



Axel Buecker
Mohit Chugh

A Guide to Writing Advanced Access Profiles for IBM Tivoli Access Manager for Enterprise Single Sign-On

Introduction

IBM® Tivoli® Access Manager for Enterprise Single Sign-On enables users to access all their applications, including web, desktop and heritage, and network resources, with the use of a single strong password. The solution helps simplify password management, protects information with strong authentication, and secures kiosks and shared workstations.

Tivoli Access Manager for Enterprise Single Sign-On helps strengthen security and meet regulations through stronger passwords and an open authentication device interface with a wide choice of strong authentication factors. It also facilitates compliance with privacy and security regulations by leveraging centralized auditing and reporting capabilities.

In this IBM Redpaper™, we take a closer look at how to integrate web-based applications into Tivoli Access Manager for Enterprise Single Sign-On by using its AccessProfile technology.

This IBM Redpaper is a good resource for security administrators who are responsible for configuring and integrating Tivoli Access Manager for Enterprise Single Sign-On into their organization's IT infrastructure.

This IBM Redpaper contains the following sections:

- ▶ “Background” on page 2
- ▶ “Document complete event and the Observer” on page 4
- ▶ “Signatures” on page 7
- ▶ “Auto-learn AccessProfile” on page 10
- ▶ “Handling basic authentication” on page 10
- ▶ “Frames and the web browser document object” on page 10
- ▶ “Differences between Firefox and Internet Explorer AccessProfiles” on page 24
- ▶ “Common issues” on page 24

Background

A typical modern web browser consists of multiple tabs, each of which can independently host a web page. A web page is a document written in HTML format and can be recognized by the starting `<html>` and ending `</html>` tags in the page source of the web page, typically available by the right-click context menu or file menu of the browser. It is possible for the browser to show other formats, such as XML, images, text, pdf, and so on, but they are not relevant from a profiling point of view.

When writing AccessProfiles, you do not have to care about how the HTML page is generated on the server side by using ASP, CGI, JSPs, and so on. On the other hand, client-side JavaScript included in the web page can be important while writing an AccessProfile because they can modify the workflow of the web page within the browser while the AccessProfile is acting on it.

HTML

HTML is a hierarchical format and thus can be thought of as a tree of HTML elements that serve as nodes.

The top level node is HTML, which usually consists of a HEAD and a BODY node, as shown in Example 1.

Example 1 Sample HTML

```
<html>
<head>
<title>Dummy page</title>
</head>
<body></body>
</html>
```

Example 2 shows that each element can have optional attributes that qualify the way the element is displayed. Here, the INPUT element has ID and type attributes.

Example 2 Sample HTML with attributes

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
<title>Dummy page</title>
</head>
<body>
<input id="uname" type="text"></input>
</body>
</html>
```

HTML and JavaScript

Elements can also have events assigned to them that can affect the way the HTML page behaves. JavaScript is the most common scripting language used to describe what must happen when the event occurs. Example 3 on page 3 shows an example.

Example 3 Sample HTML with Javascript

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
<title>Dummy page</title>
</head>
<body>
<input id="button1" type="button" onclick="javascript:alert("button clicked");" />
</body>
</html>
```

Here the `BUTTON` element has an `onclick` event, which when the button is clicked, shows a button clicked message by using JavaScript.

For non-trivial cases, the event is usually invoked as a function call like the one shown in Example 4.

Example 4 Sample HTML with Javascript function call

```
<html>
<head>
<title>Untitled Page</title>
</head>
<body>
<script type="text/javascript">
function clicked() {
  alert("button clicked");
}
</script>
<input id="button1" type="button" onclick="clicked();" value="Click me" />
</body>
</html>
```

JavaScript and DHTML

It is possible for an HTML element's event's JavaScript to modify other elements of the page, making the HTML more dynamic, hence the name *Dynamic HTML* (DHTML). It is important to watch out for cases like these because they can lead to unexpected SSO behavior. See Example 5.

Example 5 Sample HTML with Javascript

```
<html>
<head>
<title>Untitled Page</title>
<script type="text/javascript">
function body_loaded() {
  uname.value = "";
}
</script>
</head>
<body onload="body_loaded();">
<input id="uname" type="text" value="start-val" />
</body>
</html>
```

In this case, if the AccessProfile injected anything in the `uname` input field before the `onload` event on `BODY` element was called, the text would have cleared by the `uname.value=""` command, giving the impression that no injection took place.

HTML load sequence

When a user types a URL in the address bar and submits it, the following events take place:

- ▶ The web page is downloaded. While the web page is being downloaded, the browser starts to parse the html page and starts to partially show its content on the screen. The Observer ignores this downloading process altogether because the browser does not provide any useful events at this time.
- ▶ The web page completes downloading and the browser finishes parsing through the HTML content and creates an internal Document Object Model (DOM) object that is a binary representation of the HTML elements, their attributes, and scripts. Only after this DOM object is created does the browser inform the Observer about the existence of this page in the form of a document complete (the DOM object is now fully formed and available) event.

This event is trapped by the Observer and made available as a web page completes loading trigger and used internally to determine if it needs to load a new AccessProfile for the web page. The details of the mechanism are explained in “Document complete event and the Observer” on page 4.

- ▶ The `onload` event for the `BODY` element fires and executes any `onload` script that is specified in the HTML page. This event is trapped by the Observer and made available as the HTML element completes loading the trigger (this is a generic trigger and can also be used for tracking HTML element completes that load events in other elements, such as `FRAME` and `IMG`).

Document complete event and the Observer

The document complete trigger plays an important role in determining the lifetime and behavior of an AccessProfile instance within the browser.

The first document complete event

We start from the time the browser starts a new tab and submits a URL. At the first document-complete, the web-sso-agent notices that there is no profile loaded currently and it contacts the DataProvider and passes to it the details of the current web page, such as domain, path, protocol, port, and so on.

The DataProvider matches the passed information against the signatures of the currently loaded AccessProfiles. Four things can happen at this stage:

- ▶ The DataProvider finds multiple AccessProfiles for the passed information. To avoid unpredictable behavior, the Observer does not load any of the matched AccessProfiles, but instead writes an error line in the observer logs and in the General tab of AccessStudio messages.
- ▶ DataProvider finds a single match for the passed information. The matched AccessProfile ID is returned to the web-sso-agent, which then makes another call to fetch the AccessProfile object itself and creates the state-machine object described in it.

In AccessStudio, if there was not an existing tab open for the browser, a new tab is created and a new log with the following log is shown:

Loaded AccessProfile: *profile-name*

- ▶ DataProvider does not find any matching AccessProfile(s), but there is an auto-learn AccessProfile defined. In this case, the auto-learn AccessProfile ID is returned to the web-sso-agent as a match, which then fetches the actual AccessProfile object and creates the state-machine object described in it.

In AccessStudio, if there was not an existing tab open for the browser, a new tab is created and a new log with the following log is shown:

Loaded AccessProfile: sso_site_web_auto_learn

- ▶ DataProvider does not find any matching AccessProfile and there is no auto-learn AccessProfile defined (or the auto-learn is disabled by a policy). Considering that the Observer passes this first document-complete event to the newly loaded state-machine as well.

Based on the document-complete triggers that are available in the start-state, either one of those triggers can fire, moving the state-machine to the next state or it can stay in the same place if no trigger fires. The exception is the auto-learn AccessProfile. If the starting web page completes loading trigger does not fire, the state-machine is unloaded immediately. The flow diagram in Figure 1 on page 6 explains this behavior.



Figure 1 State machine behavior w.r.t. document complete events

In AccesStudio, there is no log for this event, so if there was no existing tab for the browser process, there is no new tab created at this time.

Beyond just being used to load the AccessProfile, the document complete event is of interest to the AccessProfile writer because usually it is a good place to inject credentials.

Subsequent document complete events

After the first document complete was rerouted to the state-machine after it caused the loading of the AccessProfile, the rest of the document complete events are treated in the following fashion:

- ▶ If there is a webpage completes loading trigger in the current state that matches this document-complete-event, that trigger fires and the state-machine continues as normal.
- ▶ If there is no webpage completes loading trigger in the current state that matches this document-complete-event, the agent goes back to the DataProvider and fetches a matching profile. In this case, the fetched profile is the same as the current profile, and the state-machine stays where it is. Otherwise, if the fetched profile is different from the current one, the current profile is thrown away, the new one is loaded, and the subsequent sequence is identical to the first document complete event previously described.

Signatures

The Observer uses two kinds of signatures for web applications: the web-signature and the HTML signature.

The web-signature

The web-signature, shown in Example 6, is the site-signature for web applications. It is used to match the right AccessProfile to a given web application. It can only be one level deep and has the following form:

```
/child::web[<attribute filters list>]
```

Example 6 Sample web signature

```
https://www.google.com/accounts/ServiceLogin?service=mail&passive=true&rm=false&continue=http%3A%2F%2Fmail.google.com%2Fmail%2F%3Fui%3Dhtml%26zy%3D1&bsv=11ya6941e36z&sc=1&tmp1=default&tmp1cache=2
```

Using the example shown in Example 6, the list of attributes in Example 7 through Example 12 on page 8 are valid. A string attribute requires quotes around it when specified as a filter in the signature, while a numeric attribute does not have quotes.

- ▶ **domain** (string): This is the text between the '/' after the protocol and the first '/'.

Example 7 Sample domain

```
/child::web[@domain="www.google.com"]
```

- ▶ **protocol** (string): This can be http, https, file, or ftp.

Example 8 Sample protocol

```
/child::web[@protocol="http" and @domain="www.google.com"]
```

- ▶ **url** (string): The complete URL of the page. This is what is seen on the address bar of the browser.

Example 9 Sample url

```
/child::web[@url="https://www.google.com/accounts/ServiceLogin?service=mail&passive=true&rm=false&continue=http%3A%2F%2Fmail.google.com%2Fmail%2F%3Fui%3Dhtml%26zy%3D1&bsv=11ya6941e36z&sc=1&tmpl=default&1tmplcache=2"]
```

The url attribute is not used often because the constituent parts of the url are available as other attributes, such as domain, query_string, and so on.

- ▶ **path** (string): This is the text after the domain but before the querystring (the text that begins with a question mark (?)). It always starts with a forward slash (/).

Example 10 Sample path

```
/child::web[@domain="www.google.com" and @protocol="https" and @path="/accounts/ServiceLogin"]
```

The path attribute is typically used in enterprise portal scenarios where paths hold several applications that require their own login.

- ▶ **port** (numeric): If not specified, the value is 80 for web pages with the http protocol and 443 for the https protocol.

Example 11 Sample port

```
/child::web[@domain="www.google.com" and @protocol="https" and @path="/accounts/ServiceLogin" and @port=443]
```

- ▶ **query_string** (string): This is the text after and including the question mark (?) in the url.

