Article ID: 1007- 2985(2003)01- 0026- 11

Security Issues for Java- Based Agents

LIU Chuan-cai
(College of Information Science and Technology, Fuzhou University, Fuzhou 350002, Fujian China)

Abstract: With the popularization of computer networks, there has been a shift proposed in distributed system programming from the remote procedure call to the remote programming paradigm, to decrease network traffic and improve performance. Software agents could be used to accomplish this task. The use of agents has several advantages and a few disadvantages, including added security issues. In order to implement autonomous security of Java agent, the author presents a new module system for Java that improves upon many of the deficiencies of the Java package system and gives the programmer more control over dynamic linking. Next, the author develops a general agent model and discusses general security issues in that model. Finally, the author proposes a practical solution that addresses some of those security issues.

Key words: software agents; remote procedure call; dynamic linking; autonomous security
CLC number: TP311 **Document code:** A

1 Introduction

For convenience of general end- users, we introduce the software agent for the security system to make the system security transparent to end- users but still preserve the power of original cryptography. The security of Java agents mainly involves Java language[1] and secure agent model.

On the one hand, the traditional way of providing software- based protection within a program is by using abstract data types and information hiding. The main purpose of ADTs has been to protect programs from non- malicious mistakes made by other parts of the same software system. Since ADTs have seldom been used to provide robust security, programming languages often provide only ordinary support for them. While perhaps good enough for use in essentially benevolent environments, the Java package system s implementation of ADTs[2] leaves much to be desired. Java packages have limited ability to control access to their member classes, they don t have explicit interfaces, and don t support multiple views of modules. These characteristics make the Java package system deficient for modular programming tasks in which security is important; for example, for writing mobile applications.

With reference to Standard ML[3] and its associated Compilation Manager[4], extend the idea of module- level ADTs by providing the facility for structuring modules hierarchically. Lower levels in a module hierarchy can communicate across more expressive interfaces; higher levels can enforce more restrictive ones.

In addition, an extra problem confronts dynamically linked programs: a piece of code is designed to behave properly only when its unresolved symbols are matched against the particular set of external objects with which the programmer intended his module to be linked[5]. But since linking is often not under the control of the programmer who wrote the module- as in the Java virtual machine, for example, steps must be taken to ensure that after linking a program will behave in a manner consistent with the programmer s intentions. The typical way of ensuring safe linking is through

Received date: 2002- 06- 13
Foundation item: Supported by Chinese 1973 Programme(G1998030600); Natural Science Foundation of Fujian Province(F00013)
Biography: LIU Chuan-cai(1963-), male, was born in Linli County, Hunan Province, associate professor of College of Information Science and Technology, Fuzhou University, Doctor; research area is cryptography and pattern recognition.

type- check. This method guarantees that the types of symbols in the interfaces between modules match, but it does nothing else to ensure that the objects with which a program links will behave in the manner that the programmer expects.

On the other hand, agents are communication and cooperation entities in the agent systems[6]. We allow software agent to be an autonomous software program that provide services to act for its end- user and to interact or even to negotiate with other software agents for appropriate security policies. In fact, Gasser and Hewitt proposed the agent negotiation concepts in their early DAI research[7,8]. Software agent must negotiate on behalf of its end- user the security criteria that include the security policy adoption, the level of security capability, digital signature mechanism, and key exchange management, etc. These software agents use secure agent communication protocols to negotiate and communicate with each other. Therefore, the secure communication protocols are more flexible while executing by software agents.

In view of above consideration, we first make research on mechanisms for secure modular programming in Java, and we introduce a hierarchical module system similar to that of Standard ML that improves upon Java packages by providing explicit interfaces, multiple views of modules based on hierarchical nesting, and more flexible name- space management. Our solution to the problem of dynamic linking is to give the programmer more control over the linking process. Regardless of his inherent lack of knowledge about and control over the circumstances under which linking will occur, we wish to be able to guarantee in advance certain properties about the linking environment. Our module system facilitates such control by allowing the programmer to specify a key with which a foreign module must be digitally signed in order for linking to be allowed. The details of the linking process remain abstract to the programmer, and the linking specifications are simple and declarative. Second, we develop a secure agent model to discuss security issues. Finally, we propose a practical solution that addresses some of these security issues.

2　Research Objectives and Requirements for Security Agents

The aims of this research are shown as the followings.

(1) Introduce a hierarchical module system to improve upon Java packages, and give the programmer more control over the linking process.

