LAVA: Secure Delegation of Mobile Applets

Shyhtsun F. Wu[*] *Matthew S. Davis*
Jatin N. Hansoty *Jim J. Yuill* *Sam Farthing*

Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206

Jeffrey S. Webster
Xueqing Hu

Department of Mathematics
North Carolina State University
Raleigh, NC 27695-8205

October 9, 1996

[*]Supported in part by the U.S. Department of Defense Advanced Research Projects Agency and the U.S. Air Force Rome Laboratory under contract F30602-96-C0325, and in part by the Center for Advance Communication and Computing under grant FAS #5-30183.

LAVA: Secure Delegation of Mobile Applets

Abstract

The security problems associated with Java have received significant attention. In a recent meeting [SP996], many people have expressed their concerns about Java not having a clear security model and policy. *Lava* is a proposed architecture to enhance the security of Java. Unlike the original Java which only protects the client (executant), Lava protects client and server (provider) together as a pair. In this paper, we will first describe the concept and two applications of *applet delegation*. Then, from these applications, we define security requirements and policy for Lava. We also explain why the current Java security is insufficient. Finally, we present the Lava architecture itself.

1 Introduction

Mobile agents/applets [CGH+95] are programs typically written in a script language like Telescript [Whi95] or compiled as byte-code segments from a high-level programming language like Java [Fla96]. They are platform independent and may be transported to any host on the Internet and executed remotely. Remotely executed applets offer an interesting way to implement many distributed applications.

Security is the major concern in mobile applets [HCK95], and Java's security problems in particular have received significant attention. For example, in [DFW96], many security flaws in the Java implementation have been identified. At a recent meeting [SP996], many people have expressed their concerns about Java not having a clear security model and policy. In fact, Java's security policy is "unspecified" in the language itself. It is the user's responsibility to implement the public abstract class *java.lang.*$SecurityManager$ [GY96] to exercise different security models or policies. As we will discuss later, this approach provides insufficient security requirements for many interesting distributed applications like network management.

In most existing Java applications, a client (*e.g.,* a Web browser like netscape) *downloads* a mobile applet from a server (*e.g.,* a http server), and *executes* the downloaded applet. Since our concerns here are viruses or Trojan horses being attached to an applet that may destroy the client's local resources, most security issues focus on how to protect the clients. In this paper, we further consider applications where a server *delegates* an applet to a client and collects useful results from the client. In a delegation process, the server initiates the process and is really a "client" because the server requests the client to execute a particular applet. In this paper, we use different terms: *applet provider* and *applet executant*[1]. The provider owns the applet, while the executant receives and executes the applet locally.

To support both *secure downloading* and *delegation*, we need to provide a mutual trust relation between the *provider* and the *executant*. We propose an enhanced Java security architecture, Lava, which supports both downloading and delegation. In the next section, we present two applications where delegation is useful. Then, we define the security requirements and policy. Next, we examine the current security issues in Java. Finally, we describe our solution, Lava.

2 Examples for Applet Delegation

In this section, we present two applications where *applet delegation* is useful. For both examples, the current Java security model does not support a safe implementation. We will use these two examples

[1] In A.S. Hornby's Oxford Dictionary, the word "*executant*" means *person who executes a design, etc.*

to identify security requirements in the next section.

2.1 Network Management for Wireless/Mobile Networks

A network management system consists of two different roles: management applications and agents. An agent collects/abstracts low-level network information and sends it to the management applications upon requests [LaB91]. A management application, on the other hand, requests information from agents, and makes management decisions based upon the management information obtained from the agents. Sometimes the management application may request the agents to perform some actions on low-level devices. For example, a device could be an IP router, an ATM switch, a wireless base station, or a mobile host. Usually an MIB (Management Information Base) agent, which is implemented in software, is running on (or very close to) that device while a management application is on the system administrator's console.

Assume that we have a service management application for wireless networks (*e.g.,* [CDP95]). This application needs to periodically poll the agents on the mobile hosts and base stations to see if they are functioning properly. Therefore, management information needs to be sent from the agents to this management application. The wireless bandwidth used to send this management information is relatively small but expensive. In order to save bandwidth, we can implement a portion of the management application as a Java applet and delegate it to the agent. The applet running on the agent site can directly access the local MIB without paying for the wireless bandwidth.

In the above example, the management application is the provider of applets, while the MIB agent is the executant of those applets. The idea of delegating code to a remote site is not a new idea. Work [YGY91] has been done in applying the concept of management-agent delegation in the domain of system and network management. The recent acceptance of the Java paradigm makes this approach even more attractive. However, the security issues have not been sufficiently considered.

