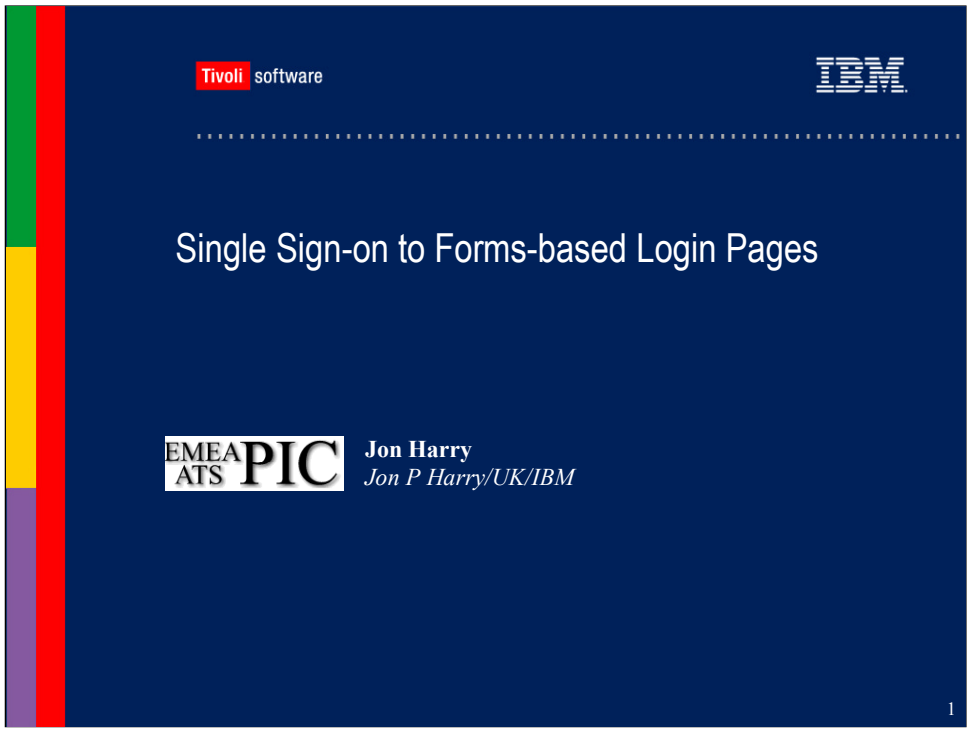
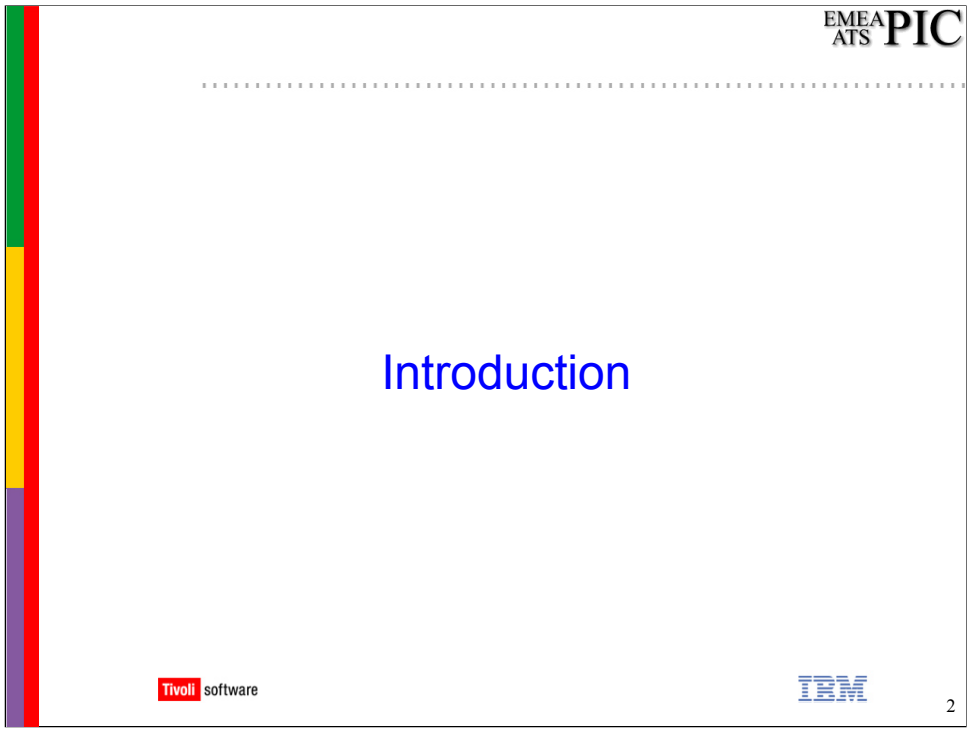
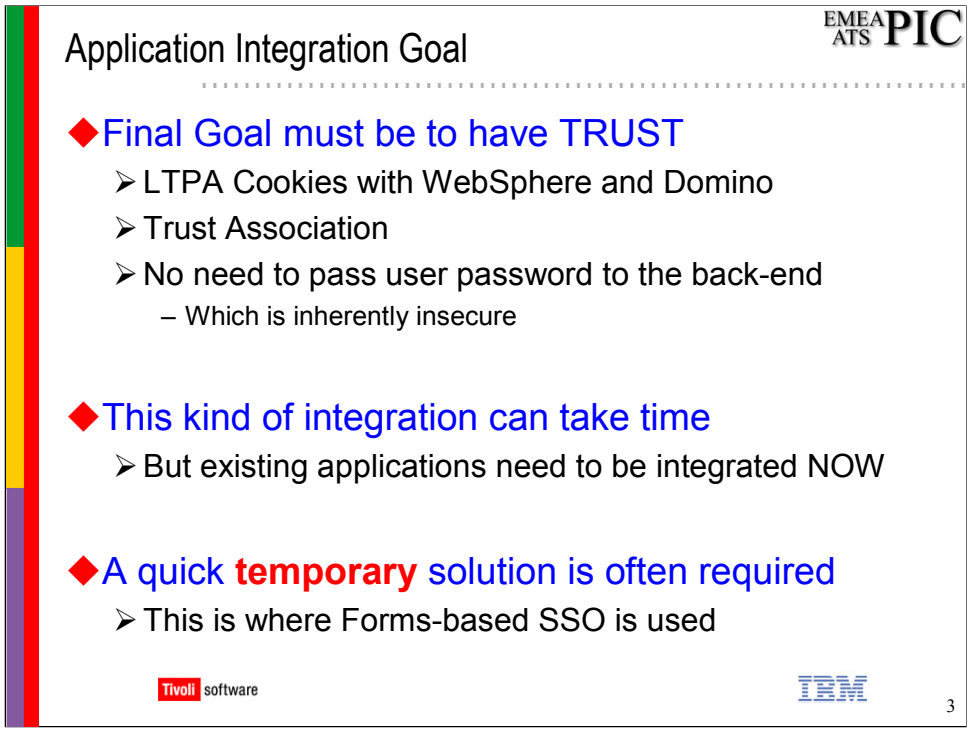


This presentation describes the new functionality implemented in AM v3.9 that allows WebSEAL to automatically sign users into a junctioned back end server that is using form-based authentication.

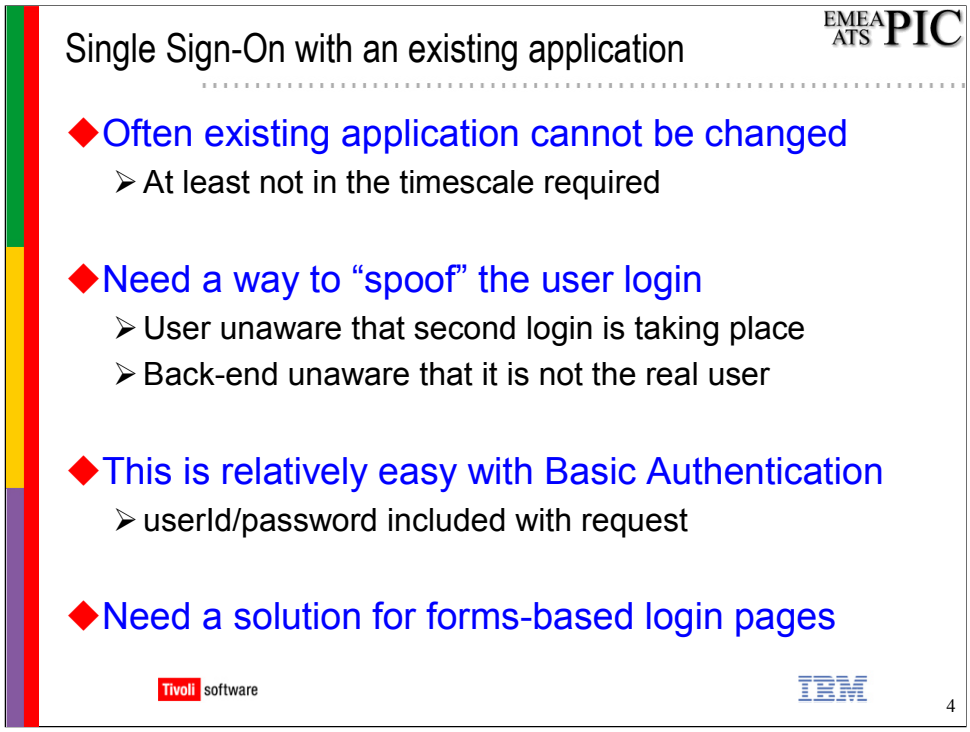

EMEA ATS PIC
Introduction
Tivoli software
IBM
2



Form-based single sign-on should not be viewed as a way to remove the necessity to fully integrate WebSEAL and back-end applications using some form of trust basis. It should be viewed as a way to allow a quick integration solution that can be used while something better is being developed and tested.

Form-based single sign-on (and other ways for WebSEAL to fool a backend server that a user has logged on) are inefficient because they require both WebSEAL and the back-end server to perform authentication for every user. In many cases WebSEAL and the backend server have different user registries which means that two sets of authentication information need to be maintained and synchronised.

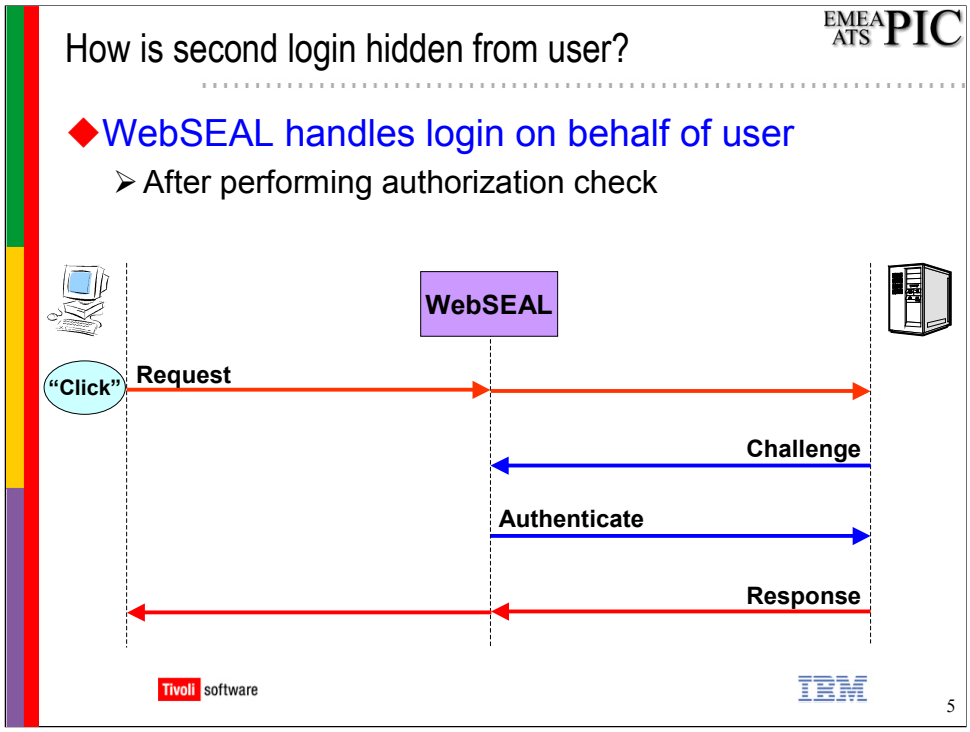
Auto login techniques also require that users password information for the backend server is stored in such a way that it can be retrieved in clear text in order to submit it to the backend server on behalf of the user. This is not an ideal situation.



Forms-based single sign-on is intended for use with existing applications that use form based authentication and cannot be modified to directly trust the authentication done by WebSEAL.

These applications expect users to authenticate via a login form that is presented on the users browser, filled in by the user and submitted. We need some way to interrupt this process so that the login form can be submitted by WebSEAL on the users behalf without them ever seeing it.

In order to achieve this WebSEAL must recognise when a login form is being presented and be able to interpret it and respond accordingly.



The diagram above shows the general principle behind how single sign-on is achieved.

A user, who has already successfully authenticated to WebSEAL attempts to access some resource on a backend server.