(2) Propose a complete process for secure agent communication protocols, which including security policy negotiation, secure agent communication protocols set up, secure agent communication protocols operation, etc.

(3) Find out how the speech- act based secure agent conversation protocols can be embedded in the object- oriented Java programming language in the secure communication protocols.

(4) Set up speech- act based secure agent conversation protocols as finite state machine (FSMs) in object- oriented dynamic model.

Generally speaking, enormous secure communication protocols have already been proposed to tackle the security mechanism[9~12] in a traditional cryptology. Usually, the end- users are the primary principals to do the secure communication. Why we still introduce the agent concepts in this ready to run secure communication protocols? Some of the reasons are shown as the followings.

(1) Handle the complexity of cryptography protocols. The complexity of cryptography makes it nearly impossible for end- user to realize and make the best utilization of this technology. Agent system developer applies the agent ideas in the security protocols processing that will hide the complexity of security from end- users.

(2) Choose the right security criteria and parameters for end- user. There are several important security criteria proposed by cryptography algorithms but not all of them are required for each transaction. We propose the software agent to recommend or even autonomous user s most favorable viewpoint.

(3) Provide the flexibility of cryptography protocols. Different computer systems might use different approach to do their system security, network security. Initially, agents exchange information to negotiate their acceptable security poli-

cies then finalizes their security algorithm, criteria and protocols.

(4) Increase the efficiency of the security processing. Once end- user grants his authority to agent to autonomously process security mechanisms, this software agent must accomplish the mission in the most reliable and efficient way. We must ensure that the software agents do provide higher efficiency than general end- user in the security policy selection, security mechanism decision, and final security protocols setting and operation.

3 Secure Modular Programming in Java

3.1 Fixing Java Packages

In our scheme, we adopt the method of Bauer, Appel and Felten[1]. In reference [1], the syntax module system contains numbers of features that either are not present or are insufficiently developed in the Java package system. The most important are explicit export interfaces and membership lists, hierarchical scalability and multiple interfaces, and convenient name- space management. In addition to their value as software engineering tools, these are all instrumental in forming a base for developing secure software systems in Java.

Export interfaces and membership lists, a well- established principle of software engineering is that the interface of a module should be separated from its implementation. This enables a client of a module to be written and type- checked against the interface before the module s implementation is written. It also allows the module s implementation to be type- checked against the same interface to ensure that the implementation adheres to its own specification. This means that the implementation of the module and any of its clients can developed in parallel and modified independently of each other, separately type- checked and compiled, and later linked safely without further checking. Separating the interface from the implementation also aids in the construction of ADTs by making it clear which parts of the ADTs form its public interface and which should remain private.

Java supports modular programming at both the class level and the package level. At the class level, Java has some notable deficiencies: it is impossible to flag methods declared in interfaces as either final or static, which limits the degree to which its clients can take advantage of separate type- checking and compilation. Java classes are also too fine - grained a structure to be particularly suitable as units of modularity for traditional modular programming.

For this purpose, Java uses the package mechanism, which provides support for modularity above the class level. Java packages do not have explicitly specified interfaces. The interface of a package is implicitly specified by the access modifiers that are part of the class declarations of its member classes, i. e. , the interface is defined by the set of classes from that package that declared themselves public. Since it is defined by the component classes of a module, the interface is inseparable from the implementation. Such a scenario is clearly incompatible with the goals achieved by separating the interface from the implementation. The only way to specify the interface to a package is to write at least the skeletons of the implementations of the visible member classes. And because the implementation of the package defines the interface, there is no way to type- check an implementation against its own interface, so there is no way to ensure that the implementation matches its specification. As a vehicle for ADTs and separate compilation, therefore, the Java package system is sorely lacking.

Except for the traditional software engineering goals, module systems have recently been asked to fulfill additional roles as well. With the widespread use of mobile code (e. g. , applets, plugins) it has become necessary to protect systems from damage that malicious mobile code might inflict, as well as to provide environments in which mutually untrustworthy groups of mobile code can run simultaneously but without danger of unwanted interaction. If mobile code systems are to rely on modules to organize code, it is important for module systems to assist in providing the security functionality needed for mobile code, or at the very least not to interfere with other mechanisms used to provide security.

The Java package system is unsuited for this role. Because of the lack of explicit module interfaces and descriptions, it is inconvenient to use packages as units for enforcing security policies. The combination of implicit interfaces and the lack of explicit membership lists would make it easy for a malicious attacker to take advantage of a system for

running mobile code that based its security facilities on Java packages[13].

