XtreemOS-MD SSO

A Plugable, Modular SSO Software for Mobile Grids

José María Peribáñez, Alvaro Martínez, Santiago Prieto, and Noé Gallego

Telefónica I+D
{e.xtreemos-jmp,amr,spm,e.xtreemos-ngm}@tid.es

Abstract. XtreemOS-MD SSO is a modular, pluggable, Single Sign-On (SSO) architecture. It has been conceived for easy integration of mobile devices into the Grid as part of XtreemOS project, but it may be reused by any other project. It offers semi-transparent integration with applications and makes easier the migration from enterprise servers to cloud computing infrastructures.

XtreemOS-MD SSO is inspired in Linux Key Retention Service (LKRS) with some enhancements and may interact with it, but it's designed to run completely in user space, not requiring any special kernel support.

Keywords: Single Sign-On, Key Retention Service, Security, Mobile Devices, Grid, Cloud Computing.

1 Introduction

When trying to integrate the Grid in a mobile phone, every solution should face two unavoidable requirements in order to succeed: the solution should be secure and it should be also extremely simple to use, as the users may not have any computing knowledge. Likewise, the mobile phone interface is very limited, so it's important to keep a minimal interaction with the user, but not exceeding some limits like storing permanently the password in the phone and acting without requesting the user's permission.

Taking into account those requirements, a good solution is a Single Sign-On (SSO) [1][2] system. The first time a user's application needs the Grid services, the session startup process will be launched and no more intervention will be requested to the user during the rest of the session.

A solution based on a SSO system offers a lot of benefits for the users, but its acceptance could fail if it's necessary to make profound adaptations in applications. Ideally, it should not be necessary to modify the source code of 3rd party applications, or at least in the worst case, just minimal changes would be required, as it's the case with Kerberos and "kerberized" applications [3]. Nevertheless, it's quite convenient to support a wider range of possibilities, as some projects and organizations prefer the use of a Public Key Infrastructure instead of Kerberos (e.g. Globus [4], XtreemOS [5]). It's also important that the solution proposed does not affect those applications not integrated with the SSO system.

A.U. Schmidt et al. (Eds.): MobiSec 2010, LNICST 47, pp. 27–38, 2010.

Moreover, the solution should be usable by any kind of program, either written in native code or in other languages like Java.

In order to design an easy-to-deploy SSO solution which does not compromise the future computer-infrastructure evolution, a modular architecture is desirable. Thereby, it should be possible to change the authentication mechanism without modifying the applications (e.g. just editing a configuration file) and following the philosophy of PAM (Pluggable Authentication Modules) [6]. It's also convenient to differentiate the part of the code involving user's interaction, in order to be able to adapt the system to the different interfaces provided in mobile phones and PCs.

XtreemOS-MD, the XtreemOS[7] version for mobile devices like smartphones or PDAs, provides an open-source SSO solution covering the requirements and goals previously mentioned. XtreemOS is a Linux-based operating system to support Virtual Organizations (VOs) for Grids, that it's being developed as an European project partially funded by the European Commission (Contract IST2006-0033576)

XtreemOS-MD SSO has been developed to cover the specific security-requirements imposed by XtreemOS (like the use of X.509 certificates obtained from a server known as CDA), but the architecture is completely modular and independent of the credential's type (private key-public key pair, password, Kerberos token, etc.), where the XtreemOS specific part is implemented by one concrete module.

It's worth noting that, thanks to its modular architecture, XtreemOS-MD could be used to implement first a non-Grid SSO system based on Public Key Infrastructure (PKI) using X.509 certificates and an internal CA, migrating subsequently the servers to a Cloud Computing system over XtreemOS without changing nothing but some lines in a configuration file.

2 SSO and Key Retention Service

There are many SSO architectures [8], but the usual model for SSO consists of splitting the process into two different phases. The first one is in charge of obtaining the user's credential and storing it in a local cache (which from now on will be referred as *credstore*). The second one implements the authentication process against any service using the cached credential.

