

capability to the statistics file. Instead of referring to the name of the file, the compiler would merely designate that capability when depositing the statistics. The capability both identifies the file and authorizes the compiler to write there. When producing the debugging output the compiler would merely refer to a capability provided by the invoker to the place he meant to hold that output. The same mechanism is used in each case—no ASCII character names are required, no authority checking mechanisms are executed. We must not only endow the compiler with authority over the **STAT** file but require the compiler to explicitly designate that authority. In this case there is no need for the compiler to know any textual file name.

Before we implemented the capability ideas, we feared that a system built on these principles would use most of the storage to hold these mysterious new capabilities. Instead it turned out that capabilities replaced so many other ad-hoc mechanisms that our capability-based systems were usually smaller than equivalent access-list based systems, because they unified not only various naming functions, but also made older basic security mechanisms largely unnecessary. That performance was excellent was a pleasant extra.

I think that the moral of this experience is that programs must be aware of their sources of authority and cite them as they invoke them. What better way than invoking a capability.

Some systems have tried to add capabilities to the traditional mechanisms only to sometimes suffer more from the combined disadvantages than benefit from the combined advantages. Our view is that capabilities must be the foundation of the system. We have carried out that program more completely in some ways in our implementation of the KeyKOS system [1, 2] than previous systems have. KeyKOS has directories and other such traditional operating system facilities—they are implemented and accessed, however, via capabilities.

KeyKOS provides patented facilities [3] to aid deputies (and defeat Trojan horses, viruses, and other related security threats), while also providing flexibility to meet a broad range of security policies—from government-style “orange book” policies [4] to useful commercial policies including those requiring the solution of the mutually suspicious users problem [5].

Bibliography

[1] Hardy, N., “KeyKOS Architecture,” Operating Systems Review, Association for Computing Machinery September, 1985. (Also available in an modified version as publication KL068 from Key Logic.)

[2] Rajunas, S.A., et al., “Security in KeyKOS,” Proceedings of the 1986 IEEE Symposium on Security and Privacy, IEEE.

[3] U.S. patent number 4,584,639.

[4] Department of Defense Trusted Computer System Evaluation Criteria, U.S. Department of Defense, DOD

5200.28-STD, December, 1985.}

[5] KeyKOS and Mutually Suspicious Users (KL108), 1987, Key Logic.

The Confused Deputy

(or why capabilities might have been invented)

Norman Hardy

This paper appeared in nearly this form in the Oct. 1988 issue of Operating Systems Review, pp 36:38

Bold face stuff should be changed for greater correspondence to Unix.

This is a nearly true story (unessential details have been changed). The events happened about 1977 at Tymshare, a company which provided commercial timesharing services. Before this happened I had heard of capabilities and thought that they were neat and tidy, but was not yet convinced that they were necessary. This occasion convinced me that they were necessary. It is an intricate scenario but such is the nature of computers.

Our operating system was much like Unix (™ of AT&T) in its protection structures. A compiler was installed in a directory called **SYSX**. A user would use the compiler by saying “**RUN /SYSX/FORT**”, and could provide the name of a file to receive some optional debugging output. We had instrumented the compiler to collect statistics about language feature usage. The statistics file was called **/SYSX/STAT**, a name which was assembled into the compiler. To enable the compiler to write the **/SYSX/STAT** file, we marked the file holding the compiler {**/SYSX/FORT**} with *home files license*. The operating system allowed a program with such license to write files in its home directory, **SYSX** in our case.

The billing information file **/SYSX/BILL** was also stored in **SYSX**. Some user came to know the name **/SYSX/BILL** and supplied it to the compiler as the name of the file to receive the debugging information. The compiler passed the name to the operating system in a request to open that file for output. The operating system, observing that the compiler had home files license, let the compiler write debugging information over **/SYSX/BILL**. The billing information was lost.

Who is to blame? What can we change to rectify the problem? Will that cause other problems? How can we foresee such problems?

The code to deposit the debugging output in the file named by the user cannot be blamed. Must such code check to see if the output file name is in another directory by scanning the file name? No—it is useful to specify the name of a file in another directory to receive output. Should the compiler check for directory name **SYSX**? No—the name “**SYSX**” had not been invented when this code was written. Indeed there might be a legitimate request for the compiler to deposit debugging output in some file in **SYSX** made by someone with legitimate access to that directory. Should the compiler check for the name **/SYSX/BILL**? No, that is not the only sensitive file in **SYSX**. Must the compiler be modified whenever new files are added to **SYSX**?—Surely not.

When the code was written to produce the output it was correct! What happened to make it wrong? The precise answer is that it became wrong when we added home files license to **/SYSX/FORT**. To determine this, however, would have required examination of every situation in which the compiler wrote a file. Even when we identify those situations it is not clear what to do.

Another indication of trouble was that the rules allowing a program to open a file grew more complex. The rules were suffering from the law that complex things grow more complex. Every time we added a clause enabling the opening of a file in a categorical situation we would introduce security problems in programs that had been secure. Every time we added restrictions to these categories we broke other legitimate programs. The last time that I wrote down the requirements for a program to open a file, it required fourteen Boolean operators (“and”s & “or”s)!

The fundamental problem is that the compiler runs with authority stemming from two sources. (That’s why the compiler is a confused deputy.)

The invoker yields his authority to the compiler when he says “**RUN /SYSX/FORT**”. (This is of course the tool of Trojan horses which is the companion problem in these access list architectures.) The other authority of the compiler stems from its home files license. The compiler serves two masters and carries some authority from each to perform its respective duties. It has no way to keep them apart. When it produces statistics it intends to use the authority granted by its home files license. When it produces its debugging output it intends to use authority from its invoker. The compiler had no way of expressing these intents!

The system was modified by providing a new system call to switch hats which could be used to select one of its two authorities. Note the increase in complexity! The compiler would then be able to use its home files license or the invoker’s license explicitly—in the later case, for example, saying “by the authority vested in me by my invoker I hereby request the opening of **/SYSX/BILL**” which would then properly fail. It soon became clear, however, that more than two “authorities” were necessary for some of our applications. A further problem was that there were other authority mechanisms besides access to files. Generalizations were not obvious and the modifications to the system were not localized. (Exercise for the reader: Show that access lists do not solve this problem.)

Another indication of poor design is that disparate mechanisms were necessary to arrange separately that the compiler (1) know what file to write on and (2) be authorized to write on that file. The crime was perpetrated through unintended application of the compiler’s authority over **SYSX** when writing the user’s data. (If you try to solve this problem without capabilities, remember that the file **/SYSX/STAT** must also be protected.)

The capability solution would endow the compiler with a direct