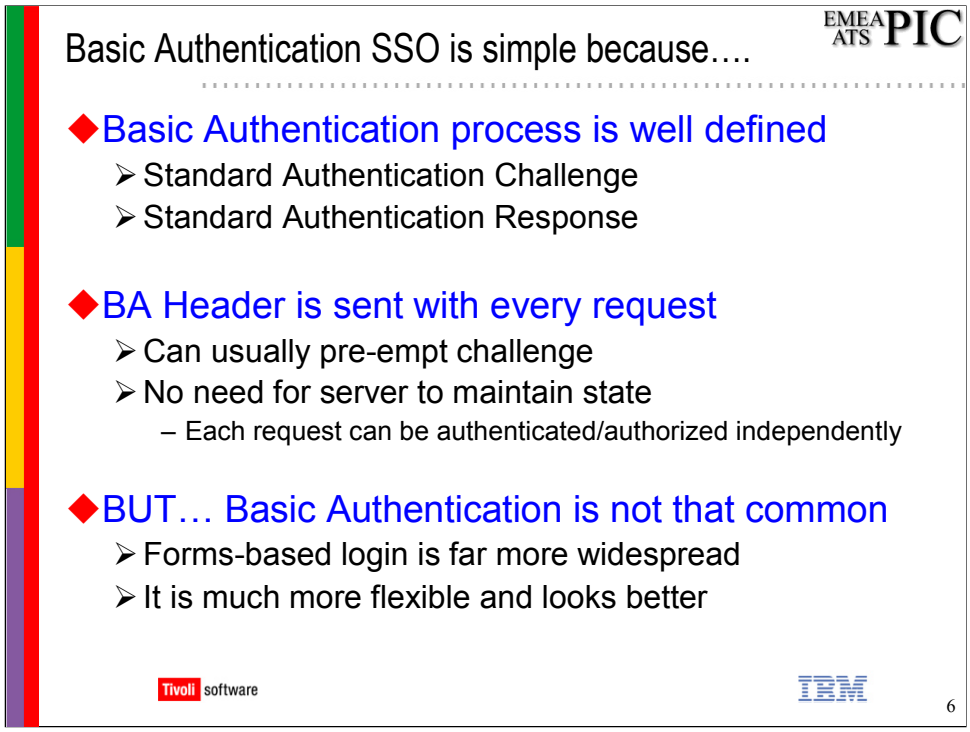
The back end server configuration specifies that the resource being requested requires it to authenticate the user and authorize the transaction and so sends an authentication challenge to the user.

WebSEAL recognises the authentication challenge and responds on behalf of the user. This requires that WebSEAL knows the appropriate response to the challenge and has the necessary information.

The back end server is unaware that the authentication information has come from WebSEAL rather than the end user and performs an authentication as it would for any other user.

Assuming that the authentication is successful the back end server then authorizes the transaction and sends a response which WebSEAL forwards to the user.

Note that the user only sees a single exchange – they make a request and the response is returned.

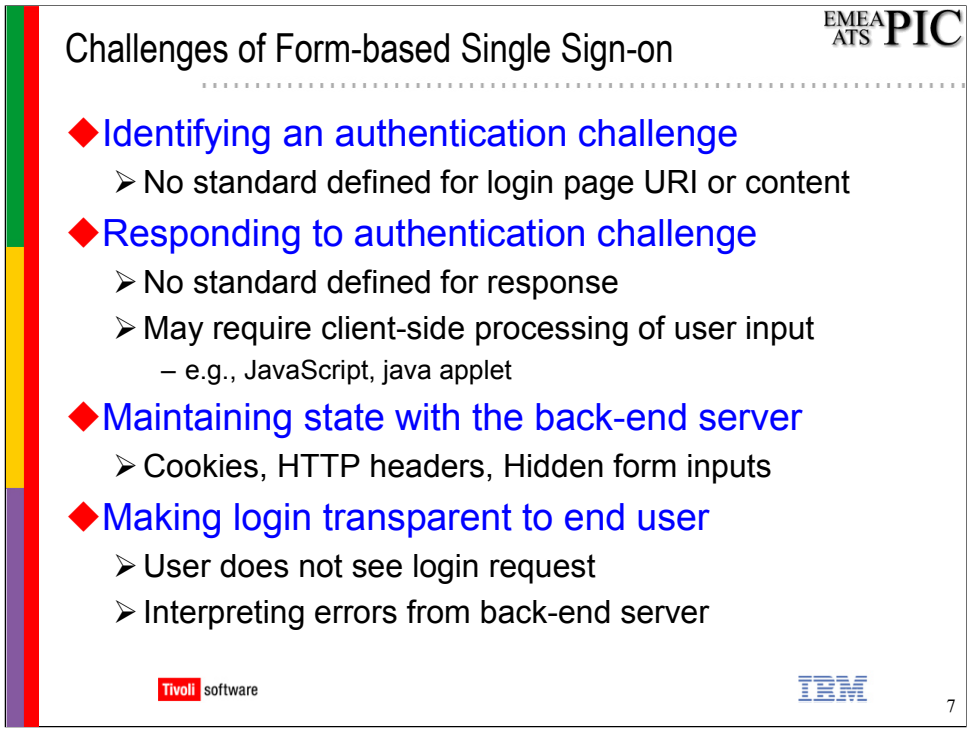


WebSEAL has been able to automatically log users into servers that use Basic Authentication for a long time. It's worth noting why this is an easier to accomplish than logging a user onto a form-based authentication system.

The Basic Authentication exchange is defined in the HTTP standard and so there is little ambiguity in how the server should initiate an authentication challenge or how the client should respond. Also, the Basic Authentication exchange does not allow state information (such as cookies) during authentication . This means WebSEAL does not have to be concerned about maintaining state with the backend server while it is performing the login.

Although performing single sign-on with Basic Authentication is relatively easy, most servers choose not to use it as a means of authentication – primarily because the User Interface is un-sophisticated and set by the browser, and also because it is limited to a single host-name.

Forms-based authentication is much more widely used because it is more flexible. Unfortunately it is this very flexibility that makes single sign-on to forms-based systems so much more difficult to handle – as we will see.

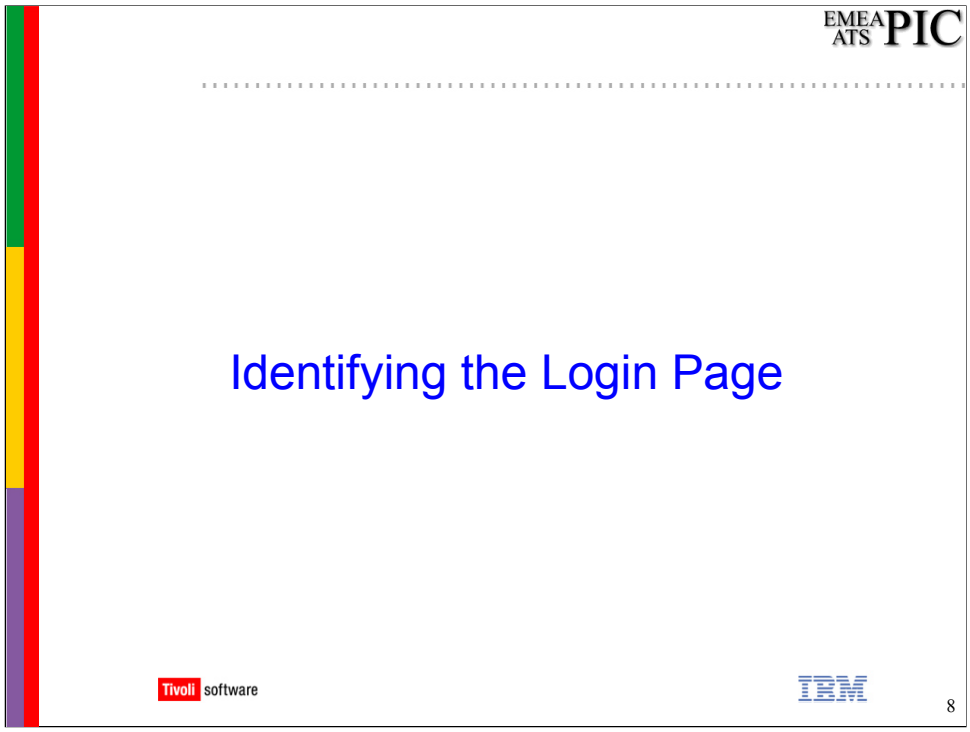


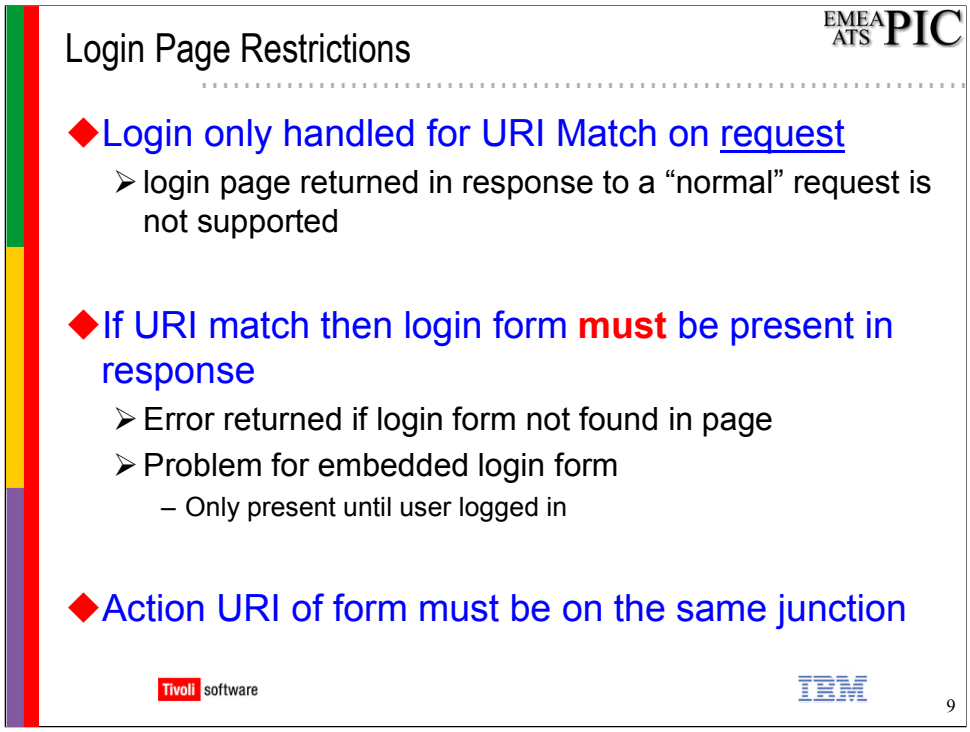
There are two main parts to form-based single sign-on.

The first is the ability to identify when the back end server sends a login page. This is harder than it sounds because a login page is not a special page – it is just like any other HTML page that has a form on it. WebSEAL could examine every page returned by the back-end to determine if it contains a login form – the trouble is that this would be very inefficient because it would require a text search of every page. A more efficient way is to watch for a request for the login page (/login.html for example) from the browser and then intercept the response knowing that it will contain a login form.

Once a page known to contain a login form is received (and the login form identified within the page) the second part of form-based single sign-on is determining how to correctly respond to the server. The fields on the form (usually username and password) must be completed and then the form action executed. In addition, hidden form fields must be returned and so must the appropriate cookies and HTTP headers that the server will expect.

Once the automatic login has been completed the session must be returned to the end user browser – hopefully with no sign that the automatic login has taken place.


EMEA ATS PIC
Identifying the Login Page
Tivoli software
IBM
8

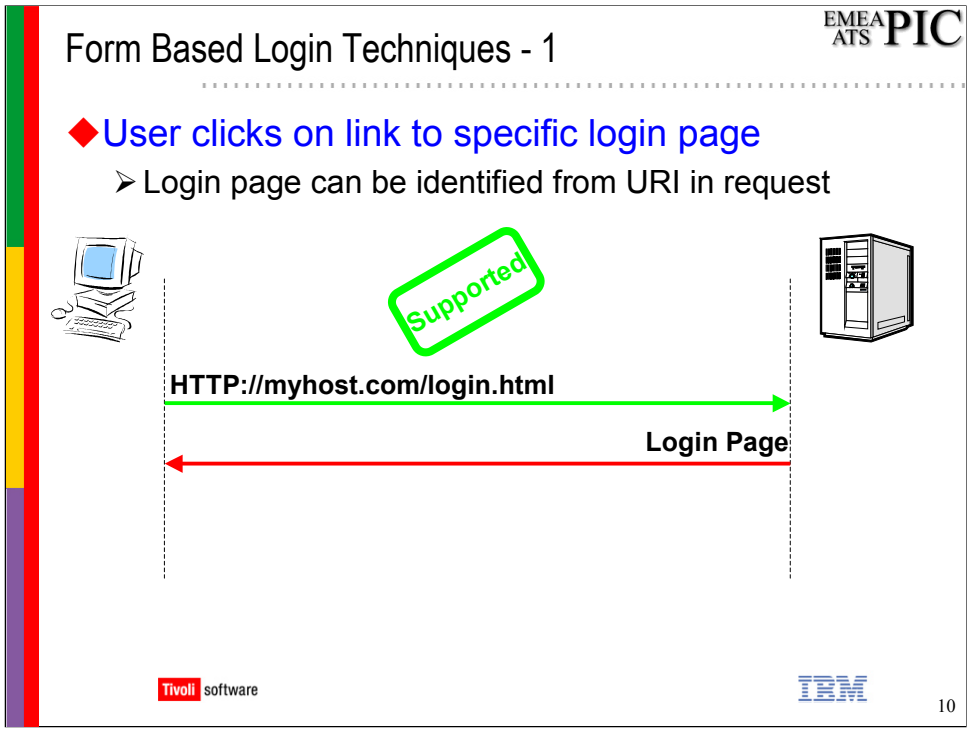


In order to maintain acceptable performance of WebSEAL in a form-based SSO environment the restrictions shown above have been imposed:

It is impractical for WebSEAL to search every received page to check whether it contains a login form that needs to be filled in. As a result, WebSEAL will only initiate its SSO functionality if it can determine from the HTTP request from the browser that the response will contain a login form. If the user request does not provide a match then any login page send will not be dealt with – it will be forwarded to the user as a normal page.

Once the SSO function has been triggered then WebSEAL will expect to find a login form in the response. If it does not then this is considered an error and an error page will be returned to the user.