Example 12 Sample query_string

```
/child::web[@domain="www.google.com" and @protocol="https" and @path="/accounts/ServiceLogin" and @port=443 and @query_string="?service=mail&passive=true&rm=false&continue=http%3A%2F%2Fmail.google.com%2Fmail%2F%3Fui%3Dhtml%26zy%3D1&bsv=11ya6941e36z&sc=1&tmpl=default&1tmplcache=2"]
```

The query_string is almost always used with a regular expression check (~ or #) rather than directly because it is usually too specific to be used on its own.

Use it as a last resort for identifying a page because it is essentially a set of parameters passed to the page, and it can change from one machine (and browser) to the other.

The HTML signature

The HTML signature identifies the HTML elements. These elements are used as a part of a *web control* in AccessStudio to identify the HTML element to inject to or capture data from and for identifying HTML elements for triggers, such as the HTML element clicked or HTML element is found.

The root 'l' element is the HTML tag, so /child::html is at the level of the BODY and HEAD elements. The elements are identified by their type using the attribute tag_name:

```
/child::html[@tag_name="body"]
```

The value of the tag_name attribute is case-insensitive, so both of the examples in Example 13 on page 9 will work.

Example 13 Sample tag_names

```
/child::html[@tag_name="body"]  
/child::html[@tag_name="BODY"]
```

Unlike window signatures that have a predefined list of properties that can be used as attributes, each HTML element can have its own set of properties. For example, an INPUT element has a `type` property which describes whether it is a text-box or a button, while an "A" (anchor) element has an `href` property which defines the URL the user will be navigated when the link is clicked.

These properties can be used as attributes while defining a signature.

Example 14 Sample HTML properties

```
/descendent::html[@tag_name="input"and @type="button" and @id="submit"]
```

HTML elements can have events described for them too, for example, an INPUT element can have an `onclick` method. These methods cannot be used as attributes while writing a signature.

Beyond just identification of HTML elements, HTML signatures can also be used to retrieve data from a given HTML element. By default a web control (like the ones used in injection, capture, or data transfer actions) extracts the `value` data of the specified HTML element. This can be explicitly overridden to get other values, such as the inner text, inner html, or any other defined property for that control.

Example 15 Sample HTML properties

```
<html xmlns="http://www.w3.org/1999/xhtml">  
<head>  
<title></title>  
</head>  
<body>  
<a href="http://localhost/d/a.html">Site A</a>  
</body>  
</html>
```

For example, the following signature will return 'Site A'. This can be useful in extracting text from the web page, which can be used for auditing purposes or other workflow decisions within the AccessProfile.

```
/descendent::html[@tag_name="a" and  
@href#"http://localhost/d/a\.html.*"]/@inner_text
```

Like `inner_text`, `inner_html` can be used to return the HTML embedded within the identified HTML element; however, it has much less real-world utility.

In fact, any attribute that is defined for an HTML element can be retrieved in this way. For example, using the following example for the HTML sample in Example 15 returns 'http://localhost/d/a.html':

```
/descendent::html[@tag_name="a" and @href#"http://localhost/d/a\.html.*"]/@href
```

Auto-learn AccessProfile

ISAM ESSO comes bundled with a default profile for web applications, which gets loaded if there is no explicitly defined AccessProfile found for a web page. This AccessProfile relies on the fact that most web-sites' login pages follow a standard pattern of a username and a password input element that is embedded in a form element.

The auto learn AccessProfile is identified by a special signature, such as:

```
/child::web[@domain="*"] (note the "*" for domain)
```

It also contains a generic set of username and password signatures, such as:

```
/child::html/child::html/descendent::form/descendent::input[@type="password"]/ancestor::form/descendent::input[(@tag_name="input") and (@type="text" or @type="")]
```

Here is another example:

```
/child::html/child::html/descendent::form/descendent::input[@tag_name="input" and @type="password"]
```

Certain customers might not want to have this auto-learn behavior turned on. It can be turned off by:

- ▶ Setting the IMS™ scope policy – `pid_sso_auto_learn_enabled` to *false* (recommended).
- ▶ Removing the `sso_site_web_auto_learn` AccessProfile from IMS.

Handling basic authentication

Basic authentication is a mechanism where the authentication to the web server is done as a part of the exchange of HTTP header information rather than through an HTML login form.

The authentication happens before the web page contents are downloaded; therefore, the common approach of automating the HTML login page does not work because there is no HTML to work with.

Instead, the browser shows its own windows dialog, which asks for the credentials for the web site. The server that the browser is connecting to is shown as a part of the windows dialog UI.

There is no need to write an AccessProfile to handle basic authentication. The standard AccessProfile shipped along with ISAM ESSO has the relevant workflow to extract the server locator from the basic authentication dialog.

For Internet Explorer 8.0, the server locator is extracted by the regular expression – `Connect to (.*)` and all that needs to be done is to associate this server locator to the right authentication service. For other browsers, use different regular expressions to extract the server locator.

Frames and the web browser document object

In web browser terminology, a document refers to an object that is responsible for displaying the contents of a web page within a browser tab. Each of the browser tabs have their own unique document object. As the user navigates from one web page to another (either by

clicking a link or by submitting the URL in the address bar), the same document object continues to download and display the web page content.

Only one AccessProfile can be loaded for a document at a given time. However, as the document moves from displaying one web page to another it can unload the existing AccessProfile and load new ones. For example, as the user navigates from Gmail to Yahoo! Mail, the document unloads the Gmail AccessProfile (if present) and loads the Yahoo! Mail AccessProfile (if present).

The reason why the document object matters as far as writing AccessProfiles is concerned is because it is possible for one document object to contain other document objects. And as AccessProfiles are loaded in context of a document object, the AccessProfile writer can end up seeing what looks like multiple AccessProfiles loaded at the same time for a browser tab.

This multiple documents per browser tab scenario is what occurs in case a web page is using frames or iframes. Essentially, a frame (irrespective if it is defined in a FRAMESET or an IFRAME tag) is an instruction by the web page to the document object that is displaying it to create another child document object and load the contents of another web page within and show it at a particular place within the page layout. The web page of this child document object might in turn instruct it to create further child document objects creating a hierarchy of document objects that are each associated with a web page.

Example 16 shows an example of two iframes loading.

Example 16 a.html (defines two iframes loading b.html and c.html)

```
<html>
<head>
<title></title>
</head>
<body>
This is is the main web page - A
<br />
<br />
<iframe src="b.html" height="100" width="300"></iframe>
<br />
<br />
<iframe src="c.html" height="100" width="300"></iframe>
</body>
</html>
```

Example 17 shows an example of a login page where clicking the login button takes the document to another web page.

Example 17 b.html

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title></title>
</head>
<body>
<script type="text/javascript">
function clicked() {
document.url="d.html";
}
</script>
This is is the contained web page - B
```

```
<br /><br />
User: <input type="text" id="username" />
<br /><br />
Pwd: <input type="text" id="pwd" />
<br /><br />
<input type="button" id="submit" onclick="clicked();" value="Login"/>
</body>
</html>
```

Example 18 shows that a document (a.html) can contain more than one child document.

Example 18 c.html

```
<html>
<head>
<title></title>
</head>
<body>
This is is the contained web page - C
</body>
</html>
```

Example 19 simulates the successful login screen.

Example 19 d.html

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title></title>
</head>
<body>
This is is the contained web page - D
<br /><br />
Successfully Logged in!
<br /><br />
<a href="b.html">Back to b.html (the login page)</a>
</body>
</html>
```

Figure 2 on page 13 shows what an html looks like when loaded in the browser.

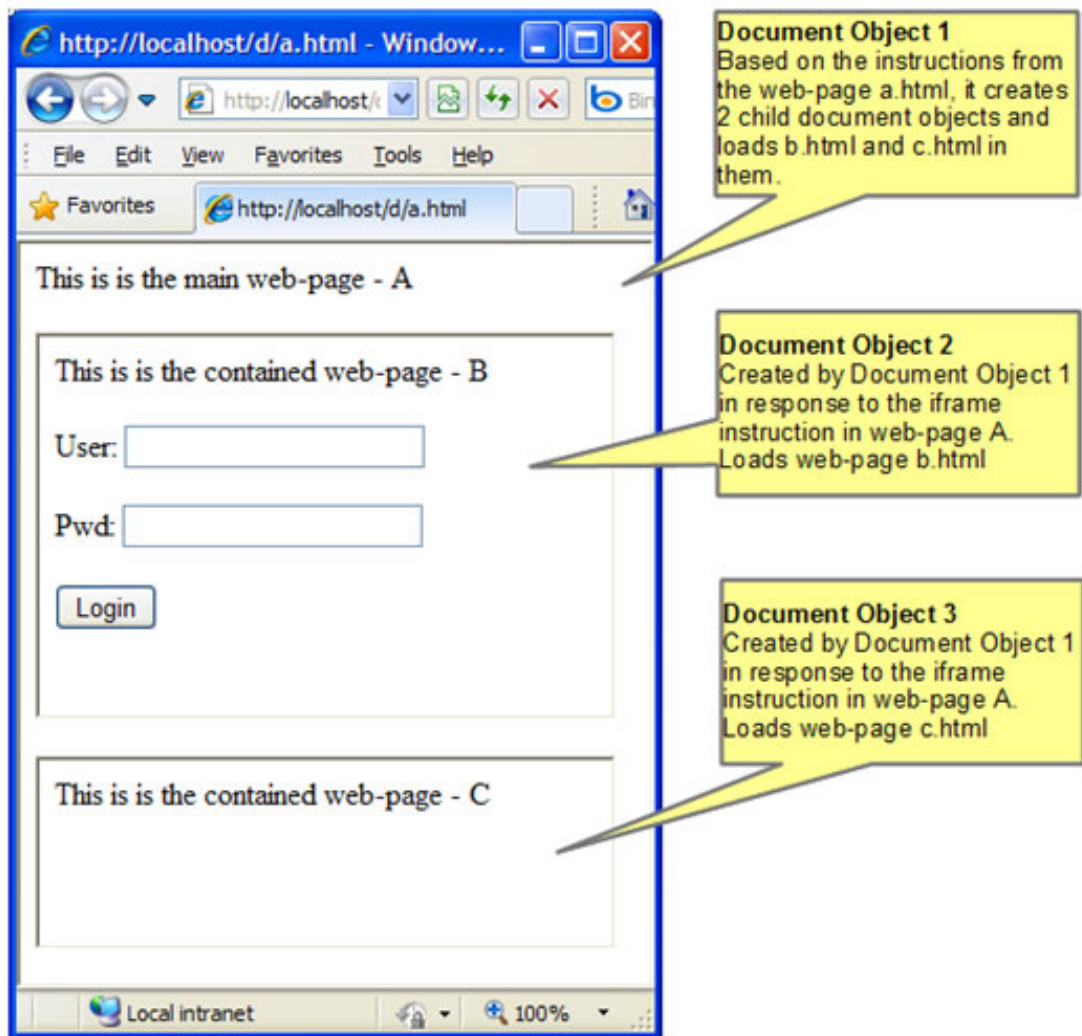


Figure 2 Page a.html with frames

Consider the cases shown in the following sections.