2.2 Distributed Intrusion Detection Systems

A *distributed intrusion detection system (DIDS)* [SRS91], in a manner similar to a network management system, collects local intrusion events from different sites. It then correlates events and decides whether or not the system is indeed under attack. One practical concern in implementing a DIDS is the performance overhead in detecting and transmitting all these local events. For example, in order to detect K types of attack, we need to check N different security rules. For every incoming packet, the local detection module performs N checks and reports one of the K possible attack types to the global DIDS. If N is too large and the local node has a relatively small memory, sometimes we can only activate or keep a small portion of the whole detection rule set in the node. Thus, DIDS is expensive because of the requirements for memory space, local computation and network bandwidth. If we do not carefully distribute the detection resources, the system performance might degrade significantly.

One smart way to save detection resources is to implement the detection rules as mobile applets. With mobile detection applets, we have the potential to place the resources in a more flexible way. For instance, we need to activate K detection rules at the beginning. After collecting and correlating events, the global DIDS suspects that the system is under a special type of attack X. Assume that we need only two active rules to detect X. Now, we can delegate these two rules to all the local detection modules to further confirm the existence of attack X.

If the global DIDS has the information about the location of the X attack source, we can do even better with applet delegation. The attack packets must go through one of the gateways surrounding the source. This fact suggests that we only need to delegate these two checking rules to these gateways

(given that these gateways can run mobile applets). This results in optimal attack isolation, and the mobile applets become efficient mobile firewalls.

3 Threads

In the previous examples, we have shown that the delegation of applets can achieve interesting results. On the other hand, in order to get the desired results, the applet delegation and downloading process itself must also be safe and secure. In this section, we examine the security requirements and define a security policy for mobile applets.

3.1 Evil Applets

An untrusted applet might contain a virus that will damage the executant's local resources. Therefore, we need to check and verify that this is indeed a good applet before executing it. This checking process can be done in two different ways. First, we can verify the applet itself using, for example, a byte-code verifier. Second, we can check whether the source of this applet is trustworthy. For example, we can check whether this applet is from a trusted applet provider or if it was compiled by a trusted Java compiler.

3.2 Invalid Access of Resources

An applet should not be able to access attributes or objects for which it does not have the access rights. For example, a private attribute in an object should only be accessed through the defined object's public interface. Attribute access control can be performed at compile-time or run-time. However, object access control can only be handled at run-time. To see this, consider the situation where both the user who is running the applet (executant) and the applet's source (provider) are used to determine whether the applet has access rights to a device object. This information becomes available only at run-time. This implies that an applet A from a provider T is allowed to print a message on the screen while the same applet A from a different provider may not have the right to do the same thing.

3.3 Privacy in Downloading

An executant who downloads applets might not want others to know what he has downloaded. Similarly, a provider might not want to reveal the information about what it delegates to other executants. Furthermore, the downloading and delegation process itself should not create a covert channel that might be used to penetrate a local firewall.

3.4 Replay Attacks

When an applet provider delegates a trusted applet to an executant, an evil entity may eavesdrop and store the whole message. This evil entity may then later retransmit the whole message. For example, say a provider (a network management application) sends an applet to shutdown and reconfigure an executant (a router). A bad guy can store and replay this valid request to shutdown the router any time he desires.

4 Security Policy

A security policy for mobile applets specifies what information and resources can be accessed by an applet and which applets can be used by a particular user. It is driven by the system security requirements and in turn drives the Lava system design. In this paper, we use Sandhu's *TAM (Typed Access Matrix)* model [San92] as a framework for constructing the security policies. The policy we introduce here is high-level and preliminary. Especially, we have not considered multiple levels of trust relations between an applet provider and an executant. For example, providers in different trust levels should access different sets of system resources.

4.1 Subjects and Objects

As shown in Figure 1, we have different subjects and objects:

Java Program: A Java program is an object written in Java.

Applet: An applet is a subject created/owned by an originator from a Java program..

Originator: An originator is a subject that originally produced a particular applet. If the originator is trusted, then certain compile-time security checks should have been performed. Please note that this only implies that known security problems in Java programs have been checked. As more bugs/problems might be discovered, the produced applets might become invalid after a period of time.

Provider: An applet provider is a subject who currently subscribes/holds a copy of a particular applet. Different providers of the same applets might have different access rights to local resources. This access right information is used by the *java.lang.*$SecurityManager$ implementation for object-level access control.

