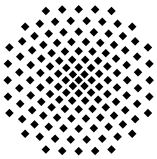


Universität Stuttgart
Fakultät Informatik

An Approach to Solve the Problem of Malicious Hosts

Fritz Hohl

Email: Fritz.Hohl@informatik.uni-stuttgart.de

Institut für Parallele und Verteilte
Höchstleistungsrechner (IPVR)
Fakultät Informatik
Universität Stuttgart
Breitwiesenstr. 20 - 22
D-70565 Stuttgart

An Approach to Solve the Problem of Malicious Hosts

Fritz Hohl

Bericht Nr. 1997/03
March 1997

An Approach to Solve the Problem of Malicious Hosts in Mobile Agent Systems

Fritz Hohl (Fritz.Hohl@informatik.uni-stuttgart.de)
Institute of Parallel and Distributed High-Performance Systems (IPVR),
University of Stuttgart, Germany

Abstract

Mobile agents are often described as a promising technology moving towards the vision of a widely distributed scalable electronic market. The deployment of electronic services, especially in the area of electronic commerce, raises essential questions closely related to security issues. This paper tries to address these issues by providing a taxonomy of security domains within mobile agent systems. The identified areas comprise protecting hosts against malicious agents, protecting agents from other agents, protecting hosts from other hosts, and protecting agents from malicious hosts. Whereas the first three security issues can be solved by applying traditional security mechanisms, new security techniques have to be developed to protect agents from malicious hosts. The paper analyzes possible attacks of hosts and presents, based on this analysis, an approach to prevent malicious attacks. The approach, which is called Code Mess Up, consists of a combination of two mechanisms: The first mechanism dynamically generates a new and far less understandable version of the agent code. The second mechanism restricts the lifetime of the agent's code and data. It is shown that the application of these two mechanism can significantly enhance the protection of agents against malicious hosts.

1 Introduction

Mobile agent systems are expected to become the base platform for an electronic services framework, especially in the area of Electronic Commerce. In this application area, security is a crucial aspect since all parties involved require the confirmation that none of the other parties will break the rules without being punished. This requirement is not always fullfilled even in the traditional, non-electronic commerce, but the anonymity of a worldwide communication network and the ease of the automatic exploitation of security gaps in electronic applications make it necessary to meet this demand when it comes to commercial transactions done by computers.

Mobile agents are units that consist of code, data and control information (e.g. thread states). Mobile agent systems are platforms that allow mobile agents to migrate between different nodes of the agent system. From a more technical view, mobile agents can be compared to programs that migrate to nodes autonomously, while nodes offer the runtime environment of these programs that include the program interpreters. Like in Mobile Code systems like the Java applet system, one aspect of security is the protection of the interpreter, the node or *host*, against possible attacks of the mobile agent. Therefore, some of the security mechanisms developed in this field can also be applied to mobile agent systems, e.g. sandbox security, i.e. the need of authorizing security-sensitive commands like the deletion of a file by a designated component. Other security mechanisms like authentification of single agents, do not have a predecessor in mobile code systems and have to be designed out of standard cryptographic techniques like encryption or digital signatures.

The reverse security issue, the protection of an agent from possible attacks by its hosts is new as there are barely other areas where this aspect is important. Nevertheless the protection of mobile agents from malicious hosts is — at least from the viewpoint of the owner of the agent — as important as the protection of the host from malicious agents. Apart from organisational solutions, no technical approaches to solve this problem exist so far. The solubility of this problem is even estimated to be very low.

This paper presents an approach to solve most of the aspects of this problem, the problem of malicious hosts. This approach will cost both execution time and communication bandwidth and will require some time-critical restrictions, but gives the agent the possibility to do some security sensitive work without the danger of an immediate exploitation of sensitive data by the host.

The rest of the paper is organized as follows: After giving an overview over related work, in Section 3 several security areas in mobile agent systems are identified and characterized according to their solubility by existing techniques. Section 4 focuses on one of these areas — malicious hosts —, and presents approaches, that have been taken into consideration so far to solve this problem. Section 5 analyses the problem of malicious hosts and Section 6 proposes a new approach to solve this problem. The two mechanisms, that are used by this approach, code messup and limited lifetime of agent code and data are discussed in Section 7 and 8. The article closes with a conclusion and some remarks about future work.

2 Related work

Since the area of mobile agents is rather new, there are only few publications in the field of security in mobile agent systems. In [FGS96a], Farmer et. al. employ two application scenarios to analyze the new threats that were added by the migration of agents, list security goals and classify them into impossible goals like "Will an interpreter run an agent correctly?", easy goals like the authentication of authors and senders of agents, and possible, but not easy goals like "Can a sender restrict his agents flexibility?". In [FGS96b], the same authors propose an architecture for mobile agents, that allows agents to be authenticated and authorized using a "trust theory". In this theory, that is called state appraisal, hosts can give different agents different permits, depending on the current state of the agents and on the task they have to do at this host. [IBM95] lists a short menagerie of threats like trojan horses, viruses and worms. Vitek analyzes in [Vit96] the problems of agents when using object-oriented programming and proposes a tupelspace approach as a secure communication means. Gray describes in [Gra96] the authentication mechanisms developed for AgentTcl. Tardo and Valente ([TV96]) describe the security infrastructure realized in Telescript. Chess et al. propose in [CGH95] a framework for mobile agents and describe observations on security issues of mobile agents. The authors state that it is impossible to verify, that an arbitrary program is not a virus. Furthermore, they list attainable goals like origin authentication or code integrity. Their conclusion is, that although security represents one of the cornerstone issues, not all goals can be achieved without the use of trusted hardware. Ordille asks in [Ord96] "When agents roam, who can you trust" and answers that a host can only trust agents that have been never on untrusted hosts before.

A huge amount of papers exist about "traditional" security, i.e. such that do not consider mobile agents. Since some security areas (like communication security, and the authentication of immobile entities) do not need new techniques, a lot of papers describe mechanisms that can also be used in mobile agent systems. The reader is refered to [Che97] for an analysis of TCP/IP threats, [Sch96] for elementary security techniques or [Sta95] for a more network-centric view.

3 Security areas of mobile agent systems

Mobile agent systems are platforms that allow mobile agents, to migrate between different nodes of the agent system. On a node, a mobile agent can interact with other agents locally (and globally in some systems) or it can migrate to another node. Interaction consists in communicating with other agents, requesting or providing services or creating new agents or terminating existing ones. Nodes, or mobile agent *hosts* are the "building block" of the agent system; each host can be maintained by another institution.

