Single Sign-On for Java Web Start Applications Using MyProxy

Terry Fleury
National Center for Supercomputing Applications
1205 W. Clark St.
Urbana, IL 61801
001-217-333-0231
tfleury@ncsa.uiuc.edu

Jim Basney
National Center for Supercomputing Applications
1205 W. Clark St.
Urbana, IL 61801
001-217-244-1954
jbasney@ncsa.uiuc.edu

Von Welch
National Center for Supercomputing Applications
1205 W. Clark St.
Urbana, IL 61801
001-217-265-7139
vwelch@ncsa.uiuc.edu

ABSTRACT
Single sign-on is critical for the usability of distributed systems. While there are several authentication mechanisms which support single sign-on (e.g. Kerberos and X.509), it may be difficult to modify a particular legacy application to utilize an authentication scheme other than username/password. A simple solution for single sign-on involves transmitting a user's password over the network. However, it is undesirable to expose a user's private password in an insecure environment. This paper describes our effort to create "session passwords" which are short-lived passwords transmitted in lieu of a user's private password. Our implementation utilizes the MyProxy X.509 credential service as an authentication service. We demonstrate our solution in the MAEviz application portal, a Java Web Start application for earthquake risk management and analysis.

Categories and Subject Descriptors
H.3.5 [**Online Information Services**]: Web-based Services – *Java Web Start*; K.6.5 [**Security and Protection**]: Authentication – *X.509 credentials*; H.3.4 [**Systems and Software**]: Distributed Systems – *grid computing*.

General Terms
Design, Security.

Keywords
Session passwords, single sign-on, grid portals.

1. INTRODUCTION
Single sign-on (SSO) is the ability to allow multiple actions to take place on behalf of a user, without requiring multiple authentications by that user. The two main aspects of SSO are (1) the ability for a user to make multiple requests to possibly disparate services and (2) the delegation of rights empowering services to make requests of other services on behalf of the user. SSO is typically accomplished by the creation of a short-lived cryptographic token based on initial manual authentication by the user (e.g. entering a password). This token can be utilized to authenticate subsequent actions. Creating instances of such tokens allows for delegation to services on remote systems. SSO is critical to the usability of distributed systems and is supported by many security technologies such as Kerberos, X.509, and SAML.

Retrofitting legacy services which use username/password authentication to support SSO can be difficult because protocol changes are typically required. Changes in protocol may dictate major modifications to existing client and server software. This can introduce compatibility problems. In practice, implementing SSO with such services often involves storing the user's password locally to re-authenticate the user. For example, even though web browsers and servers support client-side X.509 authentication, it is rarely used since the vast majority of web authentication is accomplished with username and password. When delegation is required, the user's password is typically sent to the remote service so that it can impersonate the user. This behavior is insecure but necessitated by the limitations of the software involved.

In this paper, we describe a method we have developed which supports SSO for username/password authentication-based services. We use short-lived, dynamically created passwords that allow repeated authentication of a user, and even delegation, without putting the user's original long-lived password at risk. We refer to these short-lived passwords as *session passwords*, a term we borrow from session keys in cryptographic protocols. These session passwords are associated with an X.509 credential and stored on a MyProxy server, which is utilized as an authentication service.

Our motivating use case is a Java Web Start (JWS) application. JWS offers a convenient way for users to launch Java applications from a portal. Java code can be downloaded automatically to the user's local system and executed (possibly in a restricted manner). A typical JWS application is launched when a user visits a web site and makes a request. A file is then downloaded to the user's machine. If properly configured, a local Java Runtime Environment takes over and downloads the JWS application for execution on the user's machine. While JWS applications are powerful, once launched they are separate from the original web browser. This is especially troublesome when the JWS application requires authentication for access to certain web resources.

SWS'06, November 3, 2006, Alexandria, Virginia, USA.

Consider the following scenario. A user authenticates to a web server through the browser. The user requests a JWS application from within the authenticated session. The JWS application launches but without any notion of the authenticated session, thus requiring the user to somehow re-authenticate to certain web resources from within the JWS application. With our solution, a session password is created and passed to the JWS application when it is launched. In this manner, the JWS application can authenticate to the original web site and even other services. The lifetime of the session password is short so as to limit its use. Additionally, each session password is unique which implies that the actions of its recipients can be audited for misbehavior.