Web page B navigates to another web page within the same Document 2

After navigation, the browser looks like Figure 3 on page 14.

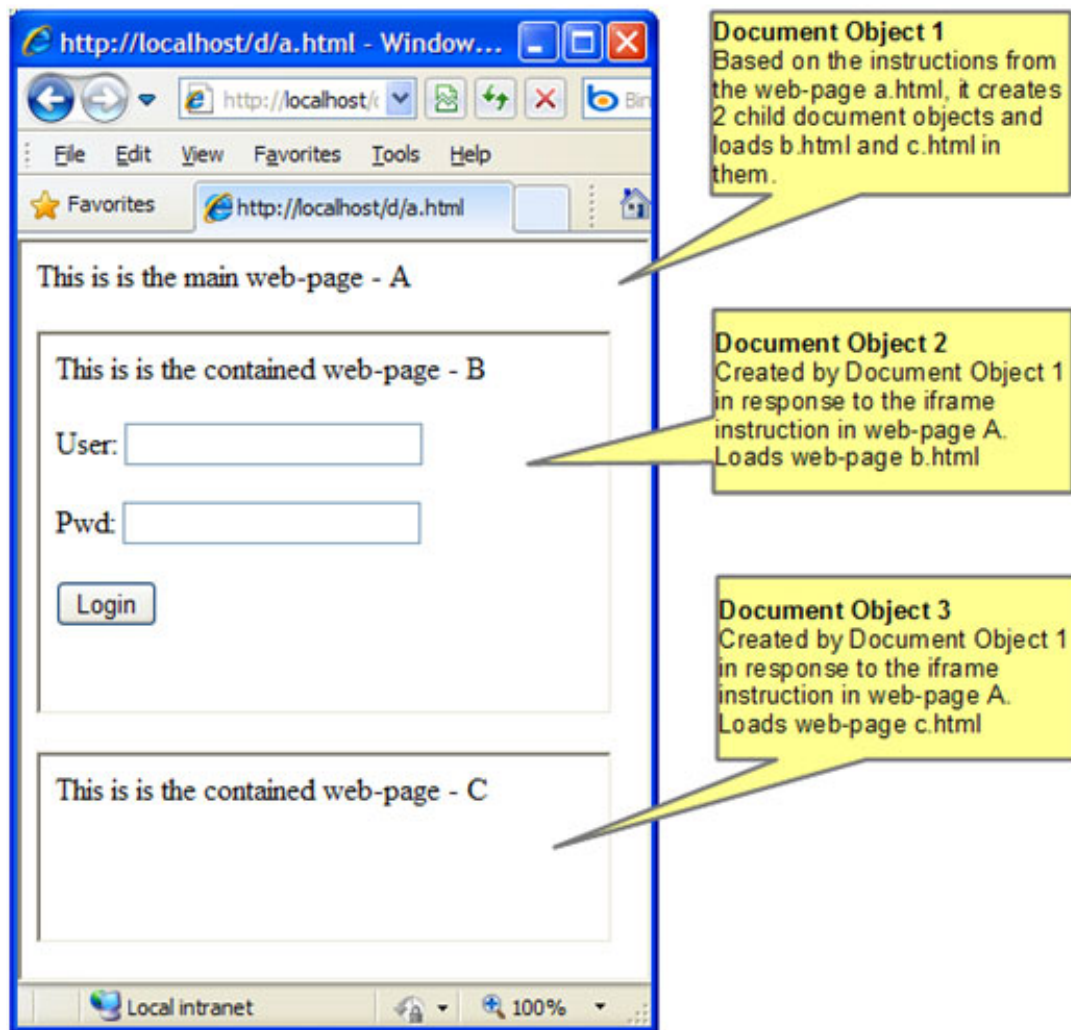


Figure 3 Frame moved to c.html

From writing an AccessProfile perspective, the fact that web pages B and D are located in a frame and not in the main document object is irrelevant because the document hosting the two web pages is the same. The AccessProfile looks exactly like what it would if web page b.html was loaded directly in the browser and not under a frame because the AccessProfile does not have to worry about any other document other than the one it was residing in.

In summary, as long as all of the workflow the AccessProfile is interested in occurs in one document object, the AccessProfile writer can ignore whether the web pages in the work flow were hosted in a child document object or the main document object.

A web page in the child document causes navigation in the parent document

Consider the same starting document in the earlier case. Figure 4 on page 15 shows page a.html with frames.

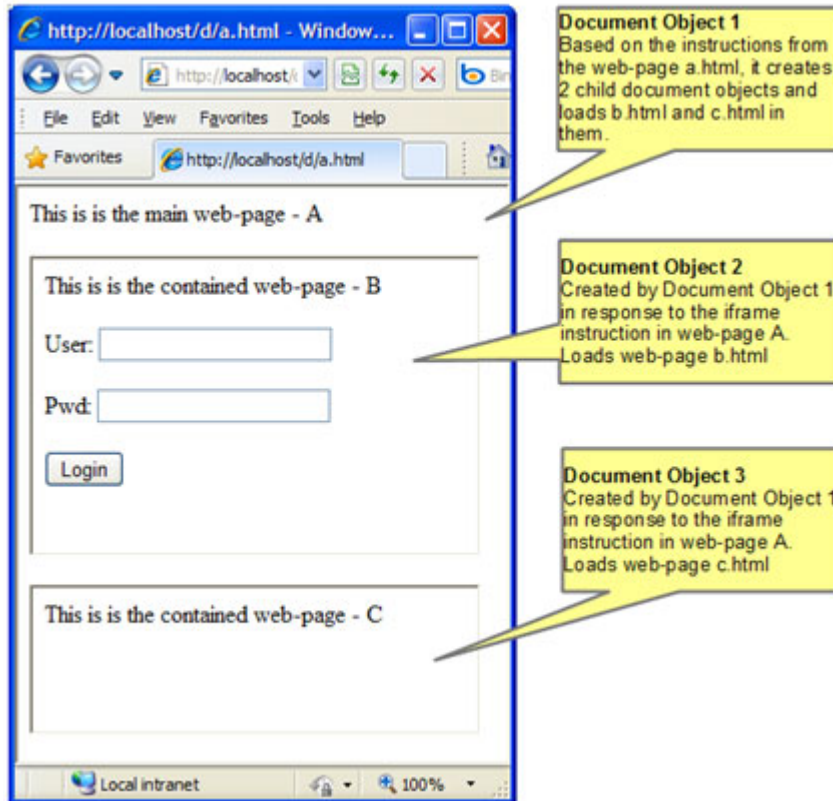


Figure 4 Page a.html with frames

This time, when the user clicks **Login**, instead of the document object 2 loading another web page, the main document 1 itself moves to another web page.

Example 20 shows the modified b.html. Compare it to the b.html of case 1. You will notice that it is the parent document (document 1) navigating this time and not the current document.

Example 20 Modified b.html

```

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title></title>
</head>
<body>
<script type="text/javascript">
function clicked() {
window.parent.document.URL="e.html";
}
</script>
This is is the contained web page - B
<br /><br />
User: <input type="text" id="username" />
<br /><br />
Pwd: <input type="text" id="pwd" />
<br /><br />
<input type="button" id="submit" onclick="clicked();" value="Login"/>
</body>

```

</html>

Example 21 shows the markup for e.html.

