In: 2005 IEEE International Conference on Web Services (ICWS 2005), Orlando, FL, July 12-15, 2005.

CredEx: User-Centric Credential Management for Grid and Web Services

David Del Vecchio
Marty Humphrey
Department of Computer Science, University of Virginia
dad3e@cs.virginia.edu, humphrey@cs.virginia.edu

Jim Basney
National Center for Supercomputing Applications, University of Illinois
jbasney@ncsa.uiuc.edu

Nataraj Nagaratnam
Application & Integration Middleware Division, IBM
natarajn@us.ibm.com

Abstract

User authentication is a crucial security component for most computing systems. But since the security needs of different systems vary widely, authentication mechanisms are similarly diverse. In particular, independently-managed Web and Grid Services vary with regard to the type of security token (credential) used to prove user identity (username/password, X.509 signing, Kerberos, etc.). Forcing users to manage and present credentials manually for each service is tedious, error-prone and potentially insecure. In contrast, we present ***CredEx****, an open-source, standards-based Web Service that facilitates the secure storage of credentials and enables the dynamic exchange of different credential types using the WS-Trust token exchange protocol. With CredEx, a user can achieve single sign-on by acquiring a single (default) credential then dynamically exchanging that credential as needed for services that authenticate a different way. We describe the design and implementation of CredEx by focusing on its use in bridging password-based Web Services and PKI-based Grid Services, illustrating how interoperability between these realms can be based upon the WS-Security and WS-Trust specifications.*

1. Introduction

A major challenge in designing and implementing Web and Grid Services is being able to satisfy both usability and security needs. Consider the two extremes: a system that denies access to all users is perfectly secure, whereas a system devoid of any security mechanisms is easy for users to access and understand. Clearly, neither of these two configurations is particularly desirable. While users recognize the importance of security in accessing Grid and Web Services, they do not want the security mechanisms and policies of these systems to hinder their scientific explorations or business operations. With regard to authentication, users might understand that it is necessary to prove their identity (using security tokens or credentials). However, they want to access services without learning new security technologies or understanding the underlying cryptographic theory.

Unfortunately, the obvious solution: using a *single* credential for authentication across multiple domains is not necessarily practical, secure or even particularly desirable. Quite simply, different systems have different security needs. While Kerberos [21] might be appropriate for one domain, public-key infrastructure (PKI [14]) might be better suited for a second domain, and the convenience of passwords might be most useful for a third. Exacerbating this problem is that different credentials are often needed for performing different roles within a single domain (with accesses ideally using the least privileged credential). Even if it were possible to mandate a single authentication type (e.g. PKI from a single CA) across all domains, the risks associated with compromise of this single credential are overwhelming. For example, using the same password for an Internet banking system as for an online chat service is a generally regarded as a bad idea. As the need to access diverse systems grows, the need for multiple credentials will undoubtedly follow suit. Accommodating a diversity of modern and legacy security systems across multiple Web Service domains remains a fundamental challenge.

Accepting the need for multiple credentials, another possibility is to let users manually manage their security tokens. Users could acquire all possible credentials upfront for each session, then select and present the necessary credentials on a per-service basis. Even for especially skilled and vigilant users, this approach is tedious and insecure.

With this credential management nightmare in mind, we present CredEx, an open-source, standards-based Web Service that facilitates secure credential storage and dynamic, user-driven (or service-driven) exchange of different credential types. A service with this functionality has been identified as being critical to the success of the

security architecture for the Open Grid Services Architecture (OGSA) [24]. The design and implementation of CredEx is inspired by MyProxy [22], the critical differences being that CredEx supports more than just the password-to-Grid Credential (GSI) exchange and that CredEx is compliant with emerging Web Services security specifications and standards.

While CredEx supports several different use cases, the primary pattern we have focused on involves users first uploading longer-term credentials (say *pkiCred1* for security domain 1, *pkiCred2* for domain 2, *username/password* for domain 3, etc.). Then to start a new session, a user would acquire his or her default credential, in a convenient format acceptable for the majority of the user's service interactions. In order to engage services that do not recognize this default credential, the default credential is *exchanged* for a credential that is suitable for the service in question. This exchange can be engaged directly by the user or initiated by the target service – in the latter case, the service receives the unsupported credential and exchanges it for a credential that would allow the service to complete the request. The primary benefits of CredEx include:

- Support for single sign-on in the presence or requirement of multiple credentials and token types
- Support for least-privilege credentials, where the default credential is exchanged for the least-privileged credential needed to accomplish the task (reducing vulnerabilities if credentials are stolen)
- A standards-based implementation, using an open-source WS-Security [20] toolkit and the contribution of our own open source WS-Trust [2] implementation
- Interoperability, as shown through the successful interaction of our .NET and Java clients with our Java/Axis service and through the successful bridging of Web and Grid Services

To our knowledge, CredEx is the first freely-available, general-purpose credential storage and exchange service that solves these security token management problems. The rest of this paper will focus on CredEx. After first reviewing some previous efforts in this area (Section 2), we present the features and design for flexible credential storage and exchange (Section 3). Discussion of the token exchange protocols and their relation to WS-Security and WS-Trust (Section 4) will be followed by some details of the implementation (Section 5). In Section 6, we demonstrate how our credential exchanger is used to bridge security realms by fostering interoperability between traditional password-based Web Services and PKI-based Grid Services, then conclude with future directions for this work (Section 7).

2. Related Work

A number of important projects address the challenge of cross-system authentication. One prominent example is Microsoft's .Net Passport system [19]. Passport aims to be a single sign-on solution for the Web: users need only one set of credentials (id, key, etc.), centrally stored and password-accessible authentication to any Passport-enabled service. Passport brings with it all the challenges of a single, universal authentication mechanism already discussed. This may in part explain why, despite over 200 million registered Passport accounts, Passport-enabled websites are far less numerous and widespread adoption remains elusive. The Entrust TruePass portfolio [9] is another system that promises convenient web-based authentication, in this case using PKI-based Entrust Digital IDs and signatures for authentication. This system does support storage of multiple credentials per user, but also requires target websites to be TruePass enabled.

In contrast to these purportedly universal single sign-on solutions, many other efforts have focused instead on exchanging specific credential formats for use in particular applications. For instance, the Kerberized Certificate Authority (KCA) [28] supports exchange of a Kerberos ticket-granting ticket for an X.509 proxy certificate [26] used in grid authentication. The KCA is not a credential repository, but instead generates the proxy certificate on the fly, based on information in the supplied ticket-granting ticket. In contrast, MyProxy [22] is a true credential repository, also designed with grid authentication in mind. Here, the repository supports storage of multiple credentials (X.509 proxy certificates) per user and retrieval as needed using username and password. Thus, MyProxy allows for successful grid authentication from anywhere, at anytime including credential delegation to third parties. MyProxy does share some important similarities (multiple credentials/user, X.509 proxy certificate support) with the work described in this paper and indeed it was a key inspiration for CredEx. Another notable effort is the IETF Securely Available Credentials (SACred) project [10] which is attempting to design a standard protocol and framework for password-based credential storage and retrieval.

While it is clear how exchanging for different token types is useful for cross-domain authentication, exchanges involving a single type of token can be useful in their own right. The Password Safe [23] program is an example of a password database that allows users to store passwords for different accounts under a single master password for password-based single sign-on. Kerberos cross-realm authentication is another case of single-type credential exchange. Here, a Kerberos user with a ticket for the local realm can interact with the Key Distribution Center to exchange for tokens valid in remote realms.

Some other fairly ambitious projects that touch on such authentication challenges are really best classified as federated identity management systems. These include the Liberty Alliance Project [17] and Microsoft Trust Bridge [11]. The Liberty Alliance is a collection of over 150 companies and organizations intending to define standardized ways to managing web identities. Rather than providing one universally accepted notion of identity, the Liberty Identity Federation Framework [27] defines how various identity providers (which store information related to user identities) can be linked together to support single sign-on across multiple distinct services. These specifications build upon those for the Security Assertion Markup Language (SAML [15]) with the assumption that SAML assertions would be used for authentication. Currently, however, the Liberty specifications fall short of addressing support for diverse authentication token types or token exchange issues (a SAML-enabled version of CredEx could prove useful here). Microsoft's Trust Bridge is federated identity effort that builds on the WS-* set of Web Service protocols, in particular WS-Security [20] which defines XML packaging of security-related, WS-Federation [16] which includes protocols for mapping between identity providers and WS-Trust [2] which defines a security token exchange protocol. Unfortunately, details of this effort are hard to come by and the planned first release is still at least a year away. Shibboleth [7] a federation effort for the academic community hopes to enable access-controlled resource sharing across institutions. Shibboleth makes extensive use of SAML assertion tokens for single sign-on. Conceptually, the credential exchanger we developed might fit well in a federated identity management framework by providing an important token exchange mechanism for bridging various services.

