Watchdogs

Mark Kegel
Marshall Pierce

2006-04-13

Abstract

Watchdogs are an unusual and flexible implementation of access control. A watchdog is executed when its associated filename is accessed, and can permit or deny access in addition to optionally filtering the contents of the file. This can be used for on-the-fly compression, or to simulate files that don't actually exist on the filesystem, in addition to implementing complicated access control policies that cannot be expressed with a simple access control matrix or list framework.

1 Introduction

Access control is a system to regulate privileges between different users on a computer system. The goal of access control is to protect sensitive components of a system from both unintentionally harmful and malicious users. The definition of "protection" is site-dependent: some sites may care about data integrity, while other sites may be more concerned with privacy, so the access control systems in place at each site would presumably be targeted towards their respective needs.

2 Access Control Types

There are three types of access control in normal use. The next three sections discuss each type.

2.1 Mandatory Access Control (MAC)

Under MAC, resource owners do not control the access privileges for resources. Instead, the system, through the system administrator or a special rule-set, decides what privileges to assign to a file. MAC is most often used when maintaining the confidentiality and security of data is of paramount importance. Military and other government institutions use MAC to prevent classified data from being accessed by personnel without the proper security clearance, and to prevent data from being moved to a lower security clearance level even when being transferred from one computer to another.

MAC works by assigning a label to each object. The label may describe who owns the file, the security level of the object, and any other information that needs to be associated with the object. A good example of MAC in use is Biba's integrity model: enforcing the data flow rules would be done by the system, not by the resource owners.

A more familiar example of MAC, though one that does not use a label, is file ownership under UNIX. Regular users will always own the files they create and once created a regular user may not change the ownership of a file they own to a different user. Only the superuser may change a file's ownership. If the system did not enforce this rule, users would be able to execute arbitrary code on the system by changing the owner on a setuid binary (which runs with the privileges of the file's owner) to root.

2.2 Discretionary Access Control (DAC)

Discretionary Access Control, or DAC, is a type of access control familiar to most users. DAC allows the resource owner to define the privileges that that resource has. Most access control schemes in use have some sort of DAC. In UNIX, the file owner can change who is allowed to read, write, or execute a file. In Windows, a user can toggle whether a file is hidden and whether it is read-only.

2.3 Role-based Access Control (RBAC)

Role-based access control, or RBAC, is a relatively recent development in access control. RBAC differentiates itself from MAC and DAC by adding in the concept of a role (though both MAC and DAC can be implemented using only RBAC). A role is a set of allowed operations. When users are assigned a specific role they are granted permission to perform any operation in the set.

RBAC is, in general, a much easier system to manage than either MAC or DAC. Under MAC and DAC, only resources are labeled. RBAC is fundamentally different: users themselves can now be assigned different labels. In this case, the label is the role that the user is currently fulfilling. This makes managing users much easier, as only the user's label must be changed and not the labels of all the resources in the system when a user's access level needs to be adjusted.

3 Access Control Implementations

There are many different access control schemes that have been implemented over the years. This paper discusses some of the most best known schemes, and their various drawbacks, limitations, and benefits.

3.1 Access Control Matrix (ACM)

The Access Control Matrix, or ACM, is the most generalized access control scheme. The ACM is a two-dimensional matrix in which the rows of the matrix are the subjects (users or processes) present in the system and the columns of the matrix are the various objects, or resources, present in the system. When a subject attempts to access (or read, or write, etc.) a file or other system resource, a lookup is made in the ACM in the appropriate row and column. The element is then checked to see if the user has the appropriate permission. If so, then the subject is allowed to access the resource, if not the subject is denied.

Overall, the general ACM scheme is not feasible to use. While simple in concept, the ACM quickly gets out of hand when moving beyond a few subjects since for each subject in the matrix, each element must have a relevant privilege for that subject. The general ACM model is also counterintuitive in that any given object may have multiple owners. Because each element is independently defined, there is nothing in the model to prevent there being multiple subjects with the Own privilege in a given object column. The complexity and size of the table grows at $O(n \times m)$ where n is the number of objects and m is the number of subjects.

Lastly, there is no way to verify whether a given ACM is secure. While this is not a failing in itself, it does help illuminate the kind of complexity inherent in this apparently simple model. The true reason a lack of verifiability is bad is that when a system administrator could not easily comprehend the structure and behavior of given ACM, they would need to rely on automated verification tools. However, because such an algorithm cannot exist, system administrators are left with a model they cannot easily use and that cannot be verified as secure. In this light, the lack of verifiability is a fatal flaw.

3.2 UNIX Permissions

The general ACM model can be pared down into the more familiar UNIX permissions system. The UNIX permissions system is a much simpler variant of the ACM model. First, UNIX permissions are fixed. Each file has a set number of bits in which to describe whether the file can be read, written, or executed by the owner, group, or the universe of users; the sticky bit, setuid, and setgid bits; and two integers to record the group and user ids. While this is not as flexible as a general ACM, the loss in flexibility is made up for by the gain in comprehensibility. Simple tools can be written to check for possibly dangerous activity like changes in a systems' setuid executables, and an administrator can easily browse file permissions for obvious errors.

