The following paper was originally published in the
Proceedings of the 7th USENIX Security Symposium
San Antonio, Texas, January 26-29, 1998

Expanding and Extending the Security Features of Java

Nimisha V. Mehta
The Open Group
Karen R. Sollins
MIT Laboratory for Computer Science

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

Expanding and Extending the Security Features of Java

Nimisha V. Mehta
The OpenGroup[1]
Cambridge, MA 02139
Karen R. Sollins
MIT Laboratory for Computer Science
Cambridge, MA 02139

Abstract

The popularity of the web has had several significant impacts, two of note here: (1) increasing sophistication of web pages, including more regular use of Java and other mobile code, and (2) decreasing average level of sophistication as the user population becomes more broad-based. Coupling these with the increased security threats posed by importing more and more mobile code has caused an emphasis on the security of executing Java applets. This paper considers two significant enhancements that will provide users with both a richer and more effective security model. The two enhancements are the provision of flexible and configurable security constraints and the ability to confine use of certain storage channels, as defined by Lampson[11], to within those constraints. We are particularly concerned with applets using files as communications channels contrary to desired security constraints. We present the mechanisms, a discussion of the implementation, and a summary of some performance comparisons. It is important to note that the ideas presented here are more generally applicable than only to the particular storage channels discussed or even only to Java.

1 Introduction

The April, 1997 edition of the *Graphics, Visualization and Users Study*[18] reports that the number of web based document authors using Java is increasing. In addition, the authors report that the respondents' belief that they understand and trust Java's security is increasing. This is a self-selecting population, listing their occupations in the fields of **computers**, **education**, **management**, **professional**, and **other**. This last category comprises only about 14% of the respondents, so the great majority of respondents are probably fairly computer literate. One must assume that a much larger percentage of the population as a whole would fall into the **other** category. We must also assume that as the "wired" population increases, it also becomes more naive on average. The reason is that more and more of that "wired" population will come from that part of the population that has less education about computers. In particular, one of the most difficult set of issues to understand and use correctly is those surrounding mobile code and security. Thus, we as technologists find ourselves with a dilemma. On one hand we wish to increase functionality, ease of use, flexibility and apparent simplicity to the user. On the other hand, to do this the supporting mechanisms must become increasingly complex and sophisticated.

Mobile code is not new. We have been moving code in files for years. Interpreted languages such as Lisp and Postscript have been particularly prone to mobility. What is changing is how and when code moves and who has what knowledge of its execution. The Web has made the most significant difference here. The naive user moves from one web page to another, perhaps considering them to be rather static resources. Meanwhile increasing numbers of authors are including Java applets or other forms of mobile code in their pages, to be executed on the page reader's machine. Although the majority of these are not intended to be malicious, some may be, and more may simply be prone to errors, leading to potential dangers for the unknowing user.

In response, Sun Microsystems, the creator of Java, the Java Virtual Machine, the Java Development

[1] This work was performed while Mehta was at the MIT Laboratory for Computer Science.

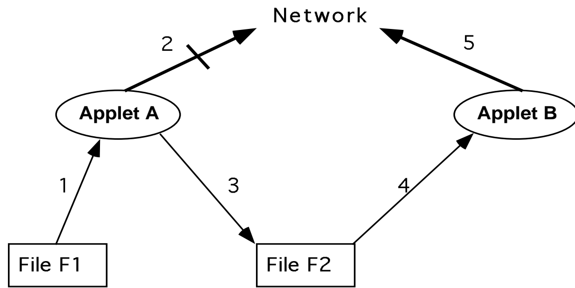

Figure 1: Conditional access and a covert channel: Applet A is prevented from accessing the net if it has accessed file F1. Later Applet B should be prevented from accessing the net after accessing file F2, which was written by A after reading F1.

Kit, etc. has made a concerted effort to close security loopholes. Wallach et al.[23] report on three efforts in the Java environment, but outside Sun Microsystems itself. The other providers of mobile code are addressing the same problems. As a proof of concept we have considered storage channel issues exclusively in Java, although the ideas presented here should be easily portable to other environments.

We began this work with a particular example problem in mind. See Figure 1. Consider applet A that could do useful work for you the reader. In order to do this work it needs to read file F1 and create new files in your home directory, so you would like to be able to give it permission to do that (1). Furthermore, you consider file F1 to be private, so you would like to insure that once A has read F1 it can no longer have access to the network (2). Now suppose that in addition to its job it also creates file F2 (3). Unless you read the code, which is beyond the means of most naive users, you may not know about this additional file. Now at some later time another instantiation of applet A or a second applet B is executed. If the security constraints are not set up properly, this file may be a simple storage channel for exposing some of the contents of F1 to the outside world (4, 5). In the work presented in later sections, we disallow behavior of the sort represented in step 4, in order to close that kind of channel. In Section 6 we discuss how one would allow step 4, but prevent step 5.

There are two significant features to this problem that will be considered in this work: the provision of a system for creating flexible and configurable policies for mobile code, and extending the scope of confinement as described by Lampson[11] to include time-delayed storage channels.

There is an increasing need for a policy language to improve flexibility and configurability of security policies. Wallach et al. describe three schemes for allowing applet access beyond the simple sandbox model. In all three, constraints need to be expressed to describe permissions within the scopes of the schemes. In the capability scheme the propagation of capabilities must be restricted somehow, while in the extended stack introspection approach an access matrix must be generated, and in type hiding there must be a specification of how the type of each object is hidden for each principal. It is worth noting that one can separate the issue of specification of constraints from evaluation of them. In Section 2, other approaches to languages are discussed.

