

Personal Security Assistance for Secure Internet Commerce * (position paper)

Andreas Rasmusson Sverker Jansson
Swedish Institute of Computer Science
Box 1263, S-164 28 Kista, Sweden
`ara@sics.se`, `sverker@sics.se`

16 September, 1996

Available as dvi, ps, html, at <http://www.sics.se/~ara/papers/NSP96.html>.

Abstract

In this paper we discuss the approach of using a personal security assistant for interacting with mobile agents visiting your computer.

Current agent security approaches are often based on trust in an external authority who guarantees that the agent is correct/benign or that your local resources have all been assigned correct access-restrictions. We argue that a more rewarding security policy is to grant the visiting agent access to resources on the assumption that it will do useful work for you and behave as expected.

Not disqualifying agents from doing useful work for you on the grounds that you have no previous experience from them facilitates the introduction of new agents into the market since trusting the sender is less crucial.

The paper contains a discussion on the security approach taken in most of today's agent systems and how security is enforced by Intrusion Detection Systems. We give a rationale for using an interactive Personal Security Assistant as an aid for detecting malicious agents visiting end-user agent environments and sketch the architecture and design criteria of such an assistant. We discuss how malicious programs could be identified and mention some preliminary experiments with Java-applets.

*Submitted to NEW SECURITY PARADIGMS '96 Workshop, Lake Arrowhead, CA, September 16-19, 1996. This work was supported by a grant from NUTEK, the Swedish National Board for Industrial and Technical Development.

1 Introduction

Why would someone even *want* to interact with a piece of code for which nothing can be absolutely guaranteed? The answer is, obviously, that the person hopes that the program will be useful to him and that it will not do any harm. As long as he has the possibility to supervise what is happening he will be prepared to take the chance and let it access his computer.

We want to manage the security and still let untrusted code do useful work. An untrusted program is not denied access to resources unless it misbehaves in some way. With this approach well-behaved programs would be allowed to do their job and the malicious programs would be detected and stopped. One could say that the expected usefulness of the program is considered when access granting decisions are made.

This approach might be called *soft* security. This is meant to distinguish it from the *hard* security methods where the trustworthiness or appropriate privileges is decided before the program is allowed to execute. Soft security system is an approach to take into consideration that many traditional security requirements are infeasible in reality and that a *risk* is always involved when granting privileges to or interacting with an untrusted party [13, 14, 16].

We intend to investigate how a personal security assistant can help the user understand what visiting agents are up to at his local computer. The assistant monitors the visiting agents as they execute and the user is given information and advice about whether the agent behaves as expected or attempts to do something the user would not want.

Soft security is very much related to what is usually termed reactive, or “after-the-fact”, security (reactive as opposed to proactive)[3]. The term reactive emphasizes the *when* the analysis is done whereas *soft* is an indication of on what grounds resource accesses are granted, hence the new term.

This approach draws a lot from the work done in intrusion detection in computer systems [11]. Intrusion detection concerns finding activities that indicate that someone is using the computer in an illegal way. This could be someone from outside, breaking in to the computer, but it could also be an authorized user misusing his privileges. If the user normally does one set of tasks, deviations in his behavior can signal that a crime might be in the making.

Below we discuss the security approach taken in most of today's agent systems and how security is enforced by Intrusion Detection Systems. We give a rationale for using an interactive Personal Security Assistant as an aid for detecting malicious agents visiting end-user agent environments and sketch the architecture and design criteria of such an assistant. We discuss how malicious programs could be identified and mention some preliminary experiments with Java-applets.

2 Related work

2.1 Agent environments

Most of today's approaches agent programming platforms [18], [6], (see also [2]) base their security on methods like access-control lists, trusted interpreters and/or known business partners who provide digitally signed agents or accept agents from you.

An agent who is not trusted will be prohibited from accessing any security-relevant resource (e.g reading files). This constraint can be relieved by the user explicitly making the resources available to any agent or agents from certain trusted locations.

