MCC End-User Management Framework

Secure Systems Laboratory
Stony Brook University

August 31, 2006

Contents

1 Introduction **2**

2 System Architecture **2**
2.1 Control Center 2
2.2 Model Repository 3
2.3 Policy Repository 4
2.4 Support Infrastructure 5

3 Overview of Operation **5**
3.1 Operations in Normal User Mode 6
3.2 Operations in Expert mode 7

4 Security Policy Development **10**
4.1 Integrity Policies 11
4.2 Confidentiality Policies 13
4.3 Classification of Applications 13

5 Security Policy Authoring using MCC **14**
5.1 Setting up new policies from scratch 17
5.2 Policy Parameterization 18

A MCC Software Setup **21**
A.1 Startup Script 21
A.2 Building the MCC components 21
A.3 Build the program model generator (optional) 22
A.4 System Configuration 23
A.4.1 configuration Files 23
A.4.2 Keys and Certificates 24

B Application Classes **25**
B.1 Multimedia and Document Viewers 26
B.2 Archive creation and Utilities 26
B.3 File Organization and Album Creation 27
B.4 Editors 27
B.5 Games, P2P Applications, Web Agents and Screensavers 27
B.6 Document Search, Vulnerability Scanners and Network Sniffers 28

C Policy Repository Operations **29**

1 Introduction

This document describes the key issues and operations involved in the use and management of MCC that a system administrator must be familiar with for its proper deployment in an enterprise. It discusses the various components of MCC and how they interact with each other with general guidelines for the system administrator to make the entire enterprise secure.

The primary purpose of MCC management infrastructure is to enable users to select security policies that should be applied to different applications. The management interface must allow for selection of one of previously defined policies, or development of customized policies for use with a particular application. It should be integrated well with other management tools such as those used in software installation.

There are two basic paradigms for building security tools, namely, the "invisible" security paradigm and the visible security paradigm. The former is required in a scenario where the end-users are typically unaware of the security mechanisms and threats. For example, the Secure Socket Layer and Virtual Private Networks are largely designed to be invisible to most users, and is hence fairly easy to use for naive users. On the other hand, the visible security paradigm is more suited to an operating environment in which the users are well-informed of security issues. In this case, a much richer interface is needed that exposes all relevant security details conducive to the user's understanding of the security risks involved. For example, intrusion detection systems and network firewalls should be administered only by users with an in-depth understanding of security issues.

In practice, neither paradigm works well by itself. What is typically needed is a combination that allows security mechanisms to be largely invisible to naive users, while expert users and security administrators seek configurability and controllability. To address these conflicting requirements, we have developed two management interfaces for MCC. The first one, exposed to naive users, is very simple, and seldom requires users to make security-critical decisions. The second interface, designed for security administrators, exposes much more information to users, and offers a great deal of control. This expert interface also provides the ability to control and configure the manner in which the naive user interface operates.

The rest of this report is organized as follows. Section 2 presents the MCC system architecture in terms of the infrastructure required for the operating environment. An overview of the operations of MCC is provided in Section 3. Section 4 provides a flavor for the development of security policies. Section 5 follows this with an extensive guide to security policy authoring using the MCC tool. A detailed guide to setting up the MCC system is presented in Appendix A. Finally, Appendix B will give our detailed classification for applications.

2 System Architecture

The MCC software is composed of several components that work in conjunction with each other as illustrated by in Figure 1. Two of these components, namely, the policy repository and the model repository may be accessed over the network. While the model repository may be hosted by a third party, the policy repository is expected to be within an organization's Intranet. This section discusses in detail all the components that make up the MCC environment.

2.1 Control Center

The Control Center is the primary component of MCC that coordinates with all other MCC components for the proper working of the system. Specifically, it interacts with the model repository, the policy repository and other components that we collectively refer to as the support infrastructure. (In Figure 1, the support infrastructure consists of the policy verifier, execution monitor and the kernel-resident interceptor.) The Control Center is active during the entire runtime of MCC. This

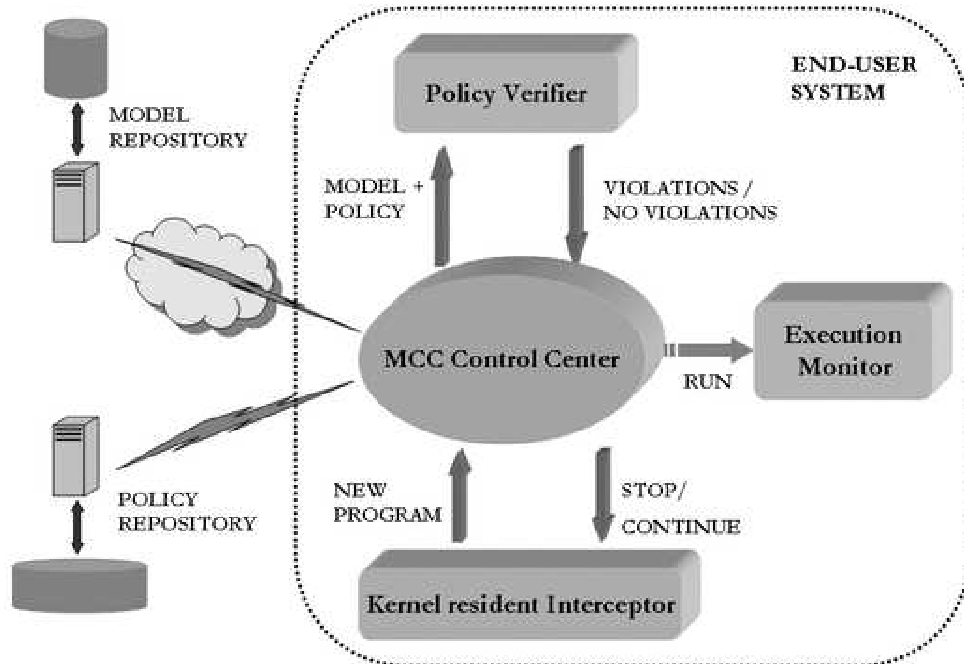

Figure 1: System Architecture

component can work in two distinct modes of operation:

- Normal User Mode: works in background in case of a naive user
- Expert Mode: becomes an interactive GUI-based interface when MCC is run by an expert, e.g., system administrator

In case of a naive user, the policy selection and verification happens without the knowledge of the user, in keeping with our invisible security paradigm for such users. However, when MCC is run by an expert, the Control Center displays untrusted applications in the GUI before they are allowed to run. She can view the policies applicable to each application, verify the model against any of these policies, select an appropriate policy for runtime enforcement and allow the application to run under the control of an execution monitor. The user may also choose to abort the application in some cases, for example, if its model violates all the policies applicable to it. Other times, she may refine the policies to permit the execution of the application. Finally, if she is the system administrator, she can enable the selected policies for other users.

2.2 Model Repository

In this document, we assume that the models may be furnished by either the code producer or a third party. The model repository may be local to the enterprise, or be hosted by a third party. These program models are indexed by a secure hash value obtained from the program's executable, specifically, MD5. The MCC Control Center will download the model and associated information like vendor and version by specifying this hash value, ensuring a secure association between the program and the model. We use an SSL connection to authenticate the model repository server and download the information through an encrypted channel. In addition to the model, the model

repository hold information about the application name, version, and vendor.

2.3 Policy Repository

In our current implementation, we have used a Berkeley database to store security policies. Along with the policies, it also stores metadata information about the policies, like the commonly identifiable name of the policy and the textual description of the policy. The operations on the database are done over a secure connection through a servlet deployed in a lightweight servlet container. Each policy or a set of policies is indexed by the following triple of attributes:

- *Application name*. This attribute specifies the name of an application, including its version number. Clearly, access requirements of an application will depend on its functionality, and hence security policies will be dependent on the application.
- *Vendor name*. The level of trust placed on an application will be determined to a large extent by its vendor, and hence this information is useful in making policy decisions.
- *End-user*. This attribute identifies the end-user that is attempting to access an untrusted application. Some users may be more trusted. or be more knowledgeable about security, as compared to others. Thus, the set of policies allowable for expert users are likely to differ from those permitted for a naive user.

To enable reuse and to simplify policy management, a notion of groups is introduced for all three attributes. Application groups are called as *application classes*. For instance, all versions of a particular application may be grouped into one class. Application classes may be nested, e.g., multiple application classes may be grouped together into a *document viewer* class. Finally, there is a special group called *all* that includes all the applications in the enterprise. It is likely that policies would be specified on the basis of an application or application class. It is less likely that policies would be associated with specific versions of an application.

Vendors are organized into vendor groups. We don't envision a deep hierarchical structure for groups for vendors. Instead, there may be a few groups, each corresponding to a different degree of trustworthiness.

Users are also organized into groups, and this grouping may be based on trustworthiness and/or knowledgeability of users.

The groups are organized into a hierarchy. The root of the hierarchy is the group called *all*. In the case of users, this group includes all users; for applications, it includes all applications; and for vendors, it includes every vendor. The leaves of the hierarchy are the specific users, application versions (or applications), and individual vendors.

The policy repository takes four attributes as inputs: application, application class, vendor and username. It returns all the matching policies. Matching is performed first using application, application class and vendor attributes, and then based on the user attribute. The first stage matching proceeds as follows:

- If there is an exact match for the application name, then vendor information is ignored, and we proceed to the next stage. (The assumption here is that the application name already implies a specific vendor, so there is no reason to look up on the vendor name.)
- If there is no match based on application name, then a match is attempted on the basis of the application class, and then vendor information
- If there is no match for the application class, then policies acceptable for the vendor specified in the query are used. These are policies that have been associated with the user group *all*.

The policies returned by the first stage are refined, based on matching with user information.

