# History-based Access Control for Mobile Code


Anurag Acharya[†]
[†] Department of Computer Science
University of California, Santa Barbara
acha@cs.ucsb.edu

Guy Edjlali[‡]
[‡] Department of Computer Science
Wayne State University, Detroit
edjlali@ece.eng..wayne.edu



TRCS97-25

Dec 1, 1997



**Abstract**

In this paper, we present a *history-based* access-control mechanism that is suitable for mediating accesses from mobile code. The key idea behind history-based access-control is to maintain a selective history of access requests and to use this history to improve the differentiation of between safe and potentially dangerous accesses. Combined with a suitable identity-based mechanism (eg, digital signatures), it can significantly expand the set of programs that can be executed without compromising security or ease of use. We present *Deeds*, a history-based access-control mechanism for Java. Apart from its history-based nature, the other important feature of *Deeds* is its capability to install and compose multiple user-specified policies. Access-control policies for *Deeds* are written in Java, and can be updated while the programs whose accesses are being mediated are still executing.


## 1 Introduction

The increasing availability and use of various forms of mobile code (eg, Java [12], Javascript [8], ActiveX [23]) has significantly changed the way in which programs are distributed and run. While code has been distributed over networks as long as `ftp` has been around, the explicit nature of the downloading process and the heterogeneity of machine architectures ensured that importing code into an environment was infrequent and was done only by a small number of people. The advent of machine-independent mobile code and the integration of mobile code with web browsing has changed that – mobile code can be transparently downloaded and executed in your environment without as much as a mouse-click.

The popularity of mobile code creates an access-control dilemma. On one hand, use of mobile code can significantly improve the visual appeal of web pages (via animation [27, 32]) as well as allow servers to handle many more clients with the same resources (by offloading computation to clients [1, 31]). In addition, integration of mobile code with web browsing creates a social expectation that mobile code should be as easily and as freely accessible as web pages. On the other hand, the very popularity and ubiquity of mobile code increases the likelihood that malicious programs will mingle with the benign ones. One would like to ensure that there is no loss or leakage of information due to the use of mobile code. Success of the "transparent mobile code" model depends on being able to do this.

Ease of use presents another conflicting requirement. The popularity of web browsing and transparent execution of mobile code is primarily due to the simplicity of its user interface. Access-control mechanisms that are complex or intrusive (eg dialog boxes that pop up frequently) are likely to be turned off or bypassed. It is critically important to keep the user interface simple – security that is turned off helps no one.

To reassure users about the safety of their data and to keep the user interface simple and non-intrusive, systems supporting mobile code have chosen to err on the side of conservatism and simplicity. Depending on its source, mobile code is partitioned into *trusted* and *untrusted* code. Code is considered trusted if it is loaded from disk [9, 11] or if it is signed by an author/organization deemed trustworthy by the user [11, 23]. Untrusted code is confined to a severely restricted execution environment [9] (eg, it cannot open local files or sockets, cannot create a subprocess, cannot initiate print requests etc); trusted code is either given access to all available resources [23] or is given selective access based on user-specified access-control lists [11].

For the programs considered untrusted, these mechanisms can be overly restrictive. Many useful and safe programs, such as a well-behaved editor applet from a company that is not known to the user, cannot be used since it can not open local files. Implementing potentially revolutionary resource-sharing models such as *global computing* [5] requires that all communication be routed through brokers, significantly limiting the set of problems that can be efficiently tackled. For programs considered trusted, these models can be too lax. Errors and bugs,[1] if not malice aforethought, can wipe out or leak important data. Combined with a suitable audit trail, digital signatures do provide the ability to take legal recourse if need be. From the point of view of protecting the data, however, this is much like closing the proverbial barn door after the horse.

In this paper, we present a *history-based* access-control mechanism that is suitable for mediating accesses from mobile code. Combined with a suitable identity-based mechanism (eg, digital signatures), it can significantly expand the set of programs that can be executed without compromising security or ease of use. We have implemented this mechanism for Java. The key idea behind history-based access-control is to maintain a selective history of access requests and to use this history to improve the differentiation of between safe and potentially dangerous accesses. For example, consider an access-control policy that allows a program to open local files for reading as long it has not opened a socket and allows it to open a socket as long as it has not opened a local file for reading. Irrespective of the source of the program, such a policy can ensure that no disk-resident data will be leaked. For another example, consider a policy that allows a program to open sockets only if it has not opened local files; to open local files for reading only if it has not opened a socket; to create local files; and to open a local file for modification only if it has been been created by itself (or a previous instance of itself) and it has not opened a socket. Such a policy can be used to allow the use of editor/spreadsheet applets.