However, the more substantial functionality that is put into the agents the more resources will they need to access or, rather, will *the user* want them to access. Security policies based on access control lists will tend to grant the agents no access, unrestricted access or the user will have to do a lot of tiresome classification and security assessments of his resources. Also, classifying resources according to how private they are is hard. How secret a piece of information is often depends on by whom it is accessed. If the size of your salary is queried for by your home-budget program you'd probably be less reluctant to reveal it than if it was asked for by a car sales agent.

Digital signatures should be used to authenticate the sender of the agent and/or guarantee that the agent hasn't been tampered with. The agents maliciousness, whether deliberate or due to bugs, cannot be decided by any level of cryptography.

If possible we want to avoid having to base our security on trust in external code or organizations *and* having to have a rigid security structure. Since agent security is a new field of research there is little experience with these things in the agent community. Rather than from existing agent environments, our inspiration is drawn from the field of intrusion detection.

2.2 Intrusion detection systems

Intrusion detection systems (IDS) can be seen as a complement to ordinary proactive security policies. Instead of enforcing any particular policy, they are tools for analyzing the audit trails to find out if an intrusion occurred (i.e whether the ordinary security was compromised).

IDSs are used to detect *anomalies* in user behavior or *misuse* of a computer. Anomalous behavior is behavior that deviates from a "normal use" pattern, e.g accessing a certain number of files per minute. Misuse means that weaknesses or flaws in the security system are deliberately used to get unauthorized access to system resources [9].

Anomalous behavior can be detected by statistical methods or different AI methods, e.g [7] and [3] whereas misuse is usually detected with specific rules or expert systems e.g [8].

The focus in intrusion detection has traditionally been on detecting *user* misbehavior. Detecting malicious programs and classifying programs according to their behavior has so far not gained as much attention. Steps in this direction have been taken by NIDES [1] and MCF [10].

MCF analyses the code prior to execution and looks for *tell-tale signs* that indicate malicious behavior whereas the NIDES experiment tried to distinguish between well known programs by comparing their audit trails to previously trained execution profiles.

The emphasis in our approach is similar to NIDES; to find deviations from an expected behavior pattern. However, we also want to give the user support to decide whether an activity might be *harmful* and should be terminated.

Classifying programs according to their behavior is not at all an easy task, especially as the complexity of the programs grow. The behavior of your mail-program is intuitively pretty straightforward, but generalizing this behavior (without over-generalization) to more than one mail program will probably require a mix of (learnt or coded) rules and some heuristics.

Important is also that the security system should show *graceful degradation*. Even if parts of the security system are tricked by an agent, other parts should signal for suspicious activities. There shouldn't be any ways to obtain unrestricted access by attacking well known weaknesses of the security system.

The security assistant could be made out of replaceable modules where each module uses a different although maybe somewhat redundant security policy. If these modules are continuously added or replaced, writing a program that avoids all possible detection aspects gets increasingly difficult. Even if it worked yesterday it might not work today.

If the computers on the Internet have partly differing methods for detecting intrusions the chances that one agent could attack large groups of computers without being detected would be small.

In a sense this is *security by obscurity*. If it's too costly, compared to what is gained, to create an agent that gets away with malicious activities, these will not be written¹.

These ideas are much inspired by the COAST-project's ideas of security similar to the human immune system [3].

2.3 Soft Security

A system-wide approach to soft security is taken by [15]. Large populations of agents interacting with each other are simulated. Distinguishing is the assumption that not all of these can be trusted and cooperation with different agents give different net return. By utilizing reputation mechanisms and other

¹Naturally, such agents will, and should, be written to demonstrate security weaknesses on certain hosts. The important point is that the economical incentive to write such agents is decreased since the attacks won't be generally applicable and will require a lot of inside knowledge of the system that is to be attacked.

social behavior patterns the agents should learn to close deals preferably with 'benign' agents. The guiding idea is that the system should not demand the agents to prove that they are honest, bug-free, certified by a central authority etc. prior to allowing them to enter the system. Only if they really *are* malicious, inefficient etc. will they be removed from the agent society.