An implementation of a SSO system may cover both phases by providing to the upper-layer applications a single library that manages both the obtainment and the use of the credential. But it is also possible to cover just the first phase, and then let the application to make use of the credential by itself. This second option is interesting as well, as the authentication mechanisms could be very dependent on the application or protocol used. For instance, for the same type of credential based on X.509 certificates, a XMPP [9] client application could use SASL [10], whereas SOAP based applications will use WS-Security[11], and other applications will work over EAP-TLS[12] or create directly a SSL/TLS[13] connection, and there may be even two solutions with the same authentication

system, such as SASL, but using two different libraries. Hence, forcing a concrete implementation on the authentication process could require major changes in the application code.

This approach, consisting on the implementation of just the first phase, is adapted by the Linux Key Request/Retention Service (LKRS) [14], which is part of the Linux Kernel. LKRS adds new system calls in order to obtain a credential at user, session, process or thread level.

3 Beyond LKRS

LKRS is a well designed and very flexible solution in order to implement a SSO. Even if LKRS does not manage modules, it can be extended to support a modular architecture without modifying the LKRS source code. This is possible because the parameters received by the LKRS `request_key` call are first used to determine the program to be invoked to obtain the credential (following the rules of `/etc/request_key.conf` file) and then are passed as parameters to such program.

Another feature that may be added without modifying LKRS core implementation is a mechanism that allows applications to invoke `request_key` without accessing their source code.

Unfortunately, sometimes LKRS may not be an option, especially when targeting embedded devices: LKRS is a specific solution for Linux and its porting to other operating systems is not trivial. Another issue is that LKRS is not compiled in the default kernel and it cannot be loaded as a module. Many embedded systems permit users to install applications, but not kernel modifications without a firmware update (re-flashing the device). This implies, that in practice, a SSO based on LKRS may be distributed by the smartphone manufacturer, but not by 3rd party vendors.

Even in systems where LKRS is already included this problem also may occurs: for example to change the quota for the *credstore* content, the kernel code must be modified.

The LKRS implementation in kernel space offers some benefits, like for example the integration with modules that are partially executed in the kernel as several network files systems. But most security developers and researchers feel more comfortable studying and adapting user space code than kernel space code, which requires different technical skills.

Moreover, some implications in LKRS may be obviated if it's re-implemented in user space. For example, if LKRS is used, the application to obtain the credentials is invoked with root privileges in a different session. Running a solution in user space allows invoking the application in the same user session, thereby running without root privileges and inheriting the environment variables.

The Xtreemos-MD SSO solution allows both the integration with LKRS and its replacement when not available. It's worth noting that it's not a complete replacement of LKRS, but just the required functionality to implement the SSO.

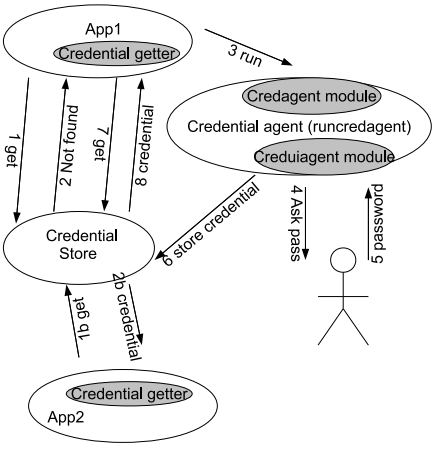

Fig. 1. XtreemOS MD SSO in action

4 Architecture

As explained before, our SSO system should work as an LKRS extension or as an independent solution when LKRS is not available. Then, in order to achieve these objectives, the implementation of the architecture has been divided into three libraries:

- credential store: in charge of securely caching credentials. It can make use of LKRS or a different implementation in user space. In the first case, just the LKRS retention service is used, but not the functionality offered by LKRS to obtain the key. This library offers an API to check if there is a credential available, to read, store or remove the credential and to set its expiration time.
- credential agent: this library is in charge of obtaining the credential through the invocation of two modules, which are specified in a configuration file. The first module involved is the *credagent* module, for the intrinsic process of credential retrieval (which could involve an authentication process against a server or reading a locally stored encrypted key, for example). If the *credagent* module needs user interaction (e.g. to ask a password or request a confirmation) it will invoke the other type of module, the *creduiagent*.
- credential getter: this library provides the only API needed by the applications to obtain the credential. It implements a functionality equivalent to the LKRS `request_key` call in user space, but offering a simpler API. First, the credential store library is invoked to check if the credential is already cached, invoking if not an specific program (let's call it *runcredagent*) that will invoke the credential agent library to obtain the credential and to store it finally (using the credential store library) in the *credstore*.