We first present motivating examples for history-based access-control and discuss issues that have to be resolved for such mechanisms. Next, we describe *Deeds*,[2] our implementation of history-based access-control for Java. Apart from its history-based nature, the other important feature of *Deeds* is its capability to install and compose multiple user-specified policies. Access-control policies for *Deeds* are written in Java, and can be updated while the programs whose accesses are being mediated are still executing. *Deeds* checks policies and enforces several constraints. As mentioned earlier, a simple and easy-to-use user interface is a critical component of an access-control mechanism for mobile code. We illustrate the operation of the *Deeds* user interface using snapshots. Next, we discuss extensions to *Deeds* and its relationship with JDK security. We conclude with a description of related work and the directions in which we plan to extend this effort.

[1] Especially in programs developed on "Internet time".

[2] Your deeds determine your destiny :)

## 2 Motivating examples

**One-out-of-k:** Consider the situation when you want to allow a program to have the functionality of either a browser or an editor or a shell. A browser can connect to remote sites, create local files, read files that it has created and display them to the user. An editor can create local files, read/modify files that it has created, and interact with the user. It is not allowed to open sockets. A shell can interact with the user and can create sub-processes. It cannot open local files, or connect to remote sites. This can be enforced by a history-based access-control policy that:

- allows a program to connect to a remote site if and only if it has neither opened a file that it has not created, nor modified a file it has created, nor created a sub-process;
- allows a program to open local files for modification if and only if it has created them, and it has neither connected to a remote site nor created a sub-process.
- allows a program to create sub-processes if and only if it has neither connected to a remote site nor opened a local file.

In effect a program gets classified as a browser, an editor or a shell depending on whether it has connected to a remote site, opened local files that it has created for modification, or created a sub-process.

**Keeping out the rogues:** Consider the situation where you want to make sure that a program that you have previously terminated due to inappropriate behavior is not allowed to execute ever again. This can be enforced, to some extent, by a history-based access-control policy that keeps track of previous termination events and the identity of the programs that were terminated. Programs that are on the list of previously terminated programs are not allowed to initiate execution.

**Slowing down the hogs:** Consider the situation where you want to limit the rate at which a program connects to its home site. One might wish to do this, for example, to eliminate a form of denial of service where a program repeatedly connects to its home site without doing anything else. This can be enforced by a history-based access-control policy that keeps track of the timestamp of the last request. It allows only those requests that occur after a threshold period.

**Limiting resource usage:** Consider the situation where you want to limit the total amount of certain resources used by a program (for example disk space, network transfers). This can be enforced by a history-based access-control policy that maps each resource into a common currency of resource usage and decrements a counter every time a resource is used.

## 3 Issues for history-based access-control

To be successful, history-based access-control mechanisms for mobile code need to be efficient, dynamic and easy to use. The need for efficiency arises from the frequency of accesses to resources one might wish to protect. Wallach et al [29] mention that a collection of commonly used Java workloads require roughly 30000 crossings between protection domains per CPU-second of execution. Given this frequency of access, it is imperative that access control checks on individual requests be fast. The need to be dynamic arises from the desire of users to be in control at all times. Since browsers and many mobile programs run for a long time, this implies a need for on-the-fly installation/revocation/modification of security policies. The need for usability arises from the ubiquity of mobile code and from the diversity in user sophistication. Access-control mechanisms that are complex or intrusive are likely to be turned off or bypassed. In this section, we discuss several issues concerning these requirements.

**Compact histories:** Given the frequency of cross-domain access requests, simple logging-based techniques for maintaining event-histories can be extremely expensive – both when an event is logged and when a constraint is checked against the log. Fortunately, the history for many useful history-based policies can be abbreviated as a count or a boolean or a list. For example, consider an access-control policy that allows a program to open local files for reading as long it has not opened a socket and allows it to open a socket as long as it has not opened a local file for reading. The relevant history for this policy can be captured in a pair of booleans – one that records if the program has ever opened a socket and the other that records if it has ever opened a local file.

**Persistence of policies and histories:** A history-based access-control mechanism has to store two kinds of information on stable storage – identity of the access-control policies that are currently active and event-histories that are used by them. Note that, given the frequency of cross-domain accesses, writing to stable storage every time a request occurs is impractical. There is, hence, a tradeoff between efficiency and completeness of event-history in stable storage.

**Dynamic updates:** Integration of mobile code with web browsing creates a social expectation that the user is in control at all times and is able to customize the operation of mobile code and/or the browser. To achieve this, it is important to be able to dynamically install and remove specific policies. For example, if a new policy becomes available, users would like to be able to replace an existing policy without waiting for a new release of the browser or the plug-in that executes the mobile code. In addition, users might wish to reduce or temporarily increase previously granted privileges for programs already in execution. For example, if the currently installed policy allows programs to write to `/tmp` and `/tmp` is filling up, the user might wish to temporarily revoke all write privileges. As might be expected, this feature is likely to see much use only if installing and removing policies is as easy as clicking a button.