Executant: An executant is the owner of local resources. Applets produced by an originator may be downloaded from an applet provider by an executant. Alternatively, the executant may also be delegated an applet to execute by a provider. After performing security checks, the executant runs the received applet under its local environment and $SecurityManager$.

System Resource: Sytem resouces are objects owned by an executant.

4.2 Policy

The Typed Access Matrix for Java is described in Figure 3. We use the same primitive operations and notations defined in [San92]. The ***Rights*** $R = \{own, read, r/w, cread, cr/cw, source, trust\}$.[2] The ***Types*** $T = \{originator, provider, executant, applet, java, resource\}$, while the subject ***Types*** $T_s = \{originator, provider, executant, applet\}$. Figure 2 contains two examples of the commands, while the complete list will appear in a technical report:

4.2.1 Trusted Originators

As shown in the **command** *download*, an executant should only run those applets produced by trusted originators. To determine if an originator is trusted, a signature or certificate by the originator should be attached to the applet. Then, the client can verify the identity of the originator.

[2] $cread$ is for confined-read.

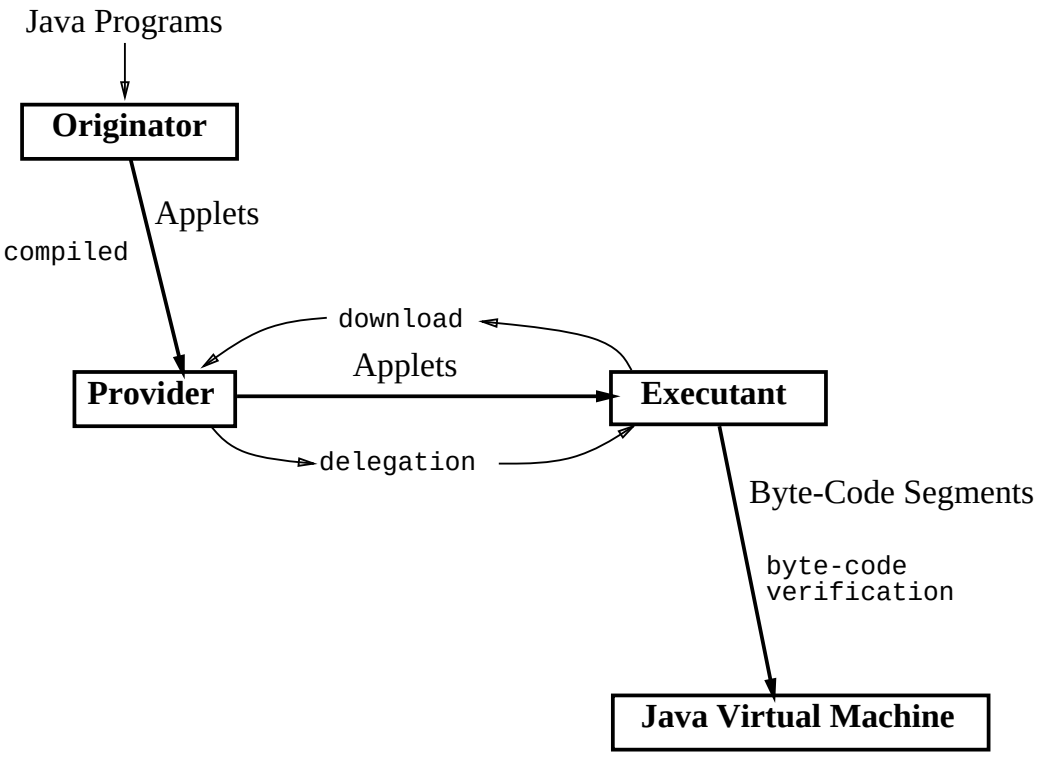

Figure 1: Subjects and Objects

command $compile(O : originator, J : java, A : applet)$
create subject A of type $applet$
enter $source$ into $[A, J]$
enter own into $[O, A]$
end

---

command $download(O : originator, P : provider, E : executant,$
$A : applet, J : java, R : resource)$
if ($trust \in [E, O]$ and $trust \in [E, P]$
and $source \in [A, J]$ and $own \in [O, A]$) then
enter $cread$ into $[E, A]$
enter cr/cw into $[A, R]$
end

Figure 2: Two defined commands

	OBJECT					
SUBJECT	Originator	Provider	Executant	Java	Applet	Resource
Originator				own	own,r/w	
Provider	subscribe		trust		cread	
Executant	trust	trust			cread	own,r/w
Applet				source		cr/cw

Figure 3: The Typed Access Matrix for Java

4.2.2 Trusted Providers

A provider's identity is associated with every downloaded or delegated applet. This identity determines whether the provider (or the current holder of an applet) is trusted. Furthermore, at run-time this identity is used to decide whether this applet has rights to access certain objects or system calls.