When it comes to security, this field can be divided in different security "areas", e.g. by identifying the different attack "fronts". These attack fronts distinguish the "parties" involved (e.g. the agent and the host) and allow to identify the possible attacks between the parties. In this paper, we distinguish four main security areas (Fig. 1):

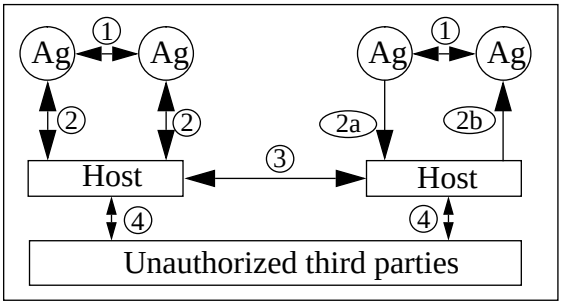


Fig. 1: Security areas of mobile agent systems

1. Security between two agents
2. Security between agents and hosts
3. Security between hosts
4. Security between hosts and unauthorized third parties

In the following, every security area will be characterized by mechanisms, that have to be offered in this area and by techniques, that may implement the mechanisms.

3.1 Security between two agents

The problem addressed here is the security of agents against other, malicious agents. Attacks of those may include code and data manipulation by having physical access to the code and data areas of the attacked agent, masking of agents (i.e. faking a wrong identity), cheating (e.g. using a service without paying for it) and denial-of-service (e.g. by filling a message buffer).

As this area does not address new security issues compared to requirements imposed in "traditional" distributed systems, mechanisms that prevent such attacks already exist, e.g.:

- the use of an agent language, that does not allow others to have physical access to "private" data, such as Java or the use of isolated address spaces
- authentication of agents, e.g. by using digital signatures
- employment of a service contract mechanism (e.g. [SL95])

3.2 Security between agents and hosts

This area can be divided in two subareas, since the relationships between agents and hosts is not symmetric: Agents are basically programs that are executed by the hosts, the hosts consists of agent code interpreters or at least runtime environments.

3.2.1 Security of hosts against malicious agents

The attacks of malicious agents are basically the same

as those against other agents (physical manipulation, masking, cheating, denial-of-service). The difference is the greater impact of hosts, since they control the execution of agents. This enables host to stop suspicious agents at any time.

The mechanisms against attacks of malicious hosts are:

- the usage of "secure" languages or isolated address spaces
- authentication by using digital signatures or other cryptographic techniques like the one described for the mobile agent system AgentTcl ([Gra96])
- the usage of accounting and contract mechanisms
- the usage of resource control mechanisms such as limited resource "accounts" and runtime restrictions

The second mechanism embodies a problem compared to "traditional" authentication: As a mobile agent system as a whole may be too large, it cannot be assured that there may not exist two or more agents using the same identification. This security gap can be exploited either by the agent owner to create copies of an existing agent or by attackers (e.g. hosts) that are able to "read" existing agents, such as hosts.

3.2.2 Security of agents against malicious hosts

In this security area, the attacker is the host itself. The host can observe every step the agent takes, read every bit of code, data and state, and even manipulate the way the agent works, as it, among other things, interprets the agent code. Attacks include "normal" threats like masking and cheating, but also privacy breaking (the host may read secret keys or electronic money), code, data and state manipulation (e.g. taking away electronic money or implanting virus code, and executing the agent in a way other than specified by the language. Denial-of-service attacks are possible, too, but it is clear that a host can easily deny the agent to work by simply not to execute it.

The existing security mechanisms are not easily applicable here since the aspect of securing a program against a malicious interpreter is a rather new aspect. The security of agents against malicious hosts will be discussed in detail in Section 4.

3.3 Security between hosts

Like in the first section, possible attacks between hosts include masking, cheating and denial-of-service. However, since hosts are stationary entities, the common mechanisms for authentication, accounting and interaction control can be applied here without modification.

3.4 Security between hosts and unauthorized third parties

We have to consider in this area the security of communication between two hosts over an insecure network. Other aspects like masking are included in the areas above. The attacks and mechanisms here are well-known and do not pose new questions. Therefore, this area is omitted in this paper. Interested readers will find overview and detail information in a couple of books, e.g. [Sta95], [Sch96] and [Che97].

4 The problem of malicious hosts

As we have seen, widely distributed, open mobile agent systems are intended to be used as a base for real-world applications, especially in the area of electronic commerce. As they transport sensitive information such as secret keys, electronic money and other private data, they require the security of agents also against potentially malicious hosts.

On the other hand, this problem has rarely occured in applications so far since a program either was invoked by the same authority that maintained the computer environment or because the maintaining institution was trustworthy. Therefore, barely any solution exist in this security area.

The only other area with comparable problems is the security of software against manipulation by users, in particular against unauthorized copying. Two approaches have been developed to solve this particular problem. The first one tries to bind the execution of a program to the existence of a special medium or piece of hardware (dongle). Therefore, the software tries to find out periodically or at the start of the program whether it can verify the existence of the dongle. Since each user is provided with only one dongle per program and because the software does not start without it, the program can be copied, but not used without the dongle. Unfortunately, this approach is not applicable to our problem since it solely solves jthe particular problem of the unauthorized execution of a program. However, this approach does not provide any means against the potential manipulation of the program by the user, and it is indeed possible to break this kind of security by "cracking" the program, i.e. by removing the part of the code that verifies the existence of the hardware. The second approach also requires the employment of special hardware. This time, it is not a special addendum, but a secured execution unit that protects the software from being manipulated. One of these devices, the Citadel coprocessor [Pal94], was designed to act as a fast encryption processor (for the DES algorithm, see [Sch96] for details), but also includes a whole microprocessor and is able to run applications inside the chip. As it is protected against manipulation from the outside, even again physical one, it can act as a secure "island" inside a hostile environment, and this also includes the maintainer of the computer system.

Apart from using trusted hardware, few approaches exist so far to solve the problem of malicious hosts. To make the problem worse, the *solubility* of this problem besides the use of trusted hardware is estimated in the literature as very small or even zero. Farmer, Guttmann and Swarup ([FGS96a]) find it impossible to protect agents from read attacks by the host, and recommend

not to let an agent carry secret informations. Harrison, Chess, and Kershenbaum ([HCK95]) find it impossible to prevent agent "tampering" unless trusted (and tamper-resistant) hardware is available in hosts. Ordille claims in [Ord96] that it is not possible to protect the agent's data while travelling, since hosts can alter an agent's data and send the changed agent into the network. Other papers about security in mobile agent systems barely mention this problem and, to our knowledge, none of these authors proposes a full-fledged technical solution.