**Grouping privileges:** History-based mechanisms can provide extremely fine-grain access-control. Not only is it possible to control accesses to individual objects/resources, it is possible to differentiate between different patterns of accesses. While this allows us to expand the set of programs that can be executed without compromising security or ease of use, there is, however, a potential problem with this flexibility. Requiring users to specify their preferences at this level of detail is likely to be intrusive and therefore ignored or avoided. It is preferable to group acceptable patterns of program behavior and associate them with intuitive names. For example, consider a policy that allows a program to open no sockets, open local files for reading, to create local files and to open a local file for modification only if it has been been created by itself (or a previous instance of itself). Since such a policy can be used to allow execution of editor applets, this group of restrictions can be referred to as the *editor* policy.

**Composition of multiple policies:** History-based access-control policies encode acceptable patterns of program behavior. Different kind of programs might have different behaviors all of which are acceptable to the user. It is, therefore, important to provide automatic composition of multiple policies. An important point to note here is that, by default, the access-control mechanism should be fail-safe [25] – potentially dangerous accesses should be denied unless explicitly granted.

## 4 Deeds: a history-based security manager for Java

In this section we describe the implementation and the user interface of *Deeds*, a history-based access-control mechanism for Java. We first describe the architecture of *Deeds*. Next, we describe its current implementation. Finally, we describe the user interface.

### 4.1 Architecture

Central to the architecture of *Deeds* is the notion of security events. A security event occurs whenever a request is made to a protected resource. Examples of security events include request to open a socket, request to open a file for reading, request to create a file, request to open a file for modification etc. The set of security events is not fixed. In particular, it is not limited to requests for operating system resources. Programmers can associate a security event with any request they wish to keep track of or protect.

Handlers can be associated with security events. Handlers perform two tasks: they maintain an event-history and check whether it satisfies one or more user-specified constraints. If any of the constraints fails, the handler raises a security-related exception. For example, a handler for the security event associated with opening a socket can record whether the program currently being executed has ever opened a socket. Similarly, a handler for the security event associated with opening a file for reading can record whether the program has opened a file for reading.

Multiple handlers can be associated with each event. Handlers maintain separate event-histories. The checks they perform are, in effect, composed using a "consensus voting rule" – that is, one negative vote can veto a decision and at least one positive vote is needed to approve. In this context, a request is permitted to continue if and only if at least one handler is present and none of the handlers raises an exception.

Access-control policies consist of one or more handlers grouped together. The handlers belonging to a single policy maintain a common history and check common constraints. For example, the *editor* policy mentioned in introduction would consist of four handlers:

- a handler for socket-creation that records if a socket was ever created by this program. It raises an exception if a file has been opened by this program (for reading or writing).
- a handler for file-creation that associates a creator with each file created by a downloaded program. It raises no exception and therefore allows local files to be created.
- a handler for open-file-for-read that records if a file was ever opened for reading by this program. It raises an exception if a socket has been created by this program.
- a handler for open-file-for-modification that records if a file was ever opened for writing by this program. It raises an exception if a socket has been created by this program. It also checks if the requesting program is the same as the creator of the file. It raises an exception if the two are different.

*Deeds* allows multiple access-control policies to be simultaneously active. Policies can be installed, removed or modified during execution. A policy is added by associating its constituent handlers to the corresponding events. For example, the *editor* policy would be added by attaching its handlers respectively to the socket-creation event, the file-creation event, the open-file-for-read event and the open-file-for-modification event. Policies can be removed in an analogous manner by detaching the constituents handlers from the associated events.

*Deeds* allows policies to be parameterized. For example, a policy that controls file creation can be parameterized by the directory within which file creation is allowed. Policies that are already installed can be modified by changing their parameters. This allows users to make on-the-fly changes to the environment within which mobile code executes.

*Deeds* provides a fail-safe default [25] for every security event. Unless overridden, the default handler for an event disallows all requests associated with that event from downloaded programs; requests from programs loaded from local disk (via the `CLASSPATH`) are allowed to proceed. The default handler can only be overridden by explicit user request – either by a dialog box or by a profile file containing a list of user preferences.

## 4.2 Implementation

In this subsection, we describe our implementation of *Deeds* for Java. We focus on implementation of events, event-histories, policies (including conventions for writing them), policy management, and program identity.

### 4.2.1 Events and histories

**Events:** Given the high frequency of security events (requests for protected resources) in mobile code, efficiency was our primary goal for the implementation of events and histories. There are two concerns: (1) a large number of handlers could be associated with individual events, and (2) the number of events is not fixed.

We considered three alternative implementations for events. First, we could use a general-purpose mechanism (similar to the mechanism used in Java Beans [6] and X [26]) to register events and handlers. The advantage of this approach is that it uses common code to manage all events and their associated handlers; the disadvantage is that all handlers must have the same type-signature which usually implies that the parameters need to be packed by the event manager and repeatedly unpacked by individual handlers.

Second, we could dynamically modify the bytecode of the downloaded program to insert/delete calls to handlers at the points where the events are generated (in a manner similar to that used in [4, 16]). To allow a user to modify an executing policy would require us to update the bytecode of running programs. This is a complexity that we did not wish to undertake.

