Enhancing User Authentication in Claim-Based Identity Management

Waleed A. Alrodhan and Chris J. Mitchell
Information Security Group
Royal Holloway, University of London
{W.A.Alrodhan, C.Mitchell}@rhul.ac.uk
http://www.isg.rhul.ac.uk

Abstract

In claim-based identity management (CBIM) systems, users identify themselves using security tokens that contain personally identifiable information, and that are signed by an identity provider. However, a malicious identity provider could readily impersonate any user by generating appropriate tokens. The growing number of identity theft techniques raises the risk of service providers being deceived by untrustworthy identity providers. We show how this vulnerability can be mitigated by adding an authentication layer, between the user and the service provider, to a CBIM system. We propose two possible implementations of this layer. The first approach requires a user to perform an additional step before the service provider completes the authentication process. That is, the user must present to the service provider certain information sent to the user by the service provider during the most recent successful use of the scheme. A proof-of-concept implementation of this scheme has been produced. The second approach involves a challenge-response exchange between the user and the service provider. This requires a minor modification to the service provider XML-based security policy declaration message.

1 Introduction

Identity management is one of the fundamental building blocks for collaborative environments. Collaborative applications like Wiki, for example, rely on identity management schemes to identify users and protect their personally identifiable information (PII), as a preparatory step for user authorisation. However, it has become common, or even necessary, for Internet users to possess multiple digital identities. Managing these identities and protecting the corresponding credentials is a difficult problem because of the threats of identity theft and phishing techniques, and also the growing number of such identities.

User-centric identity management [9, 11] has been proposed as a means of easing the user task of managing digital identities, by providing users with more control over their identities. Such systems have been developed primarily from the perspective of end-users, enabling a user to maintain control over how PII is both created and used, thereby enhancing user privacy. Examples of user-centric systems include OpenID[1] and Windows CardSpace[2] (henceforth abbreviated to CardSpace). *Claim-based systems* [1, 12] are one particular class of such systems.

In this paper we aim to enhance user authentication within claim-based identity management systems. We propose two possible ways in which such an enhancement could be provided. In the first approach, a user must present certain information obtained during the last interaction with a service provider in order to be authenticated to that provider. The second approach involves a challenge-response procedure between the user platform and the service provider, requiring a minor modification to the service provider XML-based security policy declaration.

The remainder of this paper is organised as follows. Section 2 provides an overview of claim-based identity management. In section 3 we propose two methods for enhancing user authentication. In section 4 we describe a prototype implementation of one of the proposed techniques, and section 5 discusses their effectiveness. Finally, section 6 concludes the paper.

2 Claim-based identity management

In this section we provide a brief overview of claim-based identity management. We then discuss the notion of 'user consent'.

[1]http://openid.net
[2]http://www.microsoft.com/net/cardspace.aspx

2.1 Overview

Many Internet identity management systems are designed to be cost effective from the perspective of service providers rather than users. For example, many service providers manage digital identities using automated systems, whereas users are required to manage their digital identities manually. Also, in most identity management systems, service providers authenticate users using an application layer technique (e.g. username and password), whereas users authenticate service providers using a lower layer technique (such as SSL/TLS). Hence, managing multiple digital identities and protecting the associated credentials can become very difficult for users. Moreover, most such systems are *isolated*, i.e. there is no co-operation between systems for user authentication purposes [11]. As shown in figure 1, when using isolated systems, users must manage multiple identifiers manually, and must maintain a distinct identifier for each service provider.

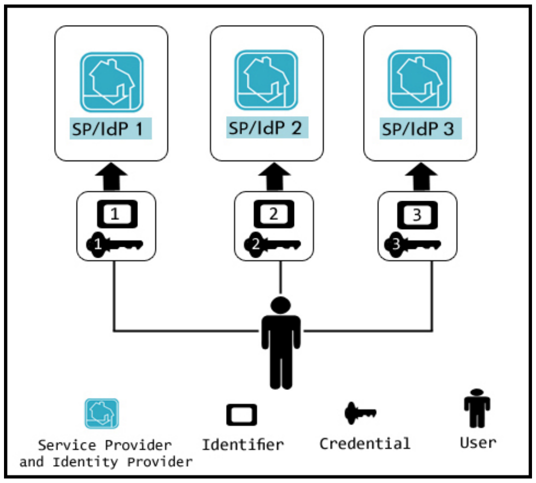


Figure 1. Isolated identity management model

Claim-based identity management (CBIM) has been designed to make identity management easier for Internet users. As shown in figure 2, every human user has specific PII. The idea is to enable users to use their PII to identify themselves to service providers, instead of using service provider specific identifiers (e.g. usernames) and access credentials (e.g. passwords).

In a CBIM system, each individual has an associated set of *claims*, where a claim is an assertion of the truth of some piece of PII for the associated user. In order to authenticate the user, the service provider can demand a security token that asserts the truth of values for certain pieces of user PII. This security token must be signed by a trusted identity provider. Figure 3 shows the claim-based identity

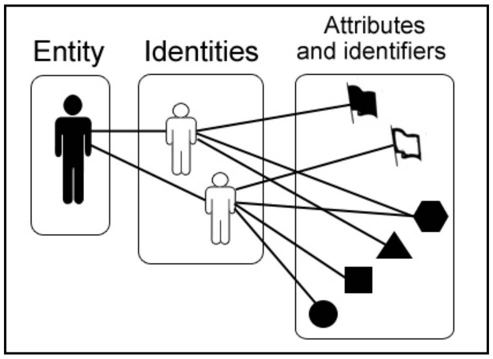


Figure 2. The relationship between entities and identifiers

management model.

