PULSE: a Pluggable User-space Linux Security Environment

A.P. Murray **D.A. Grove**

Defence Science Technology Organisation
PO Box 1500, Edinburgh, South Australia 5111
Email: {alex.murray, duncan.grove}@dsto.defence.gov.au

Abstract

The discretionary access controls (DAC) employed by traditional operating systems only provide system administrators and users with a loose ability to specify the security policies of the system. In contrast, mandatory access controls (MAC) provide a stronger, finer-grained mechanism for specifying and enforcing system security policies. A related security concept called the principle of least authority (POLA) states that subjects should only have access to the specific resources that they absolutely require to function properly at any given time.

Although a number of existing projects (Plash and Polaris) seek to provide POLA implementations, these are not enforced using strong MAC. Conversely, existing MAC implementations (SELinux and AppArmor) do not provide rigorous POLA because they do not provide an effective mechanism for dynamic policy modification based on user preferences.

This paper presents our solution to fill this void, called the Pluggable User-space Linux Security Environment (PULSE), which implements a MAC enforced, dynamic, user-level POLA implementation. Through the use of user-space plug-ins to specify security policy, PULSE provides a high degree of dynamism, flexibility and usability which is not available in existing security architectures.

Keywords: POLA, MAC, RBAC, LSM, Linux.

1 Introduction

The past two decades have seen globally networked computing systems become ubiquitous. These systems run a variety of failure prone software from a wide range of sources, many of which cannot be fully trusted. Unfortunately, these two conditions violate a primary assumption about the access control models of traditional operating systems (including Windows, Linux and UNIX), namely that users only need to be protected from one another and not the software that they use (Loscocco et al. 1998). Despite this, mainstream operating systems mostly employ a technique called discretionary access control (DAC) to implement basic system security (Nat 1987). In this model it is up to the 'owner' of a file to specify the security policy for that file, such as who can access or modify it. In general this often grants users and applications more authority than they really require, which when coupled with malicious or failure prone software can lead to security and reliability problems.

Copyright ©2008, Commonwealth of Australia. This paper appeared at the Australasian Information Security Conference (AISC2008), Wollongong, Australia, January 2008. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 81. Ljiljana Brankovic and Mirka Miller, Eds.

Mandatory Access Control (MAC) systems seek to further limit the authority granted to subjects within a system, and so provide a higher level of security. MAC requires all security policy to be explicitly specified and non-bypassable, enforcing any restrictions upon all users of the system (including the super-user) (Virijevich 2005). A related concept is the Principle Of Least Authority (POLA) (Miller & Shapiro 2003), which specifies that subjects (users and/or processes) should only have access to the specific resources that they absolutely require to function properly at any given time. A strong security implementation should enforce POLA via MAC, to ensure that it cannot be bypassed.

In Sections 2 and 3 we describe some existing POLA and MAC implementations. As we explain in Section 4, none of these systems fully integrate both concepts to provide a true, MAC-enforced POLA computing environment. We present our solution to this problem in Section 5 and analyse its performance and security properties in Section 6. We conclude with a discussion of several issues we have identified that seem to hamper the development of MAC and POLA systems and present some potential directions for future research.

2 Experimental POLA Implementations

A small number of research projects are exploring ways of adding POLA enforcement to existing computing platforms. These generally rely on the concept of *capabilities*. Capabilities are tokens that combine, in one indivisible entity, the designation of a resource with the authority required to perform actions on it (Dennis & Horn 1966).

2.1 Plash

Plash (Seaborn 2007) is a set of tools for implementing practical least authority within Linux operating systems. Plash can enforce POLA over file operations for an existing application by running it in a `chroot()` 'jail', which prevents it from directly accessing the normal file-system. Instead, applications are linked to modified versions of `libc` and some other common system libraries so that any file `open()` operations are mediated by a trusted server that, based on policy, may perform the operation on the application's behalf. Any resultant file-descriptors are passed (like capabilities) back to the calling program, thus enabling it to continue execution without any further special intervention.

2.2 Polaris

Polaris (Stiegler et al. 2004) is package for Windows XP which allows users to configure applications to be launched with minimal authority. It provides the

ability to statically allocate authority, allowing applications access to their required system libraries, as well as dynamic authority allocation when the user chooses to open files. Applications are launched using the Windows RunAs command in unique restricted accounts, which are configured to only have the required least authority. This allows Polaris to utilise the existing security mechanisms provided by the operating system to enforce containment of the application.

