Exploiting OS-level Mechanisms to Implement Mobile Code Security

Viktoria Felmetsger and Giovanni Vigna
Reliable Software Group
Department of Computer Science
University of California, Santa Barbara
rusvika,vigna@cs.ucsb.edu

Abstract

Mobile code systems provide an infrastructure that supports autonomous mobile components, called mobile agents. The infrastructure implements services for the transfer, execution, and protection of mobile agents. Security services are usually provided by implementing new security mechanisms that are explicitly tailored to mobile components. Unfortunately, developing sound, reliable security mechanisms is a non-trivial task, and a history of vulnerable and/or incomplete implementations of these mechanisms led to the idea that mobile code systems are inherently insecure, too complex, and very difficult to deploy. To overcome these problems, we developed a mobile code system that relies as much as possible on the security mechanisms already provided by the underlying operating system. By doing this, it is possible to develop, with reduced effort, security services that rely on well-known, well-understood, and well-tested security mechanisms. Also, by describing the security of the mobile code system in terms of the OS security mechanisms, system administrators can better evaluate the security implications of deploying the system. This paper describes the design and implementation of our system and compares its performance to several existing mobile code systems.

Keywords: *Mobile Agent Security, Mobile Code, OS Security.*

1. Introduction

Mobile code systems provide a distributed infrastructure that supports the execution of mobile components. Different forms of code mobility have been identified [5]. The most common form of code mobility is *code on demand*, which is the downloading of executable content in a client environment as the result of a client request to a server. A well-known example of this approach is the download of Java applets or JavaScript code in a Web browser. A different form of code mobility is represented by the upload of code to a server, where this code is executed. This form of mobility, also known as *remote evaluation* [15], allows the client to execute a computation close to the resources located at the server's side, so that network interaction can be reduced. Common examples are represented by the use of SQL to perform queries on a remote database or the upload of PostScript code to a remote printer. A third form of code mobility is represented by *mobile agents*. In this case, mobile components can explicitly relocate themselves across the network, usually preserving their execution state (or part thereof) across migrations. Examples of systems supporting this type of mobility are Telescript [20], D'Agents [6], Aglets [8], and JADE [3].

The mobile agent paradigm represents the most powerful form of code mobility because it provides support for multi-hop migration and autonomous execution [5]. Unfortunately, while other forms of code mobility are in widespread use, mobile agents have not been received well by the Internet community. This has been mostly because of the security issues associated with code that moves across the network, executing on many different hosts [17]. The complexity associated with enforcing a secure environment for these systems has prevented network service providers from deploying mobile agent technology. There is a need for mobile agent systems whose security design can be easily understood and evaluated. We believe that this is an important pre-requisite for the wide-spread acceptance of this type of systems.

Many of the security issues in mobile agent systems have been studied in different contexts by both the distributed systems and the computer security communities for a long time. The results achieved could be used and extended to secure mobile agent systems. Unfortunately, these results have seldom been used as a basis for the design and implementation of secure mobile agent systems. In most cases, these systems are proof-of-concept prototypes whose focus is on mobility mechanisms, and, as a consequence, security is left as future work. Other systems provide some basic se-

curity mechanisms and primitive support for the definition of security policies, but the provided mechanisms are far from being a sound, comprehensive security solution [4]. Also, the developers of mobile agent systems often feel that they need to re-implement well-known security mechanisms because they assume that mobile agents require specialized security mechanisms. In many cases, this approach leads the developers to "re-inventing the wheel," sometimes in insecure ways.

To address these issues, we followed a novel approach and developed a mobile agent system, called "Distributed Agents on the Go" (DAGO), which uses as much as possible the security mechanisms and tools already provided by modern operating systems. The goal of this effort is to demonstrate that the existing OS-level security mechanisms can be relied upon to implement a secure mobile code infrastructure. Note that the proposed system, just as many other existing mobile agent systems, does not solve the problem of protecting mobile agents from malicious hosts.

The design of the DAGO system provides two main advantages. First, by using well-tested security mechanisms it is possible to increase the assurance in the overall security of the system; second, because the OS security mechanisms are well-known and well-understood, the implications of the deploying a mobile code system in an existing environment can be better understood, from both the administrator and the developer points of view.

This paper presents the design and implementation of the DAGO system prototype and the evaluation of its performance with respect to existing mobile code systems. Even though performance is not a key criteria of the design, the evaluation is used as evidence to support the claim that a system that relies on operating system-level mechanisms can be a viable solution.

The rest of this paper is structured as follows. Section 2 discusses related work. Section 3 presents the design of the DAGO system. Section 4 discusses the security features of our design and elaborates on how they can be extended. Section 5 compares the performance of the DAGO system with respect to mainstream mobile agent systems. Finally, Section 6 draws conclusions and outlines future work.

