SELinux Policy Development for the JADE Agent Framework

Ariel Segall

October 1, 2005

1 Introduction

One objective of the Secure Distributed Computing task for this year was the creation of an SELinux policy to secure the JADE agent framework. This is built upon last year's work creating an SELinux policy for the Aglets framework. As with last year, this year's policy attempted to segregate agents and the agent framework from users and other processes on the host. In addition, this year's work takes advantage of modifications made to JADE as part of this task which break agents into distinct processes. As a result, the policy produced this year can give individual agents their own security constraints, instead of forcing all agents to run in the same security context, as was the case with the Aglets policy. This allows individual agents to have customized security policies that meet their individual needs. Moreover, it allows the creation of a policy that enforces separation between agents, something that was not possible in last year's work.

In this paper, we provide a detailed overview of the security policy developed this year to secure the JADE agent framework. We include all of the information that a JADE administrator should need to set up an SELinux-enforced security policy for JADE agents. We also provide an outline of the procedures and tools we used to develop this policy, so that anyone undertaking similar policy development efforts in the future can take advantage of our experience. Finally, we suggest avenues for research and development that we think would benefit the SELinux community and make similar policy development tasks easier in the future.

2 A Brief Overview of JADE

The framework selected for this project was the Java Agent DEvelopment, or JADE, Framework. This framework was selected because it was open source, actively supported, had cleanly separated functionality, and appeared to offer the best chance of needing minimal modifications to distinguish agents from each other at a process level. JADE turned out to be a good choice in these regards.

The most fundamental unit of JADE is the *container*. Containers serve as environments in which agents can run. They handle some basic administrative functions such as directing interagent communication. Containers may support any number of simultaneous agents, including zero. Containers combine to form a *platform*. A platform is defined as all containers which have the same manager, whether located on a single host or distributed over many hosts. One container in a platform is designated as the *main-container*. The main-container is simply a container which holds the management agents for the platform, which handle centralized administration functions such as maintaining an agent registry. Containers and agents in JADE are designed to be location-ignorant, using RMI calls to automatically route communications to the correct IP address and container without the container needing to be aware of those details.

JADE is a platform independent Java-based system, and was not designed with SELinux in mind. A container, along with all of its agents, lives in a single Java Virtual Machine (JVM), which is one complex process. Sometimes, multiple containers will run in one JVM. SELinux's type enforcement system is based on regulating processes and their access to resources, and has no way of enforcing distinctions within a process. A JVM, or any other process, is a black box to SELinux, and any agents running within it will not be distinguishable.

In order to regulate JADE agents using SELinux, part of this year's task was to modify JADE to separate agents at the process level. The JADE modifications are described in detail in the **JADE Technical**

Modification Summary. With the modified JADE, a container can be configured to accept only agents which should be run in that container's security context, and to spawn new containers in separate processes for agents which should run in different contexts. This modification allows SELinux to regulate the behaviour of a process containing a container. The container will only contain agents that belong in a given security context, which is a reasonable approximation of regulating the agent itself.

3 Policy Goals

Our primary goals in creating the JADE policy were to:

1. Protect the host and agent framework from hostile or misconfigured agents
2. Protect the user and his data from hostile or misconfigured agents
3. Protect agents from malicious users or processes
4. Allow separate policies to be easily created for individual agents

Goals 1, 2, and 3 were accomplished with the Aglets policy last year. Goal 4, however, could not be accomplished using Aglets, since SELinux was unable to distinguish between Aglets agents. With the new modified JADE, we have created an agent framework in which different agents can have different policies, allowing us to complete the full set of goals.

The use of separate agent policies allows us to take full advantage of SELinux's security features to regulate the behaviour and access of potentially untrustworthy agents. A single agent platform might include, for example, a system monitoring agent which checks to see if the system software is up-to-date, an auditing agent which logs accesses to sensitive files, and an agent which monitors network activity for suspicious access attempts. We don't want the agent responsible for up-to-date software to be able to access arbitrary network ports, let alone have any information about sensitive data on the system. By maintaining separate policies for each agent, we can make sure that each agent has those permissions it requires for its job, but cannot take advantage of the permissions granted to a more trusted agent.

The policy is not intended to prevent covert communications between agents and other entities, nor is it intended to hide the framework or agents from users. Many routine and required actions on the part of a program can be used to create covert information channels, and there are many forms of indirect monitoring which will reveal the presence of as complex a program as JADE and which cannot be trivially blocked. The removal of covert information channels remains a difficult and open area of research. We merely try to restrict access to system resources so that neither agents nor host processes have direct access to resources they should not.