We start with a discussion of existing security technologies and their shortcomings in addressing this problem. We then provide a brief overview of technologies on which our solution is based. In section 3 we mention our motivating use case taken from a real-world deployment. We then present our solution in detail in section 4. Plans for future work and references follow.

2. RELATED TECHNOLOGIES AND PRIOR WORK

In this section we briefly review technologies related to our solution, and explain why they fail to solve our problem. We also present some background on the technologies utilized in our solution.

2.1 Java Web Start

Java Web Start[9] (JWS) is a framework developed by Sun Microsystems that enables a user to launch a Java application by clicking on a link in a browser. JWS serves as the motivating use case for our solution. A JWS application does not run as an applet in the web browser but rather as a full application on a user's desktop. Hence, JWS applications can be granted greater access to local system resources, such as read/write access to the file system. A JWS application can be written for a particular version of the Java Runtime Environment (JRE), which is then automatically downloaded to the user's machine, thus easing distribution of the software to various clients. Once the JWS application is installed on the user's machine, the user can create a shortcut to the application and launch the application again without having to click on a browser link. Users do not need to manually update the application since it is transparently updated from the web each time it is executed.

Closely related to Java Web Start is the Java Network Launching Protocol[6] (JNLP). JNLP defines an XML file format which specifies how a JWS application is started. The JNLP file contains information such as the (remote) location of the `jar` package file, the name of the application's main class, and any parameters required for program execution. When a user clicks on a link for a JWS application, this JNLP XML file is downloaded by the browser and (if properly configured) is passed to a JRE, which in turn downloads the application and runs it.

2.2 Existing Secure Single Sign-On Solutions

Our main goal in this project is to provide the user with a single sign-on (SSO) experience, i.e. the user authenticates to a portal a single time regardless of the number and types of systems involved. However, we also require that whatever occurs behind-the-scenes protects the user's main means of authentication, i.e. his private password. Several existing protocols satisfy these conditions, but each has limitations that make it unsuitable for our motivating use case of a Java Web Start (JWS) application launched from a portal.

First there is the category of "custom protocols" which provide SSO. In order to utilize a protocol in this category, all processes must understand the protocol. For example, Kerberos[11] and X.509[7],[17] have cryptographic software tokens that serve to authenticate the user multiple times based on a single manual authentication. These protocols also allow for the delegation of tokens to remote services. In order to use a protocol of this type, all processes involved must understand and utilize the technology's protocol. This is not feasible in our use case. For example, implementing Kerberos with Java's Generic Security Services API[8] would not work in our scenario as both the grid portal, which is possibly non-Java, and the JWS application would need to be able to handle these tokens, and the tokens would have to be passed via an XML file.

Session cookies constitute another category of providing SSO, in this case between a web browser and a web server. One commonly employed solution is the inclusion of a `JSESSIONID`[13] within a dynamically created JNLP file. This allows the resulting JWS application to "inherit" the browser security context from the web server. For any web resources required by the JWS application, the `JSESSIONID` is appended to the URL of the request. There are two problems with this approach for solving our use case. The first problem is that the JWS application may require access to the web resources after the session expires, causing future requests of the secured resources to fail. To reestablish the session, the user would have to re-authenticate in the web browser and download a new JNLP file with a new `JSESSIONID`. The second problem is that the security context provided by the `JSESSIONID` is valid only in the web server with which it was originally established by the browser. Additionally, applications are restricted to the usage of underlying protocols which support session cookies.

Then there is the category of "browser-based" SSO. In order to utilize protocols in this category, all aspects of the system must occur within a web browser. Solutions in this category include Microsoft's Passport[10], Pubcookie[14], Shibboleth[16], and Liberty[3]. Since the JWS application is not browser-based (and has limited abilities to mimic a full-featured web browser) these protocols are not suitable for our architecture.