4.2.3 Trusted Executants

Before an applet is downloaded or delegated, the provider should check the identity of the executant. Sometimes only specific executants are allowed to run certain applets. For example,

1. The provider might attach some confidential information with delegated or downloaded applets. This information is sensitive and only the trusted executants should be allowed access.
2. The executant must be a healthy and secure run-time environment. Consider the bugs found in the Java byte-code verifier (BCV), and we will have a version of Java run-time environment which fixes all the known bugs. Under this situation, it is certainly desirable to request that the executant's BCV must be the latest one. If an executant is still running an old version of BCV, the provider's delegated applet (including attached confidential information) might be in danger. Furthermore, the results generated by delegated applets may not be trusted.

4.2.4 Secure Downloading/Delegation

It is desirable to check the following:

- When an executant receives an applet from a trusted provider, it must also verify that the provider indeed intended to send this applet to this executant. This is especially important for the case of delegation. Furthermore, the request from the provider must be fresh, *i.e.,* it is not a replay. (*I.e.,* source authentication and access control.)
- The received applet must have a secure check-sum. (*I.e.,* integrity.)
- When a provider delegates/sends an applet to an executant, it must make sure that only the intended executant can understand and execute the applet. This protects the applet's privacy. (*I.e.,* confidentiality).
- A provider, after making a delegation request to an executant, can not deny that he made the request. Similarly, an executant, after downloading, can not deny what it did. (*I.e.,* nonrepudiation.)

5 Analysis of Java's Security Architecture

5.1 Java Security Holes

In [DFW96], many security flaws in Java were identified and all of them can be fixed by modifying the implementation. The most serious Java flaw is the bug in the byte-code verifier. Through this security hole, a vicious applet can extend the *java.lang.*$ClassLoader$ and load any piece of native code which might destroy the user's important information.

Since the Java implementation is not secure, it has been recommended that the netscape users turn off its JAVAbility. A Java compiler performs extensive compile-time checking to detect errors in the Java programs. However, since the applet is transferred in the byte-code segment format, the compiler itself does not offer sufficient security. *The source of the problem is that, when an executant receives an*

applet, it can not tell whether this byte-code segment is generated by a trusted Java compiler or not. It turns out that the Java run-time system has to verify all sorts of potential problems associated with byte-code segments.

We suggest that this is in fact a problem in Java's security architecture. To verify/check byte-code segments is a difficult task. As indicated in [DFW96], the byte-code is in a linear form, so type checking requires a global dataflow analysis. This analysis is further compilcated by the existence of exceptions and exceptional handlers. While the BCV bug is merely an implementation bug, it is hard to guarantee that there are no more similar bugs in the new release. We suggest that, with a proper security policy and architecture, this bug might not have happened in the first place. Please note that the BCV attack we just mentioned would have been impossible if every accepted byte-code segment is generated from a trusted Java compiler.

5.2 *java.lang.*$SecurityManager$

Java's security policy is "unspecified," and its security depends on its implementation. It is the user's responsibility to implement the public abstract class *java.lang.*$SecurityManager$ to exercise different security models or policies. The default $SecurityManager$ simply disables all access to system calls like `connect` or `listen`.

5.2.1 Java in Netscape

A mobile applet in Netscape is restricted in networking access. If the applet security mode is "applet host" (default mode for Netscape), then the applet is only allowed to connect back to the provider. This implies that the provider must offer a network port open to receive messages from applets. Since the messages pass through the normal socket interface, the provider has no way to tell whether a particular message is really from its own applet. Without network-layer security support [Lin94], it may be even impossible to tell that it is really from the executant. This implies everybody else on the network can also access this port. Therefore, only non-critical information can be accessed through this port. To summarize, the mobile applet has absolutely no better rights to access any resources in either the executant or the provider. In other words, whatever an applet can access, anybody else can access as well.

We observe that Java/Netscape takes a rather naive approach to implement the $SecurityManager$. From the architectural point of view, it is safe because it technically disallows any remote access to critical information. This is highly undesirable for applications like distributed network management.