4 Policy Results

To meet our goals for this task, we produced a set of policy macros which can be used to create SELinux policies for individual JADE agents. As part of our development process, we also created a similar set of policy macros which can be used to create SELinux policies for Java programs. In this section, we will outline the structure of our macros, and how they relate to each other and to policies for JADE agents. We discuss details about the purpose and use of the Java Virtual Machine and JADE macros. Finally, we will discuss examples of JADE policie which use our macros.

4.1 High-Level Policy Overview

The JADE framework is a structure which agents use to provide basic functions such as interagent communication, rather than an application in its own right. As such, we never create a JADE policy. Instead, we create SELinux policies for agents. Each agent runs in a JADE container, which in turn runs in a Java Virtual Machine. The macros we created for this task are designed to make the process of creating policies

for JADE agents easier by providing the permissions necessary for that underlying support structure to perform correctly.

Permissions in a JADE agent policy come from three layers: permissions for the individual agent; permissions for the JADE container in which the agent is running; and permissions for the underlying Java Virtual Machine in which the container is running. All three of these layers run in a single process. SElinux cannot detect distinct layers within a process, and our policies must therefore include all three sets of permissions. We created layered macros for the JVM and JADE container permission sets so that a policy writer creating an agent policy could call a single macro which would incorporate all of the necessary functionality for the agent's support structures.

One of our design goals when creating the JVM and JADE macros was that they should be easy for a system administrator who is only minimally familiar with SELinux to use. The permissions for each program which were most dependent on the host system's configuration were pulled out into separate macros. These system-specific macros were designed be easily modifiable central locations for those permissions which a system administrator is most likely to need to change.

4.2 JVM Policy

The JADE framework, as its name suggests, is a Java program, and its containers run in Java Virtual Machines. The first step in creating a policy for a JADE agent, therefore, is to create a policy for the JVM. We began our development process by updating and modifying an older JVM policy for Fedora Core 2's targeted system policy written by Machon Gregory. This policy contained the permissions that became our JVM macros.

The JVM macros are designed to grant a Java program all of those permissions necessary for the underlying JVM to run, and no more. Permissions which the JVM requested but did not require, such as access to the temporary directory to record performance monitoring data, were denied. We determined which permissions were fundamental to the JVM using a minimal Hello World program which required only write access to the screen and read access to its installation directory.

Our final JVM macro set had two parts: the three-line `uses_java`, which contains permissions based on our Java installation in `/usr/bin/java` with a type of `jvm_exec_t`, and the fourteen-line `runs_in_jvm` which contains `uses_java` as well as all other permissions necessary for the JVM process to run. We distinguished between these macros in order to give the system administrator who installed them a central location to find the permissions which he is most likely to need to modify. The inclusion of `uses_java` in `runs_in_jvm` means that an SELinux policy writer only ever needs to call a single macro in order to include the JVM permission set.

In order to use the JVM macros, two additional files are necessary: `jvm.te` and `jvm.fc`, which together define the new type `jvm_exec_t`.

4.2.1 Types

Only one new type is created for the JVM macros, `jvm_exec_t`. `jvm_exec_t` is used strictly to constrain the execution of the `java` executable.

The `jvm_exec_t` type should be declared in `jvm.te` and assigned to a file in `jvm.fc`. The `java` executable, `/usr/java/j2sdk1.4.2_08/bin/java` on our system, should be given the `jvm_exec_t` context. Sample `jvm.te` and `jvm.fc` files are included in the Appendix.

Because a Java program running on an SELinux system should run in that program's domain rather than a generic Java domain, type transitions through `jvm_exec_t` are not required. An `execute_no_trans` permission is included in the `uses_java` macro, discussed below.

4.2.2 Using the Macros

Two macros make up our JVM macro set: `uses_java` and `runs_in_jvm`.

uses_java `uses_java` is not intended to be called in user-created policy. Instead, it is designed to be a helper macro called from within `runs_in_jvm`. The `uses_java` macro contains those permissions directly

connected to the location of the java executable on the system. It is designed to be an easily modifiable central location for those permissions which a system administrator is most likely to need to change.

`uses_java` takes one argument, a domain. It grants the permissions required to access `/usr/bin/java`, as well as permission to run the `java` executable without changing the security context of the calling process.

runs_in_jvm The `runs_in_jvm` macro is designed for use in application policies for programs running within a Java Virtual Machine. Calling this macro ensures that the application, and by association the underlying virtual machine, is able to access all of the files and resources that Java requires to operate.