In our scheme, the module system prevents any such security breach by using module description files that explicitly specify both the memberships of a module and its public interface by listing all the classes that belong to each. Furthermore, our scheme would prevent a hostile applet such as the one described from even linking with the trusted application.

```
Module
        Graph
        Node
        NodeList
is
        Graph.java
        FlowGraph.java
        Node.java
        NodeList.java
        FlowNode.java
        GraphUtils.java
```

Figure 1　The Module Description File of a Sub-Module of a Register Allocator

The module description file in figure 1 demonstrates the use of explicit export interfaces and membership lists. Only classes defined in the listed source files are considered to be part of the module. The module defines several classes, but only Graph, Node, and NodeList are visible to clients outside the module.

Though a significant improvement from the standpoint of security and program organization, the interfaces of our module system don t address the issue of separate compilation. The interfaces are merely lists of classes and do not describe their types, so an implementation cannot be type-checked against them. In order to overcome this defect, our approach to organizing modules is similar to the mechanism for defining units in MzScheme[14], which does support separate compilation. But whereas the primary motivation in that work is extensibility and code reuse, we are concerned with the security aspects of modular programming.

Hierarchical scalability and multiple interfaces, the basic ways in which our modules support ADTs are dissimilar from those offered by Java packages. Java s module interfaces are implicit; ours are explicit, but our interface descriptions consist only of classes, and don t describe public fields and methods of classes that are also part of a full interface. Though our module system is not powerful enough to fully describe the types of modules, it makes it simpler to control and enforce the visibility of member classes. The interfaces of both systems have similar access control capabilities: a class can be either publicly visible or visible only to other classes inside the same module. The feature that sets our module system off from Java packages, however, is the ability to structure modules so as to provide different views to different clients.

Java s methods of controlling accessibility (through making classes and their fields private, protected, package-scope, or public) aren t expressive enough, so Java resorts to using a security manager to determine at run time whether a client is allowed to access a particular restricted class. The security manager suffers from a number of problems, from run-time overhead to the inability to interact with the programmer except interactively. Its complexity and ambiguities have made it vulnerable to security breaches and made it difficult to reason about and form security policy[15].

An elegant approach to the problem of multiple interfaces has been presented by research in hierarchical modularity. Hierarchical modularity is the idea of grouping several modules and attaching to such a group its own interface. The group is itself a module whose publicly visible members can be imported by other modules. The members of the group can communicate among themselves through their own interfaces, which can be much less restrictive than the group s top-level interface. This approach can be applied repeatedly to create a hierarchy of modules. For a comprehensive treatment of hierarchical modularity see Blume and Appel[4]. We use a similar approach for Java.

The natural way to use hierarchical modularity to provide different levels of access to different modules is to group together the modules that wish to share a high level of access with each other, and let them have appropriately unrestricted interfaces. The entire group can have a more restrictive interface that exports only those parts of its member s interface that ought to be available to the public. Our module system supports hierarchical modularity by allowing modules to explicitly list the sub-modules on which they depend. Modules can export not only classes which have been defined in their own source files, but also classes that have been defined in imported modules. When its module descrip-

tion file begins with the keyword library, compiling a module produces a JAR file that includes the byte- code of all the imported modules, which are then kept hidden by the export interface.

The modules that comprise a compiler, for example, are likely to need a high degree of access to each other. At the same time, we may wish to treat the entire compiler as a module that exports only a few of its classes. Figure 2, adopted from reference [1], is a module description file of the main module of a compiler, it illustrates this approach. The main module imports all the sub- modules that implement different parts of the compiler and defines only a few classes that tie the sub- modules together into a working system. The hierarchical structure is transparent to a user, he has no way of knowing that the compiler module is composed of sub- modules.

```
Library
        Main
is
        Main.java
        NullOutputStream.java
imports
        Codegen../Codegen/
        RegAlloc../RegAlloc/
        Absyn../Absyn/
        Tree../Tree/
        ...
        Types../Types/
        Util..Util/
```

Figure 2 The Module Description File of the Top- Level Module of a Compiler

Name- space management, extra software engineering benefit is our module system s flexible and convenient name- space management scheme. Although the naming convention used with Java packages suggests that they support a hierarchical naming scheme, packages with names like Java.awt and Java.awt.color have no more in common than packages with completely different names.