5.2.2 Advent's Java SNMP Package

Advent used Java to implement a SNMP MIB Browser[3][Adv96a]. The MIB browser is implemented as an applet and any Java-capable Web browser can be converted into a SNMP MIB browser by downloading that applet. Because of the network restriction in Java, Advent's SNMP product has provided a means of *bypassing* these security restrictions. The browser can communicate with a host other than its provider. To do this, the Web server (provider) runs a JAVA application called *SNMP Applet Server (SAS)* [Adv96b]. Whenever the browser applet needs to communicate with another SNMP agent, it does so via the SAS as shown in Figure 4. In other words, the SAS (and the applet provider) offers an open port for the executant to access the rest of the world.

[3]To our best knowledge, the SNMP agent has not been implemented in Java.

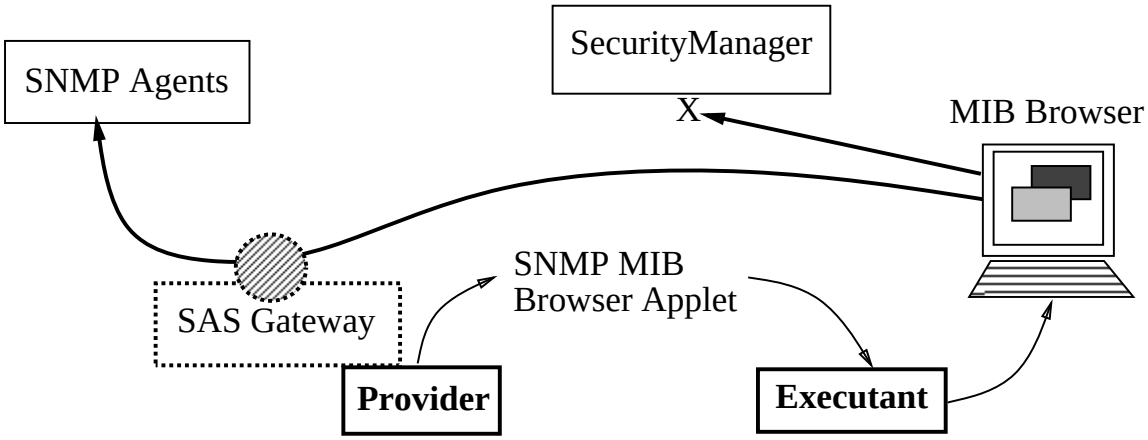

Figure 4: Bypassing Java Security in JavaSNMP

In [hid96a], a few security concerns have been identified for the Advent's SAS approach. These holes can possibly damage the provider, the executant, as well as the SNMP management agents. In general, we suggest that bypassing the security mechanism is not a safe approach. On the other hand, we can understand the reason for Advent taking this approach: they want to use Java to do something useful. In order to do these things properly and securely, we definitely need a better security architecture.

5.2.3 Summary

It is possible to implement the $SecurityManager$ in a better way such that some of the issues in Java/Netscape and JavaSNMP might be resolved. The idea of the $SecurityManager$ is good but insufficient by itself. Specifically, it is unable to support the security requirements and policies for many interesting applications.

6 The LAVA Architecture

Lava is a proposed architecture to enhance the security of Java. Unlike the original Java which has a very vague security policy, Lava is developed on top of a defined security policy as describe in Section 4. The LAVA architecture consists of one relation and a few secure protocols. The relation is the basis for two parties to securely interact with each other. The first protocol is the compiler authentication technique to verify the applet's originator. Then, we consider how to build up a trust relation between an applet provider and executant.

6.1 Secure Association

Before two parties (a provider and an executant) can securely interact with each other, they must build up a secure association relation. This relation, which should be generic to support various applications, defines party identification, shared secrets (master keys), security transformation schemes, and secret exchange protocols. Party identification is a string uniquely identifying a user, a process, a host, or a security gateway. Secure transformation schemes, for example, might be authentication plus confidentiality using RSA. If session keys, in addition to master keys, are needed, the secret exchange protocols decide how to obtain them.

The security association is part of the $SecurityManager$ class implementation. From an executant point of view, different applet providers should enjoy different levels of local resource access. For example, a normal trusted applet provider has read-only access to the value of the executant's routing table, while the trusted network administrator has read-write access to that table.

