# Using the Flask Security Architecture to Facilitate Risk Adaptable Access Controls

Machon Gregory
mbgrego@tycho.nsa.gov

Peter Loscocco
loscocco@tycho.nsa.gov

*National Security Agency*

## Abstract

Risk Adaptable Access Control (RAdAC) is an important emerging technology that has gained the attention of many people as a way to change the current information dissemination policies. Systems implementing RAdAC have the ability to enforce a flexible mandatory access control policy based on various changing factors, such as situational and environmental factors. Unfortunately, commonly deployed systems are unable to reliably support this type of access control, but by using existing technology the desired capability could be built. Applications of the Flask Security Architecture is one such use of a existing technology. This paper describes how the Flask Security Architecture can be used to provide RAdAC. It describes how this was done in the creation of a prototype RAdAC system.

## 1. Introduction

The Global Information Grid (GIG)[2] architecture describes a significant change in the way government systems will interact. Whereas in existing architectures, information dissemination is predominately governed by a "need to know" philosophy that limits information sharing to only those mission elements able to demonstrate a need, the massively connected nature of the GIG and its requirement to support many interrelated missions mandate a change in information dissemination philosophy. Emphasis needs to shift to a "need to share" between different mission elements that require ready access to each others information in order to achieve success.

Current access control schemes adhere to a well defined policy that is often static, but the new paradigm calls for a more flexible model. To facilitate this change a new access control model, Risk Adaptable Access Control (RAdAC), described in the GIG Information Assurance documentation, has gained popularity among many groups. RAdAC systems can be characterized by their ability to enforce access control decisions according to a security policy that can dynamically change. As risk is perceived to change, the security policy governing access control decisions changes accordingly, where risk can be defined by many independent factors. Factors that might affect risk include environmental and situational factors as well as things like characteristics of people or IT components.

RAdAC implies that there exists a characterization of factors that would indicate the policy should be switched. Defining policies for each distinct set of risk postures is difficult, as is designing an architecture that can support changing between them. Any RAdAC architecture must have the ability to reliably enforce whatever policies exists for each risk posture. It must be able to dynamically change access to information, escalating privileges or revoking accesses as required by the change in risk.

Although truly supporting RAdAC is beyond the capabilities of commonly deployed systems, technology does exist on which a RAdAC system could be built. This paper describes a prototype that demonstrates that the Flask Security Architecture[10] is an example of suitable technology for RAdAC.

## 2. Risk Adaptable Access Controls

A RAdAC model uses various static and dynamic factors to influence access control decisions. These factors form the basis for deciding what security policy, or set of access rules by which access decision are computed, to enforce. By considering each of these factors, the system can dynamically select a policy that results in different access decisions as appropriate to the perceived risk. Under RAdAC, a situation could allow data to flow to places that might normally be considered restricted. Any system that correctly implements RAdAC must always keep all of its data under its control, even when it is allowed to flow to restricted places. This condition has to be met because the conditions that allowed data to flow there might change which would require a revocation.

For example, policy could dictate that in a time of national crisis first responders need access to some forms of sensitive information in order to save lives. Under these conditions, “need to share” outweighs the “need to know.” At the conclusion of the crisis, “need to share” would no longer take precedence. Security policies could be written for each of these situations. A RAdAC system would be capable of switching between them.

Network applications under the control of RAdAC might deliver data to remote systems and later be required to recall that data from one or more of those machines. In order to reliably do this, any system to which data is released must be capable of controlling where that data can flow. It must be able to respect any information flow restriction imposed upon it by the originating system. If this can not be done, the RAdAC system would lose control of the data, making revocation impossible.

For example, a server accepting requests from clients on the network should never release data that would be subject to revocation unless it is satisfied that the client's system is capable of restricting what happens to the data once the client's request is fulfilled. How the server is satisfied is an architectural constraint on the entire networked RAdAC system. If this constraint can not be architecturally met, it will not be possible to know that security policies are not violated.

The only way to architecturally meet the requirement to support a RAdAC system is through the appropriate use of mandatory access controls (MAC) on the collection of systems comprising the RAdAC system. If any system fails to correctly implement MAC, the RAdAC system is subject to failure. No data protection guarantees involving that system will be meaningful.