`runs_in_jvm` takes one argument, a domain. It grants all of the permissions required to run a JVM, including access to shared libraries with the `uses_shlib` macro. It also incorporates the `uses_java` macro, to ensure that the user only needs to call a single macro in order to enable the full functionality of the supporting Java layer.

4.3 Modified JADE Policy

Our goal when creating macros for the modified JADE framework was to produce tools which a system administrator could use to create secure policies for JADE agents without worrying about the support layers which the agent relies on, namely the JADE container the agent runs in and the JVM that the container runs in.

The modified JADE framework being used here is that described in the **JADE Technical Modification Summary**. The modified JADE has an -SELinux option that causes containers to be context-aware, and to reject any agent sent to them which should not run in their context. These containers therefore require the ability to interact with SELinux in order to determine whether they are running in the correct context for the incoming agent. In addition, the modified JADE allows containers to optionally spawn a new container in the correct context for an incoming agent, and redirect the agent to the new container. Because the SELinux context of new containers is determined by the context of the launch script, containers which are allowed to create new containers run previously specified scripts which transition to the correct context. In order to make restricting modified JADE agents and containers easier, we separated out the container-creating permissions into a different macro than the rest of the modified JADE permissions.

4.3.1 Types

The modified JADE macros introduce one new type, `secure_jade_file_t`. This type is used to regulate access to the classes, libraries, and other files associated with JADE. It is not associated with any executables.

Unless only one JADE agent policy will be installed on the system, a minimal support policy similar to the `jvm.te` and `jvm.fc` described above should be created to define `secure_jade_file_t` and associate it with the JADE files. On our system, all non-executable files in `/usr/share/secure-jade/` were assigned the `secure_jade_file_t` context.

In addition to `secure_jade_file_t`, the user will need to create a new domain type and an executable type for each agent's application policy. Each new security context for a JADE agent requires a launch script or other executable that creates a new JADE container. SELinux determines a process' security context based on the type of the file which was executed to launch it. Therefore, although the contents and function of the executable can be identical, but the files' security contexts must be distinct.

4.3.2 Type Transitions

The core type transition in a JADE agent policy is the `domain_auto_trans` from the type of the launching process or process to the agent's desired security context, via the agent-specific executable which launches a new JADE container. Usually the launching process context will be that of another container, although it could also be the system or a user. This transition allows a process to create a new JADE container in a given security context.

Dispatching containers, which can spawn new JADE containers for incoming agents, need an additional `can_dispatch` macro. This macro grants `domain_auto_trans` permission from the type of the dispatching container to the type of a new container, via that new container's executable script.

4.3.3 Macros

The JADE macro suite contains three new macros:

- `access_secure_jade_code`, which contains location-specific permissions
- `runs_in_secure_jade_container`, which contains `access_secure_jade_code`, `runs_in_jvm` and additional permissions required to run an empty JADE container
- `can_dispatch`, which contains the permissions required for a JADE container to launch another JADE container running in a different security context using an executable script

access_secure_jade_code The `access_secure_jade_code` macro, like the `uses_java` macro described previously, is a helper macro designed to be a central location for a system administrator to find those permissions most likely to depend on a system's configuration. It grants permission to access JADE files with the `secure_jade_file_t` context, as well as those permissions necessary to access JADE's installation location.

`access_secure_jade_code` is normally called from within `runs_in_secure_jade_container`, and takes a single domain as its argument. It grants access to resources of `secure_jade_file_t`. Because our JADE isntallation was in `/usr/share`, it also grants directory navigation access within `/usr`.

runs_in_secure_jade_container The `runs_in_secure_jade_container` macro is designed to be the all-in-one macro for agent policies running in a modified JADE container. In addition to granting all of the permissions necessary for the JADE container to operate, it uses `runs_in_jvm` to add permissions required by the JVM in which the container runs. Permissions required for the JADE container include:

- networking permissions for intercontainer communications
- screen printing permissions to allow error messages to be shown to the user,
- SELinux permissions to allow the container to determine whether its context is appropriate for a new agent

This macro accepts a single domain as an argument. It includes the `access_secure_jade_code` and `runs_in_jvm` macros, as well as a suite of permissions necessary for the underlying container functions which support the agent. Every JADE agent policy should call this macro.

can_dispatch In modified JADE, SELinux-aware containers will only accept incoming agents which should run in their own security context. Some containers can start up new containers of a different context in response to an incoming agent. These are called dispatchers.

