# Secure Applications Need Flexible Operating Systems

David Mazières and M. Frans Kaashoek
MIT Laboratory for Computer Science
545 Technology Square, Cambridge MA 02139

E-mail: `dm@lcs.mit.edu, kaashoek@lcs.mit.edu`

## Abstract

*As information exchange over wide area networks becomes an increasingly essential component of new applications, firewalls will no longer provide an adequate defense against malicious attackers. Individual workstations will need to provide strong enough security to contain malicious processes and prevent the domino effect of a pierced firewall. Some of the most commonly found security holes today result from the fact that simple operations can be surprisingly difficult to implement correctly on top of a traditional POSIX-like interface. We claim that by combining hierarchically-named capabilities, a novel generalization of the Unix user and group ID concept, with the low-level system calls of an exokernel operating system, we can achieve a system call interface flexible enough to avoid much of the complexity that often leads to security holes in discretionary access control operating systems like Unix.*

## 1 Introduction

The lack of flexibility in today's multi-user operating systems seriously hurts system security. In particular, inadequate system call interfaces force programs to run with more privilege than they should need. Moreover, coarse-grained kernel access control mechanisms fail to meet the protection needs of applications. This pushes many access control and validity decisions into over-privileged user-level software, where a cumbersome system call interface can vastly complicate correct implementation. A large class of security holes has consequently resulted from bugs in code to perform such seemingly simple tasks as opening a file only if it belongs to a particular user. In this position paper, we therefore argue that flexible operating systems can simplify the construction of secure applications, thereby decreasing the likelihood of implementation mistakes.

As the Internet continues to grow, information exchange over wide-area networks will become an increasingly essential component of new applications. Already, many applications fetch their help files and other data over the world wide web. The Quicken personal finance program, for instance, transparently downloads stock quotes over the network to update portfolios. The FreeBSD Unix operating system even installs and upgrades itself over the network.

Unfortunately, this need for seamless network connectivity directly conflicts with present-day network security policies. Such policies necessarily assume that a determined cracker can break normally configured systems attached to the Internet. Consequently, security-conscious sites must get their machines away from the wide area network. This approach typically means installing firewalls to keep outside attackers from exploiting the known deficiencies of internal networks. In the short term, these firewalls may hinder the deployment of distributed applications near sensitive data. Eventually, however, user demands will force sites to change policies, adopt more complex proxy servers, and perhaps even relax packet filtering. Such changes will render internal machines increasingly vulnerable to outside attack. Since the security of all machines behind a firewall depends on the security of the weakest machine, the appeal of wide-area applications will require better security for all machines.

We propose improving security by increasing the flexibility of operating systems, which we argue will both decrease the level of privilege needed to accomplish many tasks and simplify the code required to make access control decisions in trusted software. Our approach combines hierarchically-named capabilities, a novel generalization of the Unix user and group ID concept, with the low-level interfaces of the exokernel[3], an operating system that attempts to provide as few

This research was supported in part by the Advanced Research Projects Agency under contracts N00014-94-1-0985 and N66001-96-C-8522, and by an NSF National Young Investigator Award.

abstractions as possible in privileged software. Our scheme provides a single kernel access control mechanism for all resources, designed to be used directly by applications. This organization should make secure applications much easier to write by freeing applications from the need to precede system calls with access checks—a practice which often causes race-conditions in Unix and prevents trusted applications from safely reusing code developed for untrusted programs.

## 2 Why are there so many security holes?

Why are there so many vulnerabilities in today's discretionary access control systems? Before delving into the details of our own proposal, it makes sense to discuss some of the major sources of security holes in today's operating systems. We therefore identify six aspects of Unix systems that have made them particularly prone to security holes[1]:

1. **Insecure network protocols**

   Most of the network protocols in use today transmit secret passwords or filehandles in cleartext over the network, or rely on the source address of packets for authentication. Even one-time password systems such as s/key[4] allow an active attacker to hijack an authenticated TCP stream. Inexperienced attackers can easily obtain software packages for exploiting various insecure but widely-used network protocols.