2. Related Work

Mobile agents, as autonomously migrating software entities, present great challenges to the design and implementation of security mechanisms. In a mobile code infrastructure, the hosting system should not only be able to efficiently accommodate the incoming agents, but, at the same time, it must ensure that its own privacy and integrity cannot be compromised by malicious actions performed by the code under execution. Thus, every mobile agent system must implement a number of security mechanisms to enforce the overall integrity, confidentiality, and availability of the system.

The security issues introduced by mobile agents have been mostly addressed by duplicating the authentication and access control mechanisms employed by modern operating systems. While both authentication and authorization mechanisms have been extensively explored, in general, little attention has been given to the implementation of comprehensive accounting and auditing mechanisms.

For example, the Aglets Software Development Kit [1] (Aglets SDK) is a Java-based mobile agent system developed by IBM Tokyo Research Laboratory in Japan. In the Aglets SDK, mobile agents are called *aglets*. Aglets are implemented as threads in a Java Virtual Machine (JVM). An aglet's execution environment is implemented by a component, called *Tahiti server*, which supports agent execution, provides mechanisms for agent mobility, and implements the security mechanisms. The Aglets agent system provides a simple authentication subsystem based on host identifiers, and no accounting, resource control, or auditing mechanisms are provided. The Aglets system defines a policy description language to define access control lists (ACLs) for resources such as files, sockets, and runtime objects. For details about available permissions, see [9].

Another example of a mobile agent system is the Java Agent Development Framework (JADE) [2] developed by Telecom Italia Lab. JADE is implemented in Java, and the agent execution environment is implemented by a JVM instance, where each agent is a single Java thread [11].

According to the JADE developers, the initial security model of JADE was purely based on the Java Security API, and only later a set of additional security mechanisms became available as add-ons [11]. The new authentication and authorizations mechanisms are based on the concept of *principals*, *resources*, and *permissions.* Principals are people, departments, or other entities whose identity is authenticated using a username/password combination. Once authentication is complete, a principal is given a certificate and a set of access permissions, which she can delegate to her agents. Every JADE agent is associated with a particular principal and has to present its identity certificate to the system in order to access its resources, which include local files, network sockets, environment variables, etc. Permissions are implemented as objects that contain resource identifiers and a set of actions allowed on the resources. To enforce the security policies, JADE uses a *Policy File* which contains all the permissions for the principals. The migration of agents is protected by using the *Secure Socket Layer* protocol [10]. No accounting or auditing system is provided. A detailed description of JADE's security policies can be found in [19].

Both Aglets and JADE implement their security systems by replicating, within the runtime, the already existing se-

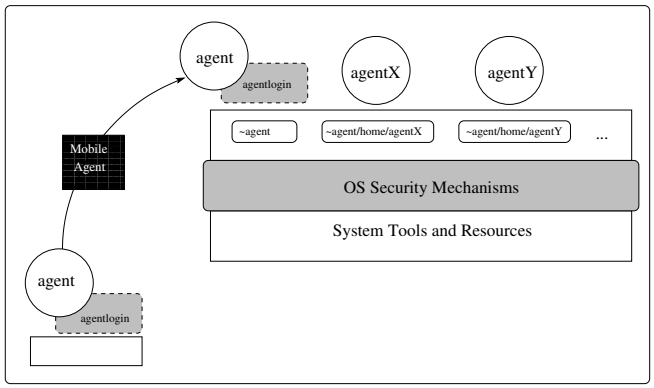

Figure 1. Architecture of the DAGO system.

curity mechanisms provided by the operating system. The fact that most of the supported security policies must be set up manually by the user makes it relatively difficult and error-prone to setup a complex policy (for example, a system-wide file access policy). Furthermore, both Aglets and JADE do not have support for either accounting or auditing in their design, while these mechanisms are provided by most OSes and could be easily leveraged by a mobile agent system.

3. System Architecture

The goal of the DAGO system is to build a computational environment that allows for the execution of incoming mobile agents, supports their further relocation, and, in addition, provides a mechanisms to protect the hosting system against possible attacks from the mobile agents. The idea that drove the design of the system is to rely almost exclusively on existing and well-tested OS-level security mechanisms, in particular those provided by the UNIX family of operating systems.

UNIX-based systems have a well-known permission-based security model that supports access control and separation of privileges within the system. We will show that by employing these existing OS-level security mechanisms, it is possible to create a secure mobile agent system that can be easily understood and configured.

In the DAGO system, a mobile agent is viewed as an ordinary system's user who logs in to the host and uses some of the system's resources for its own needs. Consequently, every incoming mobile agent is given an individual account and a unique user identifier (UID) for the duration of its execution on a host. This approach allows the hosting OS to apply to mobile agents the same set of rules and policies that are applied by the OS to all of its users.

3.1. The Agent Shell

Each host that participates in the DAGO system has a statically created account, called *agent*, which is used for creation and deletion of temporary accounts for incoming mobile agents. The *agent* account has a custom execution shell, called *agentlogin*, which is invoked each time somebody tried to log in as the user *agent*. The task of *agentlogin* is to accept all incoming mobile agents and to provide them with a secure execution environment.