To correctly implement MAC three conditions must hold. The system must be in control of all subject and objects at all times. All security decisions must be based on all security relevant information. The policy governing security decisions must be under one central control[8]. Failure to meet any of these conditions impacts a system's ability to restrict data flow and protect system integrity.

A correct implementation of MAC on all systems, although necessary, is not sufficient for a RAdAC system. Supporting policy changes is fundamental to the nature of RAdAC. Each of the systems being relied on to enforce a MAC policy must be flexible enough to support the necessary changes to policy imposed by the RAdAC system. Furthermore, the implementation must be capable of enforcing any revocations necessitated by a policy change.

## 3. Problems Impeding RAdAC

There is a fundamental problem inhibiting effective RAdAC systems from being built. RAdAC requires flexible MAC environments that surpass the current capabilities of commonly deployed systems. They are incapable of meeting the MAC requirements outlined in section 2. Even systems that do offer some form of MAC lack the flexibility necessary to support policy changes.

Commonly deployed systems have discretionary access controls (DAC). DAC systems are capable of supporting policy changes. However, they are incapable of making the guarantees needed to meet data flow constraints and revocation requirements. For example, data transmitted to a remote machine may be printed, copied, or sent to another machine. When these things happen the RAdAC system has lost control of the data and their is no hope of revoking access to it on a policy change.

## 4. Demonstrating RAdAC

Although commonly deployed systems cannot be used to create a system with RAdAC capabilities, technology does exist that has the desired MAC properties with the policy flexibility required for supporting policy changes and revocation of access based on those changes. With this technology, it is possible for networked applications to release data to certain remote systems, knowing that the constraints of RAdAC can be met. That technology is the Flask Security Architecture applied at appropriate points throughout the RAdAC system.

The goal of this work was to demonstrate that the Flask Security Architecture (Flask) can provide the functionality needed to support a RAdAC capable system. This was achieved by constructing a prototype system. The prototype uses SELinux, an implementation of Flask in the Linux operating system, to meet the system MAC requirements. It meets the requirements for network MAC, by using Flask applied to a network application.

The prototype demonstrates that currently available technology can be used to build a RAdAC system. SELinux provides a secure operating system that is a strong foundation for each component of the RAdAC system. The support that SELinux offers in applying Flask to user-space components facilitates creating the components of the system responsible for network MAC. A RAdAC capable system was successfully built using the capabilities provided by these two applications of Flask.

To demonstrate the RAdAC capability a network application was needed. Any network application, such as a database server or web application, could have sufficed. The principles involved are easily generalized. The application that was chosen was an existing XML-based document server designed to run on SELinux[1].

To best demonstrate the RAdAC capability, several things were needed from the network application. There needed to be a data source and consumer. An ability to enforce policy on involved systems needed to be shown. In particular, on the system hosting the data consumer, it needed to be possible for the RAdAC system to maintain control of any data released from the source. Policy flexibility and the ability to revoke access to data once released was key to the demonstrations success.

The demonstration was implemented on a network of SELinux systems, using the capabilities of the SELinux system for protection. Several security policies, representing different possible RAdAC situations, were constructed for the demonstration. An ability to indicate that a policy change was required throughout the system was also constructed.

## 5. Flask Security Architecture

The Flask Security Architecture is designed to provide support for diverse security policies through the separation of policy from the enforcement mechanism. Flask has two main components. The first is a security server that provides a centralized point where all security policy decisions are made. The security policy is a set of rules, governing access and labeling decisions, that are specified through configuration files.

The second component is an object manager that implements functionality. Object managers are responsible for requesting access and labeling decisions from the security server when needed. They are also responsible for binding labels to the objects they manage and enforcing all decisions provided by the security server. The implementation of any object manager must control all security relevant operations on its object.

There is also a third component called the access vector cache (AVC). AVCs are incorporated into object managers. The AVC stores previously computed access decisions. The security server is only consulted on cache misses.

