Protecting Mobile Web-Commerce Agents with Smartcards

Stefan Fünfrocken

Department of Computer Science, Darmstadt University of Technology,
Alexanderstr. 6, D 64283 Darmstadt, Germany
Email: fuenf@informatik.tu-darmstadt.de

Abstract. Mobile agents add a new communication paradigm to traditional network communication mechanisms. In contrast to the classical mechanisms like remote programming, RPC, or client-server systems, mobile agents have specific advantages when used in a heterogeneous networking environment such as the World Wide Web. So far, the pervasiveness of publicly available mobile agent platforms is not given. Offering a seamless integration of mobile agents into the widespread and well-accepted WWW environment is crucial for the success of mobile agents. One of the growing fields of interest in the Web is the area of electronic commerce. Mobile Web-commerce agents could play a prominent role in future electronic commerce scenarios, if the malicious host problem could be solved. Our paper describes the integration of mobile agents into the Web and the use of Java cards to allow a mobile agent to store and transport data securely. This should promote the usage of mobile agents for electronic commerce purposes.

1 Introduction

Mobile agents can be seen as an extension to the traditional communication mechanisms. With the possibility to explicitly control the location of the execution of the code, the mobile agent paradigm offers specific advantages especially when used in a heterogeneous and dynamic networking environment like the World Wide Web. The potential benefits include:

- *Reduced communication bandwidth*: If vast amounts of server data have to be processed (e.g., a Web index at some agent enhanced Web index server) and if only a few relevant pieces of information have to be filtered out, it is more economical to transfer the computation (i.e., the agent) to the data than to ship the data to the computation.
- *More dynamics*: It is not necessary to install a specific procedure at a server beforehand and to anticipate specific service request types; a client or a service provider may send different types of agents (e.g., realizing new service handlers) to a server without the need to reconfigure the server.
- *Improved support of nomadic computing and intermittently connected devices*: Instead of being online for a longer period, a mobile user may develop an agent request while being disconnected, launch the agent during a brief connection session, and receive back the agent with the result at some later time.
- *Asynchronous task execution*: While the agent acts on behalf of the client on a remote site, the client may perform other tasks.

To make use of the mobile agent paradigm and its advantages, it is required to have a mobile agent platform available at any place where an agent will go to. The mobile agent platform is the execution environment for any mobile agent that travels to a host in order to access the local resources and services. The mobile agent environment takes care about the security of the local host by intercepting each agent access and by checking its conformance to the local security policy, taking into account the access rights granted to the agent. The mobile agent enviroment also provides different agent services like an agent to agent communication service, or a directory service of locally available agents and services, or a naming or trading service that are all essential for the agents during their task execution.

2 Seamless Web-Integration of Mobile Agents

The ubiquitous availability of agent environments is a precondition for any successful usage of the mobile agent paradigm. So far there are no publicly available servers that offer to host mobile agents. Because of this, the integration of the mobile agent infrastructure at a local server should not impose much overhead. In our opinion, the only way to promote fast dissemination of mobile agent platforms is the usage of a well established and widely used technology: the World Wide Web. However, to allow an easy migration to a mobile agent enhanced World Wide Web, it is not enough to just use Web technology as for example when using the HTTP protocol as communication protocol between agent systems. The more appropriate way seems to be a more seamless integration of mobile agents and the WWW. This integration can be done by:

– Enabling WWW servers to host mobile agents. WWW servers are ideal places for mobile agents, since most of the accessible data and electronic commerce shops in the Internet reside on WWW servers.
– Using WWW browsers to allow users to access mobile agents in the same well-know manner as they access Web data. Almost every user has a WWW browser which then can be used to control and communicate with mobile agents running on WWW servers, and which could even serve as a home base for personal mobile agents.
– Using standard WWW technology for the realization.

Based on this idea we started the WASP (Web Agent Based Service Providing) project in 1996 and developed a mobile agent enhanced Web server. This is realized by the use of a so-called 'Server Agent Environment (SAE)' (see [4, 5] for more details). Because Java is the programming language with the best Web integration, is object oriented, and offers good resources for the realization of mobile objects by its object serialization feature, we have chosen Java for our Web server, the SAE and also as the programming language for our Web agents.

To allow for a flexible usage and ubiquitous availability of mobile Web agents, the enabling resource in form of our SAE could be encapsulated in different ways:

– as servlet, to allow a Web server to easily integrate mobile agent support,
– as cgi-bin, to allow the integration for Web servers that do not support the servlet interface,
– as plug-in for a Web browser, to allow to use a user's browser to host local agents.

With the availability of mobile agent enabled Web servers, it is possible to consider Web agents as an additional type of data available on the Web. When inactive, a Web agent (i.e., its code) is stored as a file in a Web server's HTML space. The locally available agents are advertised on a Web page. To start one of the agents the user accesses the agent index page with a browser and selects an agent by just clicking on the corresponding hyperlink. The resulting request sent to the server indicates that the requested data is of type 'Web agent', which causes the server to forward the agent request to its SAE, which processes the request by loading and starting the Web agent. As an answer to the user's request, the graphical user interface of the Web agent is sent to the user's Web browser in form of a Web page containing the agent's GUI as a Java applet. Depending on the agent's functionality, the user interacts with the agent by the means of additional applets or simple Web pages.