Many of the tasks carried out by the *agentlogin* shell require administrative privileges. One of the obvious ways to extend the privileges on the *agent* account is to make the *agentlogin* a set-UID (SUID) program belonging to the user *root*. Even though a SUID program may introduce new security holes in the system, this is a necessary and reasonable tradeoff to overcome the limitations of the ordinary accounts on UNIX-based systems. Moreover, a careful implementation of the *agentlogin* shell can significantly reduce the security risks.

The *agentlogin* program is designed as a non-interactive shell that is invoked to handle an incoming mobile agent. More precisely, the very first task of the *agentlogin* program is to dynamically create a new user account on the host system and to assign this account to the newly arrived mobile agent. The *agentlogin* program has a predefined range of UIDs which can be assigned to mobile agents. When an agent arrives, *agentlogin* consults the */etc/passwd* file to check if any of the UIDs in the assigned range are unused and can be assigned to the mobile agent. If none of the UIDs in the range is available, the program simply refuses to execute the agent and exits. Otherwise, *agentlogin* creates a new account with a username of *agentX*, where *X* is the UID chosen for the arrived agent. To set up an account and to ensure that *agentX*'s UID is unique in the system for the time of its execution, *agentlogin* updates the */etc/passwd* file to include the *agentX* account and creates a home directory for that user. Then, it copies the code of the agent to

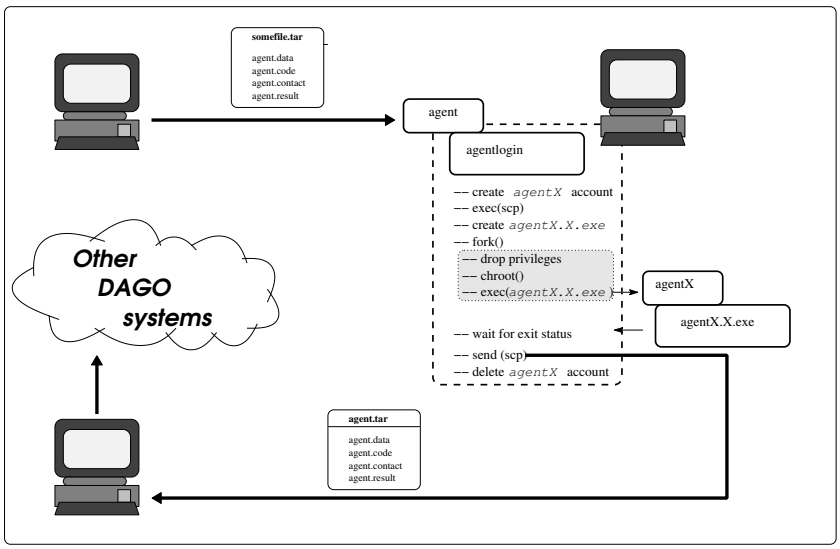

Figure 2. Execution model of the DAGO system.

agentX's home directory and `forks` a new process to execute the agent's code. The child process first changes its UID to the UID of *agentX* and then invokes the appropriate interpreter. By dropping its privileges, the *agentlogin* program makes sure that all mobile code will be executed with the (limited) privileges of the *agentX* user.

Therefore, all mobile agents accommodated by the *agentlogin* program are executed within their own user accounts, and, as a result, have to comply with all the security policies enforced by the hosting OS. When *agentX*'s execution completes, *agentlogin* examines the exit status of the process and, based on its value, proceeds with further actions. For example, a specific exit code is used to denote that the agent has requested to relocate itself. As another example, a specific exit code is used to request the results of the agent's execution to be sent (by email) to a specific address. Once *agentlogin* completes all the requested tasks, it deletes the *agentX* account, together with all its files and running processes.

3.2. Agent Transfer

One of the fundamental services provided by a mobile agent system to the mobile agents is the support for their mobility. Since the mobile code can be easily tempered with when in transit between two hosts, it is the responsibility of the mobile agent system to provide a secure communication mechanism. Due to the lack of the OS-level mechanisms for the secure transfer of information between hosts in the UNIX-based systems, we employed the Secure Copy command (`scp`), a widely-available remote file copying command that runs on top of the Secure Shell protocol (SSH) [14], as a relocation engine in the DAGO system. The use of `scp` command provides the DAGO system with the full benefit of the Secure Shell protocol (e.g., traffic encryption and host authentication) and makes communication mechanisms easy to understand and deploy.

