The Harpoon Security System for Helper Programs on a Pocket Companion

Gerard J.M. Smit Paul J.M. Havinga Daniël van Os

University of Twente, deptartment of Computer Science
P.O. Box 217, 7500 AE Enschede, the Netherlands
{smit, havinga, os}@cs.utwente.nl

Abstract

In this paper we present a security framework for executing foreign programs, called helpers, on a Pocket Companion: a wireless hand-held computer. A helper program as proposed is this paper, is a service program that can migrate once from a server to a Pocket Companion or vice-versa. A helper program is convenient, provides environment awareness and allows asynchronous interaction. Moreover, helpers can be used to save processing power and to reduce communication. By migrating to the location of a resource, a helper can access the resource more efficiently. This is particularly attractive for mobile computing, where the network conditions can be poor and unreliable, and because it does not require a permanent connectivity.

Security is a significant concern for helpers, as the user of a Pocket Companion receiving a piece of code for execution may require strong assurances about the helper's behaviour. The best way to achieve a high security is to use a combination of several methods.

We are designing a prototype of a helper system, called Harpoon, on top of the Inferno operating system.

1 Introduction

This paper is written as part of the Moby Dick project[1] [15]. In this project we develop and define the architecture of a new generation of hand-held computers, so-called *Pocket Companion*. It is a small portable computer and wireless communications device that can replace cash, cheque book, passport, keys, diary, phone, pager, maps and possibly briefcases as well. These devices are resource-poor, i.e. small amount of memory, limited battery life, low processing power, and they are connected with the environment via a network with variable connectivity.

The aim of the Moby Dick project is to exploit the availability of these Pocket Companions by including them in larger systems. The Pocket Companion is more than just a small machine to be used by one person at a time like the traditional organizers. If users incorporate them into a global distributed system, they must be confident that the system is trustworthy. In order to use the full potential of these personal machines, the user must be in full control over its machine, its information flow, and who can access it.

The design challenges of the Moby Dick project lie primarily in the creation of a *single architecture* that allows the integration of security functions, externally offered services, personality, and communication.

Security will play an important role in the design of such an 'open' architecture. A user will not allow any foreign service on his very personal machine unless security is handled well enough. Vice versa, his machine should not provide services to guest machines either, unless security is guaranteed. The integration of a security module with the Pocket Companion and the wireless network can for example provide the basis for a secure and seamless integration of payment services.

Helper programs are service programs that a Pocket Companion can receive and execute locally. Additionally, it can also be a service program that is executed on a *remote* machine on behalf of the local machine. These helper programs are security critical, as they run on a machine on which you might have stored valuable and sensitive data. Furthermore, helper programs might use the machine in a way that you do not like, but where you are not aware of immediately. It might for example drain your batteries, or communicate via expensive communication lines.

Our aim is to create a secure environment on a Pocket Companion to contain untrusted helper programs. We are currently building the Harpoon helper system, that allows untrusted programs to run in the Lucent Technology's *Inferno* operating system [9].

The main goals of the Harpoon system are security, flexibility and user friendliness.

Outline

In this paper we will first describe the particular advantages of using so called helpers in an environment envisioned for Moby Dick (section 2). Then we will investigate

1. The Moby Dick project is a joint European project (Esprit Long Term Research 20422).

in particular the *security* problems of downloading a foreign service on a Pocket Companion and give a survey of possible security mechanisms (section 3). In section 4 we will provide a security model in which several methods of detecting and preventing the helpers of doing harmful things are combined. Finally in section 5 we sketch the architecture and design of a prototype implementation in a modern operating system.

2 Moby Dick Helper programs

In this section we describe the concept of helper programs and why we think this concept is useful.