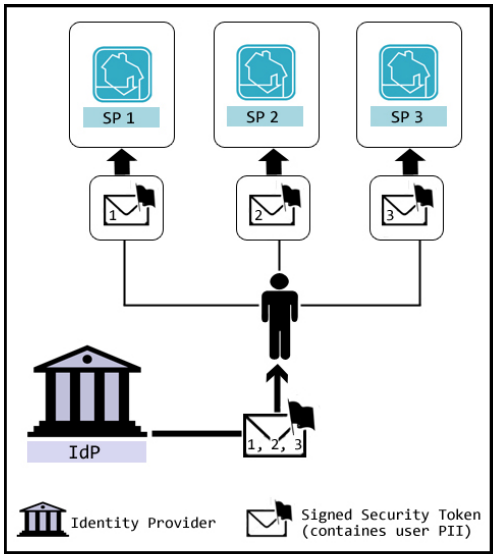


Figure 3. Claim-based identity management model

The most widely discussed example of a CBIM system is CardSpace. CardSpace is designed to satisfy the requirements of the Laws of Identity [3, 4]. The CardSpace framework is based on the identification process we experience in the real world using physical identification cards. Within the CardSpace framework, an identity provider issues Internet users with virtual cards called Information Cards (or *InfoCards*), that hold (relatively) non-sensitive meta-information related to them, including a list of the supported claims. Subsequently, the Internet user selects one of the InfoCards that supports the claims that the service provider wishes to have assurance of, and requests a signed security token from the identity provider that issued this InfoCard.

The request sent to the identity provider specifies the types of the claims sought by the service provider. The provided security token lists the values of the requested claims and, after receiving it, the CardSpace-enabling component on the user machine forwards it to the service provider. InfoCards can also be self-issued by the Internet users themselves.

Other CBIM systems include OpenInfoCard[3], Higgins, and DigitalMe[4]. Higgins and DigitalMe are supported by the Bandit Project[5]. We focus here on CardSpace because of its ubiquity as part of Windows Vista; however, many of the observations made below also apply to other CBIM systems, since they have strong similarities to one another.

2.2 User consent

In the currently deployed implementations of CBIM, service providers are not provided with a proof of the legitimacy of the user that wishes to log in. Instead, service providers are given a proof of rightful possession of the security token (i.e. a proof that the user who forwarded the security token has the right to possess it). However, all the implemented "proof of rightful possession" techniques are based on information included within the security token itself [20]. Moreover, the only means for the service provider to judge the validity of a security token is by verifying the identity provider's digital signature. This means that, if the identity provider is lying, then the 'proof of rightful possession' will be false.

The first of the Laws of Identity is to provide a high degree of 'user control and consent'; we therefore suggest that support for user consent in systems that adhere to these laws, such as CardSpace, needs to be enhanced. Also, there is a potential vulnerability because of the lack of robust evidence of user consent.

We believe that our proposed schemes will enhance the service provider ability to make more accurate decisions about the legitimacy of the user by adding an additional authentication layer. This layer could be deemed as providing implicit indication that the log-in attempts are initiated by the legitimate users.

Finally, we observe that the problem we have described is less significant for self-issued InfoCards.

3 Enhancing user authentication in CBIM systems

In this section we propose two methods for enhancing user authentication in CBIM systems. The proposed methods are independent, and can be combined if desired. We describe the techniques as they apply to CardSpace; however, we believe that they could also be applied to other CBIM systems.

3.1 The CardSpace framework

CardSpace provides a way to represent identities using claims, and a means to bridge technology and organisational boundaries using claims transformations [13]. It is a CBIM system, and is not a single sign-on system. It aims to reduce the reliance on passwords for Internet user authentication by service providers (or Relaying Parties (RPs) in Microsoft terminology), and to improve the privacy of personal information.

The CardSpace identity management architecture is designed to provide the user with control over his digital identities in a user friendly manner, and to tackle identity management security problems such as breaches of privacy and identity theft, with no single identity authority control. CardSpace works with Internet Explorer browsers (CardSpace plug-ins for browsers other than Microsoft Internet Explorer have also been developed, such as the Firefox Plug-in[6]).

Digital identities in CardSpace are represented as claims made by one digital subject (e.g. an Internet user) about itself or another digital subject. A claim is an assertion that certain identifying information (e.g. given name, social security number, credit card number, etc.) belongs to a given digital subject [4, 13]. Under this definition, user identifiers (e.g. a username) and user attributes (e.g. user gender) are both treated as claims.

CardSpace requires users to install an enabling component on their machines called the *Identity Selector* [14]. This component performs several important tasks including: providing a user-friendly interface for Information Card management and security token viewing, negotiating the security requirements of the RPs and identity providers (IdPs), supporting identity provider discovery, controlling and managing user authentication to the IdP, and generating self-issued security tokens. These tokens contain assertions made by the users about themselves, and are generated by the Self-issued Identity Provider (SIP), part of the Identity Selector.

The *Security Token Service* (STS) is a software component in the CardSpace framework, responsible for security policy and token management at the IdP and, optionally, at the RP [10].