Although technical solutions, i.e. such that try to secure agents by technical means, are not known yet, non-technical solutions for non-open systems exist. As we have seen above, we can reduce the problem of malicious hosts to an inner-institutional problem, if we let hosts being maintained only by trustworthy parties. This approach seems to be taken by the only existing commercial mobile agent system, AT&T's PersonaLink [Jer94], which uses General Magic Telescript technology [GM96], and which offered mainly an advanced email service. Further services could have been offered by other companies, but this aspect never seems to have been realized. The problem of this *organisational* approach is, that the resulting mobile agent system is not "open" any more. It is thus not possible for a foreign service provider to connect a new host as part of the platform to the mobile agent system. Unfortunately, this organisational approach also reduces the usefulness of a mobile agent system as an infrastructural middleware component, since it depends directly on the dissemination of the system and on the interconnectivity to existing services and information sources. As a consequence, a single platform provider has to start with a system that already has a critical mass in order to get third parties using this system as a platform. The advantage of the organisational approach is, that generally, no host is malicious, and the (hopefully) rare events, that hosts are "turned up" by employees or foreign attackers are handled and backed up by the maintaining organization.

One could imagine to obtain a mobile agent system without the problem of potentially malicious hosts by an organisational approach that allows interested third parties to connect their infrastructure to the system by establishing an institution that examines these parties and that offers an "ombudsman" service to both service and platform providers on the one side and clients on the other side. The problem is, that it may be extremely difficult to detect whether a given damage is caused by an agent (and thus the client) or by the host (and thus the platform providers). Moreover it is unclear how to get compensation for it in a world-wide distributed system that covers a lot of countries, and therefore juristic frontiers.

Before we come to an analysis of the features of the problem of malicious hosts, an example is presented that will be used in the rest of the paper.

An example of an agent

An agent shall buy some flowers for a human user. Therefore it inquires a trading component for a list of addresses of service agents that offer the `BuyFlowers` service. The agent migrates to every service provider of the list, asking for the price for the requested flower package. If this price is lower than a user-specified maximum price and lower than the lowest price so far, the agent stores the new lowest price and the address of the service provider. After having asked all the providers of its list, the agent migrates to the provider with the lowest price and buys the flowers using electronic cash that it is carrying.

This "purchasing agent" carries sensible data that determines its behavior (e.g. the lowest price and the address of the best provider) and is a typical example for agents that carry confidential data.

Some authors would argument, that such an agent should not exist, since it is comparable to a tourist which obviously carries cash in the wallet and which roams through a quarter that reacts on this behavior in an unpleasant way. Therefore, an agent, as well as a well-advised tourist should never carry sensitive information through unknown territory. As we will see, this conclusion does not need to be true, if we convert this information into a form that villains cannot use easily. Back to our example.

The purchase agent contains a data and a code area. Entries in the data area may include:

```
Address home = "PDA, sweet PDA"
Money wallet = 20$
float maximumprice = 20.00$
good flowers = 10 red roses
Address shoplist[] = empty list
int shoplistindex = 0
float bestprice = 20.00$
Address bestshop = empty
```

The central procedure `startAgent`, that is called by the host every time the agent arrives, could look like this:

```
1  public void startAgent() {
2
3    if (shoplist == null) {
4      shoplist = getTrader().
5       getProvidersOf("BuyFlowers");
6      go(shoplist[1]);
7      break;
8    }
9    if (shoplist[shoplistindex].
10     askprice(flowers) < bestprice) {
11     bestprice = shoplist[
12                  shoplistindex].
13                 askprice(flowers);
14     bestshop = shoplist[
15                     shoplistindex];
16   }
```

```
17   if (shoplistindex >=
18            (shoplist.length - 1)) {
19     // remote buy
20     buy(bestshop,flowers,wallet);
21   }
22   if (shoplistindex >=
23            (shoplist.length - 1)) {
24     go(bestshop);
25   }
26   else {
27     go(shoplist[++shoplistindex]);
28   }
29 }}
```

By now, we have seen that the problem of malicious hosts is a relevant one, and that no satisfying solutions exist so far. Therefore, we will now have a closer look on the problem by analyzing its features.

5 Analysis of the problem of malicious hosts

For the analysis of the problem of malicious hosts, we will examine the possible attacks a malicious host could execute on this agent. After that, we will discuss how these attacks could work and analyze which circumstances lead to the possibility of the attacks.

Possible attacks by the host

Now the possible attacks of a malicious host are examined:

1. Spying out code

The code of the agent has to be readable by the host. Although this requirement can be restricted to the next instruction at a single point of time, this is no real help since some hosts see almost all the code because they execute much of the code (like e.g. the last host that is visited in our example). If the code of the agent is not significant for every agent, but for a whole class of agents, the whole code of the agent may be known even before execution time. If an agent is constructed out of standard building blocks (which is no bad idea in terms of code migration costs and ease of agent construction), also the detail specification is available for building blocks like libraries or classes. Furthermore, these blocks can be explored by blackbox tests. Knowing the code leads to knowledge about the execution strategy of the agent, knowledge about the exact physical structure of code and data in the memory of the host and sometimes (by using data statements like initial variable assignments) to knowledge about parts of the agent data.

2. Spying out data

The threat of a host reading the data of an agent is very big as it leaves no trace that could be detected (although this is not necessarily true for the consequences of this knowledge, but they can occur a long time after the visit of the agent on the malicious host). This is a special problem for data classes such as secret keys or electronic cash, where the simple knowledge of the data results in loss of privacy or money. In our example, the money variable would be security sensitive when it is represented in a way that the binary number of the “coin” **is** the money and therefore can be used as real world cash. But there are also other classes of data, which can be used for an attack although they have not the nature of classes like e-cash. In our example, the knowledge of the maximum price or the best price so far can be used by a malicious host to offer flowers for a slightly lower amount than the competitors, although the regular price is much lower.

3. Spying out control flow

If the host knows the entire code of the agent and its data, it can determine the next execution step at any time. Even if we could protect the used data somehow, it seems rather difficult to protect the information about the actual control flow. This is a problem, because together with the knowledge of the code, a malicious host can deduce more information about the state of the agent. In our example, we can recognize, whether an offer is better or worse than the best offer so far by watching the control flow, even if we could not read any variable.

4. Manipulation of code

If the host is able to read the code and if it has access to the code memory, it can normally modify the program of an agent. It could exploit this either by altering the code permanently by implanting a virus, worm or trojan horse, or, temporarily, by altering the behavior of the agent on that particular host only. The advantage of the latter approach consists in the fact, that the next host of the migrated agent cannot detect a manipulation of the code (since it is not modified). Applied to our example, a malicious host could modify the code of the agent with the effect that it prefers the offer of a certain flower provider, regardless of the price.

5. Manipulation of data

If the host knows the physical location of the data in the memory and the semantics of the single data elements, it can modify data as well. In our example, the host could shorten the shop list after setting the offer of the local flower provider as the lowest, regardless of the correctness of this information.

