SHAD: A Human-Centered Security Architecture for the Plan B Operating System [*]

Enrique Soriano, Francisco J. Ballesteros, and Gorka Guardiola
Laboratorio de Sistemas – Universidad Rey Juan Carlos
Madrid, Spain.
`http://lsub.org`

Abstract

This paper describes SHAD, a novel architecture for security in pervasive computing environments, and a prototype implementation. SHAD is a Peer-to-Peer and human-centered security architecture. It is based in a general purpose personal device that manages the user's security: the UbiTerm. There are several other systems that, at first sight, seem to provide Single Sign-On in ubiquitous environments. We argue that they fail to do so in practice, and that SHAD offers for the first time a real SSO *that works well in ubiquitous environments that require using multiple machines and services simultaneously. SHAD permits users to share their resources in an easy, natural, and intuitive way, even while being disconnected from the rest of the world. The architecture we propose is able to exploit context information, when it is available. It does not require hard administration tasks, and permits users to manage their own resources. We have been using SHAD for one year. This paper describes our prototype implementation, the experience using it, and some measures that confirm that our approach is reasonable in practice.*

1 Introduction

Machines in a pervasive computing environment are supposed to disappear in the background and make up an *unique pervasive computer*, but this illusion vanishes when you have to explicitly authenticate yourself at different machines, in one way or another. What is needed is a mechanism to access all services from anywhere through an unique explicit authentication. In this paper, we present a solution for this problem, which is highly configurable, traceable, and predictable.

Pervasive computing is supposed to make computers vanish in the background. But for this to happen, authentication and security mechanisms for using and sharing resources in the environment must be unobstrusive. There are Sign-On systems [9] that provide authentication while requiring the user intervention just a single time. However, we have found that most SSO systems are indeed *per-machine* SSO, and therefore not well suited for pervasive applications. Sadly, those that work in a distributed setup, and are not strictly per-machine SSO systems, rely on a central server to authenticate the user to all other machines in the environment. This is a serious problem, because such approach requires a central administration, not to talk about the introduction of a single point of failure for all users in the system.

Using magnetic cards or other physical tokens to authenticate users (which we also tried) still remains uncomfortable and raises new issues: card readers (or the device involved) are not available for all kind of machines, users leave their cards at offices, etc.

In the last PerCom we presented Plan B [6], an operating system for ubiquitous computing that offers simple mechanisms to operate with resources using a network file system approach. A Plan B system is formed by multiple machines and devices. This is the common case in all systems and middlewares for building smart spaces and ubiquitous tools.

Before the work described in this paper, Plan B used a Kerberos based authentication scheme [9] together with a classic ACL based access control and a SSO system [9]. Although this scheme is very popular, it raised some issues. First of all, the SSO system was not really SSO. It was *per-machine*. Second, the security system had a single point of failure for all users, and required connection to the authentication server at all times. Third, ACL did not provide comfortable mechanisms to control the access to our devices. Fourth, users were not able to mutually authenticate themselves and share their resources when they were disconnected from the smart space. Last but not least, the scheme depended on central system administrators to add and remove new users and resources.

[*]This work is supported in part by spanish MCyT TIN-2004-07474-C02-02.

SHAD is based on a general purpose mobile terminal (e.g., a Pocket PC or a SmartPhone) that the user always carries around: the UbiTerm. The UbiTerm is intended to control the security of his owner and to control his activities [13]. We aim at using a general purpose mobile terminal because it provides other services (phone calls, mail, etc.) to the user. This is an important fact, because users will care more about the UbiTerm if it also provides other important services.

The scheme is simple. Terminals and networked devices[1] run a SHAD agent that provides authentication to local applications. The UbiTerm also runs a SHAD agent that provides authentication to all other agents running on terminals that belong to the user. SHAD agents are also able to cooperate through a Peer-to-Peer protocol to avoid obstruction, and support disconnections from the UbiTerm.

The agent running in the UbiTerm enables human to human authentication and controls the access to the user's resources [19]. When someone needs to operate on a resource that belongs to another user, their UbiTerms perform mutual authentication and a role based access control (RBAC).

The architecture may exploit context information [6], to reduce interruptions and to automate some operations. However, it does not depend or require this service, and still works at isolated locations.

SHAD follows the same approach used in Plan B, and avoids the need for middleware. Applications operate with the SHAD agent through a file system interface.

This offers a innovative approach to manage security in a permissive ubiquitous environment. Other security architectures for ubiquitous computing are highly centralized and depend on middleware or frameworks [1, 3, 4, 10]. There are another works that are based on authentication devices (see for example [12, 2, 8, 7]) and personal authentication servers, for example the Pervasive Authentication Gateway [17] and the Master Key [23]. But none of them provide both real Single Sign-On and a complete security architecture for resource sharing. Indeed, authors of [23] proposed to extend their architecture for providing authentication for all pervasive services, as future work. That is indeed done by SHAD.

2 Real Single Sign-On

The first advantage of carrying a personal security server is that it enables a *real Single Sign-On*. Real Single Sign-On means that the user only has to explicitly authenticate himself once to access to all services (both conventional client/server services and pervasive services) at any location. As far as we know, no other SSO system offers a real Single Sign-On in practice. This fact is further discussed in section 6.

The UbiTerm runs a SHAD agent that holds all secrets of the user, such as passwords, keys, certificates and so on. These secrets are obtained at boot time from a strongly encrypted file, the *secrets repository*. This file can be retrieved from a remote server or a memory card inserted in the UbiTerm. Note that the second case permits the user to boot the UbiTerm at isolated locations. It is decrypted with data obtained from the **unique explicit authentication** made by the user.

Once the agent knows all the user's secrets, it will serve them to agents running in other machines that belong to the same user. Protocols are detailed in the Appendix. The agent running in the UbiTerm is known as the *main SHAD agent*.