3 Personal Security Assistant

Agent and electronic commerce security is today a field of study without real empirical data. This data will not come, however, until sufficient assurance can be given on how secure the agent environment is. The agent environments will probably differ a great deal from current UNIX operating systems environments and the traditional intrusion problems with set-uid and sendmail will probably bear little resemblance to what the agents will do to get data they are not supposed to get access to. Malicious agents will probably most often use Trojan horse-like attacks.

Instead of making up rules for what agents are allowed to do and how they are allowed to interact with the rest of the system, the user should be given information about what the agent is doing. In a dialogue with a personal security assistant the user can decide whether what the agent does is illegal or not.

We believe that that monitoring agents and comparing their actual behavior with what the user expects the agent to do could be a way of tackling agent security.

An agent wishing to execute at a computer will often have a known role where some resources are needed and others are not. An editor rarely reads system-files or binaries, and a compiler usually doesn't read your mail. If the user makes an estimate of what the agent is or what it might want to do, the agent can be granted access to only those resources *usually* needed to accomplish its tasks.

The security assistant makes the user aware of when the agent deviates from its expected behavior and tries to provide enough information for the user to understand what the agent might be up to. If the assistant knows what the user can accept from the agent, less information will have to be presented to the user.

What is needed is a balance between how many decisions the user has to make and how well protected he is. If the user can classify programs by what *kind* of program they are, rather than by which resources they should be granted, the demands on what the users need to know about resource consumption etc should decrease.

4 An open assistant architecture

The architecture should be as open as possible to allow for experimentation and evolution of the assistant. This section describes the initial design and the design considerations.

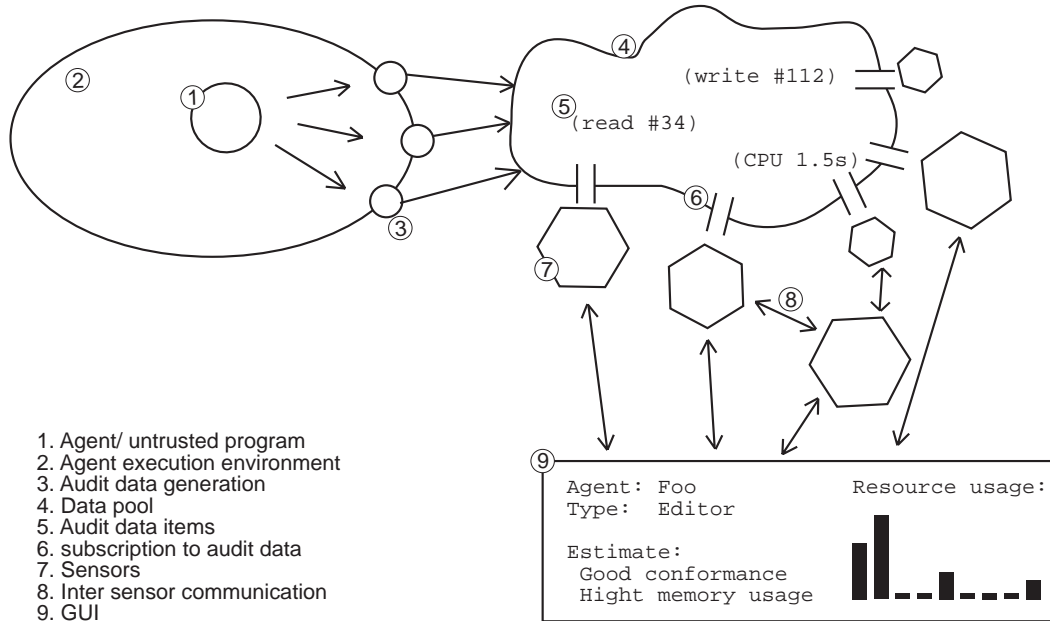


Figure 1: Schematic picture of the Personal Security Assistant architecture.