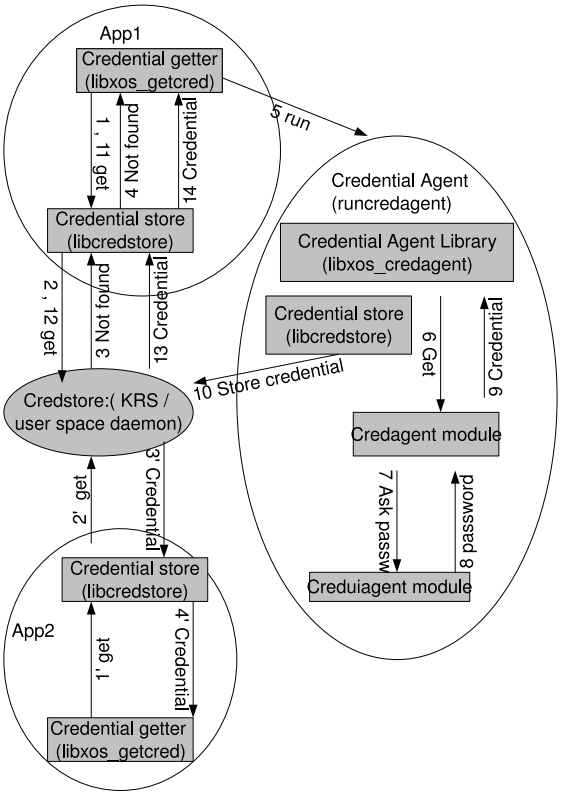

Fig. 2. XtreemOS MD SSO detailed interactions

Figure 1 shows a general view of XtreemOS-MD SSO in action: an application tries to retrieve the credential, and not finding it, launches *runcredagent*, while an application invoked lately gains access to the credential directly through the credential credstore.

Figure 2, shows the same case, but with deeper detail on the internal communication flow between the different modules. First, the credential getter library invokes the credential store library to try to recover a cached credential. When the credential is not found (steps 1-4), it launches *runcredagent* (step 5), which instantiates the *credagent* module using the credential agent library, and invokes the method to get the credential (step 6). The *credagent* module requests a password to the user through the *creduiagent* module (step 7 and 8). The *credagent* module obtains the credential (e.g. reading it from disk and decrypting with the user's password) and returns it to *runcredagent* (step 9), which invokes the credential store library to cache the credential (step 10) and exits. Meanwhile, the credential getter library waits until *runcredagent* ends and then invokes again the credential store library: this time the credential is found (steps 11-14).

A second application using the credential getter library obtains the credential directly from the *credstore* (steps 1'-4').

Apart from the three mentioned libraries, the architecture also provides some additional mechanisms and libraries to let the applications invoke the credential getter library in a transparent way, without any source code modification. This solution is available for applications that read the credential (e.g. a X.509 certificate and private key) from a configurable file path.

4.1 Credential Store Library

The credential store library, abstracts some LKRS features making them available in systems where LKRS is unavailable. In order to implement a SSO system, just part of the functionalities offered by LKRS are needed. It's not necessary for instance the implementation of the *credstore* at process or thread level, but just at user level. Even if it's not strictly needed, it's also interesting the implementation of a session-scope *credstore*, limiting the access to the credential just to the descendant processes of the process that originally started the session. The credential store library offers three different implementations:

1. a LKRS wrapping named *krs*
2. *zkrs*, a different wrapper which uses LKRS but includes live credential compression/decompression
3. *uskeystore*, which is an implementation through a user space daemon

The concrete implementation to be used can be selected at runtime; by default, *zkrs* is selected if LKRS is available and *uskeystore* if not.