6. Manipulation of control flow

Even if the host does not have access to the data of the agent, it can conduct the behavior of the agent by manipulating the control flow. In our example, the host could simply alter the flow at the second or third if statement, resulting in working incorrectly and e.g. taking the price of the host as the lowest.

7. Incorrect execution of code

Without changing the code or the flow of control, a host may also alter the way it executes the code of an agent, resulting in the same effects as above.

8. Masquerading

Normally, it is the deed of a host that sends an agent to

a receiver host to ensure the identity of that receiver (and that what it will do in most cases). This attack is listed here, because a third party may eventually intercept or copy an agent transfer and start the agent by masking itself as the correct receiver host. This attack may result in one of the other attacks, e.g. in reading code and data of the agent.

9. Denial of execution

As the agent is executed by the host, i.e. passive, the host can simply not execute the agent.

10. Simultaneous attacks

Most of these attacks can be performed simultaneously, which normally leads to a greater attack potential, e.g. when knowing the maximum price and manipulating the control flow in order to let the agent buy the flowers at the maximum price.

Attack analysis

Now, the different attacks will be analyzed by giving some observation about the nature of the threats together with some remarks about how potential countermeasures could work.

ad 1 (Spying out code)

We can distinguish here between the knowledge of

- what the code as a whole does exactly, i.e. the detail specification and
- what each line of code does exactly

As it was mentioned before, the first point cannot be protected if we use standard building blocks such as libraries, and which is not very security sensitive. In contrary to that, the second point is very sensitive, as it is the knowledge of *where to start* in order to get a modified behavior.

ad 2 (Spying out data)

Here, we can distinguish also between

- the knowledge about which data elements exist
- the contents of these elements
- the coding of the elements in the memory

Again, the first point cannot be protected, but the second has to be protected in order to evade read attacks. Again, the third point allows the attacker to know which bit to modify in order to modify the data element, an attack which is well-known in the area of computer games, where players can “poke” specific values into the memory (often supported by special hardware), resulting in getting more game lifes or reaching higher game levels. The requirement of the second point, the protection of the contents of data elements against read attacks can be lowered from a time-independent requirement to a protection for a guaranteed amount of time if it is possible to attach an expiration date to every element after which this element cannot be used any longer.

Another observation is that data can be divided in three classes: The data that does not need to be protected from read attacks (e.g. the home address), the data that need to be protected only for the execution interval at a host (e.g. the maximum price), and the data that must be kept secret from the attacker (e.g. a secret key).

ad 3 (Spying out control flow)

The different points here are the knowledge about

- the actual control flow after the end of the execution
- the relation of control flow and execution semantics

Again, the first point cannot be protected (as the host executes the agent), but if it is possible to solve the second one, the host would not know where to manipulate in order to alter the control flow in a directed manner.

ad 4 (Manipulation of code)

Permanent code manipulation is impossible if you can cryptographically sign the code of the agent. If the code does not change over execution time (i.e. over the whole lifetime of the agent), this is no problem, as the owning party can sign the agent at start time. If it is possible to use standard libraries, it is very easy to sign them by the library programmer and store them at different locations of the agent system, resulting in a more efficient code migration. If the code changes over the execution time, other cryptographic models can be used, but the performance and programming advantages make the use of standard libraries more attractive anyway.

As we have seen in the first remark, code can only be manipulated in a directed manner, if the attacker knows what each line of code does exactly and where this line is stored. If we could protect this information from the host, even temporary code manipulation attacks would be impossible.

ad 5 (Manipulation of data)

Data elements can be divided in such elements that might be modified by correct code during a visit on a host and elements that will not be modified. The latter portion can be signed (or even crypted when the code does not read it) and is therefore protected during that visit. The first class of data could also be protected, if it were possible to hide the information about the coding of the elements in the memory.

ad 6 (Manipulation control flow)

Like in the last two paragraphs, we could protect the control flow from being manipulated if we could hide the information about the relation of control flow and execution semantics.

ad 7 (Incorrect execution of code)

Is solved if the manipulation of code and control flow can be protected.

ad 8 (Masquerading)

Any party can authenticate another party if it can verify the existence of a secret date of the other party undisturbed, i.e that the authenticating party has to be autonomous from the other party. Unfortunately, this is a problem for agents when trying to authenticate their

hosts as they were executed by the hosts. If we could give an agent its autonomy back then even a passive component like a mobile agent could check its host. In this context, autonomy means that the host cannot read at least some of the agent data (e.g. a public key), and that it cannot modify the code and control flow of the agent.

ad 9 (Denial of execution)

Of course, a host may deny to execute any agent, but if we can construct an organizational framework that forces the host to proof the execution of an agent, his attack could be avoided. One way to do this is to e.g. marking unwilling hosts as "bad" with the consequence that they do not receive agents any more. This approach seems not to be of real concern, as this attack is rather obvious and does not lead to bigger problems like the loss of privacy or money and cannot be distinguished from a failure of the host. If this kind of approach is followed, the proof of execution could be a data element that is generated periodically by the agent, and which can be probably protected by hiding the creation process in the code.

Summary

As seen above, most of the security problems can be solved if the host is not able to determine the relation between single lines of code and their semantics and the relation between memory bits and the semantics of data elements, respectively. A host can of course modify code, data and control flow anyway, but not with a computed effect. This results for a host in three choices: it can either execute the agent undisturbed, execute the agent by switching some bits, not knowing about the effect on the execution or the host can take the agent without executing it.

Unfortunately, a program together with its inital code can be reengineered by a human. A good example are the crackers of the eighties, people that were able to find and remove protection code implanted in software such as games. Although game programmers took strong effort to hide protection details, advanced crackers successfully broke every protected program, and they had not a detailed specification of the code as we will have with standard libraries. This was only possible because the crackers had sufficient time to analyze a program. If we transfer this knowledge to mobile agents, we can state that a human is able to reveal any of the relations listed above and to write a tool that allows manipulating code, data and control flow if it has enough time.

6 Code Mess Up: an approach to solve the problem

An attacker needs a certain amount of time to read the data, understand the code and, thereafter, manipulate both in a meaningful way. The basic idea of the approach described now is simply not to give him enough time to do this. This can be achieved by a combination of the following two ideas: *code mess up* and *limited lifetime of code and data*.