2. **The use of C and libc**

   Many security problems stem from the use of C functions such as `gets()` and `sprintf()` that do not check the size of the output buffer and consequently allow malformed input to corrupt memory arbitrarily. Such problems surface with alarming frequency in places where they can be remotely exploited, for instance mail transport agents and web servers. Worse yet, however, people seem resigned to accept these problems in ordinary untrusted utilities. As an extreme example, a comment in *zlib*—a compression library used by the popular *gzip* compression tool (as well as in some kernel-level implementations of PPP!), recommends that applications catch segmentation fault signals because some forms of corrupt input cause the library to "go nuts." Though *gzip* cannot be invoked directly by a remote attacker, people commonly uncompress foreign files from unknown sources, making this a potentially serious vulnerability.

3. **Violation of the principle of least privilege**

   The principle of least privilege states that a process should have access to the smallest number of objects necessary to accomplish a given task. Unix basically only provides two privilege levels: "root" and "some user." Much of the system software must consequently execute as root, providing a large selection of complex daemons in which an attacker can search for flaws. While people have long complained about the need to run a great deal of software as root, equally problematic is the requirement that a user give every program he runs access to all of his files. In the previous example, for instance, the least privilege *gzip* needs is read access to the input file and write access to the output file, but Unix provides no way to restrict access in this way. Thus a subverted *gzip* has the potential to cause great damage.

4. **A namespace decoupled from underlying files**

   The decoupling of file names from actual files in Unix has lead to numerous race conditions, called Time of Check to Time of Use or TOCTTOU bugs[1]. Consider an extremely common example: Many sites periodically delete old files in the scratch directory `/tmp`. The command used to do this usually resembles this:

   ```
   find /tmp -atime +3 -exec
          rm -f -- {} \;
   ```

   As shown in Figure 1, however, an attacker can trick this command into deleting the system password file by creating and moving a file called `/tmp/etc/passwd`. The *find* program uses `lstat()` to check that `/tmp/etc` is not a symbolic link, but by the time it runs *rm* to delete `/tmp/etc/passwd`, the attacker has had time to create a symbolic link in the place of `/tmp/etc`[2].

5. **System calls that use all available privilege**

   The Unix kernel bases access control decisions on all privileges available to a process. If a privileged program wishes to perform an operation only if a particular user or group can do so, it must separately check permissions beforehand, a potentially tricky task given the risk of TOCTTOU problems as described in the previous section. Such checks

[1] We have chosen to focus on Unix rather than Windows NT, though both systems are popular in networked environments. We believe that many aspects of this analysis apply equally to NT, but the widespread availability of source code for Unix utilities and operating systems has allowed much better analysis of Unix's vulnerabilities than NT's.

[2] Note that in practice, this race is much easier to win than it might appear. By creating a large number of files in `/tmp/etc`, the attacker can make the `readdir()` call take a long time.

| Root: | Attacker: |
|---|---|
| | mkdir (“/tmp/etc”) |
| | creat (“/tmp/etc/passwd”) |
| readdir (“/tmp”) | |
| lstat (“/tmp/etc”) | |
| readdir (“/tmp/etc”) | |
| | rename (“/tmp/etc”, “/tmp/x”) |
| | symlink (“/etc”, “/tmp/etc”) |
| unlink (“/tmp/etc/passwd”) | |

Figure 1. Exploiting a typical garbage collector of temporary files.

are also cumbersome to perform in conjunction with the C standard I/O library, so that programmers sometimes neglect them. As an example, the NetBSD Unix *at* command runs as root to queue jobs in a protected spool directory, but can also take its input from a user-specified file. Until recently, it was possible to read any file on the system by specifying it as an argument to *at -f*.

6. **No process to process authentication**

   In order for users to perform privileged actions in a controlled way (for instance editing the password file to change their own passwords), Unix allows “setuid” programs that take on the privilege of a user other than the one invoking the program. In practice, setuid programs have caused considerable security problems because of the large amount of untrusted state they inherit from the invoking user. Until recently, for example, any user could trash the sendmail alias database by setting his maximum file size to 0 and running *sendmail -bi*. This command causes sendmail to rebuild the alias database. However, sendmail did not originally anticipate inheriting a small file size limit, so hitting this limit caused it to die and leave a truncated database.