2.3 MyProxy

MyProxy[2],[12] is an X.509 credential management system. As its name suggests, it can properly handle both X.509 end entity certificates[7] and X.509 proxy certificates[17]. MyProxy was originally created to allow for delegation of X.509 credentials from a web browser to a grid portal. A grid portal is a web server that allows the user to interact with grid infrastructure utilizing X.509 certificates for authentication. MyProxy enables a user to delegate an X.509 credential to a MyProxy server and associate a username and password with that credential. The username and password can then be passed to a grid portal. The grid portal contacts the MyProxy server, presents the username and password, and obtains a delegated X.509 credential to execute grid services on behalf of the user.

Since its inception, MyProxy has had a number of features added to it. Of relevance to our work presented here are the abilities for (1) a user to associate multiple credentials with a single username, and (2) MyProxy to act as an online certificate authority (CA). When MyProxy associates multiple credentials with a given username, each credential has its own password. These stored credentials can have different lifetimes or other constraints. A user selects which credential to delegate to a grid portal by sending the associated session password.

If a username and password do not match any stored credentials, MyProxy has the ability to act as a CA. If the username and password authenticate the user via an external authentication server, the MyProxy CA can dynamically generate X.509 credentials for that user. Users can be authenticated by several external authentication methods such as Pluggable Authentication Modules (PAM), Simple Authentication and Security Layer (SAML), Kerberos, One Time Passwords (OTP), and LDAP.

3. MOTIVATING USE CASE: MAEVIZ APPLICATION PORTAL

MAEviz[1] is a visualization application developed for the Mid-America Earthquake (MAE) Center. The MAE Center provides on-line information about the potential effects from earthquakes striking Mid-America. Through their Access2 program, industry and government organizations can access the Center's latest research results via published reports, technical papers, and software and data. The main software project of the MAE Center is MAEviz, a visualization utility for Consequence-based Risk Management (CRM). CRM incorporates identification of uncertainty in all components of seismic risk modeling and quantifies the risk in terms applicable to societal systems. For example, if an earthquake occurs near a large city, what is the possibility of bridges and roads being destroyed which thus affects how well emergency personnel can respond? This information can be used to develop contingency plans for various earthquake disaster levels.

The MAE Center portal, MAEport[5], is based on Sakai[4], a community source development effort to provide a Collaboration and Learning Environment (CLE) for higher education. It consists of an application framework and associated Content Management System (CMS) tools designed to work together to support course management and collaborative research efforts. The MAE Center chose Sakai for its pluggable architecture and configurable web interface.

Figure 1 shows a high-level overview of the system. A user logs in to MAEport (the Application Server in the figure) and is presented with a customized Sakai logon page. From there, the user can select the MAEviz page which brings up a list of scenarios available in the repository. Upon selecting a scenario, the MAEviz JWS application is downloaded and executed with the selected scenario data set. Behind the scenes, the Sakai portal connects to a WebDAV server (the Data Server in the figure) to generate a listing of scenarios available to the user. When the MAEviz JWS application is launched, it too connects to the WebDAV server, but this time to download all data necessary to generate a graphical rendering of the data set.

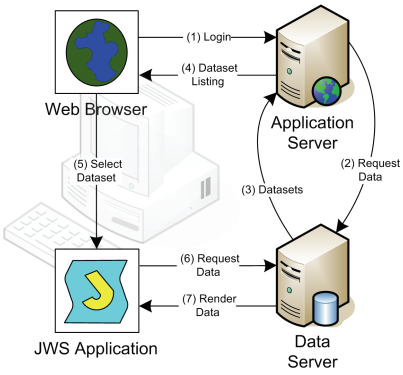

Figure 1. Grid Portal + Java Web Start Application

Access to data on the WebDAV server must be restricted as some of the data is of a sensitive nature. Both the Sakai portal server and the MAEviz JWS application need to authenticate with the WebDAV server. Since the user logs in to MAEport, the username and password are already available to the Sakai portal for authentication to the WebDAV server. However, in order for the MAEviz JWS application to authenticate the user, either the username and password need to be sent in the JNLP file, or the user must re-enter his username and password. The former is undesirable since the JNLP file can remain on the user's local file system after the JWS application exits and thus could be accessed by other users of that machine, possibly giving them the user's password. The latter is undesirable since it would detract from the user's experience.