The framework is based on the identification process we experience in the real world using physical ID cards. InfoCards are issued either by an IdP or by the SIP on the user machine (in which case they are referred to as self-issued cards). Infocards are stored on the user machine,

[3]http://code.google.com/p/openinfocard

[4]More implementations are listed at: http://www.osis.idcommons.net

[5]http://bandit-project.org

[6]http://xmldap.blogspot.com/2006/05/firefox-identity-selector.html

and are XML files with the extension '*.crd'. An InfoCard is signed by the IdP, and contains (relatively) non-sensitive meta-information, such as: the name of the IdP that has issued it, a list of the claims that can be asserted by the IdP, the types of security tokens that can be requested from this IdP (e.g. a SAML 2.0 assertion), and the InfoCard creation and expiry times.

When a user tries to log-in to a CardSpace-enabled RP, the RP declares its security policy to the Identity Selector. The RP security policy can be retrieved using the WS-MetadataExchange protocol [7], and is expressed using the WS-SecurityPolicy [17] and WS-Trust [18] protocols. The policy includes: the claims to be asserted, the requested security token type, a list of IdPs trusted to issue the requested token, and the required proof-of-rightful-possession method. The RP security policy also specifies constraints on the retrieved security token (e.g. the maximum token age).

After processing that policy, the Identity Selector checks which InfoCards satisfy it, and prompts the user to select one of them. The Identity Selector retrieves the IdP security policy from the IdP that issued the selected InfoCard. This policy specifies how the Identity Selector must be authenticated, and how to retrieve a security token from the IdP. The policy is contained within a WSDL description [6], specifying the protocol messages to be used to access the IdP-STS. The policy contains details of the security measures that should be applied to the request token (e.g. whether the security token should be encrypted by the IdP using a short-term symmetric session key, or if the encryption provided by SSL/TLS is sufficient) [5]. The IdP security policy must always contain the IdP's X.509 public key certificate.

The security token is then requested from the issuer IdP. After receiving the request, and prior to authenticating the user and generating the token, the IdP checks its policy to decide how it should authenticate the user, what claims it can assert, and whether its policy permits it to generate the requested security token. On receipt of the token from the IdP, the Identity Selector optionally shows its contents to the user (the displayed information is deleted from the system after the user has given consent to proceed). Finally, the Identity Selector forwards the security token to the RP, which will deem the user authenticated if the received token is valid and meets its requirements.

Figure 4 provides a simplified sketch of the CardSpace framework. In the figure it is assumed that the user has already been issued an InfoCard by an IdP, and has retrieved the RP web page that offers a CardSpace-based log-in. In step 1, the user clicks on the CardSpace icon in the RP web page using a CardSpace-Enabled User Agent (CEUA), also known as the *Service Requester*, which is essentially a CardSpace-enabled web browser. In step 2, the RP identifies itself using a public key certificate (e.g. a certificate

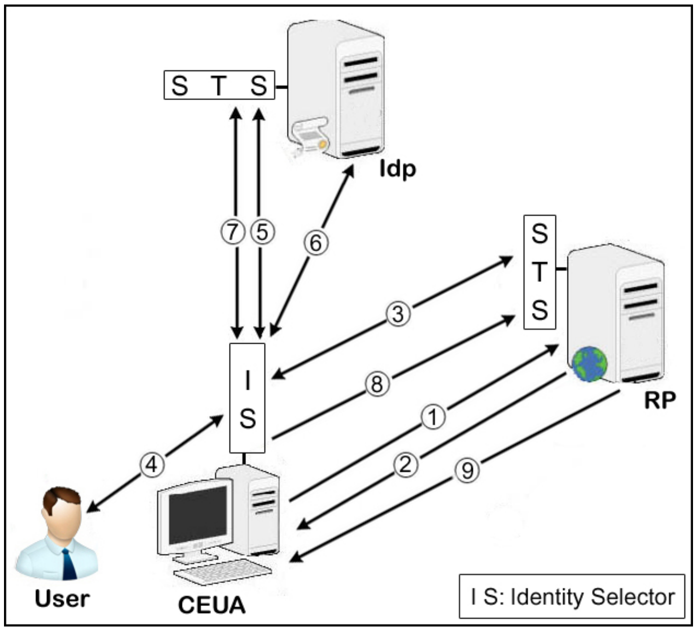


Figure 4. CardSpace Framework.

used for SSL/TLS), and triggers the Identity Selector using XHTML code or HTML object tags. After the Identity Selector has been triggered, it retrieves the RP's security policy from the RP-STS in step 3 [20].

In step 4 the Identity Selector matches the RP's security policy against the InfoCards possessed by the user in order to find one that satisfies the policy. If one or more suitable InfoCards are found, the user is prompted to select one of them. After the user has selected an InfoCard, the Identity Selector initiates a connection with the IdP that issued that InfoCard, and retrieves the IdP security policy in step 5. In step 6, the user performs an authentication process with the IdP via the Identity Selector. The current version of the Identity Selector supports four authentication methods, namely: username/password, Kerberos V5 ticket, X.509 certificate (either software-based or using a smart card), or self-issued SAML 1.1 assertion (generated by the SIP) [20].

Then, in step 7, the Identity Selector requests the IdP to provide a security token that asserts the truth of the claims whose types are listed in the selected InfoCard; this request is sent in a *request security token* (RST) message. The IdP then checks whether its security policy permits it to generate the requested security token. If so, the IdP replies by sending a security token within a *request security token response* (RSTR) message. Finally, the Identity Selector forwards the security token to the RP-STS in step 8 (after, optionally, showing its contents to the user) and, if the RP verifies it successfully, the service is granted in step 9.

