Your Security Policy is *What*??

Matt Bishop
Dept. of Computer Science
University of California at Davis
bishop@cs.ucdavis.edu

Sean Peisert
Dept. of Computer Science & Engineering
University of California at San Diego
peisert@cs.ucsd.edu

Abstract

Systems and infrastructure rarely enforce a site's security policy precisely. Conversely, determining the policy (or policy components) that the systems and infrastructure do enforce is difficult because of the plethora of configuration files and systems at the site. We propose a way to unify these problems by applying a bi-directional method of enforcing and reverse-engineering system and infrastructure policy. The process uses a platform-independent intermediate policy representation (IPR) to bridge the gap between a high-level expression of policy and a machine-dependent, system configuration. The result of these methods, shown along with a detailed example, is that both policy discovery and enforcement can be made into a much more rigorous process.

1 Introduction

The disconnect between a site's stated security policy, and the policy actually enforced by the network and system configurations, is often large. An institution may ban the use of unencrypted connections, yet its servers may be configured to accept such connections. Conversely, an organization may require the use of specific protocols, and internal divisions may ignore the dictates due to fiscal or security constraints.

The gap becomes larger when a site has no stated security policy and decides to create one. The proper way to create such a policy is to ask each administrative subunit to determine what security requirements it has, and develop a policy that takes all these needs into account. The usual aproach is for the central administration to decide what a security policy should contain, and issue it, possibly creating procedures for exceptions in case some subunits need latitude to perform their function. This can create problems, especially if the supplied policy is divorced from the reality of the work that the subunits perform.

The situation for auditing compliance with a policy is worse. Auditors can make spot checks to determine if specific policy requirements are met, but such spot checks are complex. The reason lies in the complexity of modern systems. For example, a policy may require that no incoming `telnet` requests be accepted. The auditor checks that the internal hosts do not accept connections to port 23, and concludes that the policy is met. In reality, many internal hosts have a line in their configuration file to run `telnetd` when the system is rebooted, but the particular set of systems that the auditor checks do not happen to be running `telnetd` at that time. Alternatively, `telnetd` could have even been running on an alternate port. Had the auditor checked the configuration files, she would have noticed the policy violation.

Extracting the *actual* policy from the system and network configuration files enables an analyst to see exactly what policy the system enforces. The analyst can then either alter the intended policy to reflect the actual policy, determine how to reconfigure the systems to implement the intended policy, or do some of both.

Such a process of unifying actual and intended policy falls short of our current abilities. We cannot express a policy in a language that policy makers can read, that also can be rigorously transformed into machine specific and operating system specific commands to implement the policy.

This paper pushes the idea of languages encapsulating policies in both directions. We argue that not only must a higher-level policy language be transformable into an implementation of a policy (or set of policies) on systems, but also that it must be deriveable from the existing configuration of systems. This means that mechanisms to handle contradictions must be defined, as must syntax and semantics to handle procedures external to the computer.

We describe methodologies to enforce a policy or set of policies. This requires that the policy (or set of policies) be checked for consistency, that systems and infrastructure be properly configured, that human-oriented procedures provide the support that technological mechanisms cannot, and that as the policy is updated to reflect changes in environment (such as new or evolving threats and laws or regulations), the changes are reflected in configurations. The element of *policy discovery* allows one to determine the actual policy of a new system being connected to the infrastructure, as well as to validate the enforcement of the existing policy or policies on systems being managed.

Our paper is organized as follows. First we discuss some characteristics of policy languages that are relevant to our thesis. Then we describe our model, and apply it to policy enforcement and discovery. We present a simple example, and conclude with a discussion of difficulties and areas for research and experience.

2 Background

Previous work in policy languages has focused on expressiveness of policy languages and the ability to transform policy languages into wrappers or other mechanisms that enforce policies. The first type typically focuses on whether a policy language can express needed constraints, or which language is more powerful or appropriate for expressing particular constraints [JSS97, CC97, ZS96, SZ97]. The second typically allows a program to translate the policy statement into sets of configuration commands that will enforce the stated policy [BSSW96, HMT$^+$90, PH99, DDLS01]. Our work is closer to the second than the first.