2.1 Concept

The concept of a helper program is related to so called *mobile agents*. In general a mobile agent is an autonomous program that can migrate under its own control from machine to machine in a heterogeneous network [6, 7]. In other words, an agent can suspend its execution at any point, transport its code and state to another machine, and resume execution on the new machine. By migrating to the location of a resource, the agent can access the resources more efficiently even when the network conditions are poor or the resource has a low-level interface. This efficiency, combined with the fact that an agent does not require a permanent connection with its home site, makes an agent particularly attractive for mobile computing since roving devices often have low-bandwidth and unreliable connection to the network.

In our concept we have a restricted view on mobile agents. A helper program, as mentioned in this paper, is *static*: the program can migrate only once. We have made this restriction for simplicity and security reasons. A helper is a service program that a Pocket Companion can receive and executes *locally*. Additionally, it can also be a service program that is executed on a *remote* machine on behalf of the local machine. When a helper migrates from a service provider to a Pocket Companion it can provide services relevant to its environment.

The main emphasis in this paper is on downloaded helpers from a service provider (also called server) to the Pocket Companion. Security is a significant concern, as a client receiving a piece of code for execution may not know anything about the helper's intentions and behaviour. In chapter 3 we will deal with this in more detail.

Helper programs have some similarities with the helper applications (like ghostview and MPEG players) that many WWW net browsers use to process data from the network. However, these helpers are already part of the system of the user, they do not origin from a remote server. Although less urgent, in such an environment it is also desirable to create a secure environment [5].

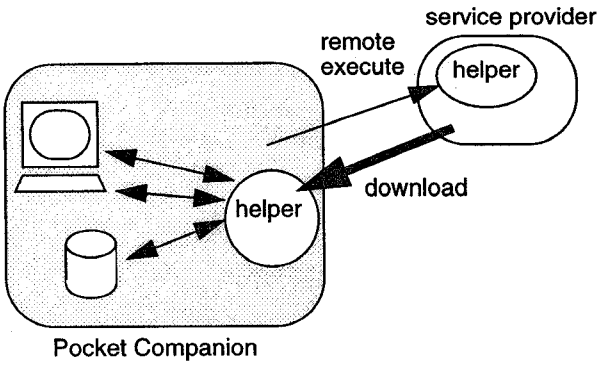

Figure 1: Helper migration with a Pocket Companion

2.2 Why using helpers on a Pocket Companion?

Executing helper programs on your Pocket Companion implies that you allow unknown foreign programs to run on your personal machine on which you might have stored valuable and sensitive data. Why would someone even want to interact with a piece of code for which nothing can be absolutely guaranteed? The answer is, obviously, that the person hopes that the program will be useful to him.

First of all it is *convenient* to receive helpers on a Pocket Companion where and when you need them. Memory of a Pocket Companion will not likely be sufficient to hold all programs that you might need. Helpers that offer these services in a particular environment can be loaded and activated dynamically.

Helpers can be used to save processing power or reduce communication, and hence *save battery power*. A helper has the ability to perform information retrieval and filtering, while returning to the client only the relevant information, thus saving communication bandwidth. For example, if a Pocket Companion runs a helper with an extensive user interface, no communication resources are used for the user-interaction because the user-interface is executed locally on the Pocket Companion. If a Pocket Companion has a relatively high computational intensive task, it can send a helper to a remote server to perform this task on his behalf, thus saving processing power for the Pocket Companion.

The machine can run *environment aware software* like a guided tour program in a museum and a ticket agent in a railway station. The user can even download personalized helpers, e.g. a Dutch version of the guided tours service in an Italian museum. Therefore it will be vital to a Pocket Companion to be able to find out what services are available in a particular place.

The Pocket Companion might only be connected to a network intermittently, hence it has only intermittent access to the server. A downloaded helper may continue

with its task, even when it is *disconnected*. Furthermore, because the programs we foresee often have a considerable user interface, a fast response time is achievable.

Because for security reasons a visiting helper program is constantly checked, the code of an unknown user can be run relatively secure on your Pocket Companion. Quite remarkably, these security measures are normally not taken during the installation of a new program!