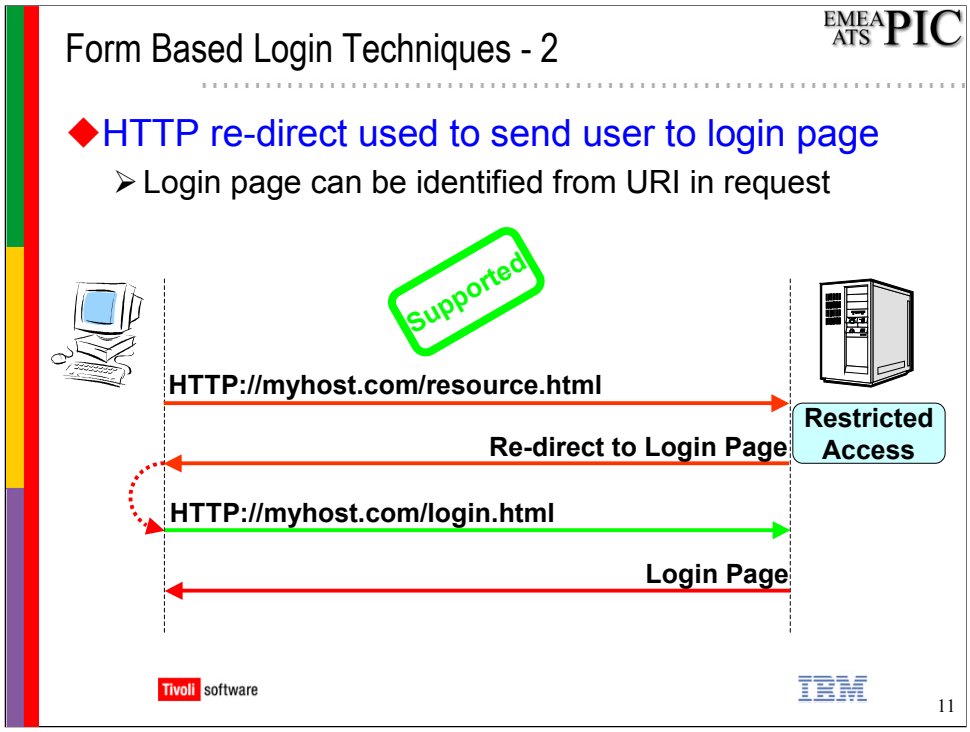
Finally, because form-based SSO is handled on a per junction basis, the login page and action URI must be on the same junctioned server.



The diagram above shows the simplest way that a form-based login can be initiated. In order to sign in the user clicks on a link to a specific login page and the server responds with the page containing the login form.

It is simple for WebSEAL to support this environment. When the user clicks on the "Login" link WebSEAL recognises the request for the login page and starts its SSO functionality. It requests the login page from the back end server and when the login page is returned is intercepted and dealt with on the users behalf.

Note that in this environment the user may still have to click on a "Login" link to initiate the single sign-on. Of course a link on another page could connect the user to the login page without the user being aware that they are being sent to a login page (which they won't see anyway).

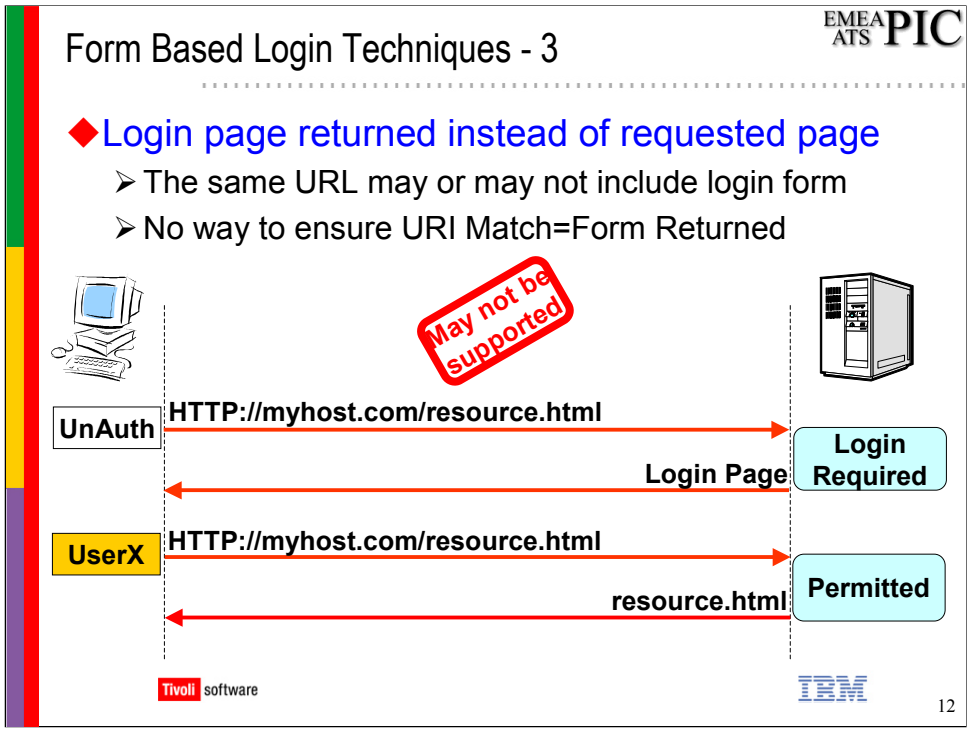


Another way that a backend server might implement its form-based single sign-on is to re-direct the user to a specific login page whenever they attempt to access a resource that is not available to an unauthenticated user.

WebSEAL can also support this environment with no problem because it is really the same as the previous example except that the request for the login page is initiated by a re-direct instead of by the user clicking on a link. WebSEAL will recognise the request for the login page and initiate the form-based SSO function. This will request the login page from the back and the deal with it on the users behalf.

Once the login process has been completed the back-end server will normally re-direct the user back to the original resource they requested. This re-direct is forwarded back to the browser and causes a new request for the resource to be made and the user is unaware that the login took place.

This implementation of form-based login is really the best for use with single sign-on because the user is unaware of the login process – they don't even have to click on a special link.



The diagram above shows an example of a form-based login implementation that WebSEAL may not always be able to handle.

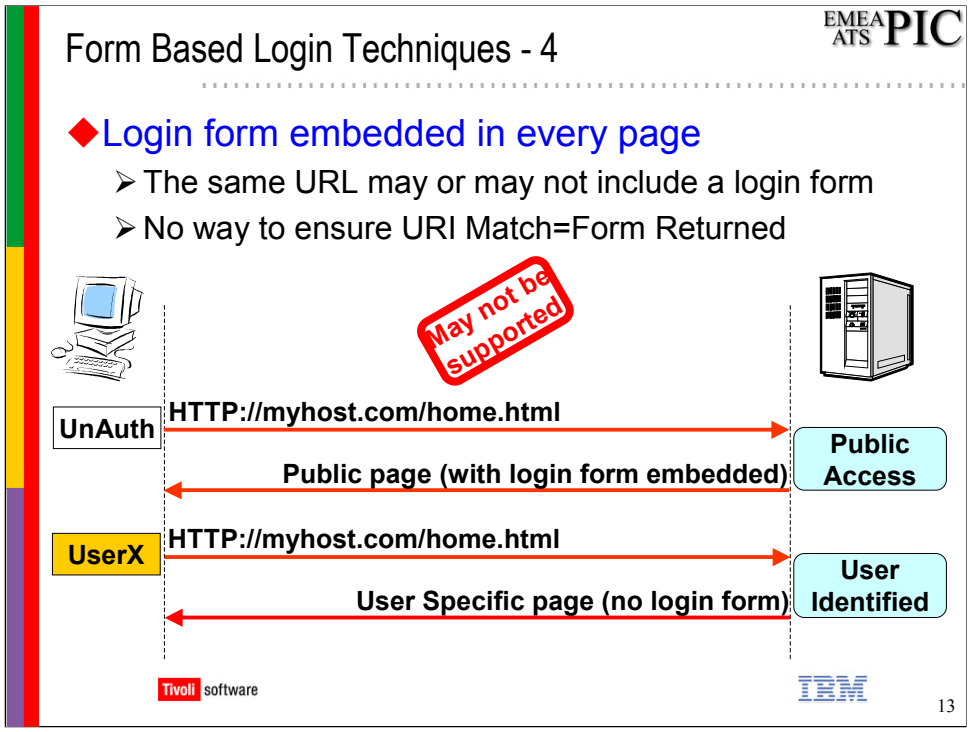
In this case the Web server does not return a re-direct to the login page in response to an unauthenticated request it simply returns the login form instead of the requested resource. This is known as “in-line” authentication and is the way that WebSEAL itself functions.

The important thing to notice in the diagram is that two requests are being made **for the same URI** but one results in a login page being returned and the other does not.

If WebSEAL were configured to match on this URI then the login would be handled successfully but then an error would be returned when the same URI is matched but the response does not include a login page.

If WebSEAL were configured to NOT match on this URI then the login would not be handled because the forms-based SSO code would not be triggered.

The only way around this problem is to find a way to ensure that the SSO code is triggered once the first time the user needs to access the site but is not triggered by subsequent requests to the same site. This is discussed later.



The diagram above shows a second example of a form-based login implementation that WebSEAL may not always be able to handle.

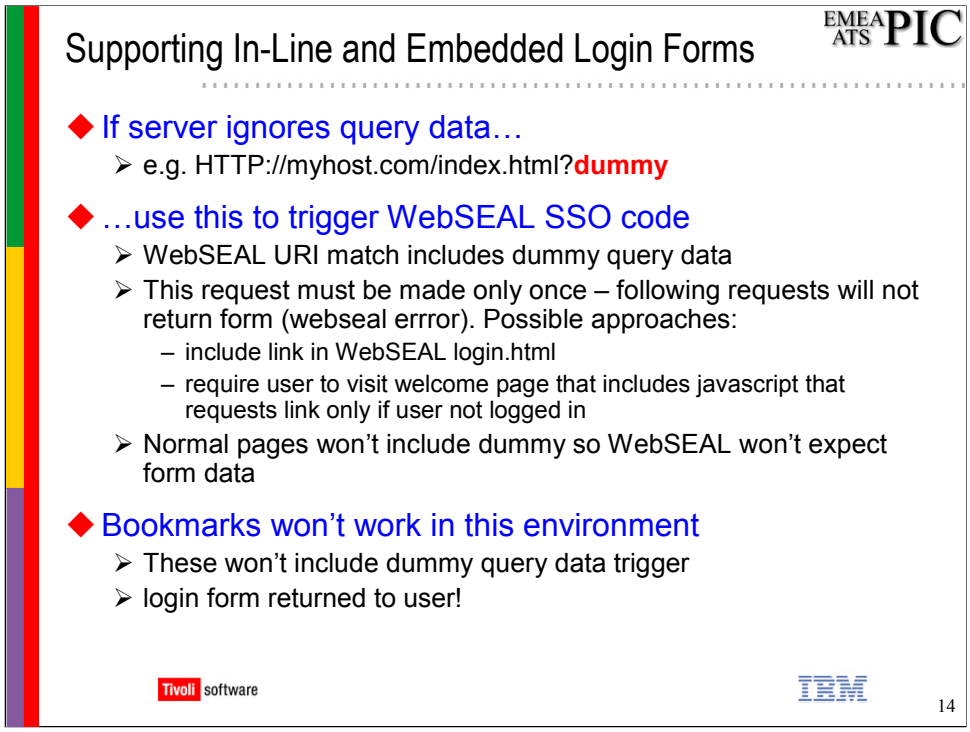
In this case the web server includes a login form with every page that is sent to the user until they authenticate. Once the user has authenticated the login page is no longer sent.

Again, the important thing to notice in the diagram is that two requests are being made **for the same URI** but one results in a login form being returned and the other does not.

If WebSEAL were configured to match on this URI then the user would be logged in but then an error would be returned when the same URI is matched a second time but the response does not include a login page.

If WebSEAL were configured to NOT match on this URI then the login would not be handled because the forms-based SSO code would not be triggered.

The only way around this problem is to find a way to ensure that the SSO code is triggered once the first time the user needs to access the site but is not triggered by subsequent requests to the same site. This is discussed later.



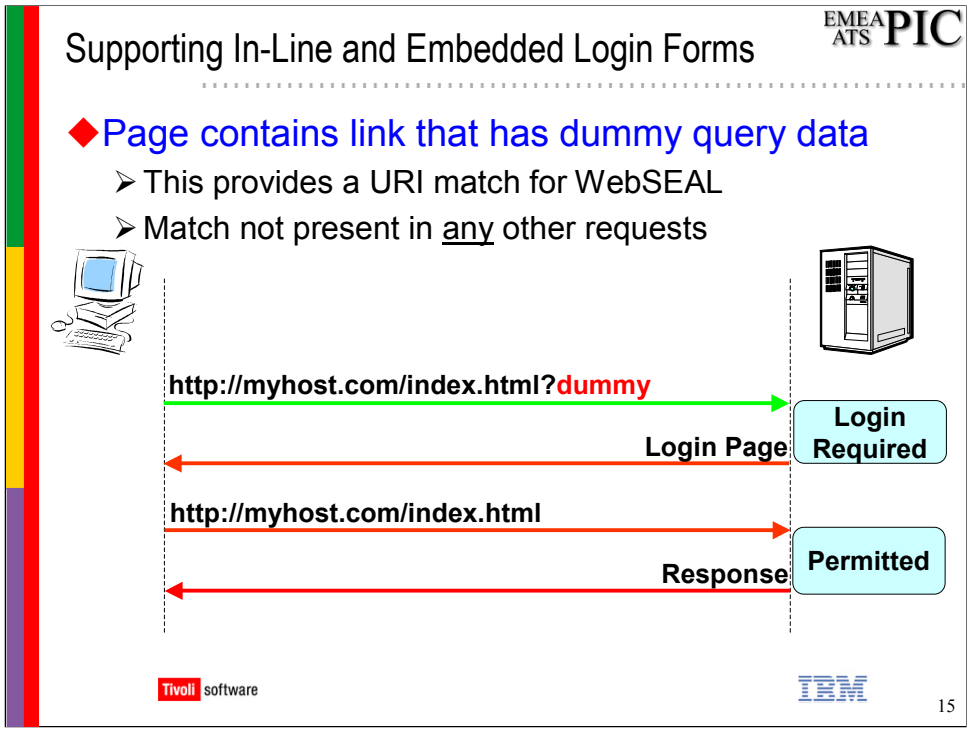
It may be possible to support environments where the same URI sometimes results in a login form and sometimes does not by changing the match that triggers the SSO code so that its initiation can be more precisely controlled.

An example of this workaround is to add some dummy text to the SSO trigger. This dummy text will not be included in "normal" requests to the back end and so the SSO code will not be initiated.