Our work adds a layer of abstraction and the ability to "reverse engineer" the resulting configuration back into the policy language. Then we can apply techniques to analyze the results and determine soundness and completeness of the actual (as opposed to intended) policy. We can then iterate. The actual and intended policies are combined, and checked for soundness and completeness. The analysts and policy makers deal with any resulting contradictions and undesireable aspects of the policies. Then the final policy is applied to the system.

Work on determining policy from existing configurations has focused on firewall rules. Typically, these involve analyzing the firewall rules to create a database that an administrator can query to determine what actions the rules allow [Woo01, MWZ00, BMNW99], or performing static or dynamic analysis of the rules to check for problems [YMS$^+$06, ASH04]. Previous work has also investigated extracting policy from logs [CGL00] and understanding the limits of on-line policy enforcement [Sch00]. Our work deals only with static discovery, and discusses abstracting configurations to an alternate, higher level of expression rather than focusing on handling queries, log entries, or execution monitoring.

3 Language Characteristics

Consider a policy language suitable for expressing policy constraints in a form that can be rigorously analyzed. Once the policy's soundness and completeness have been verified, the policy constraints determine how systems and infrastructure are to be configured. View the policy constraint language as a high-level programming language.

Taking this analogy further, the language must compile into an intermediate form, in which the constraints are expressed in terms of services offered by the systems and infrastructure. While not focused on specific implementations, this level would specify (for example) what accesses file systems, web servers, and other components must provide. Think of this level as a bytecode-like language. We will call this a "macro-assembly language." Then this form is assembled into machine-dependent, specific commands to configure individual systems and infrastructure components.

The advantage of this view to others are twofold. First, it provides a high degree of abstraction; if the infrastructure or details of systems change, the high-level policy constraints need not change; a simple recompilation (or reassembly) will provide the appropriate changes. The services are divorced from the details of their implementation, much as a macro assembly language abstracts the details of bit patterns from the machine language into macros. The second advantage is the explicit tie to reverse engineering policy. In this case, one can determine low-level (macro assembly) policy constraints for services by "disassembling" the configurations of systems, and then, ideally, "decompile" the server-based constraints into the higher level policy language. The result could then, for example, be compared to the (desired) policy constraints, and checked for inconsistencies.

Such a policy language must have several attributes. It must be extensible. For example, if digital rights management is not included in the set of projects studied in our security architecture, it would need to be added. Perhaps specific components of digital rights management require language elements that are not present in the original language. The security management architects must be able to add those elements easily. The language must include procedural, as opposed to technical, elements. How to do this best is unclear (perhaps treat this element as a system, and generate specific rules much as the language mechanisms generate specific configuration rules for specific systems), but what is clear is that humans are central to any effective security system. A policy-based security architecture that does not take the human element into account is fatally flawed. Finally, the language must be able to capture environmental constraints. Security is a product of people, technology, and the environment in which people and technology must function. For example, the health care environment (where availability and confidentiality are paramount) has different security constraints than the financial environment (where integrity of bank accounts is paramount).

A policy constraint language also expresses a distributed computation, because the policy constraints might apply to many systems on a network, and the systems must work together to enforce the overall policy. For this purpose, policy changes that apply to more than one machine must occur simultaneously, for example if two machines control access to a single resource (e.g. on a webfarm). Natural language policies must also translate to compatible configurations across multiple platforms. For example, if a policy requires encrypted remote logins, and one machine is only running ssh1, and another machine is only running ssh2, the policy enforcement mechanism must be able to re-configure one or both systems to be compatible with each other.