6.2 Secure Originator and Compiler Authentication

A "trusted compiler" is a program such that the byte-code fragments produced by this program are safe from a compiler point of view. In other words, this trusted compiler will perform extensive safety and security analysis before byte-code generation. The "trusted" attribute attached to a compiler might change over time. For example, today we might consider JDK-1.0X as trusted, but after a few months, we may find bugs in this version so that it becomes untrusted. Furthermore, a trusted compiler must associate with a party named "trusted compiling server (TCS)." For verification, the receiver of this compiled applet must have a secure association relation with this TCS.

The TCS generates a *"compiler certificate" (CompileCert)* along with the produced byte-code fragment:

$$CompileCert(Applet) = E_{TCS_{private}}[TimeStamp, MAC(Applet), CompilerIDVersion],$$

where $TimeStamp$ is the compiling time, and MAC can be MD5, SHA, or other secure transformation functions. Since the receiver knows the public key of TCS, the byte-code verifier can use $CompileCert(Applet)$ to verify the originator of this applet.

When transmitting an applet, we should send:

$$[Applet, TCS_{id}, CompileCert(Applet)].$$

The receiver of this applet will perform the following checks:

1. Check if the TCS_{id} is a trusted party or not. If yes, use the public key of TCS_{id} to decrypt $CompileCert(Applet)$.
2. Check the timestamp. The receiver can set a freshness constraint (for example, 30 days) to reject old applets.
3. Check the $CompilerIDVersion$.
4. Run MAC for this Applet, and verify if the result is equal to the $MAC(Applet)$ value.

If an applet passes all four tests, we can conclude that this applet is indeed from a trusted originator.

If the applet failed any test, the receiver should send an error message to notify the provider of this rejected applet. The reason of rejection is attached to the message and is used by the provider to handle the problem. For instance, a bug is just found in one particular version of a previously trusted compiler. Immediately, the receiver eliminates that $CompilerIDVersion$ from its trusted list. Therefore, it is very likely that a previously trusted applet suddenly becomes untrusted. Furthermore, some previously loaded applets may be invalid.

6.3 Secure Provider and Executant

Having a trusted originator does not solve the whole security problem. Two different applet providers using the same applet from the same trusted originator may be treated differently. Thus, it is important to have a mutual trust relation between a pair of provider and executant. In this subsection, we assume that there exists a security association relation between the provider P and the executant E.

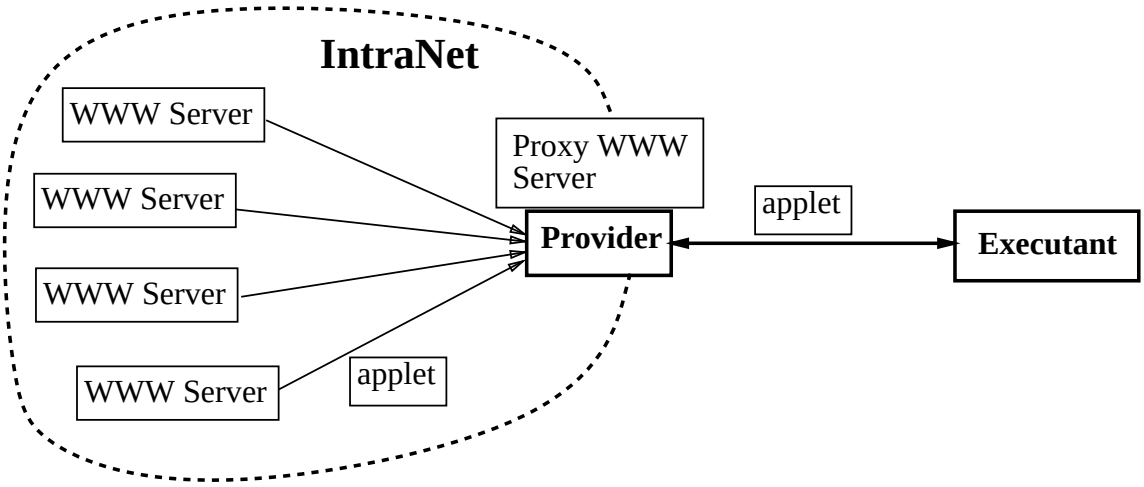

Figure 5: A Proxy WWW server as a Provider

6.3.1 Secure Downloading

E would like to download an applet A with URL_A. It first checks that P is the trusted provider for URL_A. Please note that P may be the WWW server for this URL but it also might be a secure proxy server for a set of WWW servers in an Intranet. In the latter case, E simply has one security association relation with P and it may enjoy all the secure applet services behind P (Figure 5).

