Using GConf as an Example of How to Create an Userspace Object Manager

James Carter
National Security Agency

Abstract

GConf is a configuration system for GNOME. It does not provide adequate security controls over the configuration data that it manages, which could allow the security goals of the system to be violated. There are various strategies that can be used to provide security controls over an application under SELinux. One strategy, which is a natural part of implementing the Flask architecture on Linux, is to turn the program into a userspace object manager. This paper discusses the general process of providing SELinux controls over a program and the specific steps taken to provide SELinux controls over GConf.

1. Background

Any program without adequate security controls over it could be used to violate the system's security goals. SELinux [1] is a Linux implementation of Flask [2], which is a flexible mandatory access control security architecture, and can be used to enforce system and application security goals. There are several different strategies for doing this; this paper discuses some of these strategies and how they were applied to GConf. GConf was chosen because it manages configuration data which could potentially be used to compromise the system's security goals.

1.1. SELinux

The Flask architecture consists of a security server, object managers, and access vector caches (AVC). One of the primary design goals of the Flask architecture is to separate the security policy from the mechanism used to enforce it. This is accomplished by having the security server encapsulate the security policy logic and make all access control decisions and by having the object managers bind security labels to their objects, request labeling and access decisions from the security server, and enforce those decisions. An AVC caches security decisions for one or more object managers to minimize the performance overhead of obtaining an access decision from the security server.

In SELinux, the security server, along with an AVC, is a separate kernel subsystem [3]. The other kernel subsystems have been turned into object managers by adding a way to bind security labels to their objects, adding Linux Security Module (LSM) hooks to bind security labels to their objects and to query the security server for labeling and access decisions, and adding the appropriate logic to handle access denials.

SELinux provides an administratively controlled policy, controls all processes and objects, and makes decisions based on all security-relevant information. SELinux controls more than just files and capabilities: It also controls sockets, interprocess communication, shared memory, and processes. Processes are labeled based on the user, role, clearance, and function of the program. This allows a process to run in different domains depending on what user is running it or what data it is processing. SELinux labels the actual processes and objects to ensure that the security characteristics of the object are uniquely and reliably identified.

1.2. GConf

GConf [4,5,6] is a configuration system for the GNOME desktop. Although used only by GNOME, GConf is not dependent on GNOME and could be used by other desktops. GConf stores configuration data for programs and provides change notification to programs. Change notification is an important feature, because it allows an application to receive notification when a configuration value changes. This allows an application to reflect the actual configuration at all times without having to periodically read its configuration data.

1.2.1. Architecture

GConf consists of a set of configuration sources, a client library, and a per-user configuration server. The client and server communicate through ORBit [10], which is a version of CORBA. See Figure 1: The GConf Architecture.

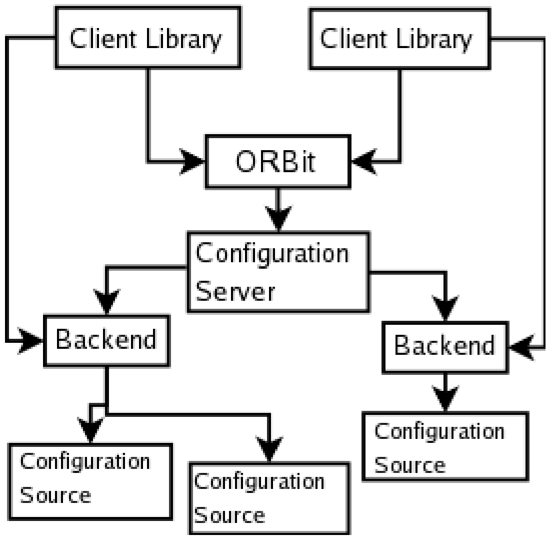

Figure 1: The GConf Architecture

Configuration sources store configuration data and metadata. Configuration data are stored as key-value pairs. Keys look like filesystem paths, but, even though every component except the last is called a directory, they are not. Configuration values can be elements, pairs of elements of the same or different types, or lists of elements with the same type. Elements can have a type of string, boolean, integer, or floating point. Configuration metadata are called schemas. Schemas contain metadata about key-value pairs such as the expected type, the default value, and a description. They are stored in their own namespace which starts with `/schema`.

