Reprinted from the

Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
Matthew Wilson, *rPath*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
Eugene Teo, *Red Hat, Inc.*
Kyle McMartin, *Red Hat, Inc.*
Jake Edge, *LWN.net*
Robyn Bergeron
Dave Boutcher, *IBM*
Mats Wichmann, *Intel*

Smack in Embedded Computing

Casey Schaufler
The Smack Project
casey@schaufler-ca.com

Abstract

Embedded computing devices are often called upon to provide multiple functions using special purpose software supplied by unrelated and sometimes mutually hostile parties. These devices are then put into the least well protected physical environment possible, your pocket, and connected to an unprotected wireless network.

This paper explores use of the Smack Linux Security Module (LSM) as a tool for improving the security of embedded devices with rich feature sets. The internet enabled cell phone is used to identify application interaction issues and describe how they can be addressed using Smack. The paper compares Smack-based solutions to what would be required to approach the problems using other technologies.

1 Mandatory Access Control

Mandatory Access Control (MAC) refers to any mechanism for restricting how a process is allowed to view or manipulate storage objects that does not allow unprivileged processes to change either their own access control state or the access control state of storage objects. This differs from Discretionary Access Control (DAC) in that a DAC mechanism, such as the traditional file permission bits or POSIX access control lists, may allow unprivileged processes to change their own access control state or that of storage objects. MAC is the mechanism best suited to providing strong separation of sensitive information while allowing controlled data sharing and communications between processes that deal with controlled data.

1.1 Alternatives To MAC

Isolation is easy. Sharing is hard.

Virtualization is currently getting the most attention of all the mechanisms available for providing strong separation. It is also the most expensive scheme, short of multiple instances of hardware, requiring additional processor speed, memory, and storage to provide multiple copies of the operating system. While sharing can be done using virtual network interfaces and authenticating application and system level protocols like NFS, it offers no improvement over having those processes on the same real machine. Further, there is no way to share IPC objects such as memory segments and message queues.

Chroot jails also provide limited isolation. While the filesystem name space can be broken up, the socket and IPC name spaces remain shared. Data sharing can also be achieved using a variety of mount options.

Mandatory Access Controls can isolate the IPC and networking name spaces as well as the filesystem name space while still allowing for appropriate sharing.

1.2 Bell and LaPadula

Prior to the current era of enlightened MAC, the only scheme available was the Bell and LaPadula sensitivity model. This model is a digital approximation of the United States Department of Defense paper document sensitivity policy. This model is fine for its intended purpose, but scales neither upward for more sophisticated polices nor downward to simpler ones. While it is possible to implement interesting protections for embedded systems using this scheme, [1] the combination of rigid access rules, the size of the implementations, and the sometimes excessive price of the products offering it prevented this model from ever gaining traction in the embedded space.

[1] HP actually sold a B&L based email appliance for some time.

1.3 Security Enhanced Linux - SELinux

Security Enhanced Linux, or SELinux for short, is a security infrastructure that provides type enforcement, role based access control, Bell & LaPadula sensitivity, and a mechanism to extend into future realms of security management including, but not limited, to control over the privilege scheme. SELinux associates a label with each executable that identifies the security characteristics of a process that invokes that program. The label applied to the process is influenced by the label of the program, but the previous label of the process has an impact as well. The access control decisions made by SELinux are based on a *policy*, which is a description of security transitions.

For an embedded system, SELinux has some drawbacks. Because the label attached to a program file impacts the security characteristics of the process, programs like busybox that perform multiple functions depending on their invocation have to be given all the rights any of its functions may require. The policy must be programmed to take into account the behavior of the applications, making it difficult to incorporate third party programs. The policy can be large, in excess of 800,000 lines for the Fedora distribution, with a significant filesystem data footprint as well as substantial kernel memory impact. If the policy changes, for example to accommodate a program being added to the system, it may require that the entire filesystem be relabeled and the policy be reloaded into the kernel. Finally, SELinux requires that the filesystem support extended attributes, a feature that can add cost to the system.