Flask supports policy changes through reloading of the security policy configuration. The security server maintains control over AVCs, enabling policy changes to be propagated throughout the system. Revocation is handled through flushing of the AVC causing future access decision to require consultation to the security server running with the new policy. Additional revocation comes from the ability of the object manager to request notification of policy changes. On policy changes, object managers can then perform special processing to ensure no access allowed under the original policy is still granted after it would be prohibited by the new policy.

Flask has been shown to be suitable to be incorporated into a wide variety of object managers. Flask has been applied to operating systems, as well as applications. Flask has been shown as a suitable architecture to implement flexible MAC.

## 5.1. SELinux

SELinux is an application of the Flask Security Architecture, resulting in a version of Linux with flexible mandatory access control over Linux abstractions. The Linux kernel acts as an object manager. A security server and an AVC have been incorporated into the kernel. The kernel has been instrumented to make all the necessary access and labeling decision requests. The security server implements a security model that is a combination of type enforcement, role-based access control, and multi-level security.[7]

SELinux is capable of supporting many types of policies enabling it to meet various system goals. Policies can be defined to provide strong process isolation and to support least privilege. They can be used to meet the integrity goals of the system. The flexibility of SELinux allows policies to be tailored to meet the specific security goals of applications.

The features of SELinux enable client/server applications to be made more secure. SELinux policy can be written to enforce separation between client and server processes from other applications on the system. Stronger guarantees about the integrity of applications are possible because of policies written to protect the application's executable image and configuration data. The interactions between all processes on the system can be tightly controlled. If the client or server is connected to the network, SELinux's integration of labeled IPSec[5,6] allows data communicated over the network to be tightly bound to the process using the connection while guaranteeing no other process has access to it.

## 5.2. Network MAC

It is possible to extend Flask to network applications. The network application is treated as a user-space object manager. Traditionally, user-space object managers implement functionality local to the system and enforce a security policy represented in the local security server. Network object managers differ in that they implement functionality over network objects and must enforce a network security policy which may be different than the local policy of any of machines involved. In all other aspects, they are fundamentally the same as user-space object managers.

Permissions are defined to control access to the objects maintained by network object managers. Like other object managers, network object managers are responsible for enforcing security decisions. They rely on a security server for all security decisions. However, the security server for network object managers is likely to reside on a different machine. This places a network security requirement on the implementation to protect the communications between the network security server and all object managers in addition to any network security requirements already present to support the network application.

Network object managers can be used to support revocation. The network security server retains control over the operation of AVCs. It can notify the object managers of policy changes allowing it to implement any special processing required as a result of a network policy change, including the revocation of access to data. Because the object manager is not under the direct control of the network security server, network object managers must reside on systems capable of protecting them and of restricting data flow from the object manager.

SELinux includes support for user-space object managers. There is an AVC library that includes all the necessary functionality for an AVC and interaction with a security server, whether that security server is in the kernel or user space. As part of this work, the AVC library was modified to have better support for network object managers by allowing the network security servers that implement security models and policies distinct from the security server on the host SELinux system.

## 5.3. SE-XML

SE-XML is an XML[4] parser that has had Flask applied to it. It enables a customized view of XML documents based on a security policy. With SE-XML, it becomes possible to put internal markings, like paragraph markings, on a document and offer distinct views of that document depending on security attributes of the viewer.

SE-XML applies security labels to XML nodes used by the security server to determine access to nodes within a document. The XML parser is an object manager enforcing access decisions about each operation on each XML node. A new security class with permission over documents was defined.

## 6. Implementation

The goal of the RAdAC prototype is to demonstrate how a network application running on SELinux systems can achieve the capabilities described previously. A document server using SE-XML was chosen as the prototype network application. This client/server application has server and viewer components each residing on different machines. The server has the ability to respond to the clients request for documents. The clients have the ability to display the documents and nothing else. Because the server is responsible for the release of the documents, it must maintain control over them, including when then they have been released to the viewer. When events dictate a policy change is required, all client access to the documents must be revalidated, and if necessary, revoked.

In addition to the server and viewer, the prototype contains a network security server and risk knob. The network security server is able to make access control decisions for network objects based on a flexible MAC policy. Network security between all components is provided by IPSec. The risk knob provides a simple graphical interface to change the network policy. The risk knob represents the system's ability to recognize changing security conditions and initiate policy changes.