If there is a match for multiple groups, then the most specific groups are used. In terms of the group hierarchy mentioned above, this means that a match would be returned for a group only if none of its subgroups (i.e., nodes that are descendants of this group) match the query. Note that this definition still permits multiple policies to be returned, e.g., if an application belongs to multiple application classes. For each of the matched policies, the policy repository returns the corresponding application class, vendor, and user (or user group) information. The detailed contents of the policies themselves are not relevant to the repository – it is up to the consumers of this data to define the details of what goes into a policy. (See Section 5 for details.)

Our interpretation for multiple matching policies is that any of the returned policies may be chosen and applied by the user, so it is the responsibility of the system administrator to ensure that every policy returned by the policy repository provides adequate security for the enterprise. However, in terms of implementation, the naive user can be given the option of choosing which policy should be enforced or the first policy in the list that can be successfully verified is used. For the expert user, the policy list is explicitly shown, and the user asked to make a selection.

2.4 Support Infrastructure

The support infrastructure consists of the kernel interceptor, policy verifier and execution monitor, each of which is described below:

- Kernel Interceptor Module: This module is a kernel-resident loadable module that intercepts all `execve` calls made in the system. It examines the argument to this system call to determine if an untrusted executable is being executed, and if so, notifies the Control Center. On receiving a response from the Control Center, the system call is either aborted or allowed to proceed.
- Policy Verifier: This component is used to check if a model satisfies a policy. The Control Center invokes the Verifier after retrieving policies and applications from appropriate repositories. The response from the Verifier indicates if the verification was successful or not. In the latter case, a compact description of the violations is provided by the Verifier to the Control Center.
- Execution Monitor: If the Control Center permits execution of an untrusted application, then the Execution Monitor comes into picture. It is responsible for enforcing policies selected by the Control Center. The Execution Monitor consists of two components: a detection engine (for detecting policy violations) that is generated from the policies, and a runtime environment for intercepting system calls and delivering them to the the detection engine. More information about the policy language can be found in [1]. A description of the runtime environment can be found in [2].

  Note that policy violations are possible even for verified policies because the corresponding models can in general be incomplete, or because untrusted code is exhibiting potentially malicious behavior that isn't captured by the model. To deal with policy violations, two kinds of violations are identified: *soft* violations and *hard* violations. Soft violations are indicative of unusual but not unacceptable behavior, and the user decides (through the use of a dialog box) whether to allow such behaviors. Hard violations indicate unacceptable behavior, and code execution is terminated immediately by the Control Center.

3 Overview of Operation

The Control Center is activated when the MCC kernel interceptor module (see section 2.4) detects execution of untrusted code. It then contacts the model repository to download the corresponding model and vendor information.

Untrusted code[1] may arrive at an end-user's computer in two ways. First, it may be implicitly downloaded and executed, e.g., by clicking a hyperlink. Second, it may be explicitly downloaded and installed. As far as MCC is concerned, the distinction between the two cases is not very significant, so we don't worry about this distinction from here on.

Note that in some cases, one may simply execute a trusted application with untrusted input, e.g., view a file with a PDF viewer. Since untrusted input can potentially subvert a trusted application, it is prudent to treat such applications themselves as untrusted, so that the end-user can be protected against damages due to a subverted application. This approach is particularly appropriate in light of prominent vulnerabilities that have been recently been discovered on multiple platforms on document and multimedia display software.

Next, we explain the operational steps with respect to the two modes of operation: *normal user* and *expert*. At this point it is assumed that the MCC software is setup and appropriately configured for the enterprise. Please refer to Appendix A for a detailed setup guide.

3.1 Operations in Normal User Mode

- **Trapping the application**. The first thing that MCC does when a new application is run is to check whether it is trusted or untrusted. The Kernel resident Interceptor intercepts every `execve` system call and performs this check. If the application that is about to be executed is an untrusted application then it contacts the Control Center and provides the application name. Otherwise, application execution is allowed to continue, as it would in the absence of the kernel interceptor.
- **Obtaining the program model**. The Control Center will compute the MD5 hash of the program to be executed and formulates a request containing this hash value and sends it to the model repository. As the program models are indexed by the hash value of the program in the repository, the correct model is downloaded. Along with the model, the Control Center will also receive an application name, version and vendor information for the application. As the model repository can be located on third party network, and the communication is through the Internet, `https` protocol (http over SSL) is used to ensure secure transmission of the model.
- **Obtaining the relevant security policies**. The policy repository is queried by the Control Center by providing the application name, application class, vendor and the identity of the user that invoked the untrusted application. The policy repository returns one or more policies that match the values of these attributes along with the metadata information for those policies, like the user identifiable policy names and textual description of policies.
- **Policy Verification**. If a single policy is returned by the policy repository, it is sent to the policy verification module for verification, together with the program model. If multiple policies are returned by the policy repository, then verification is attempted on each one of them, and the first policy in the list that verifies successfully is used. If none of them can be verified, then a policy violation error message is produced and application execution is aborted.
- **Running the program with policy enforcement**. If the policy verification fails, the Control Center displays a message to that effect and notifies the kernel interceptor to disallow the execution of the program. See Figure 2. On the otherhand, a successful verification will result in the execution of program.

  During execution, the selected policy is enforced on the program, thereby ensuring that the policy won't be violated even in cases where the model is incomplete, or the untrusted application turns out to be malicious. As mentioned earlier, two kinds of policy violations are possible

[1]Throughout this document, the term "untrusted code" is used to refer to code that is not fully trusted, and hence needs to be executed under the control of MCC.

Figure 2: MCC Control Center displaying Policy Verification failure in Normal User Mode

at runtime. Hard violations cause the program to be terminated, and an error message to be displayed to the user, and possibly logged on a central server. In the case of soft violations, the user is expected to make a decision as to whether the access should be permitted or not. A policy-specified prompt is displayed to the user, and if the user answers "yes" then the process is allowed to continue. Otherwise it is terminated. Figures 3 and 4 illustrate the two cases.

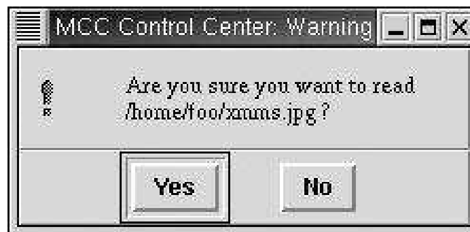

Figure 3: Example of a soft violation dialog-box for a text editor Gedit

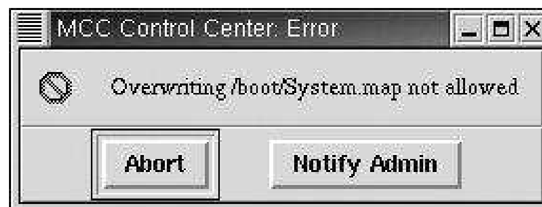

Figure 4: Example of a hard violation error message

3.2 Operations in Expert mode

In the expert mode, MCC provides a powerful wizard based interface that offers a significant degree of flexibility for the user in terms of policy selection. For instance, an expert user, if he is given appropriate privileges by the MCC system administrator, can create a new policy for an application, rather than simply using one of the policies returned by the policy repository. The process of setting up security policies for programs is described in section 5. This section talks about policy selection and verification in expert mode.

The initial few steps for running a program in MCC's expert mode is similar to that of normal user mode.

- Application execution is trapped by the Kernel Interceptor module
- The Control Center downloads the program model and related information for the program from the model repository

- The Control Center queries the policy repository for the security policy corresponding to the program. In return it receives one or more policies, along with the metadata for the policies.

The remaining steps differ from the normal user mode and are explained below:

- **Policy Verification**. A wizard guides the user through a step-by-step process of selection of policy, changing policy rules and parameters, enforcing the desired policy and resuming the monitored execution of program. A new wizard is launched for each instance of execution of the untrusted application. This process has three steps and is described below.

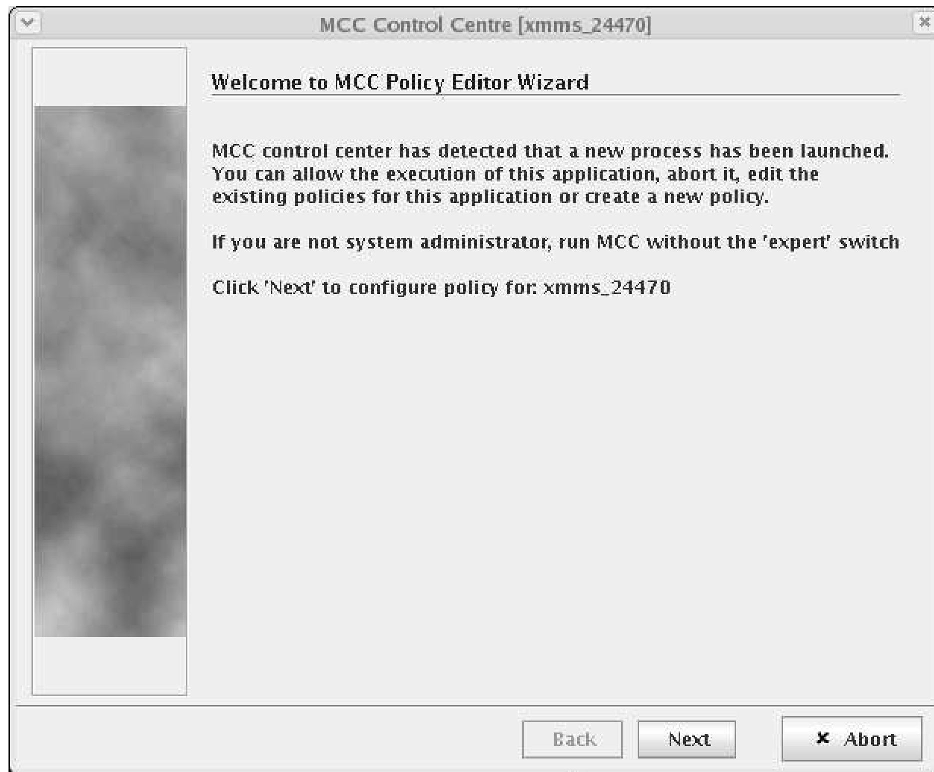

Figure 5: The Information Panel of MCC Control Center wizard for XMMS

1. **Information Panel**. This panel is displayed when an untrusted application is launched. It tells the user an untrusted application was executed and will be monitored by MCC. Only the name of the monitored application is shown. This screen does not provide options to select or modify application policies. See Figure 5.