When the agent/program process (1) accesses resources from within its execution environment (2), audit data is generated (3) and delivered to an unstructured pool (4) of audit data (5). The audit data gets delivered to those sensors (7) that have announced their interest (6) in particular types of data. The sensors can aggregate data, and detect trends and deviations in the audit data. More complicated sensors might exchange information (8) amongst each other to reduce the redundancy in the analysis work. The sensors are given opportunity to present their findings on a GUI (9). If the audit data is analyzed before the the process is allowed to resume its execution, the user is can intercept and interrupt the agent's actions. This will, however, probably make the execution less smooth, from the user's point of view. Perhaps it is often sufficient to understand, after-the-fact, that the agent misbehaved so that the word can be spread to other users and that the program won't be used again.

As agents/programs we have considered Java-applets and ordinary UNIX programs. The agents/programs are executed in an environment from which audit data can be drawn. This could be an interpreter like the Java Virtual

Machine or a UNIX environment where an audit package like the Sun BSM is used. Initially, SimICS, an instruction-set simulator for Solaris, will be used as execution environment.

If SimICS is used, runtime analysis can be done, breakpoints can be set at optional places in the programs without recompilation and very fine-grained information can be extracted. This gives flexibility and increases the set of programs available for experimentation. On a small computer it's not likely that completely emulating the agents execution is feasible. In the exploratory phase, however, this is not of primary concern.

The execution environment is adapted to leave audit data in a predetermined data format. This is modeled as a set of audit data generators transforming the raw audit output from the execution environment into the standard audit data format. This should be the only part that needs to be rewritten if an other execution environment is to be used.

When monitoring programs two important problems are how to handle the large amounts of audit data available and what to look for in the audit trail (data reduction and feature selection). Since our main focus is on highlighting potential misbehavior of the agents, these problems are not addressed in the initial design. See for instance [4] for a survey of feature selection techniques.

The audit data is gathered in a pool (or blackboard) from which it is made available to interested sensors. The sensors might implement different intrusion detection strategies ranging from pure rule based approaches to fuzzier methods like genetic- or neural net algorithms for finding patterns in the audit data. Since experiments with different aggregation and deduction methods will be performed on the audit data it should be easy to add and replace sensors.

If the sensors are able to communicate their findings among each other complex sensors could share analysis work and reduce the overall system overhead. KQML[5], or parts of it, will be explored as a communication language among the sensors. The use of a language for communicating what data a sensor needs reduces the need for predefined hierarchies or dependencies between the sensors. A sensor might implement the full functionality it needs, but use the services of others when possible by asking the other sensors to send it data they have aggregated.

Language based communication and cooperation between sensors generalizes naturally to communication between Security Assistants. Information on known malicious agents, good heuristics and detection strategies can be traded between sites, hence an agent that has been classified as malicious at one site would be treated with more suspicion by the other sites².

Visualization and exploration of the audit data will be important for presenting what is happening. TCL/TK-interface is used to let the sensors draw themselves. The sensors are individually responsible for their appearance.

²Of course the other sites accountability will then have to be considered. Do other sites gain from revealing their findings?

5 Recognizing behavior

What can be said about an agent's behavior?

As well as people, programs engage in particular activities on the computer. It has been shown that a profile can be made to distinguish programs from each other based on their audit trails. NIDES [1] used a statistical model to distinguish the behavior of different UNIX commands with promising results. This is encouraging for continued experimentation with statistical as well as other classifying methods.

The measures drawn from the audit data that comprise a profile used by NIDES fall into four basic categories [7]:

- Activity Intensity Measures – which indicate bursts of behavior such as sudden bursts of generation of I/O audit records.
- Ordinal or Counting measures – which measure some quantity, for example number of files read.
- Categorical Measures – which measure the relative frequency of a category compared to other categories. One measure might be with what frequency some directories are visited relative to others.
- Audit Record Distribution – which basically is a categorical measure of the distribution of the entire set of audit records.

These measures can be compared to the expected profile for the agent. Since the user picks a category for the agent, the assistant can make statements about the agent in spite of the lack of historical data for this particular agent.

Some other approaches from the intrusion detection branch of computer security for recognizing agent behavior are; matching behavior to known dangerous patterns and comparing behavior with well-known, approved behaviors.

Matching behavior to known dangerous patterns is needed to hinder direct attacks on known vulnerabilities. These patterns are likely to be implementation-specific. Patches for detecting these particular attacks have to be distributed as they are detected.