In this work, we have taken the approach in the constraint language of addressing time-delayed storage channels, denial of services, and Trojan horses by means of history-based policies, object-based policies, subject-based policies and floating category labels. The Java Development Kit 1.2 (JDK1.2)[5] provides one such language although not quite adequate for our needs. In parallel with that work, we built ourselves a simple constraint language to meet our immediate needs. Clearly ideas from one can be merged into the other in the long run.

When Lampson described the *confinement problem*, he described three sorts of channels, *storage*, *legitimate*, and *covert*. Moskowitz and Kang[15] address only covert channels and divide them into *storage*, *timing*, and *mixed* channels. What is happening here is that there are two orthogonal axes along which to describe the channels that are at the crux of the confinement problem. These axes have to do with original intention of use and mode of communication. Lampson was interested in distinguishing between channels that were intended to be used as communication channels, albeit perhaps contrary to security policy, and those channels that were not intended to be communication channels. Moskowitz and Kang were moving along the other axis in distinguishing between channels "where the output alphabet consists of different responses all taking the same time to be transmitted" from one in which the alphabet consists of "different time values".

The second problem we are addressing here falls into

Lampson's definition of a *storage* channel, with a twist. The particular problem is that timing delays between input into the storage channel and output may hide the existence of a confinement problem. As a proof of concept, this work focuses on files within this context. We have created two distinct logging facilities one that tracks principals' (applets') accesses and one that tracks applets as file owners. A central component of our approach is the need for a constraint language and set of constraints that are evaluated in order to provide adequate security.

This paper presents concepts. We have designed and implemented them, for demonstration purposes, but the contribution of this work is in the concepts, not the realization. The paper will proceed as follows. Section 2 provides background and presents some of the most closely related work. That is followed by a description of the two logging facilities needed to address the problem with respect to the file system. Section 4 describes our constraint language, followed by a section on implementation issues and some preliminary performance numbers. The ideas in the language could valuably be incorporated into an existing language, such as that described by Gong[5]. It should also be noted that since this effort was generally done as a proof of concept, it was not tuned for performance, although Section 5.5 presents some preliminary performance numbers. The paper concludes with a discussion of further issues related to this work. For a more complete description of this effort see Mehta's thesis[13].

2 Related Work

Any work in the security area is based on an enormous background of previous work. One cannot give complete credit to all the preceding work. Since this work concentrates on extending the security model for Java applets, we will review the security models of the current major Web browsers. We will not address the literature on logging, although there has been a great deal in the areas of systems and databases. We are using only simple, straightforward logging techniques here. In addition, we will briefly examine prior work on authorization languages, followed by comparisons of our work with other secure systems for mobile code. This will include a brief discussion of the applicability of tools that statically analyze mobile code as an extension of authenticating the source of an applet. We will conclude this section with a review of the current situation with respect to Java itself, focusing on the specification of constraints.

2.1 Current Web Browsers

The three current browsers, Internet Explorer 4.0[14] from Microsoft, Netscape's Communicator 4.0[17], and JavaSoft's HotJava[8, 9] provide a variety of mechanisms to authenticate and authorize Java applets. Although none is currently adequate for our needs, they can be extended to satisfy our requirements as specified in this paper.

Netscape Communicator 4.0's capabilities-based feature allows for an extension of an applet's execution space beyond the sandbox. Applets that want permissions beyond the normal sandbox, must notify the browser of the capabilities they require. Unsigned applets are confined within the limits of the sandbox. When a signed applet arrives on the user's system, the user is notified of the identity of the applet's signer and of the capabilities that applet requests. Adhering to the principle of least privilege, the user can then give permission for only that capability. The disadvantage of this design is that the user must give permission for each applet independently. We are concerned that although initially users may consider the permissions seriously, it will not be long before they stop paying attention, and the utility of the authorization will be nullified. Furthermore, since authorizations are given dynamically, this does not allow for the creation of a coherent policy.

On the other hand, Microsoft's Internet Explorer 4.0 (IE) allows users to set permissions statically based on configurable "security zones" from which applets arrive. IE allows the user to set "security levels", permitting extensions beyond the sandbox for certain sets of applets. A **High** level would not allow the applet to go beyond the sandbox, a **Medium** level would allow it but only after warning the user, and a **Low** level would let the applet wreak havoc. For example, applets from the "Restricted sites zone" would be assigned a **High** security level, while those from the "Trusted sites zone" would be assigned to a lower level. Extending the capabilities approach in Communicator 4.0, IE also supports capability signing where the requested capabilities are listed within the applet's digital signa-

ture rather than within it's code. Nonetheless, IE is still limited in the configurablity and flexibility of specifying one's policy to preserve confinement and restrict resource consumption.

JavaSoft's HotJava 1.1 supports users in configuring their policies and constraints. As with the other browsers this is based on the signature of an applet, and is then configured by a collection of choices; the user is provided with a checklist of choices. The advantage of this scheme is that the user specifies the constraints only once and is not required to give explicit permission for each execution of each applet. The disadvantage is that the user still cannot specify conditional rules, as is needed to address our problem.