One goal of the WASP project was to use Web agents to program the Web by providing additional services based on Web data. Web agents can be used for various purposes:

– Encapsulation of dynamic Web pages. A Web agent can be used to display the content of Web pages that depend on other resources (e.g., databases, user identity, real time data). On requesting the dynamic page the corresponding Web agent is started. The agent collects the desired information and produces the Web page. This scenario may require the agent to move to different Web resources either by itself or by starting sub agents, and to periodically update the dynamic page.
– Web Access to legacy applications. A mobile Web agent could be used as customized Web gateway to any legacy application. If a user wants to access a legacy application at some server which provides no or an uncomfortable Web interface, he could sent a customized mobile agent to the server which is able to use the legacy service's native interface and can provide a customized Web interface to the user.
– Provision of additional services on Web data. This can be seen as the general concept for the usage of Web agents. The possible services can be as simple as broken link checking or checking for updates of other pages, or as complex as realizing a whole web store by mobile agents.

For the WASP project we realized several applications with Web agents:

– a simple broken link checker, which accesses all documents locally by migrating to the link's location,
– a customizable Web newspaper application with serveral different and communicating agents, in which the agents collect Web pages according to a user profile combining them into a personal newspaper,
– an application for system administrators where a mobile agent is used to trace user logins in a local network by accessing and analysing the information from the unix syslog system service,
– and we realized a application management scenario, where we use our Web agents to manage a cluster of serveral WASP Web servers and SAEs.

We found that it is very easy to program a set of different agents, but that there is no easy functional decomposition to map application functionality on different agents. In addition, most applications consist of stationary agents and only less mobile agents. But we also found that mobile agents can actually be seamlessly integrated into the Web.

By using a user's Web browser as mobile agent platform - this could be done by loading a locally installed mobile agent browser plug-in, when a Web link to an agent is activated by the user - the user's browser can be used as trusted agent environment and home base for the personal agents of the user. The agents running in the user's browser could also be used as personal browsing assistents that allow for a personalized view on the whole World Wide Web. Besides having a trusted agent environment for each user, which he or she can use to launch mobile agents, the general availability of trusted agent environments is also desirable for another proposed mobile agent application scenario: electronic commerce.

2.1 Electronic Commerce and Mobile Agents

From the beginning of mobile agent research, electronic commerce was one of the aimed application areas: Already in 1994 Jim White of General Magic envisioned a global electronic market place [15] where mobile agents roam from host to host in the Internet to perform commercial transactions on behalf of their users. Although very appealing, so far this vision hasn't become true.

Currently Web-based electronic commerce in form of Web shops are using typical Web technology like scripting, applets, active server pages, and Web based database access methods to sell goods using the Word Wide Web. A user who tries to buy goods over the Internet has to search an appropriate Web store, mostly by using one of the Web search engines, then clicking through the store's Web pages looking for an product of his choice.

With the availability of mobile agents, a user willing to buy electronically could simply start a mobile Web agent which offers the service to search for the best product corresponding to the user's preferences. The mobile Web-commerce agent scenarios range from these simple 'best buy agents', over mid range applications like a personalized Web newspaper using pay per view, to highly complex special agents which are able to plan and to arrange business travels. All the information which is needed to fulfill these tasks is present on the Web and there will be even more information and Web shops that sell goods and information in the near future.

In consequence of the evolution of electronic commerce, electronic payment mechanisms have been developed in the last years. Although there are also very elaborated electronic payment mechanisms [9, 2, 8, 14], which also use electronic coins to represent real world money, current Web stores rely on more traditional payment mechanisms like credit card payment.

Since the Web is an open environment and information travels over multiple hops which are possibly not even visible for a Web user, dealing with sensible information like credit card numbers, which will result in a loss of real world money when misused, requires protection mechanisms that enforce security when doing electronic business. When transferring credit card numbers by the HTTP protocol, current Web store implementations often use the encrypted form of the protocol (either HTTPS or SHTTP) to protect the information.

When using mobile agents to buy goods or information, one is faced with a completely different scenario which results from the agents ability to quickly move around the Internet. For the new scenario the following aspects have to be taken into account:

- To buy goods, an agent has to carry information that allows the agent to pay. This can be the user's credit card number or even electronic coins.
- It seems not very desirable that a commerce agent has to contact home each time it has found another resource for the desired goods. The agent was sent out to work independently. This means that the agent itself needs to be able to decide wether to buy or not to buy at a specific store.
- The agent can quickly move from one host to another host. This is one of the advantages an agent offers. It can process all relevant data much faster than any human.
- Because the World Wide Web spreads almost the whole world it is very easy for a mobile agent to 'jump' over great geographic distances.
- The machines offering to host mobile agents by running a mobile agent system and which are visited by a mobile agent are not known beforehand. In addition it is generally not known who is running the mobile agent system.