4 Overview of Our Approach

Consider how a program expressed in a high-level programming language such as C or Pascal might be translated into a form that a computer can execute. First, a compiler transforms the high-level language into an intermediate form. The high-level language is (usually) machine independent. The statement "$x = 1 + y$" says nothing about how the variables "x" and "y" are represented internally, nor about how addition is managed. But the compiler translates this into an intermediate form that expresses the specific operations that must be performed. One such form is the abstract syntax tree (AST) that shows the variable "y" and the integer "1" under a node containing the operator "+", and that tree as a child of a node with "x" as the other child, and the node contains the "=" operator. This intermediate form is then translated into a machine-specific form, machine language, that the target system can execute.

The intermediate form has two advantages and one disadvantage over the higher-level form. It abstracts the relationship of the components of the program. This allows the program to be analyzed in ways that the original form, and the final (machine language) form, do not allow. For example, the AST can be checked for type conflicts, or analyzed to find dependencies on user input. Secondly, the AST is not tied to a particular system, or indeed to a particular higher-level language. For example, the GNU compiler provides several front-ends for langauges such as C, C++, FORTRAN, and Ada. It uses the same code generation mechanism to generate code for any of these languages for a wide variety of systems, including Motorola 68000 systems, Intel x68 chips, and Sparc and MIPS-based systems. The disadvantage is the unreadability of the intermediate form. For example, a programmer would have difficulty looking at an AST for a large routine and figuring out what it does, or programming in that form. The usual notion of a program is considerably easier to read.

We advocate a similar approach to expressing policies in policy languages. The high-level language should be designed with humans in mind, to ease the task of those writing policies (or translating policies from a natural language to the high-level language). Then a "policy compiler" would transform this expression into an intermediate form, called the "intermediate policy representation" (IPR). The IPR is independent of system and infrastructure architecture, and expresses relationships in terms of services and rights. Then a "policy assembler" translates the IPR into a set of configuration commands appropriate to a specific system.

This also eases the difficulty of reverse engineering a policy from a specific system configuration. One first determines what services the system is configured to offer, and who may access them. This is translated into the IPR. Then the IPR can be "decompiled" into the higher-level policy language. Techniques of reverse engineering applied in programming languages may be useful here (see for example [VO01, CG95]).

5 Applying Policies to Systems and Sites

The first step in enforcing a policy is "compiling" the policy into the intermediate policy representation (IPR). The process of converting a policy to an IPR involves a validation process, as well as building a hierarchical representation of the policy components, from the most general to the most specific.

A key aspect of this conversion is handling conflicts. A hierarchy provides one way to resolve conflicts. Specifically, one can allow the more specific policies to violate the more general policies. For example, "no network access" could be overridden by "except responding to pings." Another means of resolving conflicts is to combine the relevant parts of the policies. For example, there should not be one policy which states that only the superuser is able to execute a program and another policy which says that anyone can execute a program. Those policies would be in conflict. On the other hand, by specifying a *single* policy which states that *both* the superuser and everyone else on the system can execute a program, a policy enforcement mechanism can be more certain that the policy is consistent and correct.

Validation must first be performed for internal consistency and next for consistency with the system to which it is to be applied. The validation for internal consistency is entirely machine-independent. There are many existing techniques for evaluating self-consistency of a policy, which includes making certain that there are no conflicting policies and that a policy is both syntactically and semantically correct. For example, this validation must be able to compare two sides of an assignment statement and be able to validate whether the right hand side is allowed to include a range of possible values, or must be binary (on or off, true or false, etc.).

The machine-dependent evaluation requires more specific domain knowledge. For example, the evaluation must make certain that specified subjects and objects exist on a given system, and that the policy's states and actions, both allowable and prohibited, are appropriate for the subjects and objects to which they apply. Consistency of a policy with a system requires that a policy must be checked to ensure that the sub-policies are applicable to a designated system. Some policies may be less precisely enforceable on some systems than others. For example, access control on traditional UNIX systems uses an abbreviated ACL, which is much less granular when compared with ACLs on modern implementations.