One of the reasons for grouping code into packages is to avoid name clashes between classes. But Java packages are themselves named so that merely lifts the problem to the package level. Instead of a name clash between two classes called Parser, we might have a clash between two classes called Util.Parser. The accepted way of solving this problem is to give package long, unique names. This isn t a particularly appealing solution, however, since it interferes with the packages system s ability to provide convenient name- space management; classes must now either be referred to individually using their cumbersome package name (e.g., java.awt.image.renderable.RenderableImage) or be imported entirely using the * notation, which again introduces the possibility of name clashes because the names of the imported classes are stripped of their unique package prefixes.

Our modules, on the other hand, are not named, so they don t suffer from this problem. Modules are assigned names only via import statements of individual module description files; this type of name- space thinning makes it easy to keep their names short and simple. In source code the names of external classes are always prefixed with the name of their module, so name clashes between classes with same names are easily avoided.

The name- space management scheme we use has been borrowed without much modification from the approach Blume and Appel have developed for Standard ML[4].

3.2 Secure Linking

The behavior of a program fragment depends not only on its own code but also on the libraries with which it is linked. Under the static linking model, compiling and linking a piece of code generates an executable that is fully self- contained. The libraries, with which the program is linked, as well as the finished product, are available for the programmer s perusal. He therefore has good reason to expect that the self- contained executable will behave in the desired manner, even if it is executed on a machine that has a different software environment and a different set of libraries.

Java adopts dynamic linking as a key feature[6]. But despite the proliferation of dynamic linking, only a few attempts have been made to extend the model of correctness that holds for statically linked code[15, 16]. Programmers believe that programs will behave in their intended manner even though much of the programs behavior depends on the system libraries of foreign and unknown systems. This belief is based mostly on the existence of standards that seek to ensure the uniformity of library code (e.g., all Java virtual machines and their associated system classes are expected to meet Sun s standard). There are very few guarantees, however, about adherence to a standard that are expressed in a way that programs can understand. The guarantees are largely verbal or written in English, and can t be reasoned about

or manipulated at the level of program code. Additionally, standardization does not apply when linking with third- party libraries. The only widely used method of ensuring safe linking, and the method used by Java, is type- checking the interfaces between program fragments. Recent research has desirable security property[7], and provided ways of ensuring that type- safety is preserved by the linking process. Still, though type- checking is useful in ensuring that programs and libraries at least agree on the types they are using, it falls far short of guaranteeing that code will behave in the expected manner.

Our module system makes headway on this issue by allowing the programmer to require certain properties of the modules on which his code depends. If the required properties are not present, the program won t link or execute. If they are present, the programmer can more realistically expect that his program, once linked, will behave in the desired manner. Furthermore, the programmer can annotate his own module with certain guarantees that are held to be valid once linking has succeeded. We thus establish a system in which a module can assert that if the modules it imports can guarantee certain behavioral properties, then it, too, will behave in a certain manner.

We implement annotation of properties through digital signatures. The JAR file of a module may be signed with one or more keys, each of which represents a property. The import statements of a module description file can specify the key alias of the key with which an imported module must be signed. One of the modules is a parser, the compiler wishes to advertise itself as unicode- friendly, but in order to make such a claim it must rely on the unicode- friendliness of its parsing module. Since the various module of the compiler might be dynamically linked at run time by the Java virtual machine, the top- level module of the compiler needs to be sure that it will be linked with a parsing module that has the appropriate functionality. An import statement in the module description file of the compiler specifies that the parsing module must be annotated with the unicode property. Linking will be allowed only if the parser s JAR file is signed with the key that corresponds to this property. The main module of the compiler can itself be signed with the same key, which makes it possible for the compiler s clients to require the compiler to have the unicode property.

Since a program will not execute unless it is convinced that its sub- components are usable, our approach complements traditional code signing well. Authorship can be regarded as just another property, and the author of a program may now actually be willing to be held responsible for the correct behavior of his code.

It should be noted that our use of explicit import interfaces somewhat restricts the flexibility of dynamic loading. In Java it is possible, at run time, to load classes whose names are unknown at compile time. Explicit import interfaces require the programmer to specify, prior to compilation, the locations of the modules on which his code depends. Though class names do not have to be specified in the import interface, the locations of the modules, at least, need to be known at compile time, which precludes some interesting uses of dynamic loading. This restriction isn t too limiting, however, since in most cases it should be possible to structure code so that even if the name of particular classes isn t known at compile time, the location of its module is.

3.3 Implementation

We use a prototype implementation derive from the literature[1]. Our main goal in designing the prototype implementation was to enable our system to be used easily with various existing Java compilers and virtual machines.