3 Security aspects and solutions

Security and safety plays a dominant role in the design of a system that accepts and uses helper programs. In this section we review *security* mechanisms that provide protection and integrity in the presence of malicious programs. *Safety* features mainly promote robustness and prevent accidents. Usually, an operating system limits the damage that an unsafe program can do, but safety alone is not sufficient.

A main security aspect is the amount of privileges the helper program can obtain when entering the client. The problem is to find a useful compromise between the desire to isolate the helper's execution environment from the system, and the need to provide sufficient privileges in order to accomplish the helper's task.

Security can be divided into issues related to network communication and issues related to the execution of a helper. The network security deals with mutual authentication, integrity of the communication and confidentiality. Crytography can be used to provide secure communication. We mainly focus on security issues related to the execution of a helper on a Pocket Companion.

Basically the security involved in executing a helper can be handled in two ways: by *preventing* the helpers doing harmful things, or by *detecting* that it does, or intents to do harmful or unpleasant things.

3.1 Survey of security mechanisms

Today most operating systems support some form of protection of system resources. However, the protection mechanism of these operating systems are typically based on the amount of privileges of a *user* on whose behalf the program runs, and not on the amount of privileges of the *program*. Furthermore, some current agent security approaches rely on a trusted external authority who is responsible for the correct behaviour of the agent. This though implies that agents of unknown principals are rejected, even if they can do useful work for you.

Another approach is to prohibit a program that is not trusted from accessing any security relevant resources. On the first sight, this seems reasonable. However, the more functionality a program can have, the more resources it needs to access. This means that the user has to make resources explicitly available to a program, and needs to make a detailed classification and security assessment of his resources.

Prevention of malicious behaviour of programs has been a research area for long. A number of mechanisms have been proposed, we will only mention some of the main mechanisms.

1 *Trusted compiler or interpreter*

The compiler or interpreter only allows access to safe resources and a restricted set of library functions. The language's type system should be safe - preventing forged pointers and checking array bounds. In addition to that, the system should garbage-collect memory to prevent memory leakage, and carefully manage system calls that can access the environment outside the program, as well as allow programs to effect each other. Examples are: Java [10], Safe-Tcl [1], Agent-Tcl [11], Telescript [18], and Phantom [2]. Recent research, for instance concerning Java [4], has exposed several security flaws.

2 *Access Control List*

An Access Control List (ACL) is a central data structure in the operating system that specifies which users have what access rights to which resources. The list can be inspected before the program is allowed to execute. This gives the user the ability to refuse the program on beforehand. Once the program is accepted and running it needs to be checked whether it behaves according the access control list. This can be accomplished in software at the system call to the OS, or - for some resources - in hardware for instance with the help of an MMU.

ACLs are inflexible and the circumstances in which a request may or may not be granted can become unclear. It is therefore essential that the policies as described in an ACL are clear and unambiguous, so that security loopholes and contradictions can be discovered. The user-interface might give a problem because classifying resources according to how private they are is hard and the secrecy of a piece of information also depends on by whom it is accessed [14].

3 *Capability list*

A capability list is data structure of an object that defines which resources it is allowed to use, i.e. its capability [16]. So, capabilities are related to objects that can be a user, but also a program. An object is allowed access to resources only when it has the right capabilities. The main advantage is that capabilities are easy to maintain.

4 *Restricted view of the name space*

In this method only a restricted set of resources can be used by the program by giving it a restricted view of the possible resources. For example, in Plan 9 [19] and

Inferno [9] this is accomplished by giving an application a mount point in a tree based file system. It can only access resources below the mount point. Resources that are available to applications all appear exclusively in the name space of the application. This implies to data, to communication resources, and to the executable modules that constitute the applications. Security-sensitive resources of the system are accessible only by calling the particular devices that provide them.

5 *Restricted environment*

Another approach is to run an untrusted program in a 'safe' or restricted environment. This is used, for example, by the Safe-Tcl interpreter, which has removed the 'dangerous' primitives of Tcl [1] and is also used by the HotJava class loader [10].