2.2 Authorization Languages

Other research[1, 7, 22] in designing generic authorization languages for secure systems has been done for both military and commercial use, but has not yet been applied to specifying policies for mobile code. This includes work on specifying Separation of Duty policies as done by Sandhu[19], and work on Adage[20] by The Open Group.

Sandhu's efforts on designing control expressions for creating Separation of Duty policies includes a mechanism for maintaining the history of transient and persistent objects using a simple syntax. These policies limit the transactions that can be applied to a particular object based on that object's history. It can be extended to write policies such as: "An applet cannot access a file written by another applet." However, separation of duty policies do not allow one to create a policy which is dependent on multiple objects. For example, it cannot specify our Chinese Wall policy that depends on two distinct objects (networks and files): "an applet cannot connect to the network after reading a protected file."

The Open Group's work on Adage, An Architecture for Distributed Authorization, includes a general-purpose authorization language that is quite flexible and user-friendly. It has an extensive grouping mechanism to categorize subjects, objects, and transactions. It also allows one to create Separation of Duty policies and Chinese Wall policies easily. However, although it maintains a history of transactions, it currently cannot specify policies to limit resource usage. Languages such as these can be applied for creating mobile code policies if they fulfill the requirements specified in this paper.

2.3 Other Secure Systems for Mobile Code

Other projects have attempted to create a secure system with a configurable policy language for mobile code. These include INRIA's SIRAC project[6] on an IDL-based protection scheme for mobile agents, Goldberg et al.'s Janus system[3] for confining untrusted helper applications, and IBM's Aglets[10]. A common goal among all of them is to create a secure system for mobile code that makes use of its features: the collaborative nature of mobile agents or the usefulness of helper applications. However, their common weakness is their limited policy languages.

In SIRAC's scheme for protecting mobile Java agents, the agent's protection policy is defined in an extended Interface Definition Language. This work focuses on protecting access to Java objects between mobile agents by exchanging capabilities between mutually suspicious agents. However, no mention is made of how the language or the system can protect the host's system resources from the agents.

Goldberg et al.'s efforts on confining helper applications involves monitoring and restricting system calls from untrusted applications. Although their approach is language-independent, it is specific to the Solaris operating system. They allow users to specify their permissions ("allow" or "deny") on system calls in a configuration file. However, because the language is quite simplistic, it cannot express policies in terms of resource consumption or histories.

The security model in IBM's Aglets includes an authorization language that allows the agent and its host to specify their policies. Their policy language is rich in that it includes mechanisms to specify the privileges of groups and labelled objects including ways to limit resource usage. However, no further constraints based on the applet's history can be made.

On a different note, others have done research on analyzing the remote code prior to its execution in order to distinguish malicious code from benign programs statically. We have seen this, for example, in

the work on *proof-carrying code* (PCC)[16] and on a *malicious code filter* (MCF)[12].

In Necula's paper on PCC the code carries with it an encoding of the fact that it complies with certain invariants or requirements. This constrains the code in meeting various requirements. The sorts of constraints about which Necula is concerned reflect its internal behavior which would be independent of the location at which the code will be executed. In our case, the criteria will potentially be different at each site, making it impossible to provide any proof of meeting useful criteria at its source. If we could offload some of the verification at the source that would be a great benefit, but it is not clear how to do that.

MCF makes use of program slicing and tell-tale signs of system calls to statically test the behavior of mobile code for certain malicious program properties. Unlike PCC, the detecting of tell-tale signs on the client end does not require the programmer to provide a formal specification of the code. One could imagine extending MCF with a policy language to allow a variant degree of code filtering for different applets. Nonetheless, even if these static approaches were extended to be more configurable, they cannot make much use of object-based or history-based policies since such policies can only be evaluated during run-time.

2.4 Security Model in JDK 1.2

Because we are working in the Java context, we must consider the current Java security architecture in the Java Development Kit 1.2 (JDK1.2)[4, 5]. For the purpose of brevity we will not review the basic Java security features of the JDK here, but assume that the reader knows or can learn those easily. We also assume for the purposes of this work, that the Java architecture works correctly, although due to its complexity we realize that the community will continue to find problems that will be addressed by JavaSoft and others.

The goals or objectives were extended in the JDK1.2 to include a simpler policy configuration, a more easily extensible access control structure, and an extension of the security checks to include all levels of Java programs, not just applets. The first of these involves the definition and use of a simple configuration language for statement of policy constraints. The second requires the addition of a **Check** method to the Security Manager in order to support the automatic handling of typed permissions. Finally, the third objective is met by allowing the same sorts of security checking for local code as for mobile code, providing such functions as verification of certification of the code, etc., rather than simply letting local code run completely trusted.

From our perspective we did not want to modify the Security Manager, so having JavaSoft provide the **Check** method would simplify our task. The specification language provides the ability to state only static non-conditional rules. As such its semantics are simpler than ours. There is also no model of past behavior. The syntax is clearly somewhat different. For simplicity of implementation we chose a simple S-expression type language. In JDK1.2, Gong has chosen a syntax that is much more in line with Java's own syntax and allows for the declaration of permission classes. As we will discuss further below, one can easily extend the Java rule specification language with features to support our ideas. We envision a merging of our and JavaSoft's constraint languages into one that embodies the features of both.

