Proceedings of the 2004 IEEE
Workshop on Information Assurance and Security
United States Military Academy, West Point, NY, 10–11 June 2004

Towards the Specification of Access Control Policies on Multiple Operating Systems

Lawrence Teo and Gail-Joon Ahn

Abstract— **In the past, operating systems tended to lack well-defined access control policy specification languages and syntax. For example, a UNIX operating system that is based on the Discretionary Access Control (DAC) paradigm has decentralized security policies based on technology that has been developed over the years. With such policies, it is difficult to identify the permissions given to each user, and who has what access to which resources. With the advent of recent security-enhanced operating systems such as SELinux, this is no longer the case; the access control policy for almost all resources is now stored centrally and applied universally throughout the system. This is certainly more manageable but is not without costs. Firstly, such policies tend to be complex. Secondly, as more of such systems are developed, each system would have its own policy specification syntax. A system administrator who intends to evaluate or migrate to a new system would have to learn the syntax of the new system. In this paper, we propose a solution to this problem by introducing the initial design of a new policy specification language that can be used to represent access control policies for multiple operating systems. To serve its purpose, this language must be flexible enough to cater to many operating systems, while being sufficiently extensible to support the specific features of each target operating system. We present the criteria, features, and approach that we are using to design the language. We also describe the role of two systems – SELinux and Systrace – in the design of our language. We also discuss our consideration of ASL as a potential candidate language, and why we chose to design our own language instead.**

I. Introduction

In the past, operating systems that are based on the Discretionary Access Control (DAC) paradigm tended to lack well-defined access control policies to resources. For example, consider the filesystem of a typical UNIX system. Each file in the UNIX filesystem is designated to have either read (`r`), write (`w`), or execute (`x`) permissions for the user/owner, group, and other users. However, there was no central policy that specifically defines the permissions and ownership for all files universally throughout the system. As the number of users grow, the access control policy becomes even less clearly defined in such systems – this results in a so-called "spaghetti of intent" scenario, which makes it very difficult to identify the permissions of a resource and who has access to it. It is clear that this ad-hoc method of specifying policies is no longer sufficient.

L. Teo: University of North Carolina at Charlotte and Calyptix Security Corporation, Charlotte, NC. Email: lcteo@uncc.edu
G.-J. Ahn: University of North Carolina at Charlotte, Charlotte, NC. Email: gahn@uncc.edu

More recently, security-enhanced operating systems such as NSA Security-Enhanced Linux (SELinux) [?] have introduced a more structured way of specifying policies. Unlike earlier UNIX systems, the SELinux policy is specified in a central directory of files, which is then compiled by a special SELinux utility. In terms of security, access control in SELinux is more manageable from the administrator's point of view. Access control to resources such as files and directories are clearer to the system administrator.

Apart from SELinux, there are many other similar security-aware or security-enhanced operating systems and operating system-related applications that are available today with like or different goals. A few of these are full-fledged operating systems (such as Trusted Solaris), while others are enhancements that are applied to existing operating systems (such as grsecurity [?] and OpenWall for Linux). Yet others are applications that provide security features to an operating system that do not support those features natively. An example of the latter is Systrace [?], which is a tool that enforces system call policies on applications such as daemons and other programs, so that they are only restricted to executing the system calls defined in the system call policy. To simplify our discussion, we shall refer to these operating systems and related applications collectively as "security-aware systems" throughout this paper.

The number of such security-aware systems are bound to increase in the future. These systems are written by separate groups of developers with different objectives. This presents a problem to a system administrator. A systems administrator who wishes to install or evaluate different operating systems would have to learn the policy syntax for each of these systems. Since the syntax for each system varies in terms of complexity and notation, a lot of time and resources would have to be spent in order to learn and implement them effectively. This is also true when the system administrator is trying to migrate an existing system to a different one. How can the administrator be sure that the policy implemented on the new system matches the old policy? Are there any "leaks" or incompatibilities in the new policy?

A solution to this problem is clearly needed. We believe that an effective solution to the problem is to introduce a new policy specification language that is both *flexible* and *extensible*. Flexibility means that the language should be

0-7803-8572-1/04/$20.00 ©2004 IEEE

able to cater to multiple operating systems. Extensibility means that our language should be able to support the specific features in the policies of each target operating system or target application that the language is used for.