The implementation of *uskeystore* is based on the execution of a daemon (named *xos_credstored*) that is automatically launched when storing the credential, ending when the credential expires or is removed. One of the main challenges of the implementation is the communication between the library and the daemon, especially concerning the access control. Unix sockets are one of the most widely used solutions (X-Window, D-BUS or OpenSSH for example use this solution), as they allow the credential transfer proving the possession of a concrete UID to the other side of the communication. But the biggest challenge of the *xos_credstored* access control is related with the session-scope of *credstore* as the access should be granted only to the process and its descendants, but not to the rest of user's processes. One possible solution consists of the use of a random virtual path for naming the Unix socket instead of a fixed default path, copying the path in an environment variable that will be inherited only by the child processes. This method is required to avoid Denial-of-Service (DoS) attacks, but it's not an effective access control by itself, as the Unix sockets paths are not secret and can be read from `/proc/net/unix`. Also, if *procfs* is installed, every user's process can search in `/proc` the UID of a process involved in the session and access its environment variables, obtaining then the virtual path randomly generated. This is the reason why passing a cookie through an environment variable it's not a fully secure mechanism.

It might be possible to verify that a process is descendant of the session creator, determining the PID of the other side (and not just the UID) and then using *procfs* to navigate through the process ancestors. Nevertheless, it's possible to design a simpler mechanism independent of *procfs*, just using Unix sockets: the session's parent process creates a Unix socket (that will be then inherited by its descendants) using *socketpair* and then it will launch the *xos_credstored* daemon. Every time a child needs to authenticate against the session creator, it will establish a connection via Unix socket with the *xos_credstored* daemon, transferring the inherited handle as a proof of being a descendant of the session creator. The daemon will verify that the PID of the handle creator matches up its own PPID, using *getsockopt* and *SOL_PEERCRED*.

As the use of *SOL_PEERCRED* is Linux specific, a more generic solution, compatible with any POSIX.1-2001 system is based on the use of a cookie. Anyway, the idea is not transferring directly the cookie copying it to an environment variable, but using a pair of Unix sockets created with *socketpair*: the parent process will write the cookie to the first socket; the children will inherit the second socket, from where they could read the cookie (they should use *recv* and the flag *MSG_PEEK* in order to not limit the access just to the first process accessing the socket).

How secure is the credential storage in the memory of a daemon being executed in user space in relation to LKRS that stores it in kernel space? The implementation in user space could offer a sufficient level of security when adopting some adequate precautions. For example, *mlock* must be used to prevent the memory being copied in the swap space on disk; the buffers used by the credential must be overwritten to prevent them remaining in not initialized RAM memory that could be assigned to a different process; the bit `set-group-ID` of the executable program may be activated to avoid a process to be "ptraced".

In any case, the native security offered by Unix is not oriented to protect from malicious applications running with the same UID, but against processes with a different UID. For the particular case of the session-scope of *credstore*, there could be attacks planned to take control over the processes of the session with access to the *credstore*, immediately before or after obtaining the credential and then, for a full protection, it's recommendable to isolate the applications executing them with different UIDs or using a *MAC* (Mandatory Access Control) implementation as SELinux[15] or Smack[16]. However, using *MAC* requires special kernel support and implies a more complex configuration.

The credential store library could be extended with additional implementations and even with new semantics. For example, a concrete implementation could be based on a proxy-certificate system instead of simply caching the credential. This way, when storing the credential, the certificate and the user's private key will be stored; but when reading the credential, a new proxy, specific for the application and with a shorter time-validity, will be generated and the application should read the user's certificate, the proxy certificate and the proxy's private key (but not the user private key).

In the future, this library could be used to establish further access and auditory restrictions, or even to make the credentials become read-only.

4.2 Credential Agent Library

The credential agent library is responsible for obtaining the credential through the invocation of two modules (named *credagent* and *creduiagent*), which are specified in a configuration file passed as parameter. These modules are dynamic objects (`.so`) under `/lib/security`, like the PAM modules.

The *credagent* module, in charge of obtaining a credential, uses the credential agent library to read parameters stored in the configuration file, like for example the name of the authentication server to obtain the credential.

The *credagent* module provides also an API for interaction with the user, done through the *creduiagent* module set in the configuration file. The *creuidagent* modules should implement a standard API to request the user and/or password, to ask for user's confirmation, to show a message, etc. The *creduiagent* could be adapted to the particular characteristics of the user's interface and even it could be used as a proxy to redirect the request to a remote user, for example to a PC administrator when it's needed a password and the device is a router, or to an adult when the device's user is a child and we want to implement some kind of parental control.