2 Smack

The Simplified Mandatory Access Control Kernel (Smack, as a name not an acronym) implements a general MAC scheme based on labels that are attached to tasks and storage objects. The labels are NULL terminated character strings, limited somewhat arbitrarily to 23 characters. The only operation that is carried out on these labels is comparison for equality.

Unless an explicit exception has been made, a task can access an object if and only if their labels match. There is a small set of predefined system labels for which explicit exceptions have already been defined. A system can be configured to allow other exceptions to suit any number of scenarios.

Unlike SELinux, which bases the label that a task runs with on the label of the program being run, Smack takes an approach more in line with that of the multilevel secure systems of the late twentieth century and allows only the explicit use of privilege as a mechanism for changing the label on a task. This means that security is a attribute of the task, not an attribute of the program. This is an especially important distinction in an environment that includes third party programs, programs written in scripting languages, and environments where a single program is used in very different ways, as is the case with busybox.

The label given a new storage object will be the label of the task that creates it, and only a privileged task can change the label of an object. This is another behavior that is consistent with multilevel secure systems and different from SELinux, which labels files based on a number of attributes that include the label of the task, but also the label on the containing directory.

2.1 Access Rules

The Smack system defines a small set of labels that are used for specific purposes and that have predefined access rules. The rules are applied in this order:

- ***** Pronounced *star*. The star label is given to a limited set of objects that require universal access but do not provide for information sharing, such as `/dev/null`. A process with the star label is denied access to all objects including those with the star label. A process with any other label is allowed access to an object with the star label.
- **_** Pronounced *floor*. The floor label is the default label for system processes and system files. Processes with any label have read access to objects with the floor label.
- **^** Pronounced *hat*. The hat label is given to processes that need to read any object on the system. Processes with the hat label are allowed read access to all objects on the system.
- **matching labels** A process has access to an object if the labels match.
- **unmatched labels** If there is an explicit access defined for that combination of process and object labels and it includes the access requested, access is

```
cardfs /card cardfs smackfsroot=ESPN,smackfsdefault=ESPN 0 0
```

Table 1: Mount Options Example

permitted. If there is an explicit access defined for that combination of process and object labels and it does not include the access requested or there is no explicit definition, the access is denied.

2.2 Defining Access Rules

A Smack access rule consists of a subject label, an object label, and the access mode desired. This triple is written to `/smack/load`, which installs the rule in the kernel.

2.3 Unlabeled Filesystems

As previously mentioned, not all of the filesystems popular in embedded systems support the extended attributes required to label each file individually. In some cases, such as that of removable media, it is unreasonable to trust the labels that would be on the filesystem if it did support them. A reasonably common situation involves an embedded system with two filesystems, one that contains all the system data and a second that is devoted to user data and which may be removable. Even if neither filesystem supports extended attributes this is easily supported by Smack via filesystem mount options. The mount options supported by Smack are:

- **smackfsroot=***label* Specifies the label to be used for the root of the filesystem.
- **smackfsdefault=***label* Specifies the label to be used for files that do not have labels stored in extended attributes. For filesystems that do not support extended attributes this will be all files on the filesystem.

An easy way to isolate the system from applications that use external data then is to run the applications with a label other than the floor label and to mount the external data at that label. An entry in `/etc/fstab` for this might resemble Table 1.

The application running with the **ESPN** label can read the system data and modify anything on `/card`. Should the application run a program found on `/card` the process will continue running with the **ESPN** label and will have the same access.

2.4 Networking

Network based interprocess communications are far and away the dominant mechanism for passing information between processes. Smack imposes the same restrictions on writing information to another process as it does writing information to a storage object. The general rule is that the sending process and the receiving process must have the same label. If an explicit rule allows a process with the sender's label to write to an object with the receiver's label then a message can be sent. For UDP packets the sender need only have write access to the receiver. For TCP connections both ends must have write access to the other, but neither is required to have read access.