The rest of machines run another kind [2] of agent: a *plain SHAD agent*. These agents try to discover the main SHAD agent at boot time (through the PAN or the LAN in which it is connected).

To enable a secure discovering, each machine needs to have a shared secret with the UbiTerm. For this reason, each machine has stored a *terminal key* in its hardware. This terminal key is also stored in the secrets repository. Depending on the configuration, a confirmation may be required to respond to the discovery request. Confirmations are performed just by pressing a button in the UbiTerm.

When the main agent is discovered, a new session for this plain agent is set up. Following communications with the UbiTerm will be secured with a *session key* received in the discovery response.

Applications ask their local plain SHAD agent to provide authentication when needed. The local agent can provide authentication by executing the authentication protocol or by passing the specific secret to the application [9]. In any case, the local agent needs to know a secret (e.g., a password or a key) in order to provide authentication (in both ways).

Agents store secrets in main memory, which is protected against debugging and swapping to disk [9]. Secrets are not stored on disk under any circumstance. Therefore, if the agent (or the computer) reboots, all secrets are removed from memory.

Secrets are represented by plain text strings that include attributes to control them [9]. For example, a SSH password may be represented by the following string[3]:

```
proto=pass server=mar service=ssh user=pez
              !password=mypass
```

When a plain agent has to authenticate an application and it does not know the required secret, it asks the main

[1] In what follows we refer to any machine in the environment as a Terminal, but, it might be a different kind of machine.

[2] In fact, there is only one kind of agent that can assume two different roles.

[3] The '!' means that the attribute is secret and should not be printed.

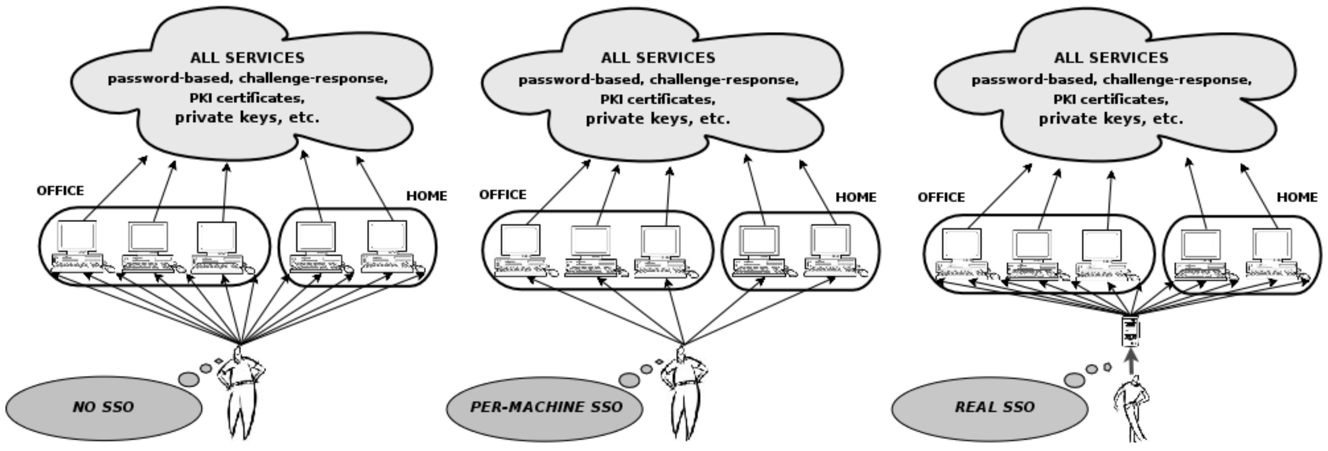

Figure 1. Obstruction vanishes when using SHAD to enable a real Single Sign-On.

agent. The main agent can distribute secrets according to additional constrains set by the user. For example: `noremoteaccess` means that the secret cannot be sent outside the local agent in any case; `needconfirm` forces explicit confirmation in order to send the secret to other machine (the user can confirm operations just by pressing a button on the UbiTerm); `userlocation = <location>` means that the secret can only be sent if the user is at the specified location. Note that if `needconfirm` is set, the operation also needs to be confirmed by the user. This location information is provided by an external context infrastructure[4]. If the context infrastructure is not available, the secret cannot be sent. Also, `clientlocation = <location>` works in a similar way, but taking into account the location of the machine that is requesting the secret. The `samelocation` attribute requires the user and the client machine to be at the same physical location. Last, `accessiblefrom = <machine>` makes the secret available only from a plain agent running on the specified machine.

Plain SHAD agents are able to cooperate in a P2P fashion to support disconnections from the UbiTerm.

When a plain SHAD agent starts a session with the main agent, it also obtains an *incarnation id* and an *incarnation key*. The incarnation id changes every time the main agent reboots. The incarnation key is also unique for each incarnation.

If a plain agent cannot reach the main agent to retrieve a secret, it will try to get it from other plain agents. Then, it broadcasts a request message announcing its incarnation id. The rest of the message is ciphered with the incarnation key. Plain agents within the same incarnation can respond to the request (if they know the required secret). The response is also ciphered with the incarnation key.

Finally, if the secret cannot be obtained from other peers, the plain agent will ask the user to provide it through any input device. In this case, obstruction is unavoidable.

Secrets that are not suitable to be distributed among plain agents, for example important secrets such as administrator passwords or private keys, must include a special attribute named `nopeeraccess`. When this attribute is set, the secret can only be sent from the main agent to plain agents (of course, taking into account the rest of its attributes). Secrets including any restrictive attribute (location, confirmation, or machine related) are not accessible through the P2P protocol.

The user can define a timeout to block the main agent. After this time of inactivity, the main agent will remove all secrets from main memory and will require a new explicit authentication to get them back from the secrets repository. This countermeasure would be effective when the user forgets the UbiTerm or loses it.