One of the most widely cited and accepted authorization specification languages that may be suitable for this purpose is the Authorization Specification Language (ASL) [?], [?]. ASL is a logical language that is designed with flexibility in mind. ASL can flexibly adapt to many kinds of systems, including databases, operating systems, and filesystems. ASL has also been used to specify privacy policies [?]. This demonstrates the viability of ASL as a flexible authorization specification language. However, since its introduction in 1997, ASL has not been fully practiced in commercial systems. To examine the feasibility of ASL to resolve the problems and issues we mentioned earlier, we attempted to implement part of ASL to represent Systrace policies. Through our implementation exercises, we discovered that ASL lacks a number of design criteria that are required to specify the policies of security-aware systems. Our main finding was that ASL is flexible, but it is *not* extensible.

In this paper, we introduce the initial design of a new language which we shall call Chameleos. Chameleos is designed to represent the access control policies of security-aware systems in a flexible and extensible manner. We discuss the criteria we determined to be important in order for Chameleos to be developed, the approach we used, features of Chameleos, and our ongoing work. We also discuss the role of Systrace and NSA SELinux in the design of Chameleos.

The rest of the paper is organized as follows. We first discuss background and related work in Section II. We then describe ASL in greater detail, along with its benefits and shortcomings in Section III. We also show our implementation exercises with Systrace in this section. In Section IV, we present the design, criteria, and features of the new language Chameleos. This is followed by a discussion of our progress in developing the language based on using actual system policies in Section V. We then proceed to describe our ongoing and future work in Section VI, before concluding in Section VII.

II. Related Work

The most relevant work that is related to our project is the Authorization Specification Language (ASL) by Jajodia et al [?], [?]. ASL is a flexible and very expressive language that can be used for multiple access control policies. It is a widely accepted language in the access control community, as can be seen by its adoption and exploration by researchers working in different areas like modular authorization [?] and logical access control frameworks [?]. ASL has also been extended to support other policies apart from authorizations, such as privacy policies [?].

Early projects on flexible languages include the work of Woo and Lam [?], who used default logic to model authorization rules. In the operating systems area, Rippert [?] has proposed a kernel-based framework called THINK to protect flexible operating system architectures.

Throughout this paper, we will frequently use two security-aware systems as examples to demonstrate ASL and Chameleos. The two systems are Systrace and Security-Enhanced Linux (SELinux). Systrace [?] is an application that enforces system call restrictions on programs. It currently runs on a variety of UNIX systems. SELinux [?], [?] is a research prototype of the Linux kernel, along with a number of specially patched programs to use the kernel enhancements. Originally based on the Flask operating system [?], SELinux now includes new architectural components that provide mandatory access control policies involving type enforcement [?], role-based access control, and multi-level security.

III. Access Control Policies on Operating Systems

In this section, we will examine an implementation exercise, where we attempt to specify the access control policies of a real world security-aware system using ASL. The system we will examine is Systrace, an application that enforces system call restrictions on individual programs. We will first introduce Systrace, followed by our experiments using an ASL implementation to represent its policies. We then describe our findings.

A. Systrace

Systrace [?] is a system that enforces system call policies by constraining the application's access to the system. Systrace is currently available for the OpenBSD, NetBSD, Linux, and Mac OS X platforms. By confining applications to a restricted set of system calls, Systrace allows the administrator to sandbox applications. This is particularly useful to examine untrusted binaries and other suspicious programs. It can also act as part of an intrusion detection system that triggers whenever an application violates system call policies.

An excerpt of a Systrace policy is shown in Figure 1. This policy is for `named`, the DNS server that is part of the BIND 9 distribution. The policy is rather intuitive. The `native-` prefix states which Application Binary Interface (ABI) the policy is for.

In Figure 1, the first line permits `named` to invoke the `getuid()` system call. The next two lines specify that `named` is allowed to perform read system calls on a file, as long as the filename is either `/etc/hesiod.conf` or `/dev/arandom`. Note that `fsread` is not actually a system call. In the context of Systrace, `fsread` refers to the group of system calls that perform read operations, such as `stat()`, `readlink()`, `access()` and so on. We also see

```
native-getuid: permit
native-fsread: filename eq "/etc/hesiod.conf" then
               permit
native-fsread: filename eq "/dev/arandom" then
               permit
native-chroot: filename eq "/var/named" then permit
native-bind:   sockaddr match "inet-*:53" then permit
native-bind:   sockaddr eq "inet-[0.0.0.0]:0" then
               permit
native-fchown: fd eq "5" and uid eq "70" and
               gid eq "70" then permit
```

Fig. 1. Subset of the systrace policy for "named", the DNS server.