The RP will get an assertion from the IdP that the security token was issued to a particular user. This assertion is bound to the user by a secret 'proof-key', where a user asserts ownership of a security token by demonstrating knowledge of the proof-key included in the token [15]. This helps to prevent token replay attacks, i.e. where an attacker 'steals' a token for another user. The RP can select one of three types of proof of rightful possession techniques, namely: bearer, symmetric and asymmetric [14].

The CardSpace identity metasystem relies on a number of Web Services protocols and SOAP [16]. Most of these protocols require the SP to have an STS server in order to process the messages [2, 10, 19]. The CardSpace message flows are as follows:

1. **CEUA** → **RP** : User clicks on the CardSpace logo on the RP log-in web page
2. **RP** → **CEUA** : InfoCard Tags (XHTML or HTML object tags), to trigger the Identity Selector
3. **Identity Selector** ↔ **RP-STS** : Identity Selector retrieves the RP security policy via *WS-MetadataExchange*
4. **Identity Selector** ↔ **User** : User picks an InfoCard
5. **Identity Selector** ↔ **IdP-STS** : Identity Selector retrieves the IdP security policy
6. **Identity Selector** ↔ **IdP** : User Authentication
7. **Identity Selector** ↔ **IdP-STS** : Identity Selector retrieves security token via *WS-MetadataExchange*
8. **Identity Selector** → **RP-STS** : Identity Selector forwards the security token (after, optionally, showing its contents to the user)
9. **RP** → **CEUA** : Welcome, you are now logged in!

The messages in steps 3, 5, 7 and 8 are carried over SOAP, and must be transmitted over an SSL/TLS channel to preserve their confidentiality. If the SP does not have an STS server, then the messages in steps 3 and 8 will be carried using HTTP over an SSL/TLS channel. The security token must be encapsulated in a WS-Trust message [18], and its integrity is preserved using an XML-Signature, as part of the WS-Security protocol [19].

3.2 A proof-of-authenticity method

We now describe an approach to the provision of an additional layer of authentication which we call a 'proof-of-authenticity' method. It requires the user platform to store a secret *proof-of-authenticity* value (known only to the client and the RP), that is sent to the RP during the authentication process. Provision of this secret value proves to the RP that the genuine user is involved, and hence implicitly gives indication that the log-in attempt was initiated by the legitimate user.

The *proof-of-authenticity* value is randomly generated by the RP, and a new value is sent to the user platform after every successful authentication (e.g. in the form of HTTP cookie). That is, when a user inaugurates a log in procedure using the CBIM system, the RP will request the current *proof-of-authenticity* value from the user platform. The RP will verify that the provided value is as expected and, if so, will continue with the authentication process of the CBIM system. If a value is not available to the user platform (e.g. because this is the first occasion that the system has been used from this platform), or the provided value is incorrect (e.g. because the user has switched platforms), then the CBIM system authentication procedure will be aborted; the user will then be requested to authenticate him/herself by some other means (e.g. user name and password). This latter means should involve the use of information not known to the identity provider. Once the user has been authenticated (using the CBIM system or by some other means), the RP will generate a new random *proof-of-authenticity* value, store it, and send it to the user via a secure channel.

The provision of the *proof-of-authenticity* value should be transparent to the user, and hence will not affect the usability of the system. To demonstrate this fact, a prototype implementation of this scheme has been developed (see section 4 below).

3.3 Challenge-response method

The second approach requires the user platform to either share a secret key with the RP or possess a signature key pair for which the RP has a trusted copy of the public key. The key is used as the basis of a challenge-response authentication of the user to the RP.

This method requires modifying the XML-based security policy declaration message sent from the RP to the user platform (i.e. the CardSpace enabled web browser) during security policy negotiation. Apart from presenting the requested token, the user platform is required to provide a valid response to a challenge sent by the RP. This response is computed using either a secret key shared by the user and the RP, or a private signature key belonging to the user. We now describe the operation of these two possibilities in greater detail.

3.3.1 MACed-response mechanism

This mechanism requires the RP and the user to share a secret key. This key can be issued by the RP during the registration phase (i.e. when the user first registers an account with the RP). It can be replaced if lost or compromised, and can be stored on the user machine or on a security token such as a smartcard. We assume that the user establishes the shared secret key with the RP before any attempt at user impersonation by a malicious IdP.

Use of the mechanism also requires the RP and the user platform to agree on the use of a Message Authentication Code (MAC) algorithm, where we write $MAC_k(x)$ to denote the MAC computed on data x using the key k.

The RP must also have the ability to request a user-consent assertion. This request can be embedded within its security policy declaration message. This requires certain minor modifications to be made to the the WS-SecurityPolicy message that contains the RP security policy. To achieve this we propose the introduction of a new tag. This new tag, which we call `<UserConsentRequest>`, contains three data fields. The first holds the mechanism to be used, the second holds a boolean value that indicates whether or not the SP requires a user-participation assertion, and the third holds the challenge value (explained below).

