Reprinted from the

Proceedings of the
Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*

Security Policy Generation through Package Management

Charles Levert
Open Systems Lab
Ericsson Research
8400 Décarie Blvd.
TMR (Qc) Canada, H4P 2N2
Charles.Levert@ericsson.ca

Michel Dagenais
Dept. of Computer Eng.
École Polytechnique de Montréal
C.P. 6079, Succ. Centre-ville
Montréal (Qc) Canada, H3C 3A7
Michel.Dagenais@polymtl.ca

Abstract

Generation and maintenance of security policies is too complex and needs simplification for it to be widely adopted and thus truly make a difference in delivering the promise of more secure computing systems (rather than just being ignored by administrators).

In practice, one of the great obstacles to the adoption of security measures in system software is the complexity of configuration that it entails. Yet, information captured by software package management systems is mostly not relayed to security configuration.

This paper covers the investigation to:

- Identify useful information already coded in packages from various package management systems (RPM, dpkg), as well as translation mechanisms to reuse this information.
- Identify missing information that would best be specified by the package integrator and included in each package.
- Identify the remaining information that is mostly site-specific and that would best be specified by a local administrator.
- Prototype the coding of the resulting design ideas.

The approach taken follows these principles: simplicity of design, best security practices as default behavior (i.e., no or minimal configuration/specification required, use of common patterns), flexibility, and least privilege (at each phase: installation, configuration, activation, and execution). It builds on existing parts of the Linux system landscape, without imposing a total revolution: package management systems, the init process and init script system, file system standards and file placement conventions, as well as current security efforts such as SE Linux (to express and enforce the policy).

1 Introduction

> "*Complexity is the worst enemy of security.*" [18]

A package management system provides a structured way to install and de-install software on a computer system. It maintains a database that accounts for (ideally) all files that are not user data on the system. Package management really came of age with operating systems built around the Linux kernel, although other systems with less functionalities, such as the `pkg` system available on Solaris, predate them.

A security policy is an explicit set of rules that govern (the configurable part of) the behavior of a system's security features.

The currently unresolved problem that is the subject of this paper is the following. Software package management systems, such as RPM, already capture much information about the nature of the files that make up a package and about the interactions of a package with other packages, yet this information is mostly not relayed to security configuration. It is quite possible that the complexity of configuration stems from the current requirement to specify, by hand, configuration information that is redundant with what could already be gleaned from packaging information.

Specific parts of the configuration information naturally correspond to the software itself, other parts to its inclusion via a package in an operating system, and yet other parts to a site-local installation. Currently, in the Linux and open source software world, too much of this configuration is unnaturally pushed to the local installation and its human manager. Since security configuration information is not strictly needed for software to perform its main task, it is often left unspecified, which in practice leads to a wide open system from a security standpoint.

It is assumed that the roles of system administrator and security administrator are distinct. Hence, the responsibilities for each of these roles should, as much as possible, center around the very nature of each. For example, software installation by itself should not grant the necessary privileges to activate services with a security impact. Conversely, security administration should not be concerned with the cumbersome details of software installation. In practice, total separation between the two roles may be impossible. However, there is still a gain in security from attempting to separate the two, if only because the human beings that assume these roles are better sensibilized about the responsibility and possible consequences for every action they take.

Note that this paper is not about packaging a security framework in itself. This other important issue is being addressed elsewhere [4].

This paper is organized as follows. The first few sections review existing parts of Linux systems that are pertinent to our endeavor. Section 2 reviews pertinent features of package management systems. Section 3 reviews existing execution control schemes for server processes. Section 4 reviews file system standards and file placement conventions. Section 5 reviews how existing security frameworks are configured. The remaining sections explore how we approach the problem stated in this introduction. Section 6 exposes proposed additions to per-package information. Section 7 then does the same for site-specific information. Section 8 suggests modifications to existing software. Finally, Section 9 covers a prototype implementation.

This work is done in support of the development of the Distributed Security Infrastructure (DSI) [17, 20] open-source effort that is targeted for use in carrier-class (telecom) clusters.

