IBM® Workplace Forms™

Version 2.6.1

Using Authenticated Clickwrap

S325-2611-00

Note

Before using this information and the product it supports, read the information in "Notices," on page 13.

First Edition (September 2006)

This edition applies to version 1, release 2.6.1 of Workplace Forms and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces version 1, release 2.6 of Workplace Forms.

Contents

Implementing Authenticated Clickwrap

Authenticated Clickwrap enables users to securely sign a form without relying on an extended PKI infrastructure. However, implementing an Authenticated Clickwrap system does require an initial investment in architecture and design time. You must first plan the elements of your system, and then implement the necessary components.

This document describes a typical implementation for Authenticated Clickwrap, and discusses the architectural concerns involved. Once you have read this document, you should be ready to plan and implement your own Authenticated Clickwrap system.

Please be aware that this document describes a typical implementation for Authenticated Clickwrap, and should not be taken to represent the only possible implementation. Authenticated Clickwrap allows for a great deal of flexibility in the server-side implementation, but has some universal requirements that are addressed in this document.

About Authenticated Clickwrap

Authenticated Clickwrap offers a way to sign a form that relies on a user ID and a shared secret (typically a password) to identify the signer.

In normal use, the user signs the form by entering an ID and secret. When the form is sent to the server, the server retrieves the user's secret from a database and uses that secret to verify the signature. Furthermore, the server can notarize the Authenticated Clickwrap signature by signing it with a digital certificate, thereby creating a secondary digital signature. This secondary signature shows that the server has confirmed the identity of the signer, and ensures that the original signature can be trusted over time.

This is how Authenticated Clickwrap works at a high level. The details of the technical process are somewhat more complicated, and are explained in the following sections. If you already know the details, you can skip forward to ″Server Architecture″ .

Using the Shared Secret

The security of the secret is key to an Authenticated Clickwrap system. Only the user and the server should have access to an individual secret.

To help ensure the security of the secret, an Authenticated Clickwrap signature never stores the secret in the form. This means that the secret itself is never transmitted in any way.

When the form is signed, the signature information is created by producing a *signature hash* that is based on a hash of the form and the user's secret. While the secret itself is not included in the signature hash, you do require the secret to confirm that the signature is authentic.

When the form is sent to the server, the server reads the signature from the form, checks to see if the form has been tampered with, and gets the user's ID from the signature. The server then looks up the user's secret in a database and, using that secret, creates a new signature hash. Finally, the computer compares the new signature hash of the form to the hash stored in the original signature - they will only match if the user provided the same secret that is stored in the server's database, thereby confirming the user's identity.

Notarizing the Signature

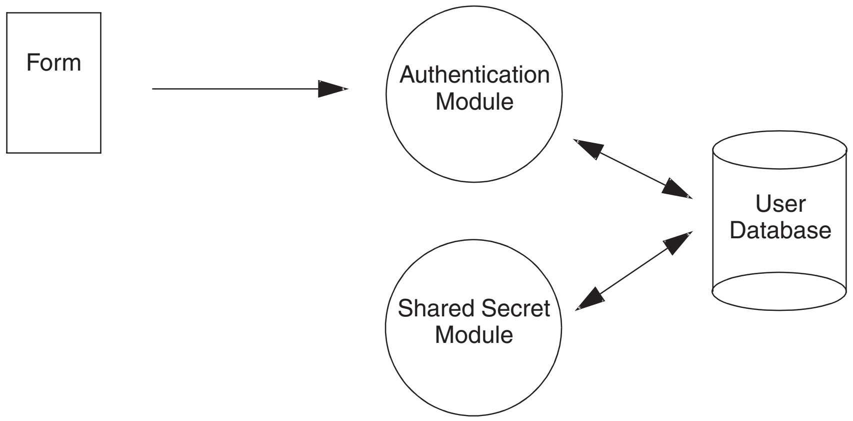
Once the server has verified the user's secret and authenticated the user's identity, the server can notarize the Authenticated Clickwrap signature. To do this, the server simply signs the original signature with a digital signature.