The IPR does not possess domain knowledge, however, and thus must be written to deal with more than one possible system. The IPR of policies like the one involving ACLs should be accompanied by an indication of how to handle situations in which ACLs are not implemented, and cannot be emulated precisely. A number of possible solutions exist. A warning could be generated, a more relaxed policy might be accepted, or an alternate solution could be found, which might include a different means of achieving similar goals on the system, or adding third-party software might compensate for functionality absent in the original operating system.If a more relaxed policy is to be accepted, then the IPR must indicate the *minimum* acceptable policy as well as the *desired* policy.

Some policies may not be enforceable on some systems at all, either because the systems possess no means of ever violating the policy ("write" permissions on a file are irrelevant when the disk that they are on is mounted read-only) or because the systems possess no configuration mechanism with which to enforce the policy (if user home directories, temporary files, and swap space are required to be encrypted, but there is no means of doing so, the policy is unenforceable). Many such situations may be unanticipated. In such cases, there should be default behavior that indicates what the policy enforcement mechanism should do. Regardless of whether the behavior is anticipated or the result of following defaults, one must specify whether the policy enforcement mechanisms should generate a warning and continue, or generate an error and halt.

We can view the translation of policies to system configurations by analogy, in much the same way that we view an ACL. ACLs are themselves simple policy grids which translate into permissions for accessing and manipulating system objects. In general, security policies also should be able to use a grid, tree, or lookup-table to map to a set of implementation mechanisms.

Once the IPR has been mapped into the policy mechanisms on the particular type of system under consideration, the "compiler" can generate the actual configuration commands.

As an example of this scheme, suppose at the highest level, only users with logins are to be able to access systems. The high-level policy language might have a statement like:

```
if (login(user).allowed == NO)
      then system.access = NO;
else
      system.access = YES;
```

At a lower level, the "compiler" may be configured to check web services, and ftp services. The resulting low-level statements might be:

```
ftp:
      -- allow anonymous login if user need not have login to access system
      anonymous = (login(user).allowed == NO);
      ...

web:
      -- require password if user needs login to access system
      authentication = (login(user).allowed == NO);
      ...
```

At the configuration level, these might be translated into the following:

```
echo ''ftp.anonymous = NO'' >> /etc/rc.local
echo ''allow any authenticated'' >> /usr/web/.configure
echo ''disallow all'' >> /usr/web/.configure
```

in which the first line explicitly turns off anonymous ftp, and the second and third force all web accesses to be authenticated, for the particular target system.

6 Reverse Engineering Policies

Policy discovery is the reverse of policy enforcement, and as such, many of the techniques are shared between both. Rather than starting at the policy and working down to the system, policy discovery starts at the system and works up to the policy.

A policy discovery process can reverse engineer policies in one of three ways: (1) it can examine the state of the system (e.g. filesystem, configuration files), (2) it can monitor events (or look at events previously logged), or (3) it can observe a system's response to stimulus. The second method is very much like *vulnerability analysis* or *intrusion detection*. The third is similar to *penetration testing*. Some information can be gleaned in more than one way, and some information might need to be cross-checked for consistency. One of the concerns with combining method #1 with either #2 or #3 is there might be policy conflicts between them that are difficult to resolve.

Complicating matters is that an observed state is not static but rather is *transient*. A transient state, as opposed to a fixed state, is one that changes over time. Unfortunately, by combining techniques, we may not be able to distinguish between static and transient events and states. Therefore, we cannot necessarily know whether we are viewing an actual policyor a deficiency in how a policy is implemented. Because implementation flaws are outside of the scope of this paper, we do not consider discovery mechanisms that tell us about transient policies. Therefore, in the rest of this paper, we will focus on method #1: examining the state of the system that does not change, except by design.