Our modules can be translated into Java packages. Some of the features of our module system, however, in a particular its ability to place various constraints on linking- cannot be expressed just using Java byte- code. Because of this, our prototype implementation needs to provide additional features both to the compiler and to the virtual machine.

4 Autonomous Security for Java Agent Systems

4.1 Security Mechanisms in Java- based Agent Systems

Mobile agents move from one machine to another and can execute on each of them. A major security problem in a network- oriented environment is that neither the agent nor the machines are necessarily trustworthy. The agent might try to harm the machine and gain access to local resources. The machines might try to harm the agent or access its pri-

vate information and resources. Either the machine or the agent may be malicious or badly programmed. However, this distinction is not of primary concern because the final effect can be the same. Security is perhaps the most critical issue in mobile- agent system.

In the current agent systems, several different approaches are used to address these problems. There is a consensus that mechanisms should be provided to keep these machines from being harmed by the agents as well as to protect the agents from these machines. However, only a few systems implement some protection of the agents from the machines.

Security in Java- based agents is also a major concern. In this system, the server wants to be protected from an incoming malicious agent. On the other hand, the agent wants to have its information protected while it is traveling from one machine to another. Each place in the system might have its own policies while each engine has an overall policy.

Two concepts are fundamental to the understanding of this system: safety and security. The term safety refers to features that mainly promote robustness and prevent accidents. The term security, on the other hand, refers to features that are intended to provide protection and integrity in the presence of malicious users. These security features protect agents and places from each other.

Every agent is uniquely identified by a telename which consists of two components: an authority and an identity. The authority identifies the owner of the agent. The identity distinguishes an agent from another agent of the same authority. The authority component is cryptographically generated and cannot be forged[17].

Each agent has a permit, which limits its capability and the resource consumption. In this way, agents and places can be protected from malicious or badly programmed agents. Two kinds of capabilities are granted an agent by its permit. The first kind is the right to execute certain commands. The second is the right to use a particular resource and by which amount. An agent s permit is granted when the agent is first created and is renegotiated whenever that agent migrates to another place with a different administrative authority.

Besides access control, secure channels are provided to support agent mobility in a distributed process environment. These channels provide an authenticated opaque pipe , normally created using cryptography[18]. Depending on the specific application, different levels of security can be provided. If authentication is required, strong mutual authentication using RSA public key encryption, session key negotiation, and session encryption is used[18].

4.2 Embed Security Services in Java Agent

There are several Java agent systems existing in the Internet but not all of them provide speech- act message communication services. Searle proposed speech- act concepts in early 70 s[19, 20]. These speech- act communication action messages can be expressed in terms of illocutionary logic[21]. In order to enable the software agents to be an autonomous conversation entities, use speech- act message for interactive agents is the major approach for most the agent researchers[22~24].

Then, we show what are security features in the Java JDK (Java Development Toolkit) environment. Since our Java agents provide more dynamic and flexible security services during their cooperative conversation. So we also explain why our secure Java agent communication protocols are more versatile when compared with existing security communication protocols, such as SSL (Secure Socket Layer).

4.3 A Model for Agent Based Java

In this section, we will introduce a generalized ASE (Agent Support Environment) model by providing many of its features. This model will then be used to explore the main security problems in agent based on a classification presented in reference [25].

4.3.1 The Model The ASE system, as one can notice from the different implementations that were presented, needs to support creation, execution, resource access, migration, communication, Java language support and additional services. For Java language support, it involves the issue of interpreted versus compiled languages. It also involves support for just one specialized language versus the support of many different languages to be used in programming agents. As concerns additional services, such services like authentication, name service, check pointing, as well as other system built- ins.

The issue here is that some of these services can be implemented through the system while others are easier to implement using agents.

Note that all of the above mentioned features can help define an agent architecture and thus provide a framework within which all current research can be viewed and security issues can be discussed. The following is a description of the security issues associated with these different features of an agent- based system.

4. 3. 2 Security in Agent Based Java　Indubitably, security is the most important issue on which the applicability of agent systems rides[25~28]. No matter what the features of an implementation are, if they can t provide an adequate security model for all the issues involved, then this implementation will definitely fail. There has even been a suggestion towards the need for security profiles in agent systems to see if any given system adequately addresses all the issues concerned[29].

In current operating systems, the system resources are the most important parts of the system and thus should be protected from malicious use. It is acceptable that, in such a system, the solution for the security problem is based on splitting the domain into a user space and a kernel space. This is not the case in an agent system where the user has a vested interest in the agent and part of its resources that the system should not have control over.