2.5 Network Labeling

Network labeling is accomplished by adding a CIPSO IP option that represents the sender's label to the packet header. With the label of the sender in hand an access decision can be made at the time of delivery, when the label of the receiver is known.

One label is designated the ambient label. All packets that have no CIPSO tag are given the ambient label. Symmetrically, packets created by processes running with the ambient label are not given CIPSO tags.

2.6 Sockets

Sockets are not themselves elements in the Smack security model. Sockets are data structures associated with processes, and can sometimes be shared. Socket attributes can be set by privileged processes to associate a particular label with outgoing packets and to change the label used on incoming checks. The labels attached to TCP connections and to individual UDP packets can be fetched by server processes.

3 Secure Embedded Systems

There are probably as many notions of what defines an embedded system as there are of what defines system security. For the purposes of embedded systems security, there is a specific set of characteristics that are interesting.

An embedded system will usually be resource constrained. Processor power, storage size, and system memory are only some of the things that can add cost and that are subject to scrutiny and reduction where possible. Security solutions that require significant additional resources introduce cost and may even push a product out of viability.

Embedded systems often do not have multiple users. Google's Android project assumes this to be universal and co-opts the userid mechanism for program isolation. Systems are designed around data flows or application sets rather than providing general purpose user environments. They may also assume that programs have restricted use patterns and limit security concerns to variations from those patterns.

Systems deployed in embedded environments are expected to function for extended periods of time without modification or with as few as possible. It is important to get the software and its configuration correct prior to release as it may never have the opportunity to be repaired. Even those systems that can be repaired in the field will usually require that changes be as few and as small as is absolutely possible. A security scheme based on regular updates to threat profiles would be inappropriate to a flight data recorder.

3.1 Filesystems

Embedded systems can have particularly sensitive filesystem requirements. The devices that they use are often slow and may have media with limits on the number of times it can be updated. They may also be limited in the amount of data they can store. These characteristics in particular encourage the use of filesystems that do a minimum of physical accesses and that are optimized for size in favor of functionality. Support for extended attributes is often eschewed because the additional media space required, the increase in code size, and the consequences of maintaining extended attributes on the media make them unappealing for the environment.

3.2 Networking

Networking is important to embedded systems for interprocess communications and external access.

Because IP protocols do not normally carry any sort of security identification information, application level protocols are often required to provide identification and authentication on their own. This requirement can add significantly to the application size and complexity, the number of libraries required, and the time required to perform communications, especially simple ones.

Some embedded systems communicate with the world at large and for many, including mobile phones and more sophisticated devices, this is their primary function. In many cases it is quite important that information be kept segregated based on its role on the device, and that the information be delivered only to appropriate applications which may themselves have come in over the airwaves. Clearly a mechanism needs to be available for distributing this information safely.

4 The Mobile Phone

It is probably impossible to identify a typical embedded system, but the mobile phone will serve for purposes of discussion because the mobile phone is familiar, some are known to run Linux, and they have obvious security concerns. Those who are unfamiliar with these devices are encouraged to have a look at Google's Android system [2] for a working example of how the software for one of these devices can be assembled.

Mobile phone service providers do not make money by selling phones. They make money by selling services that use the information network with which the mobile phone communicates. It is thus very important to the service company that the system software and user account data stored on the phone be protected from any user of the device. It is also important that access to features of the phone that the user is expected to pay extra for is tightly controlled. It is further a significant concern that only the applications the service provider gets paid for wind up on the phone, otherwise the consumer may not have to buy the provider's offerings to get the functionality desired.

[2] `http://code.google.com/android`

An important but often overlooked aspect of the mobile phone is that one of the most critical design criteria for the software it runs is time to market. It can be the case that a particular date, usually early in the holiday shopping season, is a hard deadline and software development must be completed sufficiently in advance of that date to allow volume manufacturing. Any architecture with a long time to market or that may interfere with the ability to deploy third party applications in a timely fashion will come into question even if it is adopted for reasons of security.