Taking this into account, it follows immediately that providing a world wide system that enables electronic commerce with mobile agents is a very challenging task. One of the basic problems in this context is the security of mobile agents carrying sensitive data. Because of this, mobile agent security is a precondition for the usage of mobile agents in electronic commerce scenarios.

3 Protecting Mobile Agents using Trusted Computing Bases

The issues of mobile agent security are well-known [3, 1] and can be divided into the following sub issues:

1. protecting local hosts from agents,
2. protecting agents from other agents,
3. protecting agent systems from other agent systems, and
4. protecting agents from the agent system.

While it is well understood that issues 1 through 3 can be solved with existing mechanisms and cryptographical methods, protecting mobile agents from their execution environment is known to be a very hard problem [3]. This mainly results from the fact that an agent has no life of its own (although the agent abstraction suggests this at the first glance) but instead, each line of the agent's code has to be read, interpreted, and executed by the agent system. By this way, all information contained in the agent (i.e., code and data) is visible to the agent system - in fact, this visibility is a necessity for enabling the agent system to execute the agent. Each server hosting the agent has full control over the agent's code and data, and could misuse its control to take advantage of this fact. Especially when mobile agents are used to do electronic commerce business, it could be very attractive to force an agent to spent its money locally, or to extract electronic cash from the agent's data. The possible attacks of malicious hosts on mobile agents are manyfold and include:

– modification of code or data,
– force execution of certain code parts, possibly with special variable values (e.g., when the code calculates the decision to buy),
– extraction of valuable data (e.g., credit card numbers, e-cash tokens)

(see [7] for more details about possible attacks). To prevent a malicious host from attacking an agent's code and data, the information contained in the code (i.e., the code semantics) and in the data must be hidden but be executable and accessible for the agent system at the same time. There are first results in achieving this property for a special kind of agents which compute polynomial functions [11]. By using an encrypted form of the function to be calculated, it is not possible for the agent system to derive the calculated function by looking at the agent's code. And consequently, the agent system cannot tamper with the code to enforce a specific outcome. There are other approaches (see 4.3 for more details) which deal with the protection of special agent parts like an agent's log, its itinerary, or specific code and data. These approaches generally use cryptographic methods - mainly signing and encryption - to protect the information. Unfortunatly, all these results are only partly solutions for the malicous host problem.

A completely different approach which might lead to a more general solution for the malicous host problem, is to use tamper proof and trusted hardware to protect mobile agents. The main idea is to equip mobile agent systems with additional hardware, which is not under control of the local system and can host and execute mobile agents, thus providing a secure execution environment for agents. There exist several research project on this issue [16, 17, 10]. Here the whole agent environment is encapsulated by the tamper proof hardware, and consequently the whole agent will not be under control of the local host. The agents inside the system will only interact by messages with the untrusted environment (i.e., the host and the services it provides). In this scenario the agents migrate between trusted environments only and access the local resources in a client-server manner. Because of this, there is no need to protect an agent's code and data. In fact, the local host does not need to supply any mobile agent system by itself: it is the trusted hardware only that provides the agent environment. What remains, is a standardized or at least agreed communication interface between the local host with its services, and the mobile agent system with the agents.

Trusted hardware comes in different flavors (see [18] for more details): as integrated chips, as PC cards, and as smartcards. In our opinion it becomes more difficult to provide tamper proof hardware the larger the hardware gets. Because of this, smartcards, which are designed and used as tamper proof devices from the beginning on, seem to be a good candidate when looking for a cheap and easy to use tamper resistant device to be used in a mobile agent system. In addition, modern smartcards like the Java card, the Multos card, or the announced Microsoft smartcard, offer the possibility to dynamically load new functionality into the card, simply by installing new code. In this context Java cards, which allow to load and execute Java code, are very interesting, because most mobile agent systems support mobile agents programmed in Java.

3.1 The Java Card as Trusted Computing Base

With the availability of smartcards that offer a Java virtual machine, it is possible to load new functionality into the card dynamically. In addition, one can load multiple applications into the card, thus resulting in a multi functional smartcard. Because of its support for multiple applications, there need to be special security mechanisms that act as firewalls between the different applications loaded into the smartcard. The Java card uses the Java sandbox approach to prevent that one card application tampers with the code or data of another card application. With the new Java card API [12] it is possible to open and escape the sandbox in a controlled way by object sharing.

Having dynamically loadable, multifunctional, and secure Java cards one could use these cards as trusted computing bases for mobile agents. Because of the limited resources smartcards offer, it is only possible to load small agent code parts into the card. Because Java is an object-oriented language, the code parts which are downloaded into the card are represented as Java objects. These code part objects can contain code only, code and data, or data only[1]. In the following, we will refer to these objects as 'code parts', which always means code and data.