6 *Signature of the code*

Programs have some principal[1] that can be held responsible for its behaviour. Digital signatures can be used to *authenticate the principal* and to *guarantee that the program has not been tampered with.* The program's maliciousness, whether deliberate or due to bugs, cannot be decided by any level of cryptography. This item has some relation with network security in which the sender of a program is authenticated. Note that the sender does not have to be the principal of the program.

None of these methods prevent the helper from doing nasty things like abundant CPU-usage or unrestricted memory usage. This misuse may also give opportunities for criminals to break the system. Therefore this potential misuse has to be detected as well.

Detection methods

The objective of detection is to find deviations from an expected behaviour pattern. The detection measurements can for example be based on: time (e.g. time of day or amount of time used), numbers (e.g. number of files or bytes read), or intensity (e.g. detect bursts of behaviour). An untrusted program is not denied access to resources unless it misbehaves in some way. As long as the user has the possibility to supervise what is happening he might be prepared to take the chance and let it access his computer.

Detection methods draw a lot from the work done in *intrusion detection* in computer systems [13]. It is used to detect anomalies in user behaviour or misuse of a computer. Anomalies behaviour is behaviour that deviates from 'normal use' pattern, e.g. accessing a certain number of files per minute. Misuse means that the weaknesses or flaws in the system are deliberately used to get unauthorised access to system resources.

Crosbie and Spafford have used autonomous agents to detect maliciously behaving programs [3].

Instead of making up rules for what programs are allowed to do and how they are allowed to interact with the rest of the system, the user can be given information about what the program is doing [17]. This however requires that the users knows enough of the system to judge whether the program misbehaves or not. To reduce the amount of knowledge a users needs to have, a program can be categorized according to the kind of the application. The program will be granted access to only those resources a typical application usually needs to accomplish its tasks. The user will be notified when the program deviates from its expected behaviour and can decide whether what the program does is illegal.

3.2 Why another method?

The mechanisms described each try to solve specific security problem in a particular way. However, we believe that individual solutions are not sufficient to solve all security risks. Moreover, there may exist security holes, unknown today, that are not covered by these solutions. Therefore we propose a structured framework in which several complementary methods - using both prevention and detection - are integrated, with which we hope to obtain sufficient security. We expect that intrusions due to holes in one security method will be detected by other methods.

What we want is a security policy that facilitates the execution of new unknown helpers without having to base the trust on an external authority, and without a security structure in which the user has to do a manual classification.

4 Harpoon security model

4.1 Design goals

The helper security system used in Moby Dick is called 'Harpoon'. This system allows foreign helpers to run on a Pocket Companion in a secure way. The proposed framework not only deals with security sec, but also with security related items such as *QoS management* and protection against abundant resource usage like processor cycles, power and memory.

The basic design goal is *security.* An untrusted helper program should not be able to access any part of the system for which it has no permission. Unauthorized behaviour and unauthorized access to system resources will be prevented and trespassing will be detected.

Another goal is *flexibility.* The system should be able to allow or deny accesses to system resources flexible, i.e. to

1. A principal can be a computer system, a programmer, or an organisation. This does not include a compiler since a compiler itself cannot be held responsible.

allow the helper only to write in some particular files, or to allow only network access to a particular network address. Flexibility also implies that helpers that do not fit into the Harpoon security model still have means to execute: the user should have the possibility to run the helper program in a restricted environment (a default minimal environment or one chosen by the user).

And last but not least, in our view helpers always run under the *control of the user*, i.e. the user on which the machine the helper runs, may refuse a helper or kill a helper. The user only trusts its own Pocket Companion, i.e. the hardware and embedded software, but not the software, nor the downloaded helper of the server. An important aspect is the user-interface of the security module. It should *help* the user rather than *annoy* him. The system should be configurable by the user. This configurabilty is useful since different users have different requirements as to which files and resources the helper should have access.