Example 21 e.html

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title></title>
</head>
<body>
This is is the web page E
<br /><br />
Successfully Logged in!
<br /><br />
<a href="a.html">Back to a.html</a>
</body>
</html>
```

Figure 5 shows what displays when the user clicks **Login**.

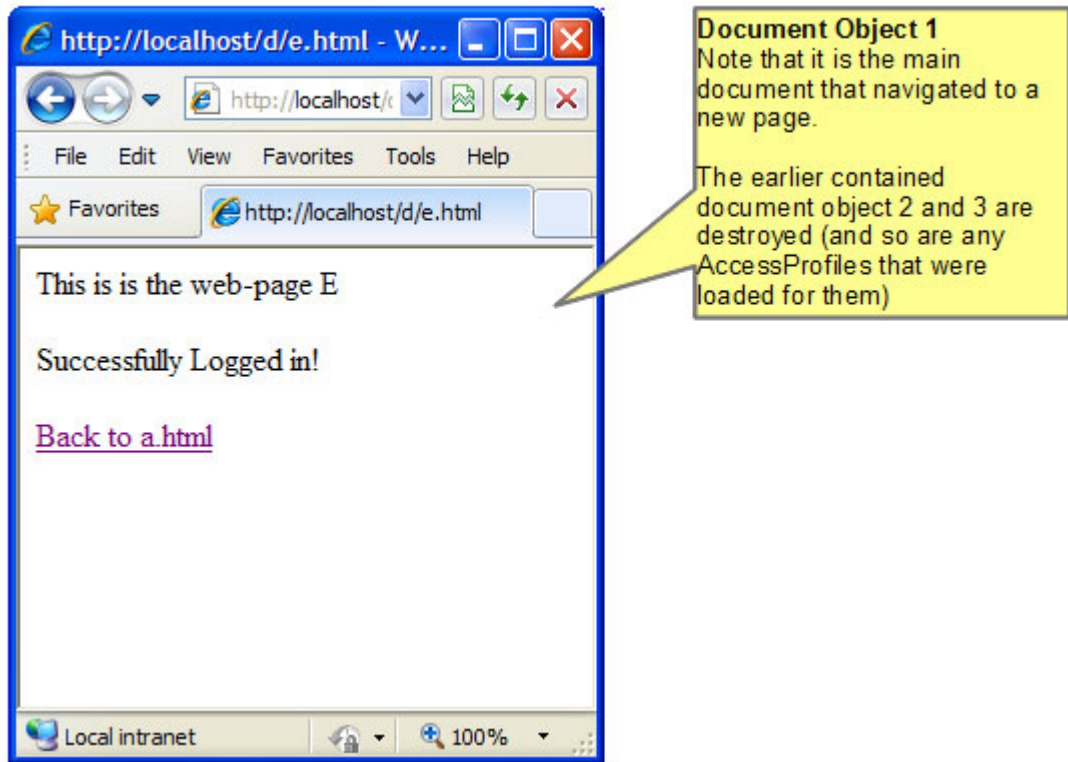


Figure 5 The main document navigated to a new page

If there was a non-validating AccessProfile for b.html (the capture and save of credentials was happening on the **Login** button click event), it works as the workflow that the AccessProfile is interested in would be complete before document object 2 (and the attached SSO AccessProfile) is destroyed.

However, an AccessProfile, which was trying to do a validated login and was thus waiting for e.html to confirm that login happened successfully, cannot work because it gets destroyed on

navigation of the main document object, which destroys all child document objects (including the one to which the AccessProfile was attached).

The typical symptom of this problem in the real world is that the logs show a successful firing of the html click trigger and the capture action, but the following web page completes loading the trigger with the save action that never happens. Instead, the logs might show AccessProfile loading logs (typically of the same AccessProfile or of auto-learn) because the main document object, when navigating to the new page, might be loading new frames for the successful login page, which requires their own AccessProfile.

Solution 1

One work around is to use a global account-data-bag (set Use local bag = No) to temporarily store the credential captured, so it can be retrieved later by another AccessProfile, which is meant to detect the successful login page, as shown in Figure 6.

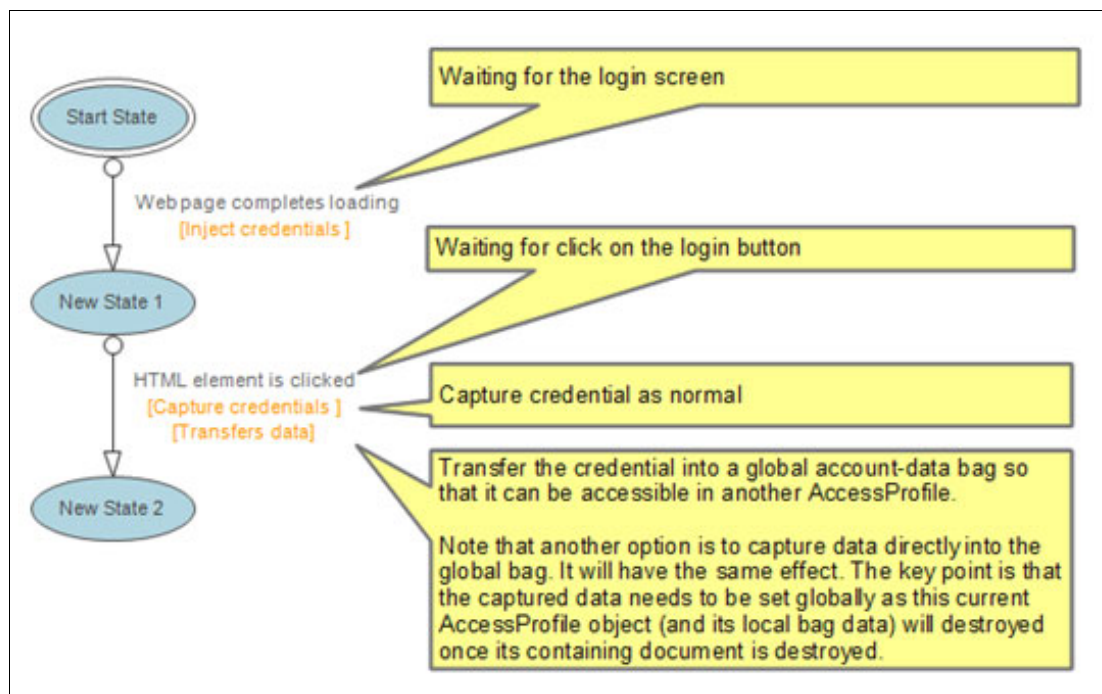


Figure 6 AccessProfile for capturing credential

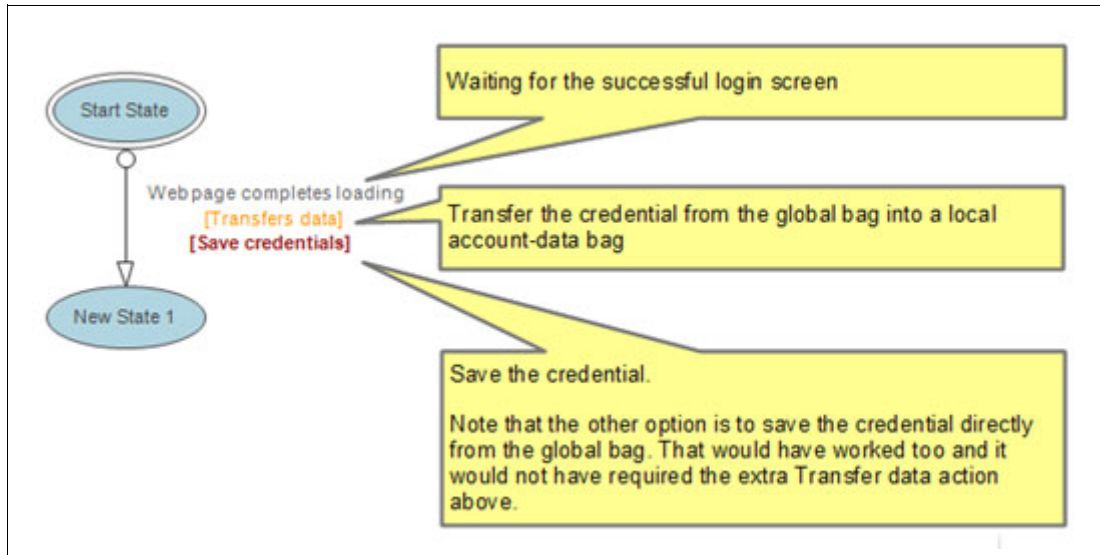


Figure 7 AccessProfile for saving credential

As an optimization, the two AccessProfiles can be combined into one, as shown in Figure 8.

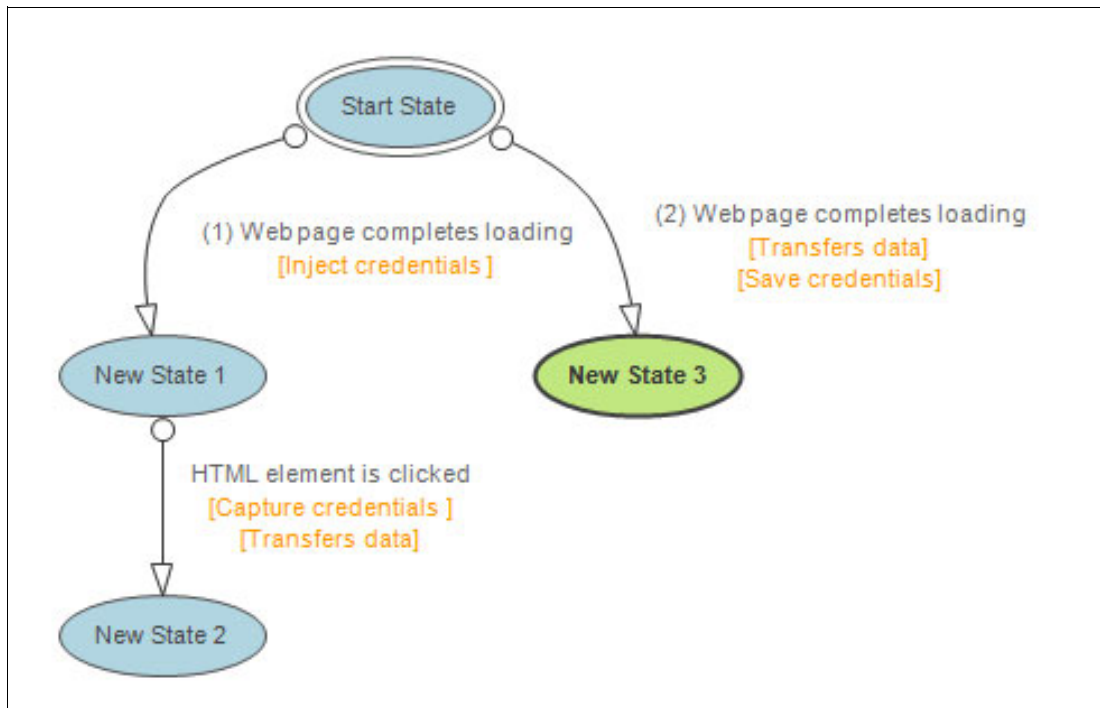


Figure 8 Combined AccessProfile

Here, the first web page completes loading trigger handles, the login screen, and data capture, while the second web page completes loading trigger handles the saving of the credential. The use of global data between these two branches remains the same as Figure 8.

The first instance of this AccessProfile will get loaded for the login screen. It will set the global account-data for capture, and it will get destroyed along with the containing frame. The second instance of the same AccessProfile has the second web page complete loading trigger fire when it sees the successful login page.

Solution 2

Another mechanism that does not involve using global bags is to write an AccessProfile for the main document with frame aware signatures used to extract the necessary user name and password data from the child documents and to monitor the click of the **Login** button.

The signatures created by AccessStudio Signature Generator assume that the AccessProfile is meant to be written for the same document object as the element where it is dragged and dropped. Example 22 shows the signature for the Login button in the example shown in Figure 8 on page 18.

Example 22 Sample signature

```
/descendent::html[@tag_name="input" and @name="" and @type="button" and  
@value="Login" and @id="submit"]
```

With this signature, the Login button is only going to be found in document object 2. Other document objects (1 and 3) will search for this button within their web pages and will not find it.

If we prepend this signature with the following signature, we get the signature shown in Example 23.