We do not consider physical access to the terminals by an attacker. If so, SHAD's SSO scheme does not introduce new security issues regarding traditional SSO schemes. Consider the following argument. When using a traditional centralized SSO system (such as Protocom SecureLogin [16]), all secrets are available from every machine (furthermore, the single sign on is lost because the user must authenticate once per machine!). If a machine was physically compromised, all secrets would be compromised: the attacker may steal the password that grants access to all secrets by many ways (e.g., sniffing the keyboard input). In SHAD, if an attacker stole a *terminal key*, he would access only those secrets reachable from the compromised machine (according to the their constraints). Physical ac-

[4]How to secure the context infrastructure is out of the scope of this work. We assume that context data is correct and reliable, but not fault tolerant. If a user does not trust this infrastructure, it suffices to avoid the location attributes. The trade-off, as we said, is evaluated by the user.

cess to terminals is critical in all cases, according to the *Big Stick Principle* [21, Chapter 4].

Our SSO scheme is highly configurable and flexible. Users preferring comfort to higher security can customize SHAD in order to serve all passwords with few or no confirmations. On the other hand, users preferring extra security (instead of more comfort) can disable the sharing for most important passwords and require confirmations for all requests. In real life, users prefer a combination of comfort and security and configure SHAD according to their own perspective.

3 Sharing resources between users

Another important advantage of using the UbiTerm is that it enables Human-to-Human interaction. The UbiTerms can manage the security when users want to share their resources. This way, centralized servers and huge office domains are not necessary, and users can share resources everywhere they meet. Centralized administration is avoided, and users can add new resources without depending on system administrators. This scheme makes the architecture Peer-To-Peer, where the Peer is the Human.

When resources of two different users are involved in an operation, the UbiTerms of their owners cooperate to provide authentication and access control.

To allow this cooperation, users with mutual trust must *pair* their UbiTerms. In this process, the two UbiTerms negotiate a *pairing key* that will permit them to authenticate each other and to communicate securely in the future. The pairing key is stored in the secrets repository together with the other secrets of the user. Thus, the pairing key is persistent. Pairing must be done only once at configuration time. It may be automated, for example using a short-range IRdA link that makes sniffing improbable. It may also be done manually, by inserting a passphrase in each UbiTerm.

We argue that trust is not transitive. Thus, pairing keys are only known by the UbiTerms of the two users involved.

Once the UbiTerms are paired, users must assign roles to each other in order to control the access to the resources. These roles are the mechanisms to fix the level of trust between users and control the access to devices.

3.1 Sharing Terminals

SHAD permits borrowing terminals that belong to another trusted user. In our case, terminals are computers (normally PCs or workstations) that run Plan B. We aim at permiting the user to work with a borrowed terminal as if it was his own one, for a bounded time.

When a user powers on a terminal that does not belong to him, the plain SHAD agent starts at boot time (just like we described in the previous section). Then, the agent gets the login name of the guest user retrieving it from the context infrastructure (the nearest person from the computer). If the context infrastructure is not available, the agent will prompt the guest user to type his login name. Note that this login name is used only to identify the person in front of the terminal, not for authenticating him.

Terminal borrowing is performed by an alternative main agent discovering protocol. We name this protocol *machine lending protocol*.

The identifier of the owner of the terminal is stored in its hardware together with the terminal key. If the owner of the terminal and the login name differ, then the machine lending protocol will be executed. To describe the protocol, suppose that Bob wants to use a terminal Ta, which is owned by Alice. The UbiTerms are represented as Ta^* (Alice's) and Tb^* (Bob's). Figure 2(B) depicts the steps of the protocol:

1. Ta broadcasts a message that announces that Bob may be trying to boot it on his behalf. This message includes data encrypted with Ta's terminal key, which can only be decrypted by Ta^*.

2. Tb^* receives the message and realizes that its owner may be booting the terminal. Then, Bob is forced to explicitly confirm the operation. If this confirmation is not performed, the protocol will fail. This confirmation, which is mandatory, ensures that the login name acquired by the terminal is correct and that no one is trying to impersonate him.

3. Tb^* asks Ta^* to authorize the booting of Ta on behalf of Bob. The message is secured with the pairing key between Alice and Bob. It also includes the ciphered data sent in step 1.

4. Ta^* sends the authorization to boot the terminal. It may require a confirmation (which is non mandatory). The message is secured with the pairing key. It includes a capability for Ta, ciphered with its terminal key.

5. Tb^* sends the authorization to Ta. This message is secured with the session key. It also includes the capability referred in step 4.

If the protocol is successful, the terminal boots and Bob can use it as one of his own terminals. Of course, the owner of the terminal may eventually revoke the lending. A simple message based on the terminal key forces the terminal to reboot in a specified time, warning the guest user about it.

Is it suitable to enable the SSO mechanisms for a borrowed terminal? We think that if the user wants to work with a borrowed terminal as if it was one of his terminals, it should take advantage of the SSO service. Anyway, we think that the SSO service has to be available in a limited

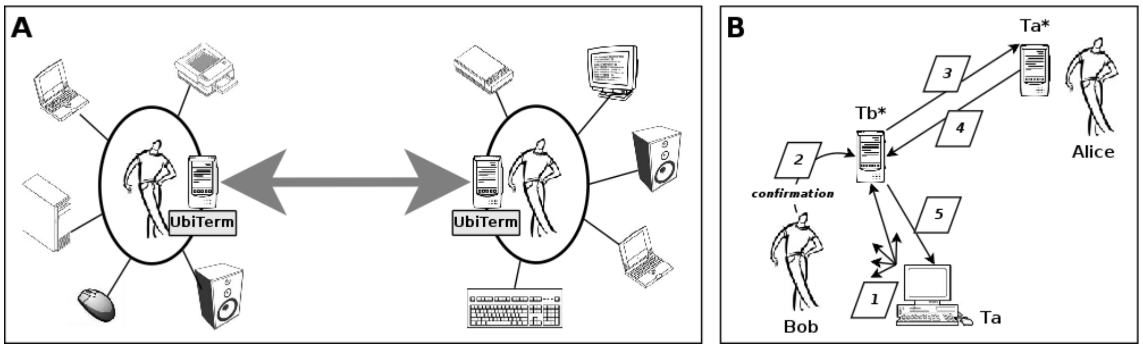

Figure 2. (A) shows the human-centered scheme adopted by SHAD. (B) depicts the steps of the machine lending protocol.

way. First, the P2P protocol in not available from borrowed terminals. Second, secrets cannot be sent to borrowed terminals by default. Only secrets including an attribute named `alienaccess` can be sent to a borrowed terminal.

