The Flask Security Architecture

A Flexible Mandatory Access Control Mechanism For Use in Multiple Secure Systems

Jeffrey Barr

Professor Tom Perrine
San Diego State University
Computer Science 574
6 December 2002

Flask is a flexible access control security architecture that supports dynamic security policies. Flask creates this flexible support by separating the security policy decisions from the actual enforcement of the security policy. Flask "describes the interactions between subsystems that enforce security policy decisions and a subsystem which makes those decisions, and the requirements on the components within each subsystem" [11]. The Flask security architecture also can be transferred to multiple operating systems as well as other systems that require a security policy decision-making and enforcement system.

Flask is an improvement of the security architecture previously developed for the Distributed Trusted Operating System (DTOS) and described in [7]. Flask and DTOS separate security policy decisions from security policy enforcement through the use of a security server that makes the decisions and object managers that are in charge of the enforcement for individual tasks. This separation allows Flask to support flexible mandatory access control (MAC).

This paper will describe the advantages of the Flask security architecture as well as some of the issues with this architecture. The specific advantages that will be discussed in this paper are the separation of access control decisions from enforcement including security policy flexibility and revocation of previously granted access rights, the caching of previously determined access rights, and the capability to implement the architecture in multiple operating systems as well as other systems that require a security architecture. The main issue to be discussed is the complexity of the implementation.

Access control decision and enforcement separation:

The capability of Flask to separate its security access control decisions from enforcement of those decisions is the central feature of the security architecture and the capability from which many other advantages are derived. Flask utilizes a separate security server to make access decisions, while individual object managers enforce those decisions. The security server provides "access, labeling, and polyinstatiation decisions" [11] between subjects and objects. Access decisions determine whether permission is granted based upon labels of two separate entities (normally a subject and an object). A labeling decision determines the label of an object based upon the label of related subjects and objects. Polyinstatiation decisions determine "which member of a polyinstatiated set of resources should be accessed for a particular request" [8].

Objects that are under the control of the security policy are labeled with a security context, which is a policy-independent data type that can be interpreted by applications or users that understand the security policy. The security context is mapped to a security identifier (SID) in the security server that is utilized by the object managers. It is the SID that allows communication between the security server and the object managers while maintaining the separation of decision-making and enforcement capabilities. The SID allows the object managers to run independent of the content of the security context. The security server computes new security contexts. A new SID is assigned each time an object is created and is mapped by the security server to a specific security context. One other feature of the separation of the SID and security context is that knowledge of an objects SID does not mean permission has been granted to know the security context of that SID. Therefore, a permission check does not require knowledge of the level of security required for the object [8,11].

Whenever a security decision is needed, a request is made to the security server with the SIDs of the subject and the object and their object class. The object class defines the specific services that are supported by a particular object and contains permissions associated with each particular class. This allows further categorization of objects so that access to a particular type of object can be distinguished even more. For example a file can be categorized into device files and regular files, which would have different access rules. Also, if a subject wishes to limit its privileges, it is allowed to use a different SID as its effective SID during inter-process communications (IPC). The permission to use the effective SID is also decided by the security server [8,11].

The separation of security policy decisions and enforcement was the central design feature to Flask that made the system advantageous to use. Three of the major advantages of Flask over other operating system security

implementations are directly due to this separation of policy. These advantages, security policy flexibility, the ability to revoke previously granted access rights, and finer grained control of access rights are described below.

Security Policy Flexibility:

The security server is used only to determine security decisions, independent of the subject and object that require the results of the decision. These decisions can be based upon a combination of MAC implementation schemes that could include multi-level security (MLS), type enforcement (TE), and dynamic role-based access control (RBAC).

However, the decision of which types and/or combinations of MAC is determined not by the Flask architecture but by the specific implementation utilizing Flask. In fact, while the prototype system utilizing the Fluke micro-kernel implemented four security policies to determine access decisions [11], SELinux implemented a combination of three policies, one of which was optional [5]. The overall security policy is defined as a container of these different policy characteristics (i.e. a type for type enforcement, a level for MLS, etc.).

Also, this security policy can be changed at any time as long as one has access to the security servers interfaces for changing the policy. Access is granted in the same manner as access to any other resource, through the use of the security servers own object manager that enforces the security decisions of the security server. The security policy is defined by a combination of the security server code and a policy database. Either the code or the database can be changed in order to change the security policy without any changes to the object managers that enforce the policy [11].