3 Existing MAC Implementations

There are a number of existing MAC implementations for Linux, the majority of which are based upon the Linux Security Modules (LSM) framework (Bovet & Cesati 2006).

3.1 Linux Security Modules (LSM) Framework

The Linux Security Modules (LSM) interface was developed to provide a general and flexible security framework within the Linux kernel. It allows a number of different access control models to be implemented as loadable kernel modules (Wright et al. 2002). These access control modules interface with the LSM infrastructure to mediate access to objects within the kernel by using 'hooks' which are executed after existing DAC checks but before the kernel would normally grant access to the requested object.

Access control can be exercised over a number of different object types including `inode`s, `file`s and `socket`s, and allow modules to control certain actions such as the granting of POSIX capabilities (Lin 2005) to processes. One example is the `socket_create()` hook, which is passed the parameters `int family, int type, int protocol, int kern`. The first three parameters correspond directly to the values passed to the `socket()` (Lin 2004*a*) system call. The fourth parameter `kern` is used to indicate whether the socket is a kernel socket or not, and finally the return value is used to allow or deny access.

The LSM interface is utilised by a number of existing MAC implementations for Linux including SELinux (Smalley et al. 2006) and AppArmor (openSUSE 2006). A number of other projects also use the LSM interface for security related purposes including TuxGuardian (da Silva 2006), DigSig (Apvrille et al. 2004) and Dazuko (Ogness 2006).

3.2 SELinux

Initially conceived by the NSA, SELinux (Loscocco & Smalley 2001) is an implementation of the Flask (Spencer et al. 2000) security architecture for Linux. It implements MAC by assigning labels to objects (files, sockets, processes, etc) and controls the interactions between objects based on a security policy, which in turn revolves around the concepts of type enforcement and role-based access control (Runge 2004). Type enforcement labels processes as belonging to a particular domain. Each domain restricts processes by constraining them so they can only access objects within their own or other specifically listed domains. Role-based access control restricts which domains a user may access, although users can have multiple roles.

The combination of object types and user roles provides an expressive and comprehensive model for specifying security policies. So far, however, the complexity involved in effectively managing the extensively detailed security policies that result from this approach has limited SELinux's wide-spread adoption (Virijevich 2005).

3.3 AppArmor

Novell's AppArmor (Nov 2005) provides an easy to use MAC system for restricting the authority of applications that connect to the network, which minimises the damage that can be done in the event that such an application is compromised (Leitner 2006). In contrast to SELinux's very complex policies, AppArmor policies are stored in short text files that can be easily understood and modified.

A policy file lists the path to the executable of the application to protect, along with the actions the program is allowed to perform. Any actions not listed in the policy file are denied. The types of actions that the policy can control include file accesses and the granting of POSIX capabilities. For file accesses, the full path to the files that the application is allowed to access are listed (although there has been some criticism of this because path names do not necessarily correspond exactly with the underlying file-system (Brindle 2006)) along with the type of access allowed (read, write and/or execute). Shell-style pattern matching can also be used, for example `/home/*/.firefox/*` specifies access to all files within the `.firefox` directory of *all* users home directories.

In addition to this very familiar means of specifying access AppArmor also provides a number of tools to aid in the development of policies that allow the system to 'learn' the normal behaviour of an application. A policy file can then be generated that lists the authority needed to allow the application to function normally.

3.4 TuxGuardian

TuxGuardian is an application based firewall that allows the system administrator to control which applications can access the network (da Silva 2006). It utilises the LSM infrastructure to control an application's ability to create and listen on network sockets. Policy is contained within a configuration file that lists the path to an executable binary along with its MD5 hash and the permissions granted to the application (i.e. whether it can create sockets, and act as a server).

TuxGuardian comprises three components: an LSM kernel module to intercept network access events, a root daemon that provides the policy to the LSM module for each access event and a graphical front-end. The graphical front-end is designed to make the system more usable by alerting the administrator when an application is trying to perform a network operation for which no policy has been defined. The administrator can then allow or deny the access and also specify whether the decision should be stored in the policy file for future reference.