In contrast to more powerful devices like PC cards, where the whole agent resides in the trusted environment without any special protection, the smartcard scenario requires that the code parts of an agent that should be executed in the smartcard only are protected by encryption when carried along by the agent (i.e. when outside the card). In addition, an agent could contain different code parts where each of them should be executed in the card of a specific host only. These parts remain in the agent when it visits all other hosts and consequently have to be protected from them. If some agent code should be executed in the Java card of the next host in the agents itinerary, the code has to be encrypted for that card. Since the current host is untrusted, the encryption can only be done by the trusted Java card of the current host. For 'bootstrapping' the system, the user has to trust the host where the agent starts. This might be the user's home location or a trusted third party host.

[1] Normally the data will be accessed by special methods that may also implement certain data access control mechanism. Because of this, there is also code inside the object.

This scenario requires additional infrastructure and information:

- The card must support encryption and decryption.
- There has to be a certification authority.
- The code parts inside the card must be managed.

Encryption and Authentication. One could use either asymmetric or symmetric encryption. When using asymmetric encryption, each card owns a private key and a public key, which has to be certified by a trust authority and has to have a property attached which indicates that the public key belongs to a Java card. Here, the agent's code parts are encrypted with the public key of the Java card which should execute the code. To be able to perform the encryption, the local card needs the public key of the next card. Since the public key of a card is certified and needs no protection, the key can be provided either by the agent or the local agent system and there is no need to establish a secure channel between the both cards. After receiving the key, the local card checks the validity of the key by verifying the key certificate and the key property[2]. If the key is valid, the agent's code residing inside the card is encrypted with the public key of the target card. After that, the agent extracts the encrypted code part and migrates to the new host where it could load the code into the target card.

When using symmetric encryption, each card has to share a secret encryption key with all other cards. Since this is no reasonable assumption, there has to be a key exchange algorithm which is executed between the cards of the current and next location of an agent. Because the key exchange must occur between cards only, there has to be mutual authentication of the two cards to prevent a malicious host from impersonating as a target card. The usage of asymmetric encryption for authentication purposes also requires the existence of a certification authority and certificates. In contrast to using public key encryption, symmetric encryption needs to execute a key exchange protocol. Because of that, we prefer using public key encryption.

To be able to decode the results contained in the encrypted code parts after the return of the agent to its home, either because the agent fulfilled its task, or because there was an error which results in transferring the agent to its home immediately (e.g., migration error, agent's life time exhausted), the code parts have to be encrypted for multiple targets:

- the next card, to allow regular migration,
- the current card, to allow decryption because of migration errors, when the target host is not reachable. In this case the agent should be able to re-encrypt its code parts for another target.
- the home card, to allow decryption on abnormal returns, if for example multiple migration attempts failed, or the default error policy is to send the agent home.

Card infrastructure: When using a Java card as trusted computing base for agent code parts, there has to be a managing entity residing on the card which:

- receives the encrypted code parts,
- decrypts and stores the code in a way that it can be executed,
- encrypts the code with the key of the next card (and with the additional keys),
- sends the newly encrypted code back to agent on request.

[2] The property stating that the key indeed belongs to a Java card prevents that a malicious host sends its own public key to the local card so the agent's code can be decrypted and manipulated by the host, and after that is encrypted with the public key of the target card.

Because of the need of authentication and encryption, there has to be a way to securely store the card's certificate and the encryption keys. When using public key encryption, the card should provide a key generation mechanism, so that the private key is never outside the card.

In addition, the trusted card manager should be able to detect and prevent replay attacks, where an untrusted host repeatingly downloads an encrypted code part to do black-box testing. This requires, that each encrypted code part contains the unique agent id of its owner agent. And that the manager records the access time and inputs for each external usage of each code part. Because of the limited resources of todays smartcards recording the information of each access is not feasible currently.

3.2 Java card requirements

The need to implement the agent support infrastructure on the card efficiently, requires some support from the card or the Java virtual machine itself. To be able to execute cryptographical mechanism in an efficient way, the card should offer a cryptographical coprocessor. Its functionality should be offered to card applications through an API. The current Java card API defines cryptographical interfaces, but most existing card do not implement this API.

To be able to execute the decrypted code there are two ways: first, the code can be dynamically loaded as a Java object by the managing entity, which mediates all communication from and to the code, second, the decrypted code is written to the cards 'file system' and gets installed as a card application. The first method requires the existence of a class loader on the card. This method should be the preferred one, because it allows for certain access controls by the card manager. The second method requires the ability to install a card application from inside the card.

During an agent's journey, the agent might collect information of electronic items that it would like to carry to its home location. To transport this data in a secure way, it will store them in the code parts that are encypted. This means, that the code parts loaded into the card will change their state which requires the ability to save the state of the code parts (i.e., the variables). Since some of the collected information needs to be available at one of the next hosts the agent visits (e.g., to perform a comparison of collected prices) the saved state of the code parts should be restored after loaded into the next card. It would be convenient to use the Java object serialization to do this. Unfortunately the current specification for Java card virtual machines does not include object serialization. Because of this, saving and restoring the object state of the code parts has to be done by the programmer of the encrypted code parts: The current state of any non volatile variable has to be written and later to be read explicitly to and from a card data file (see Figure 1).