To give an impression of this approach, an example shows a typical "lifecycle" of our example purchase agent which is protected by this mechanism:

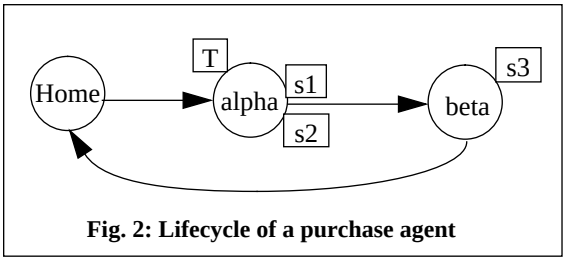


Fig. 2: Lifecycle of a purchase agent

Our agent starts from its home host (Home) and migrates to host alpha. Once arrived, it contacts a trader agent T, which provides a list of addresses of agents that offer the flowershop service. Two agents, that offer this service (s1 and s2) reside at the same host (alpha), so our purchase agent contacts them, asking for their price for the flowers. Thereafter, the agent migrates to the last address of its list, which points to an agent residing s3 on host beta. Again, the agent asks for the price. With the gathered information, the agent wants to migrate to the service provider with the lowest price, which is s3 (therefore, it does not need to migrate). The purchase agent acquires its flowers with some electronic cash. Finally, our agent migrates to its home host.

Let us take a closer look on how the agent got protected. For the first migration, the agent's home host constructs a new form of the agent's code which does exactly the same as the one before, but which "looks different", i.e. has another inner structure. At the same time, the home host takes the agent's data, mixes the original data elements up and distributes this mixture to a list of new data elements. The new form of the code reflects this data conversion by using the right "decryption" code instead of just naming the original variables. Before the conversion, expiration dates are attached to the data elements which are used for interaction with the "outside world" (i.e. outside the agent) in a manner, that no one can falsify the dates by using digital signature mechanisms like DSS (compare, e.g. [Sta95]). This means that electronic cash now also bears an expiration date, secret keys are valid only for a certain amount of time and so on. Finally, the home host signs the code of the agent together with another expiration date. Then the agent is sent to host alpha. We assume now hosts alpha and beta to be malicious, i.e. to attack the agent. As the host (or the human which has access to it) has now first to analyze the agent, some seconds pass (the problem for the malicious host is, that it has never seen the form of this code before, although it knows the original code very well). We further assume that the expiration date of both the code and the data elements expires as they are set only to a few seconds. The host, after having analyzed the agent, can arbitrarily read and manipulate data. But if it tries to use the read data, or to relay the manipulated agent, the receiving host will reject the data or the agent

because their corresponding validity dates are expired. Although the host is malicious, it invoked the agent normally, and after some time,it migrates to host beta, where the same game goes on. Since also host beta needs more time to analyze the agent than code and data are valid, beta cannot do anything with it. Even if host alpha and beta collaborate, they cannot crack the agent in time as it does not matter whether host alpha or both hosts exceed the expiration dates. The only possibility to attack the "turtle shell" of the agent is when it communicate with others, e.g. the host or other agents, as long as the agent cannot use crypted communication. The usage of cryptography in communication with other agents is useless when the communication partner is an "agent" of the malicious host or when the host can attack the other agent.

The underlying idea of using expiration intervals and conversion mechanisms can be used in mobile agent systems if:

1. we can find the minimal time for "cracking" the new code and data
2. we set the expiration date of data elements to this duration
3. the agent can do an significant amount of work in this interval
4. we can periodically create new code, which
   5. is created automatically, i.e. by a tool
   6. does the same as the old code
   7. is as hard to analyze both by tools and humans, i.e. that creates a new relation between elements as variables or lines of code, and application semantics
8. the code variations build up a large space (i.e. one with many variants)
9. the usage of data that bear expiration dates does not restrict the applications too much

If we can meet these requirements, we would obtain an agent system where agents can be protected against

- being spied out/directed manipulation of code before the expiration date
- being spied out/directed manipulation of sensitive data before the expiration date
- being spied out/directed manipulation of control flow before the expiration date
- directed incorrect execution of code

There is no protection after the expiration date, therefore all effects that can occur from the knowledge of these elements and their potential manipulation have to be handled by not accepting "expired" code and data by every host of the system (if a host accepts expired code or data, it has a problem, because no other host will accept them, so the host itself has a vital interest in rejecting expired elements). Of course the host may modify code, data and control flow at its will, but as it cannot see what it is doing, it cannot forsee the effects, as well as it can read the lines of code and the data bitstring, but this knowledge is not easily connected to the knowledge of the essential program structure or the knowledge of semantic data units.

As the agent is now autonomous again for a certain interval, it can also authenticate the host (by bearing a public key of it and checking the existence of the private key by the known mechanisms), so masquerading is not possible any more.

We will now concentrate on the question, whether and how the above requirements could be met.

If we have mechanisms that allow to derive a new messed up code version from the original automatically, we should be able to determine the minimal crack time of humans without tools by measuring the time needed from test "crackers". This duration will last at least some seconds and should grow at least linear with the lines of code that have to be read. The minimal time for an automated approach can be determined with the same mechanisms that allow to compute the equivalent time for the decryption of cryptographic data, i.e. we can determine this time by constructing an corresponding mechanism. So if we adapt the mechanism, we should have at least a few seconds as the expiration duration, which should be enough to let the agent do some useful work.

Requirements four to eight depend on the way, the mechanisms for constructing new versions out of old one work.

7 Mechanisms to "mess up" code

We need a way to transform a "normal", i.e. good readable and understandable piece of code (e.g. a part of the standard library) to a form which is as less readable and understandable as possible, i.e. we want to find a way to *mess up* code. There are barely mechanisms to mess up code in a structured way (although some might say that such a mechanism is called "ordinary programming"), but there is some research in the field of software engineering, that tries to figure out how code has to look like to be readable, and we simply can try to invert these guidelines. Such guidelines can be found e.g. in [KP78] or [SPC89] and they often state things like:

- Use variable names that mean something.
- Modularize. Use subroutines.
- Choose a data representation that makes the program simple.

Code mess-up mechanisms can use these guidelines by creating code that violates these directives. One potential mechanism, which can be called *variable recomposition,* takes the set of program variables, mixes the contents of the variables up and creates new variables that contain some bits of data from some of the original variables and adapts the corresponding variable accesses in the program code. In Fig 3a, you can see the original variable access, Fig. 3b defines a scheme for recomposing two new variables, v23 and v19 from the contents of three of the original variables. The new var-

iables access code, Fig. 3d, can therefore be created automatically (given the recomposition scheme) by using conversion functions (see Fig. 3c) that create the original values out of the new variables. As there is no direct relationship between variables and processing model elements (e.g. the maximum price from our example), the variable names do not mean anything anymore and the data representation is rather complicated.