1.1 Basic Principles

- **Simplicity.** All design and code, whether they implement security or non-security features, contribute to the total security of a product. Security vulnerabilities creep in with ordinary bugs when design or code are not produced and then verified (audited) with a security-minded approach. By far the best way to ease this process is to keep things as simple as possible: to have design and code that do only one thing at once, to limit the size and number of functions and modules, to specify things in only one place, etc. This applies to the definition of a security policy framework.

- **Default behavior.** Whenever possible, no explicit security policy rule should have to be specified when a situation is the most typical one. Furthermore, custom startup scripts for server programs with typical behavior should not have to be provided. Good practices, from a security standpoint, should be used as default. On the other hand, bad practices should require explicit manual configuration from the security administrator, so as to discourage them. Security relevant effects (e.g., permission modifications) should be made to be the result of existing actions when that is what one would expect from these actions (e.g., service activation).

- **Flexibility.** If a system prevents its users from accomplishing their tasks, it won't be used. For the security features of a system, it means being turned off, which cancels out all their usefulness. Therefore, there must always be a way to specify behavior that deviates from the default.

  Lack of flexibility can also impose an abrupt transition from an old way of doing things to a new one. This is turn can cause this new way to never take off the ground. Deployment of package managers on Linux systems is pervasive. For the security related package management changes that will be advocated later in this document to be adopted, they must account for flexibility.

- **Least privilege.** The different tasks that are accomplished in relation to a given software package each require their own minimal set of security privileges. These tasks (or phases of system activity) include: installation/De-installation/upgrade, configuration, service assignment/activation, and regular use/execution.

2 Features of Existing Package Management Systems

Package management systems already carry information about their content (installed files, mainly) that can be relevant to the generation of a security policy. Although several package managers are considered in this section, most of the rest of this paper will focus more on RPM.

2.1 Red Hat Package Manager (RPM)

RPM [13] is used by Red Hat distributions, as well as others such as SuSE and Mandrake. All information specified by the maintainer of an RPM package is included in a `*package*.spec` file inside the `.src.rpm` source version of the package. Configuration files can be explicitly designated as such in the `.spec` file of a package. There are several possible declarations (or directives, or file attributes):

- The %config directive is used to flag the specified file as being a configuration file.
- %config(missingok) indicates that the file need not exist on the installed machine. It is frequently used for files like /etc/rc.d/rc2.d/S55named where the existence of the symbolic link is part of the configuration in %post, and the file may need to be removed when the package is removed. The file is not required to exist at either install or de-install time.
- %config(noreplace) indicates that the file in the package should be installed with extension .rpmnew if there is already a modified file with the same name on the installed machine.

(Parts of these descriptions are plain transcripts from [1].)

There is also the %ghost file attribute. It indicates that the file is not to be included in the package. It is typically used when the attributes of the file are important while the contents is not (e.g., a log file).

The package format itself is a binary that can be handled using the rpm library (and the include file <rpm/rpmlib.h>). It begins with a header composed of several tag-and-value pairs. Headers tags are 32-bit integers (RPMTAG_*), so extensions should be possible. File tags (RPMFILE_*) appear as individual bits in an integer, so extensions should also be possible, but there are much less free bits available and conflicts are likely with future versions.

Each package has the opportunity to provide various scripts to be run before and after the installation and de-installation steps.

2.2 Debian's dpkg

Under the Debian Packaging scheme, binary packages are distributed in a single file with a .deb extension. This file is an ar archive that itself contains a file named control.tar.gz. This file is in turn an archive that contains plain text files, including one named control and possibly one named *package*.conffiles. This last file contains a list of installed files, one per line, that are configuration files.

The specification for the .deb file allows for the future inclusion of new members and explicitly defines the behavior that current programs that manipulates those files should take in order to maintain backward and forward compatibility. This makes it easier to add features to dpkg while allowing for a smoother transition.

As for RPM, each package has the opportunity to provide various scripts to be run before and after the installation and de-installation steps.