This accomplishes two goals:

- When the signature is verified, the digital signature is checked in addition to the Authenticated Clickwrap signature. Before notarization, the Viewer will show an Authenticated Clickwrap siganture as ″Verified, but Not Authenticated″ because the server has not yet authenticated the signer. Once notarized, the Viewer will show an Authenticated Clickwrap signature as ″Valid″ since the signer has been authenticated. This tells the next user of the form that they can trust the signature to be authentic, and is particularly useful for forms that require overlapping signatures.
- The digital signature will remain valid over time. In contrast, the Authenticated Clickwrap signature relies on the shared secrets stored in the user database. If the secrets change and a history is not maintained, it becomes impossible to verify old signatures. Furthermore, you can notarize an Authenticated Clickwrap signature any number of times, allowing you to renew the notarization if the original digital signature is going to expire.

Server Architecture

A simple Authenticated Clickwrap system uses the following architecture:



As shown, this architecture relies on three central components:

- **User database —** This database stores a table of user IDs and shared secrets.
- **Authentication module —** This module receives forms and authenticates the signatures, using the shared secrets from the user database.
- **Shared secret module —** This module allows users to update their shared secrets in the user database.

How the Architecture Fits into Your Overall Application

The components for Authenticated Clickwrap will likely form part of a larger application. For example, you may want to route the form to the next user in the process or archive the form in a document repository.

You should consider how the form is going to be used when planning your application. For instance, if the form is only going to be signed once, notarized, and then archived, you can build a fairly simple system in which the authentication module forwards the form to the archive.

However, if the form involves overlapping signatures, it's best to notarize the form between each signature. For example, the first user signs the form and the server notarizes that signature, then the second user signs the form and the server notarizes the second signature, and so on. This ensures that each user in the process can trust earlier signatures, since they have been notarized by the server. In this case, you will need the authentication module (or an additional component) to route the form appropriately.

About the User Database

The user database stores a list of user IDs and shared secrets. You can create a new database for use with Authenticated Clickwrap, or you can use existing systems, such as an LDAP server or an existing table of web site permissions.

In general, you can use any system you like to store the user information. However, you should give serious consideration to the security of the system. Authenticated Clickwrap signatures rely completely on the secrecy of the shared secrets. If you store the user information in an open system, or in a system that users cannot update themselves, you will compromise the security of those secrets.

Key Data Elements

The user database should store the following data elements:

- **User ID —** This is a unique ID that identifies the user. Typical IDs may be employee numbers, the user's first initial and last name, the user's email address, and so on.
- **Shared Secret —** This is typically a user password. We recommend that you store a hash of the user's secret rather than the secret itself. This increases the overall security of the system by ensuring that the original secret is never stored anywhere - not on the server and not in the signature. The API accomodates this by providing a function for hashing secrets as well as a function for verifying Authenticated Clickwrap signatures using a hashed secret.
- **Date —** This is the date at which the shared secret became active. We recommend that you keep a history of secrets and their activation dates. This ensures that you can authenticate signatures that were created using old secrets, and is especially useful if you require your users to regularly change their shared secrets.

Using Multiple IDs or Shared Secrets

Authenticated Clickwrap signatures allow for multiple IDs or shared secrets. This is useful for the following reasons:

- Once the form is signed, all IDs will appear in the signature button as comma separated values. For example, if yours users used both their name and email address as IDs, then the button would show:

`<user name>, <email>`

- You may want your users to use multi-part secrets, such as a credit card number and expiration date. This is less confusing if they can enter the numbers in separate fields.

If you decide to use multiple IDs or shared secrets, follow these guidelines:

- Ensure you set up your database properly to store all IDs and shared secrets.
- If you are storing hashed secrets, you must concatenate the secrets and then hash the combined secret. For example, if the secrets were ″blue″ and ″red″, you would store a hash of ″bluered″ in your database.
- Do not use a single signature button for multiple signatures. For example, do not set up a signature button to accept both an employee's and a manager's ID and password. Instead, create two separate signature buttons - one for the employee and one for the manager.

About the Authentication Module

The authentication module verifies and notarizes Authenticated Clickwrap signatures. In most cases, it will also route the form to the next stage of processing. For example, it might send the form to the next user in an approval process, or simply archive the form in a document repository.