3.3 Java Card Usage in the WASP Project

For the WASP project [4] we currently use Java cards in two ways: first, as authentication and signing device when a user starts an agent, and second as a trusted computing base attached to our server agent environment (SAE).

Agent startup. In the WASP system a user starts an agent by using a browser to load the page of available agents from the WASP Web server. By selecting one of the links on that page, the corresponding agent is started (i.e., loaded into the SAE). Then the agent sends its GUI applet to the users browser. Because of security reasons, the agent's GUI is encapsulated into the so-called 'WASP GUI applet' which allows the user to attach capabilities to the agent (the agent itself has no knowledge about these capabilities).

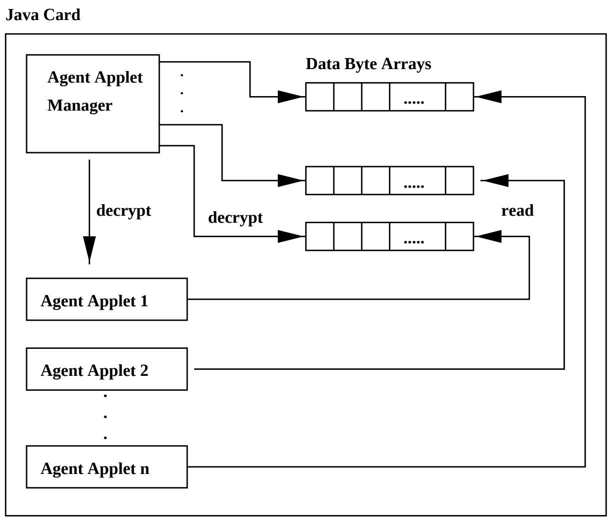

Fig. 1. On-card architecture: Saving the state

Among others, the user has the possibility to specify a list of Web servers, user names, and passwords which allow the agent to run under the corresponding user restrictions at the associated Web server [3].

The current version of the WASP GUI applet allows to read and select these data items from a Java card. To do so, an encrypted channel is established between the applet and the smartcard. This is done by exchanging a symmetrical session key over a secure communication channel. The secure channel is provided by using public key encryption.

After the user has attached the security restrictions to the agent by the means of capabilities, the WASP GUI applet sends the capabilities to the user's Java card which signs the capabilities. By this way, the user authorizes the agent to act on his behalf, while granting it the specified access rights.

Trusted Computing Base. In the more appropriate Java card usage scenario, we use the card as trusted computing base as described above. The card is attached to the SAE and offers the agents the possibility to download, encrypt, and execute code parts. Because of the current limitations of the Java cards, we implemented a prototypical scenario only, where certain actions which should be executed from inside the card are executed at the outside. This is mainly due to the lack of object sharing and applet installation possibilities from within the card.

From the security point of view, this clearly renders the current prototype less useful because information that should not be visible outside the card is disclosed. But this will change with the availability of Java cards which conform to the Java card API 2.1 standard.

For WASP agents, we defined a special class to encapsulate the code parts an agent wants to be executed inside a smartcard. Since the code parts need to run on the card, they have to conform to the card's applet class. When downloaded to the card, the applet will be decrypted and installed, so the agent can access it as a 'normal' card applet.

[3] The access rights granted to the associated user account (i.e., the user) at a Web server can be additionally restricted by the user, thus not granting all his or her access rights to the agent.

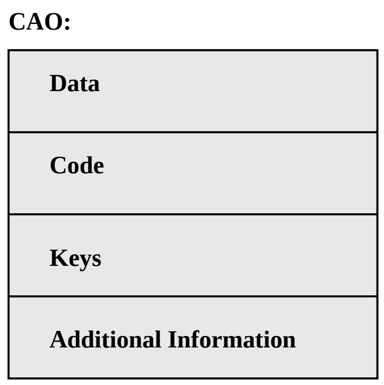

Fig. 2. Structure of an Crypted Agent Object

The encapsulating WASP class `CAO` (Crypted Agent Object, see Figure 2) not only contains the byte code representation of the card applet in a format that can be downloaded to the card, but also contains an object that holds the encrypted object state of the card applet. Inside the card runs the managing entity which is called CAOS (Crypted Agent Object Service, see Figures 1 and 3) manager. This object is responsible for the management of all downloaded CAO entities. Inside the WASP SAE (Server Agent Environment) the card is offered as a service for agents by the SAE's card manager. In fact the local manager is simply a proxy for the CAOS manager residing in the card. The CAOSProxy is not directly accessed by an agent, instead a new instance of a CAOSConnector object is returned to an agent contacting the local card manager. This object manages all CAO instances of its associated agent. The agent requests are mediated to the local card manager which serializes all requests to the card manager. Figure 3 depicts the whole architecture.