Fig. 3a: original variable access

```
8      buy(bestshop,flowers,wallet)
9      go(home)
```

Fig 3b: variable recomposition

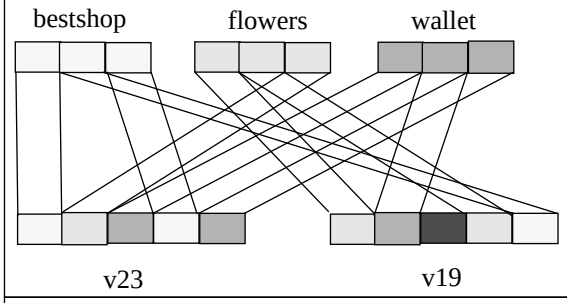


Fig 3c: conversion functions

```
public Address c7(Bitstring b)
public Good c4(Bitstring b)
public Money c3(Bitstring b)
public Address c34(Bitstring b)
```

Fig 3d: new variable access

```
7   buy(c7(v23[0]+v19[4]+v23[3])
      ,c4(v19[0]+v19[3]+v23[1]),
       c3(v23[2]+v19[1]+v23[4]))
8   go(c34(v21[4]+v19[2]+v21[2]))
```

Another family of mechanisms, *structure dissolving*, tries to eliminate program structure like blocks or procedures and creates a piece of code that contains almost no inner structure anymore. Some of the mechanisms that belong to that family are:

- dissolving small variable scopes into global ones
- replacement of procedure calls by procedure code
- replacement of blocks by using “goto”-like statements

The possibilities of structure dissolving is restricted by the “outer” structure of the code, i.e. by procedures and functions that are visible to the outside world and which have therefore to exist in the program.

The last presented mechanism is called *conversion of compile-time control flow elements into run-time data dependend jumps.* Control flow elements like `if` and `while` statements allow the programmer to imagine the potential control flow even at compile time as these statements make control flow explicite. If we convert these elements into a form that is dependent of the content of variables, the control flow cannot be determined as easily as before. This dependence can be achieved by the usage of jumps that are bound to variable contents, e.g. switch-statements. The effect can even be strengthened by using complex variable expressions instead of using simple variables.

While the Software Engineers give us statements about how to mess up code in order to restrict the timely comprehension by humans, there is a related area which is especially interesting for our purposes: the field of reengineering because there the aim is to transform bare code to a documented form of software, as this is exactly what we want to prevent, and as it can give us valuable informations about what aspects are hard to analyze and the role of tools in doing the transformation. For information about which approaches exist in the field of program understanding, the reader is refered to e.g. [Rug96].

Another source of information about how mess up mechanisms should look like is the work about automated code analysis in the field of programming languages and compiler construction (see e.g. [ASU86]). An example of an algorithm that can be used for code mess-up purposes is the *elimination of common expressions* where, at compiletime, expressions, that have been computed before, are used for replacing expressions that compute the same value. This mechanism results in fewer code, higher speed and, important for us, a more difficult program, as some semantic relationships of the original program might be deleted by this optimization technique. As the target of such techniques is the optimization of code in terms of speed and space, software engineering values like readability might be violated (which does not matter as these are compiler mechanisms, the programmer will hardly ever see the optimized code).

Another example of the usage of mechanisms for optimizing code are the approaches to detect dead, i.e. unused code. This time, we do not want to use this mechanism for our purpose, but we want to prevent the successful usage of this technique as the extistence of dead code also makes it harder to understand code. Therefore, when we want to insert dead code into an agent, we have to take precautions in form of finding ways to circumvent the detection mechanisms. For that purpose, we have to analyze the detection algorithm. One of these algorithms uses data flow analysis to detect statements that produce results that are never used afterwards. If we simply fake the use of the results of the inserted code, it cannot be detected by this method any more.

As we saw, there are several mechanisms that can be used for messing up code, and it is far better to use some of them in parallel than just using a single mechanism since it is easier for a human to concentrate on one issue than analyzing a complex code structure.

To give an impression of how code could look like, here’s a messed-up version of the purchasing agent example:

```
1  int v1[] = new int[11];
2  Object v2[] = new Object[10];
3  float f1 = 22002.0f;
4  Agent2 v3 = this;
5  Agent2 v4 = null;
6  Agent2 v5[] = null;
7  v1[0] = 3;    v1[1] = 1;    v1[2] = 5;    v1[3] = 7;    v1[4] = 2;
8  v1[5] = 6;    v1[6] = 4;    v1[7] = 8;    v1[8] = 9;    v1[9] = 10;
9  v1[10] = 1;
10 v2[0] = “BuyFlowers”;       v2[1] = null; v2[2] = “xxv”;
11 v2[3] = null;     v2[4] = “10 red roses”;v2[8] = null;  v2[9] = null;
12 public void startAgent() {
13 for (int i=0; i<1; i++) {
14   if ((((String)(s12(6,v5,null, 0.0f))).equals(v2[2]))) {
15     v5 = ((Agent2[])
16        (s12(10,(Agent2)(s12(7,v2[8],v2[3],10.0f)),(String)v2[0],20.0f)));
17     s12(1,(Agent2)(s12(5,v2[3],v2[8],(float)v1[1])),null,20.0f);
18   }
19   if (((Boolean)(s12(9,new Float(((Agent2)
20               (s12(5,null,null,(float)v1[10])))).askprice((String)v2[4])),
21               new Float((f1 - ((int)f1 / 1000))),0.0f))).booleanValue()) {
22     f1 = (float)((int)f1 / 1000) +
23       ((Agent2)(s12(5,null,null,(float)v1[10]))).askprice((String)v2[4]);
24     v4 = (Agent2)(s12(5,null,v2[1],(float)v1[10]));
25   }
26   if (((Boolean)(s12(8,new Integer(v1[10]),
27             ((Integer)s12(4,v2[8],v2[3],13.0f)),0.0f))).booleanValue()) {
28     s12(2,v4,v2[4],(float)((int)f1/1000));
29   }
30   if (((Boolean)(s12(8,new Integer(v1[10]),
31             ((Integer)s12(4,v2[8],v2[3],0.0f)),35.3f))).booleanValue()) {
32     s12(1,v4,null,0.5f);
33   }
34   else {
35     s12(1,v5[++v1[10]],v2[1],0.0f);
36   }
37 }}
38 public Object s12(int i, Object o, Object p, float f) {
39   if (i == v1[1]) s24((Agent2)o);
40   if (i == v1[4]) s26((Agent2)o,(String)p,f);
41   if (i == v1[0]) return(new Integer(v5.length));
42   if (i == v1[6])
43     return(new Integer(((Integer)s12(3,null,null,0.0f)).intValue() - 1));
44   if (i == v1[2])
45     return(v5[(int)f]);
46   if (i == v1[5])
47     if (o == null) return(“xxv”);
48   if (i == v1[3])
49     return(s27());
50   if (i == v1[7])
51     return(new Boolean(((Integer)o).intValue() >=
52                                              ((Integer)p).intValue()));
53   if (i == v1[8])
54     return(new Boolean(((Float)o).floatValue() <
55                                              ((Float)p).floatValue()));
56   if (i == v1[9])
57     return(((Agent2)o).s28((String)p));
58   return(null); }
```

Although it does the same as before, it is far less understandable. It could have been produced automatically (for this paper, it was done manually) and does not run much slower than the original (in fact, a corresponding Java program is a factor 1.5 slower and twice as long). The used code mess-up mechanisms (variable recomposition, conversion of compile-time control flow elements into run-time data dependent jumps, and partial replacement of procedure calls by code insertion) have been described above.