4 Limitations of Existing POLA/MAC Implementations

Both Plash and Polaris implement POLA by using capability-like semantics to designate authority. Unfortunately, however, the overall security benefits derived from both of these systems are reduced because each requires the application to be launched within the specific restricted environment to allow POLA enforcement. If the user instead chooses to lauch the application directly these systems will be bypassed, since their POLA mechanisms are not built on top of a strong, kernel enforced MAC layer. SELinux

and AppArmor on the other hand do use a strong MAC layer for authority checks. In their cases, however, although they both provide POLA to some degree, neither provides an effective means to modulate *when* authority is given. Both SELinux and AppArmor allocate too much authority "just-in-case" it is needed, when it would be better to allocate it "just-in-time" (Miller et al. 2005). The root cause of this deficiency is that neither system provides users (as opposed to specialised security administrators) with a simple and direct means to control the authority that is granted to their applications.

TuxGuardian does provide a degree of interactivity by allowing new policy to be created as needed, but it does not provide an easy means of revoking that policy if it becomes necessary to do so in the future. Furthermore, because TuxGuardian's interactive front-end runs as the root user it does not allow policy to be specified on a per-user basis. SELinux and AppArmor also suffer from this problem because they do not differentiate which user actually executes a specific program. In general this results in security policies that allocate too much authority so that programs can operate under all conceivable circumstances. Consider the Firefox example for AppArmor from Section 3.3, for example, which specifies access to the `.firefox` directory in *all* users home directories instead of only for any particular user who is running the application.

The design of TuxGuardian also includes a fundamental flaw since the graphical front-end is designed to be run as the root user, but can in fact be run as a non-privileged user. Since the daemon does not authenticate the front-end, there is no means to prevent non-privileged users from controlling the security policy for the entire system. Although our system (which we present in Section 5) is similar to TuxGuardian in a number of ways, the design is such that we avoid many of the fore-mentioned limitations.

5 Pluggable User-space Linux Security Environment

The Pluggable User-space Linux Security Environment (PULSE) provides the basis for a modular, dynamic, user-centric security framework for Linux operating systems. Although PULSE enforces policies from kernel-space using the LSM framework for MAC, authorisation decisions can be mediated by user-level components. As a result, PULSE can be user-interactive, and allow both the system administrator and users to dynamically control the authority that is granted to their applications. It consists of three main components:

pulsek is a loadable kernel module that hooks into the existing LSM infrastructure to intercept access requests.

pulsed is a system wide daemon running in user-space that acts as an intermediary between pulsek and the user-space plug-ins.

plug-ins running as distinct user-space processes implement the desired access control model. They can be asked by pulsed (on behalf of pulsek) to authorise or deny access control actions on behalf of the system.

An illustration of the system is shown in Figure 1, which presents the case where a user-level process has tried to create a socket via the `socket()` system call. The standard error and discretionary access control checks are performed, and assuming no error occurs, the LSM hook is invoked. Since pulsek is hooked into

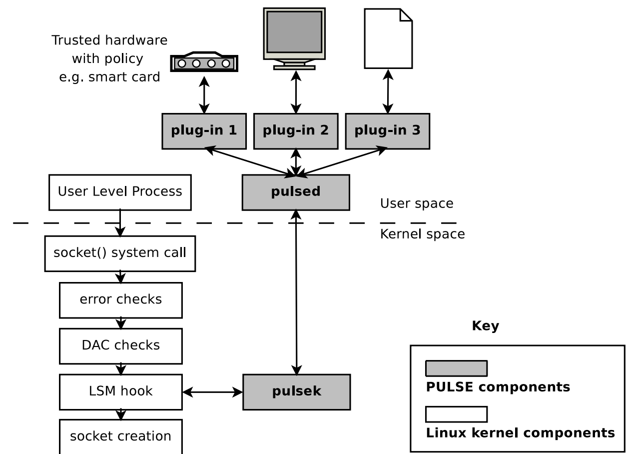

Figure 1: An overview of the PULSE architecture.

the LSM framework and provides a function to perform the access control check, the pulsek module will then query the user-space daemon pulsed, which in turn will query one of the plug-ins. The result is propagated back to pulsek and will be used as the return value for LSM.

5.1 pulsek Kernel Module

The pulsek kernel module facilitates this entire process of delegating access control decisions to user-space by performing three main functions:

Rule Based Access Control by hooking into the LSM infrastructure to intercept access control decisions.