For this to work, the "dummy text" link must only be requested once (because subsequent requests will not return the login form which is webseal error case).

This may be accomplished by including this dummy-link text on the webseal login form (login.html). ch limiting the dummy-text linke to trigger the SSO code the user must click on a link (on some other page) that DOES contain the dummy text. This will trigger the SSO code and, assuming that back-end ignores the dummy text, the login will be successful.

The downside of using this technique is that the user MUST click on the link to access the back-end server for the first time in each session. If they access in some other way (going direct to other pages from a bookmark for example) then they will not be logged in and will see a login form. This might be a problem if they don't know the password for manually logging into the backend server.



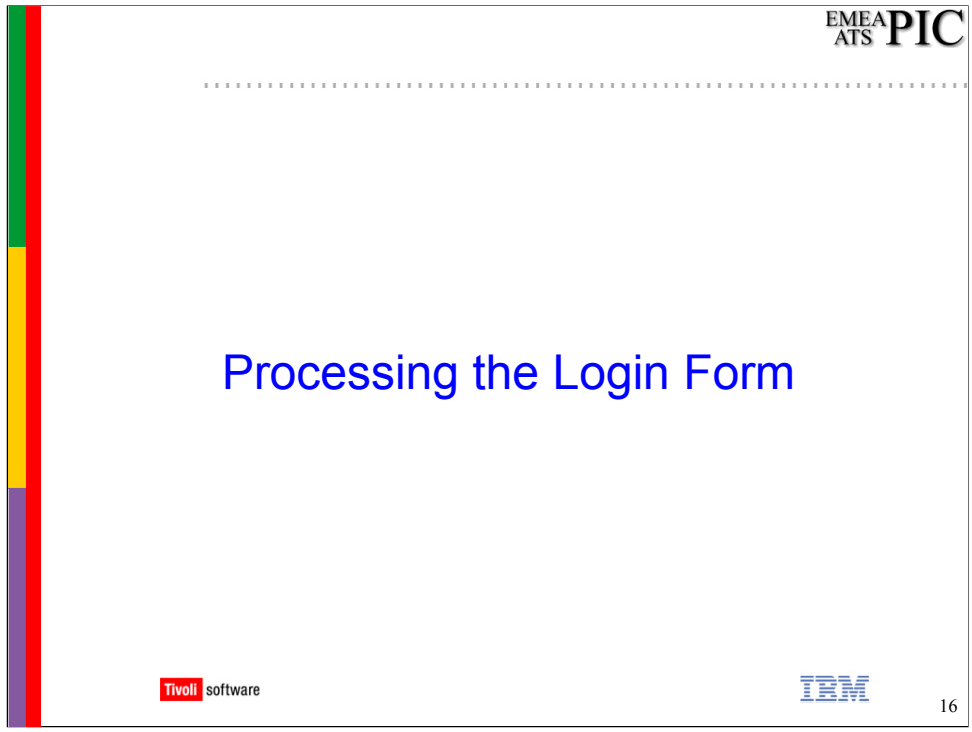
The diagram above shows how the triggering of the SSO function can be controlled by modifying the match required.

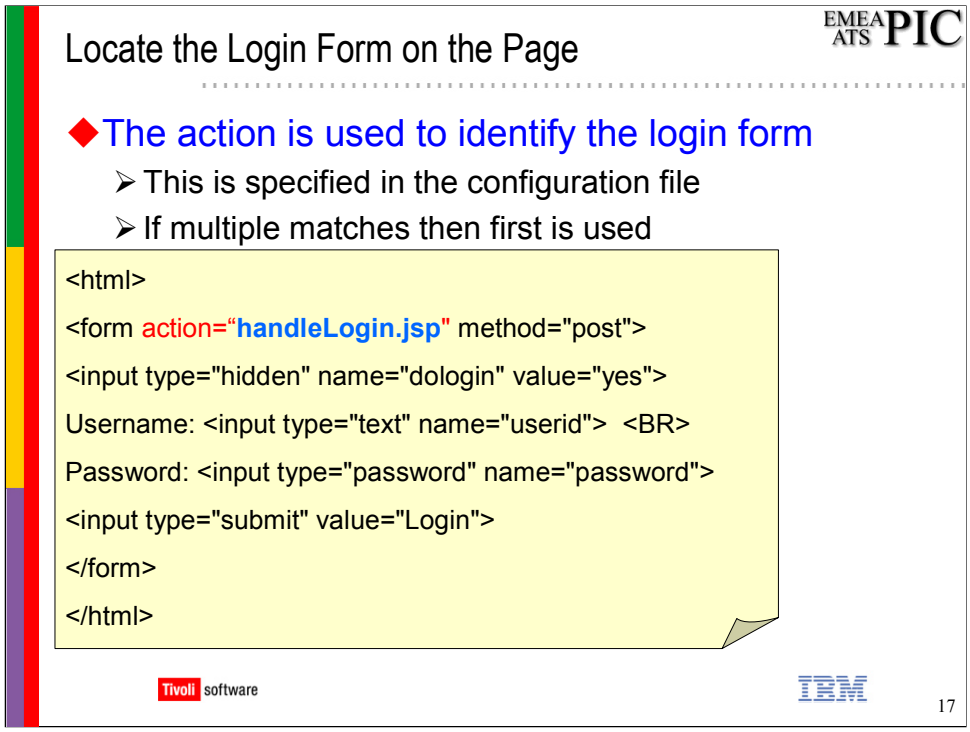
In the example above WebSEAL could be configured so that the SSO code is only invoked if /index.html**?dummy** is requested.

As long as the first page requested from the back-end server (which will result in a login page) includes the **?dummy** extension then the login page returned will be handled.

This could be achieved by including this **?dummy** link on the WebSEAL login page. All other Subsequent requests for the same resource (which will not result in a login page) should not include the **?dummy** extension.

As mentioned previously, this technique will fail if the user requests a page on the back-end server that they have book-marked. This will result in a login page but the SSO code will not be triggered and so the user will receive the login page. This may be acceptable if they can log in manually but this may not be possible if they don't have the UserID and password required.


EMEA ATS PIC
Processing the Login Form
Tivoli software
IBM
16

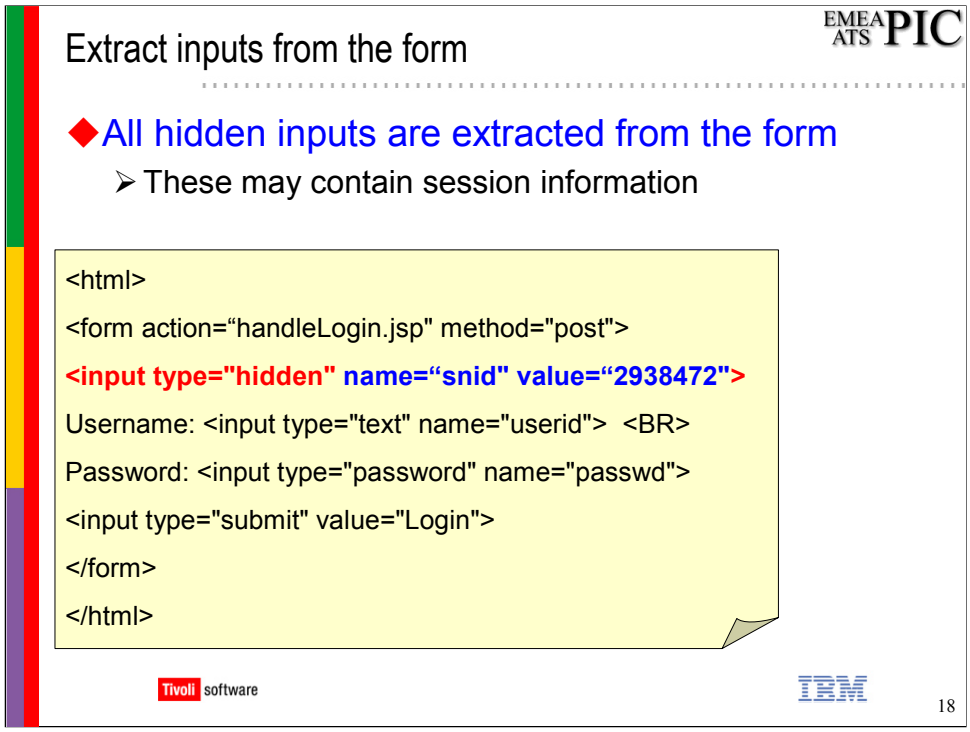


Once the Form-based SSO code has been triggered, and a page returned by the back-end server, WebSEAL must next locate the login form on the page. This step is required because the page returned may contain several forms.

The text extract above shows how a single form is described in HTML – there may be a number of these on a page. The *form* tag includes a number of attributes – one of which is the *action* attribute. This attribute specifies where the form data should be sent when the form is submitted. WebSEALs configuration matches this attribute to identify which form it should process.

If several forms on the page have an action that matches what is configured in WebSEAL then the first will be used.

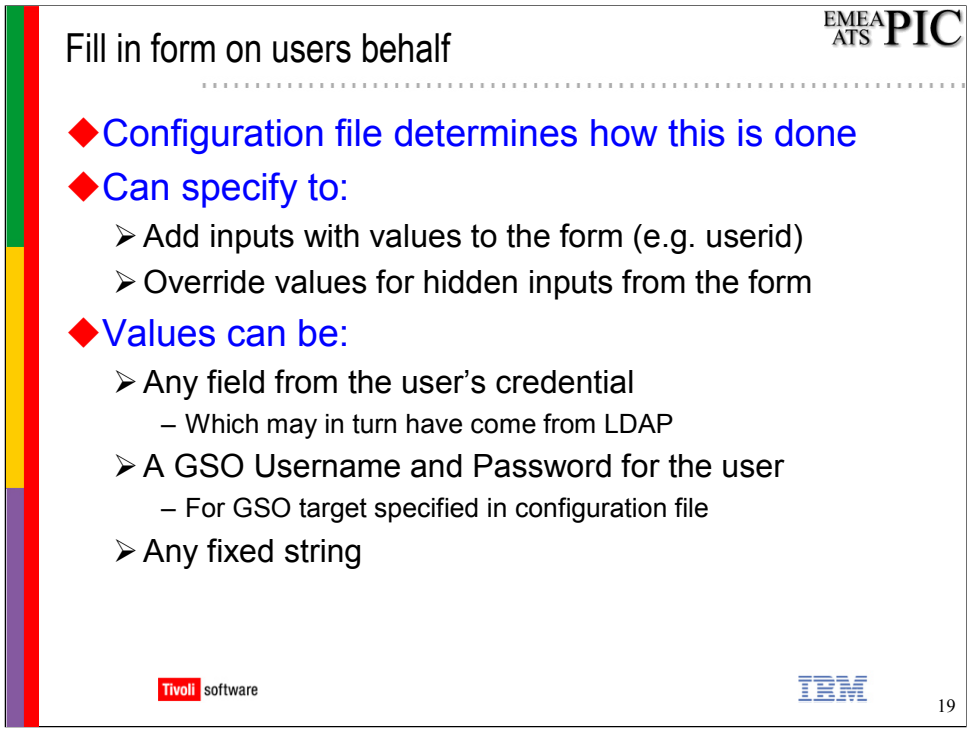
Note that the action shown above is a relative URL. This does not have to be the case – it may be an absolute URL or a server relative URL. WebSEAL can handle all of these, but in the case of an absolute URL the server must be the same server that served the login page (and therefore on the same junction) because form based-SSO works on a per junction basis.



Once the login form has been identified, WebSEAL must extract any information contained in hidden inputs. This information is likely to be required by the back-end server in order to process the login.

Hidden inputs often contain session information, that allow the server to link the initial login request with the users response, or something that will trigger the login processing then user submits the form.

WebSEAL examines all of the *input* tags and stores the name and value any that have a *type* attribute of *hidden*. All of these inputs will be sent to the back-end server by WebSEAL along with the inputs specified in the Form-based SSO configuration.



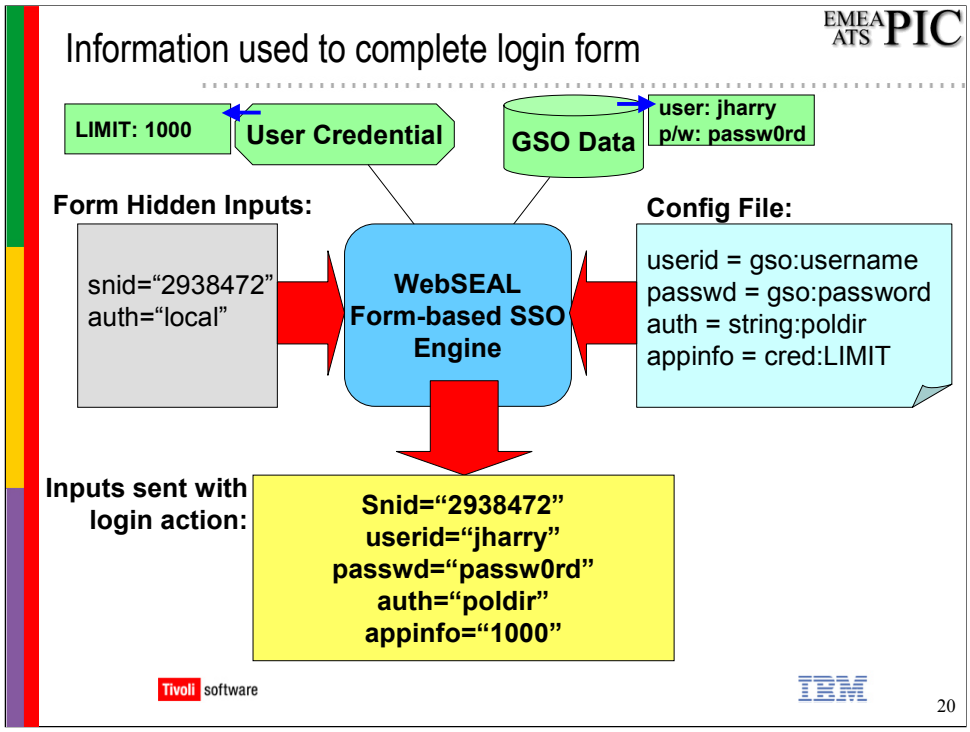
The final part of the forms-based single sign-on process is to fill in the form on the users behalf and submit it to the back-end server.