A backend is used to access a configuration source. There are currently XML and LDAP backends in GConf, but others could be used [11]. The XML backend supports both a directory tree structure in the filesystem, with key directories being actual directories, and a single XML file, with key directories being sections specified with the `dir` tag.

The client library provides the interface for a program to access the configuration sources either through the configuration server or directly through a backened. It caches configuration values from previous queries, allows a specific set of configuration sources to be specified and used, and, in conjunction with the configuration server, notifies the client when the value of a registered key changes.

The configuration server, gconfd, is a per-user daemon that accesses the configuration sources through the appropriate backend and presents a unified set of configuration data to the client. It also notifies the client library of all clients effected when the value of a key changes. The configuration sources used by default are specified by the GConf configuration, which also designates the backend to use and whether the source is read-only or read-write. The configuration server also allows a client to specify the specific set of configuration sources that should be used by it when acting for the client.

The order the configuration sources are used by the configuration server is important. The configuration server will read configuration data from the first source from which it can read the data. It will write data to the first source that is writable, but, even then, only if it couldn't read it from an earlier source. Since only the configuration source in the user's home directory can be written by the user, any changes made by the user are stored in it and only effects that user. The configuration sources listed earlier cannot be changed by users, so the values contained in them must be used by the user.

ORBit is the version of CORBA used on Linux [10]. An interface is created using the Interface Definition Language (IDL), which creates a client stub and server skeleton when compiled. An instance of the interface is created by calling the interface's initialization function. Each instance created has a different ORBit object reference associated with it. The ORBit object reference can be advertised externally with a Interoperable Object Resource (IOR) string. The ORBit object reference is used to invoke the interface instance's methods. The Object Request Broker (ORB) passes all requests to the appropriate interface instance.

There are three ORBit interfaces used in GConf. The configuration server creates the *ConfigDatabase* and *ConfigServer* interfaces. The ConfigDatabase interface is used by the client to access the configuration sources. The configuration server creates a separate instance of the ConfigDatabase interface for each set of configuration sources used. The ConfigServer interface is used by the client to get a reference to a ConfigDatabase instance for a specific set of configuration sources. The client creates the *ConfigListener* interface. The ConfigListener interface is used by the server to notify the client of a change in a directory that has been registered.

1.2.2. Operation

When the client wishes to perform an operation on the configuration sources, it calls the appropriate client library function and passes the key. The client library checks the cache for the value, if appropriate for the desired function, and checks the key for validity. If the configuration server is not being used, then the configuration sources are accessed directly through the appropriate backend. If it is, the reference to the ConfigServer instance is retrieved and used to get the reference to the appropriate ConfigDatabase instance from the server through ORBit. If the ConfigServer instance does not exist, then the configuration server is started. With the ConfigDatabase object reference, the appropriate interface function is called and the request is passed through ORBit to the configuration server. The configuration server checks the key's validity and then performs the desired operation on the sources that are associated with the GConfDatabase instance used by the client. For a query, if the configuration server does not find the key in any source and the client specified that the default value could be used, the default value from the schema will be retrieved.

The client registers to receive a notification by first registering the key and a callback function for a client-side notification and then registering the key's directory for a server-side notification. The client library will consolidate server-side registrations to minimize the number of server-side notifications registered on the server. The server will return the identification number used to track the server-side notification, and the client will use this number later to identify the notification being referred to by the server.

After any operation that changes a configuration value, the configuration server builds a list of notifications to make by going through its tree of listeners and adding the notifications registered for each directory from the root directory to the changed key. For each notification, the key, the new value, and the identification number for the notification are sent to the client using the reference to the ConfigListener instance. The identification number is used by the client libraries to find which client registered the server-side notification. The client's cache is then checked for the new value. If it is there, then the client has already been notified of the change and no further action is required. If not, then the new value is cached and the key is added to the client's list of pending notifications. Later, when GNOME gets to it, the pending notification will be removed from the list and the callback registered for the client-side notification will be called.

1.2.3 Why GConf Needs to be Secured

The configuration data stored on a system can have important consequences for the security of a system. Any application with access to the configuration data can read or modify it. There is no separation between the configuration data of two programs, and no way to provide it.