4. OUR SOLUTION

As described in the previous section, our motivation is to allow for single sign-on (SSO) of a user. Initial authentication occurs in a web browser. Subsequent authentication occurs in a Java Web Start (JWS) application. Authentication is required for access to the initial web portal and other data services. We use MyProxy as an authentication service for all other services involved. The services authenticate a user by passing username and password to a MyProxy server. MyProxy indicates successful authentication by responding with a positive answer and/or the X.509 credential for that username/password.

At this point, SSO could be achieved by passing the user's secret password from the web portal to the JWS application. However this is undesirable from a security perspective since it risks exposure of the long-lived password. Our solution is to create a *session password*. This short-lived password can be passed to the JWS application and utilized to contact both the original portal and any other service which uses MyProxy as an authentication service. The lifetime of the session password is set to the expected duration of the user's actual session. Upon expiration of the session password, it can no longer be utilized to authenticate the user. The short lifetime mitigates risk of password theft.

The generation and validation of a short-lived session password is performed by a client and a MyProxy server. Figure 2 shows how this is accomplished.

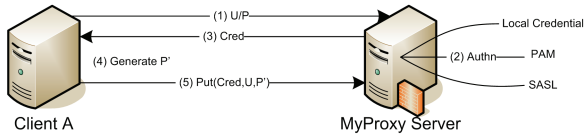

Figure 2. Creating a short-lived "session password" with MyProxy

The details of the steps illustrated in Figure 2 are as follows.

(1) Client A wants to authenticate a Username using a given Password. It sends U/P to the MyProxy Server.

(2) MyProxy uses various mechanisms to see if U/P authenticates. For example, it can check any locally stored credentials for matching U/P, or it can authenticate to external sources via a Pluggable Authentication Module (PAM) or Simple Authentication and Security Layer (SASL).

(3) If successfully authenticated, the MyProxy Server returns a new Credential.

(4) Client A generates a new random (Session) Password (P') which will be used the next time a process wants to authenticate the Username. Note that while the client generates the password, the MyProxy server can enforce policies with respect to that password, e.g. minimum string length.

(5) Client A stores the Credential on the MyProxy Server under U/P'. The client specifies the lifetime of this credential, which is then the lifetime of the Session Password. The MyProxy server may impose limits on this lifetime.

At the end of this process, Client A has a session password P' which can be sent along with the Username to another machine/process. That machine/process would authenticate the Username using P' as the Password. This is illustrated in Figure 3.

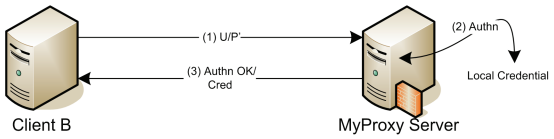

Figure 3. Using a short-lived Session Password with MyProxy

The details of the steps illustrated in Figure 3 are as follows.

(1) Client B has been given a short-lived Session Password P' and wants to authenticate a Username. It sends U/P' to the MyProxy Server.

(2) MyProxy checks all local credentials stored under the Username for one with a matching Session Password P'.

(3) Upon finding a matching stored credential, the MyProxy Server returns an "authentication okay" message and/or the stored credential to Client B.

One important thing to realize is that the MyProxy server behaves identically whether it receives the user's "original" password as in Figure 2 or the session password as in Figure 3. In either case, the MyProxy server will authenticate the password and return a credential for the user (or a failure message if appropriate). It is then up to the client machine to take the extra steps of generating the session password and storing the credential on the MyProxy server, as shown in steps (4) and (5) of Figure 2.

The remainder of this section discusses in detail the modifications made to support session passwords.

4.1 MyProxy Server Modifications to Support Session Passwords

When a client stores a credential on a MyProxy server, the credential written to the file system is given the name `<username>-<credname>.creds` where the username must be specified by the client and the credname is optional. When the credname is absent, the credential is the "default" stored credential for that username. A client may store any number of credentials under a particular username, but each credential other than the "default" credential must be given a unique credname. In order to retrieve a credential, the client must again specify a credname. In a typical MyProxy server installation, if no credname is given then only the "default" credential can be accessed.