```
/child::html/ parent::frame/descendent::document[@url#"*.f.html.*"]
```

Example 23 Sample signature

```
/child::html/  
parent::frame/descendent::document[@url#"*.f.html.*"]/descendent::html[@tag_name="i  
nput" and @name="" and @type="button" and @value="Login" and @id="submit"]
```

This signature can access the Login button from all of the documents currently loaded in that browser tab.

Implementation Note: The directive `/child::html/parent::frame` looks arbitrary and, unfortunately, it is. It is a leftover of an older signature mechanism that refers to a hypothetical parent to all of the documents in the browser tab, and it cannot be removed because of backward compatibility restrictions. In a future release, the Observer will provide a more intuitive directive to refer to all of the documents, but it will keep on supporting this directive, so it is safe to use this mechanism.

Similarly, the signatures of the username and password fields can be converted to the signatures shown in Example 24 and Example 25.

Example 24 Sample signature

```
/child::html/  
parent::frame/descendent::document[@url#"*.f.html.*"]/descendent::html[@tag_name="i  
nput" and @name="" and @type="text" and @id="username"]
```

Example 25 Sample signature

```
/child::html/  
parent::frame/descendent::document[@url#"*.f.html.*"]/descendent::html[@tag_name="i  
nput" and @name="" and @type="text" and @id="pwd"]
```

This allows writing the AccessProfile in context of document object 1 and not document object 2. Because document object 1 is still around when the navigation occurs to the success screen (unlike document object 2 which is destroyed), just a web page completes loading trigger and can be added to the AccessProfile of Document object 1, which handles the success screen and saves the credential when that trigger fires.

Fortunately, the web page completes loading trigger of the top level document (document object 1) occurs after all the other child documents' web document complete events already occurred. This means that we can reliably expect the child documents and their HTML elements to be present if we want to access them (like for injection, as shown in Figure 9) in the webpage completes loading trigger of the top level document.

The AccessProfile's site signatures thus must match the top level document. Example 26 shows the signature.

Example 26 Sample signature

```
/child::web[@domain="localhost" and @protocol="http" and @url~".*a.html"]
```

The AccessProfile structure looks like Figure 9.

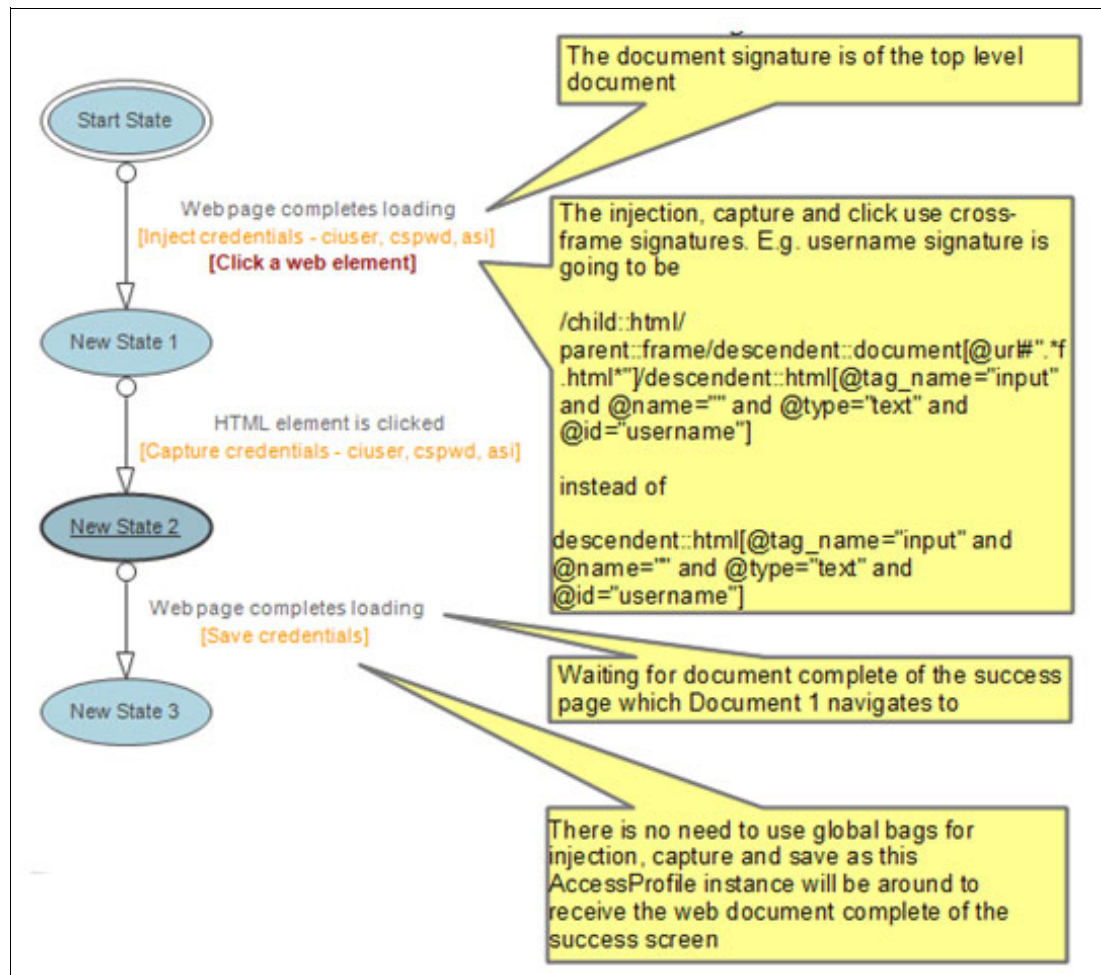


Figure 9 The AccessProfile structure

Solution 1 versus Solution 2

Solution 2 (using cross frame signatures) simplifies the AccessProfile; however, it is written with the assumption that document object 1 is the top level document. Some web sites reuse the login page document (document object 2) by both embedding it in another document, such as in Figure 9 on page 20, in certain cases, and in others scenarios showing it as the top level document.

So, the user can see both of the following in several workflows for the web application. Alternately, this means that the AccessProfile, which was relying on document object 1, can no longer get loaded in this independent login page.

Solution 1 (using global bags) does not have this problem because it does not care where the login screen is hosted (independently or under a frame). It does however have to deal with global bags that need to be cleared at an appropriate time to avoid reusing stale data and other problems.

The SSO/automation workflow needs to access fields from several documents. Consider the web pages shown in Example 27, Example 28, and Example 29.

Example 27 h.html (contains 2 frames)

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title></title>
</head>
<body>
<script type="text/javascript">
function clicked() {
window.parent.document.URL="e.html";
}
</script>
This is is the main web page - H
<br /><br />
<iframe src="g.html" height="100" width="300"></iframe>
<br /><br />
<iframe src="f.html" height="100" width="300"></iframe>
<br /><br />
<input type="button" id="submit" onclick="clicked();" value="Login"/>
</body>
</html>
```

Example 28 g.html (contains the username field)

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title></title>
</head>
<body>
This is is the contained web page - G
<br /><br />
User: <input type="text" id="username" />
</body>
</html>
```

Example 29 f.html (contains the password field)

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
<head>
<title></title>
</head>
<body>
This is is the contained web page - F
<br /><br />
Pwd: <input type="text" id="pwd" />
</body>
</html>
```

When loaded, h.html looks like Figure 10.

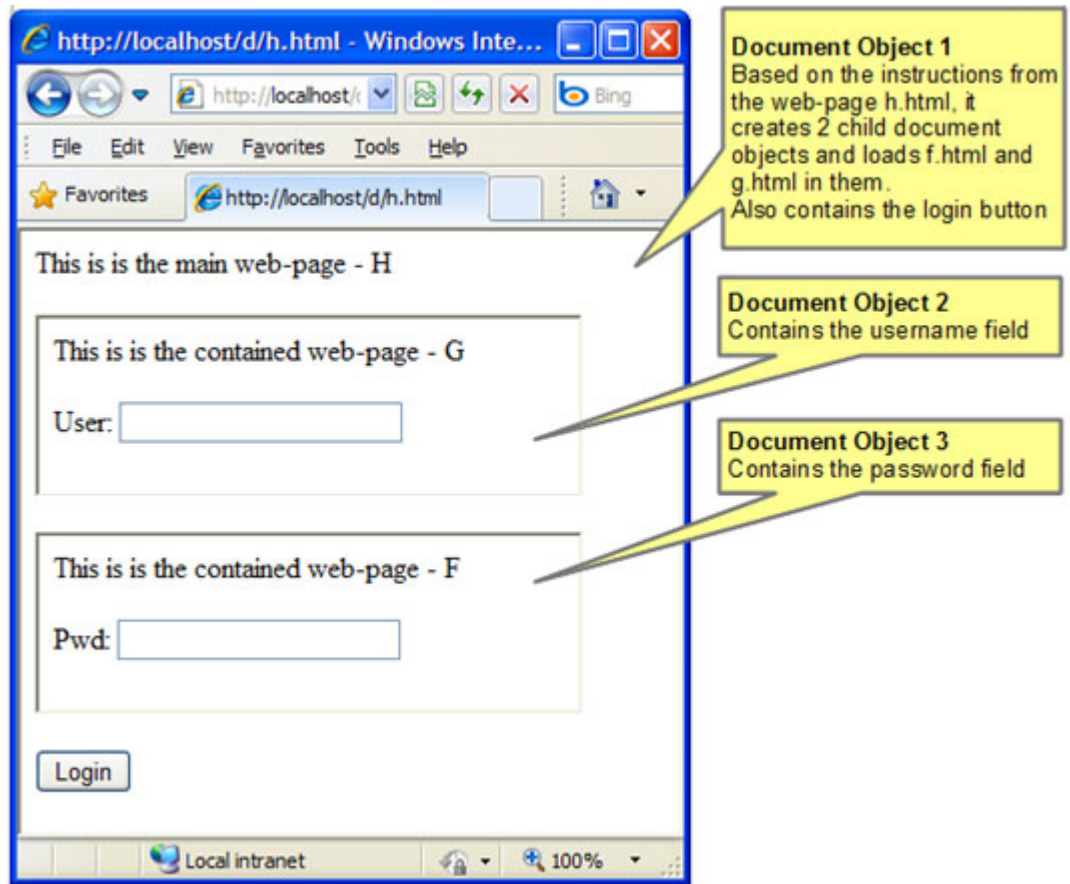


Figure 10 h.html

For successfully capturing the credentials, the AccessProfile must retrieve data from both document object 2 and document object 3. In this case, the solution is to use a similar technique described in "Solution 2" on page 19.

The idea is to use cross-frame signatures to refer to HTML elements that are not contained in the same document. It is usually easiest to write the AccessProfile for the top level document object (document object 1) and use cross-frame signatures to get data and monitor events for elements not in the same frame.

The AccessProfile thus looks like the one for solution 2, case 2, with the difference that the signature for the login screen will stay as the following example because it is in the same document as the AccessProfile instance:

```
/descendent::html[@tag_name="input" and @name="" and @type="button" and @value="Login" and @id="submit"]
```

The username signature changes to that shown in Example 30.

Example 30 Sample username signature

```
/child::html/parent::frame/descendent::document[@url#"*.g.html.*"]/descendent::html[@tag_name="input" and @name="" and @type="text" and @id="username"]
```

The password signature changes to that shown in Example 31.

Example 31 Sample password signature

```
/child::html/parent::frame/descendent::document[@url#"*.f.html.*"]/descendent::html[@tag_name="input" and @name="" and @type="text" and @id="pwd"]
```

Note that the url attribute of the document is the one differentiating between f.html and g.html.

Frames support in v8.1

IBM Tivoli Access Manager for Enterprise Single Sign-On V8.1 introduced the concept of a state-machine-id that helped track the life-time of an AccessProfile instance. This state-machine-id view can be turned on by adding the registry entry shown in Example 32.

Example 32 Registry key entry

```
Key: HKEY_LOCAL_MACHINE\SOFTWARE\Encentuate\AccessStudio  
Name: ShowStateEngineIds  
Type: DWORD  
Value: 1
```

This displays the state-machine-id next to every AccessStudio log. Using this log, it is possible to track the state-transitions, triggers fired, action executed, and so on against the right state-machine when multiple state-machines are getting loaded because of frames.

It is also possible to log the current URL for which a state-machine is loaded. This can be done by inserting a dummy Data Transfer action with the following parameters.