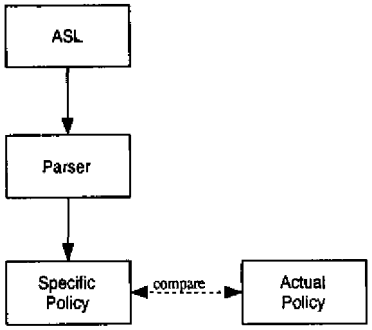

Fig. 2. Our methodology for the implementation exercises.

that named is allowed to invoke the chroot() system call on the /var/named directory. Systrace also supports regular expressions, as we observe from the line that allows bind() to be invoked only on port 53 ("inet-*:53"). Note that expressions using regular expressions use the match operator. The conjunction of boolean expressions is also supported, as seen from the last line, where fchown() is only allowed on file descriptor 5 and when the UID and GID of the process are both 70.

B. ASL Representations

We will now describe our experiments with using ASL to represent the Systrace policy presented earlier. It should be noted that ASL is a logical language, with no known implementation at this time. As such, we have used ASL in a "computer-readable syntax" based on our own interpretation of how ASL's logical syntax should look like if ASL was actually implemented. Our methodology for these implementation exercises are shown in Figure 2.

Also, because of the nature of the Systrace policies described, we ascertained that the authorization rule of ASL is the most appropriate rule to represent the policies. Loosely defined, the authorization rule is a rule of the form:

$$\texttt{cando}(o, s, < sign > a) \leftarrow L_1 \& \ldots \& L_n.$$

where o is an object, s is a subject, a is an action, $< sign >$ is either $+$ or $-$, and each L_i is a literal, where $0 < i \leq n$. The exact, formal defintion is given by the authors in their original papers on ASL [?], [?].

We wrote Perl scripts to convert the original policies to their ASL equivalent forms. The generated policies were then compared to the original policies to identify any similarities and differences. A semantic comparison was used; we ignored syntactic entities like comments and blank lines.

B.1 Systrace

We shall now examine the Systrace policy (Figure 1) and its ASL equivalent form (Figure 3). At first glance, the ASL-equivalent policy seems to resemble the original policy. However, closer examination shows that it was difficult

```
cando(null, native-getuid, +permit) <- .

cando("/etc/hesiod.conf", native-fsread, +permit)
    <- typeof("/etc/hesiod.conf", filename)

cando("/dev/arandom", native-fsread, +permit)
    <- typeof("/dev/arandom", filename)

cando("/var/named", native-chroot, +permit)
    <- typeof("/var/named", filename)

cando("inet-*:53", native-bind, +permit)
    <- typeof("inet-*:53", sockaddr)

cando("inet-[0.0.0.0]:0", native-bind, +permit)
    <- typeof("inet-[0.0.0.0]:0", sockaddr)
```

Fig. 3. The Systrace policy subset from Fig. 1 represented in ASL.

to represent certain features like the conjunction of boolean expressions. Handling similar but different operators like match and eq was also not trivial. This demonstrates yet again that ASL is not able to cater to specific features of system policies.

C. Findings and Observations

The major observation from our implementation exercises is that ASL excels as a theoretical, logical language, but is challenging to use in practice. The main reason is because ASL is flexible, but it is not extensible. ASL can represent the access control policies for many systems; however, it is difficult to support extensibility using ASL, which is the ability to support the specific features of those policies.

First of all, ASL lacks comprehensiveness. While ASL provides excellent support to flexibly represent many access control policies, it does not have a proper facility to specify sets and groups in the first place. Apart from that, while it is possible to assign permissions in ASL, it is impossible to revoke those permissions.

Secondly, there are inconsistency issues. The syntax of some ASL predicate symbols has not been fully discussed in a consistent manner [?], [?].

Lastly, a few design characteristics of ASL make it difficult to use in practice. For example, ASL supports fine-grained temporal access control (though it is not described in detail). Fine-grained temporal access control is not widely used in operating systems, as it can incur a large performance overhead. Also, the syntax of ASL tends to be repetitive (as we can see from Figure 3), which makes it difficult to use in practice.