   As an alternative to setuid (and a useful mechanism for other purposes), users might change their passwords through client-server interaction. Servers don’t inherit any state from their clients, which makes them quite a bit safer than setuid programs. Unfortunately, however, sockets—the usual Interprocess Communication (IPC) mechanism in Unix—neither provide a server with any assurance of the caller’s identity nor allow the server to take on the caller’s identity. Setuid programs often need the identity of the invoking user to perform access checks and logging, and they sometimes even need to revert to the the invoking user’s identity.

Of the categories above, only network protocols are being widely addressed and seem likely to improve in the near future. As for C and libc, there are plenty of open research questions in programming languages which we don’t intend to address here, except to the extent that the principle of least privilege can prevent subverted programs from doing much damage. We do, however, plan to attack problems 3–6 directly. A great many security holes can be attributed to these problems, yet typically such vulnerabilities get addressed only through specific patches on a case-by-case basis. Though one can indeed work around specific problems this way, the code required is generally quite tricky. For this reason, patches have trouble keeping up with newly discovered security holes. Our goal is therefore to make such vulnerabilities easy to avoid and the code for doing so simple and straightforward.

## 3 Security and Flexibility

With some common sense and a few ideas from previous systems, we can develop operating systems much more conducive to security than any widely used operating system today. We can’t make application programmers more careful about security, but we can build operating systems that make it easy to avoid common security holes. Under a better operating system, race-free implementations of simple tasks would never require complex code; most bugs would not automatically lead to a root shell; finally, a uniform kernel access control mechanism would be flexible enough to replace any non-atomic application-level access control decisions. We therefore propose developing a system call interface based on the following realizations:

- The kernel should provide authenticated IPC, and should allow credentials to be passed between processes. Not only will this permit the elimination of setuid programs, but, as we shall see below, passing credentials can dramatically reduce the level of privilege needed by a number of login-like applications.

- Access control decisions and system calls should operate in terms of low-level file objects rather than mutable path names that can change and lead to TOCTTOU bugs. Thus, name to file translations should move to user-space to give programmers control over all aspects of the process including such details as whether symbolic links are followed or filesystem boundaries crossed. The exokernel operating system[3] already provides applications with such control, and is consequently the platform on which this work is based.

- Trusted applications should rely on the kernel to make access control decisions. The kernel should do so atomically with each system call, and should base the decision on a universal access control mechanism visible to and consistent across all applications. Again, this will help prevent TOCTTOU bugs. If the kernel provides access control mechanisms flexible enough to capture an application's intent, the application will no longer need to precede system calls with tricky, non-atomic sanity checks.

- As a corollary to the previous point, applications should explicitly specify the credentials with which they intend to authenticate each system call. For instance, it should be trivial for privileged software to make system calls on behalf of unprivileged users. The kernel should never assume default credentials, as this would prevent code that uses implicit credentials from being reused in privileged applications. Though previous systems (for instance Taos[7]) have used such explicit credential arguments, POSIX has nothing like this and Windows NT requires credentials to be assigned per thread rather than for each system call.

- All applications, even unprivileged ones, should rely on the kernel's universal access control mechanism to achieve safe sharing of resources. If, however, one access control mechanism is to meet most application needs, it must be flexible enough that it allows applications to define their own notions of identity. This means allowing on-the-fly creation of new principals or credentials under which mutually distrustful applications can safely and easily share resources.

All but the last of these points should be relatively straightforward. Allowing untrusted applications to create new principals on-the-fly, on the other hand, could seriously complicate the issues of accounting and controlling buggy programs. For instance, a user should clearly not have the ability to create a process he cannot kill, otherwise debugging an application that takes on newly-created credentials might require constant intervention on the part of an administrator. Though one can imagine several ways of avoiding unkillable processes and unfreeable resources, we propose *hierarchically-named capabilities*, a novel protection scheme with a simple and efficient implementation. Despite the name, these capabilities more closely resemble a generalized form of Unix user and group ID than traditional capabilities[2]. Our terminology and the hierarchical naming we use come from the VSTa[6] operating system, though exokernel capabilities function differently from VSTa ones.