4. 3. 3 A Practical Solution　Most of the research about security on Agent Based Computing focuses on the protect the machine from agents and protect an agent from other agents problems[26,30,31]. Moreover, those works that address the other secure agent- based computing issues either intend to solve the problem but don t have a real solution yet[25] or just mention that these problems are important and need to be solved.

Here, we propose a solution that also addresses the protect the agent from the machine issue, which is a very important concern from the user s point- of view. One major design is to conceive a practical solution using current technology. Therefore, our solution is both simple to understand and to implement, and does not depend upon a net- wide service, like a secure public key distribution system. The PEM acceptance problem (Privacy- Enhanced Mail[32]) has shown that this kind of new distributed net- wide directory infrastructure turns into a barrier to the adoption of the solutions based on it[33]. Note that if such an infrastructure became available, the rational for this choice changes completely. But we don t expect this to be the case in the next few years.

Another important design decision is that an agent not concerned about security should incur any security overhead. In other words, our goal is to design a flexible solution that does not require the use of the security mechanisms by the agents that don t need them. Thus, all the schemes described here are to be added to the traditional features found in an ASE.

Our approach to the protect the machine from agents problem is the common one: the agent runs in a restricted environment and all accesses to resources have to pass through a security monitor. This monitor decides if the agent is interpreted. For example, the Java Agent provides a framework to implement this kind of monitor.

The problem resides in deciding whether a particular agent has access to a given resource. An easy solution is defining the same rights to all agents (as Java does[30]). However, this is a very restrictive solution. A better approach would be to grant different accesses depending on who owns the agent. Since we have decided not to use a distributed authentication service, we have to rely on passwords to securely identify the agent owner. Protocols that safely negotiate cryptographic keys (like SSL[34]) must be used to avoid the discovery of passwords by monitoring the network. Default access rights can be supplied for unknown users. Of course, a group of hosts can share the same password database, but our solution doesn t require this nor does it define how this can be done. We call this step the authentication phase.

In our scheme, we choose a conventional Access Control List (ACL) mechanism to decide if an agent can or cannot access a resource, given that the system knows who owns the agent.

It is clearly impossible to protect the agent from the machine on which it runs. Agent executes when and if its host machine wants. An agent needs to care about this question before it moves into a machine. Our solution requires that the machine send a counter- password to the agent during the authentication phase. This means that both agent and target

machine need to know a triple (user, password, and counter- password) and allows the agent to determine if it is moving to a known machine.

In spite of that, an agent can still have some critical information stolen by a known machine. The shopping around agent is a perfect example of that situation. One of the digital stores the agent visits can steal the money being carried by the agent. To avoid that, we require that an agent concerned about security return back to its home (and so, completely secure) site after it finishes its job at the target site. Additionally, an agent is to carry only the resources and information it may need into the destination host. Observe that this implies that a secure agent cannot move about; it has to do a single round- trip. However, the same kind of service can still be provided because an agent can travel again after its returns to its home site. Actually, it has to be modified to carry useful information in the new destination, but this also doesn t limit the development of any type of agent based application. We call such a specialized, restricted agent a minimal agent.

The protect an agent from other agents problems can be seen as a special case of the protect the machine from the agents problem, except for the communication issues. One could think of using the same password plus ACL solution to cope with the secure communication issue. But, because any message could be intercepted by the host which is running the agent (even if the message is encrypted, the host can wait until the agent decrypts it), this is not worthwhile. Thus, we don t provide a special mechanism to make message- based communication more secure. If an agent needs this kind of service, it should migrate to the destination host (returning first to the home host) and communicate locally.

Protect a group of machines from an agent is a very difficult problem because even if the resource- hoarding agent is detected and killed, the owner of that agent can send a new instance of it again. Consequently, it is necessary to determine who the owner (or at least, the originator machine) of the resource gathering agent is and log this information in order to figure out who is creating this kind of malicious agent. The problem is how to know who the owner of a particular agent is or from which machine it has been sent. Because, in our solution, we can determine this just for the agents that are using the services proposed here, there is no way to solve this problem for the general case of completely mobile agents.

The currency- based resource- allocation scheme suggested by Gray doesn t solve this problem in all cases either. In many cases, the machines on which an agent can run don t want to ask for some form of electronic currency because it is in the machine s best interest that the agent run and obtain as much information as it wants. The perfect example is a digital shopping center. In these cases, currency schemes don t really work.