You will have to write the authentication module yourself, using IBM® Workplace Forms™ Server - API to access and verify the Authenticated Clickwrap signature. IBM typically uses a servlet architecture for these modules. Furthermore, you should thoroughly understand how to retrieve the shared secrets from the user database, as you will have to write the code that interfaces with the database.

The Basic Algorithm

In general, the authentication module will locate a signature node and then verify and notarize the signature. For example, the following algorithm finds and notarizes a specific signature.

Note: As you must use the API to provide the necessary functionality for reading the form and verifying signatures, each step in the algorithm below also lists the corresponding methods in the Java™ edition of the API.

1. Read the form.
   - *readForm*
2. Locate the signature node.
   - dereference for a known node. Use *getChildren*, *getParent*, *getNext*, *getPrevious*, and *getLiteral* to traverse the form.
3. Check for previous notarization. If previous notarization exists, stop processing. (Note that you may want to take a different action depending on whether the notarizing signature is valid.
   - *getDataByPath* (Signature)
4. Retrieve the ID of the signer from the *signer* option of the signature.
   - *getLiteralByRef*
5. Use the user ID to retrieve the user's secret from the user database.
6. Retrieve the certificate you want to use to notarize the signature.
   - *getEngineCertificateList* retrieves available certificates.
   - *getDataByPath* (Certificate) identifies specific certificates.

7. Verify and notarize the signature.
   - *validateHMACWithSecret*
   - *validateHMACWithHashedSecret*

   Note: For more information about these functions, see "HMAC Signature Validation Functions Quick Reference" on page 7
8. Write the updated form, or route it as necessary.
   - *writeForm*

Depending on your overall application, you may need the authentication module to offer sophisticated routing. You will have to incorporate these features into the module yourself, or write another module to handle this aspect of your application.

Alternate Scenarios

The basic algorithm assumes a simple scenario in which a single signature node is known and is easily located. However, there are other scenarios that require more sophisticated processing. For example:

- **Sequential Signatures —** In this case, the form is viewed and signed by a number of people in turn. As each person signs the form, it goes to the server for processing and notarization, and is then sent on to the next person. Each time the server receives the form, it has to locate each signature button in turn and determine (a) whether it has a signature, (b) whether the signature has already been notarized, and (c) whether the notarization is valid. In each case, the server must act accordingly. For instance, if there is a notarizing signature but that signature is no longer valid, the server may need to throw an error or otherwise notify a system administrator.
- **Multiple Notarizations —** You may find it useful to add multiple notarizations to a signature. For example, you may want to notarize existing signatures on an annual basis to prevent the digital signature from expiring with the certificate that created it. In this case, your server module will have to parse each form to locate notarized signatures, verify that the last notarization is still valid, and then notarize the signature again.

Regardless of your specific needs, you should be able to adjust the basic algorithm accordingly.

About the Shared Secret Module

The shared secret module allows users to connect to the user database and change their shared secret. If you are using an existing system, such as an LDAP server or a web site permissions table, you may already have a module like this in place.

If you do not, you will have to write the shared secret module yourself, and should have a thorough understanding of your user database. IBM typically uses a servlet architecture for these modules.

Regardless of your particular implementation, we recommend that you hash each secret before storing it in the database. This is more secure, as it ensures that the original secret is never stored anywhere. The API provides a hashing function that you can use for this purpose, as well as a function that allows you to use the hashed secret to verify a signature.

Note: If your Authenticated Clickwrap signature uses more than one shared secret, you must concatenate the secrets, hash them, and then store the hash of the combined secret in your database. For example, if the secrets were ″blue″ and ″red″, you would store the hash of ″bluered″ in your database.

HMAC Signature Validation Functions Quick Reference

validateHMACWithHashedSecret

Description

This method determines whether an HMAC signature is valid. HMAC signatures include both Authenticated Clickwrap and Signature Pad signatures.

For Authenticated Clickwrap signatures, you must know the hash of the signer's shared secret to use this method. For Signature Pad signatures, you may use this method without the shared secret if the signature was created without one. In any case, the shared secret should be available from a corporate database or other system.