Considering the variety of research into credential and identity management, it's fair to question the need for another credential exchanger. The primary limitation of previous efforts is a lack of interoperability and flexibility. Most existing credential exchange services only support a single direction of credential exchange (i.e. only password for certificate, not the reverse), and tend to be limited to specific application domains (e.g., grid services) using proprietary protocols on a single platform. In light of the ever-present diversity in the Web Services world, being able to exchange multiple credentials in a flexible, interoperable way is extremely important.

3. CredEx Features and Design

Our credential exchanger, CredEx, aims to bring flexibility to the authentication token exchange process through several key design features:

- Web-services based protocol built on open standards
- Support for multiple platforms and languages
- Support for multiple types of tokens and exchanges
- Central storage of multiple credentials (identities) per individual

By adopting a Web Service interface, we ensure that accessing the service is straightforward and consistent from a variety of client platforms. Building upon well-established protocols such as WS-Security and in particular the WS-Trust protocol for security token exchange via Web Services further maximizes the potential for interoperability. This approach facilitates client development since direct support can be inherited from commercial tooling.

Currently our credential service allows for two types of token exchanges: username/password for X.509 certificate and X.509 for username/password. Emerging credential bindings (like the Kerberos WS-Security binding [29] and the WS-Security SAML token profile [30]) In addition to token exchange, our system also provides a central storage location for user credentials. Individuals can securely store and manage multiple sets of

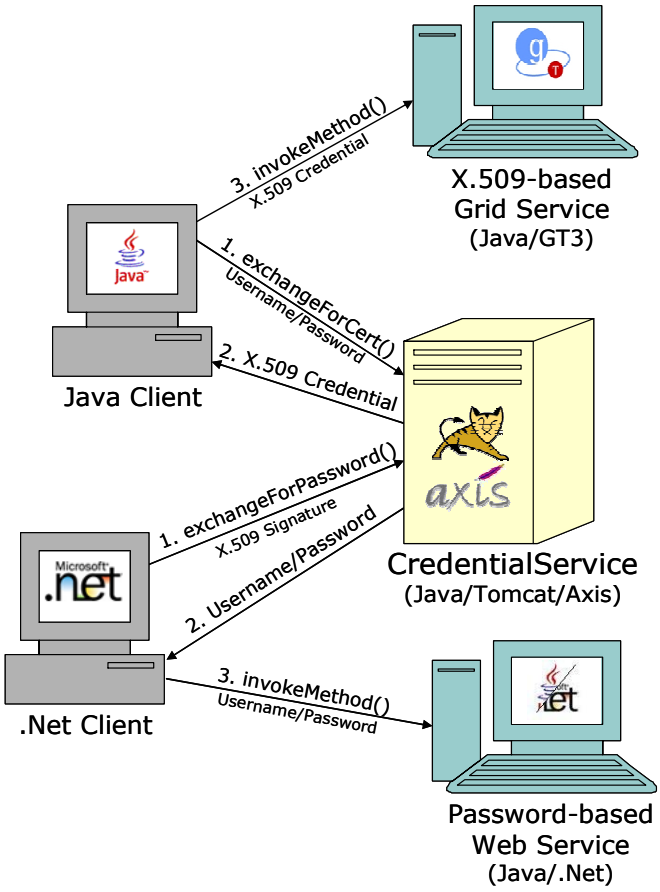

Figure 1: System overview, showing CredEx, a pair of clients exchanging security tokens and a pair of services consuming those tokens for authentication. Security tokens indicated are sent in WS-Security headers.

credentials allowing them flexible authentication to a wide range of systems even when away from their desktop.

For the most part, we assume a single centralized CredEx service for the credentials of all of the users in an organization or virtual organization. However, this single service then bears a significant risk of failure or compromise. It is certainly possible to run several instances of the CredEx service, the challenge then becomes deciding what credentials to store or replicate where. One could imagine a frontend service to handle mapping credential requests to an appropriate storage service. If replication were used, this frontend could alleviate service failures as well (although storing multiple credential copies has it's own security problems). In general, while CredEx does not explicitly address scalability concerns, its design is not inconsistent with them either. A complete analysis of these issues is unfortunately beyond the scope of this work.