4.1 A Simple Application Example

Let us now consider a mobile phone that incorporates third party applications to provide its differentiation. The third party applications will have been delivered slightly late and will have little or nothing to say about any interactions they might have with their operating environment. The applications will certainly not have gone through any sort of security analysis. If only to prevent the applications from accidently interfering with each other it is prudent to isolate them.

For the sake of simplicity, our phone implements a display manager, a keypad manager, and a radio manager. These managers communicate with applications and each other using UDP datagrams. Each of these managers runs as a separate process, started when the phone is turned on, and all with the Smack label **phone**. Because they all have the same label they can share information freely, but because they are not running with the floor label they cannot modify the system files. All the device files that these managers access are also labeled **phone**.

The first application might be a news update service that queries a database at ABC, presenting a ticker tape of interesting headlines on the display. It is run with the Smack label **ABC**. To achieve this, the application will send messages to the radio manager asking it to call out for updates, and sending text to the display manager to put on the ticker tape. The radio manager needs to send responses to the ABC application. To allow this communication two Smack access rules are required.

- `phone ABC w`
- `ABC phone w`

Notice that read access is not provided in either case. Processes with either label can send datagrams to processes with the other, but neither can read their peer's data.

The second application is from a sports network and offers animated recreations of football highlights. It is run with the Smack label **ESPN**. The application will send messages to the radio manager asking it to call out for updates, and send animation frames to the display manager. The radio manager needs to send responses to the ESPN application and the keypad manager needs to send keystrokes for control purposes. As before two Smack access rules are required.

- `phone ESPN w`
- `ESPN phone w`

Notice that even though both ABC and ESPN processes can communicate with the manager processes they cannot communicate directly with each other.

It turns out that in the example here both applications provide service based on information from a common source, that being the shared parent company of the news service and the sports network. If the sports animation application understands the data stored by the news application and has read access to that information it could pre-load sport event information that appears on the ticker tape, improving the user experience. A single Smack rule makes this possible.

- `ESPN ABC r`

Now the sports animation application can read the news application's data and take whatever actions it deems appropriate. Notice that it cannot execute the news application or search directories because it does not have execute permissions.

If at some point in the future the parent company sells the sports network, access can be revoked without relabeling any files by changing the access rule.

- `ESPN ABC -`

Now access is explicitly denied.

4.2 Software Update

A common problem on embedded devices is live, controlled application software update. While getting new software onto the device may be straightforward, making sure that the transition occurs and that the new software is used while retaining the old in case of unforeseen issues can be tricky. One popular solution to this problem is to provide multiple filesystems, each of which is loaded with a different version of the complete set of software. One filesystems is mounted in the active path while the others are mounted to the side and the transition is made by unmounting the active path and mounting an alternative in its stead. Updates are performed on the out-of-path filesystems.

The Smack solution is to include all possible paths, but to label each set differently, and to determine which gets used by a particular process by access rules. The startup script running with an appropriate label, in this case **ESPN**, sets its path

```
export PATH=/slot-a:/slot-b
```

then invokes the desired program

```
spiffyapp -color -football
```

which will of course use the version in `/slot-a` if it is accessible, and the version in `/slot-b` if it is not. The installer labels `/slot-a` and all the files therein with the same label, for simplicity **Slot-A** and similarly the contents of `/slot-b` with **Slot-B**. To allow access to either version the rules would be

- `ESPN Slot-A rx`
- `ESPN Slot-B rx`

When it comes time to update `/slot-a` setting the access rule

- `ESPN Slot-A -`
- `ESPN Slot-B rx`

ensures that the version in `/slot-b` gets used. Once `/slot-a` is updated setting the access rules

- `ESPN Slot-A rx`
- `ESPN Slot-B -`

ensures that the new version is used. Note that the files in each of the slot directories do not get relabeled as part of this process, they retain the label that they are given by the installer. The only change required is in the access rules.