Finally, we could require that the handlers for each event be managed by a different event manager. This approach allows us to avoid the packing and unpacking of parameters as each event manager is aware of the parameters corresponding to its event. The disadvantage of this scheme is that a separate event manager has to be written for each event. However, event managers are highly stylized and can be automatically generated given a description of the event (see figure 1 for an example).

We selected the third approach for implementing security events in *Deeds*. Combined with automatic generation of event managers, this allows us to balance the needs of efficiency, implementation simplicity and ease of programming. Examples of *Deeds* security events include `checkRead()` and `checkConnect()`.[3]

**History:** The common technique for maintaining an event-history is to logging. In this context, logging has both advantages and disadvantages. The advantages of logging are: (1) a policy can

[3]For the readers who are familiar with the Java `SecurityManager`, we would like to point out that all the `check*()` methods can be considered to be security events.

```
public class checkReadMananger implements EventManager {
  private static HandlerVec hvec = new HandlerVec();

  public static void checkRead(String fileName) throws PolicyException {
    int i; Interface_checkRead p;

    for (i=0; i < hvec.policies.size(); i++) {
      p = (Interface_checkRead)(hvec.policies.elementAt(i));
      p.checkRead(fileName);
    }
  }
}

public interface Interface_checkRead {
  public void checkRead(String fileName) throws PolicyException;
}
```

Figure 1: Example of an event manager class. Managers for other events would share the same structure but would replace `checkRead` by name of the particular event. The interface `Interface_checkRead` is used to allow type-checking of the classes used to implement access-control policies. Some administrative details have been left out of the example; these details are common to all event managers.

check for events that occurred before it was installed; (2) a general-purpose mechanism can be used and a common log can be maintained. The disadvantages are: (1) given the expected frequency of events the log will grow rapidly, particularly since it is unclear when such a log should be truncated; (2) searching a log can be slow especially for a large log.[4]

Given the concern about the size of the log, we have chosen to avoid logging as a general technique for maintaining event-histories. Instead, we have left the decision about how to store histories to individual policies.[5] All policies that we considered were able to effectively summarize event-histories using counters, booleans or lists. This includes, for example, all the event-histories in the *editor* policy.

#### 4.2.2 Access-control policies

A *Deeds* access-control policy consists of data structures to maintain event-histories, handlers for each of the events that are mediated by the policy, and auxiliary variables and operations to facilitate management of multiple policies. Concretely, an access-control policy is implemented as a Java class that extends the `AccessPolicy` class shown in Figure 2. Handlers are implemented as methods of this class that update the history and check constraints. For example, a handler for the open-file-for-reading event could:

[4]Note that this can be alleviated to some extend by incremental processing of the log.

[5]Note that this decision has the potential disadvantage that if policies do desire/need to log events, the lack of a common logging mechanism can result in the maintenance of duplicate logs. This can be fixed by using a selective logging mechanism that logs an event only if requested to do so by one or more handlers associated with the event.

```
abstract synchronized public class AccessPolicy {
 public String name         ; // name of policy instance
 public Vector parameters   ; // policy parameters
 public Vector targetEvents ; //
 public String srcFileName  ; // source file location
 // these functions have to be provided by every policy
 public abstract String documentation();
 public abstract void   saveHistoryToDisk();
 public abstract void   restoreHistoryFromDisk();

 public Policy(String name) {
   ...
 }
}
```

Figure 2: Skeleton of the AccessPolicy class. The `synchronized` keyword ensures that at most one handler is updating the event-history at any given time.

1. check if a socket has yet been created by the program. If so, raise the `PolicyException`.
2. set a boolean to indicate that a file has been opened for reading.

*Deeds* policies are expected to follow a set of conventions. These conventions are checked either at compile-time by the Java compiler or at runtime by the *Deeds* policy manager. The conventions are:

- Handler methods must include a `throws PolicyException` clause and must have the same name and type signature as the security event that they are intended for.
- A handler method must have the same number of variants as the security event that it is intended for. For example, a checkRead event (corresponding to the checkRead method in the Java `SecurityManager`) has three variants – `checkRead(FileDescriptor)`, `checkRead(String)`, and `checkRead(String, Object)`. Handlers for this event must have three variants with the same signatures.
- Parameters of the policy must be explicitly identified. Each parameter must have a default value and a documentation string.
- The vector `targetEvents` specifies which events the handlers in this policy are to be attached to. The specification is in the form of a regular expression which is matched against fully-qualified names. For example, the target events for a checkRead handler could be specified as `''FileIO.checkRead''` or `''*.checkRead''`. The former expression specifies only the checkRead event defined in the `FileIO` package whereas the latter specifies all checkRead events irrespective of the package they have been defined in. This specification is needed as Java's hierarchical namespace allows multiple methods with the same name to exist in different regions of the namespace. Since a security event is implemented by a method in an EventManager class and since every package can have its own security events, the possibility of

name clashes is real. For example, a library to perform file I/O, and a library to interact with a database could both wish to create a `checkRead` event. Since packages are independently developed, extensible systems, such as *Deeds*, cannot assume uniqueness of event names.