IV. The Design of Chameleos

Through the discussion in Section III, it is interesting to note the various compromises that authors of flexible languages have to make in order to support various systems. Designing such languages is clearly not a trivial task. In this section, we describe how we think those issues should be addressed, and what we think would be a good way to address them.

A. Objectives

We begin the discussion of our language by describing its two main objectives, as follows:

1. **Support for multiple security-aware systems.** The language must be able to specify access control policies for multiple security-aware systems. Recall that our definition of security-aware systems includes both operating systems and programs that work with operating systems.
2. **Focus on implementation.** We strive to build a language that can be implemented and used in practice, as opposed to a theoretical, logical language.

We name our language Chameleos, which we derive from the chameleon. We chose the name due to the chameleon's ability to blend into the specific features of different environments. The name ends with "os" to demonstrate the language's focus on operating systems.

B. Approach

We now discuss the approach that we take to design the language. We present the two key decisions that we made in the design of the language, and why we made them.

Firstly, we have to decide whether to develop an extension of ASL or develop an entirely new language altogether. Developing an extension of ASL has some benefits, since most of the groundwork has already been done by ASL's authors. ASL is also a suitable candidate for flexible systems. However, as demonstrated earlier in Section III, ASL may not be suitable for operating systems and is not very adaptable to practical systems. With this in mind, we chose to develop a new language, which we design based on lessons learned from ASL. Developing a new language would also allow us to design syntax that is consistent with what system administrators are familiar with.

Secondly, we need to consider whether to use a top-down approach or a bottom-up approach. In other words, we need to decide whether to develop our language by testing it regularly with general concepts, or on actual operating systems. General concepts in this context refer to implementation-independent paradigms, such as access control lists and role-based access control. The top-down approach is suitable for flexibility, and is the approach used by ASL. For Chameleos, however, we believe that developing for actual systems (the bottom-up approach) would be more beneficial in the long run, since Chameleos has to be implemented on real systems in the end.

These decisions lead to an evolutionary design model for Chameleos. By evolution, we mean that the development of Chameleos will go through a number of iterations until the actual language design and syntax is finalized. This also implies that we will use a small number of security-aware systems as target systems initially, and increase the number as development progresses. By using this evolutionary process, we believe that we will be able to support the specific features of each system more effectively.

C. Criteria

We now present the criteria for Chameleos, which we have developed based on the objectives and approach outlined in the previous sections. Each criterion is explained as follows:

1. **Flexibility.** First and foremost, Chameleos should be able to support the access control policies of multiple operating systems.
2. **Extensibility.** Of equal importance is the extensibility of Chameleos. Chameleos must be able to support the specific features of operating system access control policies. A benefit of this is that it can allow a system to use only the features that it requires and nothing more. For example, Systrace works with system calls and does not have anything to do with type enforcement. Chameleos should be able to allow Systrace to work within its environment.
3. **Able to be implemented.** Chameleos must be a practical language as opposed to a theoretical one.
4. **Well-defined syntax.** Chameleos must have a well-defined syntax to promote clarity and reduce the ability to introduce ambiguity.
5. **Comprehensiveness.** Chameleos must provide facilities to enable the support for any target system. This includes the ability to define groups and sets.
6. **Policy specification only.** In order to support flexibility, we separate policy specification from mechanism. This allows us to focus on developing a language that is superior for policy specification.
7. **Text-based language.** By text-based language, we mean that the language will not be confined to a GUI IDE. The language itself should be expressible in ASCII text. This is in contrast to "languages" such as Visual Basic, which has to be edited and compiled via a graphical user interface.

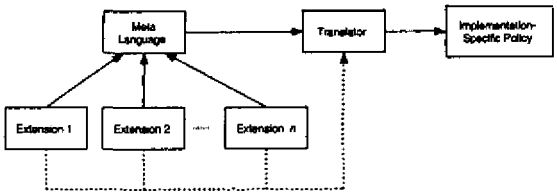

Fig. 4. The Chameleos architecture.

8. **System independence.** We strive to develop a language that can work across multiple systems, with the major requirement that the system must be in the operating systems domain.

D. Design

The three major objectives of the system are high flexibility, extensibility, and practicality. With this in mind, we will now discuss the design of the components that are needed to implement the language. The overall architecture is shown in Figure 4.