4.2 Structure of Pocket Companion helpers

The basis of the framework is a profile that specifies the resources the helper needs or expects to use. The framework encloses a number of detection and prevention concepts, for example: a capability list, a restricted view on namespace and QoS management. The code of the helper and the profile can be authenticated with a signature.

The helper programs have to follow certain rules. We assume that all helpers have a *principal* that can be held responsible for its behaviour. In our view helpers consist of: a piece of code, a profile and the signature of the principal (see figure 2).

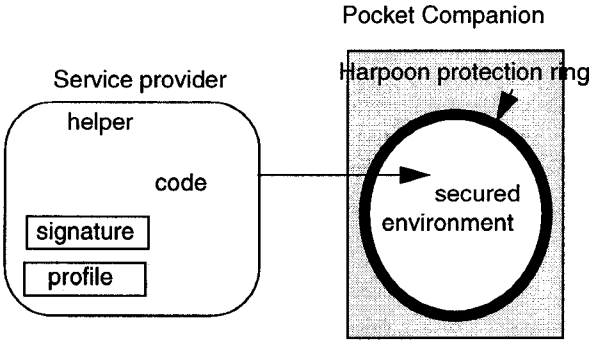

Figure 2: Helper migration to a Pocket Companion

- *The signature*

  The signature is used to authenticate the helper's principal and to guarantee that the helper's code and profile have not been tampered with. The sender's authentication and integrity of the received data is dealt with in the network security.

- *The profile*

  The profile defines the resources the helper needs or expects to use; and can also include the capabilities to access these resources (a capability list).

  The profile is used for two purposes. First the Pocket Companion will use it - in interaction with the user - to decide whether the helper is allowed to use the requested resources of the machine. During execution of the helper, the profile is used to detect malicious behaviour. Although at first sight, the profile seems to have similar properties as Quality of Service (QoS) parameters, the profile extends this view. The profile is not only related to the 'common' resources such as disk, network, display, but also power consumption, CPU usage, etc. A profile might even include the access rights of a particular file, e.g. reading a password file[1]. A *capability* is a certified statement from a principal that a service can use as credentials when accessing client's resources. The capability consists of a list of access rights it has and a signature. This is normally a signature from the user of a Pocket Companion but may also be from some principal that the owner trusts.

  When a helper is loaded the profile will be presented to the Pocket Companion. Once accepted the helper gains access to the resources it needs, albeit not exceeding the granted QoS.

- *The helper's code*

  This code can be either a machine language executable or an interpreted language. A trusted compiler can be useful to provide additional security.

4.3 Security procedure

In first instance, the user decides whether to accept or refuse a helper. The credibility of a helper depends on its profile and the identity (trustworthiness) of the origin of the helper or principal. Before a helper is activated on a Pocket Companion the following actions have to be taken:

1 *Signature verification*

The user checks the signature to authenticate the helper and the origin of the helper; and to detect whether the helper has been modified. The profile is authenticated also and needs to be verified as well. When needed, the helper can be secured with cryptography, for instance to allow only specific clients to execute the helper.

2 *Profile acceptance*

The profile can be inspected and might be compared with a list of privileges of known helpers, servers and

1. Access rights to a file can also be gained with delegation certificates, which will authorise access to files [8]. The certificates can be part of the profile.

principals. This gives the user the ability to accept or refuse the helper. Helpers that are not known to the system can be allowed only restricted access according to the kind of helper and its principal. When the profile includes capabilities the Pocket Companion only needs to check whether these are valid.

Once the user has accepted the helper and it is running, the system needs to check whether it behaves according the profile, the *profile verification*. System resources (such as files and devices) that a helper may not access, are shielded. Furthermore, to detect malicious behaviour the system will support a number of detection mechanisms. It detects excessive resource consumption of a helper beyond the accepted profile (e.g. power, bandwidth, memory or CPU usage).