5 Conclusions

With the development of computer networks, software agents are perhaps a very useful approach in building a large set of network applications. However, the security problems that can be raised by supporting agents can prevent the wide use of agent based applications. Moreover, distributed security concerns tend to become more important as we begin to use open computer networks to transfer information of more direct economic value. There, a secure way to use agents is fundamental to make viable their application in public networks, like the Internet.

The security of Java agent mainly involves two problems: one is Java language itself, the other is secure agent model. Therefore, we first introduce a new module system to Java that improves upon many of the deficiencies of the Java package system and gives the programmer more control over dynamic linking. Second, we develop a general agent model and discuss general security issues in that model. Finally, we propose a practical solution that addresses some of those security issues.

However, the authentication does not guarantee that the machine will not attack the agent. In order to reduce the damage that such kind of attack could incur, we introduce the concept of minimal agent, which carries only the information it may need into the host and always returns back to its safe home host when its job is done.

It is obvious that our solution involves some restrictions in the way agents can be used. Nevertheless, any application that can be built in the standard insecure agent model,

The agent security problems are very hard. An appealing alternative way to address these problems is the soft security approach[27,28]. Soft security means that privileges are granted, as they are needed, with the current risks taken into consideration. As opposed to soft security, hard (i.e., traditional) security uses methods that don t reevaluate granted privileges.

References:

[1] BAUER LUJO, APPLE ADREW W, FELTEN EDWARD W. Mechanisms for Secure Modular Programming in Java[R]. Technical Report: TR-603-99, Department of Computer Science, Princeton University, 1999.

[2] GOSLING JAMES, JOY BILL, STEELE GUY. The Java Language Specification, the Java Series[M]. Reading, Massachusetts: Addison-Wesley, 1996.

[3] MILNER ROBIN, TOFTE MADS, HARPER ROBERT. The Definition of Standard ML[M]. Cambridge, MA: MIT Press, 1990.

[4] BLUME MATTHIAS, APPLE ANDREW. Hierarchical Modularity[M]. ACM Transactions on Programming Language and Systems, 1999.

[5] CARDELLI LUCA. Program Fragments, Linking, and Modularization[A]. 24th ACM SIGPLAN-SIGACT Symposium on the Principle of Programming Languages[C]. Baltimore, 1997. 266-277.

[6] LINDHOLM TIM, YELLIN FRANK. The Java Virtual Machine Specification[M]. Reading, Massachusetts: Addison-Wesley, 1997.

[7] LEROY XAVIER, ROUAIX FRANCOIS. Security Properties of Typed Applets[A]. Conference Record of POPL 98: The 25 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages[C]. San Diego, California, 1998. 391-403.

[8] SUN MICROSYSTEMS. Java Core Reflection[EB/OL]. http://www.java.sun.com/products/jdk/1.2/docs/guide/reflection/spec/java-reflection.doc.html, 1998.

[9] GRAFINKEL, SIMSON, GENE SPAFFORD. Practical UNIX & Internet Security[R]. O Reilly & Associates, Inc., 1996.

[10] KAUFMAN CHARLIE, PERMAN RADIA, SPECINER MIKE. Network Security Private Communication in a Public Worlds[M]. Englewood Cliffs, NJ: Prentice Hall, 1995.

[11] PFLEEGER, CHARLES P. Security in Computing(2nd Edition)[M]. Englewood Cliffs, NJ: Prentice Hall, 1997.

[12] SCHNEIER, BRUCE. Applied Cryptography(2nd Edition)[M]. San Francisco, US: John Wiley & Sons, 1996.

[13] DEAN DREW, FELTEN EDWARD W, WALLACH DAN S, etc. Java Security: Web Browsers and Beyond[A]. DOROTHY E DENNING, PETER J DENNING. Internet Beseiged: Countering Cyberspace Scofflaws[C]. ACM Press, 1997.

[14] GREG NELSON. Systems Programming With Modula-3[M]. Englewood Cliffs, New Jersey: Prentice Hall Series in Innovative Technology, Prentice Hall, 1991.

[15] DEAN DREW. The Security of Static. Typing with Dynamic Linking[A]. Fourth ACM Conference on Computer and Communications Security[C]. Zurich, Switzerland, 1997.

[16] DEAN RICHARD DREWS. Formal Aspects of Mobile Code Security[D]. Princeton University, 1999.

[17] LUIS VALENTE. Safety in Telescript[EB/OL]. http://www.catless.ncl.ac.uk/Risks/15.39.html# subj6, 1994.