To allow the detection of exchanged CAO instances resulting from malicious hosts that extract one COA instance from one agent and put it into another agent, each CAO instance has to carry the agent id of the owning agent. By this way, an agent can detect the exchange at the next non-malicious host.

4 Discussion

The approach we presented, allows agents to carry encrypted code parts. The code parts are contained in the agent in encrypted form only and the clear text form of the code is visible in the smartcard only. Because of this, the local agent environment can only access the encrypted form of the code parts (inside the agent). This is ensured by using public key encryption with certified public keys and can be seen as having a security container or secure store available, which protects everything what's inside the store. Of cause, since the mobile agent environment actually can access the encrypted form of the code parts it can use so-called 'denial of service attacks': The agent environment could simply change random bits of the encrypted parts, so that the local smartcard will not be able to decrypt the data. Another attack would be that the environment does not send the encrypted parts to the local smartcard at all when requested by the agent. Since this kind of denial of service attacks cannot be used to attack the encrypted code in a more reasonable way (e.g., by changing the code in such a way that a decision is made in favour of the local host), we will not consider denial of service attacks and concentrate on the more reaonable case where the agent's code parts contain valuable data, to which a malicous host tries to get illegal access to.

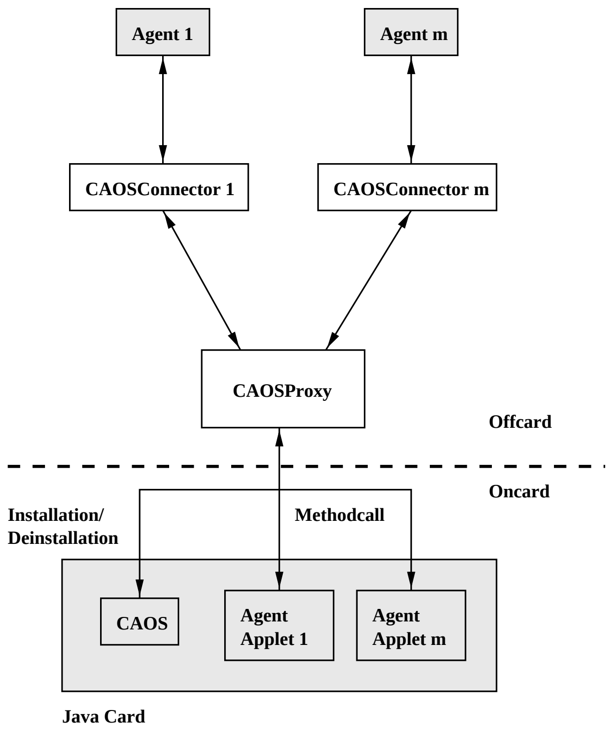

Fig. 3. Crypted Agent Object Architecture

Because the code and data is available in its clear text form inside the smartcard only, the only way for the agent environment to access the code is using the access interface in form of the corresponding card applet interface: The local agent environment could impersonate as the agent that downloaded the code and could call any of the methods of the agent's card applet residing in the card. From within the card, it cannot be distinguished between the situation where the agent itself accesses the code or where the agent environment accesses the code thus acting as agent.

Having this in mind, the question arises, what kind of code can usefully be contained in the security container, and what security precautions should be considered by the programmer. To answer this question we will look at the properties the security container is able to offer to the agent (i.e., what kind of protection is offered).

4.1 Protecting an agent's secrets

Because the data contained in the security container is always encrypted with the public key of a smartcard when outside a card, and because the encryption is done inside the trusted smartcard, the security container owns the following security property: The code and data contained in the security container are protected from unauthorized reading, writing, and modification, unless this is explicitly (directly or indirectly) possible through the access interface. This means:

– Any data put into the security container at agent startup will be kept secret until certain conditions will release the data, which means that the data need not be kept secret any more.

– Any data put into the security container during the journey of the agent, will be kept secret and protected from manipulations by any other host during the rest of the agent's journey. The data is stored securely only, there can be no validity check on whether the data arriving at the secure store is the data sent to the store by the agent (i.e., it cannot be detected from inside the card, if the local environment changed the data).

In the context of electronic commerce this enables that mobile Web commerce agents can carry precious data like electronic money, credit card numbers, or collected price information in a secure way. This is a prerequisite for any usage of mobile agents for electronic commerce. Although our security container is able to protect code and data from unauthorized access when outside the trusted card, some problems remain unsolved.

The access interface: Using the front door to break in. Although it is possible to store code and data securely inside our security store, the data is of no worth, when not accessible when needed. This results in an access interface, which allows an agent to communicate with the security store loaded into the local smartcard. The interface is used to transmit data to and from the security store when needed. As stated already above, the problem arises that the communication interface can be accessed by an malicious host also, thus having certain access to the data and code inside the security store.