Static policies can be reverse engineered from a finite number of key sources. For a BSD-like operating system, the key sources are:

1. Configuration files (including everything in `/etc` and `/usr/local/etc`). This includes user IDs and group IDs. Note some of these files may be in users' home directories (such as `.rhosts` and `.shosts` files).

2. Ownership, locations, and permissions of files in the filesystem, including “special” files, such as device files; and

3. Build configuration files, such as those for the kernel and other compiled binaries.

The most challenging part of policy discovery is assigning meaning to the data sources. While file ownership, location, and permissions are simple to represent and automatically process, configuration files or compiled binaries are significantly more difficult to process. This is another example of where analogy to source code compilation is useful. Binaries may actually require decompilation to fully understand, although if the build files are present, it is possible to extrapolate the policies built into the binaries from those. For simple system configuration files, it may be sufficient to apply C++-like *templates* to subjects and objects. In doing so, a reverse-engineering process can distinguish subjects from objects and determine what is allowable in a particular context. For more complex system configuration files, a language specifying how to parse other languages might be needed. For example, regular expressions that indicate how to perform token lexing, and a grammar specifying productions (e.g. Backus-Naur Form (BNF)) with which to parse configuration files, might be needed. For example, the following configuration:

```
/etc/ssh/sshd_config:
PermitRootLogin no
```

might be read with this C++-style template:

```
/etc/ssh/sshd_config:
PermitRootLogin <boolean>
```

But this configuration:

```
/etc/master.passwd:
root:$1$SzebsWei$sjWoeMzpTjJkRVbWsDgJrk:0:0::0:0:Charlie &:/root:/bin/csh
```

requires a more intricate regular expression, indicating optional fields and syntax of paths, and other fields in `/etc/master.passwd`. It should also be noted that not all configurations are relevant to security policy. The field in the file above containing “`Charlie &`” provides general information about the user and is irrelevant to the security policy.

As a more complex example, the following configuration potentially requires not only regular expressions to lexical analysis tokens, but a more complicated grammar to understand the hierarchical (and sometimes even recursive) productions. This configuration file controls access to the `root` account whe a program called `sudo` [Mil] is used. That program verifies the user’s identity by checking his or her password, and then granting access to other accounts based on the contents of the configuration file. The file below gives anyone access to all accounts.

```
/etc/sudoers:
ALL    ALL=(ALL) ALL
```

Once this, and other configuration commands, are decompiled into the IPR, the result might contain:

```
login(root).remote = NO;
allowchangeEUID(root,ALLUSERS) = YES;
allowchangeEUID(root,ALLUSERS) = NO;
```

These policies demonstrate the need of the next step: once individual policy components generated by each configuration file have been checked, the components must be analyzed together, in a hierarchy, and checked for consistency. Discovered policies should not conflict. When they do, the resolution of the conflict depends (again) on domain-specific knowledge. For example, within one component, are the rules interpreted by the last match overriding the first, or the first match causing future matches to be ignored? Across components, what does having a non-empty password for the `root` account mean when at the same time the `hosts.equiv` file is set to allow any remote user to log in as `root`? In the former, the actual policy must be discerned based on an analysis of the system components affected by the configuration file. In the latter, the actual policy has two components, one that bars access to `root` and the other that grants it to anyone. Perhaps other aspects of the system (such as a lack of servers that use the `hosts.equiv` file) provide the needed resolution.

individual policy components generated by each configuration file have been checked, the components must be analyzed together, in a hierarchy, and checked for consistency. Discovered policies should not conflict. When they do, the resolution of the conflict depends (again) on domain-specific knowledge. For example, within one component, are the rules interpreted by the last match overriding the first, or the first match causing future matches to be ignored? Across components, what does having a non-empty password for the `root` account mean when at the same time the `sudo` program gives `root` access to all users of the system? In the former, the actual policy must be discerned based on an analysis of the system components affected by the configuration file. In the latter, the actual policy appears to be that remote access to the `root` account is denied, but any local user may have access to that account.