2. **Policy Selection Panel**. It displays all the policies returned to the Control Centre. At this point, the user can select the desired policy that he wants to enforce. It provides for deletion of policies for that application that the expert user thinks might no longer be necessary.

   Each policy returned by the policy repository consists of several components described below. Each of these components is mapped to a corresponding user interface control in Figure 6. This mapping enables the user interface design to be decoupled from the specifics of the policies that are supported.

   (a) **Name of the Policy**. The commonly identifiable name of the policy. Usually same as that of the application.

   (b) Application name, class, vendor and related information.

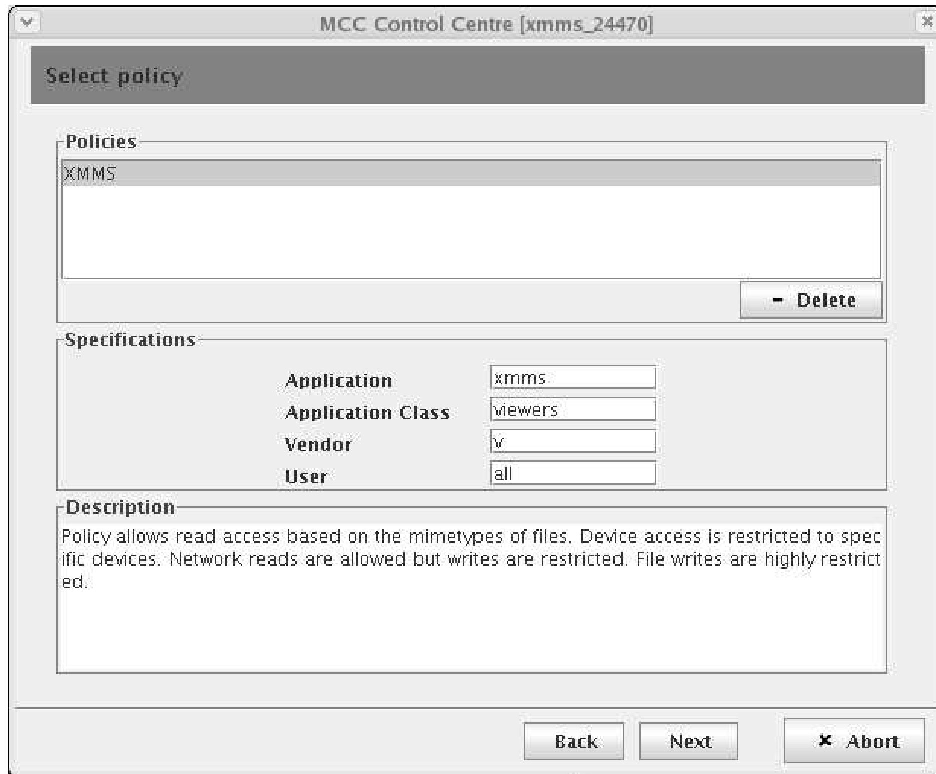

Figure 6: The Policy Selection Panel of MCC Control Center wizard for XMMS

(c) **Description**. High-level textual description for the policy

The list on top in Figure 6 shows the list of policies retrived for a particular application. Here `xmms` happens to have a specific security policy, named XMMS, configured for it. If some application has multiple policies, policy name for each is displayed here. The user can select any policy he wants to be enforced. Different policies can have same name. The "Specifications" section can be used to distinguish between such policies and select the appropriate one. As policies are selected, the Control Center maps each piece of information corresponding to that policy to an appropriate component in the pane.(Section 2.3 outlined the set of information that is returned for each policy by the policy repository).

3. **Rules Panel**. This pane provides controls to modify the rules and/or parameters for the selected policy. It is illustrated in Figure 7. The components of rules panel are described below.

(a) **Policy details**. This component specifies the actual policy, written in MSPL language described in [1].

(b) **Rules**. High level names of the rules comprising the policy (refer to Appendix B for a full description of the rules). A checkmark next to the rule indicates whether that rule is enabled or disabled. Only those rules with a checkmark are included in verification, or enforced at runtime.

(c) **Rule Parameters**. Each rule is associated with a set of parameters. These parameters and their values can be viewed by clicking on the "?" button next to the rule. Section 5.2 explains the concept of parameters in detail.

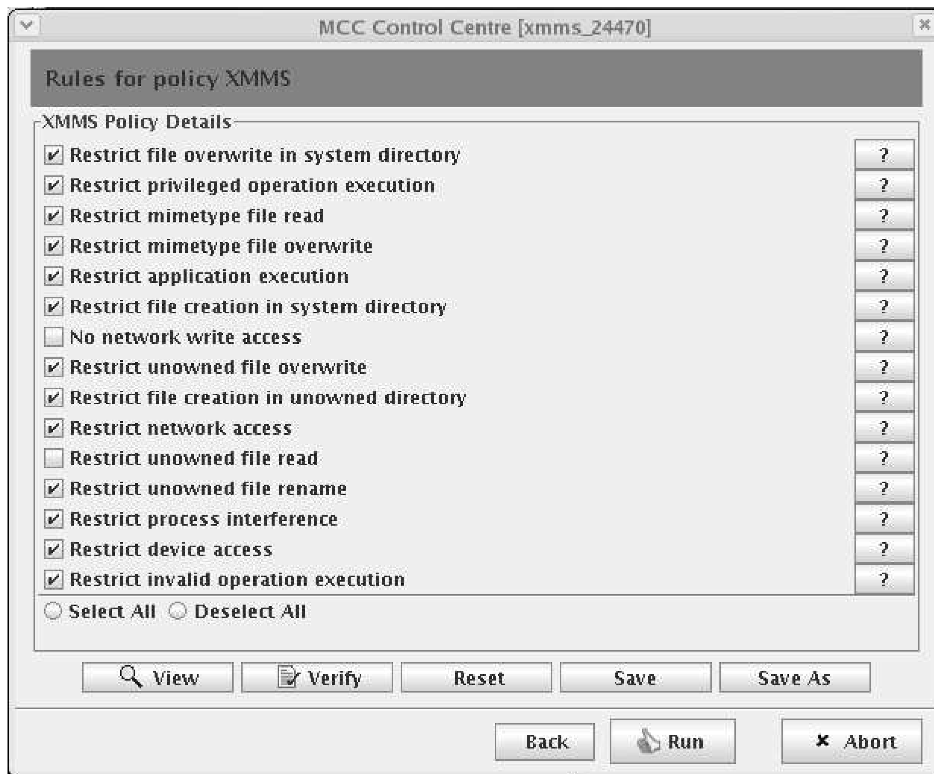

Figure 7: The Rules Panel of the MCC Control Center wizard for XMMS

In addition, the modified rules can be overwritten onto the existing policy or a new policy with a desired policy name can be created.

After making any changes to the rules and/or their parameters, the user can click on the "Verify" button to verify the modified policy against the application's model. If the verification completes without any errors, a message is displayed that conveys the same. If there are violations, an error message is shown, with the violating rules highlighted in red. (See Figure 8.) The user can either abort the program at this time by clicking on the "Abort" button, or allow execution to proceed any way by clicking the "Run" button. More likely, the user will review the verification violations and proceed to refine the rules. More information on interpreting the verification violations and fixing them are dealt in Section 5.

After a successful verification process, the administrator can proceed to run the program by clicking the "Run" button. Only those rules with a checkmark are enforced.

4 Security Policy Development

A policy is a set of rules that govern the execution of an application, ensuring that it does not hamper the security of the enterprise. MCC has different policies for different types of applications based on their functionality and access needs. The system administrator is responsible for maintaining these policies.

The high level objective of MCC policies is to protect the integrity and confidentiality of the system on which MCC is deployed. MCC policies are typically focussed on integrity for two reasons. First is a technical reason – confidentiality policies cannot in general be enforced by simply monitoring security-critical operations made by a program; they require an examination

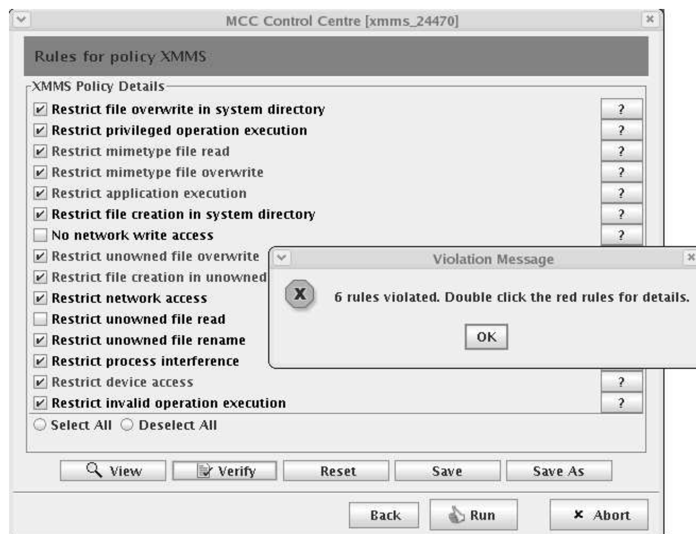

Figure 8: Message indicating verification violations for XMMS in Expert mode

of the internal logic of the program as well. Specifically, only coarse-granularity confidentiality policies can be enforced, e.g., not permitting an application to communicate outside an enterprise, as opposed to finer-granularity policies that allow such communication but regulate the source and content of this information. Second, integrity is typically much more important that confidentiality in most business settings where untrusted applications are used.