We wanted the ability to store any number of credentials under a particular username and later check all of those credentials for matching criteria specified for access by a client. In this manner, any number of session passwords could be created for a given username and associated credential, and existing systems based on username/password authentication would not need to be modified to track an additional credname parameter. So we added a new server run-time configuration option `check_multiple_credentials`. This option affects the way credentials are accessed by a client when no credname is specified during access attempts. The option directs the MyProxy server to check all stored credentials for the username in question. It returns the first matching credential. If it cannot find a matching credential on the file system, it will fall back to any other authentication mechanism for which the MyProxy server is configured, e.g. Kerberos login via PAM.

The `check_multiple_credentials` configuration option is used in the following scenario. A client stores any number of credentials for a particular username on the MyProxy server, each with a different credname, a random password, and a short lifetime. The random passwords can be utilized as short-lived session passwords and passed to other clients/processes requiring authentication of that username. The requesting process needs only the username and the random password. When no credname is specified during an access attempt, the MyProxy server checks all credentials stored under that username for one with a matching password. The credential could then be deleted from the MyProxy storage or allowed to expire after the short lifetime, thus rendering the password ineffective for future authentication attempts.

4.2 Deployment in MAEviz

To reiterate, our goal is to give the user a single sign-on experience in MAEviz, i.e. the username and password should be requested of the user only a single time. Referring to Figure 1, when the user indicates that he wants to see a list of datasets available to him for rendering in MAEviz, the Data Server is contacted. The Data Server has its own user authentication step. For the sake of simplicity, we will assume that the same username and password utilized to access the Application Server can be

utilized to access the Data Server. So far, the user has not had to re-enter his username and password. When the Data Server returns the available datasets, the Application Server generates a listing and presents it to the user. The user selects one dataset for visualization.

At this point, the web browser launches a Java Web Start application on the user's machine. Since the JWS application requires access to the Data Server, it needs a username and password. If we do not want to require the user to enter his username and password a second time, that information needs to be passed from the web browser to the JWS application. This can be accomplished by putting the username and password in the JNLP file which launches the JWS application. This is a security risk since the JNLP file is not deleted when the user exits the JWS application. Our solution is to put a short-lived session password in the JNLP file in lieu of the user's original password. For this we utilize a MyProxy server as an authentication service.

With the addition of the MyProxy Server, user authentication is now ultimately handled at a single location, the MyProxy server. Before the Application Server requests data from the Data Server, it authenticates with the MyProxy Server. Since the Application Server knows that the Data Server will also require user authentication, it creates a session password. The Data Server uses this session password to authenticate with the MyProxy server before returning datasets.

Figure 4 shows the entire scenario with detailed steps indicating when the session password is created and passed between processes.

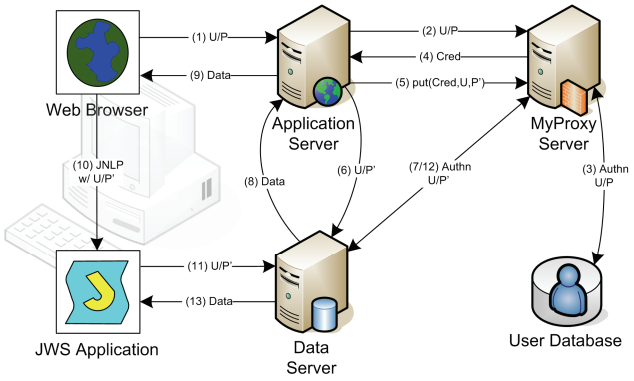

Figure 4. Single sign-on for grid portal and JWS Application using MyProxy authentication

The details of the steps illustrated in Figure 4 are as follows.

Step 1: From a web browser, the user connects to an Application Server web portal and enters his Username and Password (U/P).

Steps 2-5: The Application Server sends the Username and Password to the MyProxy server for authentication and then generates the random Session Password (P') as illustrated in Figure 2.