After identifying P, E sends a request to P:

$$[Request, E_{id}, E_{K_{pe}}(URL_A, RunTimeVersion, Nonce(E), E_{id})].$$

P receives this request, verifies it, and replies:

$$[Reply, P_{id}, E_{K_{pe}}(URL_A, [Applet_A, TCS_{id}, CompilerCert(Applet_A)], (Nonce(E)+1), P_{id})].$$

The executant should verify the provider and check the originator of $Applet_A$. Then, at run time, the system's $SecurityManager$ should grant the local resource access according to P_{id}. The $Nonce$ is to prevent replay atacks, while the $RunTimeVersion$ is used to check if the run-time environment of the executant E is recent enough.

6.3.2 Secure Delegation

P would like to delegate one of its applets, A, with URL_A to a router R under an executant E. Please note that R and E might refer to the same node or E is an application-layer security gateway for a bunch of hosts, switches, and routers. Thus, a remote network management application P can go through the gateway E to monitor/manage a network of devices behind E.

After identifying E, P sends a request to E:

$$[Request, P_{id}, E_{K_{pe}}(Node_{id}, URL_A, [TCS_{id}, TimeStamp, CompilerIDVersion], Nonce(P), P_{id})].$$

E checks if it has security association relations with both P and TCS_{id}, and then decides to accept or reject:

$$[Reply, E_{id}, E_{K_{pe}}(Node_{id}, URL_A RunTimeVersion, Nonce(P+1), Nonce(E), E_{id})].$$

Finally, after verifying the $RunTimeVersion$, P sends:

$$[Final, P_{id}, E_{K_{pe}}(Node_{id}, URL_A, [Applet_A, TCS_{id}, CompilerCert(Applet_A)], (Nonce(E)+1), P_{id})].$$

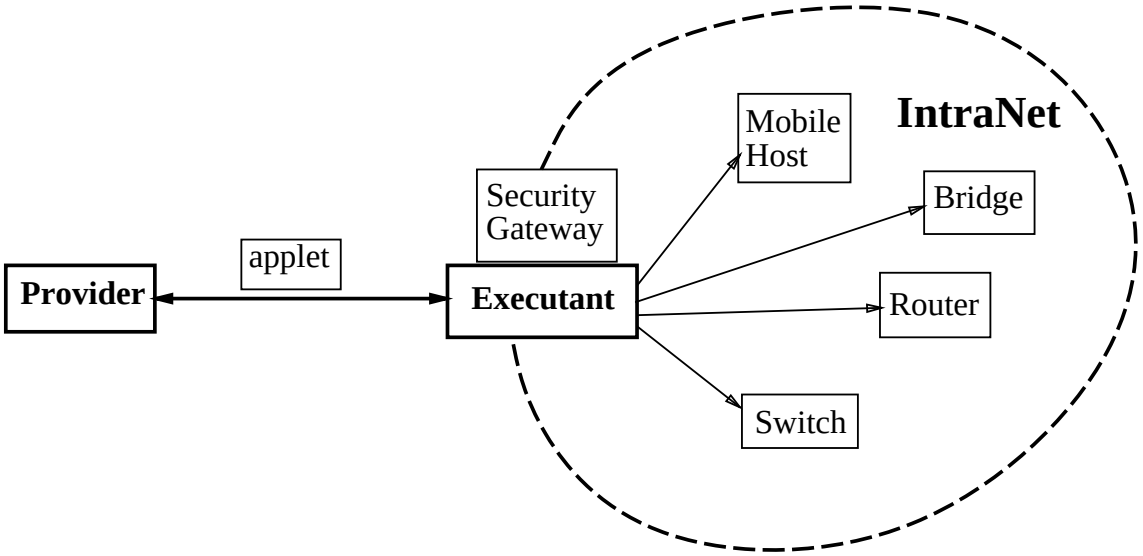

Figure 6: A Security Gateway as an Executant

7 Related Works

The idea of delegation for network management originated from the MbD project [YGY91]. In [KH92], a programming model was proposed to support dynamic reconfiguration. In [Zen95], the idea of delegation was used for efficient communication in mobile computing. Recently, the idea of delegation is applied to security management and intrusion detection [hid96b]. However, our work presented here is the first one to consider the security issues for the delegation process itself.

8 Future Works or Advanced Security Requirements

8.1 Result Verification

In the delegation case, the result generated by the delegated applet is usually useful for the provider of this applet. It is thus desirable for the provider to verify the result is really generated from the applet it delegated earlier. Verification of remote execution results is a very difficult problem in general. We are currently studying solutions for some special cases like applications in electronic commerce.