5 Comparisons and Conclusions

From the examples provided it should be clear that many of the security concerns that are typical of an embedded system can be addressed readily by Smack. It is not enough to provide the security facilities, it is also necessary to provide them in a way that is appropriate to the problem at hand. Other schemes, including virtualization and SELinux, can be used to address specific security concerns, but Smack is better suited to the resource-constrained embedded environment.

5.1 Distributions

The purpose of a distribution is to provide a set of configuration files, documentation, programs, libraries, scripts, and various other digital components with which a complete system can be composed. Most distributions available today are full-featured, offering as complete a set of utilities as possible, often even including multiple alternatives for email services, web servers, and window systems environments. Distributions targeted for the embedded space will offer a slightly different set of content and configuration, but are not fundamentally different from their desktop or enterprise peers.

A MAC scheme based on the behavior of applications will have to be customized to each distribution on which it is available. The Red Hat distributions include customized SELinux policies that match the programs they contain. The SuSE distributions include configurations for AppArmor. Other distributions claim support for SELinux as well.

The embedded systems developer is typically not looking for the advantages of integration that a distribution provides. The embedded systems developer will be carefully choosing the components that go onto the box and while it will be convenient if they all come from the same place it is perfectly reasonable for a legacy version of certain applications to be chosen for size, compatibility, or performance. This is a major problem for

a system like SELinux that depends on specific versions of specific applications for the policy to be correct. A system like Smack that is strictly based on processes, rather than programs, in its security view has a serious advantage.

5.2 User Space Impact

The user space component of a security mechanism ought not to be a major concern for an embedded system. Because Smack rules are trivial, the program that loads them into the kernel need only ensure that they are formatted correctly and can hence be kept very small. Because labels are text strings there is no need for functions that compose or format them. The current Smack user space library provides only two functions.

- `smackaccess` Takes a process label, an object label, and an access string as arguments and returns an access approval or denial based on the access rules currently loaded in the kernel. Using this function an application can make the same decisions that the kernel would. Because the kernel table is readable, any program can use this function to determine what the answer is to a specific access question.

- `smackrecvmsg` This is a wrapper around `recvmsg` that does control message processing associated with `SCM_SECURITY`. It is typically used by label-cognizant server programs that may change their behavior based on the label of a connection. These programs will require privilege to allow connections at multiple labels and will hence be required to be treated as trusted components of the system.

One reason that there are so few library functions is the direct scheme that Smack uses for labeling. Because labels are text strings that require no interpretation, their manipulation is limited to setting and fetching. The existing extended attribute interfaces are sufficient for manipulating labels on files. Process labels are dealt with through the `/proc/self/attr/current` virtual files. Socket labeling is manipulated using `fsetxattr` to set outbound labels and set inbound labeling, but only by privileged processes.

5.3 Configuration Issues

Embedded systems are usually designed to be as simple as possible. Sophisticated configuration requirements go against this design principle the same way that excesses in scripting would.

One problem with a virtualization solution is having multiple operating system configurations to maintain. Another is the hypervisor configuration. Finally, there is the configuration required for the virtual machines to share or communicate.

SELinux is notoriously difficult to administer. Because the security model labels programs based on their behavior, any change, even a simple version update, may require a change to the system security policy configuration. A policy that does not take the entire set of applications on the system into account does not provide the controls necessary for accurate containment. This is true regardless of how much of the full utility of SELinux is actually required to achieve the security goals.

Simplicity is a design goal of Smack. The coarser granularity of access control provided by a process-oriented scheme requires much less detail in the configuration than does a fine-grained scheme such as SELinux. Because it is an access control mechanism that can be configured, it is much easier to use than the multiple configurations required in a virtualized scheme.

5.4 Summation

Embedded systems are not general purpose computers. Smack is intended to address clearly identifiable and specific access control issues without requiring extensive theoretical understanding of security lore. It does not require the intervention of a highly trained security professional. The low impact and strong control of Smack make it ideal for solving the controlled access problems of applications in embedded systems. Freedom from dependence on a distribution makes it attractive to developers inclined to "roll their own" system software. With process oriented access control emphasis can be placed on the pragmatic security issues that matter in the embedded space.