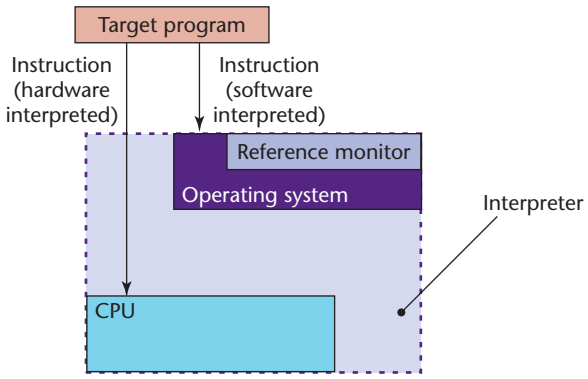


Figure B. An interpreter that leverages direct execution of instructions on hardware.

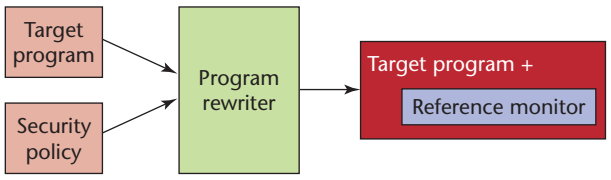


Figure C. A system supporting inlined reference monitors.

Historically, concerns about raw execution speed have dominated concerns about providing flexible means for enforcing security policies, and interpreters were realized by combining hardware and software. The CPU was the sole interpreter for most instructions; the operating system's (OS) lowest levels interpreted the remaining instructions. Figure B depicts a schematic representation of this architecture.

With this architecture, reference monitor checks are located in the OS. 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 are prevalent:

- The effect of executing certain instruction opcodes causes a transfer of control to the OS.
- 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 OS.

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

With an inlined reference monitor, the system adds security checks to a machine-language target program some time before that program starts executing.[2] Effectively, the reference monitor is inlined into the target program. The security checks are designed so that the target cannot circumvent them. Because 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 inserted by the program rewriter. Moreover, with this enforcement scheme, context switches into the OS are not required each time a security check is made. Figure C shows a system structure supporting inlined reference monitors.

References

1. J. Anderson, "Computer Security Technology Planning Study," tech. report, ESD-TR-73-51, Electronic Systems Divison, Hanscom Air Force Base, Hanscom, MA, 1974.
2. U. Erlingsson and F.B. Schneider, "SASI Enforcement of Security Policies: A Retrospective," *Proc. New Security Paradigms Workshop* (NSPW 99), ACM Press, 1999, pp. 87–95.

> 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."[3]

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

> "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."[4]

The Saltzer-Schroeder formulation does not speak explicitly about privilege granularity. But actions speak louder than words, and their contemporaneous Multics mainframe timesharing system offered a fine-grain access control mechanism as part of its virtual memory system. So, in that sense (and others), we can view Multics as superior to today's OSs, with their relatively coarse-grain access control targets (namely, programs and files). The Saltzer-Schroeder formulation also does not explicitly mention the benefits of allowing a program's set of associated privileges to change as execution progresses, but Multics did provide some support (with its rings of protection) for dynamic policies.

Least privilege

Policies consistent with the Principle of Least Privilege depend not only on the code to be executed but also on that code's purpose. For an extension *Ext* and some specification Σ_{Ext} describing what a user expects of *Ext*, we define $\mu Priv_B(Ext,\Sigma_{Ext})$ to be the policy that grants the minimum privileges for execution of base system *B* when augmented by *Ext* to satisfy Σ_{Ext}. For 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 $\mu Priv_B(Ext,\Sigma_{Ext})$ to be a policy that gives the spell-checker read (but not write) access to *F*, read (but not write) access to a file containing a spelling dictionary, and read and write access to a file containing user-added spellings for local jargon. Formally, we might define policies to be mappings from system histories to sets of privileges; $\mu Priv_B(Ext,\Sigma_{Ext})$ would then evaluate to such a mapping.

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

- *Approach 1*: The base system could compute $\mu Priv_B(Ext, \Sigma_{Ext})$.
- *Approach 2*: The base system could fetch $\mu Priv_B(Ext, \Sigma_{Ext})$ from elsewhere.

Approach one. The first approach presumes that $\mu Priv_B(Ext, \Sigma_{Ext})$ can be computed—a questionable supposition. Implicit in computing $\mu Priv_B(Ext, \Sigma_{Ext})$ is establishing that extension *Ext* indeed satisfies Σ_{Ext}, and we know that the question is undecidable for general-purpose programming and specification languages. Specialized languages might exist for which $\mu Priv_B(Ext, \Sigma_{Ext})$ could be computed; this research question bears closer scrutiny.

This first approach 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 we might expect that a known, high-level task-oriented specification Σ_{Ext} was the impetus for downloading *Ext*. But any high-level specification Σ_{Ext} 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 earlier. A high-level task-oriented specification for *Ext* would likely only discuss the single file *F* in which we seek to find misspellings. Yet, this extension actually accesses two other files (a spelling dictionary and a jargon dictionary) and might, in addition, store these files or other information remotely, over a local network. Such lower-level implementation details—the two other files and the remote storage—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, $\mu Priv_B(Ext, \Sigma_{Ext})$ must include privileges for accessing the spelling dictionary, the jargon dictionary, and the network. So high-level specification Σ_{Ext} lacks information about *Ext* that is needed for deducing $\mu Priv_B(Ext, \Sigma_{Ext})$.