Rule Management as plug-ins request the addition, modification or removal of rules.

Rule Evaluation using the list of rules to evaluate access control requests.

5.1.1 Rule Based Access Control

PULSE uses the concept of rules to specify the access control policy for a system. Rules are created by the user-space plug-ins and used by pulsek to determine the access control action to take for a given event. This rule based approach provides a simple yet effective framework for expressing a dynamic, user-centric security policy. If desired, the user-space plug-ins can interact directly with the user to dynamically create or modify rules. This allows access control decisions to be made in real-time based on prevailing requirements and hence provides a very strict POLA framework (Miller & Shapiro 2003).

In the current implementation of PULSE, access control rules are matched against access control events based on processes and users. Access control rules can specify four types of information:

Executable path name of the process being executed.

Process ID of the process being executed.

User ID of the owner of the process.

Type of event to match, corresponding to each type of LSM hook.

Since each LSM hook takes a number of parameters, 'type of event' rules can specify any or all of these parameters. Any criteria can also be specified

as 'wild' such that they will match any value. Rules are then dynamically evaluated for each intercepted access control action, which provides an expressive means of reacting to access control events. The lifetime of a rule within the kernel begins when it is received by pulsek from the plug-ins (via pulsed) and ends when it is deleted due to either a request from the plug-in, termination of the process which it specifies via process ID, or the removal of pulsek. Rules also specify one of three possible actions to take when they are matched:

Ask for the decision to be delegated to user-space.

Allow the access immediately.

Deny the access immediately.

Plug-ins can use the *ask* action to request that they are consulted about whether an event should be allowed or denied, which will be evaluated in real-time. Currently, if no rule matches a given event then the access will be implicitly allowed. Technically speaking this introduces a loophole in strict MAC-POLA enforcement, but it does allow extant programs to continue operating as normal until more complete policy specifications can be generated. Deployed systems would choose to operate either in this lower security compatibility mode or, where reasonably complete policy specifications are available, in a very secure 'default deny' mode for the price of slight performance and usability overheads.

Allow rules in compatibility mode and conversely *deny* rules in high-security mode may initially seem redundant, but this is not actually the case. Coupled with wild-card matching they may be used to facilitate low-overhead, fine-grained control policies using overrides. In compatibility mode, for example, a general *ask* rule could exist to match a particular access event for any processes executing as a particular user. The results from each *ask* event can then be stored as specific rules for each application as it matches the general rule. This ensures future events for those applications are matched by their specific rules, while all other applications are matched by the more general rule.

For our initial version of PULSE, we decided that a single LSM hook would be sufficient to illustrate the potential functionality of the system. We chose the socket_create hook, which is called when a process tries to create a socket of any kind (Torvalds 2007). This provides the basis for implementing network access control and thus a simple dynamic firewall using the PULSE infrastructure. It also allows for a basic re-creation of the functionality of TuxGuardian but on a per-user basis and with improved performance because rules are cached in the kernel.

5.1.2 Rule Management

Although rule additions, modifications or deletions originate from user-space plug-ins, for performance reasons the rules themselves are always stored within the pulsek kernel module. This allows initial rule processing to be carried out quickly within the kernel, instead of in user-space. In addition, any *allow* or *deny* actions associated with a rule can be cached in the kernel, although *ask* rules must always be delegated to user-space.

The rules are stored within a multi-level index. The first level data structure is a hash-table keyed by the 'owner' of the rule, based on the user ID of the plug-in that created the rule. At the next level, secondary structures are used to store the rules for that user, including:

Process ID hash-table for indexing rules that apply to a specific process ID.

Executable Path Name ordered tree for rules that apply to specific executables.

Remaining Rules ordered trees that apply for a given user ID or LSM event type.

These data structures provide computationally efficient mechanisms for rule evaluation, which is discussed in the following section. In particular, finding the rules that match a specific plug-in user ID and target process ID is just $O(1)$, while finding the rules that match an executable path name or other remaining rule is only $O(\log n)$.