Gong highlights in both his papers that one perspective on the model that the new Java security architecture provides is that of security domains. Each is defined by the scope of accesses permitted to a principal. We are doing the same. In carrying that description further, one can describe the problem of storage channels as follows. First, some of those security domains may overlap. These enable the potential storage channels. Second, the security policies in the overlapped regions may be fuzzy, permitting the potential communications channels more invisibility, further enabling the use of them. Our initial approach is to remove the existence of those overlaps. As discussed in Section 6, with a further extension one could allow the overlap, but clarify the policies to be enforced in them and with respect to the non-overlapped sections of those domains. We did not pursue this view further with respect to this piece of work, but it helps to clarify the work on a more architectural level.

The remainder of this paper presents the work that was actually done to address the problem described above. In this context we were addressing two problems simultaneously, that of conditional constraints and that of storage channels, especially those channels which span time by taking advantage of per-

sistent storage in the form of files. In order to do this we found we needed two forms of logging, one to log applets' accesses to files and the other to log applet "ownership" of files. The access log keeps track of activity that is used in the evaluation of conditional rules, while the ownership log tracks files as potential storage channels. The use, realization, and management of these logs is addressed first. We can then describe the language used to express constraints, and finally implementation details and issues, as well as performance. The paper concludes with a discussion of possible extensions to and further thoughts on this work.

3 Logging Facilities

To set a useful policy on applets, users need a way to qualify applets not only by their identities (their origin, their signers, etc), but also by their actions. For example, an applet that only accesses public information can be considered a benign visitor while an applet that tries to modify private system information can be considered suspicious requiring a higher level of security. In order to keep track of such actions by applets, a log needs to be maintained.

Logging will allow users to create conditional policies. For example, when determining whether an applet should be allowed to connect back to its host, a user may want to allow this only if the applet would not be able to compromise the user's privacy. In order to assess this safely, the user would need to determine whether the applet had accessed any private information. (This is better than having our user blindly trust the applet if it came from a trusted source, and distrust it if not.) Providing an auditing mechanism on applets allows users to inspect applets' past histories and check whether any private information was accessed. Of course users are free to allow or disallow all access to their private information, however, this logging feature provides them a way to be selectively restrictive.

Secondly, logging allows a system to keep track of information transfers through storage channels. Such inter-applet and intra-applet communication can be detected by inspecting the past actions of applets. An applet B that reads a file that was written (contaminated) by another applet A can be considered to have communicated with applet A. By this transaction, applet B could have acquired knowledge of any information gathered by applet A. The logging feature allows us to detect and/or prevent such information exchange.

3.1 Logging Accesses

Each time an applet accesses a resource, that action is logged for that applet. An applet's past history can be considered to be the union of all the log entries for that particular applet. To allow the placement of quotas on accesses, each log entry includes a count totalling the number of accesses to a particular resource. For example, one can set a maximum limit on the number of files to which an applet can write, or the number of times network connections are made.

3.2 Logging File Owners

Information exchange between applets via the file system opens the possibility of information leakage from higher privileged applets to lower privileged ones as shown in Figure 1. Such communication can be prevented at transfer 4 by introducing the notion of applet file ownership. This essentially divides the file system into applet domains. In other words, each file written by an applet is associated with its applet owner. Operationally, an applet becomes an owner of a file F once it has written to a pre-existing (but not previously owned) file F, or has created a new file F. This ownership lasts as long as the applet's stored information remains accessible, i.e. until the file is deleted (by the applet itself or by the user.) In the strictest sense, files owned by an applet cannot be read by other applets. This segregation of applets allows us to keep an accurate record of what information an applet has accessed. In our implementation, we have used the conservative approach of preventing communication at transfer 4, however the possibility of allowing file sharing and preventing communication at transfer 5 is discussed in Section 6.

Although information exchange between applets can also occur through other storage channels such as via a network connection, we are more concerned with communication via the file system. We assume that normally users would want to allow applets to continue accessing the file system even after they have accessed protected information, while further

access to the network would have been prevented. Hence, keeping a separate log for applet file owners is done for efficiency reasons. However, one can still create a policy to prevent two applets from using a network port as a communication channel.

3.3 Log Storage and Cleaning

As long as the applet is a file owner, an applet's log should last through the stopping and restarting of the applet, and through the exiting and reexecuting of the browser. Once the applet is no longer an owner of any files, it can be safely assumed that it is no longer storing any information from its past history and accesses, and thus it can start from a clean slate. Thus, the algorithm for cleaning entries in the logs is a function of whether the applet currently owns any files.

4 Constraint Language

In order to implement our prototype, we defined a constraint language. Our goals for this were that it address the problem as described earlier of time-delayed storage channels, allow for the control of resource usage, and permit the owner to eliminate certain Trojan horse programs. This was done by providing the ability to write constraints which are a combination of subject-based, object-based, and history-based policy statements. An additional important feature is the ability to assign labels to applets dynamically. Subject-based policies are those based on the identity of the subject or active entity, in this case the applet signature, as embodied in our global applet variables in Section 4.1, as is done in other such languages. Object-based policies are those based on the resource to which the applet wants access. These are also discussed in Section 4.1 in the form of global resource variables. In extending the language significantly, we have added the ability to write constraints in terms of the history of the behavior of the applet, allowing the owner to define denial of service behaviors, and limit them. This feature is realized in the **Any** and **All Past** expressions described in Section 4.2. Finally, our language allows for dynamic labelling of applets based on source or history as described in Section 4.3. The combination of such labelling and histories allows for the identification of certain Trojan Horse applets as well as other untrustworthy applets. The significant strength of this language is the ability to combine these features into constraints.