So, the only way to resolve conflicts, without having to generate an error and require human intervention, is by assigning priority of policies using a hierarchy. For example, if "owner" and "world" permissions on a binary permit execution, but the "group" permissions do not, since "group" permissions are contained within "world" permissions in a hierarchy, we report that the policy allows anyone to execute the binary.

7 A Detailed Example

In this section, we will show examples of enforcing and reverse engineering policies. The examples involve enforcing and discovering policies related to a user's ability to alter both their "real" and "effective" user IDs. The examples demonstrate how general the security policies need to be, and how specific the system configurations must be. For example, most systems have the concept of a "superuser" but only UNIX-like operating systems call that user `root`. Therefore, the policy must reflect the more general concept. These examples also demonstrate a portion of the hierarchy of the machine-independent policy.

7.1 Enforcing Protected Superuser Access

Suppose that we wish to impose a policy on a computer system that states that a password is required for a user to perform any superuser functions on a computer system, and that remote access directly to any superuser privileges is prohibited. A high-level policy might be stated as simply as this:

```
system:
// next line prevents remote logins as the superuser directly
login(superuser).remote = NO;
// next line requires that the superuser account have a password
password(superuser).exists =  YES;
// next line prevents any user from being able to execute all programs as superuser
allowchangeEUID(superuser,ANYUSER,ALLCOMMANDS)  = NO;
// next line prevents any user from executing a command to change their effective
// user ID
allowchangeEUID(ANYUSER,ANYUSER,changeEUIDCommand) = NO;
```

Skipping the IPR, on a machine-specific level, the configuration might be translated into these instructions on a UNIX-like system, shown in pseudocode:

```
grep '^root' /etc/master.passwd;
if (<password_field> is empty) {
    open(/etc/master.passwd);
    set root's password = ''xxxxxxxx'';
}
replace or append 'PermitRootLogin no' in /etc/ssh/sshd_config
function checkSudo () {
    /* This function parses the EBNF-formatted sudo grammar and removes any productions
     *  that allow any user to have sudo abilities (to root or any other user) on ALL
     *  commands.  This function also specifically removes any function which permits a
     *  user to run the 'su' command as root, as well.
     */
}
```

The pseudocode instructions translate the policies into consistent, machine-specific configurations.

7.2 Reverse Engineering Superuser Access

To reverse-engineer policies relating to elevating user privileges on a system, we need to identify the relevant commands in the relevant files. First, we determine what network servers are available, because remote access requires the ability to connect to the system. In this case, we assume that machines are all UNIX-like, and that the remote secure shell, `sshd`, is the only server allowing remote logins. We can derive the IPR from the configuration file as follows:

```
if sshd is running then:
      output ''RemoteAccess:'';
          find value of 'PermitRootLogin' in configuration file /usr/ssh/sshd_config;
          output ''login(superuser).remote = (PermitSuperuserLogin.value is yes)'';
```

Next, we consider other configuration files that control access to `root`. Clearly, the password file is one because it contains a representation of `root`'s password. Another one is `sudo` as discussed above. For our purposes, we only consider these two. The appropriate IPR can be generated as follows:

```
output ''login:'';
     find value of root's password in configuration file /etc/master.passwd;
     output ''superuser_has_password = (password_field_length > 0)'';

output ''changeEUID:'';
     /* Parses the EBNF-formatted sudo (or equivalent) grammar and computes
      * productions to determine their higher-level meanings; return YES if any
      * user can become superuser
      */
     output ''can_execute_any_program(superuser) = '', checkSudo(ANYCOMMAND);
```

Then taking this to the high-level language, we derive:

```
login(superuser).remote = NO;
password(superuser).exists =YES;
allowchangeEUID(superuser,ANYUSER,ALLCOMMANDS) = NO;
```