5.1.3 Rule Evaluation

Rule evaluation takes place whenever a PULSE LSM hook is invoked. pulsek searches through the list of available rules to find those that match the current event and hence what action should be taken. Since a number of rules may match, a specific order of precedence is used to ensure consistent and secure behaviour. This is shown in Figure 2 and described

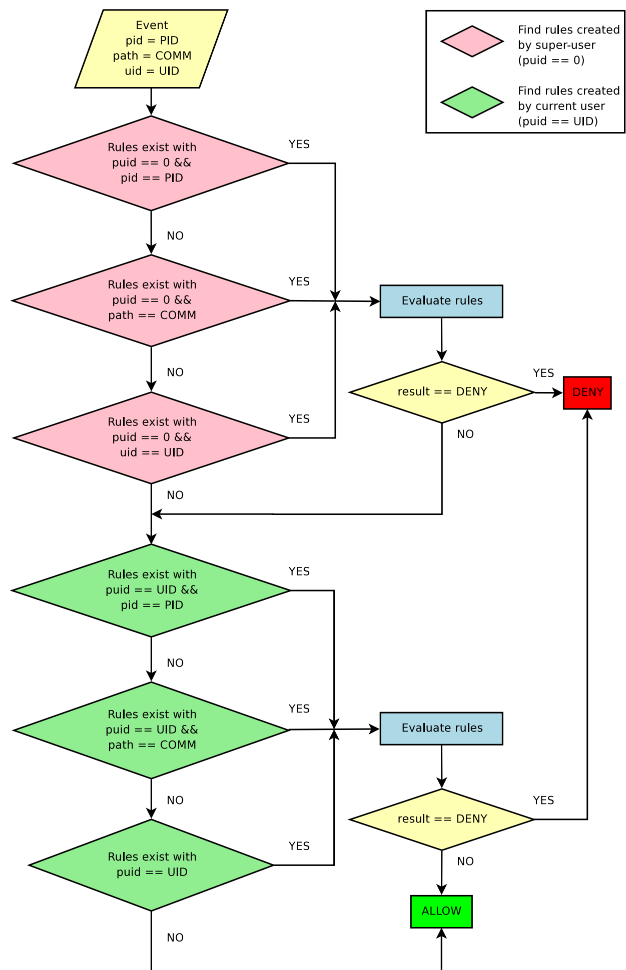

Figure 2: Flow-chart of rule evaluation within pulsek

further below.

Rules created by the system administrator (i.e. where the plug-in user ID is '0') have precedence over rules created by regular users. This allows the system administrator to set mandatory policies that may not be overridden. Users may only further restrict any authority that has already been given to them.

Precedence relations also exist for the various criteria specified by rules. Consider, for example, the case where two rules match a given access control event, the first specifying only the executable path name and a second specifying only the process ID. The rule matching on process ID has greater precedence because it is more *specific*: it applies to a unique process, whereas the executable path can match a number of processes. In the case where multiple plug-ins specify identical rules but with different actions, the result of the individual rule evaluations are combined via the boolean AND operation to determine the final action (where '0' represents *deny* and '1' represents *allow*). Consequently any plug-ins' *deny* action is treated emphatically, overriding any other permissive rules and thus assuring fail-secure semantics.

For each access control check that results in delegation to user-space (as the result of an *ask* rule), the calling process must wait for the delegate's response before continuing execution. To ensure that the process requesting the access does not impact on the execution of other processes in the system, it is put to sleep until the policy decision has been made. It is then woken up using the Linux kernel's `completion` mechanism, a semaphore construct designed specifically for this sort of process synchronisation (Corbet et al. 2005).

5.2 pulsed User-space Daemon

The pulsed daemon provides the link between the access control *infrastructure* within the kernel and the access control *policy* provided by the plug-ins. Communication between pulsek and pulsed is carried over a Netlink socket (Lin 1999), which provides a dynamic, bidirectional communications channel. On the other side of the equation, pulsed listens for connections from plug-ins over a named UNIX domain socket. pulsed then multiplexes / demultiplexs any communication between the plug-ins and the kernel. It can also limit the authority of plug-ins because it has the opportunity to process any data before it is sent to the kernel.

To protect the integrity of the rules database, pulsed requires that plug-ins provide their credentials (process and user ID) whenenver they initiate a connection. Secure credential exchange is guaranteed by the kernel, which ensures that credentials passed over a UNIX domain socket are accurate (Lin 2004*b*). This entire process ensures that any future rules that the plug-in creates specify the correct user ID; any rule that does not is dropped. Plug-ins running as the super-user (i.e. plug-in user ID is '0'), however, are exempt from this check and are allowed to create rules that apply to all users. This allows the system administrator to create rules that affect all users' processes, but limits users to creating rules that affect only their own processes.