### 6.1. Document Server

The SE-XML document server was implemented as a user-space object manager running on SELinux. Local SELinux policy was written to provide the necessary protections from the rest of the system. The objects that it controls are documents. These are network objects and are labeled with a network label that is distinct from the local SELinux labels. SE-XML is used to bind network labels to documents. Documents are stored and protected on the server by the SELinux filesystem access controls.

There are two kinds of access decisions that must be made in servicing any request for a document. The first is a check to see if the requester is authorized to use the server. This is handled by SELinux on the server's machine. No requester will be granted access to the server without being approved to use the IPSec connection. From the IPSec connection, the IP address for the client can be reliably extracted. The IP address is then used to determine the security label in all future access checks for that session.

The second kind of access decision comes from the network security server. The network security server handles all labeling and access decisions about the documents. This kind of access control is what enables the document server to determine which files can be released to authorized users. Access checks are of the form "can a document with a particular network label be released on request from a system with a particular session label."

### 6.2. Document Viewer

The document viewer is a simple GTK[3] application running on SELinux. The document viewer requests documents by connecting to the server using IPSec. In order for a client to receive a document, it must have access to both the IPSec security association and pass the network access checks for that document at the server. Local SELinux policy protects the executable image from the rest of the system. Because labeled IPSec is used, there is a tight binding between the network connection and the viewer application.

The viewer application is only capable of displaying documents it receives. Local SELinux policy was written to close all channels that might be used to release the document beyond the viewer's control. This in effect creates a confined space on the viewers machine to which documents can be released while remaining under the control of network security policy. When a revocation is required as a result of a network policy change, the viewer is able to support the revocation of access to the document since it is still within its control.

It is important to limit the functionality of the viewer to just viewing documents. If more functionality is allowed, it becomes increasingly difficult, if not impossible, to keep the document under the control of the network policy. There would have to be a much more complex policy controlling the interactions with all parts of the system to which the document could be released. Any attempt to revoke access to that document would have to involve all the areas where the document could have migrated.

For example, if the viewer were able to write the document to a database, any application able to read from the database could potentially have access to the document. In order for the network security server to maintain control over the document, the database would have to enforce decisions about the documents released and the local SELinux policy would have to isolate the database and any other application that could potentially access the document. Additional release of the document would have to be restricted. Furthermore, revoking access to the document would require that it be purged from the database and all applications that have touched it. This only becomes more complicated as more applications are involved. So by constraining the functionality of the viewer, the confinement and revocation problems were kept manageable.

## 6.3. Network Security Server

The network security server is implemented on SELinux. It is an example of a user-space security server for a network application. Its clients are document servers. The network security server implements the network policy over the documents. Local SELinux policy protects the network security server, and IPSec protects the communications and access to it, as was the case for the server and viewer. The network security server is itself a user-space object manager controlling access over its own services.

When labeling was added to the network, a network wide abstraction was created that made it possible to have a consistent view of attributes associated with network objects. All documents throughout the network, regardless on which document server they reside, must be labeled with a network label. However, the documents are represented on their host system by system objects with labels specific to that system. Similarly, sessions need network labels but are established with one local to the system. In order for the network security server to be able to make network access decisions, all labels must be translated into a corresponding network label prior to any access check. This translation is done by document servers in the prototype.

## 6.4. Policy Changes due to Changing Risk

In a system that supports RAdAC, the security policy that is in effect at any given time is in some way determined by risk. When the perception of risk changes, the system must change its security policy to reflect the new perceived risk. How different perceptions of risk are quantified is beyond the scope of this effort, but the prototype demonstrates that given several different security policies for different levels of risk, it is possible to switch between them safely. The mechanism that initiates that switch is the risk knob.

In the RAdAC prototype, the risk knob was implemented as a GUI tool that allows a security administrator to alternate between a collection of predefined security policies. The risk knob resides on an SELinux system, using the MAC capabilities of SELinux to protect the use of the knob. When an administrator indicates that a change in risk has occurred, the correct security policy corresponding to the new risk is selected and the change initiated.