The design of ASL has shown that separation of policy specification and mechanism is very important in order to allow flexible representation of policies [?], [?]. We have taken that notion into consideration in the design of Chameleos. The Chameleos architecture primarily involves the translation of a Chameleos policy file into a system-specific policy. If we have two similar systems, the architecture is sufficiently flexible to allow the same Chameleos policy to be translated into specific policies for the two systems.

The Chameleos architecture consists of three main components: the meta language, extensions, and a translator. The meta language is the core Chameleos language itself. The meta language consists of the generic syntax of Chameleos. This includes operators, statement terminators, reserved keywords, variable types, and other related entities. The meta language would also clearly show how functions and procedures should be defined.

To write a Chameleos policy for a specific system, say SELinux, we would need to use the specific features of that system. For example, a Chameleos SELinux policy would need support for specifying type transitions and roles. A Chameleos policy for another system like Systrace would focus on system calls. To support such extensibility, a Chameleos extension can be used. An extension would consist of the necessary variables, library of functions (which can be implemented as a well-defined API), and other state variables of a specific system. For example, a Chameleos SELinux extension would consist of convenience functions to specify users, user-role assignments, type transitions and so forth. We could think of an extension as a module, much like a C `#include` file, or a Java class package that could be imported into a Java program. It is clear that the meta language has to be generic enough to support a wide variety of extensions in order for extensibility to be achieved.

Having extensions in this manner would also allow future systems to be supported easily in the Chameleos architecture. A new extension could be developed for a new system, which could then be integrated into an existing Chameleos system. Another advantage of using extensions this way is that it isolates the components needed for a given system. For instance, if we are working with SELinux, we could just load the SELinux extension, and totally ignore the other irrelevant extensions like Systrace's. This results in less system overhead when actually running the translator, which we shall describe next.

The last component of the Chameleos architecture is the translator. As its name implies, a translator would convert a Chameleos policy into a system-specific policy. For example, a translator would translate a Chameleos Systrace policy into an actual Systrace policy. The translator would need to know the meta language natively, and be able to load extensions into the system when required.

E. Language Features

We now discuss the features of the language that are required to conform to the criteria and objectives discussed earlier. At a minimum, Chameleos should consist of the following features:

1. **Generic subjects and objects.** In order to be flexible, Chameleos must be able to support generic subjects and objects.
2. **Subject types and object types.** Closely related to generic subjects and objects, Chameleos must be able to label the subjects and objects with types. For example, a subject can be of type process, while an object can be of type file.
3. **Variables.** Variables clearly need to be supported for various applications.
4. **Macros.** Macros would allow a convenient way of expressing repetitive patterns in the Chameleos policy.
5. **Arbitrary sets.** Since many systems use the notion of sets in their access control policies, support for arbitrary sets is required. Facilities to perform the regular set operations, such as union and intersection, should also be provided.
6. **Groups.** Groups are closely related to sets. One way to implement a group is to implement it as a set, but allow it to be manipulated from a higher level of abstraction. Depending on the system's specific features, a group can be restricted so that it can never be changed once it is defined. Groups are also generic enough to represent roles, and role hierarchy implementations need to be addressed as well.
7. **Compound expressions.** Compound expressions would allow multiple expressions to be joined together, such as the conjunction of multiple boolean expressions.

8. **Association.** Associations allow us to link one entity to another, such as assigning a user to a role.
9. **Arbitary comparison operators.** A system could use a new comparison operator that other systems do not have. For example, most of the time, the equality operator is present. However, Systrace has a comparison operator `match` that matches regular expressions instead of being strictly an equality operator. This presents the need for arbitrary comparison operators.
10. **Aliases.** Aliases are similar to the `typedef` functionality in C. Aliases allow us to substitute one word for another. If used correctly, this promotes readability and clarity, and it would make it easier to understand the policy writer's intentions.
11. **Mode of operation.** The motivation for implementing mode of operation came from observing the differences between the Systrace and SELinux policies. In a Systrace policy, every single statement is strictly a part of a system call policy. However, in SELinux, different statements specify different things; for example, one statement specifies a type transition rule, while another may specify a user-role assignment. Therefore, we could use the mode of operation feature to define statements which have different functionalities.
12. **Arbitrary permissions.** Permissions vary wildly across different systems. For example, a `getattr` permission may be relevant when dealing with obtaining the attributes of a file, but it is not required when working with roles. Furthermore, one filesystem may support more permissions than another. For instance, the Andrew File System (AFS) allows users to specify the permission to list files in a directory. Such functionality is not present in regular UNIX systems without AFS.
13. **Allow/Deny.** This feature is related to permissions, and allows the administrator to allow or deny a permission.
14. **Hierarchies.** Hierarchies are usually difficult to implement, but are necessary to implement operating systems that support role hierarchies.
15. **Constraints.** Constraints allow us to restrict the number of ways in which a policy statement or a set of policy statements can be specified. This helps us to validate policies. For example, we could use constraints to limit the range of values that can be assigned to a certain variable.
16. **Specification of defaults.** When developing extensions, it is important to be able to specify defaults for each extension. This way, even if certain system-specific variables are not used, they would already have been initialized with reasonable default values.