4.1 Variables

Global variables are provided as a common vocabulary for receiving information and setting permissions. The identity of an applet can be accessed through the variables:

```
Applet.Name,
Applet.CodeBase.Name,
  Applet.CodeBase.Host.Name,
  Applet.CodeBase.Host.IP,
Applet.Document.Name,
  Applet.Document.Host.Name,
  Applet.Document.Host.IP
```

Information about resources can be accessed through the variables:

```
File.Name, File.Path, File.AbsPath,
File.Parent, File.Size, Host.Name,
Command.Name, Property.Name
```

Permissions on resources can be set (true or false) through the variables:

```
File.read, File.write, File.delete,
Host.Connect.To, Host.Connect.From,
Command.Exec, Property.Read,
Property.Write, Window.Create
```

With JDK1.1's *java.security* package, additional variables can be further included to identify applets by their digital signers, etc. In the future, if the JVM can partition the CPU and memory usage by applets, then the variables for **CPU** and **Memory** can also be used.

4.2 Past Primitives

Two boolean procedures are provided for determining the usage of resources in the applet's past history (**Any** and **All**). A procedure was needed that would not only test whether a certain action had

occurred in the past, but that would allow the user to request more information about those past accesses. This led to the following syntax for these procedures:

```
(All <identifier> in Past <X> <predicate>)
(Any <identifier> in Past <X> <predicate>)
```

All is used to verify the truth of the <*predicate*> for every <*X*> accessed in the past; while **Any** is used to confirm the truth of the <*predicate*> for at least one <*X*> accessed in the past. <*X*> can be one of two things: 1) a resource or 2) an access. For example, if <*X*> is **File** then it corresponds to all the files that the applet has accessed in the past. However, if <*X*> is **File.Read**, then it corresponds to only the files that the applet has read in the past. The <*identifier*> is used to provide a nomenclature to identify the particular past resource inside the <*predicate*>. When the identifier appears in the <*predicate*>, it assumes the role of the current past resource being tested. Information about the resource can be requested within the <*predicate*> by using any of the variables for resources. For example, if <*X*> is **File**, and the <*identifier*> is **f**, then when **f.name** and **f.parent** appear in the <*predicate*>, they refer to information about the past file currently in question. See the example in Section 4.6 for a sample of this syntax.

4.3 Labels

We have also provided the variable **Applet.Category** for the labelling of applets. This is a modifiable label which can be used to group applets during runtime according to some condition. Having labels makes it easier to identify a set of applets in the access rules. A label can be based simply on the applet's origin such as a label for *is-trusted-to-access-my-mailbox*. More powerfully, a label can be set once an applet has done something in its history. This can be used for keeping track of applets' secrecy levels: *has-read-protected-files*, *has-read-only-public-files*, *has-not-read-any-files*. Also, labels can be applied to applets, as trust in them falls. For example, the label *suspicious* can be set if the applet has accessed more than a certain threshold of protected files. Policies for applets labelled *suspicious* can then be more restrictive. See an example of this in Section 4.6.

Labels have an ordinal ranking which can correspond to secrecy levels. If there are any conflicts in setting an applet's label, the minimum of the label values is conservatively set.

4.4 Primitive Procedures

Various primitive procedures are provided to determine useful information during runtime. These include boolean operators (**And, Or, Not**) and comparison operators (**<, >, =?, !=, <=, >=**). Additionally, procedures that do pattern matching on strings (**Match**) and test whether an element is part of a list (**OneOf**) are provided. Two procedures for totalling the number of accesses to a particular resource are given (**Count** and **CountAll**). For example, **(Count File.Read)** will return the total number of times the applet read the file in question, and **(CountAll File.Read)** will return the total number of times the applet read any file. See the example in Section 4.6 for a sample of their usage.

4.5 Rules Resolution

When a permission is to be checked for a particular resource, the user's applet policy is referenced. For efficiency reasons, instead of verifying all the rules each time an access is to be granted, only the rules that affect that access are verified. The exceptions are those rules that affect **Applet.Category** (they are always re-evaluated) since the applet's label can change at any time. For example, if permission to read a system property is requested, only those rules that assign **Property.Read** and/or **Applet.Category** are evaluated.

In evaluating the access permission for an applet, by default, the access is false. If there are no rules allowing the access, permission is not granted. If there are such rules, the final permission is the "and" of all the permissions set by the rules. So if there is at least one rule that denies access, the permission is denied. This resolves any conflicts that may arise.