- Each policy must be accompanied by its source code and the name of the file containing the source code should be available as a member of the class implementing the policy. We believe that availability of source code of a policy is important to instill confidence in its operation and its documentation.

#### 4.2.3 Policy manager

The *Deeds* policy manager makes extensive use of the Java reflection mechanism [15]. This mechanism allows Java code to inspect and browse the structure of other classes. The policy manager uses reflection to: (1) identify methods that are to be used as handlers (they are declared `public void` and throw `PolicyException`); (2) identify parameters and their types; (3) initialize and update parameters; and (4) extract specification of the events that the handlers are to be attached to. In addition, it performs several administrative checks such as ensuring that all policy instances have unique names.

The policy manager is also responsible for ensuring that policies are persistent. It achieves this by storing the parameters for each policy instance on stable storage and using them to re-install the policy when the interpreter/browser is restarted.

Note that all the handlers for a policy should be attached/detached atomically. Not doing so can result in breach of security. A similar situation occurs when updating the parameters for a previously installed policy. In this case, all parameters should be atomically updated. To achieve atomic updates, *Deeds* suspends all executing threads before it attaches/detaches/modifies policies and resumes them after the update has been completed.

#### 4.2.4 Program identification

To maintain a long-term history associated with individual programs, it is necessary to be able to identify these programs. For example, in the *editor* policy, programs are allowed to modify local files that they have created. Program identification in a mobile code environment introduces new complications as the mapping between names used to access programs (ie. URLs) is not under the control of the user and can be changed at any time.

We considered two alternatives for identifying programs – URLs and URLs augmented by a digital signature [24] of the author or the organization providing the program. A URL-based identification is susceptible to spoofing attacks [7]. Another potential problem with a URL-based scheme is that if the same program is accessible from multiple URLs (eg. from mirror sites), it gets labeled as two different programs. Augmenting the URL with the digital signature eliminates the possibility of spoofing as long as the provider's key is safe. Note that signatures are not used to grant and/or check authorizations (as in ActiveX [23]) but are merely used to ensure continuity of identities across browsing sessions. This scheme does not care who the signature belongs to as long as the signature is the same every time the program is accessed.

It is possible to (almost) eliminate the problem of multiple names for a program using a digest function such as MD5 [14]. However, when/if a program is updated without changing the URL, an MD5 based scheme would label it as two different programs and may not allow it, for example, to open files created by the previous version. The current version of *Deeds* uses URLs as program identifiers. We plan to investigate the use of a scheme that combines URLs and digital signatures in the near future.

### 4.3 User interface

Our primary goal in the design and implementation of the *Deeds* user interface was that it should be non-intrusive. That is, it should query the user only if absolutely necessary. As mentioned in the introduction, interfaces that are complex or intrusive are likely to be turned off or bypassed. Accordingly, the *Deeds* user interface comes up only on user request and is used for infrequent operations such as browsing/adding/deleting/modifying policies. In this section, we describe the functionality of the user interface and present snapshots.

**Browsing/viewing/instantiating policies:** *Deeds* allows users to browse the set of available policies, to view documentation and source code for these policies and to create instances of individual policies. Note that every policy is required to have documentation (via the `documentation()` method) and access to its own source code (via the `srcFileName` member). In addition, every parameter has associated documentation which can be viewed. To instantiate a parameterized policy, users need to specify values for all the parameters of the policy. Note that every parameter has a default which is displayed. *Deeds* uses the Java reflection mechanism to figure out the type of the parameters for display and parsing purposes. These functions of the user interface are illustrated in figures 3 and 4. In figure 3, a policy is selected by clicking on its name and operations are selected using the buttons. Browsing and instantiation of individual policies is illustrated in figure 4.

**Adding/deleting/modifying policies:** *Deeds* allows users to install instantiated policies as well as remove or modify currently active policies. The list of instantiated and active policies can be browsed. For an illustration, see figure 3. A policy instance can be installed (or *activated*) using the `Add` button, and an active policy can be removed using the `Remove` button. The modifying operation uses the same interface as the instantiation operation (see figure 4).

**Checkpointing event-histories:** *Deeds* allows user to checkpoint the current state of the event-histories for all policies. This is a compromise between writing to stable storage every time an event-history is modified and losing event-histories in case of involuntary termination of a browser/interpreter. In figure 3, checkpointing is initiated using the `CheckPoint` button.