For example, suppose that user *tony* wants to submit an agent from his account on host *pizzaworld* to the *agent* account on host *mandolino*. Then, *tony* packs all the files required for his agent's execution into an archive file, for example *agent.tar*, and executes the following command on the host *pizzaworld*:

```
scp agent.tar agent@mandolino:
```

When this command is executed, the `scp` client contacts the `sshd` daemon on the host *mandolino*, performs two-way authentication and then invokes the *agent*'s shell, which is the *agentlogin* program. When invoked, *agentlogin* receives the following string passed as an array of arguments:

```
agentlogin -c "scp -t ."
```

This means that *agentlogin* is expected to execute the "`scp -t .`" command in order to complete the transfer of the file and that the file will be stored in the home directory of the *agent* account. At this point, *agentlogin* modifies the arguments to be passed to the local `scp` command, so that the destination path is set to be the home directory of *agentX* and then invokes the local `scp` command. When `scp` completes, the mobile agent's code is contained in the home directory of *agentX* and the *agenlogin* can proceed with its execution.

When migration is requested by the mobile agent, the *agentlogin* program uses `scp` in batch mode to submit an

agent to the remote machine. For that purpose, *agentlogin* uses the configuration file *config* in the *.ssh* directory of the *agent* account and executes the following command:

```
scp -F /home/agent/.ssh/config
          agent.tar agent@destination:
```

The file *config* contains the directives to the `ssh` client which specify the type of authentication to be used. Some of the SSH authentication methods that can be employed by the DAGO system are discussed in 4.1.

In summary, the use of `scp` as a submission and relocation tool almost does not require any additional setup and provides the DAGO system with: secure encrypted communication between *agent*-enabled hosts, initial invocation of the *agentlogin* program, and automated relocation of an agent to another host. Also, since the `scp` command on a destination host is invoked by the *agentlogin* shell, the actual arguments supplied to this program are parsed and possibly redefined by *agentlogin*, which guarantees that the agent's file is copied to the proper location.

3.3. Agent Execution

In the DAGO system, an agent is transfered between hosts in the form of a *tar* archive, which contains the files *agent.data*, *agent.code*, *agent.contact*, and *agent.result*. The *agent.code* file contains the code to be executed on every host visited by the agent. In the current implementation, the code is a Bourne Shell script. Since the state of the agent (e.g., the value of a variable indicating the next host to visit) might be changing over the agent's lifetime, the file *agent.data* stores the current state of the agent and is used to the build agent's executable file on a host. The file *agent.contact* is optional and provides an email address to be contacted in case of execution errors. The file *agent.result* is used by the agent to store the results of its execution. This file's contents are sent by email to the agent owner when the agent terminates with a specific exit code.

The *agentlogin* shell copies the tar archive to the newly created *agentX* home directory and properly updates the archive's permissions. Then, *agentlogin* adds a number of procedures to the agent's code. These procedures allow an agent to execute commands supported by the DAGO system. For example, when *agentX* wants to migrate to another host, all it needs to do is to call the procedure `go()`, passing as a parameter its new destination. Similar procedures are included to help *agentX* to update its data. Thus, the *agentlogin* program, in addition to providing the execution environment to the agent, also extends the interpreter with additional commands.

Unfortunately, even though the interpreter that executes an agent runs with the UID of *agentX*, a process may try to escalate its privileges by exploiting vulnerabilities in certain programs installed on the host OS. To limit the agent's access to system's resources, *agentlogin* executes the `chroot()` system call on the *agentX* account and puts the agent in a restricted file system. The use of the `chroot()` system call confines the execution of untrusted agents to a limited portion of the file system (called "jail") while still allowing them a certain freedom of action. By making a careful selection of UNIX utilities available inside the jail, it is possible to set up a flexible and secure execution environment.

Once the agent finishes its execution, the control goes back to the *agentlogin* shell, which is responsible for final packaging and relocation of the agent. In our present implementation, an agent communicates with *agentlogin* program by using exit codes. These exit codes are produced as the result of invoking the custom procedures that are inserted into the agent's code during the initial setup. The final task of *agentlogin* is to terminate all *agentX*'s processes and remove the *agentX* account from the system.

4. System Security

In this section we discuss how, by employing various UNIX system tools and built-in features, different levels of security that can be achieved in the DAGO system. Section 4.1 elaborates on the degrees of authentication possible in the system when *scp* is used as a mobility mechanism. Section 4.2 shows additional ways to create more secure agent execution environments by employing various UNIX tools. Section 4.3 shows how network access control is implemented in the DAGO system. Finally, section 4.4 describes how UNIX logging, auditing, and accounting mechanisms can be used in a multi-agent environment.

4.1. Mobile Agents Authentication

One of the major security issues when executing mobile code is to determine the level of trust that can be given to the code, based on the identity of the principal associated with the code. Unfortunately, many possible identities may be associated with the same agent. For example, an agent may be associated with the agent's developer, the agent's code signer, the agent dispatcher, or the host from which the agent was received. Moreover, even an authenticated agent might have been "brainwashed" when executed on a malicious or compromised host, and, therefore, cannot be trusted completely. Because of this, instead of trying to determine the identity of an agent and the level of its trustworthiness, we created an environment in which the execution of an arbitrary agent will not be able to compromise the security of the host. Nevertheless, the use of the authentication mechanisms provided by the `scp` adds valuable verification of the agent's origin.