A few examples relevant to security[9]:

- GNOME has a GConf setting to determine whether or not to allow remote access.
- There are gnome-vfs settings to save a username, password, and proxy-server name.
- There are many settings which store the name of a program to run for a menu selection.

The ORBit interfaces are also important. Any process that can read the ConfigServer IOR can get the reference to a ConfigDatabase instance and access the client's configuration sources. Any process that can read the ConfigListener IOR can send a fake notification to the client. The client would cache the value provided by the notification, and return that value when the key is queried in the future.

2. Providing Security Controls over a Program

For any new application it must be determined if it could provide the means to compromise the security goals of the system. On a system using SELinux to provide security, the overall security goals are specified by the security policy and enforced by SELinux. All programs on a system running SELinux are controlled by the SELinux policy. In many cases, adequate control over a program is achieved by running the program in the domain of the process that runs it.

If there are not adequate controls over the program, then either the program must not be run, the security goals of the system must be reduced to allow the program to run, or security controls must be added. The first two options do not involve any changes to the security controls of the system and will not be discussed further.

There are four strategies for adding security controls over a program. These strategies, which can be used by themselves or together, are as follows:

- Add SELinux policy for the program,
- Add additional or finer-grained controls to SELinux,

- Re-architect the program to make use of existing SELinux controls, or
- Modify the program to become an userspace object manager.

2.1. Add SELinux Policy

Adding SELinux policy does not require modification of the program and is the least obtrusive method of adding security controls over a program. In some cases, all that may be required is to run the application in the same domain as a similar program which already has a security policy written for it. An example of this would be netscape and epiphany, which are web browsers, using the mozilla policy. Otherwise, the flexibility of SELinux allows a custom policy to be written for the application. A custom policy for a program involves specifying the security label the process will run in, labeling the security-relevant objects such as files and IPC, and writing rules to allow the appropriate interactions between the process and these objects, the process and other objects, and other processes and these objects.

2.2. Add Additional Features to SELinux

If merely adding policy does not address the security concerns of running that application, then other strategies must be pursued. Another strategy is to add additional or finer-grained SELinux kernel controls. An example of this was the creation of netlink socket classes and additional permissions to provide fine-grained control over netlink objects. This allows an object to be able to send data to the audit subsystem without being able to write firewall configuration data.

SELinux is meant to have comprehensive controls over kernel objects, so new kernel controls shouldn't be required often. If they are, then new policy must be written to take advantage of the new controls.

2.3. Re-Architect the Program

If the first two strategies do not work, then more invasive modifications are needed to control the application. Programs can be re-architected to make better use of the existing SELinux controls. One way this could be done is to decompose an application into a small process that does a few privileged operations and a larger process that handles the normal operation. The small process runs in its own domain and is given the necessary permissions to do the privileged operations. The larger process runs in a domain with less privilege, so if it is compromised it cannot compromise the security goals of the system. This is a good strategy to use even if SELinux is not being used.

An application doesn't always have to be modified to better fit into a SELinux system, sometimes the way it is used can be changed. For instance, instead of running one copy of a program, multiple copies can be run in different domains.

Sometimes more radical modifications are necessary. The Replacement of FAM with Gamin was an example of this. FAM was a system-wide daemon that monitored the system for file changes. It needed to be able to read the entire filesystem and communicate with every process. This made it a huge communication channel. Gamin was structured to be a per-session daemon requiring a more reasonable set of permissions. It only provided a communication channel between a user's programs.[1]

2.4. Create an Userspace Object Manager

Sometimes re-architecting an application is not an option. In this case, the flask architecture can be extended to the application making it an userspace object manager over its objects. SELinux provides object managers for kernel objects, but new object managers are needed for any object not controlled by the kernel. Creating userspace object managers is a natural part of implementing the Flask architecture on Linux. The X server and D-Bus are examples of userspace object managers. To create an userspace object manager, the program must be modified to bind security labels to the objects that it controls, request labeling and access decisions from the appropriate security server, and to enforce the decisions returned by the security server. A userspace object manager is only trusted to control its objects. It does not become trusted in all of its operations and it is still controlled by the system's security policy for it.