When an agent's behavior is compared to well-known approved behaviors, patterns are stored that describe different approved actions. As long as the agent's behavior conforms to one of these it is deemed benign.

It should be possible to experiment with these methods as well, since the sensors can be arbitrarily complex. This is, however, not of first priority.

5.1 Detecting malicious behavior

Detection of a maliciously behaving agent would typically consist of the following events:

An agent arrives to a computer, either because it was requested by that computer or on the initiative of the agent, asking that computer to receive it.

If the user knows what the agent is supposed to do, e.g. present some product information based on his customer profile from some company, he tells the security assistant that he thinks that the agent is a 'normal advertising agent' and what company it comes from. This sets the thresholds for the different sensors looking at the audit data.

When the agent reads the users customer profile, the security assistant will not complain, since this is an action that the user would probably want the agent to do.

If the agent tries to access a file not associated with this company, the sensors will observe the departure from normal 'advertising agent behavior' and the security assistant will alert the user of this unexpected behavior. If the user understands why the agent does this or is willing to let the agent continue with its business, he can grant access to the agent. If not, the agent will be stopped.

6 Monitoring Java Applets

6.1 Experiment

To get some hands on experience with what information is needed to predict an agent's maliciousness Java's SecurityManager[17] was modified to leave audit data and show access statistics for different properties. The SecurityManager is implemented as an object whose methods are called when security considerations need to be made, e.g. when a socket or file is to be opened. The system libraries are wrapped inside procedures who first call the SecurityManager and then the library function. The SecurityManager can examine the stack for the class types and determine whether it should throw an exception or not.

The SecurityManager is quite flexible in what it can do to determine whether or not access to the resource should be granted. However, since the SecurityManager is also part of the Java environment it cannot, without using a modified Java interpreter, break the restrictions on what it can see in the objects it wishes to examine.

Although obtaining audit data from the SecurityManager is straightforward, the main problem with applets today is that they very rarely use any local system resources. Most applets so far are graphical animations that only draw on the screen, others might also pass user input back to the remote WWW-server to query a database or proxy the users input to other applets (there are quite a few IRC³-like applets available...).

³Internet Relay Chat

6.2 Experiences

Some limitations in the current version of Java (JDK1.0) concerning what audit data can be extracted from the appletviewer could be noted. These are most likely deliberate design decisions to enable Java programs to run on smaller CPUs, but makes Java less satisfactory for unconstrained experimentation.

First, since memory management is part of the language it is not possible to gather data concerning each applet's memory consumption. Since the garbage collection is automatic, it is not possible to prevent an applet from keeping a reference to an object thereby hindering it from being removed. Secondly, it's not possible to place limits on or to measure the applet CPU usage.

The current implementation of the appletviewer also has some unpleasant artifacts. The Java security approach is more focused on hard security. If the agent is allowed to create a file, further accesses to it are not auditable from the SecurityManager. Once allowed to open a socket to its source URL it will also continuously be allowed to use this connection.

In the appletviewer all code from the same base URL is considered as belonging to the same ClassLoader. This means that two distinct applets might be loaded with the same ClassLoader. In this case their identities will be intertwined in an unwanted fashion.

7 Monitoring SimICS processes

Instead of waiting for more complex applets to become available, we will look at the behavior of ordinary programs available today. We believe that useful experience can be gained from studying common UNIX utility programs whose behavior is simple enough to grasp and complex enough to leave a sufficiently rich audit trail.

As Java applets get more complex and more dependent on accessing system resources it will be interesting to do more experiments. Right now, however, experiments will have to be done on existing UNIX utilities.

SimICS is a system-level instruction-set simulator for Solaris 2.x (Sparc) platforms[12]. It can be used to extract virtually any characteristic from an unmodified Solaris binary. Since it simulates the program instruction by instruction in the non-kernel code, breakpoints can be set freely in program code and library code. Hopefully this will allow a very free exploration of programs and their characteristics.

8 Conclusions and future work

After some preliminary experiments with Java applets work continues with design and implementation of a framework for soft security analysis of programs. Issues to be investigated are how to detect when a program behaves in

an unwanted way and how to design modular cooperative sensors for analysis of the audit data.