Depending of the desired authentication level, different SSH authentication methods can be used. For example, every SSH server has a host key, which uniquely identifies the server host. Each client keep its own database of known host keys in the */etc/ssh_known_hosts* and *~/.ssh/known_hosts* files. It is possible to use the *StrictHostKeyChecking* option to block connections from hosts whose keys are not in the files mentioned above. By doing this, it is possible to limit the set of hosts that will be involved in agent execution.

Another possible option is to leave a number of strict authentication methods enabled and use the trusted-host authentication scheme for known installations of the DAGO systems. This form of authentication is quite weak and assumes full trust between the hosts that are a part of the infrastructure. This setting will require an administrator to create and update *~agent/.rhosts* and *~agent/.shost* files on each computer with an *agent* account.

In our implementation, we use a pure public-key authentication scheme, which verifies a client's identity based on public-key cryptography, and which, generally speaking, is the most secure authentication option in SSH. In this case, a client has to provide a public key to the server and, if this key is found in the *authorized_keys* file on the server, the client has to prove that it has access to the corresponding private key.

Thus, even though authentication might not be completely relied upon when executing mobile code, there still are a number of ways to enforce a reasonable level of authentication and trust in the DAGO system, through the use of common OS-level security tools and mechanisms.

4.2. System Resources Usage Control

The goal of a mobile agent system is to provide incoming agents with an execution environment, which allows mobile code to perform its tasks, and, at same time, is able to control actions of the code being executed. The UNIX family of OSes has a number of built-in tools which can be employed by a mobile agent system to apply additional restrictions on incoming agents.

In the DAGO system, the creation of an account for a mobile agent is intended to be a means to compartmentalize agents, and, therefore, no logins are desired or allowed for such accounts. We chose to use */bin/nologin* as the default user shell for the *agentX* accounts. Another possible option would be to execute a mobile agent inside a restricted shell and to make this restricted shell the only shell available inside the `chroot()` jail. The use of a restricted-mode shell puts even stronger limitations on the *agentX* account since it disallows the modification of certain environment variables and account configuration files. Thus, it restricts *agentXs* from running any commands that are not in the directories contained in the `PATH` environment variable.

The fact that in our design a mobile agent is in the form of a bash script allows us to apply even further restrictions on the usage of system resources by the executed code. `Bash` shell has a build-in command, called `ulimit`, which controls the amount of system resources used by the current shell and all processes created by it. For instance, important limits as the maximum size of files created by a user, the maximum amount of CPU time, the maximum amount of both virtual and physical memory available to the shell can be set with the `ulimit` command. It is important to note that, if the original limits are set by *root*, non-privileged users will not be able to increase their values by subsequent calls to `ulimit`.

To enforce limits on the amount of resource used by mobile agents, we created a new *bashrc_mobile* file belonging to *root*, which is passed as the *–rcfile* option to the `bash` shell used to execute the agent. This file, in addition to usual information contained in */etc/bashrc* file, contains calls to `ulimit` with chosen limits. For example, the *bashrc_mobile* file can contain the following information:

```
# Source global definitions
if [ -f /etc/bashrc ]; then
. /etc/bashrc
fi

ulimit -Ht 60     # max CPU time in seconds
ulimit -Hv 10240  # max virtual memory
                  # available to the shell
```

Even though `ulimit` is a built-in command of the `bash` shell, similar commands exist for other types of agents implementations (e.g., `Tcl`). Moreover, UNIX offers additional ways in which execution of mobile agents can be controlled by the execution environment. For instance, enabling of the quota system for *agentX* accounts will limit the disk space available to each mobile agent; the invocation of `setpriority()` command by *agentlogin* will set the upper limit on *agentX*'s processes scheduling priority. Thus, depending of the desired degree of control, various OS-level mechanisms can be applied to the hosted mobile agents.

4.3. Network Access

Controlling how mobile agents access the network is important issue to consider in the design of a secure mobile agent system. In general, mobile agents come to a host to use its local resources and, therefore, should be prohibited from the use of network completely. However, in certain cases, a limited amount of network access should be allowed. For example, suppose that we want to give access to an SQL database to locally executed agents. In this situation, we have to allow mobile agents to open connections to some local ports, but we still want to disallow for any other network access.

In the DAGO system, we used the *iptables* Linux kernel module to implement a fine-grained network access control mechanism for mobile agents. *Iptables* is an administrative tool that allows to specify packet filtering rules on a host. The default table, called *filter* supports three built-in rule chains: INPUT, FORWARD, and OUTPUT. Here, we are mostly concerned with the OUTPUT chain. Every packet originating on the local host will be matched against the rules in this chain. Rules for OUTPUT can be specified based on the group or user IDs of the sending processes. Consider the following set of commands:

```
iptables -N agentTEST

iptables -A OUTPUT
         -m owner --gid-owner 500
         -j agentTEST

iptables -A agentTEST
         -p TCP
         -i lo
         -m multiport --dport 4444
         -j ACCEPT
```

This will create new chain agentTEST with default policy to reject a packet. When a packet is originated from a process with GID 500, it will be matched to the rules in agentTEST chain. If it is not a packet from the loopback device to TCP port 4444, the packet will be rejected.