4.6 Example

The following is an example of a simple policy that gives applets access to certain directories, to the network, and to the windowing system, while limiting

their usage. The number of file writes is limited to 50, the size of these files is limited to 500K, the number of network connections is limited to 20, and the number of windows that can be created is limited to 50. In addition, trusted applets are given read access to some protected directories. The policy also makes use of the **Applet.Category** variable in order to label trusted, contaminated (have read protected files), and suspicious applets.

```
// Define directories.
(Define PublicDirs ("/Public/*"))
(Define ProtectedDirs ("~/Mail/*"
                       "~/Diary/*"))
(Define WriteableDirs ("/tmp"))
(Define ReadableDirs (WriteableDirs
                      PublicDirs))

// Define security labels.
(Define Suspicious 0)
(Define Contaminated 5)
(Define Trusted 10)

// Default permissions on applets.
 // Reading files.
(If (and (!= Applet.Category Suspicious)
         (OneOf File.path ReadableDirs))
    (File.Read = true))

 // Writing files.
(If (and (!= Applet.Category Suspicious)
         (OneOf File.path WriteableDirs))
    (File.Write = true)
    (File.Delete = true))

 // Connecting to the Network.
(If (!= Applet.Category Suspicious)
    (Host.Connect.To = true))

 // Creating Windows.
(Window.Create = true)

// Define trusted applets.
(Define TrustedSources ("web.mit.edu"
                        "lcs.mit.edu"))

(If (OneOf Applet.CodeBase.Host.Name
           TrustedSources)
    (Applet.Category = Trusted))

// Trusted applets get more privileges.
// Allow them to read protected files.
(If (and (=? Applet.Category Trusted)
         (OneOf File.path ProtectedDirs))
    (begin
      (File.read = true)
      (Applet.Category = Contaminated)))

// But keep protected information inside.
(If (=? Applet.Category Contaminated)
    (Host.Connect.To = false))

// Limit number of files created.
(If (>= (CountAll File.Write) 50)
    (begin
      (File.write = false)
      (Applet.Category = Suspicious)))

// Limit file size to 500K.
(If (>= (CountAll File.Size) 500000)
    (begin
      (File.write = false)
      (Applet.Category = Suspicious)))

// Limit connections to the network.
(If (>= (CountAll Host.Connect.To) 20)
    (begin
      (Host.Connect.To = false)
      (Applet.Category = Suspicious)))

// Limit number of windows created.
(If (>= (CountAll Window.Create) 50)
    (begin
      (Window.Create = false)
      (Applet.Category = Suspicious)))
```

5 Implementation

Our prototype that includes the above features is developed on the Sun SPARC platform. We modified the Security Manager of the appletviewer in Sun's JDK1.0.2. Neither the JVM nor the system classes are modified. Our implementation uses the 1.0.2 API of the Security Manager and is built with the 1.1 JVM. In this section, we will first describe the implementation of the Security Manager, the rules, and the logs. We will then highlight a collection of further security issues and conclude with a summary of our performance evaluation.

5.1 Applet Security Manager

The applet Security Manager is questioned by the Java system classes when access to a system resource is requested. When one of the Security Manager's checkX methods is called, it uses the rules and the logs to determine whether the permission should be granted. Not all the checkX methods in JDK1.0.2's appletviewer's security manager were modified. 'checkCreateClassLoader' and 'checkExit' still throw security exceptions for applets since they would otherwise introduce major security hazards. Letting applets create their own classloaders would imbalance the safe foundation set by the lower level security in Java, and allowing applets to halt the JVM seems unnecessary. In addition, checkLink (which unconditionally throws a security exception) is not modified in the current system; however, more flexibility can naturally be provided in the future.

5.2 Rules

The rules are scanned and parsed using Sun's Java Compiler Compiler (JavaCC)[21]. Given lexical and grammatical specifications, JavaCC generates Java code that can parse the rules. The rules must conform to the grammar specified, otherwise a parsing error would be raised. Other errors including mismatched types, global redefinitions, assignments to read-only variables, illegal identifiers, negative applet categories, and so on, are also caught during parsing.

5.3 Logs

The applet file owner logs and the applet access logs are implemented using cached hashtables. The sizes of the two caches are specified as constants which can be easily modified. For now, they are arbitrarily set to size 16. The caches use a least recently used replacement policy.

In order to account for failure in the system, all the data in the cache is written back to the log after a certain number of events or minutes. In case the system fails or is exited abnormally, these regular writebacks will prevent major loss of information. The loggers include constants to specify the maximum number of events and the maximum number of minutes between writebacks. If the application exits normally, writebacks are also done upon exit.

5.4 Security Notes

In the implementation of the system, certain security issues needed to be addressed. As in Java, a safe design is only a support structure for a secure implementation. Five are highlighted here.

5.4.1 Error Resolution

How does one resolve various errors? Errors include I/O errors, parser errors, applet security errors, other runtime errors (i.e. out of bounds, unknown host), and other unexpected system errors. Since this system involves the maintenance of multiple simultaneous running applets, we need to classify these errors into *fatal system errors* and *applet-specific errors*. The first class of *fatal system errors* includes those errors where the entire system is halted, since otherwise, security violations would occur. If the system is inside a web browser, then the browser should halt all its Java operations when encountering a *fatal system error*. The latter class of *applet-specific errors* includes those errors where only a particular applet's execution is halted. These errors that affect only one applet should not halt the entire system. If the entire system were halted from such an error, then that would allow an applet to affect the execution of other applets by simply causing those errors (denial of service attack). It seems apparent that only those errors which affect all applets and the security of the entire system should be considered *fatal system errors*. This includes parser errors and I/O errors from reading the rules, and format and I/O errors from the file owner log. On the other hand, a security violation by a single applet and a formatting or I/O error in a single applet's access log need not affect the entire system. Instead, these errors are signalled for the benefit of the user and the execution of that applet is terminated.