The fact that different implementations can implement different security policies depending on their individual needs is a dramatic improvement over previous secure operating systems architectures. While MULTICS was an extremely secure operating system, eventually achieving an Orange Book Class B1 evaluation, it had a specific MLS implementation based upon the Bell-LaPadula model. This inflexibility limited the usage of the MULTICS to those systems, such as military, that had specific needs of an MLS based system and that were not limited by the liabilities of MLS such as its lack of dynamic separation of duty or least privilege [5,6].

Also, not all security needs require or want a MLS based policy. For example, a RBAC system as defined in [2] may be much more appropriate in private organizations such as a hospital where the roles of doctor, nurse, etc. may restrict the operations allowed on the same patient data, but allow multiple roles to access that data. Utilizing Flask, the security needs of the specific group or organization utilizing the system can define its own security requirements and configure the security server accordingly.

Furthermore, the network security section of the Flask prototype described in [1] is more flexible than common network protection systems such as firewalls. This is because these systems only offer solutions that either allow or block packets from passing through the system without any finer grained control. Flask makes each node/machine responsible for controlling I/O via the operating system. Therefore, the node security now has partial trust and can implement access control checks for various types of access requests. In addition, since the security is implemented as part of each machine, the security architecture can help prevent insider attacks, making the system even more powerful.

Revocation of previously granted access rights:

The security policy flexibility in Flask is not limited to the types of security implementations allowed. Flask implemented dynamic security policy requirements so that it could enforce a new policy when the policy changed instead of a policy that is obsolete. This dynamic security policy requirement came out of a Fundamental Property that Flask defined as: “Any time a service is provided by the system, that service is allowed according to the current security policy” [10]. In other words, services require the current policy to be utilized instead of possibly older ones already in use. Thus, the Fundamental property is the reason revocation of a previously granted access right is both a capability and a requirement of the Flask security architecture. This requirement is fully specified as part of the Dynamic Security Policy Study in section 6 of [10].

In the Flask architecture, the security server has two views of the security policy it maintains, the current security policy and the policy currently being implemented by the object managers. When no security policy changes have occurred, the current policy of the security server and the policy being implemented should coincide. It is when the current policy changes that revocation of previously granted access rights might be required.

The security policy is represented by two security contexts and the permission being sought between them mapping to an access decision. Both positive and negative access decisions are saved. If any permission currently in the security server's current permission set needs to be removed, the security server will first check the current implementation to see if this permission is being used by any of the object managers. The security server and object managers are then required to communicate with each other to revoke the permission by ensuring any access decisions between two security contexts in the security server exactly match the access decisions stored in each object manager [8].

There are three types of permissions within an object manager, permissions retained within a data structure accessed via the SIDs (the Access Vector Cache to be discussed later), within a data structure accessed by the object (page table), and within the "state of a request currently being processed" [8]. In the case of objects not currently being processed, the object manager can revoke permissions by altering the current data structures. It is the third type of permission that causes problems with revocation on most systems including Flask based implementations. There are two basic methods to achieve revocation of a permission being utilized by a currently running process: undo the progress of the process and revert to a state before the permission was granted or allow the process to continue to run as long as it finishes in a timely manner [11].

The Flask architecture has an informal property of timeliness of changes that states, "Whenever a change in policy is identified by the security server…any permission revoked by this change will be removed from the policy…in a timely manner" [10]. The fact that the revocation is timely and not instantaneous is mainly due to the fact that revocation of permissions must occur utilizing communication between the security server and object managers and therefore, an atomic operation is not possible. Nevertheless, if a currently running process is allowed to run to completion before it is revoked, the process must be trusted to complete and not block indefinitely on the revocation request in order to assure that timeliness is attained.

The capability to revoke previously granted permissions, even with some time delay, is a significant achievement. While an operating system such as UNIX allows the owner of a file to change and even revoke permissions, the permission is only checked on file open and cached in the file descriptor. Thus, as long as a process utilizes the file descriptor it received on opening the file, the process maintains permission to write to the file even if it is not currently utilizing the descriptor. Even in the security architecture developed for DTOS, which is the most closely related architecture to Flask, revocation of permissions was not possible.