**Browsing/setting default handlers:** *Deeds* provides a fail-safe default for every security event. Unless overridden, the default handler for an event disallows all requests associated with that event from downloaded programs; requests from programs loaded from local disk (via the `CLASSPATH`) are allowed to proceed. The default handler can only be overridden by explicit user request – either by a dialog box or by a profile file containing a list of user preferences. *Deeds* allows users to browse and set default handlers for all security events. In addition to the `AllowNoRequests` handler, users can also chose the `AllowAllRequests` handler as the default handler for an event (see figure 5).

## 5 Discussion

**Pre-classified program behaviors:** The *one-out-of-k* policy described in section 2 classifies program behaviors in an on-line manner. A program gets classified as a browser, an editor or a shell depending on whether it has connected to a remote site, has opened local files for modification, or has created a sub-process. To be able to do this for a wide variety of program behaviors, the

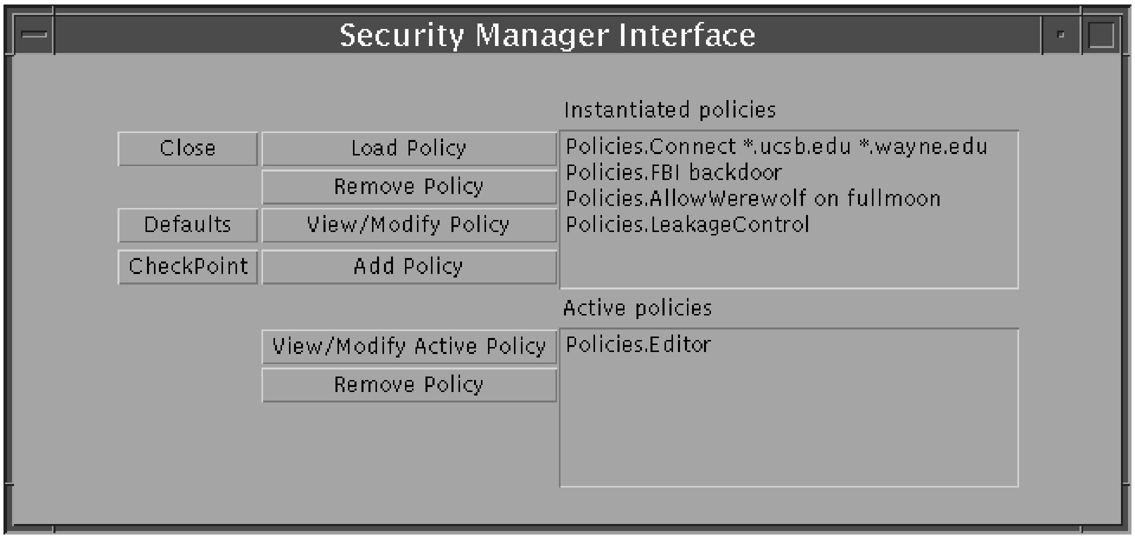


Figure 3: Graphical interface to the SecurityManager

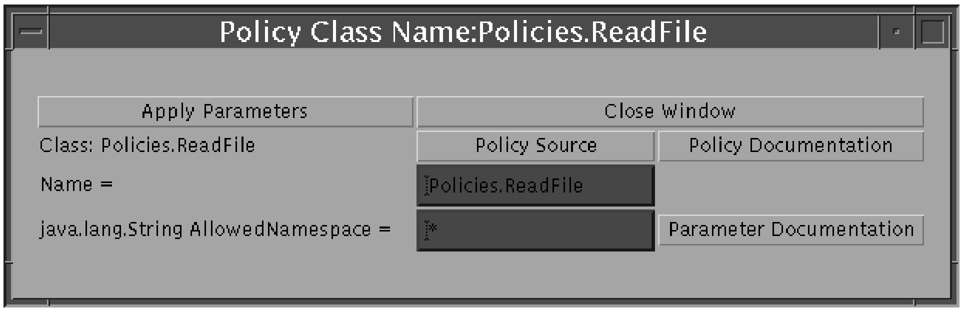


Figure 4: Policy to allow opening a file

policy that does the classification and subsequent management of privileges has to contain code to handle all these behaviors. An alternative scheme would be to allow program-providers to label their programs with pre-classified behavior patterns and to allow the users to specify which behaviors they would like to permit. The policies governing individual behaviors could be added/deleted as need be. While this scheme would require agreement on the labeling scheme, it is no more complex than the MIME-types-based scheme that is already in use for displaying/processing different data formats. This scheme is similar to *program-ACLs* and similar defenses proposed for trojan-horse attacks [19, 30].

**Joint-authorization:** Commercial applications, such as contracts and purchase orders, may require multiple authorizations since the organization may wish to reduce the risk of malfesance by dispersing trust over several individuals. History-based access-control policies can be used to implement joint-authorization [28] or *k-out-of-n* authorizations [3]. For example, a policy may require that three out of five known individuals must make the same request within the last $T$ units of time for the request to be granted; else the request is denied. In this case, the history consists of the requests that have been made in the last $T$ units of time.