3.2 Sharing Devices

Our system uses the P9SK1 authentication protocol [9], similar to Kerberos. We have placed a P9SK1 server in each UbiTerm. This server offers authentication to access the user's resources. In other words, it is a personal domain authentication server. The name of a personal domain is *user*`-shad`, being *user* the name of the owner.

The P9SK1 server has it own *key database*, that holds keys only for paired users. Accounts are automatically created when users pair their UbiTerms. Account keys are randomly generated by the UbiTerm when needed. It is important to remark that the UbiTerm's P9SK1 server is only another component and that it is transparent for the user.

To use a device that is exported through SHAD, the client must be authenticated by the owner's domain authentication server. No other authentication domain is allowed.

Next, we enumerate the steps for authentication when a paired user wants to mount a Plan B device. Figure 3 depicts the components and protocols involved in this process. Steps are numbered in the figure. Different kinds of lines indicate different protocols (SHAD protocols, P9SK1, and 9P). Being Tb and Ta terminals owned by Bob and Alice respectively, suppose that Tb needs to mount a device that is served by a terminal Ta:

1. The mount operation in Tb needs authentication to access to the domain named `alice-shad`. Then, the local plain SHAD agent running on Tb tries to obtain the required secret from its main agent (which is running in the UbiTerm, Tb^*). The required secret is the password of Bob's P9SK1 account in Alice's domain, `alice-shad`.

2. Tb^* doesn't have any secret related to this domain. Then, it realises that the required secret is related to a SHAD domain (`alice-shad`) and starts a *key refreshing protocol*. Tb^* broadcasts a message that only can be understood by Alice's UbiTerm, Ta^*. This message requests a new password for the P9SK1 server running on Alice's UbiTerm.

   This step may require a confirmation in the UbiTerm, depending on the UbiTerm's configuration.

3. Ta^* receives and validates the message. If it is correct, it will assign a new password to Bob in the P9SK1 database. This step may require an optional confirmation

4. Ta^* responds to Tb^* with a message that includes the new password for the domain `alice-shad`. The *key refreshing* protocol ends.

5. Tb^* sends the secret to Tb, and stores it in main memory together with the other secrets for future requests. The plain agent running on Tb stores the new secret together with the other secrets.

6. The plain agent of Tb^* performs the P9SK1 authentication protocol. Three parties are involved in this protocol: Ta^* 's P9SK1 server, Ta's SHAD agent, and Tb's SHAD agent.

7. The device file system is mounted through the 9P network file system protocol. The application is able to operate with the file system on behalf of Bob.

Messages in steps 2 and 4 are encrypted with the pairing key between Alice and Bob. Messages in steps 1 and 5 are encrypted with Tb's terminal key.

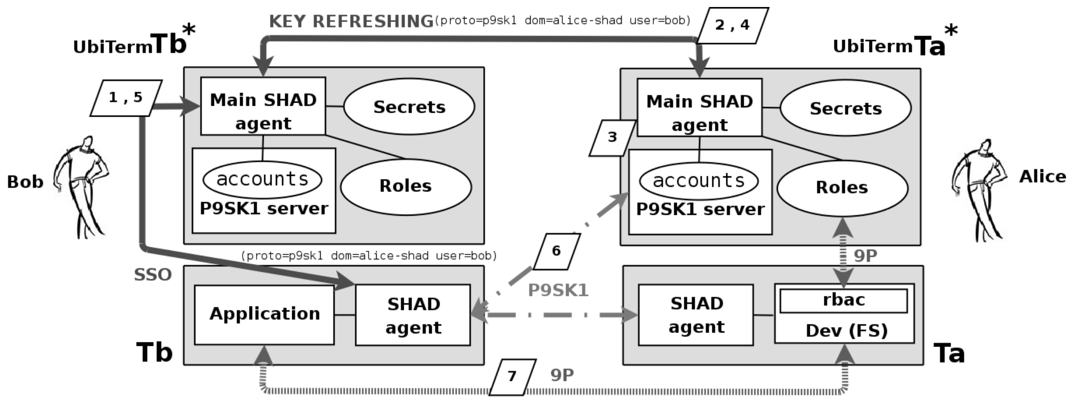

Figure 3. Diagram for device sharing that depicts components and protocols.

Note that these mechanisms provide authentication, but access control has not been performed yet. Authentication is performed once per mount. Subsequent account modifications wont affect already opened connections.

If Tb mounts another device owned by Alice and its local plain agent already knows a key for the domain `alice-shad`, its tries to authenticate using it. If the authentication fails, the plain agent will force the main agent to activate the protocol again in order to acquire a fresh key for this domain. The main agent is also forced to execute the protocol when a terminal retrieves the key through the SSO protocol and it has expired.

This scheme permits to control the authorization lifetime. Key modification in the P9SK1 database invalidates the old key and forces the client to run the protocol again.

SHAD uses a role based access control (RBAC) for the user's devices. Devices exported through SHAD use a modified access control library that enables RBAC. On the other hand, devices that are required to be exported in a classic way use the traditional access control library based on ACLs.