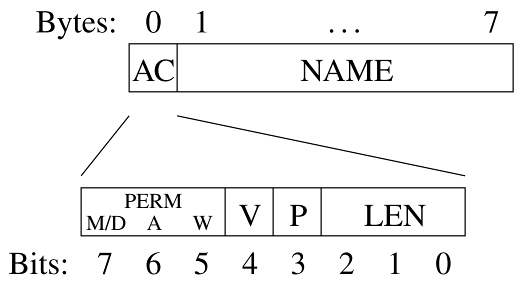


Figure 2. Structure of hierarchically-named capability

Figure 2 shows the layout of a hierarchically-named capability. A capability is 8 bytes long. The first byte records the properties of a capability and the other 7 bytes its name. The properties in the first byte include the length of the name (from 0 to 7 bytes), a valid bit, a pointer bit (for extended ACLs), and three permissions bits: modify ACL/duplicate capability, allocate resources, and write. If one capability's name is a prefix of another's, then the shorter capability *dominates* the longer one, meaning that the shorter capability can automatically authorize any request the longer one can.

The kernel maintains a list of capabilities owned by each process, and an access control list (ACL) of capabilities allowed access to each object in the system. When a process requests access to a particular resource, it must explicitly specify which of its capabilities it intends to gain access with. The kernel then traverses the resource's ACL. If it finds an ACL entry that either matches or is dominated by the designated capability, and if the appropriate permission bits are set both in the process's capability and in the access control list entry, then the request is granted. Otherwise, it is rejected.

(*a*)

| csh | |
|---|---|
| dm | `daw 1.115.0` |
| users | `d-w 2.96.0` |
| exopc | `d-w 2.127.0` |
| other | `d-w 0.0.0.0.0.0.1` |

(*b*)

| Mobile Code | |
|---|---|
| dm.x | `daw 1.115.0.5.0.0` |

(*c*)

| file | ACL |
|---|---|
| `~dm` | `maw 1.115.0, --- 0.0.0.0.0.0.1` |
| `~dm/xfiles` | `maw 1.115.0, -aw 1.115.0.5.0.0` |

Figure 3. Safe execution of untrusted, unverified mobile code. The letters `d`, `m`, `a`, and `w`, represent the corresponding permissions bits in the hierarchically-named capabilities. (*a*) The list of capabilities granted upon login to user `dm` in groups `users` and `exopc`. These correspond to 16-bit Unix uid 115 (tagged with the initial byte `1` to avoid conflicting with gids of the same number, and represented in little-endian format as `115.0`), and gids 96 and 127 (tagged with the initial byte `2`). (*b*) A restricted list of capabilities given to the untrusted code. (*c*) Directory permissions which ensure that the mobile code can only read/write files in `~dm/xfiles`.

Applications can create or *forge*[3] new capabilities at will, but a new capability must be dominated by an existing one. This allows mutually distrustful applications to share resources under newly created capabilities. Using the new capabilities, applications can manipulate shared resources without fear of being tricked into manipulating more sensitive objects. Because originally issued capabilities are a prefix of all subsequently created capabilities, this scheme also allows for full accountability of all resource usage. A prefix of the capability that owns any resource will show exactly who is responsible for having allocated the resource, and that user will always be allowed to reclaim the resource. Some examples will illustrate the use of hierarchically-named capabilities.

### Example: Mobile Code

There are two prevalent types of mobile code today: Java applets and Active-X binaries. Java applets rely on a type-safe language to prevent untrusted code from doing any damage. Unfortunately, one has to trust a great deal of software, from the byte-code verifier to class libraries, in order to execute untrusted Java code. Furthermore, there are obvious performance and functionality penalties from forcing people to use an interpreted, garbage-collected language. Active-X solves the performance problems by shipping digitally signed raw x86 binaries around as mobile code. However, one is required to trust the author of an Active-X binary before running it.