**Relationship with JDK security:** Currently, *Deeds* has been implemented in addition to the

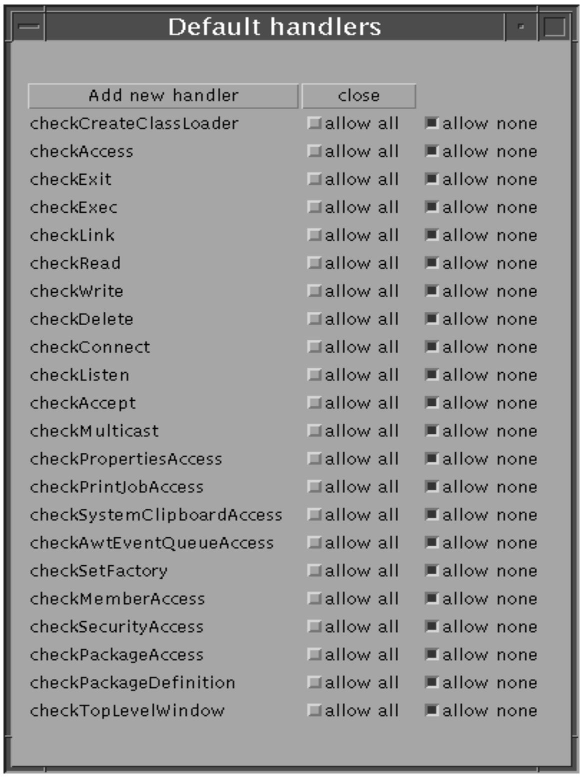


Figure 5: Interface for browsing/setting default handlers

existing JDK security mechanism. We envisage/hope that a later version of *Deeds* could be integrated with the JDK security mechanism. We see two possible interactions. First, the JDK `SecurityManager` class already defines a set of security events in form of the `check*()` methods. An integrated system would use these as the base set of events. Second, the `java.security` package provides support for keeping track of identities of various principals. *Deeds* can use this support to implement better program identification (as discussed in section 4).

# 6 Related work

The primary problem for access-control mechanisms for mobile code is to be able to differentiate between different programs executing on behalf of the same user and to provide them with different privileges based on their expected behavior and/or potential to cause damage. A similar problem occurs in the context of trojan-horse programs and viruses.[6] Several researchers have developed mechanisms to limit the privileges of individual programs based on their expected behavior [10, 19, 20, 21, 22, 30]. Karger [19] uses information about file extensions and behavior of individual programs to determine the set of files that a program is allowed to access (eg. a compiler invoked on `x.c` is only allowed to create `x.{o,u,out}`). Lai [22] replaces the inference mechanism by an explicit list of files accessible by a program. Wichers et al [30] associate *program-ACLs* with each file thereby limiting the set of programs that can access each file. King [20] uses a regular-expression-based language to specify the set of objects each operation can access. Ko et al[21] use a language based

[6] In fact, mobile code and trojan-horses/viruses are both examples of guest code; the difference, of course, being that the former arrives by invitation whereas the latter is an uninvited guest.

on predicate logic and regular expressions to specify the security-relevant behavior of privileged programs and propose to use this specification for intrusion detection. All these approaches assume that the set of programs to be run are fixed and their behaviors are known. The mobile code environment is different as the set of programs that will execute is inherently unknown. History-based access-control is able to classify programs in an on-line manner and to thereafter execute them within an environment with appropriate privileges. For example, the *one-out-of-k* policy classifies downloaded programs into one of three classes: browsers, editors and shells.

Apart from history-based nature, the other important feature of *Deeds* is its capability to install and compose multiple user-specified policies. Several researchers have proposed languages to allow users to specify access-control policies and frameworks to compose these policies [3, 13, 17, 18]. Three of them [13, 17, 18], propose logic-based declarative languages and use inference mechanisms of various sorts to compose policies. Blaze et al [3] propose a language that contains both assertions and procedural filters and use a mechanism similar of that used in *Deeds* to implement composition. Access-control policies for *Deeds* are entirely procedural. Furthermore, they can be updated while the programs whose accesses are being controlled are still executing.

The event model used in *Deeds* is similar to that used in the SPIN extensible operating system [2]. An interesting feature of SPIN is the use of dynamic compilation to improve the performance of event dispatching [4]. If the performance of event dispatching becomes a problem for *Deeds* (eg. if individual events have a large number of handlers) we can use a similar technique.

## 7 Current status and future directions

*Deeds* is currently operational. We are in the process of identifying a variety of useful patterns of behaviors and evaluating the performance and usability of *Deeds* in the context of these behaviors. *Deeds* can currently be used only for stand-alone Java programs. We hope to integrate it in a suitable browser.

In the near term, we plan to explore the possibility of using program labels to indicate pre-classified behaviors (see section 5 for a description) and automatic loading/unloading of access-control policies to support this. We also plan to extend the program identification mechanism by using digital signatures.

