LaunchKey .NET SDK Documentation

API Documentation & Examples

Contents

LaunchKey .NET SDK Usage

Overview

This section covers using the LaunchKey .NET SDK library directly and covers basic use cases.

Initializing the Library

To begin using the library, we must first create an instance of `LaunchKeyRestClient`, supplying it with your app key, app secret, and app private key.

```
var appKey = "123457890";
var appSecretKey = "mysecretkey";
var appPrivateRsaKey = File.ReadAllText("myprivatekey.rsa");

var client = new LaunchKeyRestClient(
          appKey,
          appSecretKey,
          appPrivateRsaKey
);
```

Authenticating a User

To authenticate a user by sending a launch request, we use the `Authenticate()` method.

```
var authResponse = client.Authenticate(“launchkey_username”, AuthenticationType.Session);
if (!authResponse.Successful)
{
          // Launch request failed for some reason, check authResponse.MessageCode, bail
}
```

Once the launch request has been initialized successfully, periodically check the status of the request. This is accomplished by calling the `Poll()` method on `LaunchKeyRestClient`, as seen below:

```
PollResponse pollResponse;
do
{
          Thread.Sleep(1000);
          pollResponse = client.Poll(authResponse.AuthRequest);
} while (pollResponse.MessageCode == LaunchKeyResponseCode.PollErrorResponsePending);
```

This will check the status of the launch request once per second until it leaves the Pending state. Once the status is no longer Pending, the launch request has either failed or succeeded. To determine the fate of a launch request, we can use the `IsAuthenticated()` helper method on `LaunchKeyRestClient`, as seen here:

```
bool success = client.IsAuthorized(pollResponse);
```

If `success` is `true`, then the user has approved the launch request, and you may proceed authenticating the user within your application. The `PollResponse` object will include a property `UserHash` which you can use to uniquely identify that LaunchKey user within your application. Additionally, the `IsAuthorized()` method will automatically call `Logs()`, informing the user that their session is active.

If `success` is `false`, then either the device rejected the request or an error occurred. In either case, the `PollResponse.MessageCode` property will indicate why the request failed. Additionally, the `IsAuthorized()` method will automatically call `Logs()`, informing LaunchKey that you did not offer the user a session.

Logout a User

If a user logs out of your application, it is important to notify LaunchKey. Notifying LaunchKey allows us to remove the application session associated with your application from the user's orbit list. The following API call will achieve this:

```
client.Logout(authRequest);
```

This notifies LaunchKey that the application session associated with the authentication request identified by **`authRequest`** has been terminated on the application side and can be removed from the user's active list of sessions.

API Reference

See LaunchKey-API.chm.

Example Applications

ASP.NET WebForms Example

Overview

In this example we will not use the default ASP.NET membership providers, but will implement our own custom forms authentication cycle, making use of a few types from the `System.Web.Security` namespace: `FormsAuthentication`, `FormsAuthenticationTicket`, `IIdentity` and `GenericPrincipal`.

Configuration

In the example application we store all of our LaunchKey configuration data in the root Web.config file:

```
<appSettings>
    <add key="lk_appKey" value="1234567890"/>
    <add key="lk_appSecret" value="my_secret_key"/>
    <add key="lk_appPrivateKey" value="my_private_rsa_key" />
</appSettings>
```

It is important that the lk_appSecret and lk_appPrivateKey values be kept private at all times.

Login Page

The login page (`Account\Login.aspx`) in this example is very basic and only contains one control: `LK:LoginControl`. The purpose of this control is to take in a user name, attempt to authenticate it, and poll until a response is received. `LK:LoginControl` functions mostly client-side, communicating with an ASMX web service (`LaunchKeyJsonWebService.asmx`) until a successful outcome is achieved. The web service in question contains two methods: `Login` and `LoginPoll`. These two methods serve to mediate communication between the `LK:LoginControl` and the LaunchKey API. The web service serves no other purpose.

If the response received is successful, the page redirects to a separate page (`Account\LoginConfirm.aspx`) which asks the user to identify themselves within our application, and continues with the authentication process.

When the user submits their name, the following event handler is invoked:

```
protected void SubmitButton_Click(object sender, EventArgs e)
{
          if (IsValid)
          {
                    var authRequest = AuthRequest.Value;
                    var lkClient = LaunchKeyClientFactory.GetInstanceFromConfig();
                    var pollResponse = lkClient.Poll(authRequest);

                    if (lkClient.IsAuthorized(authRequest, pollResponse))
                    {
                              this.SetAuthCookie(pollResponse.UserHash, authRequest, FriendlyName.Text);
                              Response.Redirect("~");
                    }
          }
}
```

Several things are happening here. First, we pull out the authentication request value stored in a hidden field. This is the identifier given to us by LaunchKey after the user approved the launch request. We then make a request to the LaunchKey API to confirm that the request is actually a valid request, and to verify that it is still valid. If it is, we now have a confirmed authentication request and a friendly name to refer to our visitor.