To demonstrate how the above flexible credential exchange features ease interoperability, we developed a usage scenario bridging password-based Web Services and PKI-based Grid Services. Figure 1 shows the components involved and gives a flavor for the messages exchanged; details of the protocol are in the next section.

4. Credential Exchange Protocol

As a true credential repository, our credential service includes a set of credential management functions for uploading, retrieving and removing credentials. Since WS-Trust only defines the request-response mechanism for credential exchange (or retrieval), the service provides an RPC-style SOAP interface for credential management features, which we will refer to as the CredentialManager. A second view of the credential exchanger is as a WS-Trust compliant SecurityTokenService. These two interfaces are described in more detail below.

4.1 CredentialManager Interface and Protocol

The CredentialManager interface exposes methods for storing, retrieving and removing both username-password pairs and X.509 certificates. Because of the desire to support interoperability between Web Services and grid services, the X.509 certificates stored and retrieved are in fact X.509 proxy certificates [26]. Proxy certificates are a simple extension to standard X.509 certificates that add an additional proxy common name (CN) field to the subject as well as some certificate extensions to limit the rights of the proxy certificate holder. Proxy certificates are the standard authentication token used for grid services; however, it would not require a significant modification to the credential exchanger for it to support issuance of standard X.509 certificates (in which case it could function as a kind of online certification authority). Note that we have focused on identity certificates for use in authentication; it's not clear if or how exchanges involving attribute certificates might work.

In order to upload a certificate to the service, the following protocol and methods are used (adapted from the proxy delegation protocol used in MyProxy [23]):

1. Client invokes remote method:
   `byte[] storeCertInit(String username, String password, String alias)`
   associating a username and password with the stored certificate in the process.
2. Service returns a base64-encoded PKCS#10 certificate request
3. Client signs request with private key to generate a new certificate
4. Client passes newly generated certificate and the rest of the certificate chain to the server for storage by invoking the remote method:
   `void storeCert(byte[] certChain, String username, String password, String alias)`

There are several important aspects of this protocol. First, at no point does the client's private key leave the client's system. Never transmitting the private key over the wire is essential to ensuring that the key is not compromised during credential upload. Second, when the client generates the new certificate in Step 3, the client is free to set the lifetime of this certificate to meet its needs. In this way, the client effectively has control over how long the delegated credential is stored to the service. Third, the `alias` parameter shows up in nearly all of the other credential management methods. By allowing the client to give a name to the stored certificate, the client gains the ability to store multiple different credentials which can be selectively retrieved on-demand. The `alias` parameter is optional, it can be set to null (nillable in the SOAP terminology), in which case a null `alias` would be used for retrieval.

When a client wishes to retrieve a stored credential, a username-password pair (established during storage) is used to authenticate the request and would then be exchanged for a proxy certificate. In terms of method invocations, the client calls:

```
byte[] exchangeForCert(byte[] certRequest,
  String username, String password, String
  alias, int lifetimeInHours)
```

This effectively accomplishes the proxy certificate delegation protocol discussed above for storage, but with the roles of the client and service reversed. Here the client generates the certificate request, and will be returned an X.509 certificate, along with the rest of the certificate chain. Once again, the private key stored at the service is

never transmitted in a message exchange. The `lifetimeInHours` parameter allows the client to indicate how long the returned certificate should be good for (though this cannot exceed the lifetime of the stored certificate).

To remove a previously-stored certificate:

```
void removeCert(String username, String
  password, String alias)
```

Password-based authentication is the same for the retrieval functionality and once again the optional `alias` parameter is present to indicate a specific stored certificate to remove. If a problem occurs with this or any of the methods, a fault is generated.

As one might expect, the methods and protocols for management of username-password tokens are conceptually quite similar. In many ways the protocol is even simpler, since the certificate delegation protocol can be avoided. The key difference, of course, is that X.509 signatures are used for authentication, rather than passwords. Here are the relevant methods:

```
void storePassword(byte[] certChain, String
  username, String password, String alias)
String[] exchangeForPassword(byte[]
  certChain, String alias)
void removePassword(byte[] certChain,
  String alias)
```

All three methods require a WS-Security header with a valid X.509 signature to prove that the client indeed possesses the private key matching the first certificate in the chain. To prevent replay attacks a WS-Security timestamp should also be included. The `certChain` parameter is used to associate a stored username-password pair with a particular identity. If successful, the `exchangeForPassword()` method will return the username and password as an array of strings of length two. Since for all of these methods, the passwords are transmitted in plain text, either XML or SSL-based encryption must be used for any kind of confidentiality (the former is less efficient, but can protect messages processed by intermediaries).