[18] GENERAL MAGIC. An Introduction to Safety and Security in Telescript[EB/OL]. http://www.genmagic.com/Telescript/security.html, 1995.

[19] SEARLE, JOHN R. Speech Acts[M]. Cambridge, UK: Cambridge University Press, 1969.

[20] SEARLE, JOHN R. The Philosophy of Language[M]. Oxford, UK: Oxford University Press, 1971.

[21] SEARLE, JOHN R, VANDERVEKEN DANIEL. Foundations of Illocutionary Logic[M]. Cambridge, UK: Cambridge University Press, 1985.

[22] CHANG M K, WOO C C. A Speech-Act-Based Negotiation Protocol: Design, Implementation, and Test Use[J]. ACM Transaction on Information Systems, 1994, 12(4): 360-382.

[23] FININ T. KQML as an Agent Communication Language[A]. The Proceedings of the Third International Conference on Information and Knowledge Management[C]. ACM Press, 1994.

[24] LABROU Y, FININ TIM. A Semantics Approach for KQML-A General Purpose Communication Language for Software Agents[A]. The Proc. of the Third International Conference on Information and Knowledge Management[C]. ACM Press, 1994.

[25] ROBERT S GRAY. Agent Tcl: A Flexible and Secure Mobile- agent System. Proc. of the Fourth Annual Tcl/Tk Workshop (TCL 96) [EB/OL]. http://www.cs.dartmouth.edu/agents/papers/tcl96.ps.Z, 1996.

[26] OUSTERHOUT JOHN, LEVY JACOB, WELCH BRENT. The Safe- Tcl Security Model[M]. Sun Microsystems Laboratories, 1996.

[27] RASMUSSON ANDREAS, JANSSON SVERKER. Personal Security Assistance for Secure Internet Commerce. New Security Paradigms 96 Workshop[EB/OL]. http://www.sics.se/ara/doc/NSP/NSP.html, 1996.

[28] RASMUSSON LARS, JANSSON SVERKER. Simulated Social Control for Secure Internet Commerce. New Security Paradigms 96 Workshop[EB/OL]. http://www.sics.se/lra/nsp96/nsp96.html, 1996.

[29] BROWNE SHIRLEY. Need for a Security Profile for Agent Execution Environments CIKM 95 Workshop on Intelligent Information Agents[EB/OL]. http://www.cs.umbc.edu/%7ecikm/iia/submitted/viewing/browne.html, 1995.

[30] SUN MICROSYSTEMS. The Java Language: An overview[EB/OL]. ftp://ftp.javasoft.com/docs/java- overview.ps, 1995.

[31] JOHANSEN DAG, RENESSE ROBERT VAN, SCHEIDNER FRED. An Introduction to the TACOMA Distributed System: Version 1.0. Technical Report: 95- 23, Department of Computer Science, University of Tromso[EB/OL]. http://www.cs.uit.no/Lokalt/Rapporter/Reports/9523.html, 1995.

[32] LINN J, KENT S, BALENSON D, etc. Privacy Enhancement for Internet Electronic Mail: Parts I- IV. Internet RFC 1421- 1424 [EB/OL]. ftp://ftp.internic.net/rfc/rfc142[1- 4].txt, 1993.

[33] BRADEN R, CLARK D, CROCKER S, etc. Report of IAB Workshop on Security in the Internet Architecture. Internet RFC 1636 [EB/OL]. ftp://ftp.internic.net/rfc/rfc1636.txt, 1994.

[34] FREIER ALAN O, KARLTON PHILIP, KOCHER OAUL C. The SSL Protocol: Version 3.0. Internet Draft[EB/OL]. http://www.home.netscape.com/eng/ssl3/ssl- toc.html, 1996.

基于 Java 的 Agents 的安全问题

刘传才

(福州大学信息科学与技术学院, 福建 福州 350002)

摘 要: 在分布式系统中, 随着计算机网络的普及, 编程方式也发生了变化, 即由远程调用方式转变为远距编程方式, 这种变化可减少网络流量、提高系统性能. 软件 agents 适合完成这项任务. 使用 agents 能带来多种好处, 且不利方面很少, 这包括附加的安全问题. 新的组件系统能改进 Java 套装软件系统的多个缺陷, 使编程者能获得更多的对动态连接的控制权, 实现 Java- agent 的自主安全. agent 模型中出现的一般安全问题, 可通过一种实用的方法加以解决.

关键词: 软件 agents; 远程调用; 动态连接; 自主安全

中图分类号: TP311 文献标识码: A