This method will also notarize (that is, digitally sign) a valid HMAC signature if you provide a digital certificate. However, notarization will not occur if the signature does not include a shared secret. Once notarized, you must use the **verifySignature** method to validate the signature.

Note: Authenticated Clickwrap is a separately licensed product. Please ensure that your company has the license to use Authenticated Clickwrap before you provide forms or functionality that rely on it.

Method

```
public short validateHMACWithHashedSecret(
   byte [] hashedSecret,
   Certificate theServerCert,
   IntHolder theStatus,
   ) throws UWIException;
```

Parameters

Expression	Type	Description
hashedSecret	**byte[]**	The hash of the shared secret that identifies the user. This should be available from a corporate database or other system. If there is more than one shared secret, you must concatenate the strings with no separating characters and then hash the combined secret. For example, if the secrets were ″blue″ and ″red″, you would pass the hash of ″bluered″ to the method. If there is no shared secret, pass and empty string. You must encode the byte array as follows: **Authenticated Clickwrap (HMAC)** UTF-8 **Signature Pad** UTF-16LE You can use the getBytes method to do this. For example: `mySecret.getBytes("UTF-8")`
theCertificate	**Certificate**	The server certificate. If the HMAC signature is valid, the function will use the private key of this certificate to digitally sign the HMAC signature. This signature is appended to the signature item, and can be verified using UFLVerifySignature. If you pass null, the method will simply validate the HMAC signature.
theStatus	**IntHolder**	This is a status flag that reports whether the operation was successful. Possible values are: **SecurityUserStatusType.SUSTATUS_OK** — the operation was successful. **SecurityUserStatusType.SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SecurityUserStatusType.SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).

Returns

A constant if the verification is successful, or throws a generic exception (**UWIException**) if an error occurs. The following table lists the possible return values:

Code	Numeric Value	Status
FormNodeP.UFL_DS_OK	0	The signature is verified.

Code	Numeric Value	Status
FormNodeP.UFL_DS_ALGORITHM UNAVAILABLE	13590	The appropriate verification engine for the signature is not available.
FormNodeP.UFL_DS_F2MATCHSIGNER	13529	The certificate does not match the signer's name.
FormNodeP.UFL_DS_FAILED AUTHENTICATION	1272	The signature is invalid or the secret used is incorrect.
FormNodeP.UFL_DS_HASHCOMPFAILED	13527	The document has been tampered with.
FormNodeP.UFL_DS_NOSIGNATURE	13526	There is no signature.
FormNodeP.UFL_DS_NOT AUTHENTICATED	1240	The signer cannot be authenticated.
FormNodeP.UFL_DS_UNEXPECTED	13589	An unexpected error occurred.
FormNodeP.UFL_DS_UNVERIFIABLE	859	The signature cannot be verified.

Example

The following example uses **getSignature** to get a signature object, and uses **getDataByPath** to get the signer's identity from the signature object. Next, it calls **validateHMACWithHashedSecret** to validate the signature.

```
public short checkSignature(FormNodeP theSignatureNode, Certificate theServerCert)
{
Signature theSignatureObject;
byte [] hashedSecret;
String signerCommonName;
BooleanHolder encodedData;
IntHolder theStatus;
short validation;

   theSignatureObject = theSignatureNode.getSignature();
   encodedData = new BooleanHolder();
   if ((signerCommonName = theSignatureObject.getDataByPath(
      "SigningCert: Subject: CN", false, encodedData)) == null)
   {
      throw new UWIException("Could not determine signer's name.");
   }

   /* Include external code that matches the signer's identity to a hashed
      shared secret, sets hashedSecret to match.  This is most likely a
      database lookup. */

      theStatus = new IntHolder();

   validation = theSignatureNode.validateHMACWithHashedSecret(
      hashedSecret, theServerCert, theStatus);

   /* Check the status in case the process required user input. */

   if (theStatus.value != SecurityUserStatusType.SUSTATUS_OK)
   {
      throw new UWIException("Validation required user input.");
   }
   return(validation);
}
```

validateHMACWithSecret

Description

This method determines whether an HMAC signature is valid. HMAC signatures include both Authenticated Clickwrap and Signature Pad signatures.