The SSO configuration for the login page specifies which inputs need to be added to the form for the authentication to succeed at the server – for example inputs that contain username and password. It is also possible to override the value of hidden fields – although this is not usually necessary.

In addition to setting the input values to a fixed string, WebSEAL can also get information to put into the inputs from:

Users Credential: Any parameter from the users credential, AM generated (e.g. *azn_cred_principal_name*) or added by a CDAS, can be extracted and put into the form. If the tag-value support has been used to store information from LDAP in a user credential then this could be used to complete the form. These attributes are prefixed with '*tagvalue_*' in the credential.

GSO Information: Part of the forms-based SSO configuration may specify a GSO target. If this is done then the current users resource UserID and password for that target can be used.

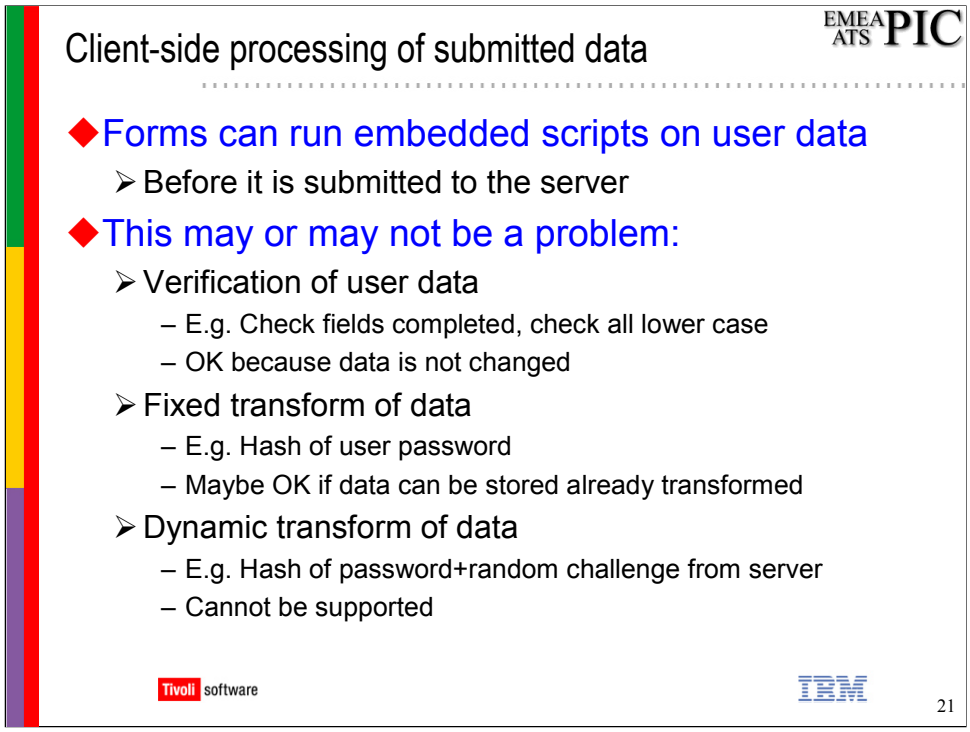


The diagram above summarises how the hidden inputs from the form and the SSO configuration are combined to build the inputs that will be sent when the form is submitted.

The order that the inputs are sent when the form is submitted is the same order in which they were received on the form. If additional inputs are specified in the SSO configuration that are not in the form then these will be at the end of the list in the order in which they appear in the configuration.

In the example above the hidden *snid* input is not mentioned in the configuration file and so is submitted to the server unchanged. The *userid* and *passwd* inputs specified in the configuration file are filled in with the current users GSO username and password. The hidden input *auth* from the form is overridden with the fixed value "poldir". The input, *appinfo,* will be filled in with the value of the *LIMIT* attribute from the users credential.

Inputs do not have to be present on the received form in order to be specified in the configuration – all inputs specified in the configuration will be submitted.



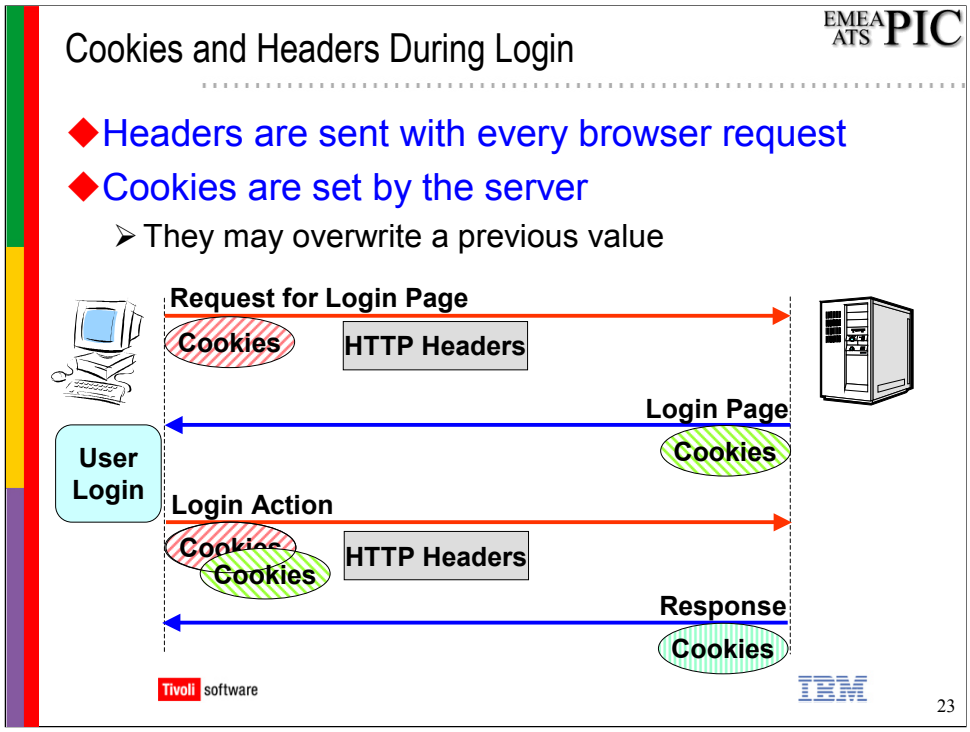
Some login pages execute script code (Javascript, ActiveX etc) before the form is submitted. WebSEAL does not execute this code and this may cause a problem if the code is required for the login process.

It is not a problem if this code is simply intended to check the input prior to submission but it may present a problem if it modifies the user input.

If the user input is modified in a fixed way then it may be possible for this to be handled by providing the input data to WebSEAL in the way that it would be finally presented to the back-end server after the modification. This pre-modified information could be stored in the GSO database or added to the users credential at login time by a custom CDAS.

If the user input is modified in some dynamic way (combined with some random data from the back-end for example) then WebSEAL will not be able to handle this.


EMEA ATS PIC
Maintaining Cookies and HTTP Headers
Tivoli software
IBM
22



Cookies and HTTP Headers are important in a form-based login enviroment because they are often used by the server to identify the users session. This is not so important in a Basic Authentication environment because the Basic Authentication header (which is sent with every request) can be used instead.

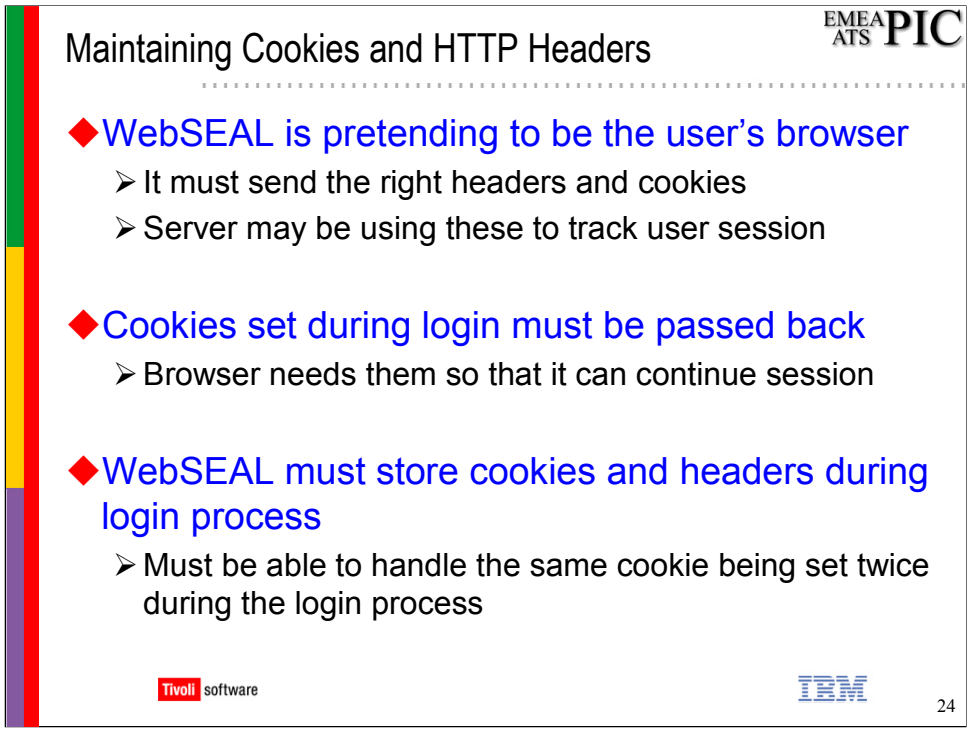
The diagram above shows what cookies and headers might be involved in a login exchange.

When the browser makes its request for the login page this will probably include both HTTP headers and cookies.

When the server sends the login page to the browser it may set some additional cookies. Some of these might be new and some might update the value of existing cookies.

When the login form is submitted the HTTP headers are sent again along with an extended set of cookies. The set of cookies sent will include all of the cookies from the previous request and all those set by the server in the previous step. A cookie can only have a single value so any cookies set with the login page will have overwritten any cookie with the same name from the original request.

When the server has completed the login process it will send a response. This may include yet another set of cookies.



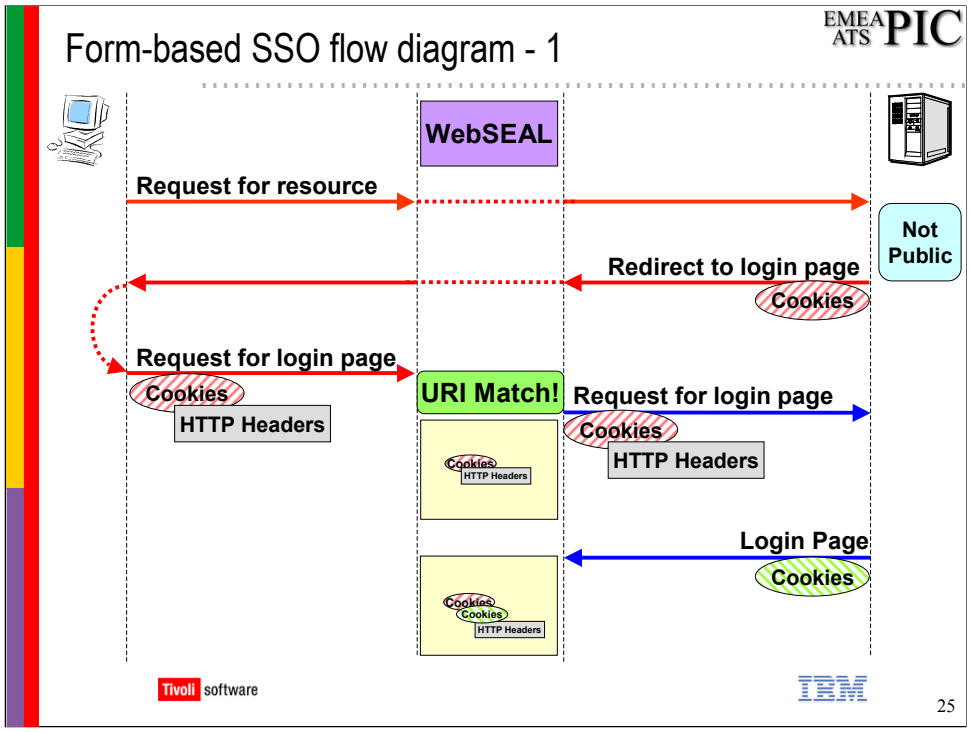
Since WebSEAL is going to fill in the login form on the users behalf, it must be able to send the correct set of cookies and headers to the server along with the login form. These may be used by the server to track the user session during the login process.

WebSEAL stores the cookies and HTTP headers that are sent with the request for the login page and adds to these the cookies that are set by the backend server when it sends the login page.

Once WebSEAL has processed the login form it submits the form along with the full set of cookies and HTTP headers it has saved. If any cookie with the same name was received from the browser and from the server the server value will be used because it was set last.

When the server responds to the login action it may set more cookies. These are passed back the the browser along with any cookies set with the login page. If any cookie with the same name was received from the server twice the value from after the login is used because it was set last.

By using the process above WebSEAL ensures that the server receives all the cookies that would have been sent by the browser and also ensures that the browser is updated with all the cookies that were set by the server.



The diagram above shows the first half of a typical form-based SSO login process. It assumes that the user has already authenticated to WebSEAL.