Figure 5 shows an *XML Schema* for the added tags, and a *Document Type Definition* (DTD) of these tags is as follows.

```
  <!ELEMENT UserConsentRequest (Type,
AssertionRequested, Challenge)>
  <!ATTLIST Type Method (MACed | Signed)
"MACed">
  <!ATTLIST AssertionRequested Enhanced (True |
False) "False">
  <!ELEMENT Challenge (♯PCDATA)>
```

Figure 6 gives an example of an XML message declaring an RP security policy expressed using the WS-SecurityPolicy standard. The policy states that the security token to be received must be issued by a specific identity provider (*contoso.com*), and that the desired proof of rightful possession method is the *Symmetric* method [20]. The policy also lists the claims to be asserted (i.e. by listing their values in the security token). In this example, the requested claims are the given name and the surname, and each claim is defined using a specific URI. The novel tags are marked by inclusion within a box.

After processing the security policy of the RP, the CardSpace enabling component on the user machine checks whether or not the RP is requesting a user-consent assertion by checking the value of the `<AssertionRequested>` field. If the value is *true*, then the enabling component extracts the value of the `<Challenge>` field, which is essentially a randomly generated *nonce* (i.e. a Number used ONCE). If the value of the `<Method>` field is `MACed`, then the user agent (or the enabling component) uses this nonce to generate a MAC:

$$MAC_k(id_{RP} \| n)$$

where $\|$ denotes concatenation; id_{RP} is an identifier for the RP, e.g. the RP's domain name; n is the nonce; and k is the shared key.

The generated response is then sent to the RP along with the security token. Finally, the RP checks the response using the shared key k. If the MACed-response is correct, then

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified">
 <xs:import namespace="http://www.w3.org/XML/1998/namespace"/>
  <xs:element name="UserConsentRequest">
    <xs:complexType>
       <xs:sequence>
         <xs:element ref="Type"/>
        <xs:element ref="AssertionRequested"/>
         <xs:element ref="Challenge"/>
      </xs:sequence>
    </xs:complexType>
 </xs:element>
    <xs:element name="Type">
     <xs:complexType>
      <xs:attribute name="Method" default="MACed">
        <xs:simpleType>
          <xs:restriction>
            <xs:enumeration value="MACed"/>
            <xs:enumeration value="Signed"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
     </xs:complexType>
 </xs:element>
 <xs:element name="AssertionRequested"/>
    <xs:complexType mixed="true">
        <xs:attribute name="Enhanced" default="False">
            <xs:simpleType>
                <xs:restriction base="xs:NMTOKEN">
                    <xs:enumeration value="True"/>
                    <xs:enumeration value="False"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>
 </xs:element>
 <xs:element name="Challenge"/>
    <xs:complexType mixed="true"/>
 </xs:element>
</xs:schema>
```

Figure 5. XML schema for the new tags

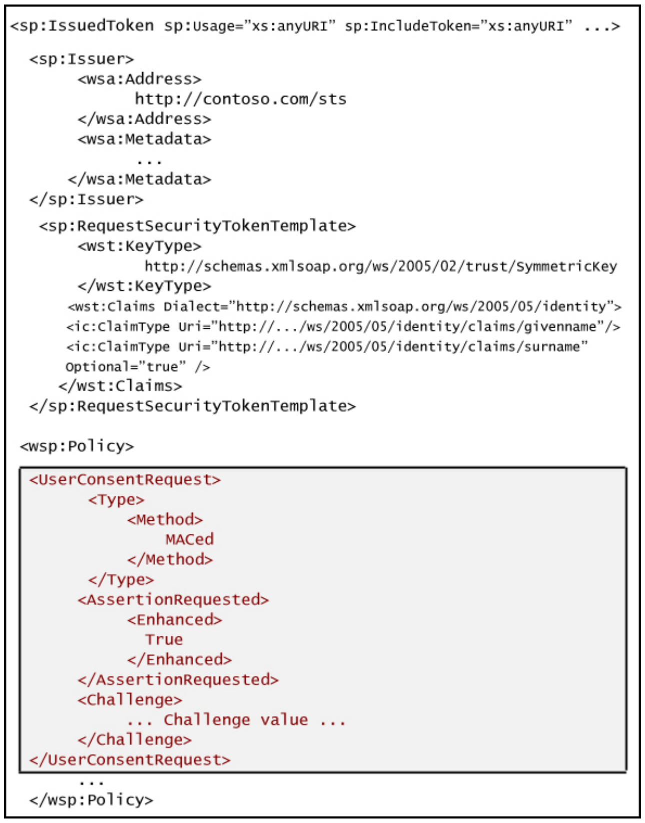
this acts as an implicit assertion of user-consent. The use of a nonce as a challenge helps to prevent replay attacks.

3.3.2 Signed-response mechanism

To use this mechanism, the user platform must have access to a key pair for a digital signature scheme. The user and RP must also agree on the use of a particular such scheme; we write $S_{user}(x)$ to denote the user signature on data x. The key pair can be issued to the user either by the RP during the registration phase (i.e. when the user first registers an account with the RP), or by a trusted Certification Authority (in this case the user must provide its public key certificate to the RP before this mechanism can be used). The key pair can be replaced if lost or compromised, and can be stored on the user machine or on a security token such as a smartcard.

An obvious question would be: Why use CardSpace if there is already a PKI in use? The answer is simple: because the user and/or the RP might wish to use CardSpace to retrieve attributes from the IdP for authorisation purposes.

This mechanism works in exactly the same way as the