2.3 OpenPKG

OpenPKG [6] is somewhat a clone of RPM. It targets systems that are not Linux based such as Solaris and FreeBSD, as well as Debian Linux. Support for some .spec tags has been left out, but that does not include any of the tags that are of interest in RPM for the purpose of this investigation.

OpenPKG otherwise includes as an extension an “rc script” that provides centralized application control.

3 Existing Execution Control Schemes for Server Processes

Most Linux distributions rely on an execution control scheme for server processes that is inherited from System V Unix. Under that scheme, the system operates at any given time under a specific run level, which is represented by a small integer that takes its value from a predefined set of values with specific meanings. Each run level defines which services should be activated and which should not. Each service is expected to provide a script that can be instructed to start the service, to stop it, and possibly to inquire about the current status of the service, to restart it, etc. Each script also usually provide suggested run levels under which the service should run, a 2-digit sequence number for service startup, and a 2-digit sequence number for service shutdown. These are provided under a `chkconfig:` field that is located in a commented-out line at the beginning of the script. Every time the run level changes, service are started and stopped in the order that is defined by those sequence numbers. (Often, both sequence numbers are chosen such that they add up to 100, so that shutdown is done in the reverse order as startup). The `chkconfig` utility command is used to configure the activation of services at various run levels. To facilitate writing these scripts, commonly used shell functions are available from a single file that can be sourced. There is no default behavior facility that would prevent having to provide these scripts and their suggested information, even for simple services that fit a common pattern.

These scripts are usually stored in the `/etc/rc.d/init.d` or `/etc/init.d` directory.

As these scripts are provided by the package that implements a given service, they are related to package management. An init script is not always provided by the author of the software that is started by the script. Indeed, this software may have initially been targeted at another type of system, such as BSD, that does not rely on these scripts. In such a case, the init script for that software is contributed by the distributor or third-party packager.

At this point in time, there is no strong coordination between the various distributors as to the meaning of the various run level values, the choice of sequence numbers, the utility shell functions that are provided, or even the instructions beyond “start” and “stop” that can be given to the scripts. As a result, the scripts for the same service that come with different distributions (e.g., Red Hat and SuSE) will not be compatible, and hence the packages themselves won’t be compatible (even if they agree to use the same dynamic libraries and file locations).

The Linux Standard Base (LSB) effort [11] now attempts to standardize many aspects of system initialization in general, init scripts in particular. The LSB standardizes run level definitions and init script actions. It also introduces a smarter way to determine the order in which the scripts should be run when the run level changes. It is based on “Provides: ” and “Required-*: ” declarations, and the new notion of facility names that refer to generic services rather than specific ones provided by a package. Some commonly used functions and the location of the file that scripts should source has also been standardize. A few other details are standardized by LSB, but they are not related to security. The recommendations of the current version of the standard, 1.1.0, are not currently implemented by major distributors such as Red Hat.

The Linuxconf configuration and activation system [8] adds a few conventions for init scripts. It supports the following information fields: autoreload, processname, pidfile, config, probe, description, and override [9]. Note that the config field is then another, redundant way in which configuration files are explicitly identified to the system (see Section 2).

This init scripts way of doing service startup and shutdown is also beginning to show its age. For instance, there is now a need for tight packet filtering rules that need to be changed dynamically at service startup and shutdown. This information is currently not provided in the package, be it in the startup script or elsewhere.

Moreover, it is less than obvious that the init script provided with a package should be trusted to actually perform a stop order. If this is a security concern, a framework needs to be devised to make sure that the service is actually stopped, and perhaps even that its permissions are revoked.

The init process also has the capability to directly launch services. It has the additional ability to monitor and restart them if necessary. Behavior of the init process is configured through the `/etc/inittab` file. Typical services that are put under init control include:

- getty processes that are started on consoles and serial lines to display a login prompt;
- the xdm process that is started on an X Window System console.

The init process is what actually manages the run level on a Linux system.