Using *credagent* modules, complex mechanisms to interact with remote servers while caching the encrypted credential on disk could be implemented, but also very simple ones: for example, storing the credential inside the configuration file and just requesting the user's permission to use it. This is not a security flaw, because configuration file is under a protected directory, preventing the access by any malicious application. *Runcredagent* can read the configuration file because it has the *setgid* bit active.

The previous example could present some vulnerabilities if the *creduiagent* uses an X-Window interface to interact with the user. A malicious application could modify the value of the `DISPLAY` variable to redirect the user confirmation's request to a remote terminal or even to manipulate the window to be shown (X-Window was not designed to prevent a *malware* affecting other applications sharing the same environment, but fortunately this is changing thanks to initiatives like SELinux [17]). The *creduiagent* module, included in XtreemOS-MD as reference implementation, provides some parameter to try to avoid this kind of attacks (the value of `DISPLAY` can be fixed for instance), but also it would always be possible to write a *creduiagent* not based on X-Window system.

4.3 Credential Getter Library Transparent Invocation

Even if the API of the credential getter library is extremely simple, the applications should be modified in order to use it, and in case of programs written in Java or Python for example, an adapter must be written in order to invoke the library, which is written in C language. However, this adapter program could be as simple as a generic program to write the credential to the standard output.

For those applications that read the credential from a file with a configurable path, a possible solution to not modify their source code consists of the interception of the system function library "open", modifying the function to invoke

the credential getter library and returning a pipe handler to read the credential. This solution, based on overwriting the system function library, is implemented in Xtreemos-MD by the library *libxos_wrapopen*, that offers additional functionality like separate extraction of certificate and private key (through special virtual paths) or a sort of auto-mounting system, achieved with an automatic invocation of the credential getter library when accessing a concrete folder.

This mechanism is inspired and quite similar to "kerberization". It requires linking each application willing to use the credential getter library with the *libxos_wrapopen* library. This means that, even not needing to modify the source code, it's necessary to access the object code files (`.o`), in order to perform the relinking. An alternative that does not require relinking consists of using the environment library `LD_PRELOAD`, but this will not work with applications that are `set-euid` or `set-egid`.

An alternative method that does not require a "kerberization-like" process, is based on the use of FUSE (File System in User Space [18]). Instead of intercepting the "open" function library, a FUSE file system would be mounted, so that in order to read the credential, an specific path under the mounting point will be used and the FUSE daemon will be in charge of invoking the credential getter library in a transparent way.

FUSE project is Linux specific, but there are similar software for other operative systems as NetBSD [19], MacOS X [20] and MS Windows [21].

5 SSO System Integration with XtreemOS

The SSO system of XtreemOS-MD was designed to be reused in other projects, but of course, its main goal is XtreemOS. Let's see the integration of this SSO system with XtreemOS.

XtreemOS security is based on the fact that the users and the resources to which they can access are all members of a Virtual Organization (VO) [22]. The VO membership is proved by means of private-public key mechanisms.

To obtain the private-public key pair that will allow the access to the Grid, the clients generate the private key and obtain the associated certificate from a Credential Distribution Authority (CDA) server, which performs the role of a Certification Authority (CA) for the users in the VO. Currently, the CDA server authenticates the user checking a password, and verifies that the user is really registered in the VO. In the future, additional authentication mechanisms (not based on passwords) will be supported.

XtreemOS-MD SSO considers as XtreemOS credential the combination of the private key and the certificate (in PEM format). It provides a *credagent* module (known as "cdaclient") for generating the private key and obtaining the certificate from the CDA server. The possible interaction with the user, is delegated to the *creduiagent* module established in the configuration file.

The credential obtained is cached during the session, but if the mobile device is switched off or restarted, the credential will be lost and a new process of private-key generation and certificate retrieval from the CDA server will be needed. This

could be avoided modifying some parameters in the configuration file, in order to permit credential caching in disk. While the credential cached in memory is ready for use in any application with access permission to the *credstore*, the credential cached in disk is encrypted with the password and stored in a protected folder, so the user's application cannot use it without user knowledge.