Finer grained control of access rights:

The Flask security policy is actually split into two separate parts, the root policy and extension policies. The root and extension policies are ultimately defined by the specific implementation of the security server. In general the root policy is the section of the policy that is defined directly in code as the MAC policies in the security server, while the extension policy is defined by simple lookup tables. As explained in the security policy flexibility section, the choice of MAC or groups of MAC chosen is determined by implementation needs. The extension policies add more capability to the Flask security by allowing policies that extend the root policy and "allow clients to define finer-grained policies within the restrictions of the root policy" [8].

The extension policies are independent of the actual root policies because they consist of only lookup tables. However, the extension policies cannot break the root security policy. The separation of the root and extension policies actually causes a separation in types of security contexts. There are root security contexts that are

defined based upon the MAC policies implemented in the security server. The extension security contexts consist of "a name by which it is known and a parent SID, which may be a root SID or an extension SID" [8].

A client, not necessarily a security policy administrator, has the ability to load new extension security contexts independently into the security server as long as the client has permission to add an extension policy to the valid parent SID. The SID created for the extension security context is actually a descendent, hierarchically, of a root security context in a sequence of SIDs and is used as a shortcut in the path of the default decision making logic of the root security policy. The client is allowed to add or remove children of the parent SID, where removal of a child includes the removal of that child's children. Also, additional permissions can be added between children of a parent SID or another SID altogether as long as a permission check is performed [10].

One example use of these extension policies is in a networked environment where each node supports the root policy and contains specific client side extension policies directly related to the needs of the particular client node. These client side extension policies are a dramatic improvement over other security systems. They provide a way to quickly setup new policy decisions while not breaking the actual conditions of the MAC policies that are in effect. These new policies can be used to create finer granularity in the policy decisions than would not be capable in an inflexible secure operating system such as Multics or VAX VMS. In these other systems the security policy administrators must setup all security policy decisions, including the actual policy. Flask allows the client around this strict rule while not short-circuiting the overall policy of the system.

Cache of previously determined access rights:

The Flask security architecture defines an access vector cache (AVC) that allows object managers to cache access decisions previously made. The AVC minimizes performance overhead because the object manager does not need to constantly communicate with the security server and the security server does not need to calculate the access decision for every security decision.

The AVC is actually what performs the communication between the object manager and the security server. The AVC is a common library between the object managers, "provides for the coordination of the policy between the object manager and the security server" [11]. The security server will return an access vector that is a collection of related permissions (e.g. all file access permissions). The AVC contains a collection of nodes which store the SIDs of the objects and their classes that the access decision involves as well as access vectors returned from the security server that define if the access decision has been decided, if it is allowed, and if it has an expiration time.

The AVC is also important in transferring all policy changes to the object managers. Since it is the communication coordinator between the security server and the object managers, any change in policy will first cause an update in the cache. The AVC then "invokes any callbacks which have been registered by the object manager for revoking migrated permissions" [11].

Nevertheless, the most important feature of the AVC is its ability to cache previously granted access rights since querying the security server has the largest impact on performance. In fact while the SELinux implementation saw some effect on the percentage overhead for specific tasks, there was negligible effects on its macro benchmark tests that would better utilize the AVC. Also, performance improvements could occur because "neither the AVC nor the security server implementations have been optimized" [5]. The network implementation also showed negligible impact to performance due to the use of the AVC. Two steps, both associated with communication with the security server, were responsible for 98.8% of the costs of the implementation to performance. In fact, the cost of security association setup dropped to zero and the costs of the access control mechanisms are "reduced over an order of magnitude due to access vector caching" [1].

Multiple operating system implementation:

One of the most important aspects of the Flask security architecture is its capability of being implemented in multiple operating systems as well as other types of environments that require security. Currently, the Flask architecture has been implemented in two separate operating systems, Flask prototype on the Fluke operating system [8,11] and SELinux [5], as well as specifically in a network server made from the Flask prototype operating system [1]. The Flask security architecture only requires the system has a reference monitor. This is so that processes have some form of separation. However, even the implementation of the isolation properties is up to the actual implementation of the operating system.

The architecture itself is proven using a formal model described in [9] instead of depending on the specific implementation for assurance the architecture works. However, the Fluke operating system was specifically used because it was a university-based research operating system and the network security system only implemented the Flask architecture for the BSD TCP/IP stack, thus allowing for better assurance that the implementations could be validated. In fact, the SELinux system was developed specifically to advance the research and expand the technology to a wider audience because of Linux's wide usage.