4.2 SecurityTokenService Interface and Protocol

The WS-Trust specification introduces the idea of a SecurityTokenService which supports a specific request-response protocol for the issuance, renewal and validation of security tokens. Our implementation is only concerned with the WS-Trust issuance binding describing how security tokens of different (or the same) format are exchanged. Though the protocol itself is very flexible with regard to exchange parameters and token types, for the purposes of our implementation, messages are likely to take one of two forms depending on whether the exchange is password-for-X.509 (equivalent to the `exchangeForCertificate()` method) or X.509-for-password (equivalent to `exchangeForPassword()`).

Figure 2 shows a typical exchange of username-password for X.509 certificate chain. The WS-Security

```
<soapenv:Envelope>
    <soapenv:Header>                                  WS-Security
      <wsse:Security>
        <wsse:UsernameToken wsu:Id="ut">
          <wsse:Username>
            user </wsse:Username>
          <wsse:Password
              Type="#PasswordText">
            pass </wsse:Password>
        </wsse:UsernameToken>
      </wsse:Security>
    </soapenv:Header>
    <soapenv:Body>                                    WS-Trust
      <wst:RequestSecurityToken>
        <wst:TokenType>
          #X509PKIPathv1 </wst:TokenType>
        <wst:RequestType>
          security/trust/Issue </wst:RequestType>
        <wst:Base>
          <wsse:SecurityTokenReference>
            <wsse:ReferenceURI="#ut"/>
        </wsse:SecurityTokenReference>
        </wst:Base>
        <wst:Supporting>
          <wsse:BinarySecurityToken
              ValueType="#PKCS10">
            MIIB....
          </wsse:BinarySecurityToken>
        </wst:Supporting>
        <credex:Alias> credName </credex:Alias>
        <wst:Lifetime> ... </wst:Lifetime>
      </wst:RequestSecurityToken>
    </soapenv:Body>
</soapenv:Envelope>
```

Figure 2: Exchanging a password (UsernameToken) for an X.509 certificate chain using WS-Trust. To simplify, full URIs and XML namespaces are not shown.

```
<soapenv:Envelope>
    <soapenv:Header>                                  WS-Security
      <wsse:Security>
        <wsse:BinarySecurityToken wsu:Id="Id3"
            ValueType="#X509PKIPathv1">
          MIIC...
        </wsse:BinarySecurityToken>
        <ds:Signature>
          ...
        </ds:Signature>
      </wsse:Security>
    </soapenv:Header>
    <soapenv:Body>                                    WS-Trust
      <wst:RequestSecurityToken>
        <wst:TokenType>
          #UsernameToken </wst:TokenType>
        <wst:RequestType>
          security/trust/Issue </wst:RequestType>
        <wst:Base>
          <wsse:SecurityTokenReference>
            <wsse:Reference URI="#Id3"/>
          </wsse:SecurityTokenReference>
        </wst:Base>
        <credex:Alias> bobspass </credex:Alias>
      </wst:RequestSecurityToken>
    </soapenv:Body>
</soapenv:Envelope>
```

Figure 3: Exchanging an X.509 certificate chain for a password pair (`UsernameToken`) using WS-Trust.

header contains the security token used for authentication: a `UsernameToken` with username and password. The SOAP body contains the WS-Trust elements, starting with `<RequestSecurityToken>`. Within each request, the client specifies the desired `TokenType` (in this case a PKIPath-encoded certificate chain), the `RequestType` (always Issue for token exchange) and the `Base` token used in the exchange (can be directly included or by reference). To support the proxy delegation protocol, an additional `Supporting` token is needed to pass along the certificate request. The client can also indicate a desired `Lifetime` for the returned token and the credential `Alias`, using a custom element not defined by WS-Trust.

The exchange of an X.509 certificate for a username and password pair is quite similar (see Figure 3). The `Base` token used and the `TokenType` requested are obviously different an X.509 `Signature` is required for authentication. Additionally, a supporting token is no longer needed and requesting a particular security token `Lifetime` is meaningless for password-based tokens.