The "cdaclient" module provided also supports the use of a proxy (named as *CDAProxy* inside the XtreemOS project) to obtain the credential. This way, the module will authenticate against the *CDAProxy* instead of the CDA server. *CDAProxy* offers the following features

- If the *CDAProxy* is executed in a user's PC, the personal credential obtained from the CDA could be reused in all user devices. Other benefit is that the *CDAProxy* could be used for the persistent credential caching, instead of caching in the mobile device disk. On the other hand, the *CDAProxy* could offer the users additional authentication mechanisms (once the CDA server will support them) that could not be supported by the mobile devices, like for example the use of smartcards.
- A smartphone may delegate the private-key generation in the proxy. This is a really heavy and resource-consuming operation.
- Executing the *CDAProxy* in the PC of a organization (a company for example), the organization could control the user access instead of delegating it to the software infrastructure of the VO, so that their own policies and authentication and auditing mechanisms could be used. Additionally, this will allow the integration of the specific organization's SSO system with the SSO system for accessing the Grid.
- As the mobile devices' interfaces are in general uncomfortable for introducing long passwords, a PIN system could be implemented. The user will just need to introduce a PIN for the authentication against the *CDAProxy*, and the latter will be the one using the full password to retrieve the credentials (as usual, a mechanism to limit the number of failed attempts to introduce the correct PIN should be implemented, in order to not compromise the security).

The concrete implementation of the *runcredagent* program for XtreemOS is called *startxtreemos*. This program not only obtains the credential, but it's also in charge of every further action needed in order to let the applications use the Grid services. More concretely, it mounts the XtreemOS file system (XtreemFS) and creates the configuration needed by the Application Execution Management (AEM), the XtreemOS service for job management [7].

It's worth noting that *startxtreemos* mounts the XtreemFS even if the users are not allowed to do it manually, not being a member of the group with access to `/dev/fuse` (normally called "fuse" group). This way, an additional security level is offered, not needing to open the access to the FUSE system to every user or to add the users to the "fuse" group. To implement this functionality, *startxtreemos* uses a `set-euid` launcher that calls `setgroups`.

6 Conclusions and Future Work

XtreemOS-MD SSO is a security solution that, even being especially designed for mobile devices accessing the Grid, may be easily reused in any other project. The solution provided is modular, configurable and it's more portable than LKRS, whose philosophy has been imitated, but without the need of a specific kernel support. On the other hand, it's quite easy to integrate XtreemOS-MD SSO with the applications, not requiring even their modification in some cases. The architecture of XtreemOS-MD SSO is very simple in comparison, for example, with XSSO [23].

There are other solutions like for example Liberty Alliance ID-FF[24], which provides SSO support but also other features like identity federation and enjoys a strong industry acceptance. However, XtreemOS-MD SSO strengths are related to its simplicity and to the fact of being technology-agnostic and avoiding the modification of legacy applications.

XtreemOS-MD SSO is one important piece in order to implement the secure and easy access to the Grid philosophy promoted by XtreemOS (and concretely by XtreemOS-MD): any application could include XtreemOS-MD as a dependency, so that a user could use the Grid "out of the box" just specifying the VO user assigned to him.

A first version of the XtreemOS-MD SSO, is included in XtreemOS-MD release 1.0, available for Nokia N8x0 devices. XtreemOS-MD can be installed as a normal application, not needing the installation of a new kernel (even if LKRS is not supported by the system). The credential store library is also used by XtreeemOS software (the version for PCs).

XtreemOS-MD SSO is still evolving; some of the improvements have been already anticipated across this article, like the authentication against *CDAProxy* just using a PIN, but there could be additional features, some of them currently under development, like:

- Support for additional platforms, like OpenMoko [25], Maemo5 [26], and even PCs and netbooks with Ubuntu packaging.
- Integration of PAM modules in the *CDAProxy*, in order to support additional authentication mechanisms (like "Bluetooth pairing" for example) and facilitate the integration with existing organizations' authentication systems.
- New *credagent* and *creduiagent* modules, specially for integration with web services and Jabber.
- Use of the API offered by the credential store library, in order to let *startxtreemos* program to set a credential caching-timeout, never caching an expired certificate.
- New functionality to permit a user to discard the cached credential and to obtain a new one.
- Additional group control access to a user's *credstore* (to limit which application would have access to the credential)

The source code of XtreemOS-MD SSO implementation is available under Free/Open Source licenses.