The `can_dispatch` macro adds the necessary permissions for a container to launch an executable script of a particular type, and spawn a new process which will transition into the appropriate new context. Because the executable scripts we use to launch new containers are shell scripts, this includes permission to execute files of type `shell_exec_t`, such as `/bin/bash`. The user should be aware that any agents running in a dispatching container will have access to the shell. However, because domain transitions are still limited to those domains explicitly permitted by the user, and the ability to use the shell does not change which resources the program has permission to access, we do not feel that this is a significant security risk.

The `can_dispatch` macro takes as arguments the domain of the dispatching container, the type of the executable script, and the domain type belonging to the new container which the dispatcher should be able to launch. It grants `domain_auto_trans` permission between these three. In addition, it grants the dispatcher `shell_exec_t:file read execute`.

4.3.4 Example: JADE Main Container Policy

A JADE main container, although a necessary component of any JADE platform, is fundamentally no different from any other JADE container running a set of agents. The agent management functionality of the main container is provided by a set of management agents. One optional feature in a main container is

the Remote Management Agent GUI, which is a user-friendly tool for the platform administrator to perform management operations such as viewing the containers and agents running on the platform and creating new agents remotely.

Because a main container is required for a JADE platform, we needed to create an appropriate SELinux policy. Since the only difference between the management agents and any other JADE agent is the nature of their functions, we will use our main container policy to illustrate the use of our JADE macros for an agent.

```
type secure_jade_main_t,domain;
role sysadm_r types secure_jade_main_t;
role staff_r types secure_jade_main_t;
type secure_jade_main_exec_t,file_type, exec_type;

domain_auto_trans(sysadm_t,secure_jade_main_exec_t, secure_jade_main_t)
domain_auto_trans(staff_t, secure_jade_main_exec_t, secure_jade_main_t)
```

The new types which must be declared in an agent policy are the domain that the agent should run in, and the context of the executable file which will launch the agent. Because the main container will be launched by the user from the command line, we add domain transition permissions for the staff and system administrator, allowing them to run the management agents.

```
runs_in_secure_jade_container(secure_jade_main_t)
```

This call to our JADE container macro adds all of the necessary permissions required by the container in which our management agents run. It also adds the permissions for the JVM in which the container, and therefore are agents, is run.

```
allow staff_t secure_jade_file_t:dir {search read};
allow sysadm_t secure_jade_file_t:dir {search read};
```

Everything in our JADE installation directory, `/usr/share/secure-jade`, was assigned type `secure_jade_file_t` by default. Although the script that launches this main container is of type `secure_jade_main_exec_t`, the users who wish to run it also require navigation access to the `secure_jade_file_t` subdirectory it was installled in.

```
allow secure_jade_main_t node_lo_t:tcp_socket node_bind;
allow secure_jade_main_t net_conf_t:file read;
allow secure_jade_main_t lib_t:file getattr;

allow secure_jade_main_t xserver_port_t:tcp_socket {send_msg recv_msg};
allow secure_jade_main_t staff_home_dir_t:dir {getattr search};
allow secure_jade_main_t staff_home_t:dir search;
allow secure_jade_main_t staff_home_t:file {write getattr};
allow secure_jade_main_t home_root_t:dir {getattr search};
allow secure_jade_main_t staff_home_xauth_t:file {getattr read};
```

These permissions are the core of our agent's policy. They define the functionality that differentiates our management agents' activities from their underlying support structure. In this case, our management agents require some additional permissions related to the network, and a minor addition to library permissions. Most of the permissions, however, are related to the Remote Management Agent's GUI, which requires access to the X windowing system. To set up a new window in X, it requires network access to the xserver's port, as well as significant permissions related to the home directory of the user running the agents, which are required to access to the .Xauthority file. Removing any one of these permissions causes our Remote Management Agent to fail, so we grant them the requested home directory access; however, if we were particularly concerned about potential malicious activity on the part of our management agents, we would set up a new user account with a distinct home directory context whose sole purpose was to allow these agents X access without potentialy sensitive file access.

4.4 Policy Limitations

One side effect of implementing the minimal policy for JADE was a noticeable performance degradation. This degradation showed up primarily in creating new agents or in operations which opened new X windows. We hypothesize that these slowdowns are the result of non-critical failures within the JADE code, in places where JADE's preferred operation was not permitted and falling back to the next option required waiting through some timeout.