On Debian, dpkg includes a wrapper named `start-stop-daemon` that features the following security-relevant command-line options:

- `-chuid` changes the user ID before executing the daemon process.
- `-chroot` changes the current directory and then changes the root of the file system to it so that the daemon process is jailed.

This wrapper is commonly used by Debian startup scripts.

Another system is D. J. Bernstein's daemontools and the `/service` directory on which it relies [2]. It obviates the need for a `/var/run/`*`name`*`.pid` file and has the ability to monitor and restart services to insure higher availability. It relies on standard UNIX features to accomplish its task.

Service availability monitoring can be performed at different levels, but it only needs to be performed once. These possible levels are the process such as `init` that starts the service, the init script that wraps around the service, or the service itself (by forking into a monitoring process and a service process).

4 File System Standards and File Placement Conventions

Various Linux distributions and other UNIX and UNIX-like systems reserve directories for specific purposes. They have naming conventions for sub-directories and files within those directories. The conventions also cover the kind of files (i.e., their purpose) that should be stored in these directories as well as the ownership and access rights that they should have. Depending on the operating system, these conventions are more or less stated explicitly.

In the Linux world, there exists a common standard for this known as the Filesystem Hierarchy Standard (FHS) [10].

There are also other, independent proposals. D. J. Bernstein's `/package` hierarchy [3] goes all the way and reuses the file system itself as the database for package management.

This is related to conventions in package management and security policy configuration. Indeed, if

- a given directory serves a very specific purpose,
- conventions related to security policy are in place for this directory, and
- there is the notion that a package's name automatically reserves a subset of the naming space for subdirectories and files within that directory,

then this in itself defines default, clear rules for security policy that can be made to embody good security practices. This removes complexity as there is then no need to specify package-specific rules for the purpose served by that directory, for most packages.

There is an opportunity to extend the set of such directories. For instance, creation of temporary files or named sockets in the `/tmp` directory has historically been the source of many security vulnerabilities. This is because this directory is a public space, no subset of it is reserved to a specific package, and a complex set of steps is then required to make sure that the temporary resource is created securely. These `/tmp` problems could all be avoided by the introduction of conventions that are properly enforced by default on the system.

The downside of such conventions is that they are difficult to adopt instantaneously. The old way of doing things must be supported for some time, while still providing incentives to move to the new, provably secure way.

5 Configuration of Existing Security Frameworks

There are two extreme approaches to specifying the security privileges that a software package (and its various components) may enjoy.

- The privileges are entirely specified by the package itself. The act of installing the package by the system administrator implicitly carries the approval of these stated privileges. The problem is that those privileges can be complex and are not restricted to anything. They are unlikely to be reviewed by the administrator. Moreover, they have to be expressed in the terms of each specific security framework that is to be supported.
- The privileges are entirely specified outside of the package by the security framework. The problem is that all possible packages have to be accounted for in advance. If they are not for a given package, the security administrator has to specify privileges for it by hand. This either lacks flexibility or is unrealistic.

The solution lies in abstracting the whole set of permissions that a package requests in a form that fits in a very small space (e.g., less than a line) and have the system or security administrator approve that explicitly. To that end, there must be a way to express these abstractions and they must be installed/activated beforehand by the administrator.

In order to express them, however, we must gain an idea of the kind of permissions that different existing security frameworks provide. We will examine two popular frameworks, but there are others [16, 21, 19, 7, 14, 12] (these are the ones for Linux).

5.1 Security Enhanced (SE) Linux

SE Linux [15] configuration is performed in two steps. First, an utility named `setfile` is used to assign a security context to every file on the system. This is done using a configuration file (`file_contexts`) that uses regular expressions to full paths of file. This configuration file has been broken down into several *`name`*`.fc` files, one for each covered package, and a `types.fc` file for all other patterns. This configuration operation can be performed during the initial installation of SE Linux, before the system is actually running under it. It can also be performed while running under SE Linux. It is easily conceivable that the operation could be customized to only touch files that are part of a package at package installation time. Note that file names are no longer used once security contexts are assigned to all inodes when the system is running.