There are seven steps in creating an userspace object manager:

- Identify the objects in greater detail,
- Provide a way to uniquely and reliably label the objects,
- Add access checks and labeling requests where needed to control the objects,
- Make the subject's label available at the access checks,

1Gamin has been replaced by a client library that registers to receive an `inotify` from the kernel when a given directory or file has changed. Now there is no daemon and no communications channel.

- Add an access vector cache (AVC) to cache the access decisions of the security server,
- Create new SELinux policy classes and permissions as needed, and
- Create SELinux policy to control the objects.

SELinux supports the creation of userspace object managers by providing an access vector cache in the libselinux library that can be used by userspace object managers. Access decisions are then requested through the AVC. The userspace AVC receives netlink messages from the kernel so that it can flush its cache upon a policy load and modify its behavior to reflect the current enforcing mode.

3. Securing GConf

GConf was selected as an example of securing a security critical application using these strategies. When looking to secure GConf, the following objects must be secured: the configuration sources, the key-value pairs, and the ORBit IORs.

All of the objects, except the key-value pairs have some protection through normal Linux file controls. The default configuration sources are either in `/etc/gconf` and cannot be modified by any user or in the user's home directory and cannot be modified by another user. Likewise, the ConfigServer IOR, stored in `/tmp/gconfd-USER/lock/ior,` and the ConfigListener IOR, stored in `~/.gconfd/saved-state`, can only be accessed or modified by the user.

3.1. Add SELinux Policy

Although configuration data stored in `/etc/gconf` cannot be modified, the configuration data stored in the user's home directory can be modified by any process run by the user. A custom SELinux security policy for GConf can be used to prevent any program other then the configuration server from accessing or modifying the configuration data of the user.

A SELinux policy was created for GConf that runs the configuration server in its own domain and only allows it to access or modify the user's configuration data. Unfortunately, any process run by the user must be able to use the configuration server to manage its configuration data, so any process can access and modify all of the user's configuration data through it. To allow a process to access and modify some configuration data, but deny access to others, there must be a way to assign different security labels to different configuration data.

There is no guaranteed granularity at which the configuration data can be labeled. The single file XML backend can only have different security labels at the configuration source level, since each configuration source is a single file. This is not an adequate granularity to provide the desired security controls. The XML directory tree backend can be labeled at the directory level, because the key directories are actual fileystem directories. It cannot be labeled on the individual key-value pair level, because all of the key-value pairs in the same directory are in the same file. Having the configuration data labeled at the directory level might be adequate to provide the desired security controls, but it would prevent the use of any other backend.

If other backends are going to be supported, then adding SELinux policy alone is not going to provide the desired security controls over the configuration data.

3.2. Add Additional Features to SELinux

There is no need, however, to modify the SELinux kernel controls to accommodate GConf. The configuration data of GConf is only visible to the SELinux object managers in the kernel at the granularity in which it is stored in the filesystem, and SELinux currently controls the filesystem objects at the appropriate level.

3.3. Re-Architect the Program

There are several ways in which GConf could be re-architected to make better use of the existing SELinux controls.

A client could be re-architected to use GConf in a different way. Since GConf allows a client to specify the configuration source to be used, a client could have a private configuration source. Even with the single file XML backend, its configuration source could have its own security label that prevents any other process from accessing or modifying it. But this would require every program that used GConf to be modified and any tool for centrally managing configuration data would have to know about all of the private configuration sources.

Another way that GConf could be re-architected is by removing the functionality that allows the client to directly access the configuration sources. This would be of limited value, though, since it would not prevent a process from accessing and modifying any of the user's configuration data through the configuration server. SELinux policy would still be necessary to control access to the configuration sources.

The advantages of these ways of re-architecting do not offset their disadvantages. The only strategy that remains is to create an userspace object manager.

3.4. Create an Userspace Object Manager

By using only the previous strategies, some progress has been made on securing GConf, but the configuration data is still not controlled adequately. The reason is clear, the configuration data is only visible at the right granularity to the configuration server. The configuration server must be made into an userspace object manager.

3.4.1. Identifying the Objects in Greater Detail

The important objects have already been identified. The configuration sources, ConfigServer IOR, and ConfigListener IOR are filesystem objects and can be protected with normal SELinux controls. The key-value pairs are the objects that GConf is going to control as an userspace object manager.