5.4.2 Applet Identification

How does one identify applets? In the appletviewer, the host of the applet's document is most commonly specified using its DNS host name. In so doing, our

implementation simply identifies an applet by the host name of its codebase. Since a server sometimes uses multiple machines (and thus perhaps multiple IP addresses) to reduce its load, our prototype does not distinguish the same applet on the different machines. Identifying an applet by its host name (and not its IP address) allows the applet to later access its files. However, this leaves room for DNS spoofing attacks, in which incorrect entries in a DNS server lead to incorrect identification of applets.[2] If the DNS server becomes infiltrated since the last execution, then the correct identity of an applet changes. This introduces a vulnerability to an external source: the DNS server.

On the other hand, the support for digitally signing applets can address this. Instead of IP addresses or DNS host names, the signature on the applets would provide a secure mechanism for identification. This functionality has now been included in JDK1.1's *java.security* package. However, one may want to consider an applet that is updated to a newer version to have the same identity as its older version so that it may access its old files. This would require a slight variant to digital signing where the identity of the applet does not change with slight modification to its code if the author and the origin remain the same. Although this problem needs to be addressed, we do not address it further in this work.

5.4.3 CheckX methods

The checkX methods in the Applet Security Manager are called when system classes want to verify whether the current thread (applet) has the authority to access a certain resource. These methods are provided to check access. However, calling a checkX method does not necessarily mean that the given applet has performed that action, but only that the action was requested. Despite this, the current implementation logs it as if the action was performed. For example, the java.io.File.canRead method uses the checkRead method to just check whether the read permission should be allowed, although the file may not even be read.

Web browsers can also give applets access to the Security Manager, enabling applets to inquire about their permissions using the Security Manager's check-X methods. This feature allows applet authors to write more robust and useful code. However, with our current implementation, the checkX methods will log these inquiries as actual accesses. The outcome is that applet's accesses would be limited by these extra loggings if the applet policy includes rules that limit future accesses based on past ones. For example, if a rule states that "an applet cannot access more than 8 files," the applet is able to access only 4 files since the author's checks prior to each access would also be counted. This does not introduce any security holes if past accesses only limit future accesses. If this were not the case, then an applet could merely call the checkX methods without actually accessing the resource but instead extending its permissions. An example would be a rule that states, "an applet can access this protected file only if it has read this copyright." The applet could simply call the checkRead method on the copyright, but not read it.

To address this properly however, the Security Manager in the JDK should have two types of methods: one for checking access (checkX) and another for both checking and actually making the access (checkLogX). The former can be used by applets that want to know their permissions, and the latter should be private to the system classes.

5.4.4 Eliminating Race Conditions

One needs to make sure that the system is not faced with the same flaw as the one in *fingerd* where a malicious attacker exploits the race condition between checking the properties of an object and giving the permission. Applying this to our scheme, let us say there is a rule that states that an applet cannot connect to the network if it has read a local file. Then between the time the SM checks whether a thread can access the network and the time when it gives the permission, another thread of that applet reads a file. So in the end, the multi-threaded applet was able to circumvent the rule. In order to solve this problem, one must synchronize the Security Manager's check for accesses by applet rather than by thread.

5.4.5 Unix File System

We have currently limited our focus to the UNIX file system. To be consistent with the capabilities on the UNIX platform, giving write access to an applet does not mean it has read access. In our implementation, in order to allow an applet to read

and write, both permissions must be assigned to true in the rule.

On another note, the JDK API is limited in that the file permissions on the UNIX system are inaccessible. Therefore, there is no way (other than writing native code) of setting a file's ACLs or discovering whether a file's SUID bit is set. This limits the breadth of the policy one can place on an applet's access to the file system. Further, any files that are newly created by the Java runtime are given ACL permissions based on the user's current *umask* value. If the *umask* value does not prevent the creation of world-readable files, then all new files would be world-readable. The file permissions cannot be changed after its creation because of the limitation in the Java API. Consequently, the log files created by the system and files created by applets are dependent upon the *umask* value.

5.5 Performance Analysis

We have analyzed the performance of our implementation by executing an applet that reads a line from a file and writes a line to a file a certain number of times. We measured the amounts of time for reading and writing 1250 times, 2500 times, and 5000 times on a Sun Sparc 5. The experiment used a sample policy that gives trusted applets access to certain public directories while restricting the file size to 100K and the file writes to 50. This simple test involved logging the past accesses, checking the past accesses each time a read or write was to be done, and analyzing the rules during runtime. We compared our times with the times for JDK1.1's appletviewer whose Security Manager needed to be slightly modified in order to allow reads and writes to the file system. The results show that the amount of time our extended system takes is 1.67 times that of the regular appletviewer. In particular, our prototype takes .131 seconds for each additional iteration compared to the regular appletviewer's .079 seconds. Performance can be improved with further work in providing additional JVM native support and by using a Just-in-Time compiler.

6 Conclusion

This paper has addressed the issues that arise with making applets less restrictive by giving them more access to a user's operating system. We have attacked this problem by 1) supplying a constraint language that can specify conditional rules based on past actions and 2) monitoring the actions of applets through logging facilities. With these two features the information exchange described in Figure 1 can be easily detected and prevented.

Our implementation isolates applets by the notion of file ownership and by disallowing applets from reading files owned by other applets. However, as a future extension of our work, this restriction can be lifted if the sharing of files among applets is needed. Such a capability would be useful if one wants to implement applets that collaborate. For example, as one applet organizes and arranges a user's schedule, another could graphically present the scheduler, while another could communicate with other agents to make appointments. These teamplayer applets would need to communicate with each other and would need to share the common schedule files. With file sharing in place, the communication control would be pushed down to step 5 of Figure 1.