Flask also provides, in the case of SELinux, the "appropriate default behaviors so that existing Linux application programming interface (API) calls can be left unchanged" [5]. In fact, most existing applications on each of the operating systems non-secure implementations (Linux and Fluke) did not need any modifications in order to be used on the SELinux or Flask prototype system. However, API calls were also either created or extended in both the SELinux and Flask prototype implementations so that security-aware applications could be used or developed [5,12]. Even the network security implementation could receive unlabeled packets from nodes that were not necessarily implementing Flask. These packets would be labeled with default SIDs so that they could continue through the network at appropriate access levels based upon the network interface or the source of the packet [1]. Thus, Flask will not even adversely affect already developed applications and systems unless they attempt to access areas they are not allowed.

One last advantage of the ability to utilize Flask with multiple operating systems is the fact that the users of the security architecture are now no longer limited to a specific hardware configuration. While Linux and Java have both shown the advantage of hardware portability, most if not all secure operating systems, from Multics and SCOMP to today's Trusted Solaris, have been hardware specific. This hardware limitation has placed a great burden on both dollars and compatibility. The VAX VMM Security Kernel was cancelled partially because the security kernel could not utilize the new hardware being developed for the non-secure version of VAX without difficult and expensive upgrades to that hardware [3].

Implementation Complexity:

While the Flask architecture has been proven using formal methods as described in [9,10], the actual implementations pose difficult problems, especially in systems that were previously not designed for the level of security that Flask provides. These systems also pose difficulties because of the inherent differences in their original design. For example, the security server is a user-space server, thus the name, on the Flask prototype implementation because Fluke is a micro-kernel, while it is a kernel subsystem in SELinux because Linux has a more monolithic design.

The appendix shows the level of implementation difficulties with a listing of both changed and new utilities and system calls that have already been implemented in SELinux. The SELinux implementation required integration of the architecture design into process, file, and socket operations. In fact, the module in SELinux that contained just the security policy code is 2.5 times larger than the entire Multics kernel [4]. This level of complexity in and of itself could be a security risk because of the difficulty in assuring/validating the implementation.

Also, the SELinux design was not completely implemented and is still undergoing work. For example SELinux has the capability to revoke permissions of previously granted access; however, it does not yet have the capability of automatically relabeling processes that have already performed a permission check. Only on subsequent permission checks will a process fail when a subject and/or object is denied access [5]. This could possibly break the timeliness property in the Flask architecture specification and shows how design robustness does not necessarily mean an implementation can be assured. Every implementation would require significant testing for validation that the design is implemented correctly.

Even the prototype version of Flask implemented in the Fluke microkernel is an incomplete implementation not fully supporting the requirements for revocation in the user-space object managers (the file, network, and process servers) [10]. Also, the network server described in [11] is just an extension of the network server described in [1]. Only the micro-kernel section of the operating system utilized a fully implemented Flask architecture including the ability to revoke previously granted access rights. Yet, these changes due to the implementation in the kernel caused the kernel to grow by 19%, with about 40% of the changes being non-trivial (trivial changes included one-line changes, #define changes, and name or parameter changes) [11]. The kernel required the implementation of object labeling and a control policy for enforcing access decisions, enhancements to the IPC mechanisms, creation of an AVC, as well as the implementation of revocation ability [8]. This also does not include the implementation of the security server, including the development of a security policy. In fact the development of an example security policy was a major undertaking of the development of SELinux. If users were required to build the security policy from scratch, just the configuration of the system would be extremely complex [11].

Conclusions:

The Flask security architecture is an excellent starting point for the development of a secure operating system. The architecture provides a proven model that allows both flexibility and assurance in security policy. It provides revocation capabilities and policy flexibility that used to be either impossible or incompatible in most systems, along with a greater degree of overall security policy control. Also, because of the access vector cache, security can now be provided with only minor performance hits.