A successful token exchange will return a `<RequestSecurityTokenResponse>` message to the client. Besides the `RequestedSecurityToken`, the response will confirm that the request parameters were met, in particular, the `TokenType` and the token `Lifetime` (if used). This way, the client does not have to parse the possibly binary token returned to extract this information. In the case of a password-to-X.509 exchange, the response would include an `X509PKIPath BinarySecurityToken;` for the reverse exchange a `UsernameToken` would be included instead.

Although WS-Trust defines an interface for credential exchange, credential management functions (specifically storage and removal in the case of CredEx) are also an important concern. These functions may be considered part of a management interface for the security token service that allows users and administrators to store tokens and set policies (using WS-Policy [13]) that control what security tokens can be obtained. Unlike a token service where administrators control credential mappings, the token service we have presented gives end-users the ability to create their own mappings between the credentials they possess, which we believe is a novel use of the WS-Trust framework.

Dealing with multiple credentials does require some intelligence on the part of the credential requester (either the user or the service on the user's behalf). For instance, choosing the least priviledged credential for a given operation would require knowing the privilige levels corresponding to various credential aliases. Similarly, retrieving a set of multiple credentials to present for authentication would involve multiple calls to the service and would require knowing which aliases to retrieve. We would expect an organization (or CredEx administrator) to define a conventions for alias names to meet the needs of their applications and services.

5. Implementation

Developed in Java, CredEx makes use of the Apache Axis SOAP implementation [5] which works as a servlet within a Jakarta Tomcat container (deployment in other containers should also be possible). A benefit of Java's cross-platform nature, the service can run in both UNIX (Linux) and Windows environments. The service relies on the open source WS-Security implementation WSS4J [3] and a Java-based WS-Trust implementation developed for this project (JavaTrust).

Perhaps the most important aspect of the service's implementation is physical credential storage. The service relies on the Java Keystore [18] functionality to handle this securely. Credentials in a keystore are individually password protected while the entire keystore resides on disk encrypted by a master password. This provides pretty good protection against credential compromise, even if the system hosting CredEx were attacked.

5.1 Client Implementations

A key goal of CredEx is flexibility, a key facet of this flexibility involves support for multiple languages and platforms. To realize this goal, we developed credential exchanger clients in both Java (Axis, WSS4J, JavaTrust, UNIX/Windows) and C# (.NET, WSE 2.0, Windows). Because of the Web Service interface, in theory, a client could be written in any language with a SOAP implementation and run on any platform with a TCP stack (availability of WS-Security toolkits do tend to lag).

Both Java and .NET implementations have the same feature set: a command-line interface to all of the credential storage, retrieval and exchange functionality described earlier in this section. The most significant difference between the two versions involves the local credential storage facilities. The Java client relies on the keystore mechanism (described above) to store credentials on user systems. The .NET client in contrast relies on the native Windows certificate store [8] for this functionality.

5.2 JavaTrust: Java-based WS-Trust Support

To create CredEx, we first needed to develop our own implementation of WS-Trust for Java (since such an implementation was lacking at the time of development). We have since contributed this implementation to the publicly-available WSS4J project [3]. Many of the WS-Trust elements (`RequestSecurityToken`, `Lifetime`, etc.) map directly to Java classes. So for a client to send an exchange request, the client can simply: create a

RequestSecurityToken object, set the necessary parameters and tokens and add the object's XML to a SOAP Body element. Further the library includes Axis serializer and deserializer classes which can automatically convert between the XML form of a WS-Trust element and its corresponding Java object. Thus, a SecurityTokenService implementer can simply define a Java method of the form:

```
RequestSecurityTokenResponse
  issue(RequestSecurityToken tokenRequest)
```

and never have to worry about the XML underlying the messages.

Our WS-Trust library relies on WSS4J for the various security token formats defined by the WS-Security working group. The measure of any implementation of an open protocol is how well it works with other implementations of the same protocol. Our JavaTrust library is compatible with the WS-Trust implementation supplied by Microsoft as part of the 2.0 version of their Web Services Enhancements (WSE) toolkit (as demonstrated through interop of our WS-Trust-based clients and service).

6. Fostering Interoperability

As mentioned in the introduction, differing authentication mechanisms is a considerable obstacle to the goal of bridging identities across systems. Our flexible credential exchanger provides the ability to swap authentication tokens between formats and goes a good distance toward overcoming these challenges.

To demonstrate this utility in the context of existing systems, we used CredEx to bridge the security mechanisms of two different service architectures: Web Services, which could use any form of authentication tokens, but most commonly, rely on usernames and passwords (or at least this is our assumption) and Grid services which require X.509 proxy credentials.