We now have some mechanisms that allow us to mess-up code, but there is a requirement that we did not have taken into account yet: the question of whether the code variants build up a space that is large enough. If this is not the case, an attacker could compute some variants in advance and then simply compare the computed code variant and the code of the agent. We can state that all mechanisms meet this requirement, that offer variants that can be produced by using a numeral argument. If we take e.g. variable recomposition, the ways we can rearrange the variables are not limited by a number that is small enough to allow to build up a “dictionary of versions” in advance and we could design this mechanism in a way where the rearrangement is described by an “index” number.

Concluding the question of how the code can be converted into a less readable form, we can state that there seems to exist promising mechanisms that can be derived from research in the field of software engineering, code optimization, and reverse engineering.

8 Limited lifetime of code and data

Apart from code mess-up, another big problem seems to be the last requirement, the question of whether the now time-dependent execution model restricts the kind of applications that may use the mobile agent system.

Since we have to take into consideration the fact that an attacker can “crack” an agent if it has enough time, we have to find a way to compensate this possibility. One way to do this is to make relevant parts of an agent invalid after a specified interval. Therefore, expiration dates have to be appended to at least some elements of the agent. Since we have to do this in an unfakeable way, we can use digital signatures (an existing cryptographic technique), that sign the combination of data element and expiration date. Unfortunately, this means, that only a trusted authority, normally the owner of the agent, can issue information that bears an expiration date, and there is no way for an agent during its life to get new expirable information elements but from such a trusted party. Fortunately, only some data carried by the agent have to bear expiration dates. The characteristics of the elements of this class of data is, that they are self-contained documents, that can be exchanged for other goods or services or that prove identity (as the identity also can be used for getting goods, services or authorisation). Elements, that have these characteristics, can be called “tokens”, and, in real world, they rarely bear expiration dates. Typical tokens of the real world are coins, identity cards or permits.

A token in a mobile agent system should consist of the

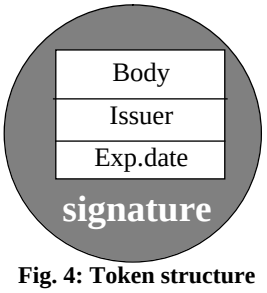


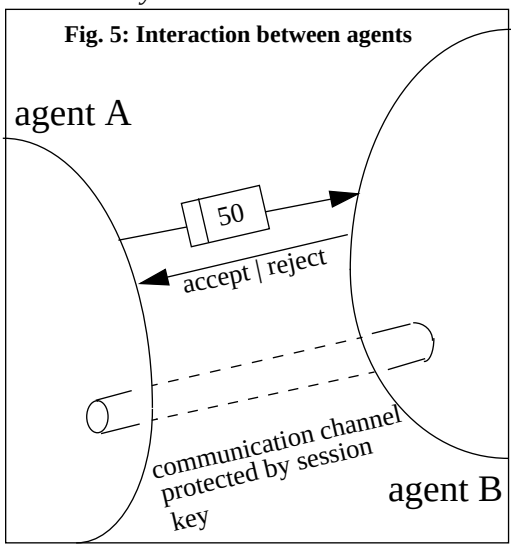
Fig. 4: Token structure

original data (the body), e.g. the electronic money bit-string, the identification of the issuer and the expiration date. The whole element is signed by a signature mechanism like DSS or PGP. These mechanisms use secret keys which belong to the signee, therefore they have to know which party pretends to have issued that document.

Security sensitive informations are e.g.:

Electronic Money

Fig. 5: Interaction between agents



Here the expiration date means, that noone will accept the money after this date. This results in the need of the party which accepts valid money to bring it to a publically trusted party, e.g. a bank, before it expires, which can be problematic when failures, e.g. a network partition, occur.

Cryptographic Keys

As also keys expire after some time, they are more of the session key type (which are often valid only for a certain interval), and are known to a second party, i.e. the partner to which the protected communication will take place. It is interesting to note that public keys are not tokens, it is enough to have the guarantee that the host may not modify them during the protection interval. If an agent wants to communicate with another agent over a protected communication channel, it has to get its public key (e.g. by bearing it). Then, a session key can be generated dynamically and sent to the part-

ner crypted with a public key scheme like RSA. The encryption of the communication then can take place using the session key, which in return has to bear a expiration date.

The agent as a whole
As it is the main target of attacks by a malicious host, especially the agent itself has to bear an expiration date. No host or other party should accept arriving agents that are expired or the interaction with expired agents as they may be "taken over" by a malicious host.

Information elements, that are not tokens, need not be protected by expiration dates. It is enough to have the affirmation, that these elements are not known to the host unexpectedly for the duration of the expiration interval of the agent. In our example, almost all elements except the electronic money are not tokens.

The difference of tokens to e.g. coins is the duration of the validity as coins are valid for a reasonable amount of time. This can consitute a problem, as it is possible by failures or by attacks of the host for the tokens to become invalid. Although not critical for identity tokens, the value of a token is severely affected by this. Therefore, token interaction has to be handled as transactions that can not be committed before the tokens are made persistent in such a way, that an attacker cannot destroy them.

Like tokens, agents can also become invalid due to failures or attacks. Although this causes no direct loss of value (except the problem of tokens described above), it is a typical denial-of-service problem, that have to be handled somehow. Fortunately, a agent system has to offer mechanisms that handle the loss of agents due to failures, e.g. network failures. Therefore, it can be argued that the same mechanisms can be used for handling agent expiration problems. The only modification that have to be made is to also consider a malicious host as a potential cause for the loss of agents, like the failure of a network connection.

9 Further attacks

If there is an agent protection scheme like the one described in this paper, one can imagine attacks that rely on the characteristics of this scheme. One attack is *sabotage*, or the trial to destroy parts of the agent without being detected. As an agent contains data that might change during execution, the attacker can simply modify single bits of the data area without knowing about the effects for the agent. Fortunately, this attack is very close to the problem of data, that is sent over an insecure network. Therefore, similar error detection or even correction mechanisms like CRC can be used as long as the attacker cannot detect the concrete structure of the mechanism (as it is easy to circumvent a CRC algorithm if one knows the exact mechanism and if it can see the borders of the protected data elements).