One possible secure implementation of file sharing would require associating a static security label with each file in addition to the dynamic label associated with each applet. The label on the file would denote the security level of its contents, while the label on the applet would correspond to the highest security level of the information that it had accessed up to that point. Then applets with the same security level could access the same files without compromising the local system. This way, if applet communication occurs through the shared files, the applets would have accrued the same security level of information. Such an implementation would require extending the constraint language to allow users to specify the security levels of files in a straightforward way, so that the rules would be less prone to error.

Although in this paper we have primarily addressed issues with applets communicating via the file system, there are also other storage channels through which applets can communicate. These include the method calling between applets from the same document through the procedures *getAppletContext* and *getApplet*, and the spawning of new applets on the

local file system. More details about these storage channels and how our prototype addresses them can be found in Mehta's thesis [13].

In conclusion, we believe that the addition of conditional rules referencing past actions and complementary logging facilities will add significantly to the usability of the Java security mechanism. These features will also allow us to address the storage channels that exist in the system. Furthermore, these features can be easily portable to JDK1.2 and other mobile code systems. We have demonstrated reasonable performance of this functionality in a prototype implementation.

7 Acknowledgments

This work was supported by the Department of Defense Advanced Projects Research Agency under contract number DABT63-94-C-0073 for work done at MIT's Laboratory for Computer Science.

References

[1] D. D. Clark, D. R. Wilson, *A Comparison of Commercial and Military Computer Security Policies*, **IEEE Symposium on Security and Privacy**, Oakland, CA, April 1987, pp. 184–194.

[2] D. Dean, E. W. Felten, D. S. Wallach, *Java Security: From HotJava to Netscape and Beyond*, **IEEE Symposium on Security and Privacy**, Oakland, CA, May 1996, pp. 190–200.

[3] I. Goldberg, et al. , *A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker*, **USENIX Security Symposium**, San Jose, CA, July 1996, pp 1–13.

[4] L. Gong, *Java Security: Present and Near Future*, **IEEE Micro paper**, **17**(3), May/June 1997, pp. 14–19.

[5] L. Gong, **Java Security Architecture (JDK1.2)**, Rev. 0.5, July 10, 1997.

[6] D. Hagimont, L. Ismail, *A Protection Scheme for Mobile Agents on Java*, **ACM/IEEE International Conference on Mobile Computing and Networking**, Budapest, Hungary, 1997.

[7] S. Jajodia, *A Logical Language for Expressing Authorizations*, **IEEE Symposium on Security and Privacy**, Oakland, CA, May 1997, pp. 31–42.

[8] JavaSoft, Sun Microsystems, *HotJava(tm): The Security Story*, May 1995, http://www.javasoft.com:/sfaq/may95/ security.html.

[9] JavaSoft, Sun Microsystems, *HotJava(tm) Browser, Version 1.1 Beta2* http://www.javasoft.com:/products/hotjava/ 1.1.

[10] G. Karjoth et al, *A Security Model for Aglets*, **IEEE Internet Computing**, **1**(4), July/August 1997.

[11] B. Lampson, *A Note on the Confinement Problem*, **Communications of the ACM**, **16**(10), October 1973, pp. 613–615.

[12] R. Lo, K. Levitt, R. Olsson, *MCF: A Malicious Code Filter*, **Computers & Security 14** (6), 1995, pp. 541–566.

[13] N. V. Mehta, **Fine-Grained Control of Java Applets Using a Simple Constraint Language**, MIT/LCS/TR-713, June 1997. Also thesis for Master's of Engineering, MIT. June 1997.

[14] Microsoft Corporation, *Microsoft Security Management Architecture White Paper*, May, 1997, http://www.microsoft.com/ie/security/ie4security.htm.

[15] I. S. Moskowitz and M. H. Kang, *Covert Channels – Here to Stay?*, **COMPASS '94**, Gaithersburg, MD, June 1994, IEEE Press, pp. 235–243.

[16] G. Necula, *Proof-Carrying Code*, **ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages** Paris, France, January 1997, pp. 106–119.

[17] Netscape Communications Corporation, *Securing Communications on the Intranet and Over the Internet*, July 1996, http://www.netscape.com/newsref/128bit.html.

[18] J. Pitkow and C. Kehoe, **GVU's 7th WWW User Survey**, Georgia Institute of Technology, April 1997, http://www.cc.gatech.edu/gvu/usr_surveys/survey-1997-04.

[19] R. Sandhu, *Transaction Control Expressions for Separation of Duties*, **4th Aerospace Computer Security Conference**, December 1988, pp. 282–286.

[20] R. T. Simon, M. E. Zurko, *Separation of Duty in Role-Based Environments*, **Computer Security Foundations Workshop**, Rockport, MA, June 1997.

[21] Sun Microsystems, Inc. , Java Compiler Compiler, Version0.6(Beta), 1997, http://www.suntext.com/JavaCC/.

[22] L. van Doorn, et al., *Secure Network Objects*, **IEEE Symposium on Security and Privacy**, Oakland, CA, May 1996, pp. 211–221.

[23] D. S. Wallach et al., *Extensible Security Architectures for Java*, **Symposium on Operating Systems Principles**, St. Malo, France, October 1997.