Because of this, the code inside the security store has to take this thread into account. This means that the release of any secret data has to be guarded by code that ensures that the data is released only under the desired conditions. For a Web-commerce agent that is able to pay by electronic coins, the release of the coin (i.e., transferring the data bytes from the security store to the agent) should not be possible by calling a method `getCoin(amount)` residing in the security store, but instead this method should only return the coin after some other code inside the security store decided to actually pay at the current location. This decision should also include to store a receipt origination from the current host that provides evidence of the transaction. By this way it is possible to track down any malicious host later.

More generally, the availabilty of the secure store results in a situation where it is possible to have code that calculates and possibly stores an agent's decisions in a secure way, but has to decide on untrusted data. The problem of untrusted input data also arises in the trusted hardware scenario, in which the whole agent system is running inside the trusted hardware: all information resulting from outside the trusted hardware is not trustworthy by definition.

Because of this, it seems to be a good idea, to release sensitive data only when communicating with certified parties (e.g., a certified legacy service attached to the untrusted agent environment). In such scenarios, any data exchanged between the security store and the service should be encrypted for example by public key encryption, after authentication of the certified service. This prevents eavesdropping by the mobile agent system.

Some data (e.g., passwords which should identify the agent at some service) have to be protected forever, which means that they must never leave the secure store: When given away to a malicous host the information could be abused to illegally access the service. To prevent the disclosure of such secrets, a special kind of protocol should be used to provide authentication without actually transferring the secret data. These kind of protocols are known as 'zero knowledge protocols'.

This shows that having a secure store is not enought to solve the malicous host problem. In addition, the code inside the secure store needs to take into account the thread of a malicous host accessing the code and the data through the access interface. Unfortunatly there is no standardized design leading to a secure access interface. This results from possible side effects and from the problem that the provided functionality of the access interface has to be flexible enough to allow the agent to fulfill its task, and needs to be restrictive enough to be able to keep it secure.

Stealing the security container. Since the security container resides in the agent, any malicious environment can exchange the whole encrypted container of one agent with the encrypted security container of another agent. This on the one hand is a denial of service attack, since the agent cannot longer use the container, since the access interface does not match anymore, and on the other hand, precious data is stolen from the agent. But since the container is always encrypted when inside an agent, the malicious system can take no advantage of the content of the container, unless it is able to break the encryption.

4.2 Protecting an agent's itinerary

The availability of a security store also allows to store the agent's itinerary in a secure way. This requires that the smartcard is dedicated to the host it is connected to by 'branding' it with the local host's address. By this way it is also possible to securely record the actual itinerary of the agent. The travel log is kept in a 'secure itinerary' object, and reading it may be protect by a password.

For a fixed itinerary, the agent can retrieve the address of the next hop from its security store. After arriving at the new location, the secure itinerary object is downloaded to the local smartcard and during its initialization compares the address of the host the current card is dedicated to with the target of the migration. If the addresses do not match, security actions can take place. If one malicious host routes the agent to a wrong but non-malicious destination, the agent can detect the malicious host after arriving at the new host. Collaborating malicious hosts can deadlock an agent by simply sending it back an forth on every migration request, and any indication whether the migration ended up at the desired destination available from the secure itinerary on the card can be manipulated by the hosts. Because of this, one of the security actions of the secure itinerary could be that after a certain threshold of reaching wrong destinations, all information inside the security store is encrypted with the home key of the agent only (even destroying some of the valuable data could be a choice), so the next download to any other card will fail. This countermeasure cannot be used to signal the agent that something is wrong, because any communication between agent and its security store can be manipulated by the local environment, but the data inside the store is securely locked up (or destroyed), so nobody can access it any more (e.g., to do black box testing).

For the more usual case of a dynamic itinerary, the agent cannot simply add the new destination to the itinerary stored on the card, because the local host may manipulate the communication to store a wrong destination address. In addition it may route the agent to this wrong address. Since the stored address matches the target address, the manipulation cannot be detected, even if the destination host is non-malicious. To prevent this, the decision of whether a destination is inserted into the itinerary or not has to be made inside the secure itinerary. This requires that any data relevant for this decision has to be sent to the card too. Because there is a probability that the address is not put into the itinerary, the local host cannot force the agent to add a specific address without black box testing, which should be detected by the card with some probability. In addition, choosing the next destination to migrate to by random from the list of hosts to be visited (and not sequentially, if possible in the context of the agents task) can help to prevent successful black box testing.

4.3 Related Work

There are several other research groups that investigate the malicious host problem and provide different solutions for the problem. Their results can be divided into solutions that do not use dedicated, trusted hardware, and solutions that do use trusted hardware.

The approaches not depending on the presence of trusted hardware use cryptographic techniques to protect the agent. Among these are:

- the black box approach presented in [7], which protects information by encryption and code obfuscation for a certain time period,
- the calculation by encrypted functions approach presented in [11], which allows the agents to contain encrypted but executable code that calculates polynomial functions,
- the cryptographic trace approach presented in [13], which allows the detection of manipulations of the agent's code and data,
- the state appraisal approach presented in [3] which allows for checking whether the agent's state is resulting from any legal execution, and
- the object signing and sealing approach presented in [6], which suggests Java objects that can be used to protect data.