Another point to be noted about MCC policies is that they can bound the risk of damage due to a misbehaving application, but cannot provide any assurances about its correct operation. As such, it would be inappropriate to rely *solely* on MCC for those applications whose *correct* operation is essential for security. We identify such applications as *security-critical* applications, and rely on other mechanisms (such as signed code from trusted vendors) to provide an adequate level of security. These mechanisms may be used in conjunction with MCC to provide additional assurances, especially regarding privacy and confidentiality.

4.1 Integrity Policies

Based on the notion of security-critical applications, we can formulate the following policy to ensure system integrity:

Permissive policy: unless the vendor of a piece of software is trusted sufficiently, it should not be permitted to interfere with the operation of security critical applications. The system administrator at this point can select any one of the policies (if more than one are present).

- it should not be permitted to modify the executables, libraries, or configuration files used by security-critical applications or the OS kernel itself
- it should not be permitted to interfere with security-critical processes by using mechanisms such as signals, or by accessing or modifying resources used by these processes
- if a security-critical application is believed to contain vulnerabilities, and/or the untrusted code is suspected to be malicious, then any input files accessed by this security-critical application should not be allowed to be modified by the untrusted application

In a sense, the above statement puts a ceiling on the maximum privilege that can be provided to untrusted code. The difficulty in using this policy is that we need an accurate list of *security-critical*

applications. If some application is accidentally left out of this list, then a malicious untrusted application can cause damage to the system in spite of MCC protection. This difficulty can be avoided by the following policy:

Conservative policy: If an untrusted application only creates new files or modifies files that are never accessed by any application other than itself, then this application can be run securely, without interfering with the operation of any other application (trusted or otherwise) on the system.

Note that the conservative policy is equivalent to the permissive policy when the list of security-critical applications includes every application installed on the system. One can use a policy that is in between, e.g., using the permissive policy with a safe upper bound on the list of security-critical applications.

In order to actually enforce these policies, we need to classify all the files in the system into two main categories with respect to each application:

- Owned Files: The files that are created by an application during its installation are said to be *owned by* the application. The application requires these files in order to run and provide the functionality it is meant for. The configuration files created by a user to customize a particular application are, however, not considered to be owned by the application. (But a "preferences" file that is created and modified by the application is considered to be owned by the application.)

  An application can read, rename, delete, or modify owned files, or create new files in owned directories without being expected to cause any harm to the overall system integrity, and would likely be allowed by typical policies used with MCC.

  When MCC starts, it loads a preconfigured database of all owned files for all untrusted applications. This database is frequently looked up in order to enforce policies as they tend to rely on ownership information. Whenever a new application is included for running under the control of MCC, the database has to be updated with the names of the files owned by that application. This is typically done from the information obtained during software installation – in our implementation, we use the RPM (RedHat Package Manager) database that is a part of all RedHat Linux instances to get this information.

- Security critical Files: The files that are crucial for the security of the system are called Security critical files. This category will typically include all the files in system directories such as `/bin`, `/boot`, and `/usr/bin`, any files owned by the packages corresponding to these applications, their configuration files, and so on. A list of directories containing security-critical applications is provided by the system administrator, and the list of security-critical files is automatically derived by MCC from this information. Moreover, untrusted applications are generally prevented from reading configuration files owned by these security critical applications if they subsequently access anything outside the enterprise network. This restriction can be relaxed by a system administrator as appropriate.

Please note that this classification is not an exhaustive one but seems to be a good starting point for building a practical system. The list of owned and security critical files is constructed as follows:

- from software installation data, as described earlier. The input required is the set of security-critical applications, which may be specified using regular-expression patterns.
- by monitoring creation of files by MCC applications. This will enable identification of files that are created by untrusted code, and hence can be considered to be "owned" by the untrusted code.

- by relying on OS-specific conventions (e.g., directory names and files beginning with the "." character in user home directories are used in UNIX to store user-specific configuration data) and knowledge about security critical files and directories.

4.2 Confidentiality Policies

Confidentiality policies can be developed by considering the flow of potentially confidential information. In general, we want to restrict confidential information as follows:

- prevent flow of confidential information into public files
- prevent flow of sensitive information from files into the network

Information flow can be prevented by restricting untrusted application from:

- reading sensitive files, or,
- writing to the network or public files

In order to concretely define policies that protect confidentiality, we need to identify the list of files whose confidentiality needs to be protected. At one extreme, we could consider every file on a host to be confidential. This can have the unfortunate side effect of producing security policies that will disallow execution of most applications. To obtain useful security policies, one may have to take the step of identifying commonly accessed files that may not contain highly sensitive data, e.g., the name of a host. More generally, we envision that OS-specific conventions can be used to identify files that contain information that is easily obtained or predicted. We can consider every file other than these to be confidential.

4.3 Classification of Applications

Based on our discussion of high level policies in the previous section, four basic categories of applications can be identified:

- Restrictive integrity policy, Restricted reads, Unlimited network access and creation/modification of files that don't compromise integrity (R/RR/UNW)
- Permissive integrity policy, Restricted reads, Unlimited network access and creation/modification of files that don't compromise integrity (P/RR/UNW)
- Restrictive integrity policy, Unrestricted reads, Restricted network access and creation/modification of files (R/UR/RNW)
- Permissive integrity policy, Unrestricted reads, Restricted network access and creation/modification of files (P/UR/RNW)

Based on our study of applications in Linux, we have identified the following major classes of applications in terms of their functionality. Along with each application class, we specify which of the above categories that the application is in.

1. ***Multimedia and Document viewers (R/UR/RNW)***. Includes viewers for common document types such as PDF, PostScript, and Word; image viewers like `xview`; and media players like `xmms`.
2. ***Archive Creation and related utilities (P/UR/RNW)***. Applications in this category include `atool`, `bzip2`, `rar` and so on. Also included are format conversion applications such as `antiword`, `ascii2pdf` and `html2latex`.
3. ***File Organization (P/UR/RNW)***. Example applications in this category include photo album generators (`album`, `picturepages`, etc.), tools for renaming and reorganizing files (e.g., `rta` and `mmv` file utlities).

4. ***Editors (P/UR/RNW)***. Includes image and photo editors like `gimp`, text editors like `emacs`, web page editors, and so on.
5. ***Games, P2P Applications, Web agents and Screensavers (R/RR/UNW)***. Includes applications like web crawlers, Instant messengers, games, web browsers, news readers.
6. ***Document Search, Scanners and Network Sniffers (R/UR/RNW)***. Example applications include `tcpdump` (network sniffer), `snort` (Intrusion detection system), `webstone, SSLperf` (performance monitor), `http-analyze, nettracker` (log file analyzers).

MCC comes with a default policy for each of these application classes. We defer the detailed discussion of these policies to Appendix B.

5 Security Policy Authoring using MCC

The policy language used in MCC, called MSPL (MCC Security Policy Language), is described in detail in [1]. Any one who wants to develop a new security policy from scratch needs to be familiar with this language, and develop the policies in this language. In addition, they need to structure their policy specifications in such a way that that it is compatible with the policy authoring wizard, so that the policies could be further customized. This is described further in Section 5.1. Here, we focus on the easier task of customizing policies that have already been specified in MSPL. This customization of previously defined policies is achieved using the following mechanisms:

- *Enable/disable rules within a policy.* An MSPL policy consists of a collection of rules, each of which regulate a particular aspect of application behavior, e.g., files created by the application or overwritten by it. The wizard interface allows an expert to enable or disable rules within a policy.
- *Modify policy parameters.* Each MSPL policy may use parameters. The wizard allows users to examine these parameter values and modify them as needed.

The Rules Panel allows an expert to enable or disable rules within a policy. It also allows users to examine specific parameters for each rule and modify them as needed.

These modified policies may be saved to the policy repositories and/or associated with specific applications. A system administrator (called "sysadmin" from now) will have total control over the policy repository, and can set policies that are allowable for other users. An expert user without system administration privileges will be able to set policies for himself, provided this is permitted by the sysadmin. We explain the process of policy authoring by taking an example where a new program `xifrac`, a 3D tetris like game for Linux, is configured for use as an untrusted application.

1. The sysadmin starts MCC Control Center (in expert mode) and then launches `xifrac`. As explained earlier, the program model is downloaded from the model repository, and the policy from the policy repository.
2. The sysadmin clicks on "Next" button after verifying that application name showed by MCC is correct. Since xifrac is unknown to the policy repository, it returns a default set consisting of six policies, each corresponding to an application class described in the previous section. See Figure 9.
3. The sysadmin realizes that the best match for xifrac would be a policy for "Games and IM." She selects this policy for xifrac. She changes the specification (Application, Application Class, Vendor, User) to (xifrac, games, all, all) and gives an appropriate description to help identify the behavior of the new policy. She then clicks on "Next" to move on to the rules panel.
4. The rules panel presents the user with an array of policy rules and a button to modify the parameters associated with each, if any. In order to determine whether the selected policy is