8.2 License Verification

Licensing of Java applets is a significant problem which we are currently working on. The goal is for every applet to have an identity for showing license information. A client should not take an applet from a server if the server does not present a proper transferable license. Similarly, the server will not send out an applet unless the client presents a valid license. Another problem is how to control the timing issue associated with the applets. For example, user A might get a temporary license for thirty days, and he downloads the applet legally during these thirty days. However, after this trial period, it is very difficult to prevent the client from continuing to use this code. A simple timing control might not work since the evil client can update its own clock value. Furthermore, how to prevent or discourage the executant from illegally redistributing licensed applets is yet another important problem.

9 Conclusion

In this paper, we consider the security problems related to mobile applets. The first contribution is to identify the security requirements and to define a simple security policy for systems like Java or Telescript. Second, we performed security analysis on the current Java-based systems like *Netscape* and *JavaSNMP*. Based on the security policy and requirements as well as our experience, we suggest that we really need a cleaner and more powerful security architecture for doing certain advanced applications. These include network management systems and distributed intrusion detection systems. Finally, we offer a new architecture Lava to fulfil the basic security requirements for mobile applets.

Currently we are working on advanced security requirements. These new requirements are mainly for electronic commerce and it is very challenging to fulfill these requirements. Also, we have started to build a Lava prototype to further demonstrate and evaluate our architecture presented in this paper.

References

[Adv96a] Advent. Java SNMP. http://www.adventnet.com/snmp_api.html, March 1996.

[Adv96b] Advent. SNMP Applet Server (SAS). http://www.adventnet.com/sas.html, March 1996.

[CDP95] Cellular Digital Packet Data System Specification. CDPD Forum, Release 1.1, January 1995.

[CGH+95] D.M. Chess, B. Grosof, C.G. Harrison, D. Levine, and C. Parris. Itinerant Agents for Mobile Computing. Technical Report RC 20010, IBM Research Division, March 1995.

[DFW96] D. Dean, E.W. Felten, and D.S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *1996 IEEE Symposium on Security and Privacy*, pages 190–200, May 1996.

[Fla96] David Flanagan. *Java in a Nutshell*. OŔeilly and Associates, Inc., Sebastopol, CA, 1996.

[GY96] J. Gosling and F. Yellin. *The Java Application Programming Interface, Volume 1: Core Packages*. Addison Wesley, 1996.

[HCK95] C.G. Harrison, D.M. Chess, and A. Kershenbaum. MobileAgents, Are They a Good Idea? Technical Report d953, IBM Research Division, March 1995.

[hid96a] hidden. hidden, 1996.

[hid96b] hidden. hidden, 1996.

[KH92] Gail E. Kaiser and Brent Hailpern. An Object-Based Programming Model for Shared Data. *ACM Transactions on Programming Languages and Systems*, 14(2):201–264, April 1992.

[LaB91] Lee LaBarre. Management by Exception: OSI Event Generation, Reporting, and Logging. In I. Krishman and W. Zimmer, editors, *Second Integrated Network Management Symposium*, pages 227–242, Washington, D.C., April 1991.

[Lin94] Mark H. Linehan. Comparison of Network-Level Security Protocols. Technical Report White Paper, IBM Research Division, June 1994.

[San92] Ravi S. Sandhu. The Typed Access Matrix Model. In *1992 IEEE Symposium on Research in Security and Privacy*, pages 122–136, May 1992.

[SP996] IEEE SP96. Java Policies. http://www.cs.princeton.edu/sip/pub/secure96.html, May 1996.

[SRS91] et. al. Steven R. Snapp. DIDS (Distributed Intrusion Detection System) - Motivation, Architecture, and an Early Prototype. In *14th National Computer Security Conference*, pages 167–176, October 1991.

[Whi95] J.E. White. *Telescript Technology: An Introduction to the Language*. General Magic, Inc., Sunnyvale, CA, 1995.

[YGY91] Yechiam Yemini, German Goldszmidt, and Shaula Yemini. Network Management By Delegation. In I. Krishman and W. Zimmer, editors, *Second Integrated Network Management Symposium*, pages 95–107, Washington, D.C., April 1991.

[Zen95] Bruce Zenel. A Proxy Based Filtering Mechanism for the Mobile Environment. Technical Report CUCS-0xx-95, Computer Science Department, Columbia University, 1995. Thesis Proposal.