3.4.2. Labeling the Configuration Data

The first step is to label the configuration data and to persistently store the labels. One approach would be to modify the configuration source backends to store the security label with the data. This has the advantage of keeping the label with the object, which is preferred, but it has several disadvantages. First, GConf can use another configuration system as a configuration source [11]. If the security labels are stored in the configuration sources, then all of the configuration source backends must be made to store security labels. Another problem is that a schema can be used by more than one key. If this is the case, there could be two conflicting labels that need to be stored in the schema. A copy of the schema could be made, but now the copy has to be kept consistent with the original schema.

The method that was chosen to store the security labels was to create a separate namespace for the security labels and to store them as normal GConf value strings in that namespace. All security contexts are stored in the `/selinux` namespace. A security label for a key is found by looking up the value of a new key created by concatenating `/selinux` and the original key. The security labels can either be accessed directly in the `/selinux` namespace with the normal GConf query and set functions or by using new get and set security context operations. The GConf query and set functions will use the get and set security context operations on the server for operations involving security labels. Both the old and new security labels are used to determine if a set security label operation will be allowed. The advantages of this method are that the security label only needs to be stored once and the same security label is used no matter where the key's value is found. There is a danger in using this method, however. If the client is allowed to specify the configuration sources used, it could add a configuration source with the wrong security labels before the configuration source with the real security labels and gain access to any key that it wants. To counter this threat, the security label is always chosen from the default configuration sources. It would also be very easy to have a separate set of configuration sources just for security labels.

These two methods could be combined. To do this, optional backend methods for getting and setting a security context would be created. If a configuration source backend implemented these methods, then the configuration server would use those methods to get a security label if it was also going to get the value from that source. The only reason this method was not used was that recent changes in the GConf XML backend causes an unknown XML tag to return an error. This makes a configuration source modified to store security labels with a new XML tag incompatible with the normal XML backend.

3.4.3. Adding Labeling Requests and Access Checks

An access decision should be made before the state of the configuration data could be changed. Since the configuration sources store the configuration data, the access checks are, in most cases, placed right before an operation on the configuration sources. This allows the security label of a key to be retrieved, an access or labeling decision requested, and the decision then enforced without the operation being performed unless allowed. If the key does not have a security label, then a default security label is used. Following the example of D-Bus and the X server, the configuration server's security context was used as the default security label.

There is one case where the access decision had to be done sooner. When registering a server-side notification, the configuration server adds the notification to its listener tree before calling the function that would pass the request to the configuration sources' backends. The access check was placed so it is done before the notification would be added, and it is not added unless the client process had access to the directory of the notification request.

There are two cases where the access decision had to be requested after the configuration sources were accessed. The first case is when the configuration sources are queried for all of the configuration data in a directory -- keys and values, not just keys. Either the request is accomplished by getting a list of all of the keys in the directory and then looking up the keys for which access is allowed individually, or the list of keys and values are retrieved first and then the key-value pairs for which access is not allowed are removed from the list. The latter option was chosen. The second case is similar and the same approach is used. It occurs when the configuration sources are queried for all of the directories in a given directory. After the list has been returned, the directories for which access is denied are removed.

A labeling request is made when a new configuration key-value pair is created. There is no separate create operation in GConf; if the key is not found in the configuration sources on a set operation, then the key-value pair is created. In addition, if the key does not exist on a query operation, then NULL is returned. The only way to determine if a key exists is to query for the key's metadata; if the key exists, a non-NULL value will be returned. Modifications were made to the set operation so that the following occurs. If a security context or metadata exists, then the set operation is not considered to be creating the value. In this case, a permission check is performed for setting the value and the value is set if the client is allowed. If the key-value pair is being created, then the security label of the key's parent is queried and a permission check is made to determine if the client can create a key-value pair in the parent directory. A labeling request is then made using the parent directory's security label and the security label of the requesting process to determine the security label of the new key-value pair. If the parent directory doesn't exist, then queries are made until an existing ancestor directory is found; at the very least / will exist. When an existing directory is found, a permission check is performed to determine if a new key can be created in it and then the security label for that key is requested. With the new security label, the permission check and label request can now be done on the next lower directory. This continues down to the desired key. If permission was granted for all of the checks and the set operation was successful, then all of the new directories and the new key-value pair are labeled with the new security labels.