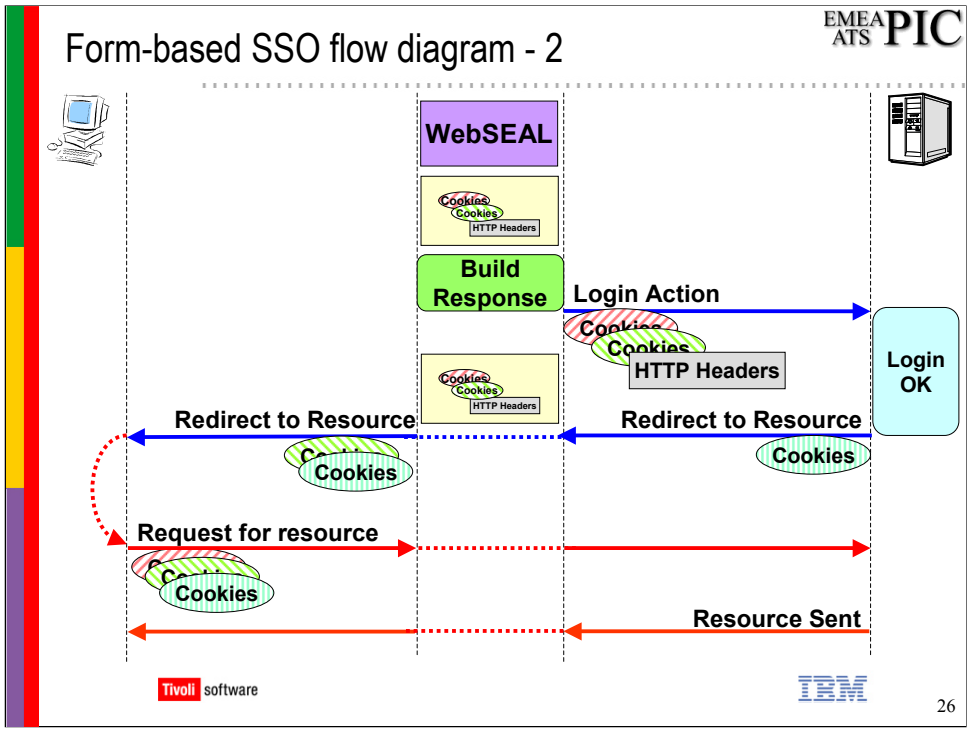
The user requests, from WebSEAL, a resource from a junctioned back-end server. This request is authorized by WebSEAL and then passed onto the back end server. The back-end server also wants to authenticate the user so a re-direct to a login page is sent back. WebSEAL passes this back to the browser.

The re-direct causes a request for the login page to be sent to WebSEAL. After authorizing the new request, WebSEAL recognises its URI and this triggers the Forms-based SSO function.

WebSEAL requests the login page from the backend server. This request includes the cookies and headers from the browser. It also stores these for later use.

The backend server is unaware that this request is from WebSEAL and not from the browser and so returns the login form to WebSEAL. If any cookies are included in the response WebSEAL also stores these for later use. Any cookies of the same name that it already has stored are overwritten.

Continued on next page…



Continued from previous page…

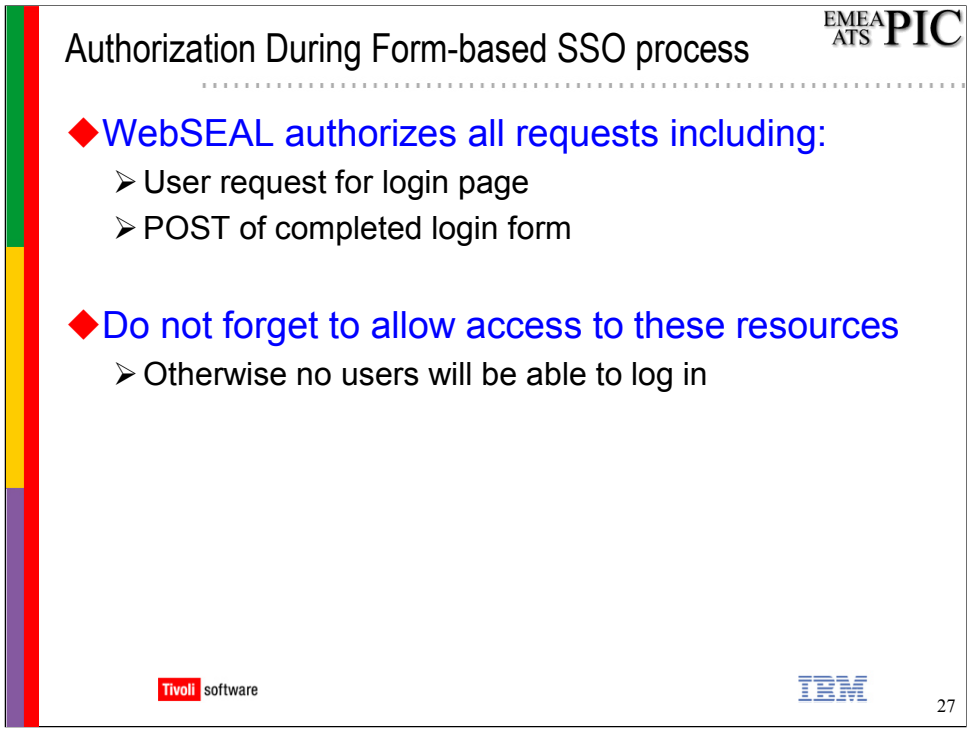
WebSEAL uses the SSO configuration associated with the junction to locate the correct form in the login page. It then extracts the hidden inputs from the form and uses them along with the junction SSO configuration to build the response for submission. This response is then sent to the backend server along with the stored cookies and HTTP headers.

The backend server authenticates the user using the information in the submited form and, assuming that the login is successful, it returns a re-direct back to the originally requested resource.

WebSEAL passes this re-direct back to the browser along with the cookies that were set by the server when it sent the login page and the cookies sent with the re-direct. If any cookies were set in both cases then the value set with the re-direct is used. This is the end of the Form-based SSO involvement.

As a result of the re-direct the browser sends a new request for the resource to WebSEAL. This request includes the cookies just set by WebSEAL that came from the back-end.

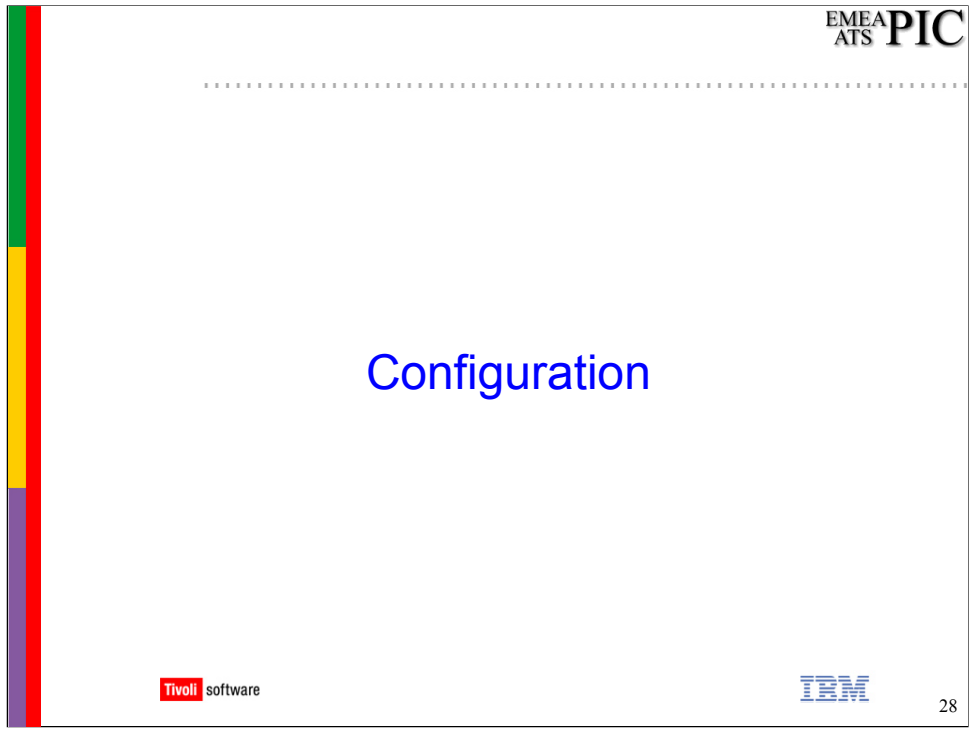
WebSEAL authorizes the request and passes it to the backend. Since the user is now logged in the backend server sends the requested response which WebSEAL passes back to the user.

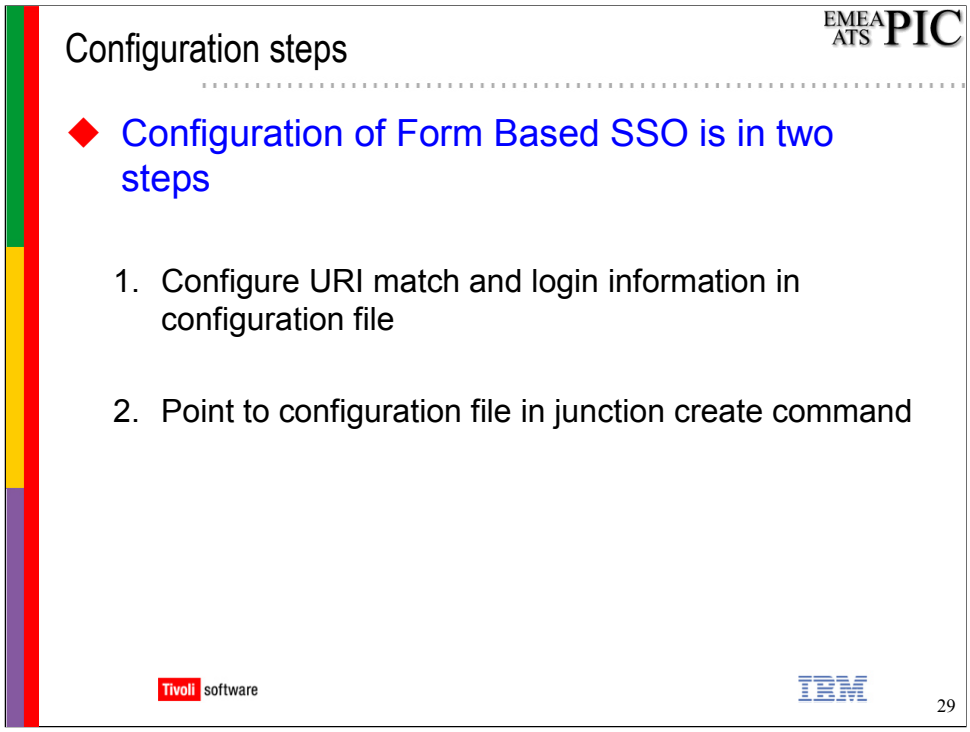


Since the primary role of WebSEAL is to protect Web resources from unauthorized access it must authorize all requests for resources even if they are part of a Form-based single sign-on process.

WebSEAL checks the ACL database before allowing access to the back-end server login page and also checks before allowing access to the URI specified in the form action (where the completed login form is sent).

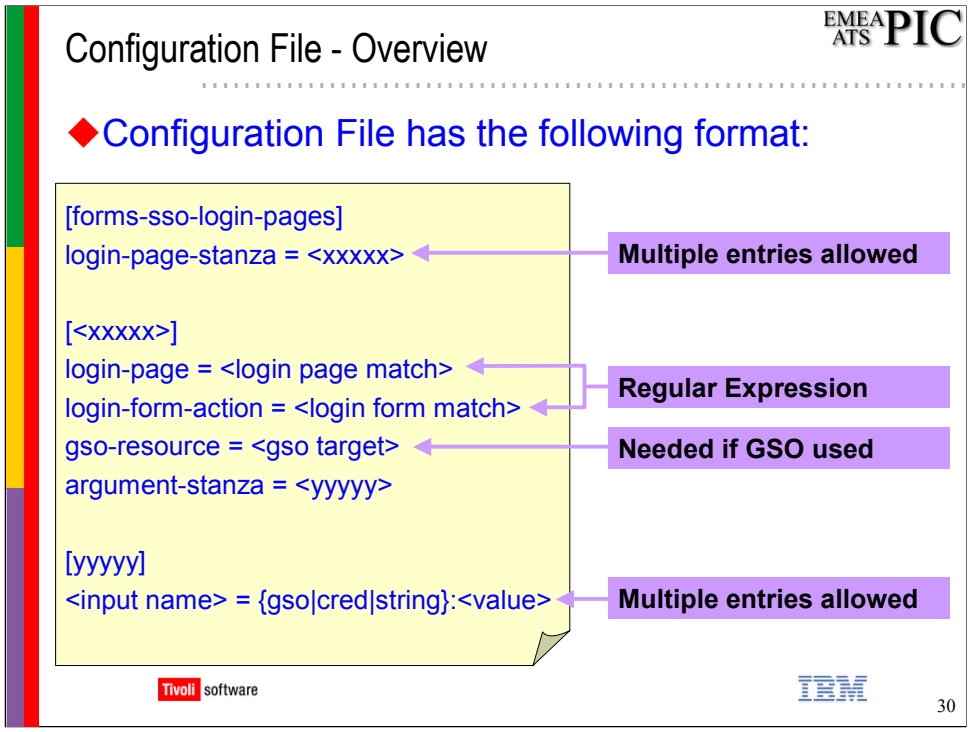
If the security policy does not give permission for the current user to access these pages then the Form-based single sign-on will fail.


EMEA ATS PIC
Configuration
Tivoli software
IBM
28



Form-based SSO is configured on a per junction basis.

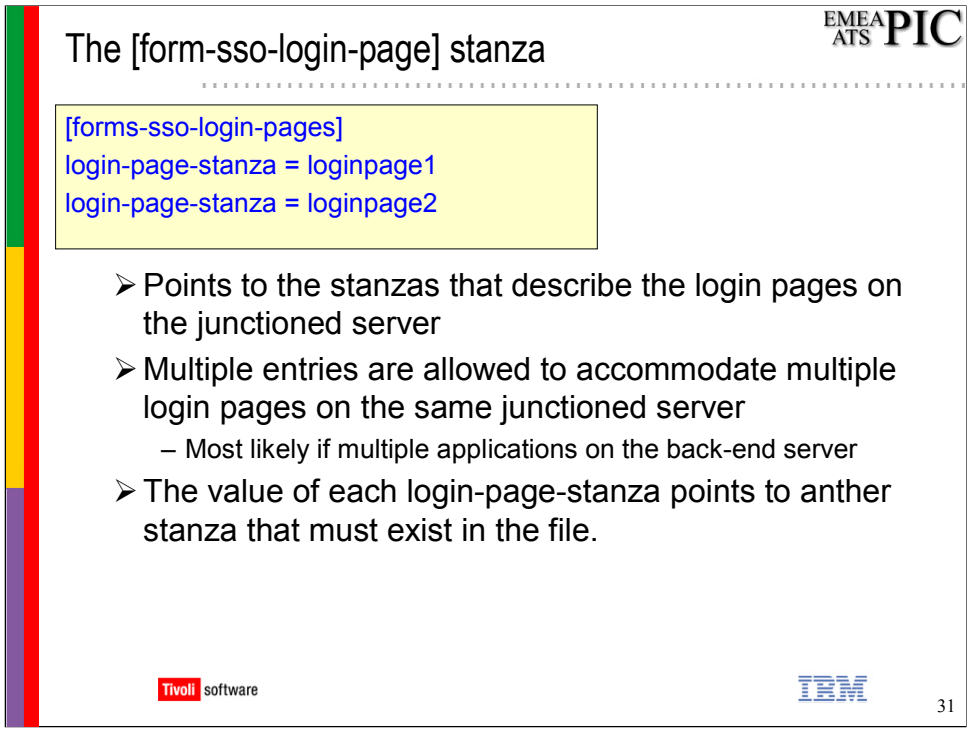
First a configuration file is created on the filesystem and then this is specified on the junction create command using the new **–S** parameter.