The first thing that happens when the GUI tool triggers a change is the network policy is reloaded. The network security server exports a load policy interface. The risk knob tool uses it to load the new policy. The network security server restricts access to that interface to authorized systems. Whenever a policy is loaded, the network security server flushes all the AVCs that are relying on the old policy and calls registered notification routines in the document servers.

When a document server receives notification of the policy change, it must ensure all documents for which it is responsible are only viewable in clients authorized to do so under the new policy. If any viewer is currently accessing a document to which it is no longer entitled, access must be revoked. The document server does this by forwarding the policy change notification to each of its client viewers. The server does the notification to relieve the security server of maintaining state of all client sessions.

When a viewer receives a policy change notification, it is responsible for revalidation of all access to all documents being viewed. The viewer does this by re-requesting access to each document from the server. This model was chosen to limit the state kept on the server. On any request that fails, access to the document is terminated. There is no effect to any document where access decision is unchanged. The decision to limit the functionality of the viewer to only displaying documents enables the revocation to be completed at this point.

## 7. Future Work

The prototype demonstrates the ability to create a RAdAC system by applying Flask to a network applications and running its components on SELinux, but more work is required before it is ready for real-world use. As work continues on a secure windowing environment and gets included into SELinux distributions, vulnerabilities within the viewer systems can be addressed. Also, work can be done to ease the burden of initial policy configuration for system administrators.

The biggest hurdle to overcome in order to mature this technology is to demonstrate how to loosen the restriction imposed on the viewer. Work is needed to allow data to flow deeper into client systems while retaining control over it. Work is required to write additional policy and modify applications to handle network policy changes. One idea is to use NetTop[9] to increase functionality in the client. The flow of data to NetTop virtual machines can be controlled, and experiments can be done to determine the best way to handle revocation.

Real applications supporting RAdAC may need a more automated way of identifying the need to change policy than a manually operated risk knob. The replacement of the risk knob with external sensors to identify changes in the environment will assist in this automation. Identifying the best way to utilize these external inputs to make the decision to change the policy is an area of future work.

## 8. Conclusion

In conclusion, the demonstration showed the that Flask could be use to build a system capable of supporting RAdAC. A network application utilizing the Flask architecture has the ability to enforce a network MAC policy and the flexibility to support changes in policy and revoke previously granted accesses according to policy changes. SELinux on the end systems of the network application provided the information flow guarantees needed to ensure the application was able to reliably enforce its policy and revoke granted access when necessary.

The network application is fairly rudimentary and is not indicative of the applications that are really wanted or needed. Additional work is needed to bridge the gap between the prototype and a deployable RAdAC system. Experience with the prototype shows that future work is warranted.

## 9. References

[1] Coker, G. Controlled Sharing Concept Paper, National Information Assurance Research Labortory (NIARL), NSA.

[2]Department of Defense Directive Number 8100.1, September 19, 2002.

[3] GTK. The Gimp Tool Kit. http://www.gtk.org/, October 2006.

[4] Harold, E. and Means, W. S. XML in a Nutshell: A Desktop Reference. Cambridge:O'Reilly, 2002.

[5] Jaeger, T. Hallyn, S. and Latten, J. Leveraging IPSec for mandatory access control of Linux network communications. Tech. Rep. RC23642, IBM Research, 2005.

[6] Jaeger, T. SELinux Protected Paths Revisited. In the proceeding of the SELinux Symposium, 2005.

[7] Loscocco, P. Smalley, S. Intergrating Flexible Support for Security Policies into the Linux Operating System. In the proceeding of the FREENIX Track: 2001 Usenix Technical Conference, June 2001.

[8] Loscocco, P. Smalley, S. Muckelbauer, P. Taylor, R. Tuerner, S. J. and Farrell, J. F. Then inevitability of failure: The flawed assumption of security in modern operating systems. In proceedings of the USENIX Security Symposium, 2004.

[9]Meushaw, R. and Simard, D. NetTop: Commercial Technology in High Assurance Applications. Tech Trend Notes, Fall 2000.

[10]Spencer, R. Smalley, S. Loscocco, P. Hilber, H. Anderson, D. and Lepreua, J. The Flask Security Architecture: System Support for Diverse Security Policies. In the proceeding of the Eighth Usenix Security Symposium, Pages 123-139, August 1999.