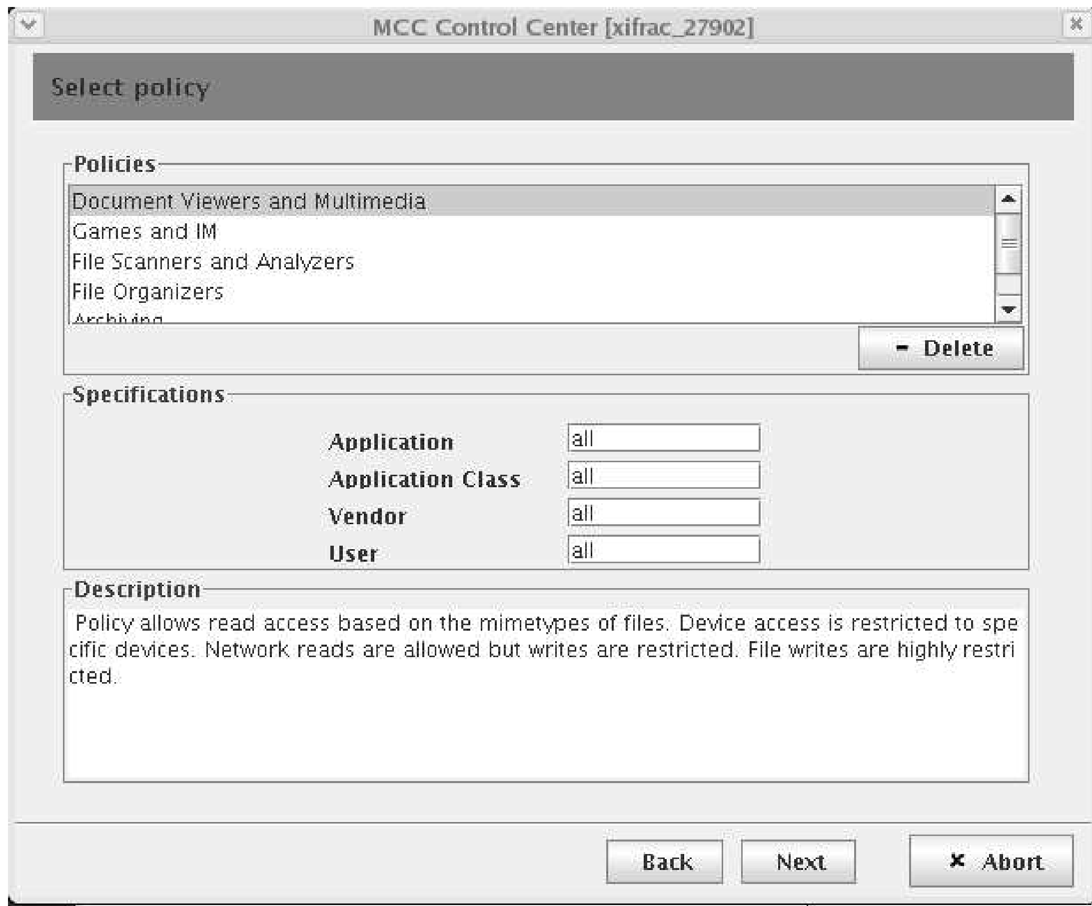

Figure 9: The Policy Selection Panel of MCC Control Center Wizard for xifrac

satisfied by xifrac's program model, the user performs the verification process for the default selection of rules by clicking on the "Verify" button.

5. The verification process generates one violation as illustrated by Figure 10.
6. The user double-clicks on the rule marked in Red in order to learn about the verification violation. She notices that the violation for the rule named *Restrict unowned file overwrite* is for the parameter WRITABLEFILE. This means that the verification violation has occured due to the parameter WRITABLEFILE being too restrictive. In general, a rule can be violated by one or more parameters.
7. She clicks on the "View" button in the Violations dialog box in order to view all those values of parameter WRITABLEFILE that caused the violation. This will bring up a dialog box that shows all the violating values. See Figure 11. It indicates that xifrac will write a file *ifrac.scores* to the user's home directory and this is not allowed by the "Games and IM" security policy. Hence the violation. Now the sysadmin has two alternate courses of action:
   - If the violating values do not come across as a threat to the saftey of the system, they can be added to the WRITABLEFILE parameter so that in future, there are no violations on this parameter. This also means that when xifrac is allowed to run, it will be allowed to write to `/home/foo/ifrac.scores`. Since this is a reasonable option, she will click on "Add" to add the selected value to the WRITABLEFILE parameter. The Violating values dialog box also

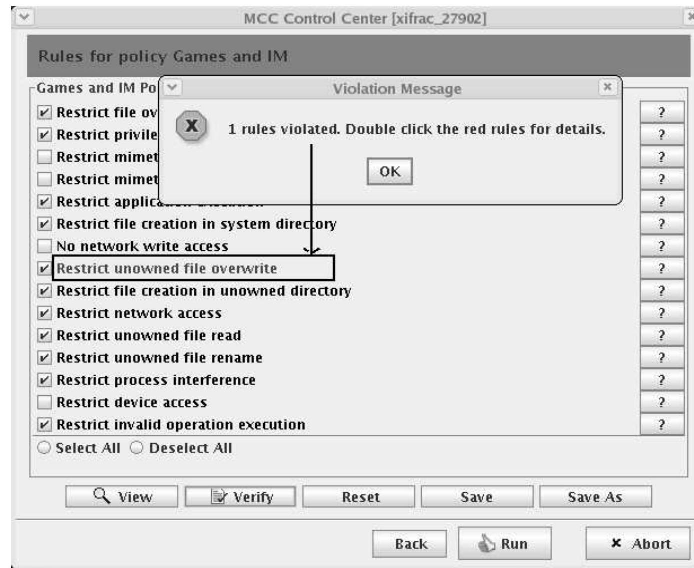

Figure 10: The dialog notifying the verification violations for xifrac

has an optional textbox for entering an aggregate value which is an expression terminating in '*' (i.e., a restricted form of regular expression). For instance, if the list consists of many filenames from a directory `/home/foo/xifrac/`, an aggregate value `/home/foo/xifrac/*` can be added to the parameter instead of adding all of the listed values. This option is just for convenience in managing the parameter.

In case that the violating parameter is a directory path, the values are displayed as a file system tree. In this case, any node along the path can be added. This has the same effect as adding a prefix of the violating parameter followed by a wild card.

- If the violating values indicate some malicious behavior or just some action that is not acceptable, it is a warning to the sysadmin that the program could potentially do the same action if it is run. This highlights one of the main benefits of program verification. It can infer the actions of a program from its model and check whether it complies with a given policy as long as the model stays correct. At this point, the sysadmin can dismiss the dialogs and click on "Abort" to abort the program in question, and perhaps uninstall it.

If the sysadmin decides to take the former action, she will proceed to inspect the verification violations in other rules as well and possibly update other violating parameters as well.

8. Finally, when she is satisfied with the program behavior and decides not to abort it, the sysadmin commits all the changes made to the policy into the policy repository by clicking on the "Save As" button and specifying "XIFRAC" as the policy name. This will create a new policy in the repository that is indexed by the specification values (xifrac, games, all, all) selected by her. Future invocation of the same program "xifrac" will result in this new policy being selected for verification and enforcement.

   If the new policy is to be overwritten over the existing policy, the user should click on "Save" button. The previous policy will be overwritten be the new policy possibly with new name, description, specification triplet and rules.

9. When a program that has been configured in MCC is about to be uninstalled, the policies created for it can be removed by starting the program and then clicking on the "Delete"

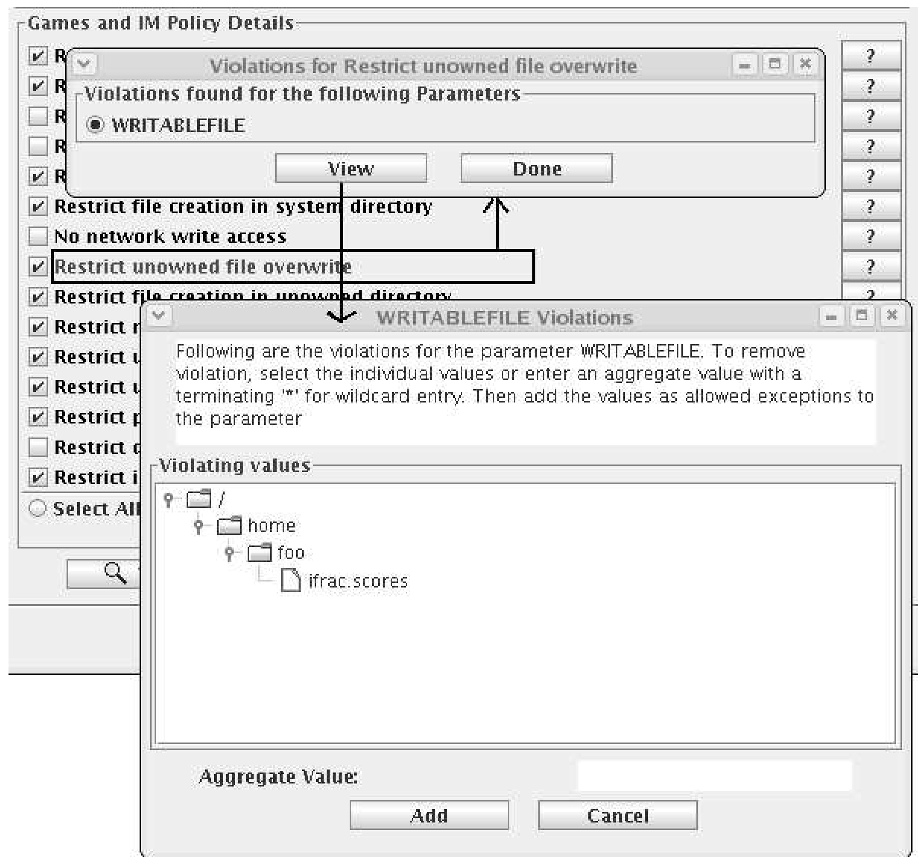

Figure 11: Viewing the parameters and their values that caused verification violations

button in the policy selection panel of the wizard. This will remove the customized policy from the policy repository and then terminate the program.

5.1 Setting up new policies from scratch

An important requirement in the design of policy-authoring tool is to make it independent of any specific policies, or knowledge specific to certain policies. Instead, the tool needs to be able to handle arbitrary policies, while continuing to provide a user-friendly interface. Below, we describe how we accomplish this.

When a new policy is being developed from scratch, it is to be specified in a particular format so that the policy can be added into the policy repository and then fine tuned using the policy-authoring tool. This is done by writing down all the basic set of information in text files and loading them into the repository. See Appendix C for the complete syntax and the steps for updating the repository.

The first step is to indicate the unique identifiers for the policy: The Application Name, Application Class, Vendor Name and the User Name. One or more of the identifiers can contain "all" if the information is unknown or is not relevant. When the policy will be put to use, these identifiers are used for searching the policy in the repository. This is followed by the user identifiable policy name, a high-level textual description of the policy so that it may be easily understood by system administrators that wish to customize or refine this policy.