We developed a pair of test services and clients, similar to Figure 1. The Web Service was implemented in Java using Axis (another SOAP toolkit like .NET would be fine) and requires a WS-Security wrapped `UsernameToken` for authentication. The Grid Service was implemented in Java using the Globus Toolkit [1] and requires X.509-based GSI authentication. The clients are also Java-based, but they too could work on other platforms. A client can invoke the credential exchanger service to convert the token the user currently has for one that will work with the target service (as depicted in Figure 1). Another possibility is to move this credential exchange burden to the service, in which case the target service, would accept a variety of authentication tokens. If, the user-provided token couldn't be used with the service's native authentication mechanism, the service would automatically invoke the credential exchanger on the client's behalf. Thus, users can seamlessly access the services they need to, using the authentication tokens they have available, meeting our originally stated goal.

All of the source code for CredEx is available for download at http://www.cs.virginia.edu/~dad3e/CredEx. We encourage people to download, experiment with the code and provide feedback and bug reports.

7. Conclusions and Future Work

In this paper, we have presented CredEx, which offers new flexibility for credential management in web and grid service environments based on open-source, interoperable standards and implementations. Bringing flexibility to the credential exchange process solves several problems, but clearly many identity management challenges still remain. The key advantage of a credential service such as the one we have been discussing (that is open, standards-based, supports multiple client platforms and several authentication token types) is its ability to facilitate the interoperability of diverse computing systems.

There are several possible avenues for future effort. The most straightforward, if not the most interesting involves adding support for additional security token formats. Kerberos is already in widespread use and interest in SAML is steadily growing; being able support exchanges involving these token types might prove quite valuable (and can be incorporated without major design changes). Such support might also make the credential exchanger more useful in still-nascent federated identity management systems like the SAML-based Liberty Identity Federation Framework or even .NET Passport which is purportedly adding Kerberos support [19].

While WS-Trust provides the flexibility for obtaining credentials of different types, a flexible interface for storing different token types would also be welcome. (Our storeCert and storePassword methods are token type specific.) Part of support for additional credential types should include an extensible storage interface in the spirit of WS-Trust. This interface would allow more flexible policies, specifying how stored credentials can be accessed. For example a Kerberos ticket could be requested via password, an X.509 certificate, or another Kerberos ticket that satisfies the associated policy.

Another future direction focuses on credential delegation. It is often desirable to have third-party systems act on behalf of individual users, and users often need to be able to delegate privileges associated with their identity. The WS-Trust protocol describes how such delegation concerns might be expressed in the context of credential exchanges and in fact has been proposed as a standard delegation protocol for grid applications [4].

While the existing credential service fails to address complex authorization issues, the credential exchanger is a powerful tool and being able to restrict access along several dimensions would certainly be worthwhile. For instance, credential service administrators might wish to manage authorization based on user or group membership or base authentication token used, time of request, etc. A flexible credential exchanger deserves flexible authorization controls; policies might be expressed using a combination of policy languages such as WS-Policy [13] for defining policies of interest to clients and the eXtensible Access Control Markup Language (XACML) [12] for expressing application level policy semantics.

8. Acknowledgements

This work would not have been possible without the strong support from Anthony Nadalin and Chris McMahon of IBM. The University of Virginia authors are supported in part by the US National Science Foundation under grants ACI-0203960, SCI-0426972, a DOE Early Career Award, and an IBM Faculty Fellowship Award.

9. References

[1] About the Globus Toolkit. Accessed 29 April 2005. http://www-unix.globus.org/toolkit/about.html

[2] S. Anderson, et al. Web Services Trust Language (WS-Trust), v. 1.1. May 2004. ftp://www6.software.ibm.com/software/developer/library/ws-trust.pdf

[3] Apache WSS4J. Apache Web Services Project, accessed 5 August 2004. http://ws.apache.org/ws-fx/wss4j

[4] M. Ashant, J. Basney and O. Mulmo. Grid Delegation Protocol. UK Workshop on Grid Security Practice, Oxford, July 2004. http://www.ncsa.uiuc.edu/~jbasney/Grid-Delegation-Protocol.pdf

[5] Axis User's Guide, v1.1. Apache Software Foundation, 16 June 2003. http://ws.apache.org/axis/java/ user-guide.html.

[6] J. Basney, et al. An OGSI CredentialManager Service. UK Workshop on Grid Security Practice, Oxford, July 2004.