4.4 Profile categories

A main problem of the security policy described above will be the user interface. The security user interface is critical for helping the average user choose and live with a security policy. The user has to do a classification and security assessments of his resources. This can be a annoying and rather complex matter, and so it is hard make the system secure. Users might disable security if they are burdened with repeated authorization requests from the same helper program. Also, annoyed users may stop reading the dialogues and repeatedly click *Okay*, defeating the utility of dialogues.

If the user can classify programs the *kind* of program, rather than by which resources they should be granted access, the demands on what the users need to know about resource consumption etc. could decrease. We have defined a limited set of *profile categories* such as: super use, payment application, normal use, file read only use, display only, etc. The profile category determines the access rights of the system resources that typical applications usually need to accomplish their tasks. A program is granted access to only those system resources that are specified in the profile category.

5 Harpoon prototype in Inferno

Some of the requirements to implement a secure environment for helper programs as mentioned previously, are already implemented in existing (prototypes of) operating systems like Inferno [9], Unix and Amoeba [16]. None of these can fulfil all of our requirements, or are not suitable for a Pocket Companion due to for instance memory usage. It is however possible to prototype a helper system on top of an existing operating system. In this project we use the Lucent Technology's *Inferno* operating system as a prototyping environment. Inferno offers some security mechanisms against erroneous or malicious applications.

5.1 Inferno

Inferno is an operating system for delivering interactive media to its users. It is under development within the Computing Sciences Research Center of Bell Labs at Lucent Technologies. It is intended to be used in a variety of network environments, for example TV set-top boxes, handheld devices like the Pocket Companion, but also in conjunction with traditional computing devices. Inferno's strenght lies in its portability and versatility. It currently runs on Intel, MIPS, Motorola 68K, and AMD 29K architectures. It runs usefull applications standalone on machines with as little as 1 MByte of memory. Inferno also provides communications security and key management. Applications and system may be split easily - and even dynamically - between client and server.

There are three main design principles in Inferno. First, all resources are named and accessed as files in a forest of hierarchical file systems. Devices are represented as files, and device drivers (such as a network interface) attached to a particular hardware box present themselves as small directories. System services also live behind file names.

Second, disjoint resource hierarchies provided by different services are joined together into a single private hierarchical namespace. Resources available to applications all appear exclusively in the name space of the application. This applies to data, to communication resources, and to the executable modules that constitute the applications.

Finally a uniform communication protocol, called Styx, is used to access these resources, whether local or remote. The Styx communication protocol is used for both local and remote file operations. To access a file, or to navigate through the file system a client sends a request to a server and then receives replies in a manner similar to Unix remote procedure calls.

Inferno applications are typically written in a new language called *Limbo* [12]. Limbo is carefully type-checked at compile- and run-time; for example, pointers, besides being more restricted than in C, are checked before being referenced, and the type-consistency of a dynamically loaded module is checked when it is loaded. All Limbo data and program objects are subject to a garbage collector, built into the Limbo run-time system.

5.2 Harpoon security system

The principles of Inferno are used to create the Harpoon security system. Harpoon has three main tasks: assisting a user in making the initial decision whether to run the application or not, setting up a safe environment and finally monitoring the behaviour of a selected application.

As said before the helper program has a *signature* and a *profile* that Harpoon can use to determine the capabilities of the program. The signature and the profile are stored in a

separate file. When a user selects a helper program that it wants to run, Harpoon will assist the user to choose the appropriate security level. After checking the signature of the application and of the profile it will display the principal of the program and the profile. Subsequently Harpoon will analyse the profile, and inform the user about how harmful it thinks the application could be. In the first prototype we provide a few customizable standard profiles. The user can now decide to grant a permission to execute the application according to the profile in the selected restricted environment.

The next security provision is that Harpoon assigns a private namespace to a helper program. This namespace is used to restrict the program's view of the file system. Figure 3 gives an example of the namespace of a helper that is constructed of the console device, and subdirectory `noot`.