```
<sp:IssuedToken sp:Usage="xs:anyURI" sp:IncludeToken="xs:anyURI" ...>
  <sp:Issuer>
       <wsa:Address>
             http://contoso.com/sts
       </wsa:Address>
       <wsa:Metadata>
             ...
      </wsa:Metadata>
  </sp:Issuer>
   <sp:RequestSecurityTokenTemplate>
       <wst:KeyType>
              http://schemas.xmlsoap.org/ws/2005/02/trust/SymmetricKey
       </wst:KeyType>
      <wst:Claims Dialect="http://schemas.xmlsoap.org/ws/2005/05/identity">
      <ic:ClaimType Uri="http://.../ws/2005/05/identity/claims/givenname"/>
      <ic:ClaimType Uri="http://.../ws/2005/05/identity/claims/surname"
      Optional="true" />
     </wst:Claims>
  </sp:RequestSecurityTokenTemplate>

 <wsp:Policy>
  <UserConsentRequest>
        <Type>
            <Method>
                MACed
            </Method>
        </Type>
       <AssertionRequested>
            <Enhanced>
              True
            </Enhanced>
       </AssertionRequested>
       <Challenge>
            ... Challenge value ...
       </Challenge>
  </UserConsentRequest>
       ...
  </wsp:Policy>
```

Figure 6. Modified RP security policy

MACed-response mechanism, except that, instead of MACing the challenge value, the user platform signs the value using its private key. The value of the `<Method>` field must be `Signed`. The response will be as follows:

$$S_{user}(id_{RP} \| n)$$

where id_{RP} is an identifier for the RP, e.g. the RP's domain name; and n is the nonce.

4 Implementing a proof-of-authenticity method

A proof of concept implementation of the proof-of-authenticity method has been successfully tested. The prototype was built on the Pamela Project's[7] implementation of the RP CardSpace component. The implementation involved modifying the component by creating two software modules to be held on the RP server; the two modules were written using the PHP programming langauge (version 5). The implementation has been successfully tested on an Apache web server (version 2.2.8) running on the Linux-Fedora operating system.

[7]http://pamelaproject.com/

The *proof-of-authenticity* (PoA) is stored on the user machine in the form of an HTTP cookie. The PoA value is generated by hashing a combination of a random value and transaction-specific information, to minimise the possibility of accidental re-use of the same value.

The two software modules that perform the required operations for the proof-of-authenticity method are called *PoASet* and *PoACheck*. These two modules are integrated with the CardSpace-enabling software on the RP's server. *PoASet* operates after the user has been authenticated using a mechanism that does not rely on information known by the identity provider (e.g. using username/password). *PoASet* creates a PoA, stores it in a server database, and sends it to the user in the form of an HTTP cookie. *PoACheck* decides whether or not the user can use the CardSpace authentication system. It first checks whether or not the user platform possesses the valid PoA. If not, then *PoACheck* denies the user request to use the CardSpace system and informs the user that it will need to be authenticated using another authentication mechanism. If the supplied PoA is correct, *PoACheck* creates a new PoA, stores it in its database, and sends a copy of it to the user in the form of an HTTP cookie. Finally, it redirects the user's browser to a web page where the user can perform the authentication process using the CardSpace framework[8].

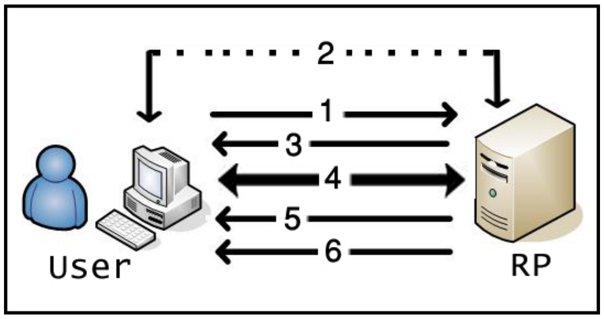


Figure 7. Initial login using the proof-of-authenticity method

Figure 7 shows the message flow for the first user login (i.e. where the user does not have a correct PoA). The message flow steps are:

1. **User** $\rightarrow$ **RP** : Login request using CardSpace.
2. **User** $\leftrightarrow$ **RP** : RP checks whether or not the user has got the correct PoA.
3. **User** $\leftarrow$ **RP** : Sorry you cannot use CardSpace this time!
4. **User** $\leftrightarrow$ **RP** : Authentication of the user using another mechanism (e.g. username/password).

[8]The source code of the two modules is given in an appendix to this paper.

5. **User** $\leftarrow$ **RP** : You have been authenticated, Welcome!

6. **User** $\leftarrow$ **RP** : PoA to be presented next time.

After being issued with a PoA, the user will be able to use CardSpace in subsequent login attempts from this host machine.

5 Discussion

The proposed methods have the capability to increase the privacy level of CBIM systems. They can also help to make the RP's judgement regarding the validity of the security token less critical.

If one of the proposed method is deployed, then dishonest identity providers are prevented from impersonating users. This will not only enhance the reliability of the system from the perspective of service providers, but will also indirectly benefit users by reducing the risk to information held by service providers on their behalf. The proposed methods will also reduce the significance of 'token-stealing' attacks, such as those described in [8].

One possible disadvantage of the proposed methods is that they have an impact on user mobility. This can be addressed by storing the PoA or the user keys on a portable security token such as a smart card, or by storing them at a trusted third party. The latter solution would, however, add complexity to the system.