Since in the DAGO system every mobile agent has been assigned a UID and a GID, every packet coming from an agent can be easily filtered with *iptables*. Note that, instead of the *–gid-owner* option in the above example, it is possible to use the *–uid-owner* option to filter packets based on the UID of the agent. By doing this, it is possible to set different filtering rules for different agents. Filtering rules can be inserted in *iptables* by the system administrator prior to enabling the *agent* account or can be created by the *agentlogin* shell on the fly.

4.4. Auditing and Logging

UNIX, as a multi-user system, provides a number of logging, auditing, and accounting mechanisms to monitor the actions of its users and the status of its resources. These mechanisms can be easily leveraged, and, if necessary, extended to a multi-agent environment. In addition to providing basic logging facilities, such as *syslog*, UNIX systems support much more sophisticated tools for auditing system activities and process accounting. These tools can be easily customized to different levels and relied upon to detect various types of system misuse.

A number of UNIX auditing tools work at the system call level and can be configured based on different types of events, such as opening and closing of files, reads and writes, programs executed, and so on. They also allow one to specify groups of system objects to be monitored for certain activities. UNIX accounting tools can track system usage by recording the statistics about CPU and memory usage, I/O operations, running time, and other forms of system resource usage along with the UIDs of the processes involved.

In addition to the standard tools provided by the OS, there is a variety of widely available tools which can provide even a greater degree of customization and security assurance. For instance, SNARE (System iNtrusion Analysis and Reporting Environment) [12], developed by InterSect Alliance, is a dynamically loadable kernel module that can be used as a stand-alone auditing system or as a distributed tool. SNARE can be configured to monitor events associated with certain groups of users, filter the monitored events with specific "search expressions", and submit reports in different formats and time frames. The type of events monitored by SNARE can be either defined by a category (for example, system calls) or by an identifier (such as "denied access").

Another example of an auditing tool that could be used in multi-agent environments is Sun Microsystems' Basic Security Module (BSM), available for Solaris OS [16]. BSM supports the auditing of actions performed by individual users as well as the kernel. BSM also provides tools to manage, filter, and analyze audit records.

Clearly, auditing and accounting tools, such as SNARE and BSM, can be efficiently applied to our OS-based agent system. First of all, the types of events to be monitored in association with agent execution are very similar to those audited for the system's users. Second, mobile agents can be easily grouped and differentiated within the system. Thus, the described design of the DAGO system in which every mobile agent is a user with a unique UID and its own account makes it possible to rely completely on OS auditing mechanism to perform intrusion detection as well as to provide timely response (for example, termination of a misbehaving agent and all its processes) when suspicious activity is detected. On the other hand, in many mobile agent systems implemented on the top of virtual machines, multiple agents are executed as threads within a single process. In this scenario, tracing of unauthorized activities and termination of malicious agents might be a much more difficult task to accomplish [13]. OS-level auditing and accounting mechanisms, which assume a one-to-one mapping between a running process and an execution unit, cannot be directly used in such systems and have to be either extended or reimplemented in order to provide a reasonable level of attack detection [18].

5. Evaluation

In order to evaluate the performance of the DAGO system, we designed a simple test case application and imple-

Execution Time			Migration Time			Total Execution Time
Spaghetti	**Maccheroni**	**Penne**	**Spaghetti**	**Maccheroni**	**Penne**	
14266	22548	16797	460	589	503	108356
14228	19580	17194	408	563	508	103986
14222	17365	16882	448	544	518	99921
14314	15087	16830	456	546	553	95450
14215	15004	16873	444	558	559	94937
14202	15357	17229	455	536	563	95443
14246	15414	16866	456	558	542	95245
14273	15330	16876	456	547	559	95247
14248	15352	16874	455	550	555	95257
14248	15431	17240	451	513	548	95653

Table 1. Execution results for the Aglets system (in milliseconds).

Execution Time			Migration Time			Total Execution Time
Spaghetti	**Maccheroni**	**Penne**	**Spaghetti**	**Maccheroni**	**Penne**	
15403	22578	14873	447	709	882	112775
16130	20520	15538	420	420	378	108496
16199	16687	11977	427	427	378	96368
16225	13989	12308	422	422	383	91265
16258	13514	12469	397	393	348	89240
16488	13479	14475	470	470	904	96021
16351	13477	15178	439	439	358	94831
16460	13588	11647	427	427	355	89313
16467	13477	11856	386	386	347	88109
16540	13362	12129	388	388	387	88977

Table 2. Execution results for the JADE system (in milliseconds).

mented it for the Aglets, JADE, and DAGO systems. The goal of this test was not to determine which system is more efficient, but rather to evaluate if the performance of a mobile agent system that uses OS-level security mechanisms is comparable with the performance of other well-known mobile agent systems.