References

1. Pashalidis, A., Mitchell, C.J.: A taxonomy of single sign-on systems. LNCS. Springer, Heidelberg (2003)
2. Kupczyk, M., Lichwala, R., Meyer, N., Palak, B., Plociennik, M., Wolniewicz, P.: Mobile Work Environment for Grid Users. In: Fernández Rivera, F., Bubak, M., Gómez Tato, A., Doallo, R. (eds.) Across Grids 2003. LNCS, vol. 2970, pp. 132–138. Springer, Heidelberg (2003)
3. Neuman, B.C., Ts'o, T.: Kerberos: an authentication service for computer networks. IEEE Communications Magazine 32(9), 33–38 (1994)
4. Globus Security Key Concepts, `http://www.globus.org/toolkit/docs/latest-stable/security/key/`
5. Coppola, M., Jégou, Y., Matthews, B., Morin, M., Prieto, L.P., Sánchez, O.D., Yang, E., Yu, H.: Virtual Organization Support within a Grid-Wide Operating System. IEEE Internet Computing 12(2), 20–28 (2008)
6. Samar, V.: Unified login with pluggable authentication modules (PAM). In: Proceedings of the 3rd ACM Conference on Computer and Communications Security, p. 10 (1996)
7. Cortes, T., et al.: XtreemOS: a Vision for a Grid Operating System (2008), `http://www.xtreemos.eu/publications/techreports/xtreemos-visionpaper-1.pdf`
8. De Clercq, J.: Single Sign-On Architectures. In: Davida, G.I., Frankel, Y., Rees, O. (eds.) InfraSec 2002. LNCS, vol. 2437, pp. 40–58. Springer, Heidelberg (2002)
9. Saint-Andre, P., et al.: Extensible messaging and presence protocol (XMPP): Core. Technical Report, RFC 3920, Internet Engineering Task Force (2004)
10. Myers, J.: Simple authentication and security layer (SASL). Technical report, RFC 2222, Internet Engineering Task Force (2007)
11. Nadalin, A., Kaler, K., Monzillo, R., Hallam-Baker, P.: Web Services Security SOAP Message Security 1.1. OASIS Standard Specification (2006)
12. Simon, D., Aboba, B., Hurst, R.: The EAP-TLS Authentication Protocol. Technical report, RFC 5216, Internet Engineering Task Force (2008)
13. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.1. Technical report, RFC 5216, Internet Engineering Task Force (2008)
14. Linux Key Retention System, `http://kernel.org/doc/Documentation/keys.txt`
15. SELinux: Security-Enhanced Linux, `http://www.nsa.gov/research/selinux/`
16. Schaufler, C.: Smack in Embedded Computing. In: Ottawa Linux Symposium (2008)
17. Kilpatrick, D., Salamon, D., Vance, C.: Securing the X Window system with SELinux. NAI Labs, Report #03-006 (2003)
18. FUSE: File system in User Space project, `http://fuse.sourceforge.net/`
19. Kantee, A., Crooks, A.: ReFUSE: Userspace FUSE Reimplementation Using puffs. In: Proceedings of the 6th European BSD Conference (2007)
20. MACFUSE: Fuse for MacOS X, `http://code.google.com/p/macfuse/`
21. DOKAN: User Mode FileSystem for Windows, `http://dokan-dev.net/en/`
22. STFC: Fourth Specification, Design and Architecture of the Security and VO Management Services, `http://www.xtreemos.eu/publications/project-deliverables/d3-5-13.pdf`
23. XSSO Architecture, `http://www.opengroup.org/onlinepubs/008329799/chap3.htm`
24. Liberty Alliance, `http://projectliberty.org/`
25. OpenMoko project, `http://www.openmoko.org`
26. Nokia: Introducing Maemo5: The software behind your computing mobile, `http://maemo.nokia.com/`