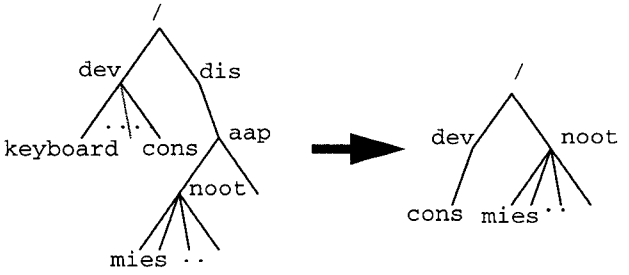

Figure 3: Helper namespace assignment

When the helper program is running, Harpoon will monitor the behaviour and checks for un-allowed actions. Since all resources are represented as files and are accessed using the Styx protocol, all specific accesses (in a finer grained sense than its address space) can be controlled by a single mechanism. By monitoring the Styx messages to the Inferno file system, the access of resources contained in the namespace can be controlled. By interpreting the Styx messages it is not only possible to restrict access to specific resources, but also to perform some form of accounting (i.e. quotas).

User interface

Helper programs that enter a machine are visualized with an icon in a special Harpoon window. The colour of the icon represents the profile category, e.g. from white (for example display only), to black (system super use). The colour is only an *indication* of the possible danger of the program. Which colour is assigned to a group of profiles, and the number of colours is flexible: the user decides the number of colours it wants to use, and how profiles are categorized. This configurabilty is useful since different users have different requirements as to which files and resources the helper should have access.

When a user clicks on the icon the profile as well as the identity of the principal will be displayed. The user can change the colour of the helper and overrule the profile. During execution the user can inspect the resource usage of a helper in a graphical way.

Prototype implementation

To realize the security provisions, a server is required between a client and the actual file server (see figure 4). In the target secure environment it should be impossible for an application to bypass the Harpoon system by talking directly to the real file server.

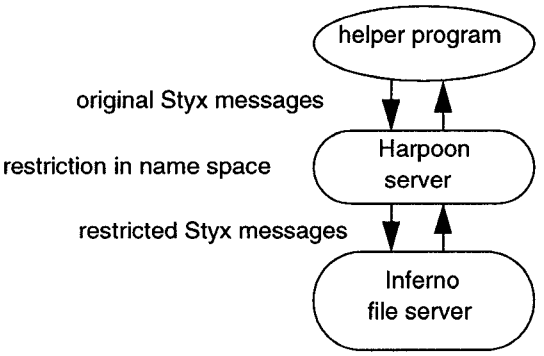

Figure 4: Harpoon server environment

The Harpoon server relays messages from a helper program to the real Inferno server and vice versa while interpreting the Styx messages. Depending on the profile, Harpoon chooses which messages should be relayed to the Inferno file server and which messages should return an appropriate error message The first Harpoon server prototype is implemented at user level, thus not requiring any kernel adaptions.

5.3 Further study

The ideas presented in the previous sections are currently being prototyped on top of the Inferno operating system. When the prototype is finished, tests have to be done on how secure these mechanisms work, how user friendly the approach is and what the performance penalty is.

Research has to be done to distinguish relevant categories that defines a limited set of *profile categories* such as: super use, payment application, normal use, file read only use, display only, etc.

Furthermore, as this system only checks accesses to resources and files, we also want to perform checks on processor utilisation and memory usage. Therefore other mechanisms, similar to accounting or currency based resource allocation, could be used. We also investigate the possibilities in adapting the scheduler of Inferno to a more elaborate QoS aware scheduler that can be used to perform

this task.

6 Conclusion

Execution of foreign code on your personal computer is not a new phenomenon, but additional work and experimenting is required to make it secure. We have designed a security framework for executing foreign programs, called helpers, on a Pocket Companion. Helper programs have a profile associated with it, that specifies what files and resources will be accessed, the way they are accessed, and the capabilities of the helper.