Step 6: So far we have simply set the groundwork for authenticating a username/password using MyProxy. Now the Application Server needs to get the data for the application. At this stage the data is high-level data such as a listing of all scenarios available to the user on the Data Server. The Username and new Session Password are sent to the Data Server. The Data Server needs to return data for that Username, but only after proper authentication.

Step 7: The Username and Session Password are sent to the MyProxy Server for authentication as illustrated in Figure 3.

Step 8: Data is returned to the Application Server.

Step 9: The Application Server processes the data to a format suitable for display to the user, for example a listing of available data sets.

Step 10: The user selects a data set to view in more detail with a Java Web Start Application. A JNLP file containing the name of the selected data set, the Username, and the Session Password is sent from the Application Server which is downloaded by the Web Browser to launch JWS on the user's machine. There are two reasons for launching a separate application rather than running a Java applet in the Web Browser. First, the JWS Application is written as a "trusted" application so as to have full access to the local file system. Second, the user could launch the JWS Application at a later time (from the Java Web Start Cache Viewer for example) bypassing the need to select the dataset from a listing again.

Step 11: At this point the JWS Application is launched automatically via the JNLP file. The Username and Session Password contained in the JNLP file are sent to the Data Server in order to retrieve the selected data set.

Step 12: As in step 7, the Username and Session Password are sent to the MyProxy Server for authentication.

Step 13: Finally, the data is returned to the JWS Application for display.

The JWS Application is written so that it can run in "standalone" mode as well. In this mode, the user starts the JWS Application from a shortcut on the desktop. The JWS Application sees that it was not launched from a browser and prompts the user for Username and Password. This is the same Username and Password that would normally be provided to the web portal. U/P is sent to the Data Server in step 11 above, and to the MyProxy server in step 12 above. In this case the password will not match one of temporary stored credentials, but rather the user's long term password in the external User Database.

5. SECURITY CONCERNS

One concern with our approach is that the user's web browser downloads the JNLP file to the local file system in the course of launching the JWS application. Neither the web browser nor Java Web Start makes any effort to ensure the privacy of the JNLP file or remove it when it is no longer needed. If the browser is running on a multi-user system and the environment is not configured to protect the privacy of a user's files (e.g. their Unix `umask` setting is permissive), the session password could be compromised.

Currently, the only fix for this issue is user education. Code could be added to the JWS application to check for insecure settings, but

since the JNLP file exists on the file system before any such code is executed by the JWS application, the session password could still be compromised. To prevent the insecure writing of files by the browser, changes need to be made at the browser level.

The question has also arisen as to whether the user having a browser session connected to a malicious site concurrently with the launch of the JWS application could be a security concern. In this case, the malicious site would need to intercept the JNLP file upon transfer. We are not aware of any modern browser vulnerabilities that would allow the hijacking of a download in progress. If the malicious site has access to files on the user's local file system, then the session password could be compromised. But at that point, the user's entire system could be compromised.

6. FUTURE WORK

There are a couple of optimizations that will be made to the system. One concern is that the Data Server must contact the MyProxy Server for user authentication every time data is requested. If there are many repeated requests for access to data by the JWS Application, the application may bog down. Fortunately, the Data Server that we are using, Scientific Annotation Middleware[15] (SAM), has a mechanism for generating a cookie which can be used for quick authentication.

On the client side, as described in section 4, several steps must be taken by the client in order to (a) verify a given username and password and (b) generate a new session password. With modifications to the client API, a single method could be created which takes a username and password and returns a new random session password and optionally the associated credential. Behind the scenes, all of the intermediate steps in Figure 2 would take place, i.e. authenticating the username and password, creating a new credential using the MyProxy CA or returning the stored credential for that password, creating a random session password, and storing the credential under this new password. Providing a single API method encompassing these steps would reduce the possibility of a developer omitting a step.

7. CONCLUSION

The solution we have shown here was motivated by NCSA's need to develop a single sign-on (SSO) solution for several projects currently in development. As work on MyProxy has progressed over time, MyProxy has evolved into more than just a credential storage and retrieval solution. By using the MyProxy CA functionality in combination with the modifications we have presented here, we have created a system enabling secure SSO through the use of session passwords, even with legacy applications incapable of supporting advanced security mechanisms.