The RBAC library uses a *role database*, which is stored in the UbiTerm. The database is kept in a file named `shadroles` and is composed by two sections (roles and assignments).

A role is composed by its name and a list of devices (file servers) that can be accessed according to some constraints. Constraints can be formed by two modifiers: `-R` and `-W`. After the modifier, a file name can be specified. In this case, the constraint is applied only for this file. If no file is specified, the constraint is applied to all files within the device's interface. If no modifiers are specified, the file system will apply the default access mode.

Roles are assigned to users in the second section of the file. Each line enumerates the roles played by a user. An special character '*' means that the user can play any role defined in first section. If two roles assigned to the same user and they define different access rights for the same device, then the least restrictive one will be applied.

When the user has mounted the file system, file permissions are adjusted dynamically according to the role database. For files exported by SHAD, permissions assigned to the group `shad` are in fact the permissions for the user; only that they are not hardcoded in the ACL.

Suppose that this is the `shadroles` file stored in Alice's UbiTerm:

```
role input = kbdfs micfs scanfs camerafs mimiofs -W
role output= mfs -Wvolume printerfs voicefs
role omero = omero
role lab   = hxfs x10fs -Wpwr:124term -Rwho:outside \
             camerafs -R
role guest = projectorfs  mousefs
paurea     = output lab
katia      = input output
foo        = guest
nemo       = input lab
elf        = *
```

Let's assume Bob wants to mount any audio device owned by Alice and located at office number 124. Audio devices are exported by the `mfs` file system. Once it is mounted, he lists the content of its root directory, reads the volume settings, sets the volume to the 50%, and plays a MP3 file:

```
; echo $user                              # show my user name
bob
; mount /srv/vol /n/audio '/devs/audio user=alice loc=124'
; cd /n/audio
; ls -l                     # list the audio device interface
---w--w---- M 95 alice shad 0 Jun 28 13:32 audio
--rw-rw---- M 95 alice shad 0 Jun 28 13:32 volume
; cat volume                    # obtain the volume settings
audio out 47
treb out 0
bass out 0
speed out 44100
; echo audio out 50 > volume               # set volume to 50%
;                     # play the  MP3 file using the device
; cp /n/music/heavy/ACDCHard.mp3 audio &
;
```

By default, the `volume` file has only read permission for SHAD users. But Bob plays the `audio` role, according to the `shadroles` file. This role has a constraint `-W` applied to the file `volume` of the `mfs` device. Therefore, Bob has both read and write access and he can set the volume. If the `-W` constraint was not assigned to the role, Bob would see the following permissions:

```
; ls -l
---w--w---- M 95 alice shad 0 Jun 28 13:32 audio
--rw-r----- M 95 alice shad 0 Jun 28 13:32 volume
; echo audio out 50 > volume
echo: can’t open volume ’volume’ permission denied
```

Devices that do not belong to an specified user (e.g., many devices located in shared offices) can either use the old security scheme, or use SHAD in a different way. For example, we can use the SHAD scheme creating a virtual user and using a shared server as its UbiTerm. In this case, the administrator should pair the virtual user with the other users.

3.3 Disconnections and Access Revocation

Disconnections from the UbiTerm are supported. If the UbiTerm leaves, read/write operations in progress keep working. In fact, once a file is opened, it can be used even the UbiTerm goes away, because authentication and access control was performed on the mount and the open operations respectively.

To revoke access to devices, the user only has to delete or comment out the corresponding line in the `shadroles` file.

To completely remove (or *unpair*) a user, the following actions must be performed: (i) remove his pairing key string from the secrets repository, (ii) remove his account from the P9SK1 database, and (iii) remove his line from the `shadroles` file. These actions can be easily automated using a tiny shell script.

Revocation is non instantaneous when the user that is being denied access is still working with an open file. In this case, he can continue using the file until it is closed. To avoid this situation we would have modified the Plan B kernel to be able to close all files opened by a specific user name.

4 Implementation and Experience of Use

The SHAD agent prototype is approximately 12500 lines of C code. The agent has a P2P design and includes both main agent’s and plain agent’s logic. This implementation is a Factotum [9] derivative. Factotum is the Plan 9 [15] security agent, which provides *per-machine* SSO through a virtual file server interface. We have added all the mechanisms to permit the agents running in different machines to cooperate. The prototype also uses Secstore [9] to store the secrets repository.

Due to its file system interface, applications using SHAD do not depend on any kind of middleware or framework. To add secrets, the user only has to write strings in a virtual file. Applications also get authentication services through the file system interface.

A SHAD graphical front-end has been built for Omero, the Plan B’s pervasive graphical system[5]. This graphical front-end uses the Plan B’s voice service to warn the user about some situations.

SHAD takes advantage of our context infrastructure, in which users are located through ultrasonic transceivers and X10 sensors [6].

The prototype uses the AES algorithm in CBC mode with 256-bit length keys to encrypt communications between agents. It also uses SHA-1 to ensure the integrity of messages.

Terminal keys are stored in NVRAM. We are looking for hardware alternatives to the NVRAM. A solution would be to use any tamper resistant hardware using challenge-response methods in order to authenticate machines.

First, we would like to show some measures about the number of explicit authentications that must be performed in a Plan B desktop formed by three terminals and three large displays controlled by Omero. Table 1 shows the number of explicit authentications required by the author in a real setting for five working days.

	No SSO	Per-machine SSO	SHAD
Machine #1 (UbiTerm)	129	10	10
Machine #2	73	10	0
Machine #3	70	16	0
Total	272	36	10

Table 1. Number of authentications required by the system in a working week.