The diagram above shows the format of an Form-based SSO configuration file.

This is a text file that contains a stanza called [*forms-sso-login-pages]* and then a number of other stanzas (that can have any names) which are pointed to from that stanza.

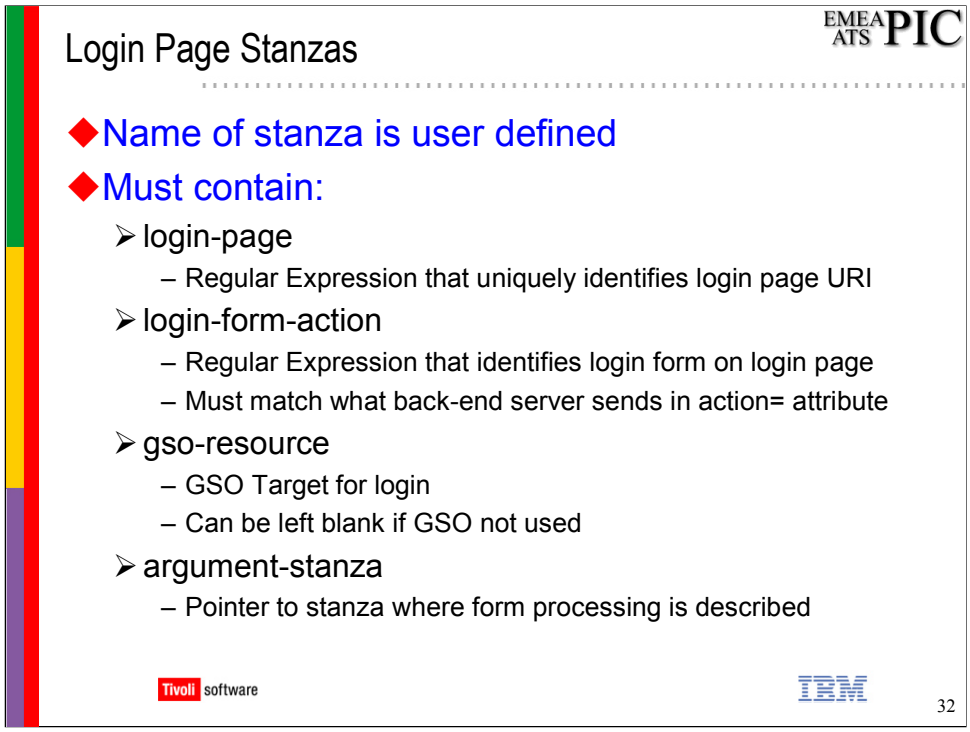
Details of the configuration file parameters are given on the following pages.



In an Form-based SSO configuration file there must be a stanza called [*forms-sso-login-pages]*.

This stanza must contain one or more *login-page-stanza* entries which point to other stanzas (by name) that contain the configuration for the different login pages found on the back-end server.

The ability to support multiple login pages on a single junction is important because a single back-end server might host several applications that each use form-based SSO with different login pages.

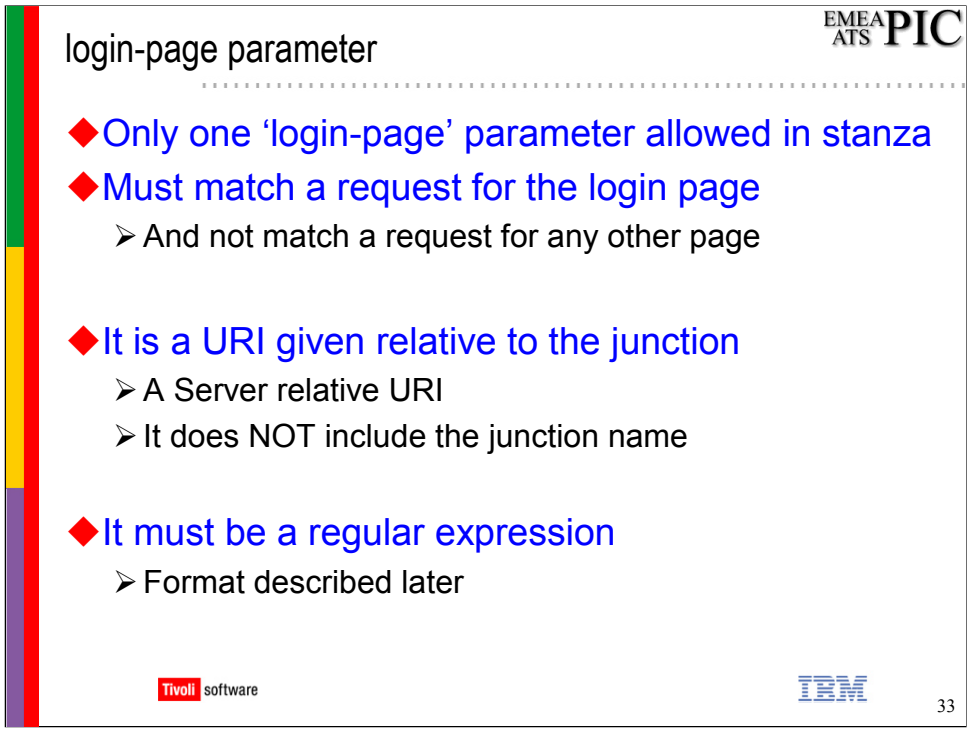


Each login page stanza contains a *login-page* parameter that specifies a pattern match to identify the login page. This is what will be matched against the URI of incoming user requests.

Once a request URI is matched to the *login-page* parameter and a login page returned, the *login-form-action* parameter is used to locate the correct form on the login page. If multiple forms match then the first will be used.

A GSO target can be specified in the login page stanza using the *gso-resource* parameter. This paramter must be present in the stanza but only needs to be completed if the GSO database is needed to complete the form.

The stanza pointed to by the *argument-stanza* parameter is examined to determine how the login form should be completed.



The *login-page* parameter is a regular expression that WebSEAL uses to determine if an incoming request is a request for a login page. Only one *login-page* parameter is allowed in each login page stanza but multiple login page stanzas can be used if necessary. The ways in which a regular expression can be built are shown later.

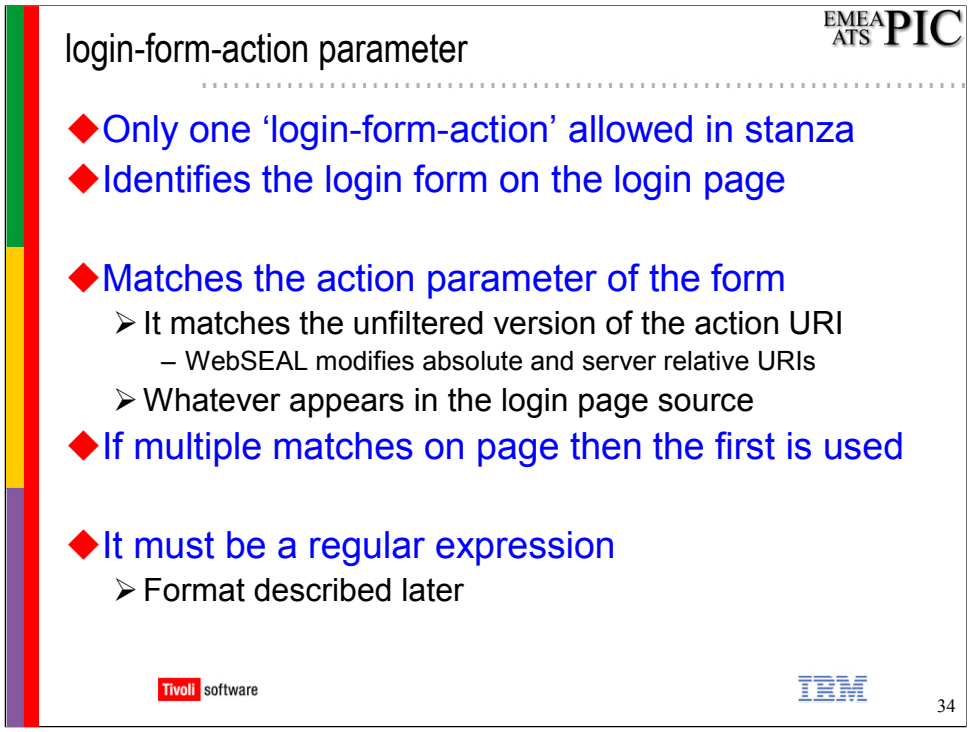
The regular expression is compared against the request URI **relative to the junction.**

For example the URI of a request to a WebSEAL server called webseal1 for a resource on a junction called junction1 might look as follows:

https://webseal1.mycompany.com/junction1/auth/login.html

The part of this that will be compared to the *login-page* regular expressions is:

/auth/login.html



The *login-form-action* parameter is used to identify the login form on the page returned by the back-end server following a request matching the *login-page* parameter.

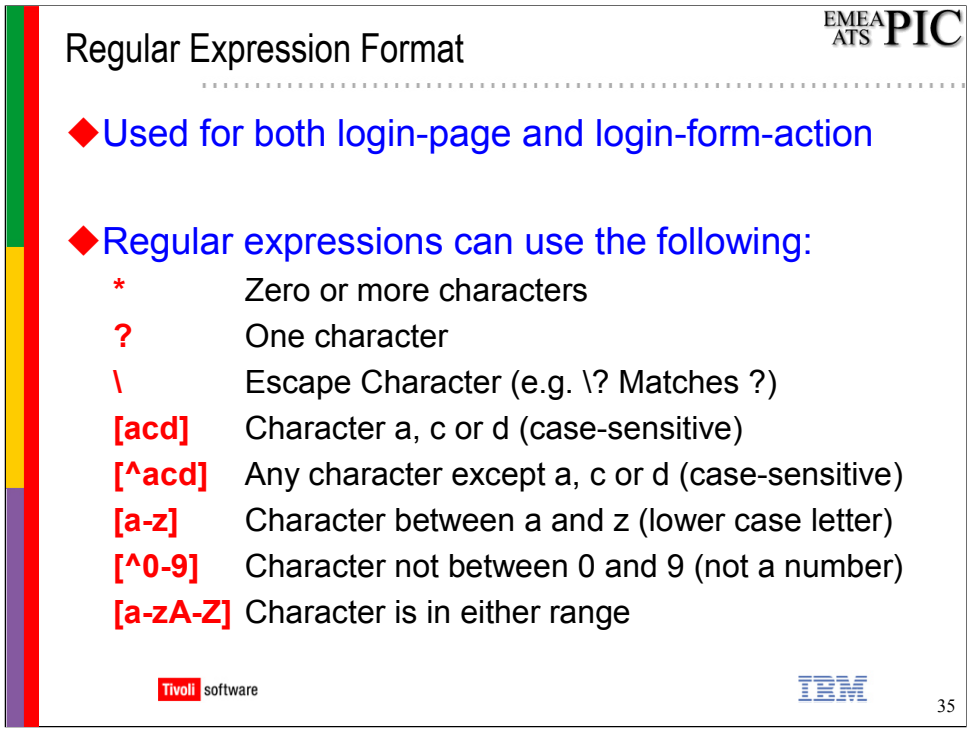
Only one *login-form-action* parameter is allowed in each stanza.

The *login-form-action* parameter value is a regular expression that is matched against the contents of the *action*= attribute of the *form* tag. The action attribute will be a URI and might be in the form of a relative, server relative or absolute URI. The *login-form-action* must match this **as it comes from the server** – even if it would normally be modified by WebSEAL before being forwarded to the client.

If multiple *action* parameters on the page match the regular expression then the first will be taken as the login form.

If the *login-form-action* parameter does not match any form on the page then an error will be returned to the browser reporting that the form could not be found.

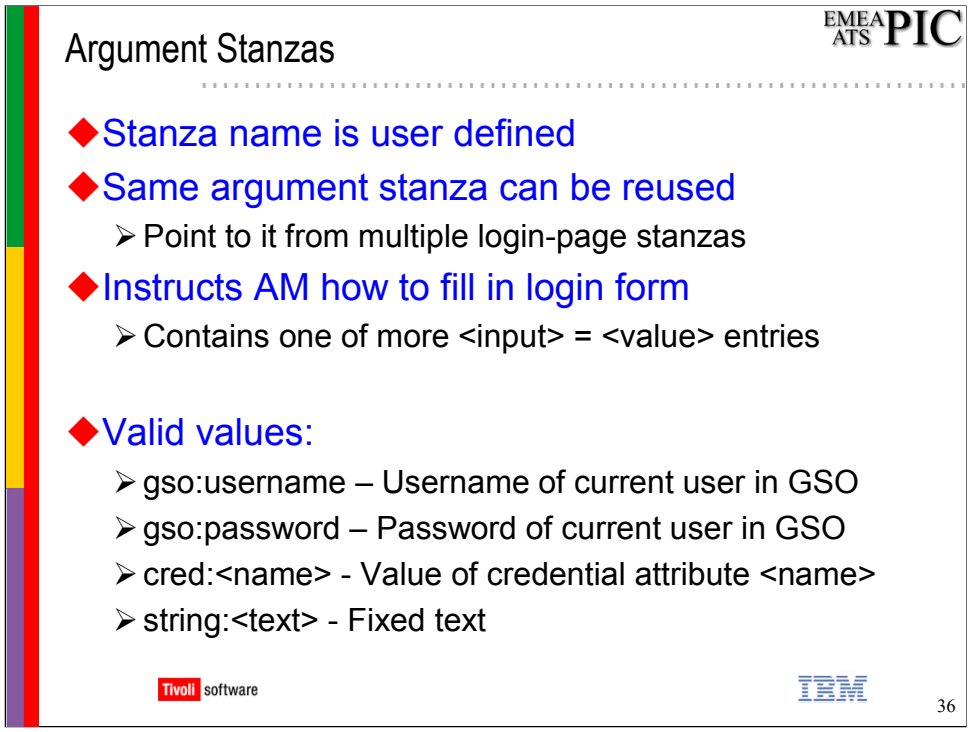
Hint: Setting *login-form-action* to ***** is an easy way to match the login form when there is only one form on the login page (which is often the case).