The JADE macros we created do not currently restrict network access by port. JADE does not use a set port, and the choice of ports may differ between systems. The security of the JADE policy could be improved by adding port restrictions, ideally using a new port type which could be adjusted according to the needs of the system administrator.

5 Development Overview

When creating the JADE policy, and the underlying policy for the JVM, we first determined how we wanted to organize our policy. This design was used to shape our policy and remind us of our goals, rather than as a restriction. To develop the policies, we first added permissions until the program ran successfully, and then removed permissions until no additional lines could be removed without preventing the program from performing correctly. Although slow, this process helped us avoid two of the largest dangers in SELinux policy development: allowing actions whose purpose the developer is unsure of, or preventing the program from functioning. There is still room for improvement in our development procedure, and our suggestions for how to make policy development easier and more secure can be found in the "Recommendations for the Future" section.

5.1 The Underlying SELinux Policy

For development, we used a Fedora Core 3 machine, which came with two different policy options: a targeted policy, which allowed most programs unrestricted permissions but locked down a handful of high-risk programs, and a strict policy, which locked down everything. The big advantage of the targeted policy is that it's trivial to get new software up and running. The big disadvantage is that while you can lock down a potentially malicious program to keep the software from corrupting the system, you cannott restrain a potentially malicious user in any way. Since one of our policy goals was to protect the agents from the user or other malicious software, we needed to use the strict policy.

The choice of strict or targeted policy affects more than just in which directories the program policy is installed. The targeted policy and the strict policy have entirely different assumptions about a user's default context. These assumptions affect the role permissions and type transitions for every program policy. These differences make it impossible to interchange application policies between the two underlying policies.

5.2 Policy Design

The scenario we used when designing our policy was a system administrator running a network of SELinux-enabled Fedora Core 3 machines. The administrator wishes to run system monitoring agents on the user machines using JADE. Both the users and the agents are not trusted, and need to be constrained to prevent possible interference with each other.

The choice of a network administrator instead of an SELinux policy developer as our primary user led to some specific design goals for our policy:

- Nothing other than the files we produce should be necessary to allow a JADE installation to minimally operate.
- An administrator should need to make only minimal changes to the JADE policy, specific to his installation, and these should be clearly visible.
- Policy for an agent should not include the policy for JADE except as a macro or similar black box, for ease in both creating and reviewing the agent policy.

Using these goals as a starting point, we decided that our policy should distinguish between the different layers necessary for a JADE agent to run, and it should do so in a way that was easy for a novice policy writer to understand and use. We wished to cleanly separate the permission set of the JADE agent which is a policy's focus from the permission sets of the JADE container in which the agent runs and the JVM in which the JADE container runs.

With those permission sets as a rough goal, we could then make educated guesses as to what we wanted our output to look like. We wanted to produce tools which novice policy writers could use to easily create policy for JADE agents with different access demands.

Because each agent runs within a JADE container, the basic container permission set needs to be included in any agent policy, ideally with the permissions clearly separated. Each JADE container runs within a Java Virtual Machine, so we also need to include the JVM permissions in any JADE application policy. The agent permissions, such as those granted to the management agents in the main container, then define the individual application policy. In order to make agent application policies as simple and agent-focused as possible, we created M4 macros for the included permission sets, the JVM and the JADE container.

We split the macros for each permission set into all-purpose macros and helper macros. These helper macros contain any permissions which might vary significantly with the system configuration, pulling them into a single location for easy adjustment on the part of the system administrator. The all-purpose macros use the helper macros in addition to establishing all of the other permissions the program needs, so that the policy writer only needs to add a single line to his policy to gain the necessary permissions for underlying layers of the application.

Because we wanted modular policy macros for our result, with Java and JADE functions clearly separated, we tackled our large problem in pieces. Our first step was to take a JVM policy for Fedora Core 2's targeted system policy created by Machon Gregory and update it for a Fedora Core 3 system running the strict system policy. Then, we created a policy for a basic JADE container. Finally, we created a policy for a main JADE container, complete with GUI-based management agents. Once we had created these policies, we compared them to determine the core permission sets for each, and used them to create our desired macros.

5.3 Policy Creation

Our first priority was to create policies, however insecure, which would allow our programs to run on an enforcing SELinux system. Because we would later be going through each policy and removing excess permissions, we didn't have to worry about accidentally creating a security risk; we could concentrate entirely on making our programs work. To create these draft policies, we used two tools: MITRE's Polgen, and the standard SELinux audit2allow tool. Each had advantages and disadvantages. For our purposes, creating new and minimal macros for broad use by comparing related policies, `audit2allow` turned out to be the best choice.