5.3 User-space Plug-ins

The plug-ins provide the most novel elements of PULSE: a framework for completely dynamic and user-centric authority delegation. Since the plug-ins run as separate processes they can implement any security model of their choice, including user-interaction.

To illustrate user intervention we created a sample plug-in that uses the libnotify package (Hammond 2006), which provides a simple means of user interaction through notification messages that appear on the user's desktop. An example of a notification created by the sample plug-in is shown in Figure 3.

Figure 3: Notification message created by a sample plug-in.

Upon initialisation the plug-in will register itself with pulsed, providing its process, user and group ID. It creates an example rule that will *ask* for access control authorisation when the Firefox application creates an AF_INET (IPv4) socket. The plug-in will then notify the user of any *ask* request it receives in response to this rule and present them with a number of options to either allow or deny the access, and to specify whether this decision should be remembered in the future. Hence the user is able to prevent the application from using the network by denying it the ability to create sockets.

We realise, however, that forcing users to make such decisions is frustrating and interrupts their workflow. As a result it may cause even knowledgeable users to be dismissive of such security prompts and unhelpfuly teach them that security issues obstruct their normal tasks and should be ignored (Yee 2004). While PULSE supports such fine-grained user-selected security decisions, this is not the only possible source for security policy. A better alternative for a security system that employs user-interaction may be to try and infer authority from the user's behaviour. If a user launches their web-browsing application for example, it is clear that they wish to access the Internet and so the browser should be implicitly granted this authority. Such a rule could easily be specified by an appropriate plug-in. In a more systematic approach, rule creation could be controlled via the shell (either command-line or GUI) that was used to launch the browser, which would create rules dynamically in response to the user's actions. For example, in the GNOME Desktop Enviroment, the user launches applications from a graphical menu. The list of applications, including their name, description, icon and executable path, are specified in *Desktop Entry* (Brown et al. n.d.) files. As an extension, default policies for applications could also be listed in these files and the menu application could *plug-in* to PULSE. The menu itself would now dynamically create rules to grant the specified authority only when the application was launched, and only for that specific instance of the application. Not only would this provide a true POLA implementation, but it would also have a high degree of usability.

User interaction plug-ins are not the only security policies that can be implemented in PULSE. Since plug-ins run in user-space, they provide an extremely flexible security policy framework. Another source of security policy that could be used was foreshadowed in Figure 1, which depicted a trusted hardware device with a complementary PULSE plug-in. Some real-world examples of such devices are the Australian Government Defence Science Technology Organisation's *Codestick* (Anderson 2004, Grove et al. 2007), which is a personal high assurance credential exchange device, or smartcard systems. In the case of the Codestick, for example, an appropriate PULSE plug-in could allow security policy stored on the device to oversee access control decisions on the user's PC. In a typical scenario a user would authenticate themselves to the Codestick using biometrics, via a fingerprint scanner, which would then authenticate

the user to their machine and log them in. On top of this, PULSE would then query the security policies stored on the Codestick whenever the user attempted to access certain restricted resources. One example, perhaps in a bank, may be that a certain power user's Codestick would allow them to access the Internet, while another user may only be authorised to access the corporate intranet. In other cases, for example where a government department such as Defence has hired an on-site contractor, this could be reversed so that they have access to the Internet but not the internal corporate network. Other examples could be policies that allowed particular users access to nominated file servers, directories or printers, or to only obtain access at certain times. Because of the way Codesticks communicate with each other, these security policies could be updated in person by a security administrator using peer to peer transfers, or securely distributed across a network using a related security device from the same product family, called the MiniSec (Grove et al. 2007). Less high assurance but less expensive versions of this framework could use NIS+ or LDAP to distribute PULSE security policies across a local area network. Another research-oriented security plug-in that we have in mind to develop is an artificial intelligence based plug-in that can learn and enforce security policies, for example based on actions selected by a user through the user interaction mechanism.