Suppose that the first step discovers that "PermitRootLogin" is set to "no" in `sshd_config`, which results in the policy: `login(superuser).remote = NO`. Then suppose that the script discovers in the `/etc/master.passwd` file that the `root` password field is not empty, meaning `root` has a password. The policy `allowchangeEUID(superuser, ALLUSERS) = NO` results. Finally, suppose that the `checkSudo()` function discovers in the `/etc/sudoers` file that sudo permissions allow any user to issue any command as `root`. The policy `allowchangeEUID(superuser, ALLUSERS, ALLCOMMANDS) = YES` results. Here, we have an inconsistency. Two policies, in the same place in the hierarchy, have conflicting settings. Given that they are in the same place in the hierarchy, we cannot resolve the intent of this configuration and must simply report both policies, the way in which they were derived, and the fact that a conflict exists.

8 Procedural Policies

Not all "good security practices" can be translated into policies. This is particularly true where human procedures are involved, and when the "good security practices" are too general for a language to be able to ultimate make them machine-specific. For example, a principle of "good forensic analysis" is to "consider the entire system, including the user space, in logging." [PBKM05] This kind of principle cannot be easily enforced, because it is too general of a policy to have any way of translating properly into an intermediate language.

On the other hand, the policy could state that state that function calls must be captured [PBKM06]. On UNIX, the policy could translate into a system-level mechanism that looks to see whether the `exec` system call has been modified to force all binaries on the system to be run through a specific dynamic instrumentation tool that records function calls.

Not all policies can be easily enforced by a computer system, even if they can be specified using the policy language. For example, if a policy states that a country requires that all cryptographic keys be registered with the police [Bis05], there is no easy way of enforcing that all cryptographic keys on the system are registered with the police short of augmenting the system to contain capabilities that it did not originally have. That is the system would transmit all of the keys to the police.

The question of how to handle procedural aspects of the policy affects the usefulness of the high-level language. Take our above example of no password for the superuser account. If the high-level language mandates that there be one, it must also mandate that the password not be removed (or provide explicit circumstances under which it may be removed). In this case, the compiler must generate some procedural instructions to support this (implicit) requirement.

One way to do this is to consider the procedural practices to be another system. Then, just as the IPR can be translated to a particular set of configuration commands for a Windows system, and a different set for a Linux system, so could it be translated to a set of instructions (or policy manual references) for humans to follow. So translation of the policy to procedural practices is similar to the generation of configuration information from the IPR.

But reverse engineering procedural practices is more complicated. Ultimately, one needs to put them into a form that can be translated back to the IPR. The development of such a form, and a demonstration of its completeness (so that any relevant procedural practices can be entered), is an area for future work.

9 Conclusions

We have described methods for enforcing security policies by translating them to machine-dependent configurations, via an intermediate, machine-independent, policy representation. We have also described methods for "discovering" policies by reverse engineering them from static system configurations, again via a machine-independent, intermediate representation. This bi-directional language translation helps to unify the difficult, and often-distinct tasks of specifying policies, enforcing them, adding new systems to a network, and verifying policy compliance.

By looking only at static system configurations, we are ignoring vulnerabilities — flaws in the *implementation* of policy — but it is not our goal to prove, or even determine, that a running operating system, let alone a network of systems, is secure. Rather, we have described a method to enforce a given policy properly, and to discover the actual policy that is being enforced. Likewise, our methods help bring *mis-configurations* to light.

Not all policies can be enforced as easily as modifying a configuration file, or modifying permissions of a file. For example we may wish to disallow some users from executing setuid/setgid programs, to prevent them changing their effective UIDs. Without the advantage of detailed ACLs, this may be non-trivial. Additionally, some changes to the system based on policies being enforced may require re-compiling binaries (or the kernel) with different configuration parameters, and even rebooting, for the changes to be reflected in running system. Also, some *actual* configuration files are machine-generated by system programs, based on human-generated, configuration-file counterparts.