While this solution was developed originally for the MAEviz project, we are working with other projects in our organization in an attempt to generalize our SSO solution. By working with software at various levels of development (middleware, end-user, etc.) we hope to offer a general solution for the SSO problem utilizing MyProxy as well as other technologies in development at NCSA.

8. ACKNOWLEDGEMENTS

The National Center for Supercomputing Applications is funded by the National Science Foundation in the United States of America under Grant No. SCI-0438712. The Mid-America Earthquake Center is funded by the National Science Foundation in the United States of America under Grant No. EEC-9701785. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Several colleagues provided valuable input to this work, including Jim Myers and Kevin Price.

9. REFERENCES

[1] Automated Learning Group, NCSA, *MAEviz Introduction & Tutorial*, Sep. 2004. http://algdocs.ncsa.uiuc.edu/TU-20040901-1.pdf

[2] Basney, J., Humphrey, M., and Welch, V., The MyProxy Online Credential Repository, *Software: Practice and Experience*, Volume 35, Issue 9, July 2005, pp. 801-816.

[3] Cantor, S., Hodges, J., Kemp, J., and Thompson, P., *Liberty ID-FF Architecture Overview*, Version 1.2-errata-v1.0, Liberty Alliance Project Website, 2005. http://www.projectliberty.org/specs

[4] Counterman, C., Glenn, G., Gollub, R., Norton, M., Severance, C., Speelmon, L., Sakai Java Framework, Version 1.5, *Technical Report Sakai Project*, Mar. 5, 2005. http://www.sakaiproject.org/

[5] Elnashai, A.S., Director, MAE Center Launches New Website, *Inside MAE, Winter 2006*, Vol. 9, No. 1, 2006, p.6. http://mae.cee.uiuc.edu/

[6] Herrick, A., Java Network Launching Protocol & API Specification (JSR-00056), Java Cummunity Process Website, 2005. http://jcp.org/aboutJava/communityprocess/mrel/jsr056/index2.html

[7] Housley, R., Polk, W., Ford, W., Solo, D., Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, *Internet Engineering Task Force Request For Comments 3280*, IETF Website, 2002. http://www.ietf.org/rfc/rfc3280.txt

[8] Khan, F. Simplify Enterprise Java Authentication with Single Sign-on, IBM Website, Sep. 9, 2003. http://www-128.ibm.com/developerworks/java/library/j-gss-sso/

[9] Marinilli, M., *Java Deployment with JNLP and WebStart*, Sams Publishing, Indianapolis, IN, 2001.

[10] Microsoft Corp., *Microsoft .NET Passport Review Guide*, Jan. 2004. http://www.microsoft.com/net/services/passport/review_guide.asp

[11] Newman, B.C. and Ts'o, T., Kerberos: An Authentication Service for Computer Networks, *IEEE Communications*, 32(9):33-38, Sept. 1994.

[12] Novotny, J., Tuecke, S., and Welch, V., An Online Credential Repository for the Grid: MyProxy, *Proceedings of the Tenth International Symposium on High Performance Distributed Computing (HPDC-10)*, IEEE Press, August 2001. http://myproxy.ncsa.uiuc.edu/

[13] Osbaldeston, R. and Bauer, G., Unofficial Java Web Start/JNLP FAQ: http://lopica.sourceforge.net/faq.html

[14] Pubcookie Website: http://www.pubcookie.org/

[15] Schwidder, J., Talbott, T., Myers, J., Bootstrapping to a Semantic Grid, *Proceedings of the Semantic Infrastructure for Grid Computing Applications Workshop*, IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID), Cardiff, UK, May 9-12, 2005. http://www.scidac.org/SAM/fd

[16] Shibboleth Website: http://shibboleth.internet2.edu/

[17] Tuecke, S., Welch, V., Engert, D., Pearlman, L., Thompson, M., Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile, *Internet Engineering Task Force Request For Comments 3820*, IETF Website, 2004. http://www.ietf.org/rfc/rfc3820.txt