6 Discussion

PULSE allows rule-based security policies to be created or removed at any time, which allows plug-ins to dynamically control the authority of other processes. It is enforced using LSM, a strong, kernel implemented MAC layer. These features provide the basis for a rigorously enforced, true POLA implementation, where each user can moderate the authority of their own processes in real-time. Furthermore, as a result of the design of LSM, the authority specified by any PULSE plug-ins is a strict subset of the authority granted by the standard UNIX security model (Wright et al. 2002). A plug-in can only deny an access that would otherwise have been granted, hence the maximum authority that a plug-in can 'grant' is that which is normally provided by the standard discretionary access control model.

PULSE ensures its own internal security and integrity by protecting the communication links between its components, although a small amount of work is still required to implement authentication between pulsed to pulsek and to mitigate against the potential denial of service if their Netlink connection fails. While not implemented in a systematic manner, controlling which applications are able to specify the security policy (and hence act as PULSE plug-ins), could easily be achieved by utilising the existing PULSE infrastructure to allow or deny applications from connecting to the UNIX socket of pulsed. PULSE also ensures that the authority given to the plug-ins is limited, in that they can only create rules that affect other processes run by the same user. This is a marked improvement over TuxGuardian, where a single front-end is used to provide access control decisions for *all* applications, even if the front-end is run by an unprivileged user.

Another advantage of PULSE over TuxGuardian is its improved performance because access control rules are cached in the kernel, although the extent of the benefit will vary widely depending on the rules created by the plug-ins. If the plug-ins create many *ask* rules then there will be a large number of deferrals to user-space with a considerable performance cost. The impact that PULSE has upon the normal operation of the system can be minimised, however, by preferring *allow* and *deny* rule types, which cache results within the kernel.

It is important to remember, however, that the security provided by the PULSE architecture is only as good as the access controls implemented by the plug-ins. Even though plug-ins are limited to dictating policy for processes run by the same user, they must still correctly enumerate that user's security policy. For example, if a plug-in requires user interaction to authorise or deny an access control decision it is imperative that the user is knowledgeable enough to ensure they make the correct decision. Whether this is reasonable is debatable. Even if a plug-in does not require user-interaction, however, it must still ensure that the security policy it provides is correct and secure. Non-interactive security policy plug-ins should therefore be stringently programmed and verified in accordance to appropriate standards and methodologies, such as the Common Criteria security evaluation and validation scheme (*Common Criteria Portal* 2007).

Even though it is a long way off, PULSE clearly suggests that it may be possible to implement a complete, MAC enforced, user-level POLA system. During our research, however, is has become apparent that trying to claw back any excess authority that has already been granted, as must be done to retrofit POLA to any traditional discretionary access control based software, results in staggeringly complex security policies. This is clearly apparent in SELinux, the most mature of the existing MAC implementations. We are aware that a complete POLA implementation using PULSE would suffer from a similar explosion in the size of its security policy, perhaps rendering it infeasible. We fervently believe, therefore, in the need for access control frameworks that directly embody the principles of MAC and POLA. A strong contender for this must surely be capability-based software, where the operations a subject can perform are defined by those those that they are *explicitly* authorised to carry out, rather than those they are explicitly disallowed from taking.

7 Conclusion and Future Work

In Section 1 we outlined the need for MAC enforced POLA security in modern computing environments. Sections 2 and 3 presented a number of existing POLA and MAC implementations. Each of these have shortcomings, which we covered in Section 4. We presented our response to this problem called PULSE in Section 5, which provides a strong basis upon which to build enforceable, dynamic, user-centric and therefore POLA-based security policies. Finally, Section 6 analysed the performance and security properties of PULSE, discussed several issues we have identified that seem to hamper the development of MAC and POLA systems, and presented some potential directions for future research.

The flexibility of the PULSE architecture allows for a number of future developments. Further work on adding support for other LSM hooks and integrating these into the existing plug-in would provide a marked improvement in the current functionality of PULSE, and could be done with minimal effort due to the flexibility of the system. Development of additional plug-ins to integrate PULSE with existing user security devices is also planned.

References

Anderson, M. (2004), 'Credential communication device'. WIPO International Publication Number WO 2004/109973 A1.

Apvrille, A., Gordon, D., Hallyn, S., Pourzandi, M. & Roy, V. (2004), DigSig: Run-time authentication of binaries at kernel level, *in* '18th Large Installation System Administration Conference', USENIX, pp. 59–66.

Bovet, D. P. & Cesati, M. (2006), *Understanding the Linux Kernel: From I/O Ports to Process Management*, O'Rielly.