The need for soft security stems from the wish to let untrusted code access local resources in a controlled way without overwhelming the user with decisions on what should be allowed.

Since there aren't many agents available yet, experiments will be performed with standard UNIX utilities. These have non-trivial behavior and all sufficient data can be extracted by instruction-set simulations using SimICS.

References

- [1] D. Anderson, T. F. Lunt, H. S. Javitz, A. Tamaru, and A. Valdes. Detecting Unusual Program Behavior Using the Statistical Component of the Next-generation Intrusion Detection Expert System (NIDES). Technical Report SRI-CSL-95-06, SRI Computer Science Laboratory, May 1995.
- [2] Joseph A. Bank. Java security. Technical report, 1995.
<http://swissnet.ai.mit.edu/~jbank/javapaper/javapaper.html>.
- [3] Mark Crosbie and Eugene Spafford. Defending a Computer System using Autonomous Agents. In *18th National Information Systems Security Conference*, oct 1995.
<http://www.cs.purdue.edu/homes/mcrosbie/research/NISSC95.ps>.
- [4] Justin Doak. Intrusion Detection: The Application of Feature Selection, A Comparison of Algorithms, and the Application of a Wide Area Network Analyzer. Master's thesis, University of California, Davis, Dept. of Computer Science, 1992.
- [5] Tim Finin and Richard Fritzson. Kqml as an agent communication language. In *the Third International Conference on Information and Knowledge Management (CIKM'94)*, nov 1994.
<http://www.cs.umbc.edu/kqml/papers/kqml-acl.ps>.
- [6] James Gosling and Henry McGilton. The java(tm) language environment: A white paper. Technical report, Sun Microsystems, 1995.
<http://www.javasoft.com/whitePaper/java-whitepaper-1.html>.
- [7] Harold S. Javitz and A. Valdez. The NIDES Statistical Component: Description and Justification. Technical report, SRI International, March 1993. <ftp://ftp.csl.sri.com:/pub/nides/statreport.ps>.
- [8] Calvin Ko, George Fink, and Karl Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proc. 10th Annual Computer Security Applications Conference*, 1994.
<http://seclab.cs.ucdavis.edu/papers/kf94.ps>.

- [9] Sandeep Kumar and Eugene Spafford. An Application of Pattern Matching in Intrusion Detection. Technical Report 94-013, Purdue University, Department of Computer Sciences, March 1994.
- [10] Raymond W. Lo and Karl N. Levitt. MCF:: a malicious code filter. *Computers & Security*, pages 541–566, 1995.
- [11] Teresa F Lunt. A Survey of Intrusion Detection Techniques. *Computers & Security*, 12(4):405–418, June 1993.
- [12] Peter Magnusson. *SimICS User's Manual*. SICS, 1996.
<http://www.sics.se/simics/>.
- [13] Andreas Rasmusson. Interactive security assistance for end-user supervision of untrusted programs. Master's thesis, Royal Institute of Technology, oct 1996. <http://www.sics.se/~ara/papers/thesis96.html>.
- [14] Lars Rasmusson. Socially controlled global agent systems. Master's thesis, Royal Institute of Technology, oct 1996.
<http://www.sics.se/~lra/exjobb/rapport.ps.gz>.
- [15] Lars Rasmusson and Sverker Jansson. Simulated social control for secure internet commerce (position paper). In *New Security Paradigms Workshop '96*, apr 1996.
<http://www.sics.se/~lra/simsocctr/simsocctr.html>.
- [16] Lars Rasmusson, Andreas Rasmusson, and Sverker Jansson. Reactive security and social control.
<http://www.sics.se/~ara/papers/nsp-panel.ps>, aug 1996.
- [17] Sun. Frequently asked questions - applet security. Technical report, Sun Microsystems. <http://java.sun.com/sfaq/>.
- [18] Joseph Tardo and Luis Valente. Mobile agent security and telescript. Technical report, General Magic, Inc, 1996.
<http://www.genmagic.com/Telescript/Compcon96.ps>.