The slide above shows the special characters that can be used in the regular expressions found in the SSO configuration files.

In most cases special characters will not be required because the login page request will be a single identifiable URI. In some cases it might be desirable to use a * at the end of the expression so that any query data at the end of the URI will not prevent the login page from being matched.

It is worth noting that if the login URI contains a ? then this should be matched with \? since ? alone will match any character.



The argument stanzas contain one or more <*name*> = <*value*> entries that describe how the inputs on the login form should be processed in order to complete a login on the users behalf.

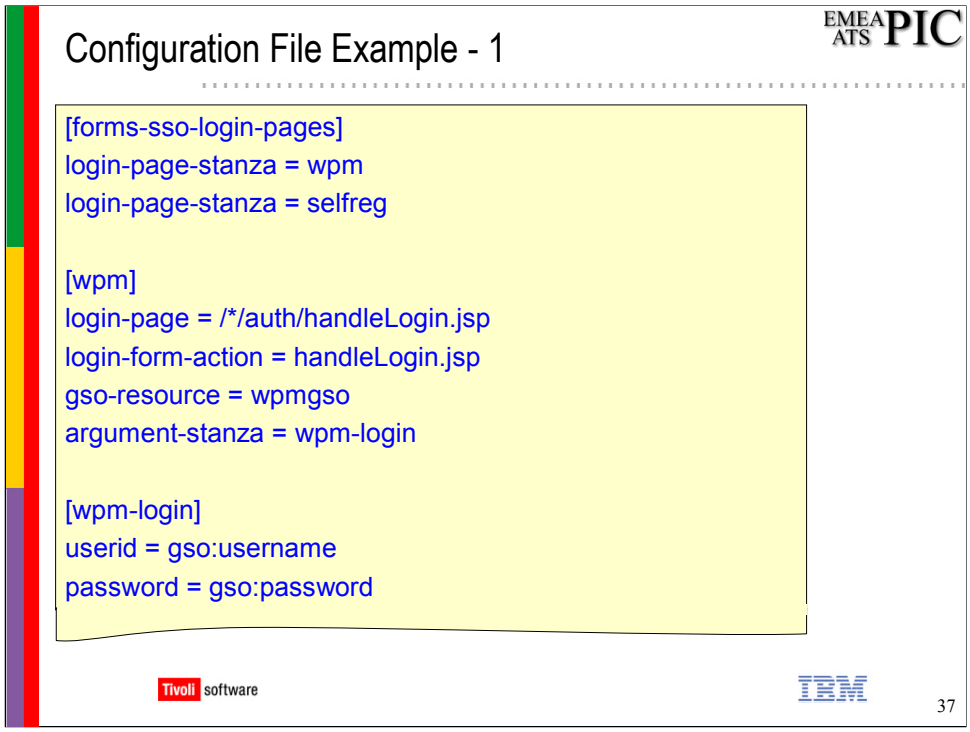
The *name* is the input name and the value specifies how it should be filled in. The are four possibilities:

gso:username – This keyword indicates that the current users GSO username (from the target specified in the login page stanza) should be placed into the input.

gso:password – This keyword indicates that the current users GSO password (from the target specified in the login page stanza) should be placed into the input.

cred:<*name*> – The *cred* keyword followed by an identifier indicates that the input should be filled in using the value associated with the named attribute from the current users credential. By default the credential contains information such as the users principalName and DN but custom credential attributes (such as those added as part of the tag-value support) can also be used.

string:<*text*> - The *string* keyword followed by text means that the input should be filled in with the text indicated.

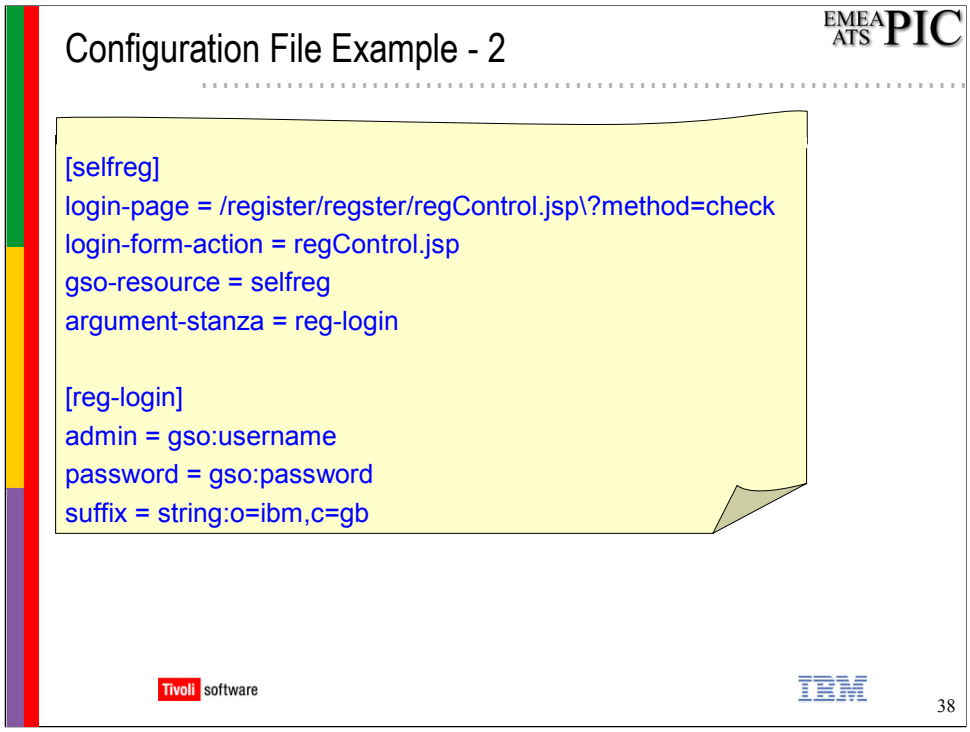


The slide above (and the next slide) show an example of a forms-based SSO configuration file. This example allows automatic login to the Access Manager Web Portal Manager (WPM) applications and self registration example.

There are two different *login-page-stanza* entries – one for each the different login forms. One points to a stanza called *wpm* and the other to *selfreg*.

In the *[wpm]* stanza the *login-page* attribute matches the login URI of both the *pdadmin* application and the *delegate* application by using a *. The *login-form-action* matches the action contained in the form tag (this could have been set to * since there is only one form on the page). The *gso-resource* parameter is used because the user and password information needed to sign the user into the WPM will come from the GSO database. The *argument-stanza* parameter points to the *wpm-login* stanza.

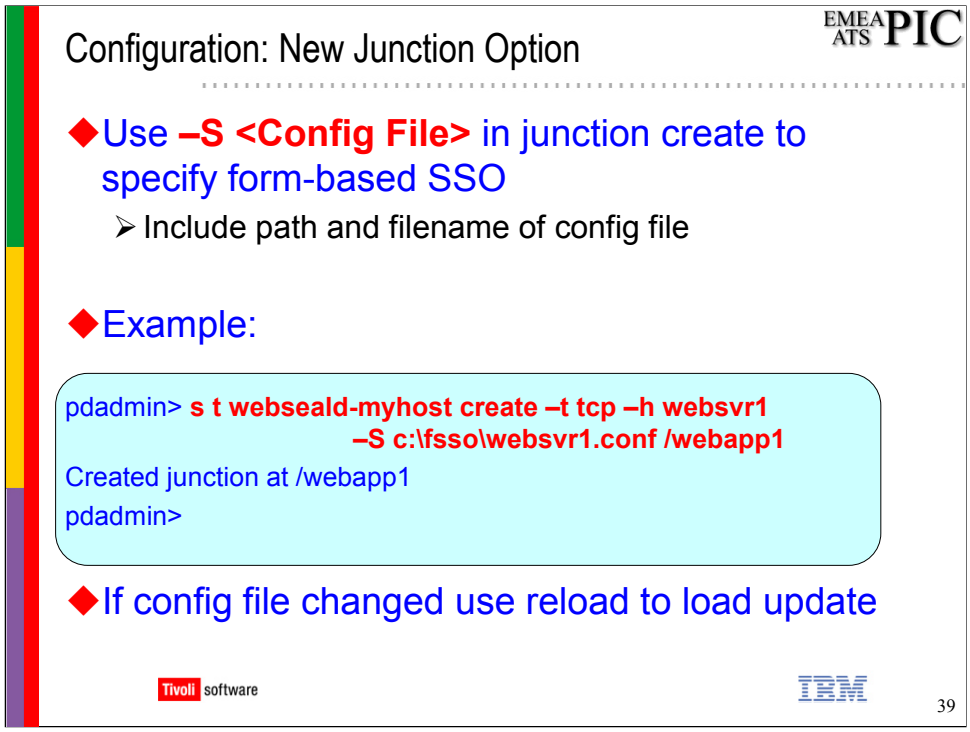
The *[wpm-login]* stanza specifies that input named *userid* and *password* should be submitted and that they should be filled in with information from the GSO database ((using the *wpmgso* target as specified above)). Any hidden inputs on the form (there is a hidden input called dologin) will be submitted unchanged as they are not mentioned in the configuration.



The slide above shows the second half of our example configuration file.

In the *[selfreg]* stanza the *login-page* attribute matches the login URI of the self registration demo application. There are no wildcards included; notice that the ? has been escaped.. The *login-form-action* is set to match the action in the form tag. The *gso-resource* parameter is used because the user and password information needed to sign the user into the application will come from the GSO database. The *argument-stanza* parameter points to the *reg-login* stanza.

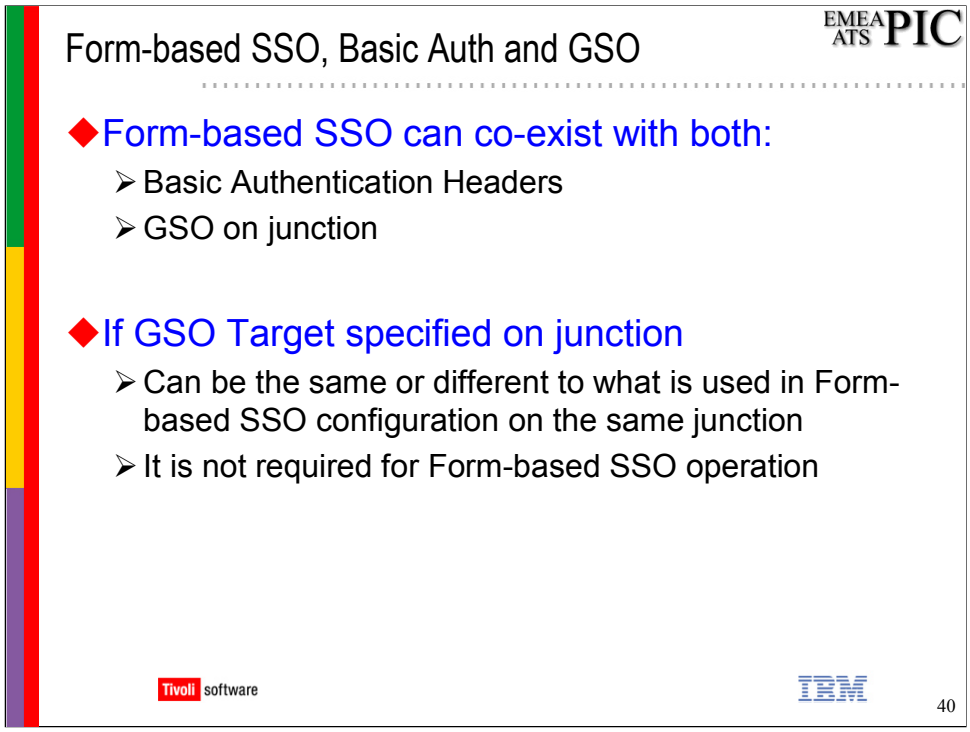
The *[reg-login]* stanza specifies that inputs named *admin* and *password* should be submitted and that they should be filled in with information from the GSO database (using the *selfreg* target as specified above). An input named *suffix* should be completed with a fixed string *o=ibm,c=gb*. If there are any hidden inputs in the form then these will be submitted unchanged because they are not mentioned in the configuration.



Once the form-based single sign-on configuration file for a junction has been completed it is specified during the junction create command to bring it into effect. A new parameter (**-S**) has been added to the junction create syntax to allow the configuration file to be specified. The presence of this attribute also enables the form-based SSO function on the junction.

The configuration file is read when the junction is created and each time WeSEAL is started. If changes are made to the file and need to be activated then reloading the junctions will cause this to take place.

If there are errors in the form-based SSO configuration file specified on any junction then WebSEAL will fail to start. If the default routing file is in use then an error will be logged to file (and also printed to STDERR if WebSEAL is running in the foreground).



Although forms-based SSO can make use of information in the GSO database it does not interfere with the GSO capability of the junction that is specified using the –b gso and –T <target> flags in the junction create command.

It is possible, if required, to have one GSO target in use to fill in the Basic Authentication headers sent to the backend server and another specified in the forms-based SSO configuration for use when filling in login forms.