Brindle, J. (2006), 'Security anti-pattern: Path based access control'. http://securityblog.org/brindle/2006/04/19/security-anti-pattern-path-b%ased-access-control/.

Brown, P., Blandford, J., Taylor, O., Untz, V. & Bastian, W. (n.d.), 'Desktop entry specification'. http://standards.freedesktop.org/desktop-entry/latest.

Common Criteria Portal (2007). http://www.commoncriteriaportal.org/.

Corbet, J., Rubini, A. & Kroah-Hartman, G. (2005), *Linux Device Drivers*, 3rd edn, O'Rielly and Associates.

da Silva, B. C. (2006), 'TuxGuardian - documentation'. http://tuxguardian.sourceforge.net/documentation.php.

Dennis, J. B. & Horn, E. C. V. (1966), 'Programming semantics for multiprogrammed computations', *Communications of the ACM* **9**(3), 143–155.

Grove, D., Murray, T., Owen, C., North, C., Jones, J., Beaumont, M. & Hopkins, B. (2007), An overview of the Annex system, *in* 'submitted to the Annual Computer Security Applications Conference'.

Hammond, C. (2006), 'Galago project homepage'. http://www.galago-project.org/news/index.php.

Leitner, A. (2006), 'Counterpoint: AppArmor vs SELinux', *Linux Magazine* (69), 40–42.

Lin (1999), *Netlink - Communication between kernel and user - Manual Page*.

Lin (2004*a*), *socket - Linux socket interface - Manual Page*.

Lin (2004*b*), *unix, PF_UNIX, AF_UNIX, PF_LOCAL, AF_LOCAL - Sockets for local inter-process communication - Manual Page*.

Lin (2005), *Linux Capabilities - Manual Page (2.6.14)*.

Loscocco, P. & Smalley, S. (2001), Integrating flexible support for security policies into the Linux operating system, *in* 'Proceedings of the FREENIX Track: USENIX Annual Technical Conference'.

Loscocco, P., Smalley, S., Muckelbauer, P., Taylor, R., Turner, S. & Farrell, J. (1998), The inevitability of failure: The flawed assumption of security in modern computing environments, *in* '21st National Information Systems Security Conference', National Security Agency.

Miller, M. S. & Shapiro, J. (2003), Paradigm regained: Abstraction mechanisms for access control, *in* 'Advances in Computing Science, ASIAN 2003 Programming Languages and Distributed Computation', Vol. 2896, pp. 224–242.

Miller, M. S., Tulloh, B. & Shapiro, J. S. (2005), The structure of authority: Why security is not a separable concern, *in* P. V. Roy, ed., 'Multiparadigm Programming in Mozart/Oz', pp. 2–20.

Nat (1987), *A Guide To Understanding Discretionary Access Control In Trusted Systems*.

Nov (2005), *Protecting Systems with Novell AppArmor*.

Ogness, J. (2006), 'About Dazuko'. http://www.dazuko.de/about.shtml.

openSUSE (2006), 'AppArmor detail'. http://en.opensuse.org/AppArmor_Detail.

Runge, C. (2004), SELinux: A new approach to secure systems, Technical report, Red Hat, Inc.

Seaborn, M. (2007), 'Plash: tools for least privelege'. http://plash.beasts.org/.

Smalley, S., Vance, C. & Salamon, W. (2006), Implementing SELinux as a Linux Security Module, Technical report, NAI Labs.

Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D. & Lepreau, J. (2000), The Flask security architecture: System support for diverse security policies, *in* 'Proceedings of the 8th USENIX Security Symposium', pp. 123–139.

Stiegler, M., Karp, A. H., Yee, K.-P. & Miller, M. (2004), Polaris: Virus safe computing for Windows XP, External HPL-2004-221, HP Labs.

Torvalds, L. (2007), 'Linux kernel source code'. http://www.kernel.org.

Virijevich, P. (2005), 'Securing Linux with mandatory access controls'. http://security.linux.com/article.pl?sid=05/02/11/2017218.

Wright, C., Cowan, C., Morris, J., Smalley, S. & Kroah-Hartman, G. (2002), Linux Security Modules: General security support for the Linux kernel, *in* 'Proceedings of the 11th USENIX Security Symposium'.

Yee, K.-P. (2004), 'Aligning security and usability', *Security and Privacy Magazine, IEEE* **2**(5), 48–55.