In the longer term, we plan to explore just-in-time binary rewriting to insert event generation and dispatching code into downloaded programs. This would allow users to create new kinds of events as and when they desire. Currently, new kinds of events are created only by trusted libraries. In addition to the enhanced flexibility, we expect that this would provide improved performance.

## References


[1] D. Andresen, T. Yang, D. Watson, and A. Poulakidas. Dynamic processor scheduling with client resources for fast multi-resolution www image browsing. In *Proceedings of the 11th International Parallel Processing Symposium*, 1997.

[2] S. Savage B. Bershad, P. Pardyak, E. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the spin operating system. In *Proc of the 15th ACM Symposium on Operating System Principles*, pages 267–84, 1995.

[3] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc of the 17th Symposium on Security and Privacy*, pages 164–73, 1996.

[4] C. Chambers, S. Eggers, J. Auslander, M. Philipose, M. Mock, and P. Pardyak. Automatic dynamic compilation support for event dispatching in extensible systems. In *Workshop on Compiler Support for Systems Software*, 1996.

[5] B. Christiansen, P. Cappello, M. Ionescu, M. Neary, K. Schauser, and D. Wu. Javelin: Internet-based parallel computing using Java. In *Proceedings of the 1997 ACM Workshop on Java for Science and Engineering Computation*, 1997.

[6] R. Englander. *Developing Java Beans.* O'Reilly & Associates, 1997.

[7] E. Felten, D. Balfanz, D. Dean, and D. Wallach. Web spoofing: An internet con game. Technical Report 540-96, Department of Computer Science, Princeton University, 1996.

[8] D. Flanagan. *JavaScript: The Definitive Guide.* O'Reilly & Associates, 2nd edition, Jan 1997.

[9] J. Fritzinger and M. Mueller. Java security. Technical report, Sun Microsystems, Inc, 1996.

[10] T. Gamble. Implementing execution controls inUnix. In *Proceedings of the 7th System Administration Conference*, pages 237–42, 1993.

[11] L. Gong. New security architectural directions for java. In *Proceedings of IEEE COMPCON'97*, 1997.

[12] J. Gosling, B. Joy, and G. Steel. *The Java Language Specification.* Addison-Wesley, 1996.

[13] C. Gunter and T. Jim. Design of an application-level security infrastructure. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997.

[14] J. Hollingsworth and E. Miller. Using content-derived names for caching and software distribution. Technical Report CS-TR-3671, Department of Computer Science, University of Maryland, 1996.

[15] C. Horstmann and G. Cornell. *Core Java 1.1*, volume I - Fundamentals. Sun Microsystems Press, third edition, 1997.

[16] B. Miller J. Hollingsworth and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *SHPCC*, 1994.

[17] T. Jaeger, A. Prakash, and A. Rubin. Building systems that flexibly control downloaded executable context. In *Proc of the 6th Usenix Security Symposium*, 1996.

[18] S. Jajodia, P. Samarati, V. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 474–85, 1997.

[19] P. Karger. Limiting the damage potential of the discretionary trojan horse. In *Proceedings of the 1987 IEEE Syposium on Research in Security and Privacy*, 1987.

[20] M. King. Identifying and controlling undesirable program behaviors. In *Proceedings of the 14th National Computer Security Conference*, 1992.

[21] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings. 10th Annual Computer Security Applications Conference*, pages 134–44, 1994.

[22] N. Lai and T. Gray. Strengthening discretionary access controls to inhibit trojan horses and computer viruses. In *Proceedings of the 1988 USENIX Summer Symposium*, 1988.

[23] Microsoft Corporation. *Proposal for Authenticating Code Via the Internet*, Apr 1996. *http://www.microsoft.com/intdev/security/authcode.*

[24] B. Pfitzmann. *Digital Signature Schemes : General Framework and Fail-Stop Signatures.* Springer Verlag, 1996.

[25] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep 1975.

[26] R. Scheifler and J. Gettys. *X Window System : The Complete Reference to Xlib, X Protocol, Icccm, Xlfd.* Butterworth-Heinemann, 1992.

[27] B. Schmitt. *Shockwave Studio: Designing Multimedia for the Web.* O'Reilly & Associates, Mar 1997.

[28] V. Varadharajan and P. Allen. Joint actions based authorization schemes. *Operating Systems Review*, 30(3):32–45, 1996.

[29] D. Wallach, D. Balfanz, D. Dean, and E. Felten. Extensible security architecture for java. In *SOSP 16*, 1997.

[30] D. Wichers, D. Cook, R. Olsson, J. Crossley, P. Kerchen, K. Levitt, and R. Lo. PACL's: an access control list approach to anti-viral security. In *USENIX Workshop Proceedings. UNIX SECURITY II*, pages 71–82, 1990.

[31] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using smart clients to build scalable services. In *Proceedings of USENIX'97*, 1997.

[32] J. Zukowski. *Java AWT Reference.* O'Reilly & Associates, 1997.