5.3.1 `polgen`

Polgen is a tool from MITRE which is designed to help the policy creator generate new and reasonably secure policies quickly by detecting patterns in a program's operation and recommending policy based on those patterns. We used version 1.1 of `polgen` for our testing.

`polgen`'s strengths are that it gives the policy creator insight into the behavior of the program, and creates a policy quickly with enough recommendations and user oversight to avoid obvious pitfalls. Because we had already analyzed the structure of JADE when selecting it for our task, the pattern insights did not give us significant advantage. The polgen-created policies use pattern-specific macros that come with the tool, which are useful for quickly understanding what purpose each group of permissions serves, detracted from our specific task. Because we were creating our policies both for export to other systems and for extremely minimal permission sets, each macro call had to be replaced with the equivalent list of permissions for testing and compatability. This delay cost us the time we'd saved using the tool to create the policy.

Although `polgen` helped us create an SELinux policy that allowed JADE to run correctly very quickly, it were going one step further and turning our policies into streamlined macros. This meant that we needed to translate each `polgen` macro back into normal SELinux policy for detailed analysis. We discovered that many of the permissions we generated by our use of `polgen` were duplicates, which we wanted to weed out in

order to have an easily understood policy. Also, the `polgen` pattern macros are part of the `polgen` package, and we did not want our hypothetical system administrator to be forced to install macros from a program he may not be using.

5.3.2 `audit2allow`

Audit2allow is the classic SELinux policy creation tool. It takes SELinux audit messages, which are generated whenever a program attempts an action for which it does not have permission, and turns them into potential allow messages which would grant the denied permissions.

In order to create our policies using audit2allow, we used its verbose option to see what the programs were trying to do when permission was denied. The `audit2allow -v` output was far more helpful for file accesses than for network or memory activity, but usually gave us enough information that we could make educated guesses about why the program was requesting the ability to perform a given action. We generated rough policies by selecting the most essential-looking permissions, adding them to the program policy, and checking to see if the program would successfully run or not. If so, we proceeded to the minimization step; if not, we used `audit2allow` again to add more permissions to the policy.

This iterative approach to adding permissions was chosen for two reasons. First, we were doing our policy development with SELinux in enforcing mode, so only the permissions which were being denied on this run were visible, and it took many runs of adding permissions before the programs were able to complete their tasks without crashing midway through due to denied permissions. Secondly, our next step was to remove permissions. When adding permissions by hand, it is possible to make educated guesses about which permissions are actually necessary, and hopefully not bother to add useless permissions which we will then need to test and remove later. This was particularly helpful when creating the JADE application policies, because we already had experience testing the JVM policy, and knew that several requested permissions were not necessary and could be left out from the start.

5.3.3 An alternate method

Since the next step in the process is to remove all excess permissions, we could skip the tedious process of expanding the policy step by step, and simply start by giving the program all of the permissions it requests when in permissive mode. We didn't realize initially how thorough the policy minimization step would be, and so have not yet tested this simpler method, which should work just as well in theory.

5.4 Policy Minimization

Even the best policy creation techniques tend include excess permissions. This can happen for a number of reasons. Preexisting macros are often designed for broader cases than those in which they are actually used, resulting in the addition of permissions the program doesn't need. Likewise, programs may generate error messages if certain permissions are denies, but these "errors" may turn out to be simple warnings whose occurance is not fatal, or even significant, to the operation of the application. Finally, in the course of generating permission lists, simple human error can often perceive the need for certain access rights where no need exists. Especially when generating policy by best guess from incomplete information, extra permissions are inevitable.

Our answer to this was to perform a line-by-line policy minimization once we had a working SELinux policy that allowed Java or JADE to successfully run. We took each line in the policy in order, commented it out, and reran the program to see if it still worked. Any line which did not cause our tests to fail was removed permanently. Lines of SELinux policy often contain several or even a dozen separate permissions, ranging from permission to get a file's attributes to permission to read, write, or execute the file. Particular permissions which might severely affect security, such as file execution, were tested individually, in addition to testing the associated allow line. Due to the sheer volume, however, we did not individually test the more innocuous permissions. For a truly minimal policy, analysis of every single permission would be required, but to do so by hand starting from a standard SELinux policy would be an extremely tedious process.