The second part of SE Linux configuration is the security policy itself (`policy.conf`), which is then compiled into a binary form under `/ss_policy` and read by the kernel. Type Enforcement rules have been broken down into several files (*`name`*`.te`), one for each covered package, and is combined with other files to form the whole security policy. The security policy has to be reloaded as a whole. This complicates (or at least make more heavyweight) what can be done at package installation time.

5.2 SubDomain

SubDomain [5] relies on a configuration that directly uses the full path names of files. Configuration profiles are stored in the `/etc/subdomain.d/` directory under the name of the program that is executed and subject to control. Sub-processes are covered in a recursive fashion by the syntax. A user-space utility relies on a `sysctl()` interface to feed these rules to the kernel-space part of SubDomain. Add, delete, and replace operations are supported, which means that updates to the in-kernel policy should be possible at package installation time.

6 Proposed Additions to Per-Package Information

6.1 Implicit, Enforced Conventions

As seen in Section 2, package management systems support many file tags to distinguish, e.g., configuration files, from other files. From a security standpoint, one may wish to introduce more tags to identify files that are specifically related to other phases of system activity, such as service activation, as detailed in Section 6.2.

An alternative to this approach is to designate specific locations (typically directories) to necessarily contain files of a given type (i.e., corresponding to what would have been new tags).

In both case, extension mechanisms would be desirable to either add new custom tags, or equivalently to designate new locations. This idea is further expanded upon in Section 6.3 with the idea of generic "abstract" packages.

6.1.1 Naming of Packages

The naming of packages tends to follow some unwritten conventions.

- Packages in a group of related packages normally share a common prefix, although there is nothing to formally separate the prefix from the rest of the name. (Hyphens can be used anywhere else inside a name, be it in a prefix or in a suffix, if any.)
- For packages within a group with that share the same prefix, commonly used suffixes include: `-common`, `-util`, `-apps`, `-tools`, `-extra`, `-devel`, `-client`, `-server`, `-lib`, `-contrib`, `-doc`, `-perl`, `-python`, and `-X11`. Some of these suffixes are sometimes found in a plural form. Some packages even have several suffixes (e.g., `openssh-askpass-gnome`).
- To complicate matters, some suffixes are not preceded by a hyphen, e.g., `kdebase` and `kdelibs`. Fortunately, these specific suffixes do not appear to be relevant to the security nature of a package.

There is no internal representation of this inside the package. If these naming conventions were clearly represented, it would be possible to assign a security policy semantic meaning to them. Some package name suffixes, such as `-util`, imply that the executables contained in the package must not carry or be given any special privileges. Conversely, a package with `-server` as a suffix contains executables that should be given specific privileges when activated to provide the service for which they were written. Although this was originally done to enable only the client or only the server to be installed, it is a good security practice to isolate a server executable and its related files in such a package, provided that the opportunity to assign specific privileges is taken. It is possible that several executable files be included in a server package, one being the main server and the others being there for support (to be executed as sub-processes of the main server). In anticipation of such a case, there must be a clear way to tell which executable is the main server.

6.2 Explicit New Facilities

> *"Any problem in computer science can be solved with another layer of indirection."*
> — David J. Wheeler.

An indirect mapping is introduced between the actual package name and the service name (e.g., TCP port), to which other information can be coupled (interface, address subset, etc.).

This separates the act of installing a package that can implement a service from the act of designating it as being currently responsible for doing so (and thus receiving the necessary privileges for doing so). Conceptually, this designation can be done in a finer grained manner. For instance, different providers for a given Internet service could be enabled for each network interface (internal/trusted and external/untrusted).

Each (transport_protocol, port_number) pair should have its own set of security contexts by default. Explicit configuration can be useful to put a group of ports in a single set of security contexts. SE Linux [15] relies on statements like

```
tcp 25 system_u:object_r:smtp_port_t
```

that have to be configured by hand. This could be generated automatically from the /etc/services file. The SE Linux syntax also allow for a range of ports (e.g., 22-23) to be specified. Specifying a port range requires explicit configuration in all cases, but it is not a frequent occurrence.