An exokernel with hierarchically-named capabilities, however, would allow people to execute untrusted binary machine code without giving it access to most data on the system. Figure 3 illustrates how such a system might work. A user `dm` logs in and acquires capabilities corresponding to his user and group IDs. Since exokernel ACLs have no notion of world readable or writable, he also acquires capability `0.0.0.0.0.0.1` that grants the equivalent of world permissions on Unix files. After this user's browser downloads untrusted mobile code, it can use capability `1.115.0` to forge a new capability, say `1.115.0.5.0.0`, and create a process for the mobile code which contains only that capability. If permissions on files and directories are set up as in Figure 3, the operating system will prevent the mobile code from accessing any files on the system except those in a special mobile code directory. Even world readable files will be inaccessible to the mobile application, as it would need capability `0.0.0.0.0.0.1` to access those.

### Example: Unprivileged Login

On a Unix system, any program that authenticates and grants permissions to users must execute with all-powerful root privileges. Examples of such programs include *login, su, ftpd, recexd, rshd, sshd, popd, cvs,* and more. A bug in any one of these programs would grant

[3] The term *forge*, used here to be consistent with VSTa, means "form or bring into being." It in no way implies any kind of counterfeiting. The kernel maintains the list of capabilities owned by a process in protected memory, so that applications have no way of faking capabilities.

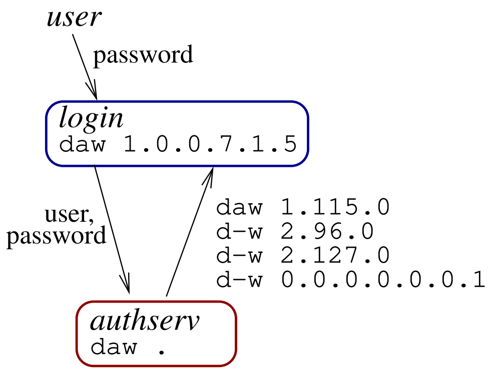


Figure 4. An unprivileged login program acquiring privileges through the authentication server.

immediate root privileges to an attacker. Given a mechanism for granting capabilities, however, a login process no longer needs any special privileges beyond the ability to read and write a terminal device or network connection. Instead of running as root, login can begin execution with a single, unique, and completely unprivileged capability. As illustrated in Figure 4, once login has a username and password, it can acquire appropriate privileges by contacting an authentication server. Thus, one privileged authentication server can replace a large number of privileged programs. With this scheme, the damage caused by a bug in login is limited to actions which can be taken by its capabilities. Since login begins with a unique capability, the worst an attacker can do is consume memory and CPU time. No data on the system could be disclosed or tampered with. Unprivileged login programs are nothing new, particularly to distributed systems such as Kerberos[5], but this simple authentication server could have saved a large number of security holes in Unix.

### Example: Using the Kernel for Access Control

The *ssh*[8] secure login program, a privileged setuid-root application, needs to read a protected file as root. Though the system must trust *ssh* to have root privileges, *ssh* itself needs to create and write files in the home directory of the invoking user, and must ensure that it does not accidentally overwrite some file not in fact accessible to that user. Between the race conditions caused by writing the files as root and the risk of the debugger attaching to *ssh* if it drops privileges, the author found no choice but to structure *ssh* as three processes communicating over pipes. If system calls had allowed explicit credentials, however, writing a file as an unprivileged user would have been trivial.

## 4 Summary

There is nothing inherently difficult in many of the tasks whose implementations cause security holes today. Thus, jumping through hoops to create correct implementations on top of a Unix-like system-call interface just doesn't seem to be paying off. We need to design operating systems that don't make writing secure code such a difficult task, and that contain the damage caused by buggy applications. In this paper, we have argued that hierarchically-named capabilities can combine with a low-level exokernel interface to achieve precisely this goal. Specifically, the resulting operating system can minimize the amount of privileged code required on a system, simplify the correct code for many operations, and help privileged programs reuse ordinary code.

## References

[1] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

[2] J. B. Dennis and E. C. V. Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[3] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, 1995. ACM.

[4] N. M. Haller. The S/KEY one-time password system. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, February 1994.

[5] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 USENIX*. USENIX, 1988.

[6] A. Valencia. An overview of the VSTa microkernel. from `http://www.igcom.net/~jeske/VSTa/vsta_intro.html`.

[7] E. P. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, 1994.

[8] T. Ylönen. SSH – secure login connections over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42, July 1996.