From:

Type = Web control

Signature = /child::html[@tag_name="BODY"]/@url

To:

Type = Property store item

Name = var_dummy_url

This would create a log like the following:

```
[State Machine Id - 8] Action: Transfers data. Property var_dummy_url is set to 'http://localhost/d/h.html'
```

This can help isolate the relevant state-machine for the URL of interest.

Frames support in v8.2

There are no changes in IBM Tivoli Access Manager for Enterprise Single Sign-On V8.2 in the way Observer internally handle frames; however, the concept of the Document is now visible throughout AccessStudio and Observer logs.

Every browser process log in AccessStudio now shows a Document and State Machine ID, making it easier to track the number of documents (and the frames) created for a page and the lifetime of AccessProfile instances loaded for it and the URL associated with it at a given instance.

Differences between Firefox and Internet Explorer AccessProfiles

In general, an AccessProfile written for a web site in IE would also work for Firefox and vice versa. As all browsers handle HTML and JavaScript slightly differently, the server side code generating the content (ASP, JSP, CGI) can return a different version of HTML back to the browser based on its type. This might, in rare cases, invalidate a signature (as the attribute or the hierarchy of HTML elements might not be the same between the two browsers).

We recommend that you test the AccessProfile in both of the browsers before deploying it in a mixed browser environment.

Some of the HTML element methods and properties available are also different between the two browsers. This is usually not too much of an issue with the signatures, but if the HTML Document is accessed directly using an AccessProfile plug-in (using the `runtime.GetHTMLDocument()` call), then the way the Document object is used by the plug-in script might need to be different for IE and Firefox, for example, the ID attribute value is case-insensitive in IE 7.0 and below, but not in Firefox and IE8 and above. As this is not an AccessProfile specific issue, a web search on browser differences while writing JavaScript directly should help reveal other such areas of incompatibility.

Common issues

In this section, we discuss common issues.

Slowness due to multiple auto-learn loads

In this section, we discuss slowness that is because of multiple auto-learn loads.

Symptom

Navigation to a web site that does not have any AccessProfile defined for it slowed down noticeably after installing ISAM ESSO.

Observations

There is a large number (more than 20 or so) of auto-learn AccessProfile loads happening as the web page loads in AccessStudio logs.

Probable cause

This might be because of the web page using a large number of frames or internally doing multiple navigations before it settles to the final page, each of which loads the auto-learn AccessProfile.

Resolution

Create two conflicting dummy AccessProfiles (just the right site signature and one start state with no triggers) for the urls seen in Observer logs for which the auto-learn AccessProfile is being loaded while the web page exhibiting slowdown is loaded.

The line in the log to look for to figure out the URL in Observer logs is:

```
[CEnBrowserListener::DocumentComplete] Analyzing document URL is <URL>
```

Follow a few lines down in the log with this:

```
[::GetApplicationObject] The AccessProfile id is sso_site_web_auto_learn
```

Typically, these URLs have a common domain or a path that can be used to create a common site signature. If not, an AccessProfile allows specifying multiple site signatures. Either way, there should not be any need to create more than one set of conflicting AccessProfiles that can take care of all the cases where a deliberate conflict must be created.

In V8.2, the URL used to load an AccessProfile is evident from AccessStudio logs.

In rarer cases, where the customer is not interested in the auto-learn facility, it is simpler to just turn it off using the `pid_sso_auto_learn_enabled` AccessProfile.

Further explanation

As described earlier in the document, the auto-learn AccessProfile is loaded for each web page where there is no explicit AccessProfile defined. If a web page has multiple such navigations before it is displayed, the cumulative overhead of fetching an AccessProfile for each of the navigations adds up causing the delay.

Recall that an AccessProfile is fetched in two steps. First, its ID is resolved based on the information provided by the agent to DataProvider and then the AccessProfile object is fetched by the agent by passing DataProvider the ID it just retrieved. The latter step takes more time.

By creating a deliberate conflict, the agent is told that multiple AccessProfiles were found for the web page whose details it passed to the DataProvider. On receiving this error, the agent does not try to retrieve any AccessProfile object from the DataProvider, saving time and hence speeding up the page load.

No injection as HTML element was not found

Symptom

Injection does not occur even though the capture works fine.

Observations

AccessStudio logs show that an HTML element was not found, but when checked with the signature generator the element highlights without any error.

Probable cause

This usually is a timing issue. A typical web AccessProfile tries to do injection and click just after the web page completes loading trigger fires. However, when the user is creating the signature of the various elements through the signature generator, other events in the web page might have already occurred that changed the element (or its ancestors' properties).

This might mean that the signature generated at this later time in the lifetime of the page is not valid immediately after the web page completes loading trigger when injection and click are being performed.

Resolution

Figure 11 shows what a typical web AccessProfile looks like.

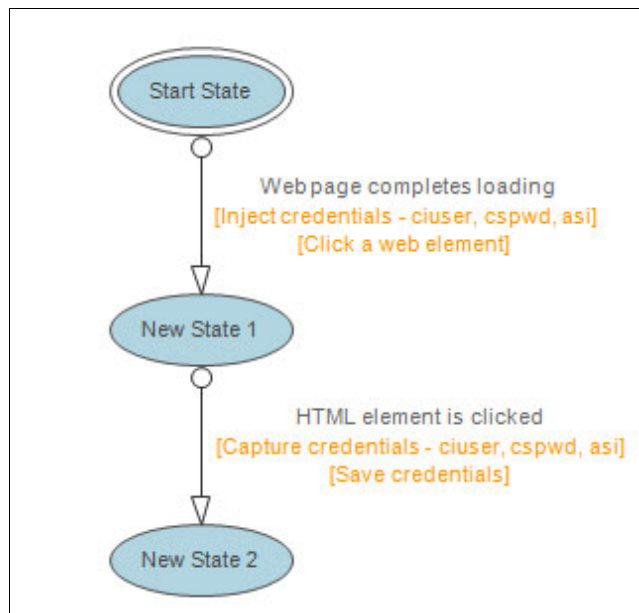


Figure 11 Typical AccessProfile

The first step is to convert this AccessProfile, as shown in Figure 12 on page 27.

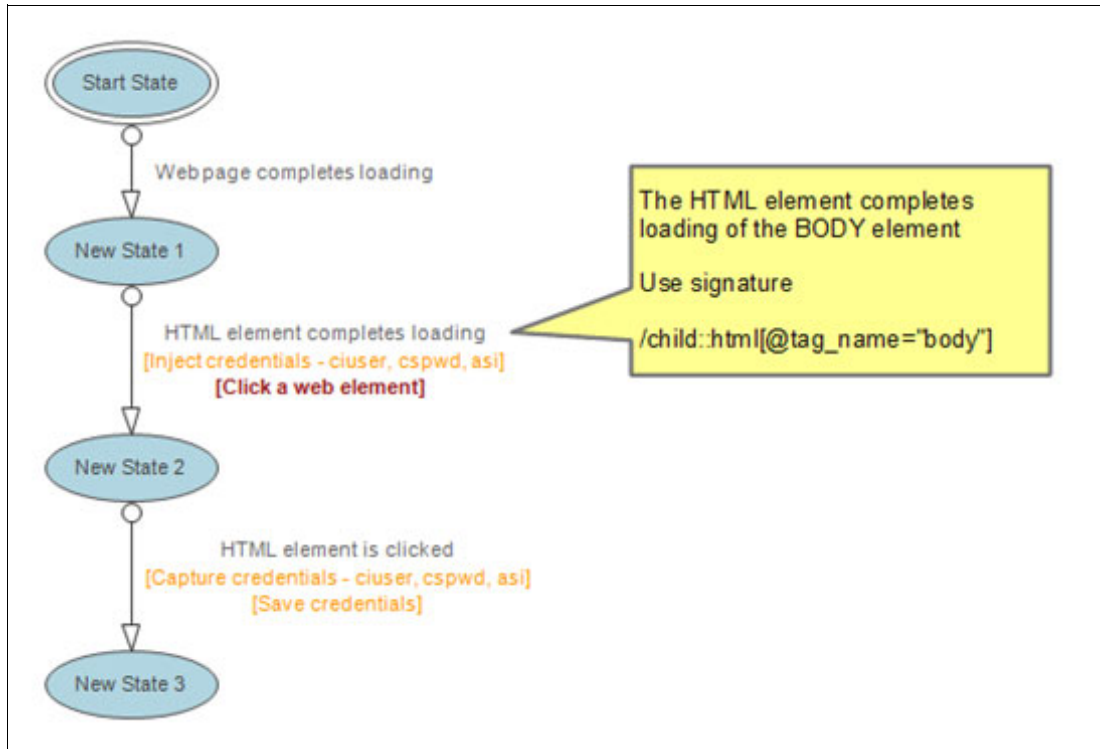


Figure 12 Converting the AccessProfile

This delays the injection and click until after the BODY element's onload script (if any) fires. As the signature in the AccessProfile was also created after the onload script had executed, it increases the chances that the signature generated matches the state of HTML elements at this time.

If AccessStudio logs still show HTML element not found error for the username, password, or Submit button, further modify the AccessProfile to the following (assuming in this example that the username input control was the INPUT control not found). See Figure 13 on page 28.