For Authenticated Clickwrap signatures, you must know the signer's shared secret to use this method. For Signature Pad signatures, you may use this method without the shared secret if the signature was created without one. In any case, the shared secret should be available from a corporate database or other system.

This method will also notarize (that is, digitally sign) a valid HMAC signature if you provide a digital certificate. However, notarization will not occur if the signature does not include a shared secret. Once notarized, you must use the **verifySignature** method to validate the signature.

Note: Authenticated Clickwrap is a separately licensed product. Please ensure that your company has the license to use Authenticated Clickwrap before you provide forms or functionality that rely on it.

Method

```
public short validateHMACWithSecret(
   String theSecret,
   Certificate theServerCert,
   IntHolder theStatus,
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theSecret	**String**	The shared secret that identifies the user. This should be available from a corporate database or other system. If there is more than one shared secret, you must concatenate the strings with no separating characters. For example, if the secrets were ″blue″ and ″red″, you would pass ″bluered″ to the method. If there is no shared secret pass an empty string.
theServerCert	**Certificate**	The server certificate. If the HMAC signature is valid, the method will use the private key of this certificate to digitally sign the HMAC signature. This signature is appended to the signature item, and can be verified using verifySignature. If you pass null, the method will simply validate the HMAC signature.

Expression	Type	Description
theStatus	**IntHolder**	This is a status flag that reports whether the operation was successful. Possible values are: **SecurityUserStatusType.SUSTATUS_OK** — the operation was successful. **SecurityUserStatusType.SUSTATUS_ CANCELLED** — the operation was cancelled by the user. **SecurityUserStatusType.SUSTATUS_INPUT_ REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).

Returns

A constant if the verification is successful, or throws a generic exception (**UWIException**) if an error occurs. The following table lists the possible return values:

Code	Numeric Value	Status
FormNodeP.UFL_DS_OK	0	The signature is verified.
FormNodeP.UFL_DS_ALGORITHM UNAVAILABLE	13590	The appropriate verification engine for the signature is not available.
FormNodeP.UFL_DS_F2MATCHSIGNER	13529	The certificate does not match the signer's name.
FormNodeP.UFL_DS_FAILED AUTHENTICATION	1272	The signature is invalid or the secret used is incorrect.
FormNodeP.UFL_DS_HASHCOMPFAILED	13527	The document has been tampered with.
FormNodeP.UFL_DS_NOSIGNATURE	13526	There is no signature.
FormNodeP.UFL_DS_NOTAUTHENTICATED	1240	The signer cannot be authenticated.
FormNodeP.UFL_DS_UNEXPECTED	13589	An unexpected error occurred.
FormNodeP.UFL_DS_UNVERIFIABLE	859	The signature cannot be verified.

Example

The following example uses **getSignature** to get the signature object, and uses **getDataByPath** to get the signer's identity from the signature object. It then calls **validateHMACWithSecret** to validate the signature.

```
public short checkSignature(FormNodeP theSignatureNode, Certificate theServerCert)
{
Signature theSignatureObject;
String theSecret;
String signerCommonName;
BooleanHolder encodedData;
IntHolder theStatus;
short validation;
```

```
   theSignatureObject = theSignatureNode.getSignature();
   encodedData = new BooleanHolder();
   if ((signerCommonName = theSignatureObject.getDataByPath(
      "SigningCert: Subject: CN", false, encodedData)) == null)
   {
      throw new UWIException("Could not determine signer's name.");
   }

   /* Include external code that matches the signer's identity to a shared
      secret, and sets theSecret to match.  This is most likely a
      database lookup. */

   theStatus = new IntHolder();

   validation = theSignatureNode.validateHMACWithSecret(theSecret,
      theServerCert, theStatus);

   /* Check the status in case the process required user input. */

   if (theStatus.value != SecurityUserStatusType.SUSTATUS_OK)
   {
      throw new UWIException("Validation required user input.");
   }
   return(validation);
}
```

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Trademarks

Program Number:

Printed in USA

S325-2611-00