Daily work consists of numerous tasks, such as programming, compiling, editing LaTeX, and so on. Note that authentication is needed to perform quite frequent tasks, for example to use the same mouse for different terminals (authentication is needed each time the mouse is redirected). In table 1, *machine #1* represents the primary terminal to which both the keyboard and the mouse are attached. This terminal runs the main agent and acts as the UbiTerm in the experiment. The two other terminals are represented as *machine #2* and *machine #3*. The first column shows the number of explicit authentications that were performed when avoiding any SSO system. The second column shows the value when using a *per-machine* SSO system. The third column shows the value when using SHAD SSO.

We can observe how SHAD reduces considerably the obtrusiveness of the system, even when using only three terminals. Note that the number of required authentications measured when using SHAD is higher than usual, due to the tasks that were performed during the week of the experiment. These tasks included kernel debugging, which required several system reboots. In the common case, the user only has to authenticate himself once per day.

Next, we present some measures to evaluate the prototype. Experiments have been performed on Pentium 4 computers connected by a 100 Mbps network.

SHAD single sign-on is fast enough from user's the point of view as shown in Table 2. The table shows the mean time to connect to a SSH server from a client using Factotum [9] and SHAD. SHAD measures are presented for two cases: (i) retrieving the key from the main agent, and (ii) retrieving the key from other plain agents (P2P protocol).

	Factotum	SHAD	SHAD P2P
Time (s)	0.12	0.14	4.06

Table 2. Average time for a SSH connection.

Note that the P2P case would be needed once per-machine and per-service because SHAD will remember the secret for the next time. The P2P mean time is highly under the influence of the main agent discovery's timeout.

Table 3 shows the overhead caused by the use of a real context infrastructure in order to avoid confirmations. The first column shows the average time to mount a Plan B file system when no confirmation is required. The second column shows the average time when using the location information provided by the context architecture to avoid a confirmation. The overhead caused by the use of the context infrastructure is barely perceptible by a user.

	no restrictions	`userlocation`
Time (s)	0.15	0.17

Table 3. Average time for mounting a Plan B volume.

Table 4 shows the average time when using a shared device through SHAD. The experiment was performed in three Pentium 4 computers. One computer emulated the two involved UbiTerms in virtual machines. The others ran the client and the file server. Measures present the average time to perform the following operations: (i) mount a Plan B volume, (ii) open a file in read mode, (iii) copy its contents to a local file, (iv) close the remote file, and (v) unmount the file system. The first column shows the result when using no authentication and ACL access control. Second and third columns show the results when using P9SK1 authentication and RBAC access control. The second column presents the best case, when the plain agent running in the client's machine already knows a (valid) P9SK1 key. The third column presents the worst case, when the P9SK1 account must be refreshed.

	No auth/ACL	SHAD/P9SK1/RBAC BC	SHAD/P9SK1/RBAC WC
Time (s)	0.024	0.207	0.308

Table 4. Average time to mount, use, and unmount a Plan B volume exported by SHAD.

Measures show that the mechanisms do not increase enough the latency to be noticed by the user. For the user, device sharing is completely transparent and immediate. Times will get worse when using poor wireless links and a mobile device as UbiTerm. But obtained results are quite encouraging.

5 Restrictions of the Architecture

The use of a mobile device to represent and authenticate the user raises an important issue that has to be taken into account. On the one hand, it enables independence of centralized entities, provides a real Single Sing-On, and permits us to build a human-centered architecture to share devices intuitively. On the other hand, it introduces a new risk: the device can be lost or stolen, enabling impersonation attacks. In this case, automated revocation is complicated.

We argue that the advantages outweigh the disadvantages. In real life, we use several important physical objects that can be lost or stolen. But these objects permit us to perform tasks that make our life easier. For example, we might lose our credit card (and therefore a lot money and time for complaining). We all are aware of the trade-off between comfort and security regarding credit cards. Of course, we care a lot about our credit cards, and we try keep them in a safe place. But they might be lost or stolen anyway. We think that the same criteria must be applied to the UbiTerm. In general, security is a trade-off between safety and costs [18]. The user must be aware of the risks of losing the UbiTerm, and configure SHAD according to his preferences. If the user loses the UbiTerm, he should notify paired users or activate available mechanisms to revoke access. Automatic revocation is part of future work.

Another important issue is the UbiTerm's power consumption. If the UbiTerm runs out of battery, all authentication services regarding the user are unavailable. This leads to enables denial of service attacks against the UbiTerm. Note that this risk is always present when using mobile devices.

We think that the problem is not critical in practice because we are facing it everyday in other services, for example in our mobile phones. In fact, we aim to locate the UbiTerm in a general purpose device in order to make the user more dependent on it. First, we understand that the user is aware of the problem and cooperates, for example keeping the UbiTerm in its cradle while working at the office or home. In addition, the user might have several batteries due to their low cost and availability.

In addition, SHAD's design also cooperates to alleviate the problem. Due to the P2P design of the SHAD agent, any agent can become a main agent, The user may replace the UbiTerm just by extracting the memory card that contains the secrets repository and introducing it in other of his terminals (mobile or not). Moreover, SHAD protocols minimize the number of messages and use the AES encryption algorithm, which is suitable for limited devices.

Finally, we want to emphasize again that SHAD offers the mechanisms and not the policies, which are set by the user according to his preferences.

6 Related Work

There is a lot of related work in the fields of Single Sign-On and security for ubiquitous computing. In this section we try to focus in the differences between SHAD and the most relevant and closest works.

Simple SSO agents permit single sign-on for a single service, for example the SSH agent. Web browsers remember passwords and provide SSO for web applications. Other SSO systems provide single sign-on for different services [9, 16]. These systems offer per-machine SSO: the user must authenticate himself at least once per machine, therefore there is not a Single Sign-On to the smart space. Moreover, most of these systems depend on a centralized server. Thus, SSO is not available at isolated locations.