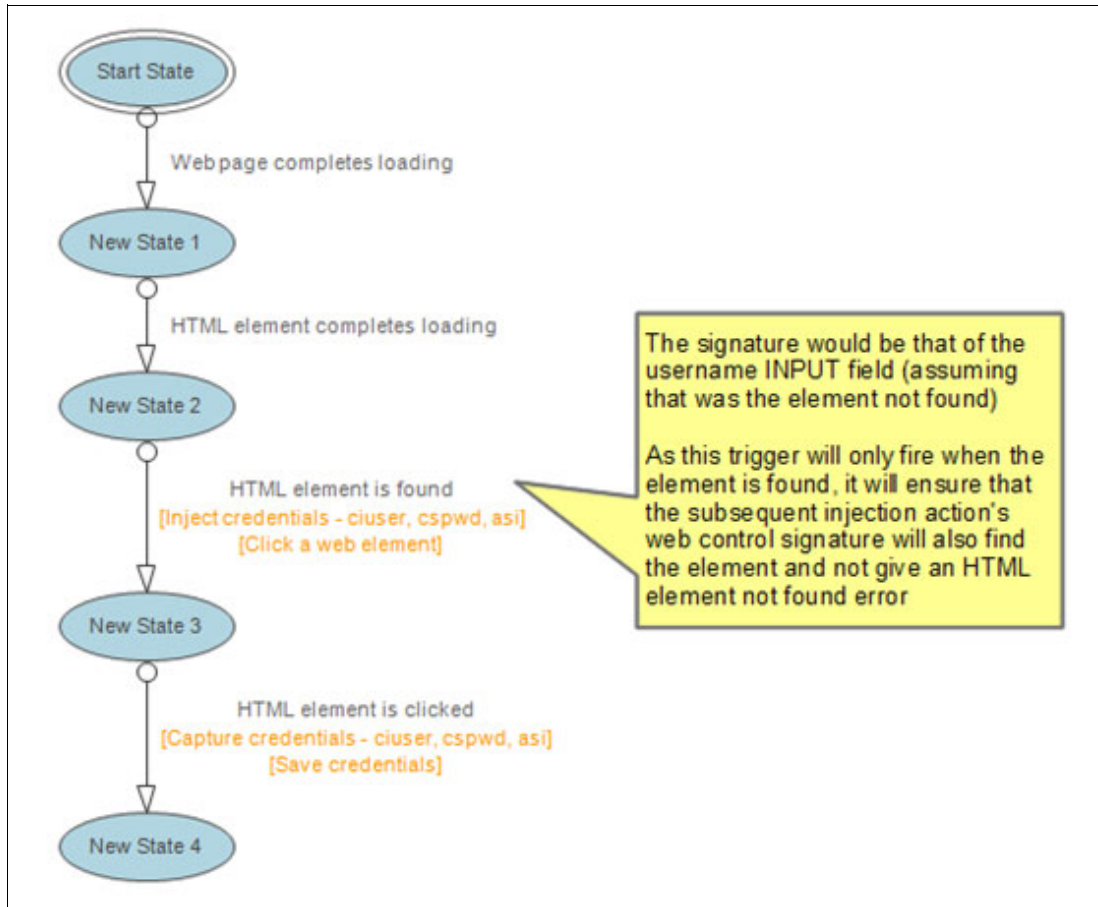


Figure 13 With HTML element is found trigger

Further explanation

Actions, such as clicking an HTML element or injection into a web control and triggers, such as waiting for a click of an HTML control assume that the application is in a state when the control described by the signature is already present. These triggers, actions, and web controls do not wait for the control to become available at a future time, they only check for its presence one time when they execute and, if not found, just fail with a log in AccessStudio.

This is done to ensure that Observer does not drag down the system performance as otherwise for every such trigger, action, or web control there are extra resources consumed to just wait for the element to be found.

Instead, Observer provides triggers, such as HTML element completes loading, which helps track the state of the application closely and, in rare cases, where the only choice is to wait for an element to show up before any trigger or an action can be performed on it, it provides an explicit HTML element found trigger (also the same logic for window found trigger for windows application).

Before the HTML element was found, trigger was made available in V8.1, the AccessProfile writer just used a Fire after specified time trigger to wait for some time before injection (and hope the controls are all present by then). The use of fire after a specified time trigger is discouraged because the necessary time-out values vary from machine to machine (based on load and processing power).

No injection and no HTML element not found error

In this section, we discuss the no injection and no HTML element not found error.

Symptom

Injection does not occur even though capture works fine.

Observations

AccessStudio logs show that injection action fetched the credentials from the wallet properly (it shows the account data bag getting populated with the right username and authentication service ID) and there are no HTML element not found errors during injection.

Possible cause

The injection might have happened, but then the INPUT HTML elements where injection was performed got cleared by some other JavaScript in the web page. This typically occurs in the OnLoad event handler of the BODY element which might be clearing the INPUT boxes.

Resolution

Delay the injection until after the OnLoad event of the BODY element occurs. Use the HTML element completes loading to wait for the BODY element to get loaded before doing injection. See Figure 14.

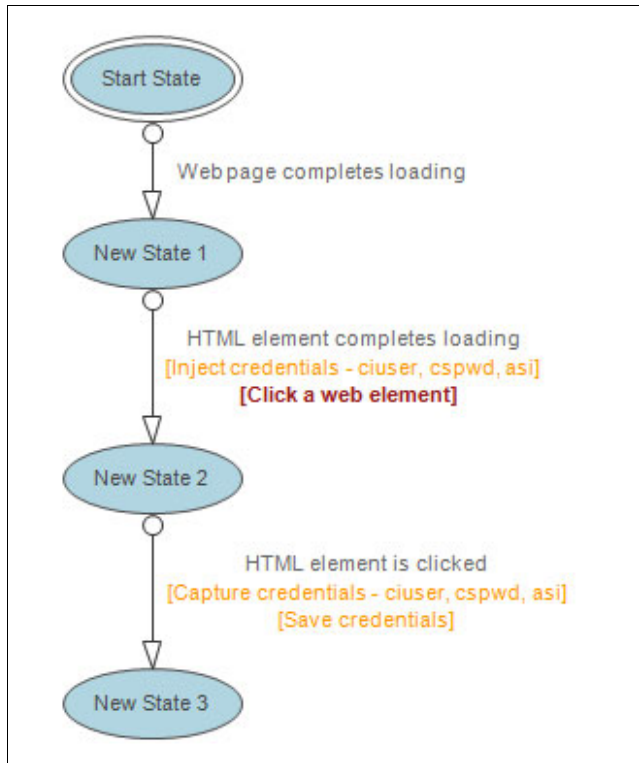


Figure 14 Using onload event to delay injection

Further explanation

The resolution is the same as the first one tried in the case where there was an HTML element not found error in AccessStudio logs. In that scenario, the HTML element's properties were being modified in such a way that the signature of the element was not matching before the OnLoad event.

In this case, the OnLoad event is not modifying any properties of the HTML element that had any impact on signature evaluation (it was still getting the right element), it is just that the OnLoad event was clearing the contents of the INPUT element and that meant that any injection done before this event was being erased just afterwards giving the impression that no injection happened.

Captured credentials are obfuscated when saved

In this section, we discuss the captured credentials are obfuscated when saved error.

Symptom

The AccessProfile is capturing obfuscated credentials, such as the following string - "*****".

Observations

In case the username is being captured obfuscated, the web page converts the entered username partially or entirely to the following string - "*****" when the user selects another field (typically password) after entering a username.

Possible causes

The web site is using a JavaScript to obfuscate the username or password display for security reasons whenever the INPUT control loses focus.

Resolution

Use the HTML element lost focus trigger to retrieve the credentials before they get obfuscated by the JavaScript. The AccessProfile looks like Figure 15.

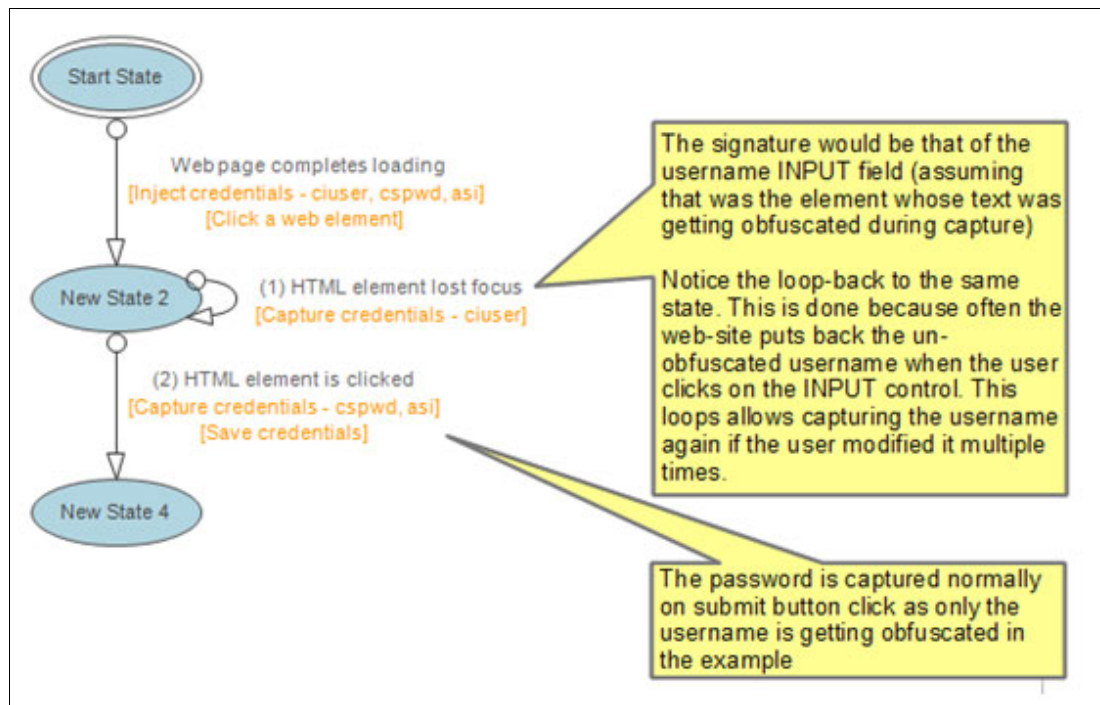


Figure 15 AccessProfile with HTML element lost focus trigger

In some cases, the web page might obfuscate the username as it is being typed. In this case, the web page keeps the actual username in some hidden HTML element, typically another INPUT element with type set to HIDDEN.

The exact mechanism can only be determined by examining the HTML and the embedded JavaScript in the web page. One option is to use the Run a VBScript or JavaScript action under the HTML element clicked trigger to access the HTML DOM directly and retrieve the value.

Example 33 shows a sample VBScript.

Example 33 Sample VBScript

```
' Get the current HTML Document
set doc=runtime.GetHTMLDocument()

' Assuming the userid was stored in a different form – frmPoster's hidden ssn
' field
userid=doc.frmPoster.ssn.value
set propcon=runtime.GetPropertiesContainer()

'Assumes that the capture bag is already created
propcon.SetAccDataItem("default_capture_bag", "aditi_ciuser", userid)
```

This VBScript retrieves the value from an HTML page, which looks like Example 34 and uses the frmPoster's hidden input controls to temporarily keep the username while the INPUT control, which is showing the username, is obfuscated.