5.5 Macro-izing Existing Policies

In order to enable the creation of agent policies by easily including permissions for the underlying JVM and JADE container layers, we needed to convert our existing JVM and JADE container policies into M4 macros. Because our goal was to enable multiple agents on the same system to be constrained separately, each agent requires its own application policy. By using macros for the permissions which are required for the underlying support structure, the application policy for an agent can focus on those permissions which define the agent's functionality.

When designing the JADE macros, we needed to figure out how the macros would be used and select divisions which would be as helpful as possible to our hypothetical JADE administrator. We eventually settled on separating out two sets of installation-dependent permissions, those specific to the location of the Java binaries and those depending on the location of the JADE classes, which were placed in program-specific helper macros; the JVM permissions, which could be reused for other Java-based policies; and the JADE container permissions. The JVM and JADE container macros both call the relevant helper macros, and the JADE macro calls the JVM macro. By placing the installation-dependent permissions in a separate macro, we ensure that the SELinux system administrator installing Java or JADE only needs to update a single short section in order to allow any policy relying on these macros to work correctly on his system. The policy writer, on the other hand, only needs to include a single macro call in his agent policies, and the policy should work correctly on any machines with our macros installed.

Because we had already created the "layered" policies we wanted from our macros by generating separate JVM, basic JADE container, and main container policies, creating the macros themselves was a simple matter of separating out what made each one distinct. The JVM macros contained all permissions required by Java, the JADE macros contained the JVM macro and all additional permissions required by a basic JADE container, and our test application policy for the JADE main container contained the JADE macros and additional permissions required by the management agents and their GUI.

Converting a policy or policy segment into a useful macro is not quite as simple as changing all of the instances of the program type into variables. Types such as `jade_file_t` which are used by all containers cannot be defined in a macro, since duplicate declarations will prevent the policy from compiling. The macro writer needs to decide whether to create separate type and file contexts files to handle types that the macros require, or whether to require the policy writer may to declare a type in his application policy which is used only within the macro. In addition, any macros which contain permissions that a system administrator may need to change upon installation should be called to the administrator's attention and explained in the macro's comments or related documentation. In our `access_jade_code` macro, for example, we explain that the original policy was designed with JADE installed in `/usr/share/jade`, and that we expect the JADE files to have the type `jade_file_t`.

6 Recommendations for the Future

- The policy minimization process could be dramatically sped up by the creation of an automated tester which takes in a draft policy, a program, and the program's expected output and processes it until no more lines can be removed without causing the program to no longer produce its expected output. This functionality requires that the tool be able to fully test a potentially GUI-using application and distinguish good operation from bad operation, which would be exceptionally difficult. Even a primitive tool which compared text output to some ideal value would be helpful, however. If the basic testing functionality is achieved, an improved version of the tool could simply be hooked up to `audit2allow` and generate its own draft policy, since any excess permissions audit2allow grants would be removed. Add to that a logging function, which records what error messages (if any) there were when the program failed to perform and which permission loss that corresponded to, and the policy writer's work would be limited to checking for violations of good security policy and debugging. Such a program, not being limited by human patience, could also check specific permissions, not just whole lines of them.

- One difficulty with creating policies for frameworks like JADE, in which many different but related programs will want application policies distinct from each other or the framework policy and which may wish to be installed separately, is the policy compiler's assumption that multiple type definitions are

always a serious enough problem to stop compilation. Certainly, in a world where every program comes with its own policy, we want to prevent accidental name overlaps which could dramatically change the effect of a policy. However, the workarounds for JADE-like frameworks are cumbersome and error-prone. A method for deliberately tagging known-to-be-identical type declarations, or reassuring the compiler that the overlap is deliberate, would drastically simplify matters.

- The `audit2allow` -v option gives the policy developer some insight into what the program is attempting to do when SELinux denies it permissions, but is of very little use when it comes to figuring out non-file functions, such as networking or shared memory access. The `strace` program provides much more information about the program's activities, but does not connect them with the SELinux error messages, and even `polgen`'s trimmed version of the `strace` output contains more information than any person can reasonably understand. If policy developers are to be able to make reasonable security decisions regarding the permissions a program requests, they will need tools which will connect detailed information about what the program is attempting to accomplish with the SELinux permissions required for it to be successful.