Like SHAD, the Pervasive Authentication Gateway [17] uses a personal device for enabling SSO. But PAG does not offer a real SSO because it forces the service providers to be modified in order to accept a challenge-response authentication scheme. This is not realistic. As a result, PAG is not potentially compatible with all services, unlike SHAD. SHAD supports long-term disconnections from the UbiTerm, because it distributes non-transitory capabilities (the secrets) to provide authentication to other machines. On the other hand, PAG distributes short-term tokens. To that effect, PAG is more conservative when distributing authorizations, although it does not offer a solution for our requirements. In addition, SHAD is P2P, because plain agents can cooperate to provide authentication to applications when the main agent is gone. There are other differences between SHAD and PAG, such as configuration options, confirmations, location related mechanisms, and so on. Finally, SHAD is not only a SSO system; it offers a complete solution to share resources in a Plan B based smart space.

The Master Key [23] also uses a personal device for authenticating the user to perform specified tasks (e.g., open door locks). Its authors suggested to extent its scheme to provide authentication to the entire smart space. SHAD does it now.

The use of authentication devices, such as Smart Cards or iButtons, has been proposed by several related works, but they are normally used to authenticate users for specific and *ad-hoc* purposes. They require the authentication hardware and are not general purpose authentication mechanisms that could be used for any service in the system. Many schemes based on authentication devices are obtrusive and offer *per-machine* SSO. For example, CryptoToken[12] is a USB device that holds the secrets of the user. When the user needs to work with a machine, he connects the CryptoToken to it. This approach is only a bit less obtrusive than typing passwords, but it is still a burden for the user. In addition, not every device has USB ports. Finally, it does not support concurrent authentications in different machines, because the user only has one CryptoToken. However, SHAD does.

Classic security systems like Kerberos [22] and Sesame [11] provide authentication, but they depend on centralized services and are hard to administer and they provide per-machine Single Sign-On.

Other architectures use trust estimations based on reputation, recommendation and experience [14].SHAD does not try to simulate human behavior. Instead, it offers an architecture that allows humans apply their real trust relationships in order to share their belongings.

Resurrection Duckling [20] presents an innovative trust scheme. It proposes secure transitory associations between devices and humans in an ubiquitous environment. This scheme supposes that the hardware is a communal good. Instead, we suppose that devices always have an owner, which is probably a more realistic supposition. SHAD sets non-transitory associations between humans and their belongings, and non-transitory trust relations between humans. This way, SHAD can subsequently set transitory associations to enable device sharing in a natural and intuitive way.

Other security schemes have been proposed for ubiquitous environments. Most of them depend on complex middleware architectures and are highly centralized. In general, middleware based architectures make developers depend on specific platforms or languages. For example, [1, 3, 4] are heavily based in centralized security schemes (like Sesame and Kerberos) and therefore have the same problems cited above. SHAD does not.

7 Conclusions and Future Work

The main contributions of this paper are: how to build a human-centered security architecture for permissive ubiquitous environments, a way to provide a real Single Sign-On when users work with several machines at same time, a simple and intuitive solution to share resources even at isolated locations, a functional prototype implementation, the results of some experiments and some notes about our experience of use.

Our architecture does not require complex administration, and it is independent of centralized services. It does not depend on any kind of middleware or framework, because it relies on distributed file system technology and can be used through a virtual file system interface.

As far as we know, all these properties make SHAD distinct and unique from all other systems presented in the literature.

Future work includes a full port of the UbiTerm to a Linux based mobile device and the design of automatic revocation mechanisms to minimize risks in case of UbiTerm usurpation.

Apendix: Protocols

SHAD protocols use timestamps and nonces to assure the freshness of messages and to avoid replay attacks. Timestamps define the freshness of received nonces. Nonces are stored in a black list for a time δ. The black list is big enough to store nonces received in a time δ under intensive usage. If there is a black list overflow, the agent will suppose that there is an attack and will shut down. Being $time$ the local time, the timestamp makes the message old if

$$|time - timestamp| \geq \delta$$

Therefore, δ defines the maximum time difference between nodes. We have set δ to 1800 seconds according to our experience of use. But note that machines usually acquire the time through NTP if they are on line.

In what follows, Ta is a terminal owned by Alice (A) and Ta^* is her UbiTerm. In the notation, $(D)_K$ means that the data D is ciphered using the key K. K_{Ta} represents the terminal key and K_s the session key. I_{id} and K_i represent the *incarnation id* and the *incarnation key* respectively.

The *main agent discovery protocol* is formed by two messages:

$Ta \longrightarrow BROADCAST$:
$A, Ta\ (A, Ta, N_1, N_2, TSTa)_{K_{Ta}}$

$Ta^* \longrightarrow Ta$:
$(A, Ta, N_1+1, Ta^*, TSTa^*)_{K_s}$
$(A, Ta, N_2+1, TSTa^*, K_s, K_i, I_{id})_{K_{Ta}}$

The response is formed by two pieces in order to decouple the session key generation from the authorization itself.

The *machine lending protocol* is formed by four messages. As we have explained before, it is an alternative to the discovery protocol. The machine lending protocol is executed when the user powers on a machine that do not belong to him. Suppose that Bob (B) boots a terminal (Ta) that belongs to Alice (A). The pairing key between Alice and Bob is represented as $K_{a,b}$. Alice's and Bob's UbiTerms are represented as Ta^* and Tb^* respectively. The messages are:

$Ta \longrightarrow BROADCAST$:
$A, Ta, B\ (B, Ta, N_1, N_2, TSTa)_{K_{Ta}}$

Bob must confirm the booting of Ta on his behalf.