Approach two. If $\mu Priv_B(Ext, \Sigma_{Ext})$ cannot be deduced locally, then perhaps we could obtain it elsewhere. We would need a basis to trust a policy, say, *LP*, obtained this way. Automatically checking *LP* has the same undecidability problems as automatically computing $\mu Priv_B(Ext, \Sigma_{Ext})$; manually inspecting *LP* requires a human to ana-

lyze a complicated policy and a program (*Ext*) that might only be available in binary form. So, to obtain $\mu Priv_B(Ext, \Sigma_{Ext})$ from elsewhere is tantamount to trusting that policy's provider; an obvious question, then, is whether trusting an *LP* provider is materially different from trusting a provider to offer a secure version of *Ext*.

And more

So at least for the time being, it seems as though obtaining $\mu Priv_B(Ext, \Sigma_{Ext})$ for use by a reference monitor interposed between a base and its extensions is infeasible, and we must seek alternatives. One such alternative, which we might call a *Principle of Most Privilege*, is to enforce a policy $\nu Priv(B)$ that merely prevents 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 used for building the base*. For example, the separate address spaces usually accorded to process abstractions bring guarantees about storage integrity; type systems in modern programming languages, such as Java and C#, bring guarantees about how we can use certain variables.
- *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 (i) to, work correctly if that invariant holds prior to execution and (ii), upon termination leaves the data structure in a state satisfying the invariant.

Notice that $\nu Priv(B)$ 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 we can express the guarantees being defended by stipulating some proscribed set of finite-length "bad" execution prefixes, thereby casting the guarantees as safety properties, then we can enforce $\nu Priv(B)$ by inline reference monitoring.

Many policies are stronger than $\nu Priv(B)$. (Policies weaker than $\nu Priv(B)$ aren't interesting because they are not strong enough to prevent attacks on the base system's enforcement mechanism.) Selection of a single one of these stronger policies 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 a third party then would classify each extension's intent 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 extension's purpose and thus is probably more restrictive than the other polices. Note that although individual users might 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.

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 system's abstraction or principle. However, some abstractions are useful even though they cannot be implemented. Laszlo Belady's optimal page-replacement policy,[5] 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 access control policies. When compared with $\mu Priv_B(Ext, \Sigma_{Ext})$, we would consider a deployed policy inferior if it either admits additional attacks or incorrectly restricts the functionality of *Ext*.

This article not only revisited a classic security principle but also a classic abstraction: the reference monitor. Many forms of fine-grain access control that are impractical with traditional reference monitors become practical with inlined 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" $\mu Priv_B(Ext, \Sigma_{Ext})$ provides is the ultimate question. For the near future, however, it seems that practical protection for extensible systems is most easily obtained using policies that grant more privileges than would $\mu Priv_B(Ext, \Sigma_{Ext})$—the least privilege and more. □

Acknowledgments

This is a revised version of a paper written in honor of Roger Needham (1935–2003) for the conference "Roger Needham: 50 and 5" at Microsoft Research, Cambridge, England (17 Feb. 2003) held shortly before his death. The proceedings will appear in Springer-Verlag's Texts and Monographs in Computer Science. *Lorenzo Alvisi, Ulfar Erlingsson, Butler Lampson, Greg Morrisett, Andrew Myers, E. Gun Sirer, and Mike Schroeder provided helpful comments on drafts of that paper.*

The author is supported in part by AFOSR grants F49620-00-1-0198 and F49620-03-1-0156, 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 author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the US government.

References

1. J. Anderson, *Computer Security Technology Planning Study*, tech. report, ESD-TR-73-51, Electronic Systems Division, Hanscom Air Force Base, Hanscom, MA, 1974.
2. U. Erlingsson and F.B. Schneider, "SASI Enforcement of Security Policies: A Retrospective," *Proc. New Security Paradigms Workshop* (NSPW 99), ACM Press, 1999, pp. 87–95.
3. R. Needham, "Protection Systems and Protection Implementations," *Proc. 1972 Fall Joint Computer Conf.*, AFIPS Conf. Proc., AFIPS Press, vol. 41, 1972, pp. 571–578.
4. J.H. Saltzer and M.D. Schroeder, "The Protection of Information in Computer Systems, *Proc. IEEE*, vol. 63, no. 9, 1975, pp. 1278–1308.
5. L.A. Belady, "A Study of Replacement Algorithms in a Virtual Storage Computer," *IBM Systems J.*, vol. 5, no. 2, 1966, pp. 78–101.

Fred B. Schneider *is a professor at Cornell University's Computer Science Department, director of the AFRL/Cornell Information Assurance Institute, and the founding chief scientist for New York State's Griffiss Institute for Cybersecurity. In addition to chairing the National Research Council's study committee on information systems trustworthiness and editing* Trust in Cyberspace*, he is co-managing editor of Springer-Verlag's* Texts and Monographs in Computer Science*, serves as associate editor in chief of* IEEE Security & Privacy*, and participates on a number of journal editorial boards. He has an MS and PhD from SUNY Stony Brook and a BS from Cornell. He is author of* On Concurrent Programming *and co-author of* A Logical Approach to Discrete Math*. He is a fellow of AAAS and ACM, is Professor at Large at University of Tromso (Norway), and received an honorary Doctor of Science from University of NewCastle-upon-Tyne. Contact him at fbs@cs.cornell.edu.*