The evaluation was carried on a testbed consisting of three PCs named *spaghetti*, *maccheroni*, and *penne*, running Linux (kernel version 2.4.22) with Java SDK 1.4.2. All hosts have Intel Pentium 4 processors, with CPUs clocked at 1.80 GHz on *spaghetti*, and CPUs clocked at 1.50 GHz on both *maccheroni* and *penne*. The *spaghetti* machine has 1024 MB of RAM, while *maccheroni* and *penne* have 256 MB and 640 MB of RAM, respectively.

Our test application is composed of agents that verify the integrity of a set of files. The agents are initialized with a list of pre-computed MD5 checksums of the files. During execution, the agents migrate from host to host and compute, on each host, the current checksum values for the given set of files. The results are then compared with the ones given to the agent at startup time, and possible discrepancies in the checksums are reported. This application implements an agent-based integrity checking service, similar, for example, to Tripwire [7]. We chose this application because it requires a substantial amount of both I/O and CPU resources.

To perform the evaluation, three agent-based applications were developed: a first one in the form of a bash script that uses the DAGO system to move between hosts; a second one in the form of a Java aglet that uses the Aglets SDK (v2.0.2); and a third one in the form of a Java agent for the JADE system that uses the JADE version 3.1. A simplified version of the mobile agent implementation for the DAGO system is shown in the figure 3. All three agent-based applications were designed to be functionally identical, and, as much as possible, structurally similar. However, the version of the application implemented for the DAGO system used traffic encryption and public-key authentication, while both the Aglets-based and the JADE-based implementations did not use any form of traffic encryption or strong authentica-

Execution Time			Migration Time			Total Execution Time
Spaghetti	**Maccheroni**	**Penne**	**Spaghetti**	**Maccheroni**	**Penne**	
11556	18619	16668	1224	1279	1224	90639
11539	17722	15754	1191	1166	2213	92324
11578	18621	16655	1257	1211	1178	91160
11645	18675	15723	1184	1192	2175	90470
11541	17561	15727	1252	1174	2224	91090
10554	17797	15745	1185	2216	2205	90644
11551	17527	15859	1179	1224	2207	90477
11575	18753	16659	1197	1204	1232	92153
11528	17733	15646	1222	2208	1222	90524
11644	18749	16792	1211	1244	1213	90623

Table 3. Execution results for the DAGO system (in milliseconds).

```
#!/bin/bash

numHostsVisited=3
destination=" "
fileList=(...)     # list of file names to be checked
md5Sums=(...)      # corresponding MD5 checksums
                   # for the files in the list

# calculate and compare MD5 checksums
checkSums () {...}

# determine the next host to go to
# set value of destination variable
nextHostToVisit () {...}

checkSums
nextHostToVisit

if [ "$numHostsVisited" -lt 6 ]
then
    let "i = $numHostsVisited + 1"
    # update agent's state
    # new value will be stored
    # in agent.data by agentlogin
    setdata "numHostsVisited" "$i"

    # move to another host
    go "$destination"
fi
```

Figure 3. Example of the agent's executable file.

tion. The three applications were given a set of 2098 binary files that had to be verified on each host. The tests were executed ten times. For each run of the tests, the agent visited each of the three hosts twice.

The results of the execution of the tests are shown in Table 1 for the Aglets-based application, in Table 2 for the JADE-based application, and in Table 3 for the application based on the DAGO system. In all the tables, the first three columns show the time that an agent spent on each host. The next three columns show the time that it took an agent to migrate from one host to another. The same itinerary was used for every execution of the test applications. More precisely, agents always traveled from *spaghetti* to *maccheroni*, then to *penne*, and then back to *spaghetti* two times. Thus, the migration times are labeled with the name of the destination host (e.g., the migration time column labeled with *penne* represents the time required to move from *maccheroni* to *penne*). The last column shows the total execution time of an agent, which visited each of three hosts twice.

In addition to the above data, the amount of bytes transfered over the network during the execution of each application was collected. The total amount of data transfered was 1,229,619 bytes for Aglets-based application and 1,279,636 bytes for the JADE-based application. The amount of data transfered during the execution of the agent based on the DAGO system (including all `ssh` communications during authentication) was 1,402,037 bytes.

The test cases presented here are not meant to be a complete and sound comparison of the three systems because the described agent applications were implemented using different programming languages and cannot be compared directly. Nonetheless, the results support the claim that a mobile agent system based on OS-level mechanisms have the potential to provide an execution environment in which mobile agents can be executed with a performance comparable to existing mainstream systems.

6. Conclusions and Future Work

Mobile agent systems provide support for the execution of mobile components, called mobile agents, which migrate between hosts to perform their tasks. Most mobile agent systems implement agent-specific security services to address the security issues associated with the execution of mobile code. By doing this, the existing security services, provided by the underlying operating systems, are dupli-

cated, often in an insecure way.