UNIX permissions have withstood the test of time and are not significantly altered from their original inception. Nonetheless, UNIX permissions do have some major problems. The primary issue is the lack of expressibility for both users and system administrators. For example, say Bob would like Alice to be able to read a file. Bob can accomplish this by having a group added to the system containing only Bob and Alice, and then assigning group read privileges. This is tedious and inconvenient since only the system administrator can add groups to the system. Alternately, Bob could give universal read access to the file. This has the downside that now every user on the system will be able to access the file, not just Alice. In many cases this is unacceptable.

3.3 Access Control List (ACL)

Another variant of the ACM is ACLs. In an ACL, a resource can be assigned an extensible list of privileges pertaining to any group or user on the system. Though technically equivalent to an ACM, ACLs are much simpler and easier to comprehend, since the full list does not need to be specified. A partial list can be given, with default options available for the other users and groups via standard UNIX permissions. Such an option is not available in the ACM model.

The real-world usefulness of ACLs has yet to be fully determined. The limitations inherent in the UNIX system make ACLs an attractive option, yet keeping the complexity of the system down remains a concern. However, ACLs are not so complex as to be completely unmanageable. Both UNIX and Windows have implementations of ACLs.

4 Watchdogs

Watchdogs are a special breed of access control scheme. Most schemes attempt to be algorithmically verifiable in some way, but watchdogs, by their very nature, are not. The basic idea behind a watchdog is to supplement the table lookup, done in, say, an ACL, with a call to a function. The function's result will then override the permissions specified by the ACL.

To keep things simple, the called function resides in a process that the kernel starts and manages. When an access request comes in, the appropriate watchdog process is started (i.e. the one designated to guard the particular file or directory). The process is passed the user credentials and operation, such as an open. It then does a modicum of processing, returning either success or an error code upon exit. The kernel then uses the return code to enforce the proper behavior, either denying access or allowing it. Optionally, the watchdog can choose to process the contents of the file, or even simulate the contents of a file, similar to the `/proc` information produced by the Linux kernel.

A watchdog is the most general of these various access control systems. It can clearly implement ACLs, ACMs, etc by simply reading the access control data off of the disk and controlling access appropriately, but it can also do many other things: allowing access on every even minute, throttling I/O usage at peak hours, on-the-fly encryption, etc.

4.1 Drawbacks

Watchdogs suffer from one major drawback: performance. In comparison to regular UNIX permissions, a watchdog is going to be significantly slower. In a conventional access control scheme, all the information necessary to enforce policy can be accessed from within the kernel. In a watchdog guarded system, however, the kernel must start a new process *for each individual system call*, leaving the kernel context to fetch policy. The overhead in creating and starting new processes is high, and the IPC mechanisms in use are slow. This is less of a problem than might be suspected, though, since few files need the complicated access control that watchdogs offer, and the processes can be kept alive for frequently used files, which reduces the overhead significantly.

Ensuring that the watchdogs themselves are secure is also problematic. It is clearly critical that any user's watchdogs cannot have any adverse affects upon the system (by, for instance, corrupting the memory of another user's watchdog and thus compromising the other user's security) and that they implement the policy as they are intended to. Accurate representation of security policies requires careful coding, and always will, but the damage that watchdog execution can do to the system can be greatly constrained by forcing users to implement their watchdogs in a sandboxed language like Embryo, which does not have the capability to do many potentially harmful activities like opening sockets, etc.

4.2 Potential

A watchdog-only implementation is probably not practical to use. The drawbacks (complexity, performance) are too great. However, when combined with a standard UNIX permission or ACL scheme, the additional flexibility would be a tremendous boon to system administrators and users alike.

Watchdogs make a number of difficult tasks much easier to implement. Consider a government system where every transaction must be monitored. A watchdog could easily log the user, access type, time, and return code. This, however, is only the most basic use of such a system.

Watchdogs could be used to implement a form of full intrusion detection. Consider this scenario: employee of three years Evy Everson has never logged in remotely, Evy has SECRET clearance, and she almost never accesses files outside her home directory. A watchdog would know this about Evy, as well as other usage history statistics. Suppose that Evy logs in remotely for the first time ever and attempts to access several files outside her home directory. She does not type either her password or username incorrectly, and so logs in on the first try. What should the system do?

In a normal system, because Evy logged in with her username and password the system will not pay any more attention to Evy than any other user. A watchdog, however, may be able to use heuristics to detect such abnormal behavior. In this scenario, the user claiming to be Evy would probably have their activity monitored, and if things became too suspicious may have their account frozen. If in fact Evy's account had been hijacked, steps could be taken to mitigate the damage done. This is, of course, just one example – it is easy to see that watchdogs could be usefully applied to myriad other tricky access control issues.