Secondly, the policy must be divided into a collection of rule subsets, each of which indicates a

security property that should be satisfied. A rule subset contains one or more rules that are coded in the MSPL language. Indicated along with a rule subset are a simple name, a short, logical, higher level textual description and possibly one or more parameters to provide customization as required by the logic of the ruleset. A rule can either make use of already defined parameters or declare new parameters. Newly declared parameters must be defined later. For each rule subset described as above, the policy-authoring tool lists it with the simple name along with the textual description provided for that rule subset.

Third, all the parameters newly declared by the rules should be defined. The properties that are to be specified for a parameter are its name, scope, type, a brief textual description of the parameter and a list of domain values (optional). The parameters are stored separately from the rules themselves so that they may be easily configured separately. More details on parameterization are presented in the next Section.

Finally, when all the aspects of the policy are defined, they can be "imported" into the policy repository by running the loader program against the text files containing the above specification. Once loaded, the policy can now be viewed through the policy-authoring tool by just starting the application for which the policy was configured. The tool allows complete customization of the policy as outlined in Section 5.

A key component of customizable policies is that of parameterization: decomposing the policies into a set of parameters. This parameterization largely controls how verification results are presented in an intuitive fashion to an expert user, and how much customization is possible. Policy parameterization is further described below.

5.2 Policy Parameterization

Each policy's behavior depends on a set of parameters. The parameters vary from a list of files that can be read/written by an application without raising a security violation to the list of external programs that it can execute. These parameters also act as exceptions to the general rules that the policy contains. For example, a policy for an editor may contain a rule that it is prohibited from reading files not owned by it. However, due to its design, it may have to read one file that it doesn't own. Such a case is best handled by combining the ownership rule with an exception for this file; the alternative of disabling the ownership rule and permitting all files to be read by the application would be too drastic. Since these exceptions may vary with deployment, it is best to specify the exception using a parameter whose value can be modified using the policy-authoring tool provided by the rules panel of MCC wizard.

Again, in order to make the policy-authoring tool operate with newly created policies, the parameterization has to be done in a certain way. Specifically, each parameter must be associated with a description that can be displayed. The type of its value needs to be specified. In addition, we need to specify if the input is free-form or is supposed to take a value for a specified list of values. We illustrate parameter specification using the default set of policies that we use for all applications. These policies make use of the following parameters.

- READABLEFILE: Files allowed for reading.
- READABLEDIR: Directories in which files can be read.
- SECURITYLEVEL: The level of security desired with OFF indicating no security and HIGH indicating maximum security.
- WRITABLEFILE: Files that can be created, renamed and modified.
- WRITABLEDIR: Directories in which files can be modified.
- ALLOWEDMIMETYPE: The mime types of files that can be created, read and modified.
- ACCESSIBLEDEV: Devices allowed for opening, reading and writing.

- EXECABLEAPP: Applications allowed for execution.

Note that these parameter values are in addition to the default values that a policy gets through the rules that it comprises of. For instance, READABLEFILE might not have any value assigned to it but that does not mean that the policy prohibits the application from reading any file. The files that the application is allowed to read due to the rules that it contains are still readable by it. So, if the policy contains the rule that allows it to read owned files, it can read all the files owned by it. Also note that not all parameters are applicable to each policy. For example, the EXECABLEAPP parameter will not be present for a policy of an application if the policy does not specify any kind of restriction on the application with regards to executing other applications. Finally, these parameters are shared across the rules in a policy. If multiple rules in the policy make use of the same parameter, the same parameter values will be shared by them as well. This is reasonable since the meaning of parameter remains the same in different rules.

In the wizard, the parameters and its values for a rule can be viewed by clicking on the "?" button next to the rule name in the rules panel. Figure 12 illustrates this.

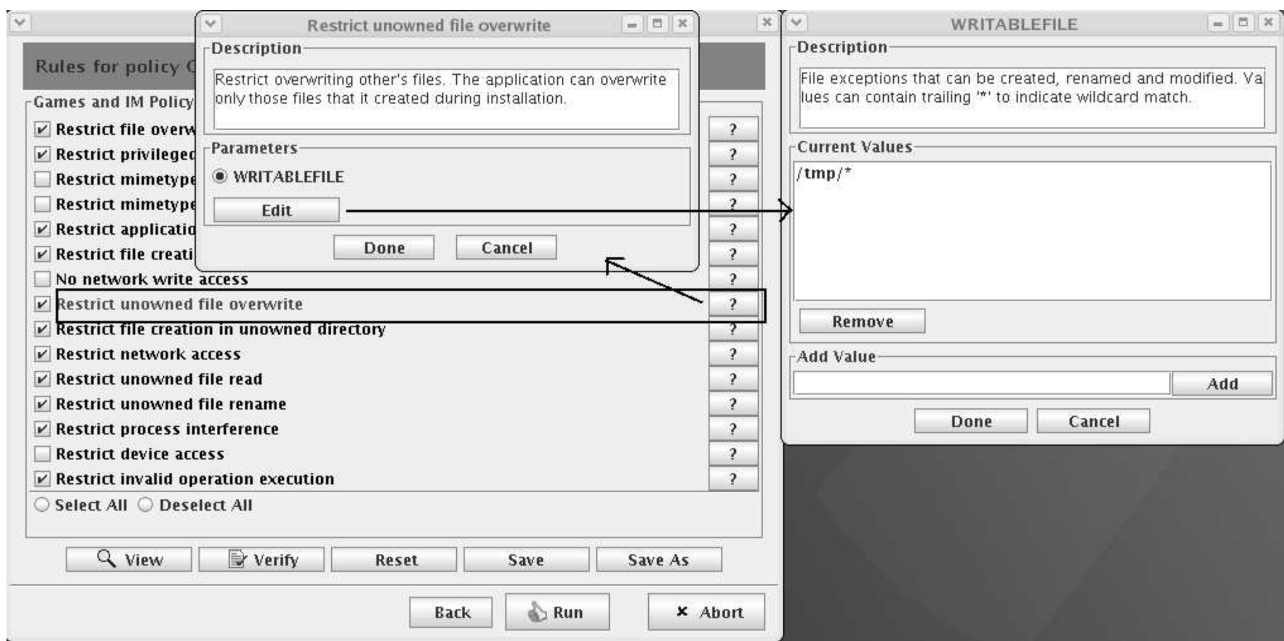

Figure 12: Viewing the parameters and their values for a rule

The Policy Repository also stores the details of each parameter in addition to the actual policies. This makes the entire MCC independent of the parameters. Therefore, new parameters can be added, old parameters can be removed and existing parameters can be modified without affecting the execution of MCC. Even the GUI adapts itself depending on the type of the parameter. Each parameter has the following attributes:

- **Name**: The name of the parameter. For example, ACCESSIBLEDEV.
- **Scope**: The scope of a parameter can be private or public. Private parameters are not exposed to the end-user and are used solely for parameterizing the rules. They determine what action should be taken if a particular rule is violated and are constant for a particular policy. Private parameters must not be changed. Only public parameters can be viewed and edited using the Control Center's policy authoring scheme.

- **Description**: Every parameter has a comprehensive description that is displayed to the user when she tries to edit it.
- **Domain**: If a parameter can take values only from a particular domain, then the domain must be specified for it. For example, SECURITYLEVEL must take any one value of four possible values: OFF, LOW, MEDIUM, or HIGH. This domain must be stored with the Policy Repository.
- **Type**: The type of the parameter is used to determine the type of GUI the tool must present for a particular parameter. The different values that the type attribute may have are:
  - SINGLE: The parameter can only have one unique value which is not restricted to any domain. There are no such parameters as of now but this type has been included for extensibility. Therefore, if a new parameter is introduced in the future which can have only a single value with no restrictions as to what that value may be, it can be incorporated without any changes in the software.
  - MULTIPLE: The parameter is multivalued and it can assume any values. For example, the WRITABLEFILE parameter can be a list of a number of files whose names are not restricted by anything but the file naming rules on that operating system.
  - SINGLE_FROM_DOMAIN: The parameter can have exactly one value chosen from a fixed domain. For instance, the parameter SECURITYLEVEL can only have one unique value that can only be one of NONE, LOW, MEDIUM, or HIGH.
  - MULTIPLE_FROM_DOMAIN: Parameters with this type can any multiple values that can be chosen from a fixed domain. Parameters such as ALLOWEDMIMETYPE can have multiple values but since there are only a fixed number of standard mime types, this parameter cannot have any arbitrary value. Hence, the values are selected from a particular domain comprising of the standard mime types.

  The tool's GUI manifests itself in a different strucute depending upon the type of the parameter. For example, if a parameter can assume only one value from a fixed domain, then the GUI has a selectable drop down list from which only one value can be selected. This allows MCC to have a dynamic GUI that assumes an interface depending upon the kind of the parameter that is being edited by the user.

The values of the parameters for a particular policy are stored in the policy repository as part of the policy. Since the parameter attributes are independent of the policy, they are stored separately to avoid duplication and redundancy in the repository.

A MCC Software Setup

This section describes the steps for for installing and setting up the MCC software. The process can be broadly classified into the following parts:

- Building the MCC components
- Build the program model generator (optional)
- System Configuration

Below we describe each of these parts in detail.

A.1 Startup Script

The script to control the startup/shutdown/setup of various components of MCC is present in the `scripts` directory and is called ***mcc-ctrl.sh***. This has to be copied to the user's `$HOME` directory in order to operate the MCC software. The usage of the script is as follows:

```
./mcc-ctrl.sh <component> <action> [expert] [chooser]
component is:
  ms => modelserver
  ps => policyserver
  cc => controlcenter
  ki => kernel interceptor
  all => all the above components

action is:
  start
  stop
  setup
  cleanup

- Expert switch applies to control center and starts the Control
  Center in Expert Mode.
- Chooser switch applies to control center. It allows user to choose the
  Application Class in the Naive user mode. It has no effect in Expert mode.
```

The "chooser" command line switch allows the user to select the class of the application when MCC is running in Naive user mode. With this option the MCC Control Center uses the user specified application class as an additional hint for selecting one or more policies from a set of matching policies were retrived from the repository.