The tasks (or phases of system activity) include the following.

- **Installation/de-installation/upgrade.** This is performed by the package manager itself. When installing files and running package-provided scripts, the package manager should minimize its permissions (e.g., by forking a subprocess) so as to only be able to modify the sub-part of the system that is appropriate for the package. A strong file placement standard can tremendously simplify the interpretation of this statement. An installation should not interfere with other packages that are already installed or that could be installed in a sub-part of the system that is reserved for them. Package-provided scripts should not be able to use services that are not strictly related to installation, such as network ports.

  A package that implements a given service (e.g., SMTP) should not, just by virtue of it being installed, be able to activate (designate) itself at the provider for this service. This is a separate action to be performed by the system administrator.

  Typical steps for installation are:

  - Check for validity of package name, type, etc. The package type may require to be specified explicitly by the administrator to signify approval of the permissions that are inherent to it.
  - Add package-specific installation rules to the security policy;
  - perform actual install under the security context that is defined by those rules (including file copy and script execution).
  - Add package-specific security rules for other tasks.

  The security policy rules are inferred from the package name and type. They are not specified by the package itself.

- **Configuration.** Globally, the responsibility for configuration can be assigned to a dedicated management package. This package can then delegate its authority to application-specific configuration packages. This means that a package is not automatically responsible for its own configuration. By default, it should not even be able to probe various unrelated part of the system during installation and execution to adjust its behavior accordingly. Configuration packages may need and be given such permissions, though.

- **Service assignment/activation.** Several packages can implement the same service (e.g.: sendmail, qmail, and postfix are all SMTP mail transport agents). It may be desirable to have more than one installed at once (testing, transition), yet at most one can be assigned the same responsibility. Service activation can be performed by the package manager (by explicit instruction from the administrator, not from the packager) or it can be performed by a service activation manager software, with can possibly delegate its task to more specialized service activation packages (e.g., Internet service activation manager).

- **Regular use/execution.** During regular use, software from a package should be able to read (and only read) its configuration (in `/etc`, possibly store some state in `/var`, etc. Depending on whether it is a service package, an utility package, or other, it can also get other specific permissions, or inherit those of its invoking process. It should not implicitly be able to modify its own installation, configuration, or activation, though.

6.3 Generic Package Types

Examples of generic package types include the following:

- network server programs
- local service programs (e.g., gpm)
- utilities (which require no special permissions other than those passed by their parent process)
  - read-only viewer/browser
  - strict filters
- installer programs
- configurator programs
- activator programs
- security session managers (program that set up a specific security context for others)
- etc.

6.3.1 Generic Abstract Packages

Generic "abstract" packages (named after the object-oriented concept of abstract class) are incomplete packages that merely include a default init script or configuration files. In effect, they define a generic package type. Using this facility, a generic service package could be able to provide an activation script (or declaration), whereas a specific service package could not because such scripts implicitly carry the definition of access rights to be handed to the specific package. Instead, the specific package can be declared as being of the generic package type defined by the generic service package.

"Multiple inheritance" of generic package types by specific packages should be disallowed by default as it can cause problems related to the combination of specific power. Installation of generic packages should stand out and require special attention from system administrators as they effectively imply the permission to install packages that follow the pattern they describe. This, in turn, means that there must be a way to explicitly identify these package as such.

6.3.2 Framework Packages

Some packages specify a framework (e.g., logrotate) under which other packages can register, but only under their own name. This is traditionally done by a `/etc` subdirectory (e.g., `/etc/logrotate.d`). Other frameworks could introduce a `/var` subdirectory instead.

From a security standpoint, a method is required to explicitly label this subdirectory. The package manager must then only allow packages to register there under their own name.

6.3.3 Sample Packages Types

Here are typical permissions that are needed by two sample package types.