- As SELinux becomes more commonly used, and programs start being distributed with associated SELinux policies, there will be a need for tools aimed at creating exportable policies instead of simply making a single system work. Macro-based policy creation, whether using tools such as polgen or home-grown macros that a particular policy creator finds useful, can be a simpler process producing more easily read policies, but those same macros make the resulting policies difficult to export. Either the receiving system needs to have identical macros installed, or the policy's macros need to be removed and replaced with the corresponding permissions. As we discovered with polgen, the need to painstakingly remove the same macros that the tool had tried to use to make our lives easier eliminated much of the effort we had saved. A tool which could process a .te file and expand included macros would dramatically increase the usefulness of tools like `polgen` which use the macro system to try to move policy creation to a higher abstraction level. Even better, a tool which created a program-specific macro file that could be shipped with the policy by allowing the user to choose for each macro called in the policy whether to selectively ignore it, expand it, or copy the macro code into a program-specific macro file would give policy writers the ability to use all the tools at their disposal, minimize policy size, and not worry about missing macros on the receiving system.

7 Summary

We have created SELinux policies to secure the JADE agent framework from itself and the user, and generated macros for both JADE containers and the Java Virtual Machine which could be used in the broader community to create more secure custom policies for JADE agents and Java programs. In the process, we have come up with improved methods for generating minimal policy, and discovered numerous potential avenues for future research and development to further the ease and security of SELinux policy development.

Appendix A: Policy Development Tips

Tips for Creating Initial Policies

- Keep an eye out for permissions and types pertaining to the printing of messages and errors, such as `staff_dev_pts_t`. Including these is often a good idea. Although they may not be necessary for the effective functioning of the program, silent failures don't make policy building any easier. The extraneous permissions can always be removed once testing is complete.

Tips for Testing Policies

- When creating policy for complex programs, establish a consistent, ordered routine to test all of the application features you care about. If you can figure out which functions are most likely to break, test

them first and save yourself the effort; or, if you have slow and non-fundamental functionality, test it only periodically and carefully track what might have changed. Consistency in routine, especially in a long and repetitious task like line-by-line testing, helps prevent the accidental skipping of a particular feature.

- List all of the features you'll care about in the final functioning of the program to make sure they're actually being tested.
- It's possible to break a working policy by adding permissions. We hypothesize that this is a result of the program being able to start an action but not finish it, without having any provisions for being interrupted.
- Error messages don't necessarily mean anything you care about is broken. JADE, for example, wants write permission to whatever directory it's launched in to create informational files for each container, and generates long error messages if it doesn't have it. The lack of permissions don't prevent it from running just fine, however.

Tips for Minimizing Policies

- Keeping good comments is even more important when creating policy with line-by-line minimization than it normally is. It's easy to accidentally skip one step of a testing routine and need to redo work, so tracking your reasoning will help you a lot later.
- Comment out lines that you plan to remove and eliminate them all at one time after you've finished most of your testing. It's a lot easier to remove a single hash mark than it is to recreate the particular line you deleted.
- If you have time, test individual permissions as well as complete allow lines. When adding permissions incrementally, we discovered that programs sometimes request specific permissions such as `getattr` for a given target type that they don't actually require for successful operation. Especially if you were generous when creating the initial policy draft, paying extra attention to potentially hazardous individual permissions in the policy minimization step is a good idea.

Tips for Creating Exportable or Macroized Policies

- Certain complex programs such as the JVM require an additional application to use. When creating policy for these complex programs, it is important to keep these applications in mind, no matter how minimal they may be. The location of the application used during testing will affect the final policy, even if it requires no additional permissions when operating. Make sure you mark and remove any application-related permissions, and test the program using an application in a different location to verify that the permissions derived from the application's location were not also necessary for the main program to function. For example, the `hello-world` program that I used to create and test a JVM with minimal additional functionality lived in my home directory. Unlike the required permissions related to JADE's installation directory, the `home_root_t` and `user_home_t` permissions required by `hello-world` would be not only irrelevant but a serious security hole for future users of the JVM policy.
- The SELinux policy language does not seem to be designed with large, modular programs with overlapping policies in mind. When we began this project, we hoped to have independently usable policies for a basic JADE container or a main JADE container that could be installed separately or together according to the system needs. Unfortunately, duplicate type declarations will prevent the policy from compiling. Both the `jade_basic` and `jade_main` policies need `jade_file_t` to be declared exactly once. Putting it in the user's hands is certainly a possibility, and one we chose because in our centrally-administered scenario there would not be a need for the main container to ever run on the same system as the monitoring agents. Other long-term solutions include using checks such as the SELinux policy `ifdef` to determine whether related policies have been installed or providing a program-specific utility

policy that defines types such as `jade_file_t` which are used by any JADE-related policy but contains no permissions.