3.4.4. Making the Client's Security Context Available

In order to make the access checks, not only is the security label of the key needed, but also the security context of the process making the request. The best place for the configuration server to get this information is from the kernel. Unfortunately, in the case of GConf, since the client process and the configuration server communicate through ORBit it is not possible for the kernel to provide the configuration server with the client's security label.

If the server cannot get the security label of the client from the kernel, then the next best option would be to get it from a process that it trusts. In the case of GConf, ORBit is the one process that could provide the security label of the client to the server. Unfortunately, modifying ORBit to provide this data would require modifying the IDL compilers. This would not be a trivial task.

Since there have been some plans mentioned about moving GConf to D-Bus [8,9], and since D-Bus would be much easier to modify to pass the desired security, it was decided to not to attempt any modifications of ORBit.

In the meantime, the current work is left in the undesirable position of trusting the client library to pass the client process's context to the server over ORBit.

3.4.5. Add an Access Vector Cache (AVC)

The Access Vector Cache (AVC) is provided by the library libselinux. The configuration server was modified to initialized the AVC when it starts. GConf specific memory allocation, logging, and audit callback functions were provided to the AVC.

3.4.6. Create new SELinux policy class and permissions

A new security class was created named gconf with permissions of get_value, set_value, create_value, remove_value, get_meta, set_meta, relabel_from, and relabel_to. See Table 1: GConf Operations and Class Permissions.

3.4.7. Create SELinux policy to control the objects

Now that the configuration data can be labeled and access decisions requested and enforced, policy can be written to take advantage of these controls to properly secure the configuration data. Sensitive keys must be identified along with what processes should be allowed to access or modify them. Processes that need to have different access to the configuration data need to run in different domains. Since most user programs currently run in one domain, this could be a lot of work.

No policy has been written at this time other then what was needed to test the mechanisms for proper function.

GConf Operations	gconf Class Permissions
query_value	query_value
query_metainfo	get_meta
set_value	set_value, create_value
all_entries	query_value
all_subdirs	query_value
unset_value	remove_value
dir_exists	query_value
remove_dir	set_value
set_schema	set_meta
add_listener	query_value
get_security_context	get_meta
set_security_context	relabel_from, relabel_to

Table 1: GConf Operations and Class Permissions

4. Conclusions and Future Work

It has been shown how various strategies can be used to provide security controls over a program. The creation of an userspace object manager was discussed in the greatest detail and GConf was used as an example of how an userpace object manager could be created.

More work needs to be done to secure GConf. The client libraries cannot be trusted to provide the proper security context, so either another approach must be found to pass the security context of the client process or ORBit will have to be modified to do it itself. A more thorough analysis must be done to determine which user processes need access to what configuration data. Finally, the processes run by a user must be separated into different domains so that their access to configuration data can be controlled.

Acknowledgments

I would like to thank Peter Loscocco for his many suggestions, and Steven Smalley and the SELinux Symposium reviewers for their comments on earlier versions.

References

[1] P. Loscocco, S. Smalley, "Meeting Critical Security Objectives with Security-Enhanced Linux," In *Proceedings of the 2001 Ottawa Linux Symposium*, July 2001.

[2] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Anderson, "The Flask Security Architecture: System Support for Diverse Security Policies," In *Proceedings of the 8th USENIX Security Symposium* , August 1999.

[3] P. Loscocco, S. Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System," In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, June 2001.

[4] *GConf Configuration System*, Available at http://www.gnome.org/projects/gconf/index.html

[5] *GConf Reference Manual*, Available at http://developer.gnome.org/doc/API/2.0/gconf/

[6] H. Pennington, "GConf: Manageable User Preferences", In *Proceeedings of the 2002 Ottawa Linux Symposium*, July 2002.

[7] *GConf Reference Documentation*, Available at http://www.gnome.org/~bmsmith/gconf-docs/C/index.html

[8] http://developer.imendio.com/projects/misc/gconf-dbus

[9] http://pvanhoof.be/wiki/index.php/Temporary_location_for_D-Conf_specs

[10] D. Ruiz, M. Lacag, D. Binnema, "GNOME & CORBA", Available at http://developer.gnome.org/doc/guides/corba/html/book1.html

[11] Avi Alkalay, email to Fedora-Devel list, 31 March 2006.