While implementation of the security policy is difficult, further study of the architecture as well as the necessity of more secure systems make the use of a Flask based operating system possible in the near future. A Flask system will never achieve the level of assurance that Multics and the VAX VMM security kernel achieved because of the complexity and size of its implementation. However, the flexibility of the system far outweighs the assurance that those two systems provide in all but the most critical systems. Also, the level of assurance that can be achieved through validation of a Flask based system is much greater than most modern operating systems that are currently used for high security systems. Therefore, a move to a system such as Flask that provides flexible mandatory access control would still be warranted for improving the security of critical functions.

Bibliography:

1. A. Chitturi. “Implementing Mandatory Network Security in a Policy-Flexible System.” Masters thesis, University of Utah, June 1998. URL: http://www.cs.utah.edu/flux/flask/

2. D.F. Ferrailolo, J.A. Cugini, and D.R. Kuhn. “Role-Based Access Control (RBAC): Features and Motivations.” In *Proceedings of the Eleventh Annual Computer Security Applications Conference*, Dec. 1995.

3. P.A. Karger, M.E. Zurko, et al. “A Retrospective on the VAX VMM Security Kernel.” **IEEE Transactions on Software Engineering**, vol. 17, issue 11: pages 1147-1165, November 1991.

4. P.A. Karger and R.R. Schell. "Thirty Years Later: Lessons from the Multics Security Evaluation." Prepared for the 18th annual ACSAC, Revision 1, Sept. 2002. URL: http://www.research.ibm.com/resources/-paper_search.shtml

5. P. Loscocco and S. Smalley. "Integrating Flexible Support for Security Policies into the Linux Operating System." Technical Report, NSA and NAI Labs, Oct. 2000. URL: http://www.nsa.gov/selinux/

6. P.A. Loscocco, S.D. Smalley, P.A. Muckebauer, R.C. Taylor, S.J. Turner, and J.F. Farrell. "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments." In *Proceedings of the 21st National Information Systems Security Conference*, pages 303-314, Oct. 1998. URL: http://csrc.nist.gov/nissc/1998/proceedings/paperF1.pdf

7. S.E. Minear. "Providing Policy Control Over Object Operations in a Mach Based System." In *Proceedings of the Fifth USENIX UNIX Security Symposium*, pages 141-156, June 1995.

8. Secure Computing Corp. "Assurance in the Fluke Microkernel: Final Report." CDRL A002, 2675 Long Lake Rd, Roseville, MN 55113, Feb. 1999. URL: http://www.cs.utah.edu/flux/flask/

9. Secure Computing Corp. "Assurance in the Fluke Microkernel: Formal Security Policy Model." CDRL A003, 2675 Long Lake Rd, Roseville, MN 55113, Feb. 1999. URL: http://www.cs.utah.edu/flux/flask/

10. Secure Computing Corp. "Assurance in the Fluke Microkernel: Formal Top-Level Specification." CDRL A004, 2675 Long Lake Rd, Roseville, MN 55113, Feb. 1999. URL: http://www.cs.utah.edu/flux/flask/

11. R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Anderson, and J. Lepreau. "The Flask Security Architecture: System Support for Diverse Security Policies." In *Proceedings of the Eighth USENIX Security Symposium*, pages 123-139, Aug. 1999. URL: http://www.cs.utah.edu/flux/flask/

Appendix: List of new and changed utility and system call in SELinux

Taken from NSA Website: http://www.nsa.gov/selinux

Utilities

- chcon
- df
- find
- id
- install
- killall
- ls
- mkdir
- mknod
- mkfifo
- newrole
- ps
- pstree
- runas
- run_init
- tar

System Calls

- accept_secure
- chsid
- chsidfs
- connect_secure
- execve_secure
- fchsid
- fchsidfs
- fstat_secure
- fstatfs_secure
- getosecsid
- getpeername_secure
- getsecid
- getsockname_secure
- listen_secure
- lstat_secure
- mkdir_secure
- mknod_secure
- msgget and msgget_secure
- msgopo
  - msgrcv and msgrcv_secure
  - msgsnd and msgsnd_secure
- msgsid
- open_secure
- recv_secure
- recvfrom_secure
- recvmsg_secure
- security_change_sid
- security_compute_av
- security_context_to_sid
- security_get_sids
- security_load_policy
- security_member_sid
- security_notify_perm
- security_sid_to_context
- security_transition_sid
- semget and semget_secure
- semsid
- send_secure
- sendmsg_secure
- sendto_secure
- shmget and shmget_secure
- shmsid
- socket_secure
- stat_secure
- statfs_secure