Our approach differs from these approaches, since it uses special hardware to enforce security. It can be best compared to the object signing and sealing approach in [6], which describes a general way to have special Java objects that offer an easy way to have data protected by signatures and encryption. The problem is, that these objects are designed for the use in a trusted environment containing other untrusted Java objects. Because of this, the technique cannot be used to protect objects of mobile agents. In that sense our approach can be seen as extension resulting in objects that can also be used in mobile agent scenarios.

The approaches using special trusted hardware, like:

- the Sanctuarity project described in [17], which aims at providing a trusted computing base that contains a whole mobile agent system, and
- the approach presented by [16], which describes the same main idea

differ from our approach in that the protection they aim at requires rather powerful hardware (e.g., a PC card) to host a whole mobile agent system. In contrast to this, our approach protects specific parts of a mobile agent only, while uncritical data and code remains unprotected.

5 Summary

Because of the lack of broad availability of mobile agent enabled hosts, we argue in this paper that the seamless integration of the mobile agent technology into the World Wide Web is crucial for the success and overall acceptance of mobile agents. Especially using mobile agents in electronic commerce scenarios can help pushing the mobile agent paradigm, if the malicious host problem could be solved. In this paper we presented a way of protecting mobile agents for Web-commerce to a certain extend from malicious hosts by the means of multifunctional smartcards. Because Java is used to program agents for most of the existing mobile agent systems, using a Java card as trusted computing base seems to be reasonable. For the WASP project we realized a first prototypical implementation, which also showed, that current Java cards lack some properties that are necessary to have a more secure implementation.

References

1. Chess D., *Security Issues in Mobile Code Systems*, in: Giovanni Vigna (Ed.), Mobile Agent Security, LNCS 1419, 1998, Springer, pp 1-14
2. DigiCash Inc., *eCash Online Documents*, available from `www.digicash.com/index_e.html`
3. Farmer W.M., Gutman J.D., Swarup V., *Security for Mobile Agents: Authentication and State Appraisal*, Proc. of the 4th European Symp. on Research in Computer Security, Rome, Italy, September 1996, LNCS No. 1146, Springer, pp 118-130
4. Fünfrocken S., *How to Integrate Mobile Agents into Web Servers*, Proceedings of the WETICE'97 Workshop on Collaborative Agents in Distributed Web Applications, Boston, MA, June 18-20, 1997, pp 94-99
5. Fünfrocken S., *Integrating Java-based Mobile Agents into Web Servers under Security Concerns*, Proceedings of the Hawai'i International Conference on System Sciences, January 6-9, 1998, Kona, Hawaii, Volume VII, pp 34-43
6. Gong L., Schemers R., *Signing, Sealing, and Guarding Java Objects*, in: Giovanni Vigna (Ed.), Mobile Agent Security, LNCS 1420, 1998, Springer, pp 206-216
7. Hohl F., *Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts*, in: Giovanni Vigna (Ed.), Mobile Agent Security, LNCS 1420, 1998, Springer, pp 91-113
8. Medvinsky G., Neuman C., *NetCheque and NetCash Online Documentation*, available from `gost.isi.edu/info/netcheque/documentation.html`
9. Mondex International Ltd., *Mondex Technology*, available from `www.mondex.com`
10. Plamer E., *An Introduction to Citadel - a secure crypto coprocessor for workstations*, Proceedings of the IFIP SEC'94 Conference, 1994
11. Sander T., Tschudin C., *Protecting Mobile Agents Against Malicious Hosts*, in: Giovanni Vigna (Ed.), Mobile Agent Security, LNCS 1419, 1998, Springer, pp 44-60
12. Sun Microsystems, *Java Card 2.1 API*, Online Documentation, available from `java.sun.com`
13. Vigna J., *Cryptographic Traces for Mobile Agents*, in: Giovanni Vigna (Ed.), Mobile Agent Security, LNCS 1419, 1998, Springer, pp 137-153
14. Visa International, *SET Specification*, available from `www.visa.com/nt/ecomm/set/intro.html`
15. White J.E., *Telescript Technology: The Foundation for the Electronic Marketplace*, Whitepaper by General Magic, Inc, Sunnyvale, CA, USA
16. Wilhelm U., *A Technical Approach to Privacy based on Mobile Agents Protected by Tamper-resistant Hardware*, Ph.D. Thesis, EPFL, Lausanne, 1999
17. Yee B., *A sactuarity for mobile agents*, in: Vitek J. and Jensen C. (Eds.), Secure Internet Programming: Security Issues for Mobile and Distributed Objects, LNCS, 1999, Springer
18. Yee B., *Using Secure Coprocessors*, Ph.D. Thesis, Carnegie Mellon University, 1994