We designed and implemented a mobile agent system that uses, as much as possible, the security tools and mechanisms that are widely available on existing operating systems. In particular, we implemented a UNIX-based prototype that uses Secure Shell for traffic encryption and authentication, and existing UNIX security mechanisms for access control.

The use of OS-based security mechanisms has several advantages. First of all, the security mechanisms provided by the operating system have been thoroughly tested by the security community. Therefore, it is unlikely that the mechanisms can be bypassed by malicious mobile code. Second, OS-level security tools and mechanisms are well-understood by the administrators of computer networks. Therefore, it is easier for them to evaluate the security impact of the deployment of a mobile code infrastructure that relies on these mechanisms, and, therefore, the introduction of mobile agent systems may become more wide-spread. Third, by relying on existing security mechanisms, it is possible to maintain the size of the code that implements agent mobility small, making it possible to thoroughly evaluate its security (e.g., by manual security audits). Finally, by using OS-level security mechanisms, the DAGO system is in large part language-independent. By modifying a small portion of the code of the *agentlogin* shell, it is possible to include support for other scripting languages (such as Tcl, Perl, and Python) and even binary applications, in the case of architecturally homogeneous networks.

We also evaluated the performance of the first prototype of the DAGO system with respect to two mainstream mobile agent systems, namely Aglets and JADE. The results show that the performance of our system is comparable to the existing mobile agent systems.

Future work will extend the system to support a number of scripting languages. Also, we plan to use OS-level auditing mechanisms to develop an intrusion detection system that monitors the execution of mobile code. In addition, we will explore how new OS-level mechanisms may be introduced to better support mobile agents. In particular, we plan to study how the OS kernel can be extended to support the migration of UNIX processes. If implemented, this mechanism would allow generic applications to move from host to host seamlessly.

References

[1] IBM Aglet Workbench. Site. `http://www.trl.ibm.co.jp/aglets/`.

[2] Java Agent Development Framework. Site. `http://jade.cselt.it`.

[3] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa. JADE. White Paper, Sept. 2003.

[4] S. Fischmeister, G. Vigna, and R. Kemmerer. Evaluating the Security Of Three Java-Based Mobile Agent Systems. In G. Picco, editor, *Proceedings of the* 5^{th} *International Conference on Mobile Agents (MA '01)*, volume 2240 of *LNCS*, pages 31–41, Atlanta, GA, December 2001. Springer-Verlag.

[5] A. Fuggetta, G. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[6] R. Gray, D. Kotz, G. Cybenko, and D. Rus. D'Agents: Security in a multiple-language, mobile-agent system. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 154–187. Springer-Verlag, 1998.

[7] G. H. Kim and E. H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. Technical report, Purdue University, Nov. 1993.

[8] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

[9] M. Oshima, G. Karjoth, and K. Ono. Aglets Specification 1.1 Draft. Whitepaper Draft 0.65, Sept. 8 1998.

[10] A. Poggi, G. Rimassa, and M. Tomaiuolo. Multi-User and Security Support for Multi-Agent Systems. In *Proc. of WOA 2001 Workshop*, Modena, Italy, Sept. 2001.

[11] G. Rimassa. *Runtime Support for Distributed Multi-Agent Systems*. PhD thesis, University of Parma, Jan. 2003.

[12] SNARE - System iNtrusion Analysis and Reporting Environment. `http://www.intersectalliance.com/projects/Snare`.

[13] S. Soman, C. Krintz, and G. Vigna. Detecting Malicious Java Code Using Virtual Machine Auditing. In V. Paxson, editor, *Proceedings of* 12^{th} *USENIX Security Symposium*, pages 153–167, Washington, DC, August 2003. USENIX.

[14] SSH Protocol Architecture. `http://www.ietf.org/internet-drafts/draft-ietf-secsh-architecture-17.txt`, 2004.

[15] J. Stamos and D. Gifford. Implementing Remote Evaluation. *IEEE Transactions on Software Engineering*, 16(7):710–722, July 1990.

[16] Sun Microsystems, Inc. *Installing, Administering, and Using the Basic Security Module*. 2550 Garcia Ave., Mountain View, CA 94043, December 1991.

[17] G. Vigna. Mobile Agents: Ten Reasons For Failure. In *Proceedings of the IEEE International Conference on Mobile Data Management (MDM '04)*, pages 298–299, Berkeley, CA, January 2004. Position Paper.

[18] G. Vigna, B. Cassell, and D. Fayram. An Intrusion Detection System for Aglets. In N. Suri, editor, *Proceedings of the* 6^{th} *International Conference on Mobile Agents (MA '02)*, volume 2535 of *LNCS*, pages 64–77, Barcelona, Spain, October 2002. Springer-Verlag.

[19] G. Vitaglione. *JADE Tutorial - Security Administrator Guide*. Telecom Italia Lab, 2002.

[20] J. White. Telescript Technology: Mobile Agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press/MIT Press, 1996.