Another attack is the *black box test*. Its aim is to determine characteristics of the inside of the "black box" by executing the box with different input parameters and by watching the effects. The effects can be formal results like output values or characteristic "activity patterns". In our example, the attacker could execute the agent until it tries to buy the flowers, starting every time with the initial agent. The only value that is changed over the trials is the price for the flowers. After the agent finally wants to buy, the attacker knows the price that is both the lowest so far and that is below the maximum price. Even if the agent would not buy the flowers immediately (as it tries to ask at least three different providers), the attacker can watch, whether the data of the agent has changed. If this is the case, it is very likely, that this agent has memorized a better price. If it comes to countermeasures, two goals have to be reached: first, the parallel execution of the same agent has to be suppressed (e.g. by using a trusted third party that is informed by the agent about its execution), second, the very fast execution of an agent has to be suppressed (e.g. by using a similiar interaction with a trusted host). Finally, activity patterns can be covered up by using dummy code.

The list of possible attacks cannot be complete as they rely only on the imagination of the attacker and the details of the implementation of the mechanisms, so this aspect is another subject of future research.

10 Conclusions and Future Work

With the employment of Code Mess-Up techniques, we have devloped a non-cryptographic agent protection scheme that is build up like any cryptographic mechanism: we transformed readable input (i.e. code and data) to an unreadable form by a mechanism, that cannot be inverted easily with the current knowledge. If it can be inverted easily by a new algorithm in the future, it would solve some major problems of other fields like software engineering. In contrary to cryptographic schemes, we did not assume our mechanism to protect data a very large amount of time like months or years, but only a comparable short time like minutes or hours, and we therefore have to handle the effects of this aspect.

As the protection is based on the unreadability of code, the protection is the harder the more unanalyzable the used agent language is. Therefore, code mess up is used best together with a machine-code like language like Java Bytecode or low Telescript, which also offer a better performance than their interpreted high-level equivalents.

Code mess up does cost something, both in terms of speed and of space, and the processing model is more complex due to expiration aspects. Therefore, this scheme should be mainly used for agents that need to be protected, e.g. because they carry money or other sensitive data. The global usage of this mechanism even for non-sensitive applications may be too expensive, but because code mess up infrastructrure is needed only for

protected agents, agents of both protection levels can exist and interact in parallel.

Since code mess up is also applicable to mobile code mechanisms like the Java applet model, it can also protect these mobile code units (e.g. applets) from attacks of their hosts, i.e. browsers and users.

As we have seen, it is practically possible to protect agents from malicious hosts by using code mess-up techniques. Future work has to prove this claim.

Further steps comprise the implementation of some code mess up mechanisms and the estimation of their protection against human, tool-aided and automatic attacks. Therefore, tools and other programs have to be written that try to “decrypt” messed up code.

Apart from the examination of the protection strength of the code mess up mechanisms, applications that use mobile agent technology have to be examined in order to find the practical problems that expiration date processing poses on them. In order to do that, the infrastructure components that allow the system to use expiration dates have to be implemented, e.g. modified host software that rejects expired agents, trusted node software that is able to mess up normal agents or “refresh” them after the expiration interval.

Finally, the costs of this protection scheme have to be compared to protection schemes that do not use code mess up.

Once implemented, code mess up offers a mechanism, that allows Mobile Agents to migrate not only to trusted nodes, but , in between times, to any host without having to fear successful attacks by malicious hosts.

Literature

[ASU86] Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D.: Compilers: Principles, Techniques and Tools, Addison-Wesley, 1986

[CGH95] Chess, David; Grosof, Benjamin; Harrison, Colin: Itinerant Agents for Mobile Computing, IBM Research Report RC 20010 (03/27/95), IBM Research Division, 1995

[Che97] Cheswick, William R.: Internet Security and Firewalls : Repelling the Wily Hacker, Addison-Wesley, 1997

[FGS96a] Farmer, William; Guttmann, Joshua; Swarup, Vipin: Security for Mobile Agents: Issues and Requirements, in: Proceedings of the National Information Systems Security Conference (NISSC 96), 1996

[FGS96b] Farmer, William; Guttmann, Joshua; Swarup, Vipin: Security for Mobile Agents: Authentication and State Appraisal, in: Proceedings of the European Symposium on Research in Computer Security (ESORICS), 1996

[Jer94] Jerney, John: AT&T PersonaLink Delivers Some Magic, in: Pen-Based Computing, November 1994
http://www.volksware.com/pbc/article/pers4-9.htm

[GM96] General Magic: The Telescript Reference Manual, 1996
http://www.genmagic.com/Telescript/Documentation/TRM/

[Gra96] Gray, Robert: Agent Tcl: A flexible and secure mobile-agent system, in: Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96), Monterey, California, July 1996

[HCK95] Harrison, Colin; Chess, David; Kershenbaum, Aaron: Mobile Agents: Are they a good idea?, Research Report, IBM T.J. Watson Research Center, 1995

[IBM95] IBM Corp.: Things that Go Bump in the Net, 1995
http://www.research.ibm.com/massive/bump.html

[KP78] Kernighan, Brian W.; Plauger, P.J.: The Elements of Programming Style, McGraw-Hill, 1978

[Ord96] Ordille, Joann J.: When agents roam, who can you trust?, in: Proc. of the First Conference on Emerging Technologies and Applications in Communications, Portland, May 1996

[Pal94] Palmer, E: An Introduction to Citadel - a secure crypto coprocessor for workstations, in: Proceedings of the IFIP SEC’94 Conference, 1994

[Rug96] Rugaber, Spencer: Program understanding. Encyclopedia of Computer Science and Technology, 1996. To Appear.

[Sch96] Schneier, Bruce: Applied Cryptography, John Wiley &Sons, 1996

[SL95] Sandholm, T.; Lesser, V.: Issues in Automated Negotiation and Electronic Commerce: Extending the Contract Net Framework, in: Proceedings of the First International Conference on Multiagent Systems (ICMAS-95), 1995

[SPC89] The Software Productivity Consortium: Ada Quality and Style, Van Nostrand Reinhold, 1989

[Sta95] Stallings, William: Network and Internetwork Security, IEEE Press, 1995

[TV96] Tardo, Joseph; Valente, Luis: Mobile Agent Security and Telescript, in: Proceedings of IEEE COMPCON’96, 1996

[Vit96] Vitek, Jan: Secure Object Spaces, in: Proceedings of the 2nd International Workshop on Mobile Object Systems, dpunkt, 1996