One obvious limitation of the proof-of-authenticity method is that it requires the user to be authenticated at least once using another authentication system before the CBIM system can be used. However, we believe that the security risk of this limitation is not significant, especially if the user is a frequent visitor to the RP's web site.

The proposed challenge-response method is built on the WS-SecurityPolicy standard, which is widely used in CBIM systems. Hence, integrating the method into currently deployed CBIM systems should be straightforward.

A limitation of the challenge-response method is that it requires modifications to the CBIM-enabling components on the user machine and the RP server (including the RP *Security Token Service*, an RP server based component responsible for declaring the RP security policy and managing received security tokens). A further limitation is the key management overhead. However, if the shared key is compromised or stolen by an attacker, then it would not by itself give immediate access to the RP, since it only provides an additional layer of authentication. That is, the key management process is arguably less security-critical than in many other applications.

6 Concluding remarks

Since collaborative environments rely on identity management to securely identify and authenticate the users, we believe that enhancing the user authentication in CBIM systems, as one of the most prominent identity management solutions, will open the door for more reliable collaborative applications.

In this paper we have proposed two independent methods to enhance user authentication in CBIM systems, namely the proof-of-authenticity method and the challenge-response method. These methods, if implemented correctly, provide the RP with an implicit indication that the log-in attempts are initiated by the legitimate users. A proof-of-concept implementation of the first method has been described.

The proposed techniques do add a certain degree of complexity and overhead to the system. However, we believe that implementing them should help to increase user acceptance of CBIM systems, and also help to enhance the accuracy of the RP judgement of the legitimacy of the user.

References

[1] V. Bertocci, G. Serack, and C. Baker. *Understanding Windows CardSpace*. Addison-Wesley, 2008.

[2] K. Beznosov, D. J. Flinn, S. Kawamoto, and B. Hartman. Introduction to Web services and their security. *Information Security Technical Report*, 10:2–14, 2005.

[3] K. Cameron. The laws of identity, May 2005. Microsoft Corporation.

[4] K. Cameron and M. B. Jones. Design rationale behind the identity metasystem architecture, February 2006. Microsoft Corporation.

[5] D. W. Chadwick. Federated identity management. In *International School on Foundations of Security Analysis and Design (FOSAD'08)*, volume 5705 of *Lecture Notes in Computer Science*, pages 96–120. Springer-Verlag, 2008.

[6] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) — version 1.1, March 2001. The World Wide Web Consortium (W3C).

[7] F. Curbera, S. Parastatidis, and J. Schlimmer (editors). Web services metadata exchange (WS-MetadataExchange) — version 1.1, August 2006. BEA Systems, Computer Associates, IBM, Microsoft, SAP AG, Sun Microsystems, and webMethods.

[8] S. Gajek, J. Schwenk, and C. Xuan. On the insecurity of Microsoft's identity metasystem. Technical Report TR-HGI-2008-003, Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, June 2008.

[9] International Organization for Standardization, Genève, Switzerland. *ISO/IEC FCD 24760, Information technology — Security techniques — A framework for identity management*, July 2009.

[10] M. B. Jones. A guide to supporting InfoCard v1.0 within web applications and browsers, March 2006. Microsoft Corporation.

[11] A. Jøsang and S. Pope. User centric identity management. In *Proceedings of Australian Computer Emergency Response Team Conference (AusCERT 2005)*, 2005.

[12] Microsoft Corporation. Microsoft's vision for an identity metasystem, May 2005.

[13] Microsoft Corporation. A technical reference for InfoCard v1.0 in windows, August 2005.

[14] Microsoft Corporation. A Guide to Using the Identity Selector Interoperability Profile V1.5 within Web Applications and Browsers, July 2008.

[15] Microsoft Corporation and Ping Identity Corporation. A guide to integrating with InfoCard v1.0, August 2005.

[16] N. Mitra and Y. Lafon (editors). SOAP — version 1.2, April 2007. The World Wide Web Consortium (W3C).

[17] A. Nadalin, M. Goodner, M. Gudgin, A. Barbir, and H. Granqvist (editors). WS-SecurityPolicy — version 1.2, July 2007. OASIS Standard.

[18] A. Nadalin, M. Goodner, M. Gudgin, A. Barbir, and H. Granqvist (editors). WS-Trust — version 1.3, March 2007. OASIS Standard.

[19] A. Nadalin, C. Kaler, R. Monzillo, and P. Hallam-Baker (editors). Web services security: SOAP message security — version 1.1, February 2006. OASIS Standard Specification.

[20] A. Nanda. Identity selector interoperability profile v1.0, April 2007. Microsoft Corporation.

Appendix

A Source code for the proof-of-authenticity method

This appendix contains the source code for the proof-of-concept implementation of the proof-of-authenticity method.

A.1 Initialisation module

```
<?