This mechanism uses a two phase approach: it checks the profile in order to make the decision whether to run the application or not, and after that it monitors the behaviour of an application.

The Inferno operating system is used as prototyping vehicle, which turns out to be a proper choice not only because it already offers some security mechanisms against erroneous or malicious applications, but also because it allows us to prototype the required security mechanisms at user level.

7 References and related literature

[1] Borenstein N.S.: "E-mail with a Mind of its Own: the Safe-Tcl Language for Enabled Mail" available from ftp://ftp.fv.com/pub/code/other/safe-tcl.tar

[2] Courtney A.: "Phantom: An interpreted language for distributed programming", in Usenix Conference on Object Oriented Technologies, June 1995 (available from http://www.apocalypse.org/pub/u/antony/phantom/phantom.html)

[3] Crosbie M., Spafford E.: "Defending a Computer System using Autonomous Agents", Proceedings 18th National Information Systems Security Conference, Oct. 1995.

[4] Dean D., Felten E.W., Wallach D.S.: "Java Security: from Hotjava to Netscape and beyond", Proceedings 1996 IEEE Symposium on Security and Privacy, Oakland CA, May 6-8, 1996.

[5] Goldberg I., et al.: "A secure environment for untrusted helper applications, confining the Willy Hacker", 1996 USENIX Security Symposium (see also: http://http.cs.berkeley.edu/~daw/janus-usenix96.ps).

[6] Harrison C.G., Chess D.M., Kershenbaum A.: "Mobile agents: Are they a good idea?", IBM research report, 1995

[7] Hartvigsen et al.: "The Virtual Secretary Architecture for Secure Software Agents", PAAM96 - The First International Conference and Exhibition of Intelligent Agents and Multi-Agents, London, April 22-24, 1996 (see also: http://www.pegasus.esprit.ec.org/people/arne/publications/paam96-f.ps)

[8] Helme A., Stabell-Kulø T.: "Off-line Delegation in a File Repository", DIMACS Workshop on Trust Management in Distributed Systems, Rutgers University, October, 1996. (see also: http://www.pegasus.esprit.ec.org/people/arne/publications/dimacs96.ps)

[9] "Inferno reference manual", Lucent Technologies 1997, document id: TM01FR10, see also http://cruel.com/vanni/

[10] The Java Language Specification, Release 1.0 Alpha 3, Sun Microsystems, Mountain View, CA, May 1995. See also http:/www.javasoft.com.

[11] Kotz D., Gray R., Rus T.: "Transportable Agents Support Worldwide Applications", Proceedings 7th ACM SIGOPS European Workshop, Connemara, Ireland, September 9-11 1996.

[12] Limbo information can be found on http://cruel.com/vanni/

[13] Lunt T.F.: "A survey of Intrusion Detection Techniques", Computers and Security, 12(4) June 1993, pp 405-418.

[14] Levitt K.N., Lo R.W.: "MCF: a malicious code filter", Computers and Security, 1995, pp 541-566.

[15] Mullender S.J., Corsini P., Hartvigsen G.: "Moby Dick - The Mobile Digital Companion", LTR 20422, Annex I - Project Programme, December 1995 (see also http://www.cs.utwente.nl/~havinga/pp.html).

[16] Mullender S.J., Tanenbaum A.S.: "The design of an capability based distributed operating system", The computer Journal, Volume 29-4, pp 289-300, 1986.

[17] Rasmusson A., Jansson S,: "Personal Security Assistance for Secure Internet Commerce", submitted New Security Paradigms '96 workshop, Lake Arrowhead, CA, September 16-19, 1996.

[18] White J.E.: "Telescript Language Reference Manual", General Magic, Inc. Sunnyvale, CA, October 1995. See also http://www.genmagic.com/Telescript

[19] Pike R., et al.: "The Use of Name Spaces in Plan 9", Proceedings of the 5th ACM SIGOPS Workshop, Mont Saint-Michel, 1992