V. Representing Access Control Policies

As mentioned earlier, we are developing Chameleos using an evolutionary approach. We are still in the process of defining the final language. We now present our progress at representing access control policies using Chameleos. Since we are employing a bottom-up approach, we chose to use specific target systems to develop the language.

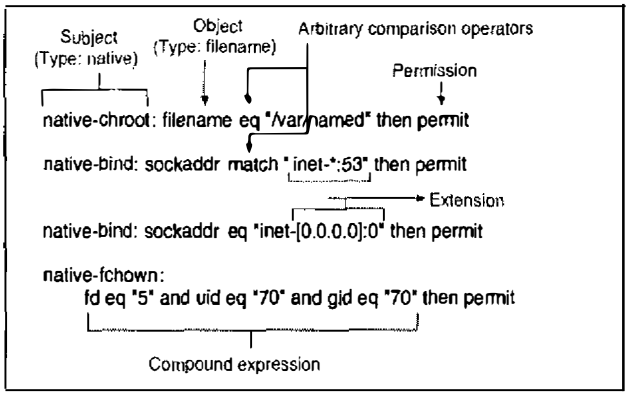

Fig. 5. Excerpt of Systrace policy, shown with Chameleos directives.

The selection of the initial systems is very important, since it influences all future development phases of the language. In this initial development phase, we focus on SELinux and Systrace as our target systems. We believe that these two systems are different enough to help us develop a language that provides flexibility and extensibility.

For instance, SELinux provides a basis for the development of comprehensive access control policies. Being an emerging open source operating system, SELinux is easily available and a lot of research still needs to be done. Apart from that, SELinux is highly suitable to develop Chameleos because a lot of its policies are based on the m4 language. m4 may not be a language that many system administrators are familiar with. An administrator who can develop an SELinux policy in Chameleos can have it translated to an actual SELinux policy without having to learn m4.

We will now discuss each system and show how Chameleos can be used to represent the policies of each system. We also give consideration to the features of the language that we presented in Section IV-E.

A. Systrace

Consider the Systrace policy excerpt in Figure 5. `native-chroot` is designated as a subject. In the context of ASL, we could use a Chameleos extension to specify the subject's type as "native" to use the host system's ABI. Likewise, we denote `filename` as an object of type "filename."

Since Systrace supports various comparison operators, we demonstrate the use of arbitary comparison operators in its policy. In Figure 5, `eq` and `match` are denoted as comparison operators. Regular expressions and other specific syntax, such as `"inet-*:53"`, could be implemented using extensions. Compound expressions, such as the one in the statement on `native-fchown`, must also be supported.

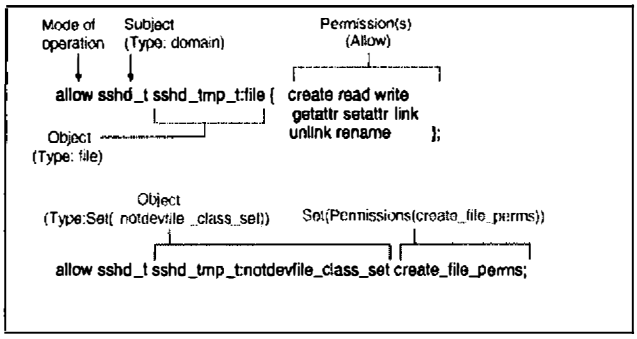

Fig. 6. Excerpt of an SELinux policy, shown with Chameleos directives.