If the components are started individually, it is recommended to start them in the order as listed above. Shutting them down has to be done in reverse order. The start order is : ms, ps, cc, ki and the stop order is: ki, cc, ps, ms.

A.2 Building the MCC components

The MCC codebase consists of source code for all the components that make up the system. The top-level directory for all the components is called "mcc" and all references to files below are with respect to this directory. The components have to be built and installed at appropriate locations for the system to work. This task is automated by the mcc-ctrl.sh script and can be done as follows:

```
mcc-ctrl.sh all setup
```

If for any reason, the source has to be re-built, the binaries can be cleaned before doing so by doing:

```
mcc-ctrl.sh all cleanup
```

The setup operation performs the following actions:

- builds and installs the Kernel Interceptor module into the system modules directory. The install requires superuser permission which means that the user should be capable of performing a "sudo" operation.
- builds and installs the Policy Compiler. The compiler takes the policy written in MSPL as the input and generates prolog and C code corresponding to the policy. The prolog code is used for policy verification along with the model (which is also in prolog form). The C code is used for building the execution monitor.
- builds the MCC Control Center wizard java application.
- builds the Policy Servlet which retrives and returns policies from a Berkeley DB repository.
- populates the Policy Repository. The default set of policies for all the application classes and a few chosen sample applications are present in the directory `bdb` in a database-friendly format. The default database state is initialized by running the script `populateBDB.sh` in that directory.
- builds and installs the system call tracer. This is the part of the execution monitor that performs the tracing of all the system calls made by an application running under MCC. The tracer is installed into `/usr/local/bin` and the tracer library into `/usr/local/lib`, which requires superuser permission (sudo capability).
- builds the Policy Verifier.

Note: Sometimes, `/usr/local/lib` is not present in the system's library search path, which can lead to an error loading the tracer library. If this issue is encountered, the path `/usr/local/lib` should be added to the LD_LIBRARY_PATH system variable.

A.3 Build the program model generator (optional)

Code producers typically distribute the program model along with the application. They will benefit from the instructions provided in this section. The system call tracer can be configured to dump the runtime trace of a program. This is utilized to record the actions of a program during a training phase in a controlled and safe environment.

- **Learner**. This is a component that learns the relationships between various variables and function arguments in the runtime trace of a program. These relationships are then referred to by the model generator. To build this component, change to directory `learner` and issue a `make` command.
- **Model generator**. This component takes the program traces generated by the system call tracer as the input and generates either prolog or XML based program model. It is helped in this task by the learner and the tracer. Hence there is a direct dependency on tracer and learner. The code for model generator is present in the directory `DRIVER`. An XML-to-prolog model converter is also included as a cgi program to be included with the HTTP server so that models in XML form deployed in the model server can be automatically converted into the prolog form before being used by the MCC Control Center. Follow these steps to build the components:

```
make
sudo make install (installs the cgi program into the HTTP server's
cgi-bin directory)
```

A.4 System Configuration

Before MCC can be run successfully, the system administrator must configure various parameters for the system.

A.4.1 configuration Files

The `config` directory in the MCC codebase consists of a number of configuration files. An overview of these files follows:

- **ownership.lst**. As discussed in Section 4.1, there is a notion of owned files for every application which indicates all the files created by that application and over which the application has complete control. The ownership.lst is the file that contains the list of owned files for every application running under MCC. The format of the file is consecutive sequence of lines of the form:

  ```
  /path/to/an/owned/file
  /path/to/the/application/owning/the/file
  ```

  At runtime, MCC Execution monitor loads the information contained in this file and constructs an in-memory data structure called Trie to efficiently operate on the data. Finally when it is shutting down, the (possibly updated) Trie is serialized back to the file.
- **seccritical.lst**. This file contains a list of all the directories and files that are considered security critical. The list is utilized by the MCC Policy verifier and Execution monitor for making policy related decisions. Note that this list is global in nature, not per-application. The concept of security critical files was discussed in Section 4.1.
- **trusted.lst**. This file is consulted by the Kernel Interceptor module whenever it traps an `execve` system call. If the *execed* program is listed in this file or present in one of the listed directories, it is considered to be trusted and hence will be allowed to run without notifying the MCC Control Center. All other programs are trapped and made to execute through MCC.
- **trap.lst**. Any program that has to be executed under the control of MCC software can be listed here explicitly so that the Kernel Interceptor can straightaway notify the Control Center.
- **scmapping.txt**. This is a private file used by the Control Center for translating from the system call number to name
- **mccConfig**. This is perhaps the most important file that includes various startup parameters for the various MCC components. The placeholders in this file have to be replaced with appropriate values and the file should be copied to the `$HOME` directory of the user as ***.mccConfig*** so that different users can have different settings. The parameters present in this file are as follows:
  - modelServer: The URL of the model server. The server may be within the local network or it may be hosted by a third party.
  - policyServer: The URL for the Policy Server. Like the model server, the policy server may be either within the enterprise or outside it.
  - installationDir: The directory where the MCC software is installed.
  - tracerHome: The directory containing the system call tracer. This is usually <installationDir>`/tracer` and does not require to be modified by the user.
  - learnerHome: The directory containing the learner component. This is usually <installationDir>`/learner` and does not require to be modified by the user.

- policyBaseDir: The directory where all the policies that are downloaded from the policy server are compiled and the execution monitor built.
- ownershipDB: The full path of the ownership.lst file mentioned above.
- secCriticalDB: The full path of the seccritical.lst file mentioned above.
- adminEmail: Email ID of the system administrator. This is required if an email notification is to be sent whenever a program is aborted at runtime due to security policy violation.
- trustedList: The full path of the trusted.lst file mentioned above.
- trapList: The full path of the trap.lst file mentioned above.

A.4.2 Keys and Certificates

The system must also be configured to provide a keystore to be used by Control Center that contains the public keys or certificates of the model and the policy servers. The keystore may also be password protected. The retrieval of program models and policies from the repositories is made secure by the use of HTTP over SSL. Without the correct certificates, it will not be possible to make a secure connection with any of the servers. The `config` directory contains a sample keystore, certificates and private keys for model and policy servers and work well for "localhost" hostnames. These must be replaced by real certificates and keys in the actual environment.

B Application Classes

The policies are constructed from a given set of rules by selecting those that are meaningful to the particular application class. The rules are listed below:

1. **Restrict file creation in unowned directory**. This rule disallows the program from creating files and directories in a directory that is not owned by itself. This rule can exist in two forms:
   - Any attempt at violating the rule will result in a hard violation and the program should be aborted
   - An attempt at violating the rule will result in prompting the user for creation. This corresponds to a soft violation.
2. **Restrict unowned file overwrite**. This rule disallows the program from writing files and directories in a directory that is not owned by itself. This rule can exist in two forms:
   - Any attempt at violating the rule will result in a hard violation and the program should be aborted
   - An attempt at violating the rule will result in prompting the user for writing. This corresponds to a soft violation.
3. **Restrict unowned file rename**. This rule disallows the program from renaming files and directories in a directory that is not owned by itself. This rule can exist in two forms:
   - Any attempt at violating the rule will result in a hard violation and the program should be aborted
   - An attempt at violating the rule will result in prompting the user for renaming. This corresponds to a soft violation.
4. **Restrict network access**. The rule disallows a program from writing to the network after reading a security sensitive file. Any such attempt will be considered as hard violation.
5. **No network write access**. The rule disallows a program from writing to the network, unconditionally. Any such attempt will be considered as a hard violation.
6. **Restrict file creation in system directory**. File and directory creation in security sensitive directories is not allowed and is considered as a hard violation.
7. **Restrict file overwrite in system directory**. Overwriting existing files in security sensitive directories is not allowed and is considered as a hard violation.

   This rule, together with the rules "Restrict file creation in system directory" and "Restrict unowned file rename" ensures that an untrusted application cannot get write access to *any* files owned by security critical programs.
8. **Restrict device access**. The rule disallows the program from opening, reading or writing to any device. Any such attempt is considered as a hard violation.
9. **Restrict application execution**. The application is not allowed to execute other programs. Any such attempt is considered as a hard violation.
10. **Restrict privileged operation execution**. This rule disallows the program from executing certain operations that are considered privileged. For instance, mount, umount, etc. Any violation is regarded as a hard violation.
11. **Restrict invalid operation execution**. This rule disallows the program from executing certain operations that are considered invalid since they are not implemented in the Linux. For instance, ftime, profil, break, etc. Any violation is regarded as a hard violation.

12. **Restrict mimetype file overwrite**. The application can only write files of specific mimetypes corresponding to it. For example, Gedit can only write files with textual content, like text/plain, text/html, text/x-c, text/x-c++, etc. This rule can exist in two forms:
    - Any attempt at violating the rule will result in a hard violation and the program should be aborted
    - An attempt at violating the rule will result in prompting the user for writing. This corresponds to a soft violation.
13. **Restrict mimetype file read**. The application can only read files of specific mimetypes corresponding to it. For example, Gedit can only read files with textual content, like text/plain, text/html, text/x-c, text/x-c++, etc. This rule can exist in two forms:
    - Any attempt at violating the rule will result in a hard violation and the program should be aborted
    - An attempt at violating the rule will result in prompting the user for reading. This corresponds to a soft violation.
14. **Restrict unowned file read**. This rule disallows the program from reading the files that are present in directories not owned by itself. Any attempt at violating the rule is a hard violation.

We now define the policies for the six application classes as a combination of these generic rules. If a particular rule is not included in a policy, it means that the restriction corresponding to that rule will not be applied. We also indicate whether a rule can tolerate soft violation by mentioning *allowed with prompt* alongside the rule name. In the absence of any such mention, violation of rules are considered hard violation and the program will be terminated.