The execution of a network server requires the permissions to (among others):

- read its own configuration file(s)
- produce its own pid file (that can also be handled by the availability monitor)
- listen to its assigned service (protocol/port)
- append to own log file (or use log service under its own name)
- spawn modules (possibly under another security context)

Software installation requires the following permissions (among others):

- to install program under same *`name`* or *`name`*`-*`
- to create and populated subdirectories of same name under `/usr/lib`, `/usr/share`, etc.
- to install an initial configuration file named `/etc/`*`name`*`.conf` or placed under *`name`* in `/etc/sysconfig/`

6.4 Self Restrictions

A package should be able to manage its own private space, such as private directories, by imposing additional restrictions through the use of policy rules. In order to do so, the rules should be expressible in a relative syntax that does not require the redundant mention of the package name. Conversely, it should not be possible to specify rules outside of that package scope. This applies to all phases of system activity for that package.

7 Site-Specific Information

7.1 Default Policy for Generic Package Types

In order to reduce the size of the site-local security configuration, each generic package type must be configurable.

For comparison purposes, SE Linux [15] relies on a system of macros to reduce the complexity of the type enforcement policy files it includes for each special user program and server program.

7.2 Per-Package Configuration

Additional restrictions (e.g., read-only server, local non-networked server) should be easily configurable as site-local options.

8 Proposed Modifications to Existing Software

The previous sections have pinpointed their requirements for many modifications to existing software. These are gathered here for every piece of software that is involved. We try not to introduce new programs, but rather to push the additional security checks into existing programs, at the point where they naturally belong.

The following assumes that dynamic updates to the security policy are possible.

- `/bin/rpm` (and other package managers). Explicit service activation scripts (default preferred), distinct from installation scripts, should be introduced, along with command-line options to specify that a service should be activated at installation time. Naming conventions for packages should be enforced; e.g., utils packages should not include executables with special permissions. Special security attributes for files included in a package should be supported; e.g., the server executable in a server package should be identified as such. Namespace conventions in the file system, as well as conventions introduced by framework packages, have to be enforced. New meta-information can be supported by introducing a new, extended,backward compatible, version of the package format. Alternatively, separate package-specific meta-information files can be used to augment the information present in existing packages. Either way, the package manager has to be able to interpret the new information.
- `/sbin/init`. Provide default init script behavior based on package declarations, process tracking, dependable stops (and restarts), and automatic cleanup of temporary storage (e.g., `/var` storage) to prevent keeping state across invocations (if appropriate). This can involve management of `/var/lock/subsys/`*`name`* files.
- `/sbin/service`. This program should have its own security context and it should perform the task that is currently handled by `run_init` in SE Linux. Init scripts should no longer be run by specifying their full path from anywhere in a distribution, but rather by invoking this program systematically. The program should be rewritten in C, rather than being an interpreted script.
- `/sbin/chkconfig`. Activating a service means enabling the initial transition into a package-specific security context (as is currently done in SE Linux with the `domain_auto_trans()` macro and `type_transition` rules). To guard against cooperating malicious packages where one transition into the other, the notion of defined but forbidden security context could be introduced for de-activated services (SE Linux has `neverallow`, but it is an assertion that can cause a policy to be rejected at compilation, and not an actual rule).
- `/sbin/telinit`. Since some services are only activated for a subset of the available run levels, their associated security context will need to be allowed or forbidden according to the current run level. Note that `/sbin/telinit` and `/sbin/init` are usually the same binary.
- Configuration programs (such as `/sbin/linuxconf`). Configuration files for a program should be put in

a security context that is not accessible for modification by the program itself. Configuration programs should be able to transition a subprocess into a security context that can modify those files, as well as restart the program if it is an activated service.

9 Prototype Implementation

The implementation of the ideas exposed in this paper is at a very early stage. Since this work is done in support of the Distributed Security Infrastructure (DSI), which is an open source project, feedback from the community is important before work proceeds to actual implementation. (As of writing, the actual presentation for this paper is two months in the future and progress will have been made on the prototype by then.)

The goals of the prototype implementation are to assess the practicality of the proposed changes and to measure their performance impact on the system.