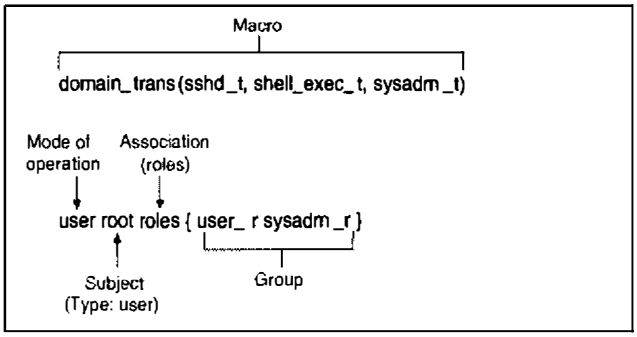

Fig. 7. Excerpt of another SELinux policy, shown with Chameleos directives.

B. Security-Enhanced Linux (SELinux)

SELinux has a very comprehensive access control policy, which also makes it difficult to implement. We introduce the reader to an excerpt of an SELinux policy shown in Figure 6, which shows two equivalent statements.

The first statement allows a subject in the `sshd_t` domain to create, read, write, link, unlink, rename, and set and get the attributes of an object of type `sshd_tmp_t` of object class `file`. Suppose we intend to give the same permissions to multiple files. To use the similar statement for each single file would require a lot of repetitive statements. This may reduce the readability of the policy. Instead of repeating the statements, the SELinux policy allows the administrator to group similar entities into sets. This can be seen in the second statement. A subject in the domain `sshd_t` can now perform any of the permissions given in the `create_file_perms` set to any object of type `sshd_tmp_t` as long as it is in the object class `notdevfile_class_set` set. In other words, sshd can now create, read, modify, delete, and rename any non-device file if it is of type `sshd_tmp_t`.

The second statement provides a more convenient way to express the first statement. However, in the process, it may lose some granularity in terms of its intent. For example, if the `create_file_perms` set includes more than the permissions declared in the first statement, the administrator might allow more permissions than originally intended. Sets and other convenience functions should therefore be used with care. Nevertheless, sets do have many uses if used correctly, and the pros far outweigh the cons.

We can support both statements in Figure 6 with Chameleos. In the first statement, `allow` can be declared as a mode of operation, since SELinux supports many types of operation in a single file. `sshd_t` is declared as a subject of type `domain`, and `sshd_tmp_t` is declared as an object of type `file` to represent the SELinux object class. The multiple permissions can also be specified as Chameleos permissions. Similarly, we can use the Chameleos set facilities to represent the `notdevfile_class_set` and `create_file_perms` sets in the second statement.

We now refer the reader to Figure 7. The first statement in Figure 7 shows an SELinux domain transition macro called `domain_trans` that accepts macros. Chameleos can also be developed to support macros. Macros help to avoid excessive repetition and encourage the development of readable policies. The second statement assigns the `root` user to two roles `user_r` and `sysadm_r`. As mentioned earlier in Section IV-E, Chameleos groups can be used to represent roles. Also, the `roles` keyword in the SELinux policy qualifies as a Chameleos association.

VI. Ongoing Work

At present, we are concurrently developing the design of the Chameleos language and working on an implementation of the Chameleos architecture. We believe that a flexible, extensible, and well-defined language can be produced after a number of evolutionary phases. At the moment, our aim is to provide a full syntax description of Chameleos in Backus-Naur Form (BNF). We are currently developing the language, architecture, and associated tools on a development platform consisting of a Linux/SELinux system and an OpenBSD 3.5 system with Systrace policies.

We determined that we would need three additional components for inclusion into the Chameleos architecture in the future: a syntax checker, an analyzer, and a reverse translator. The *syntax checker* would be used as the foundation for all syntax checking requirements in the other components. The role of the *analyzer* would be to analyze a Chameleos policy to check for conflicts and ambiguity. This is extremely important, since ambiguity in a policy may degrade the security of a system. The analyzer would also have to load Chameleos extensions to fulfill this purpose. The *reverse translator* would be used to translate a system-specific policy into a Chameleos policy. Such functionality would be useful when performing a comparison of system policies, or during system migration.

Two issues that need to be further investigated are safety analysis and safety checks. These concepts are similar to type safety checks on programming languages. Without

safety checks built into the language architecture, enforcing a policy may result in some unintended consequences due to ambiguity in the policy definition. For example, if we define a policy to deny access to a user on a language architecture without safety analysis and checks, the user may be able to gain access by invoking a series of operations that involve an ambiguous part of the policy.