Example 34 Sample HTML

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title></title>
</head>
<body>
<form name="frmPoster" id="frmPoster" method="POST" action="abc\def" >
<input type="hidden" maxlength="15" name="ssn">
<input type="hidden" maxlength="12" name="pin">
</form>
</body>
</html>
```

Another option is to set the signature of the web control in the capture field to refer to the HIDDEN field, as shown in Example 35. Note that it is not possible to create this signature by the Signature Generator's drag and drop facility as it is a hidden facility and there is nothing visible to drag and drop the control selector on to.

Example 35 Sample signature

```
/descendent::html[@tag_name="form" and
@name=""frmPoster]/descendent::html[@tag_name="input" and @name="ssn" and
@type="hidden"]
```

Further explanation

HTML elements provide an `onBlur` event that is called when the control loses focus and can be handled by using the HTML Element lost focus trigger. It is possible for a web page to trap this event and write JavaScript for it which transfers the original text in the INPUT element to a hidden HTML and modify the display text.

Save action not firing

In this section, we discuss the save action not firing error.

Symptom

Save not happening for a validating AccessProfile.

Observations

The trigger that contains the save action (typically a web page completes loading trigger of the validation page) never fires.

When the click happens, the AccessStudio logs show the correct capture of the credentials and then it unexpectedly shows the auto-learn AccessProfile getting loaded and there are no further logs for the current AccessProfile instance (identified by the State Machine Id in the logs).

If there was no auto-learn AccessProfile present (usually happens in AccessStudio Test mode when there is no auto-learn AccessProfile available), instead of the auto-learn AccessProfile loading log, the following is shown:

```
AccessProfile: <AccessProfileId>. Unloaded because it did not handle the URL: <URL>
```

In both cases, the URL used to unload the current AccessProfile instance and load the auto-learn AccessProfile can be seen in the Observer logs as well.

Possible cause

This is probably happening because the state-machine was in a state that did not handle the intermediate navigation to the URL, and the site signatures of the current AccessProfile did not match this URL either. This caused the unloading of the AccessProfile instance and hence the necessary web page completes loading trigger that was to detect the presence of the validation screen never fired.

This can also be happening because the web application was using frames in a way that the document object under which the save and capture were supposed to occur was destroyed. In this case, use the techniques outlined in “Frames and the web browser document object” on page 10.

Resolution

Either put in a webpage completes loading trigger for the URL, causing the AccessProfile unload in the state with the web page completes loading trigger containing the save action, or add or modify the site signatures such that this URL matches this AccessProfile.

In both cases, the AccessProfile instance will no longer unload. If there were multiple such navigations happening that caused the AccessProfile instance to unload, this addition of the webpage completes loading trigger or the site signature modification might need to be done multiple times.

Figure 16 on page 33 describes the webpage completes loading trigger process.

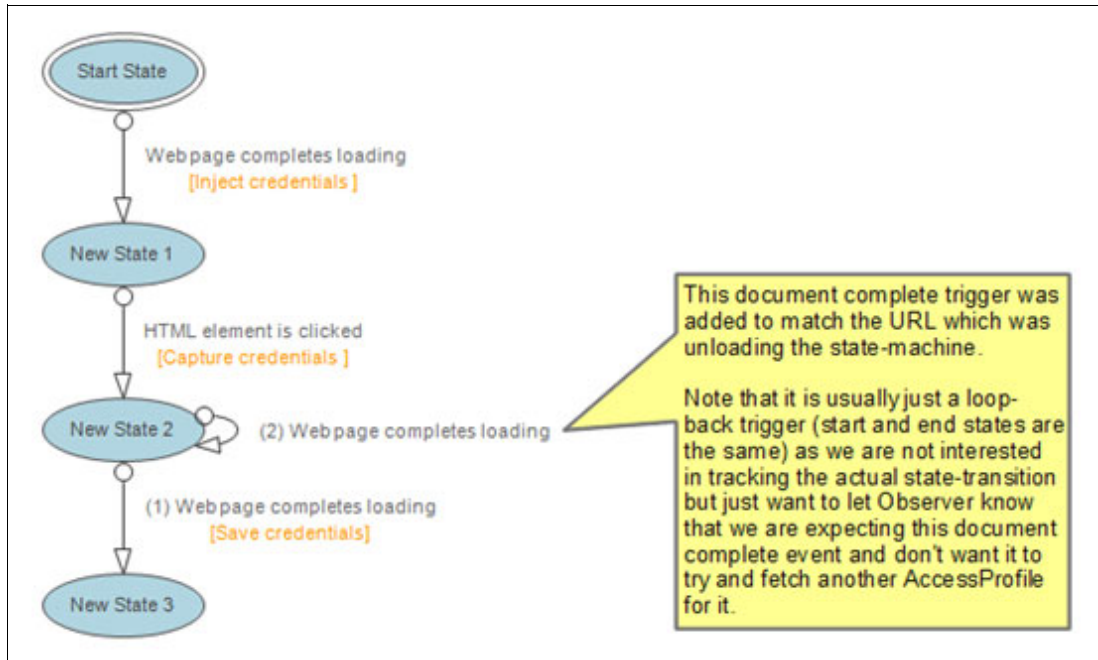


Figure 16 Using webpage completes loading trigger to prevent AccessProfile unloading

Further explanation

See “Document complete event and the Observer” on page 4 for further details about the lifetime management of an AccessProfile instance.

Slow injection or clicking when using fire after some time trigger

In this section, we discuss the slow injection or clicking when using fire after some time trigger error.

Symptom

Slow credential injection or a button click in a web page.

Observations

It takes a couple of seconds for the injection into web controls to occur when the injection is being done under a fire after specified time trigger. Similarly, clicking an HTML element is slower as well.

Possible cause

This is an implementation limitation on Observer side. Internally, the fire after specified time runs its actions under a different thread. Windows OS does additional processing when the HTML Document is accessed from a different thread than the one which created it.

This is the same reason why AccessStudio signature creation and highlight functions are slower than a typical injection to the same web control.

Resolution

In V8.1, avoid doing injection under a fire after specified time trigger and after a wait for some time action. Instead, use the HTML element found trigger if the delay was necessary for the HTML element to be created. This limitation will be removed in V8.2.

Further explanation

The HTML element found trigger runs in the same thread as the one that owns the Document object. It is also a better way to wait for an HTML element to be created rather than using a fire after specified time trigger, which can give unpredictable results as different machines and browsers might take different amounts of time to create an HTML element.

No AccessProfile loading for a non-browser hosted web page

In this section, we discuss the no accessprofile loading for a non-browser hosted web page error.

Symptom

No AccessProfile is being loaded for the web page.

Observations

The web page is being displayed in a non-browser application.

Possible cause

The web SSO agent is not loaded for non-browser applications and hence there is no automation or SSO.

Resolution

Use the Start Installing BHO action for the window that needs to have automation and monitoring support enabled. A separate web AccessProfile is written for the web page that is hosted in this application, the same way it would have been if this web page were hosted in a normal browser.

Figure 17 describes the Start Installing BHO action.

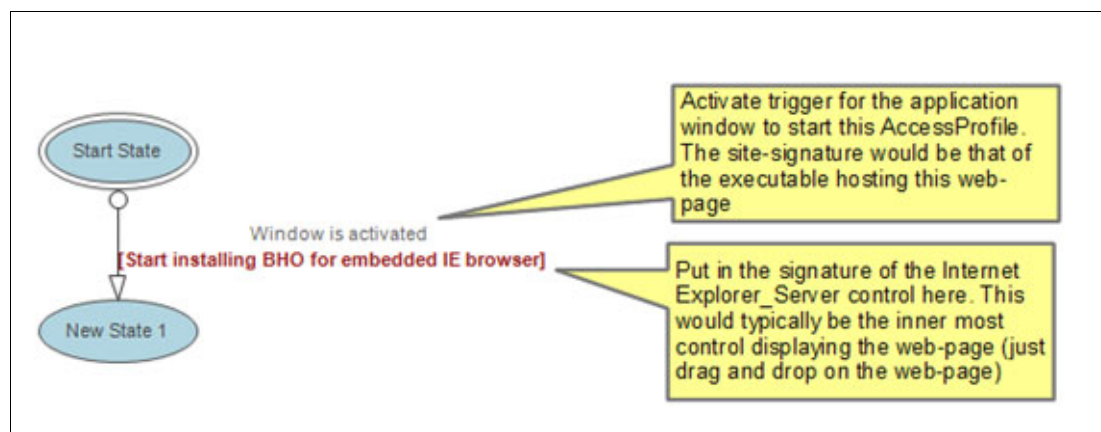


Figure 17 Using start installing BHO action

Further explanation

Internet Explorer provides a reusable browser window that can be embedded in other applications, such as Word, Powerpoint, and other applications that at times embed the browser window within their own window.

Observer uses a plug-in mechanism in Internet Explorer called Browser Helper Object (BHO) to load itself in its process and monitor the web content. If the Internet Explorer component for displaying the web page is embedded in another application, the BHO does not load for that

application and thus there is no SSO. The start installing BHO action instructs the Internet Explorer component to load the BHO and enables the usual web monitoring and automation.

This mechanism to force the application to load the BHO only works if it is using the Internet Explorer_Server component that comes with Internet Explorer. This is the reason for identifying the Internet Explorer_Server control in the Start installing BHO action.

There is a corresponding Stop installing BHO action as well. Usually, there is no need to call it, but in rare cases the application has multiple embedded browser windows and the SSO support needs to be made available for only one of those windows, then the Stop installing BHO action can be used after the BHO has been installed by start installing BHO action for the window specified by its signature.

The team who wrote this paper

This paper was produced by a team of specialists from around the world working at the International Technical Support Organization, Rochester Center.

Axel Buecker is a Certified Consulting Software IT Specialist at the International Technical Support Organization, Austin Center. He writes extensively and teaches IBM classes worldwide on areas of Software Security Architecture and Network Computing Technologies. He has a degree in Computer Science from the University of Bremen, Germany. He has 25 years of experience in a variety of areas related to Workstation and Systems Management, Network Computing, and e-business Solutions. Before joining the ITSO in March 2000, Axel worked for IBM in Germany as a Senior IT Specialist in Software Security Architecture.

Mohit Chugh is an Engineering Manager with Tivoli Access Manager for Enterprise Single Sign-on where he runs the team that is responsible for the client-side UI automation and Single Sign-on. He has a degree in Electronics and Communication from Delhi University and has 12 years of software development experience in the area of enterprise security and consumer software.

Thanks to the following people for their contributions to this project:

Matthew Duggan
Ken Yian Chow
Nandagopal Seshagiri

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Obtain more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Stay connected to IBM Redbooks

- ▶ Find us on Facebook:
<http://www.facebook.com/IBMRedbooks>
- ▶ Follow us on Twitter:
<http://twitter.com/ibmredbooks>
- ▶ Look for us on LinkedIn:
<http://www.linkedin.com/groups?home=&gid=2130806>
- ▶ Explore new IBM Redbooks® publications, residencies, and workshops with the IBM Redbooks weekly newsletter:
<https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm>
- ▶ Stay current on recent IBM Redbooks publications with RSS Feeds:
<http://www.redbooks.ibm.com/rss.html>

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 give 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.

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 websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites 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.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

This document REDP-4767-00 was created or updated on August 15, 2011.



Send us your comments in one of the following ways:

- ▶ Use the online **Contact us** review Redbooks form found at:
ibm.com/redbooks
- ▶ Send your comments in an email to:
redbooks@us.ibm.com
- ▶ Mail your comments to:
IBM Corporation, International Technical Support Organization
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400 U.S.A.




Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.shtml>

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

IBM®
IMS™

Redbooks®
Redpaper™

Redbooks (logo) ®
Tivoli®

The following terms are trademarks of other companies:

Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.