10 Conclusion

We have explored the possibility of generating a system's security policy, or at least part of it, from the information that is or can be encoded in software packages that are installed on the system.

The approach described in this paper impacts many people: distribution makers, packagers, software designers and implementers, security framework developers, and system administrators.

Since security is a very sensitive subject, community review of this kind of work is primordial. Also, since this work involves modifying many existing subsystems, building commitment from the community is essential.

This work highlights the following requirements on the security policy:

- Dynamic updates. Long running systems cannot afford to be rebooted. A complete reload of the security policy at every one of its modifications also doesn't scale well with the size of the policy itself (which is proportional to the number of packages that are installed on the system).
- Elaborate security contexts. Associated package, run levels, etc., need to be represented in the security contexts to avoid overly complex rules or unnecessary updates to the policy. A balance must be achieved between those concerns and the complexity of the security contexts themselves (and of their evaluation).

References

[1] Edward C. Bailey. *Maximum RPM*. Sams, 1997. `http://www.rpm.org/local/maximum-rpm.tar.gz`.

[2] D. J. Bernstein. Daemontools and its `/service` directory. `http://cr.yp.to/daemontools.html`.

[3] D. J. Bernstein. The `/package` hierarchy. `http://cr.yp.to/slashpackage.html`.

[4] Russell Coker. Packaging NSA SE Linux for Debian. In *Proceedings of*

2002 Ottawa Linux Symposium, Ottawa (On) Canada, June 2002. `http://www.linuxsymposium.org/2002/`.

[5] Wirex Communications. SubDomain. `http://www.immunix.org/subdomain.html`.

[6] Ralf S. Engelschall and Michael Schloh von Bennewitz. OpenPKG. `http://www.openpkg.org/`.

[7] Tal Garfinkel and David Wagner. Janus. `http://www.cs.berkeley.edu/~daw/janus/`.

[8] Jacques Gélinas. Linuxconf. `http://www.solucorp.qc.ca/linuxconf/`.

[9] Jacques Gélinas. Linuxconf Enhanced System V Init Script. `http://www.solucorp.qc.ca/linuxconf/tech/sysvenh/index.html`.

[10] Free Standards Group. Filesystem Hierarchy Standard (FHS). `http://www.pathname.com/fhs/`.

[11] Free Standards Group. Linux Standard Base (LSB). `http://www.linuxbase.org/spec/`.

[12] Serge Hallyn. DTE for Linux. `http://www.cs.wm.edu/~hallyn/dte/`.

[13] Red Hat. Red Hat Package Manager (RPM). `http://www.rpm.org/`.

[14] NAI Labs. Low Water-Mark Integrity Protection for Linux (LOMAC). `http://www.pgp.com/research/nailabs/secure-execution/lomac.asp`.

[15] National Security Agency (NSA). Security Enhanced (SE) Linux. `http://www.nsa.gov/selinux/`.

[16] Amon Ott. Rule Set Based Access Control (RSBAC). `http://www.rsbac.org/`.

[17] Makan Pourzandi, Ibrahim Haddad, Charles Levert, Miroslaw Zakrzewski, and Michel Dagenais. A Distributed Security Infrastructure for Carrier Class Linux Clusters. In *Proceedings of 2002 Ottawa Linux Symposium*, Ottawa (On) Canada, June 2002. `http://www.linuxsymposium.org/2002/`.

[18] Bruce Schneier and Adam Shostack. Results, Not Resolutions. `http://www.securityfocus.com/news/315`.

[19] Huagang Xie, Philippe Biondi, and Steve Bremer. Linux Intrusion Detection System. `http://www.lids.org/`.

[20] Miroslaw Zakrzewski. Mandatory Access Control for Linux Clustered Servers. In *Proceedings of 2002 Ottawa Linux Symposium*, Ottawa (On) Canada, June 2002. `http://www.linuxsymposium.org/2002/`.

[21] Marek Zelem, Milan Pikula, and Martin Ockajak. Medusa DS 9 Security System. `http://medusa.formax.sk/`.