5 FreeBSD MAC Framework

The release of FreeBSD 5 and the corresponding architectural changes heralded the addition of MAC and ACLs from the TrustedBSD project. TrustedBSD is essentially the security research arm of the FreeBSD project. It is funded by DARPA, the NSA, NAL, McAfee, and other groups with a strong interest in security, and has developed not only MAC and ACL support but also SEBSD (an implementation of FLASK and SELinux type enforcement), a kernel-level audit framework, and OpenBSM support, to name a few.

FreeBSD's MAC implementation allows developers to write kernel modules that register callback handlers for anything from polling a vnode to toggling swap partitions to creating a POSIX semaphore. Any time any of those events happens, the handler registered by the module will be called, and the kernel will use the return code of the handler to decide if access should be granted or not. An example handler (in this case, the default stub handler called when a user tries to listen to a socket) follows.

```
static int
stub_check_socket_listen(struct ucred *cred, struct socket *so,
    struct label *socketlabel)
{
    return (0);
}
```

The `struct ucred` holds information about the user's credentials (real and effective user id's, etc), `struct socket` is self-explanatory, and `struct label` contains module-specific label information for that specific socket. (That is, multiple modules may each maintain different label information for that specific socket, so `*socketlabel` must hold them all.) In this stub handler, it simply approves access always (i.e. returns 0), but clearly very complicated policies could be enforced by implementing a more complicated handler. The MAC framework comes with several modules that implement various security models, including Biba and Low-watermark integrity policies, MLS, process partitioning, TCP and UDP port ACLs, and more.

The label support in the MAC framework is somewhat complicated, and bears further explanation. Not all MAC modules use labels (for instance, mac_ifoff, which lets the administrator completely turn off a given interface, has no need for labels, but mac_biba, an implementation of the Biba integrity model, obviously does), so whether or not a module needs to use labels is specified when registering the module. The `label` struct contains an array that actually holds the label information. The size of the array is specified with a cpp `#define` in `/usr/src/sys/sys/_label.h` that defaults to 4, meaning that up to four modules that use labels can be active at once.

5.1 POOCH (Poorly Organized Overly Complicated Hack): A Watchdog Implementation

Our idea for POOCH was to investigate the viability of implementing a watchdog via FreeBSD's MAC framework, but unfortunately our progress was severely hampered by the MAC framework. It is only lightly documented and has basic architectural flaws that make it very difficult for us to implement the kind of watchdog we had in mind.

We were able to quickly get some basic filesystem monitoring functioning that logged every read operation on a vnode (what UID made the read call, and what type of vnode was accessed), but since a vnode is essentially just an abstraction of an inode, there is no way to map a vnode to a filename, so there is no way to call specific procedures based on what filename was accessed. Furthermore, we did not find a good way to manage the transition from running in kernel space to executing a process in user space. While there are ways around this, namely embedding an interpreter for a language like Small or Embryo inside the kernel module, we could not implement them given our time constraints.

The label support in the MAC framework was also problematic. The limit of 4 modules that can use labels would not normally be a problem, since we obviously are only trying to implement one module, but unfortunately the code to re-use slots is broken, so our module (which uses labels) could only be loaded 4 times before failing mysteriously, which required a reboot to reset. After tracing down the relevant code in `kern/kern_mac.c`, we partially worked around the issue by simply using the maximum permissible value (30), rather than rework code deep in the internals of the kernel. The kernel uses a 4-byte bitfield to represent the label slots, but due to compiler warnings when using 31 as the maximum number of slots (which translates to the maximum number of left-shift operations) due to overflow in certain situations, we settled on 30. While clearly better than 4 module loads between reboots, it is still inconvenient to have to reboot often during development.

Furthermore, the `label` struct only seems to allow for numeric labels, which does not mesh well with what we had in mind for our watchdog. The chief developer on TrustedBSD, Robert Watson, acknowledged the architectural problems in the MAC framework when we contacted him, and explained that MAC is undergoing a major rewrite that is due for release in FreeBSD 7.x. Given these constraints, we were unable to meet our original goals, but once the MAC framework becomes more mature, a watchdog implementation would be worth investigating again.

References

[1] Brian N. Bershad and C. Brian Pinkerton. Watchdogs - extending the unix file system. *Computing Systems*, 1(2):169–188, 1988.

[2] Sylvia Osborn. Mandatory access control and role-based access control revisited. In *Proceedings of the Second ACM Workshop on Role-based Access Control (RBAC '97, Fairfax, VA, Nov. 6 –7)*, pages 31–40, New York, NY, 1997. ACM Press.

[3] Tom Rhodes. *The FreeBSD Handbook*, 2006.

[4] Various. IRC channel #freebsd on efnet, April 2006.

[5] Robert Watson. Source of `mac_biba` in FreeBSD kernel, 1999-2002.

[6] Robert Watson. Source of `mac_partition` in FreeBSD kernel, 1999-2002.

[7] Robert Watson. Personal communication, April 2006.

[8] Robert Watson and Chris Costello. *The FreeBSD Architecture Handbook*, 2006.

[9] Robert Watson and Various. IRC channel #trustbsd on efnet, April 2006.