[7] S. Carmody. Shibboleth Overview and Requirements. 20 February 2001. http://shibboleth.internet2.edu/docs/draft-internet2-shibboleth-requirements-01.html

[8] Certificate Stores. MSDN Library, accessed 5 August 2004. http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/sag_cmuncertstor.mspx.

[9] Entrust TruePass Product Portfolio: Technical Overview. July 2003. http://www.entrust.com/contact/download.cfm/entrust_truepass_tech_overview.pdf

[10] S. Farrell, Ed. Securely Available Credentials Protocol (RFC 3767). IETF Network Working Group, June 2004. http://www.ietf.org/rfc/rfc3767.txt

[11] M.J. Foley. Microsoft 'TrustBridge' Resurfaces. eWeek, Ziff Davis Media, 26 May 2004. http://www.eweek.com/article2/0,1759,1601790,00.asp

[12] S. Godik and T. Moses, Eds. eXtensible Access Control Markup Language (XACML), 1.0. OASIS Standard, 18 February 2003. http://www.oasis-open.org/committees/download.php/2406/oasis-xacml-1.0.pdf

[13] M. Hondo and C. Kaler, Eds. Web Services Policy Framework. 28 May 2003. http://www-106.ibm.com/developerworks/library/ws-polfram

[14] R. Housley, W. Ford, W. Polk and D. Solo. Internet X.509 PKI Certificate and CRL Profile (RFC 2459). IETF Network WG, 1999. http://www.ietf.org/rfc/rfc2459.txt

[15] J. Hughes and E. Maler. Technical Overview of the OASIS SAML v1.1. OASIS Security Services TC Draft, 11 May 2004. http://www.oasis-open.org/committees/download.php/6837/sstc-saml-tech-overview-1.1-cd.pdf

[16] C. Kaler and A. Nadalin, Eds. Web Services Federation Language (WS-Federation). 8 July 2003. http://www-106.ibm.com/developerworks/webservices/library/ws-fed.

[17] J. Kemp, Ed. Liberty ID-WSF. May 2004. https://www.projectliberty.org/resources/whitepapers/Liberty_ID-WSF_Web_Services_ Framework.pdf

[18] keytool - Key and Certificate Management Tool. J2SE SDK Docs, accessed 5 August 2004. http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/keytool.html.

[19] Microsoft Expands Passport to Enable Universal Single Sign-In. Microsoft Press Release, 20 September 2001. Available at http://www.microsoft.com/presspass/press/2001/sep01/09-20PassportFederationPR.asp

[20] A. Nadalin, et al., Eds. Web Services Security 1.0 (WS-Security). OASIS Standard 200401, March 2004. http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf

[21] B.C. Neuman and T. Ts'o. Kerberos: An Authentication Service for Computer Networks. IEEE Communications, 32 (9), September 1994, pp. 33-38.

[22] J. Novotny, S. Tuecke and V. Welch. An Online Credential Repository for the Grid: MyProxy. Proceedings, HPDC-10, August 2001, pp. 104-111.

[23] B. Schneier. Password Safe: The Security of Blowfish in a Password Database. Accessed 6 October 2004. http://www.schneier.com/passsafe.html

[24] F. Siebenlist, et al. OGSA Security Roadmap: Towards a Secure OGSA. July 2002. http://www.globus.org/ogsa/security/draft-ggf-ogsa-sec-roadmap-01.doc

[25] D.D. Skow. Fermilab Public-Key Infrastructure. 26 March 2004. http://computing.fnal.gov/security/pki.

[26] S. Tuecke, et al. Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile (RFC 3820). IETF Network WG, June 2004. http://www.ietf.org/rfc/ rfc3820.txt.

[27] T. Wason, Ed. Liberty ID-FF Architecture Overview. Version 1.2, 2003. http://www.projectliberty.org/specs/liberty-idff-arch-overview-v1.2.pdf

[28] KX.509/KCA. http://www.nsf-middleware.org/documentation/NMI-R5/0/gridscenter/KX509KCA/index.htm

[29] G. Della-Libera et al. WS-Security Kerberos Binding. December 2003. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-security-kerberos.asp

[30] P. Hallam-Baker et al. Eds. WS-Security: SAML Token Profile. Draft 15, 19 July 2004. http://www.oasis-open.org/committees/ download.php/ 7837/WSS-SAML-15.pdf