B.1 Multimedia and Document Viewers

- Restrict file creation in unowned directory - allowed with prompt
- Restrict unowned file overwrite
- Restrict unowned file rename
- Restrict network access
- Restrict file creation in system directory
- Restrict file overwrite in system directory
- Restrict device access
- Restrict application execution
- Restrict privileged operation execution
- Restrict invalid operation execution
- Restrict mimetype file overwrite
- Restrict mimetype file read

B.2 Archive creation and Utilities

- Restrict unowned file overwrite - allowed with prompt
- Restrict unowned file rename
- Restrict network access
- Restrict file creation in system directory
- Restrict file overwrite in system directory
- Restrict device access

- Restrict application execution
- Restrict privileged operation execution
- Restrict process interference
- Restrict invalid operation execution
- Restrict mimetype file overwrite

B.3 File Organization and Album Creation

- Restrict unowned file overwrite - allowed with prompt
- Restrict network access
- Restrict file creation in system directory
- Restrict file overwrite in system directory
- Restrict device access
- Restrict application execution
- Restrict privileged operation execution
- Restrict process interference
- Restrict invalid operation execution

B.4 Editors

- Restrict file creation in unowned directory - allowed with prompt
- Restrict unowned file overwrite - allowed with prompt
- Restrict unowned file rename
- Restrict network access
- Restrict file creation in system directory
- Restrict file overwrite in system directory
- Restrict device access
- Restrict application execution
- Restrict privileged operation execution
- Restrict process interference
- Restrict invalid operation execution
- Restrict mimetype file overwrite
- Restrict mimetype file read

B.5 Games, P2P Applications, Web Agents and Screensavers

- Restrict file creation in unowned directory - allowed with prompt
- Restrict unowned file overwrite
- Restrict unowned file rename
- Restrict file creation in system directory
- Restrict file overwrite in system directory
- Restrict network access
- Restrict privileged operation execution
- Restrict invalid operation execution
- Restrict unowned file read

B.6 Document Search, Vulnerability Scanners and Network Sniffers

- Restrict file creation in unowned directory - allowed with prompt
- Restrict unowned file overwrite - allowed with prompt
- Restrict unowned file rename - allowed with prompt
- Restrict file creation in system directory
- Restrict file overwrite in system directory
- No network write access
- Restrict privileged operation execution
- Restrict invalid operation execution

C Policy Repository Operations

Typically, all interactions with policy repository are handled by the MCC Control Center and the Policy-Authoring tool. The expert users and sysadmins can modify existing policies, create new policies from existing generic policies by using the Policy-Authoring tool. However, there is one scenario where the policy repository needs manual updation.

When Security policies are developed from scratch by hand coding the rules, the rule subsets, all the associated parameters and the relevant metadata have to be written down in a simple format in a set of files and imported into the policy repository. The relevant text files are located in the directory `bdb` inside the top-level MCC directory. They already contain the textual forms of all the default policies that come with MCC. Any new content has to be appended to these files and reloaded into the database. The existing policies codified in these text files also serve as good examples for the user adding new policies from scratch. Below, we describe each of the component that has to be updated for adding a new Security policy.

- **RuleDesc.txt**: This file contains all the rulesets and the following information about each ruleset:
  - **RuleName**: A simple name of the ruleset.

    Example: "RuleName:Restrict mimetype file overwrite"
  - **Params**: A comma-separated list of parameters for this ruleset.

    Example: "Params:WRITABLEFILE,WRITABLEDIR"
  - **EventMap**: A ruleset can contain multiple rules in terms of many abstract events. The EventMap associates each of the parameters in "Params" to one or more abstract events. This association helps the policy verification process in the following manner. When there is verification violation for a particular rule on a particular abstract event, the EventMap relates the violation to the appropriate parameter. This helps the policy-authoring tool to aid the user in resolving violations by updating specific parameters.For example, consider a rule that contains a "write" event and makes use of the WRITABLEFILE parameter. When there is a violation on the "write" event, one way of resolving is to update to WRITABLE-FILE parameter to allow the specific filename. In order to achieve this, we create an association between the write event and the WRITABLEFILE parameter.

    Every event of the rule that requires a mapping must have a RuleEventMap. One event could map to multiple parameters if a violation could be removed by updating *any* of those parameters. In this case, the parameters are comma separated.

    Example: "EventMap:write=WRITABLEDIR,WRITABLEFILE"
  - **Desc**: A short, logical, higher level textual description of the ruleset.

    Example: "Desc:Restrict modification of files of particular mime type(s)".
  - **Body**: This is the actual ruleset consisting of one or more rules written in MSPL language.
  - **EndRule**: After coding the MSPL rule, it is followed by an "EndRule" line to indicate the end of the specification of this ruleset.

  The following example illustrates the complete details.

  ```
  RuleName:Restrict device access
  Params:ACCESSIBLEDEV
  EventMap:open=ACCESSIBLEDEV
  ```

```
EventMap:read=ACCESSIBLEDEV
EventMap:write=ACCESSIBLEDEV
Desc:Restrict access to devices on the system. The application cannot
 arbitrarily access any device on the system.
Body:

/* Restrict access to devices on the system. The application cannot
 * arbitrarily access any device on the system
 */
any*. open(num,p,res,nam,op,unqid,flag,ret,arg)
| (res.get() == DEV) && !isaccessibledev(nam.get())
-->
{
  abort(''Opening device '', nam.get(), `` is disallowed'');
};
any*. read(num,p,res,nam,op,unqid,len,buf,off,flag,ret,arg)
| (res.get() == DEV) && !isaccessibledev(nam.get())
-->
{
  abort(''Reading from device '', nam.get(), `` is disallowed'');
};
any*. write(num,p,res,nam,op,unqid,len,buf,off,flag,ret,arg)
| (res.get() == DEV) && !isaccessibledev(nam.get())
-->
{
  abort(''Writing to device '', nam.get(), `` is disallowed'');
};
EndRule
```

- **ParamDesc.txt**: If the rulesets coded above introduced any new parameters, they have to be defined in this file. Section 5.2 details the various attributes that have to be defined for any new parameters. The following example illustrates the details.

```
Name:WRITABLEFILE
Scope:public
Type:MULTIPLE
Desc:File exceptions that can be created, renamed and modified. Values can
contain trailing '*' to indicate wildcard match.
EndParam
```

- **Policies.txt**: This file describes the high-level policy information and lists all the Parameters and Rulesets pertinent to the policy. The Parameters and Rulesets should have already been included in the appropriate files as explained before. By maintaining their definitions separate from their actual usage, we achieve reusability and easier maintainability. For a given policy search criteria comprising of application name, application class and username (note that vendor name can be supplied in place of the application name), there could be multiple matching policies. To account for this scenario, we use the notion of PolicyList for each set of search

criteria used as an index.

This file consists of the following data about the policy:

- **PolicyList**: This indicates the search criteria as:

  ```
  ``Application Name|Application Class|User''
  ```

  Example:

  ```
  ``PolicyList:gedit|editors|all''
  ```

- **PolicyName**: User identifiable name for the policy.
- **PolicyDesc**: A short, textual description of the policy.
- **Parameter Specification**: This is a list of Name-Value tuples. The entities expressed here include all the policy parameters applicable to this policy and all the rulesets enabled for this policy.

  The rulesets are expressed as follows. For every ruleset that needs to be enabled for this policy, a "ParamName" line is added with the RuleName of the ruleset. The "ParamValue" for this ParamName will be "1" to indicate that this ruleset has been enabled for the policy. Finally, an "EndParam" line is added to indicate the end of specification for this entity. It is not necessary to list those rulesets that are disabled for this policy here.

  The policy parameters are expressed as follows. For every parameter, a "ParamName" line is added with the name of the parameter. The values for this policy parameter are specified using "ParamValue". A parameter, depending on its type, can have one or more ParamValues associated with it. Finally, an "EndParam" line indicates the end of specification for this entity.

  After the parameter specification, an "EndPolicy" line is added to indicate the end of specification for this policy. If we have multiple policies for the current PolicyList, the next policy specification can be started right after the EndPolicy tag, beginning with the PolicyName tag.

  Finally, the specification of this PolicyList is terminated with an "EndPolicyList" line.

  Example:

  ```
  PolicyList:gedit|editors|all
  PolicyName:Gedit
  PolicyDesc:The policy restricts access to files that do not belong to the
  application. Certain operations deemed as privileged and applications are
  not allowed to be executed.
    ParamName:READABLEFILE
      ParamValue:/proc/meminfo
      ParamValue:/etc/localtime
      ParamValue:/etc/hosts
      ParamValue:/usr/share/locale/*
      ParamValue:/etc/gtk/*
      ParamValue:/usr/share/themes/*
    EndParam
    ParamName:WRITABLEFILE
  ```

```
    ParamValue:/dev/null
    ParamValue:/tmp/*
  EndParam
  ...
  ParamName:Restrict file creation in unowned directory
    ParamValue:1
  EndParam
  ParamName:Restrict unowned file overwrite
    ParamValue:1
  EndParam
  ParamName:Restrict unowned file rename
    ParamValue:1
  EndParam
  ParamName:Restrict network access
    ParamValue:1
  EndParam
  ...
EndPolicy
EndPolicyList
```

- **Import to Repository**: Once the three text files RuleDesc.txt, ParamDesc.txt and Policies.txt are updated with the new policy as described above, the information can be imported into the Policy Repository by running the script `populateBDB.sh` present in the `bdb` directory. The script gives the option of clearing up the current database before loading the data in the text files.

References

[1] A Language for Specifying MCC Security Policies, Secure Systems Laboratory, SUNY @ Stony Brook, 2005. Report prepared for Computer Associates, Inc.

[2] An OS-Neutral Abstraction of System Call Interface, Secure Systems Laboratory, SUNY @ Stony Brook, 2005. Report prepared for Computer Associates, Inc.

[3] An XACML Representation of MCC Security Policies, Secure Systems Laboratory, SUNY @ Stony Brook, 2006. Report prepared for Computer Associates, Inc.