Another important issue to look into is conflict resolution. A single system could produce many conflicts in its security policy [?]. If we are working with multiple systems, it is inevitable that the systems will produce even more conflicts. The question that needs to be asked is whether checking for conflicts should be done while the system is running, or whether the design of the language itself should include mechanisms for preventing or resolving conflicts. Constant checking would ensure that no conflicts occur; however, it incurs a high performance overhead. In line with our criterion for system independence, we suggest that conflict prevention and resolution facilities should be built into the language itself. This issue is one that definitely warrants further exploration.

Another notion to explore is on dependencies among extensions. As we develop extensions for more systems, we will undoubtedly find that many of these systems and extensions are related. For example, a UNIX system, whether it is SELinux or OpenBSD, would have at least three permissions – read, write, and execute – for each file. We could abstract common requirements into another extension, and have other extensions that use those common requirements to reference the former extension. This form of modularity is certainly very desirable. At the same time, however, having a large number of extensions that are dependent on each other would introduce complexity. We believe that the answer would be a compromise between the two extremes of entirely no dependencies and heavy dependencies. In line with our evolutionary, bottom-up approach, we intend to explore this issue by implementing Chameleos on our target systems and examing the feasibility of each approach.

VII. Conclusion

In this paper, we presented the initial development of a new language called Chameleos that is designed to support access control policies of multiple operating systems. Having such a language would be very beneficial to system administrators, since they would no longer need to re-learn the syntax of each and every new access control policy specification language for every system they encounter.

We also discussed ASL as a potential candidate language for this purpose. Despite its flexibility and expressiveness, we found that ASL is difficult to use as a practical language for operating systems. We demonstrated this by implementing ASL to represent Systrace policies. This led us to develop an entirely new language instead of extending ASL.

We presented the objectives, criteria, and initial design of Chameleos along with the roles of two systems – Systrace and SELinux – in the development of Chameleos.

Acknowledgments

The work of Gail Ahn was partially supported at the Laboratory of Information of Integration, Security and Privacy at the University of North Carolina at Charlotte by the grants from National Science Foundation (NSF-IIS-0242393) and Department of Energy Early Career Principal Investigator Award (DE-FG02-03ER25565).

References

[1] NSA, "Security-enhanced linux." http://www.nsa.gov/selinux/.
[2] B. Spengler, "grsecurity." http://www.grsecurity.net/.
[3] N. Provos, "Improving host security with system call policies," in *Proceedings of the 12th USENIX Security Symposium*, (Washington, DC), August 2003.
[4] S. Jajodia, P. Samarati, and V. Subrahmanian, "A logical language for expressing authorizations," in *Proceedings of the IEEE Symposium on Security and Privacy*, (Oakland, CA), pp. 31–42, May 1997.
[5] S. Jajodia, P. Samarati, V. Subrahmanian, and E. Bertino, "A unified framework for enforcing multiple access control policies," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 474–485, May 1997.
[6] G. Karjoth and M. Schunter, "A privacy policy model for enterprises," in *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, (Nova Scotia, Canada), IEEE Computer Society Press, June 2002.
[7] H. F. Wedde and M. Lischka, "Modular authorization," in *Proceedings of the 6th ACM Symposium on Access Control Models and Technologies*, (Chantilly, VA), pp. 97–105, 2001.
[8] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca, "A logical framework for reasoning about access control models," in *Proceedings of the 6th ACM Symposium on Access Control Models and Technologies*, (Chantilly, VA), pp. 41–52, 2001.
[9] T. Woo and S. Lam, "Authorizations in distributed systems: A new approach," *Journal of Computer Science*, vol. 6, no. 2,3, pp. 107–136, 1993.
[10] C. Rippert, "Protection in flexible operating system architectures," *ACM SIGOPS Operating Systems Review*, vol. 37, pp. 8–18, October 2003.
[11] P. Loscocco and S. Smalley, "Integrating flexible support for security policies into the linux operating system," in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX '01)*, June 2001.
[12] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau, "The flask security architecture: System support for diverse security policies," in *Proceedings of the 8th USENIX Security Symposium*, pp. 123–139, August 1999.
[13] W. Boebert and R. Kain, "A practical alternative to hierarchical integrity policies," in *Proceedings of the 8th National Computer Security Conference*, 1985.
[14] T. Jaeger, A. Edwards, and X. Zhang, "Managing access control policies using access control spaces," in *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies*, (Monterey, CA), pp. 3–12, June 2002.