/* ISSUES
    # 1: if the user deleted the cookie and then tried to
    log-in again, the previous record in the session DB
    will not be deleted and a new session record will
    be created.
*/

$server = "DB server"; $db_user = "DB user"; $db_pass = "DB
password"; $database = "DB name";


########################################################
    function fetch_substr_ip($ip)
    {
     /*
     ipcheck
     0|255.255.255.255
     1|255.255.255.0
     2|255.255.0.0
     */
     $ipcheck = 1;
     return implode('.', array_slice(explode('.', $ip), 0,
     4 - $ipcheck));
    }
##########################################################

    function fetch_ip()
    {
        return $_SERVER['REMOTE_ADDR'];
    }
###########################################################

function vbrand($min, $max) {
    $seed = (double) microtime() * 1000000;
    mt_srand($seed);
    return mt_rand($min, $max);
}

############################################################

Function ReDirectPage($message, $To){
    echo ("
<html> <head> <meta http-equiv=\"refresh\" content=\"6;
URL=$To\"></head> <body> <BR><BR><BR><BR> <div
align=\"center\"><center>
  <table border=\"1\" cellpadding=\"0\" cellspacing=\"0\"
  style=\"border-collapse: collapse\" bordercolor=\"#111111\"
  width=\"80%\"
  id=\"AutoNumber1\" bgcolor=\"$tcolor1\">
```

```
    <tr>
      <td width=\"100%\" align=\"center\">
      <font size=4><B>
        $message
    </font></b></td>
    </tr>
  </table>
  </center>
</div> </body></html>"); } ?>
```

A.2 PoASet module

```
<?php

include('init.php');

    $HOST = $_SERVER['REMOTE_ADDR'];
    $USERAGENT = $_SERVER['HTTP_USER_AGENT'];
    $SESSION_IDHASH = sha1($USERAGENT .
    fetch_substr_ip($HOST));
    $SESSION_HOST       =  substr(fetch_ip(), 0, 15);
    $TIMENOW = time();
#
    $sessionhash = sha1($TIMENOW . $SESSION_IDHASH .
    $SESSION_HOST . vbrand(1, 1000000));
#
      $connection = @mysql_connect($server, $db_user,
      $db_pass) or die("Database Connection Error #
      : 100");
//    $query = "INSERT INTO session ('sessionhash',
//    'userid', 'host', 'idhash', 'lastactivity',
//    'useragent', 'loggedin')
//    VALUES ('$sessionhash', '1','$HOST',
//    $SESSION_IDHASH', '$TIMENOW', $USERAGENT',
//    '$TIMENOW');";
      $query = "INSERT INTO session VALUES
      ('$sessionhash', '1',
      '$HOST', '$SESSION_IDHASH', '$TIMENOW',
      '$USERAGENT', '$TIMENOW');";
//    echo $query. '<BR>';
      @mysql_db_query($database, $query) or
      die("ERROR DB on Create New Session D#101");
      @mysql_close($connection);
#
    setcookie("sessionhash", "$sessionhash", $TIMENOW
    + 3600, '/');
#
    ReDirectPage("Please Wait ...", "Homepage.php");
?>
```

A.3 PoACheck module

```
<?php include('init.php');
 if (!isset($_COOKIE["sessionhash"]))
   {
    echo ('Sorry you cannot use CardSpace this time,
    we have to ask you to enter your username and
    password');
    ReDirectPage($messege, "passwdcheck.php");
   }
   else
    {
     $OLD_sessionhash = $_COOKIE["sessionhash"];
     $HOST = $_SERVER['REMOTE_ADDR'];
     $USERAGENT = $_SERVER['HTTP_USER_AGENT'];
     $SESSION_IDHASH = sha1($USERAGENT .
     fetch_substr_ip($HOST));
     $SESSION_HOST =  substr(fetch_ip(), 0, 15);
     $TIMENOW = time();
#
   $connection = @mysql_connect($server, $db_user,
   $db_pass) or die("Database Connection
   Error #: 100");
   $result = @mysql_db_query($database, "SELECT *
   FROM session WHERE sessionhash =
   '$OLD_sessionhash'") or die("ERROR on Find
   Session DB#102");
   if (@mysql_num_rows($result) > 0){
     $db = mysql_fetch_array($result);
   if ($db[idhash] != $SESSION_IDHASH){
    setcookie("sessionhash", "", time()-3600, '/');
    $messege = "Uncorrect PoA! (Found in DB but from
    different host). Sorry you cannot use
    CardSpace this time, we have to ask you to enter
    your username and password";
    ReDirectPage($messege, "passwdcheck.php");
      }
   else{//Update old session
       #
    $NEW_sessionhash = sha1($TIMENOW . $SESSION_IDHASH .
    $SESSION_HOST . vbrand(1, 1000000));
       #
        $connection = @mysql_connect($server, $db_user,
        $db_pass) or die("Database Connection Error #:
        100");
    //  $query = "INSERT INTO session ('sessionhash',
    //  'userid', 'host', 'idhash', 'lastactivity',
    //  'useragent', 'loggedin') VALUES ('$sessionhash',
    //  '1', '$HOST', '$SESSION_IDHASH', '$TIMENOW',
    //  '$USERAGENT', '$TIMENOW');";
        $query = "UPDATE session SET sessionhash =
        '$NEW_sessionhash',
        lastactivity = '$TIMENOW' WHERE sessionhash =
        '$OLD_sessionhash';";
    //  echo $query;
         @mysql_db_query($database, $query) or
         die("ERROR DB on Update New Session D#101");
         @mysql_close($connection);
         #
           setcookie("sessionhash", "$NEW_sessionhash",
           $TIMENOW + 3600, '/');
         #
           echo "Correct PoA! We will proceed using
           CardSpace...";
           ReDirectPage($messege, "CardSpace.php");
       }}
   else{
        setcookie("sessionhash", "", time()-3600, '/');
        $messege = "Uncorrect PoA! (Not Found in DB).
        Sorry you cannot use CardSpace this time, we have
        to ask you to enter your username and password";
        ReDirectPage($messege, "passwdcheck.php");
   }}?>
```