This methodology may help people and institutions to determine what their system allows and prohibits. System software developers could use this approach to help users resolve conflicts when configuring their software. This would ensure a more stable system that meets the intent of the policy, rather than one that unintentionally drifts into a new, less desireable, policy, thereby compromising the institution.

10 Acknowledgements

Matt Bishop was supported by awards CCR-0311671 and CCR-0311723 from the National Science Foundation to the University of California, Davis. Sean Peisert was supported by award ANI-0330634, "Integrative Testing of Grid Software and Grid Environments," from the National Science Foundation.

References

[ASH04] Ehab Al-Shaer and Hazem Hamed. Discovery of policy anomalies in distributed firewalls. In *Proceedings of IEEE Infocomm*, March 2004.

[Bis05] Matt Bishop. The Insider Problem Revisited. In *Proceedings of the 2005 New Security Paradigms Workshop (NSPW)*, Lake Arrowhead, CA, October 20–23, 2005.

[BMNW99] Yair Bartal, Alain Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A novel firewall management toolkit. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, May 1999.

[BSSW96] Lee Badger, Daniel F. Sterne, David L. Sherman, and Kenneth M. Walker. A domain and type enforcement UNIX prototype. *Computing Systems*, 9(1), Winter 1996.

[CC97] Laurence Cholvy and Frederic Cuppens. Analyzing consistency of security policies. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 103–112, May 1997.

[CG95] Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Software–Practice and Experience*, 25(7):811–829, July 1995.

[CGL00] Christina Yip Chung, Michael Gertz, and Karl Levitt. Discovery of multi-level security policies. In *Proceedings of IFIP Workshop on Database Security (DBSec)*, pages 173–184, 2000.

[DDLS01] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder policy specification language. In *Proceedings of Policies for Distributed Systems and Networks*, pages 18–38, 2001.

[HMT$^+$90] Allan Heydon, Mark W. Maimone, J.D. Tygar, Jeannette M. Wing, and Amy Moormann Zaremski. Miro: Visual specification of security. *IEEE Transactions on Software Engineering*, 16(10):1185–1197, October 1990.

[JSS97] Sushil Jajodia, Pierangela Samarati, and VS Subramanian. A logical language for expressing authorizations. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 31–42, May 1997.

[Mil] Todd Miller. `sudo` ("superuser do"). `http://www.sudo.ws/`.

[MWZ00] Alain Mayer, Avishai Wool, and Elisha Ziskind. Fang: A firewall analsis engine. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, 2000.

[PBKM05] Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo. Principles-Driven Forensic Analysis. In *Proceedings of the 2005 New Security Paradigms Workshop (NSPW)*, Lake Arrowhead, CA, October 20–23, 2005.

[PBKM06] Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo. Analysis of Computer Intrusions Using Sequences of Function Callls. *Submitted to the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006.

[PH99] Raju Pandey and Brant Hashii. Providing fine-grained access control for Java programs. In *Proceedings of the Thirteenth European Conference on Object Oriented Programming (ECOOP)*, June 1999.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, February 2000.

[SZ97] Richard T. Simon and Mary Ellen Zurko. Adage: An architecture for distributed authorization. Technical report, OSF Research Institute, Cambridge MA, 1997.

[VO01] Gustavo Villavicencio and J. N. Oliveira. Reverse program calculation supported by code slicing. In *Proceedings of the Eighth Working Conference on Reverse Engineering*, pages 35–45, October 2001.

[Woo01] Avishai Wool. Architecting the Lumeta firewall analyzer. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[YMS+06] Lihua Yuan, Jiannina Mai, Zhendong Su, Hao Chen, Chen-Nee Chuah, and Prasant Mohapatra. FIREMAN: A toolkit for FIREwall Modeling and ANalysis. *to appear in the Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[ZS96] Mary Ellen Zurko and Richard T. Simon. User-centered security. In *Proceedings of the 1996 New Security Paradigms Workshop (NSPW)*, pages 27–33, Lake Arrowhead, CA, September 1996.