$Tb^* \longrightarrow Ta^*$:
$(B, Ta, N_3, TSTb^*)_{K_{a,b}}$
$(B, Ta, N_1, N_2, TSTa)_{K_{Ta}}$

$Ta^* \longrightarrow Tb^*$:
$(B, Ta, N_1+1, N_3+1, TSTa^*, K_s)_{K_{a,b}}$
$(B, Ta, N_2+1, TSTa^*, K_s)_{K_{Ta}}$

$Tb^* \longrightarrow Ta$:
$(B, Ta, N_1+1, Tb^*, TSTb^*)_{K_s}$
$(B, Ta, N_2+1, TSTa^*, K_s)_{K_{Ta}}$

The *SSO protocol* is formed by two messages. It is executed when a plain agent needs a secret to provide authentication to an application. Suppose that an application running on Ta needs authentication. Let R be a string that enumerates the attributes for the required secret (protocol, server, client, user, etc.). Let S be the string that defines the secret and all its attributes. Let K_s be the session key assigned to Ta. The messages are:

$Ta \longrightarrow Ta^*$:
$A, Ta\ (A, Ta, N_1, TSTa, R)_{K_s}$

$Ta^* \longrightarrow Ta$:
$(A, Ta, N_1+1, TSTa^*, S)_{K_s}$

Key refreshing protocol:

$Tb^* \longrightarrow BROADCAST$:
$A, B\ (A, B, N_1, TSTb^*)_{K_{a,b}}$

$Ta^* \longrightarrow Tb^*$:
$(A, B, Ta^*, N_1 + 1, TSTa^*, K)_{K_{a,b}}$

Remaining SHAD protocols are similar to the ones described above.

References

[1] J. Al-Muhtadi, M. Anand, N. D. Mickunas, and R. Campbell. Secure smart homes using jini and sesame. In *Proceedings of the 16th Annual Computer Security Applications Conference*, New Orleans, USA, 2000. IEEE Computer Society.

[2] J. Al-Muhtadi, D. Mickunas, and R. Campbell. Wearable security services. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 226–232, 2001.

[3] J. Al-Muhtadi, A. Ranganathan, R. Campbell, and N. D. Mickunas. A flexible, privacy-preserving authentication framework for ubiquitous computing environments. In *Proceedings of IWSAEC 2002*, 2002.

[4] J. Al-Muhtadi, A. Ranganathan, R. Campbell, and N. D. Mickunas. Cerberus: A context-aware security scheme for smart spaces. *IEEE International Conference on Pervasive Computing and Communications*, 2003.

[5] F. J. Ballesteros, K. Leal, E. Soriano, and G. Guardiola. Omero: Ubiquitous user interfaces in the plan b operating system. In *Proceedings of the Fourth IEEE International Conference on Pervasive Computing and Communications*, pages 77–83, 2006.

[6] F. J. Ballesteros, E. Soriano, K. Leal, and G. Guardiola. Plan B: An operating system for ubiquitous computing environments. In *Proceedings of the Fourth IEEE International Conference on Pervasive Computing and Communications*, pages 126–135, 2006.

[7] A. Beaufour and P. Bonnet. Personal servers as digital keys. *Second IEEE International Conference on Pervasive Computing and Communications*, 2004.

[8] L. Bussard and Y. Roudier. Authentication in ubiquitous computing. In *Proceedings of Ubicomp 2002, Workshop on Security in ubiquitous computing*, 2002.

[9] R. Cox, E. Grosse, R. Pike, D. Presotto, and S. Quinlan. Security in plan 9. In *In Proceedings of the 11th USENIX Security Symposium*, pages 3–16, San Francisco, USA, 2002.

[10] R. Hill, J. Al-Muhtadi, R. Campbell, A. Kapadia, P. Naldurg, and A. Ranganathan. A middleware architecture for securing ubiquitous computing cyber infraestructures. *IEEE Distributed Systems Online*, September 2004.

[11] P. Kaijser, T. Parker, and D. Pinkas. Sesame: The solution to security for open distributed systems. *Computer Communications, vol. 17, pp. 501-518*, 1994.

[12] H. Kopp, U. Lucke, and D. Tavangarian. Security architecture for service-based mobile environments. In *Proceedings of the 2nd International Workshop on Middleware Support for Pervasive Computing, IEEE PerCom 2005*, pages 199–203, Hawaii, USA, 2005. IEEE Computer Society.

[13] K. Leal, F. J. Ballesteros, G. Guardiola, and E. Soriano. UbiTerm: a hand-held control-center for user's activity mobility. In *Proceedings of the IEEE International Conference on Pervasive Services 2005*, pages 127–136. IEEE Computer Society, 2005.

[14] H. Lee and K. Kim. An adaptive authentication protocol based on reputation for peer-to-peer system. *Symposium on Cryptography and Information Security (SCIS) 2003*, 2003.

[15] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from bell labs. In *In Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9, London, UK, 1990.

[16] Protocom, Protocom Development Systems. *SecureLogin Single Sign-On White Paper*, 2003. http://www.protocom.com/html/whitepapers/.

[17] R. Sailer and J. R. Giles. Pervasive authentication domains for automatic pervasive device authorization. In *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 144–148, 2004.

[18] B. Schneier. *Beyond Fear: thinking sensibly about security in an uncertain world*. Copernicus Books, New York, NY, 2003.

[19] E. Soriano. Shad: A human centered security architecture for partitionable, dynamic and heterogeneous distributed systems. In *Proceedings of the 5th ACM/IFIP/USENIX International Middleware Workshops (1st International Middleware Doctoral Symposium)*, pages 294–298. ACM Press, 2004.

[20] F. Stajano. The resurrection of duckling - What next? In *Proceedings of the 8th International Workshop, LNCS 2133*, Cambridge, UK, 2000. Springer-Verlag.

[21] F. Stajano. *Security for Ubiquitous Computing*. John Wiley and Sons, 2002.

[22] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 USENIX Conference*, Dallas, USA, 1988. USENIX Association.

[23] F. Zhu, M. W. Mutka, and L. M. Ni. The master key: A private authentication approach for pervasive computing environments. In *Proceedings of the Fourth IEEE International Conference on Pervasive Computing and Communications*, pages 212–221, 2006.