The identifying information is passed to a helper method, `SetAuthCookie()`, which does the dirty work of packing all of this information into a `FormsAuthenticationTicket`, encrypting it, and delivering it to the client in the form of a cookie. This cookie will serve as the identifying marker for the user for the remainder of their session. The following code snippet shows the `SetAuthCookie()` function.

```
private void SetAuthCookie(string userHash, string authRequest, string friendlyName)
{
          FormsAuthenticationTicket ticket = new FormsAuthenticationTicket(
                    2,
                    friendlyName,
                    DateTime.Now,
                    DateTime.Now.AddMinutes(15),
                    false,
                    string.Format("{0};;{1}", userHash, authRequest)
          );
          string encryptedTicket = FormsAuthentication.Encrypt(ticket);
          HttpCookie cookie = new HttpCookie(FormsAuthentication.FormsCookieName, encryptedTicket);
          cookie.Expires = DateTime.Now.AddMinutes(15);
          Response.Cookies.Add(cookie);
}
```

The above code creates a `FormsAuthenticationTicket` with the name of the user, their unique LaunchKey user hash, and the unique ID for this particular authentication session. The ticket is encrypted, stored in a cookie and set to expire within 15 minutes. The cookie is attached to the HTTP response using the cookie name supplied by `FormsAuthentication`, and delivered to the browser.

Changes to Global.asax

Since we are implementing a custom Forms Authentication process, there are additional steps that must be taken. We have supplied additional information to the `FormsAuthenticationTicket` which are relevant to the LaunchKey authentication process, and we're going to want to make those parameters available on subsequent requests. To do this, we create a method in `Global.asax`: `FormsAuthentcation_OnAuthenticate`. This will be automatically called by ASP.NET on each request. This is where we will retrieve the information we placed in the cookie.

We do this by implementing our own `IIdentity` type (`LaunchKeyIdentity`), and putting that in a `GenericPrincipal` object. Once we have constructed a `LaunchKeyIdentity` by decrypting the incoming `FormsAuthenticationTicket` cookie, we make a call to the LaunchKey API to determine if this particular user has not de-orbited our session.

If the session is no longer valid, we kill the cookie, inform LaunchKey that we have complied with the deorbit, and redirect the user to the home page.

If the session is still viable, we place the `LaunchKeyIdentity` in a `GenericPrincipal` and assign this to the `FormsAuthenticationEventArgs.User` property, which will make our custom Identity available throughout the application on this request. The code for this is below.

```
public void FormsAuthentication_OnAuthenticate(object sender, FormsAuthenticationEventArgs args)
{
          HttpCookie authCookie = Context.Request.Cookies[FormsAuthentication.FormsCookieName];
          if (authCookie != null)
          {
                    FormsAuthenticationTicket ticket = FormsAuthentication.Decrypt(authCookie.Value);
                    string[] userDataTokens = ticket.UserData.Split(new string[] { ";;" }, StringSplitOptions.None);

                    LaunchKeyIdentity identity = new LaunchKeyIdentity(userDataTokens[0], userDataTokens[1],
ticket.Name);

                    // verify user hasn't de-orbited
                    var lkClient = LaunchKeyClientFactory.GetInstanceFromConfig();
                    var pollResponse = lkClient.Poll(identity.AuthRequest);

                    if (pollResponse.Successful)
                    {
                              args.User = new GenericPrincipal(identity, null);
                    }
                    else
                    {
                              // unset cookie
```

```
                                FormsAuthentication.SignOut();
                                lkClient.Logs(SDK.Rest.LogsAction.Revoke, SDK.Rest.LogsStatus.Granted,
identity.AuthRequest);
                                Context.Response.Redirect("~");
                    }
          }
}
```

Conclusion

This example application is very basic but demonstrates how one might incorporate LaunchKey into an ASP.NET WebForms application. A more complete example would have better error handling, some way to persist the user accounts in a database, and would be smarter about how often to check the status of the LaunchKey session.

This is but one of several possible approaches. You might want to implement LaunchKey as one factor in a multifactor authentication scheme, use LaunchKey's OpenAuth endpoint, or just implement this in an overall different way.

Related Reading

It is worth investing some time in related reading:

1) http://msdn.microsoft.com/en-us/library/9wff0kyh(v=vs.100).aspx – Details the Forms Authentication Provider (MSDN)
2) http://msdn.microsoft.com/en-us/library/330a99hc(v=vs.100).aspx – ASP.NET Web Application Security (MSDN)
3) http://www.codeproject.com/Articles/98950/ASP-NET-authentication-and-authorization -- ASP.NET Authentication and Authorization (Shivprasad Koirala via CodeProject)

ASP.NET MVC 4 Example

Overview

In this example, similar to the ASP.NET WebForms example, we will not use the default ASP.NET membership providers, but will implement our own custom forms authentication cycle, making use of a few types from the `System.Web.Security` namespace: `FormsAuthentication`, `FormsAuthenticationTicket`, `IIdentity` and `GenericPrincipal`.

It is worth noting that this example does not use WebMatrix [TODO: Link], either. It is conceivable that one might want to use WebMatrix for base authentication using a password based security system, and bolt on LaunchKey as a secondary factor in a multi-factor authentication scheme.

In this example we will use a SQL Server Local Database to store users, something we didn't do in the ASP.NET WebForms example. This demonstrates a basic and imperfect way of tracking a LaunchKey user across multiple log in attempts.

Configuration

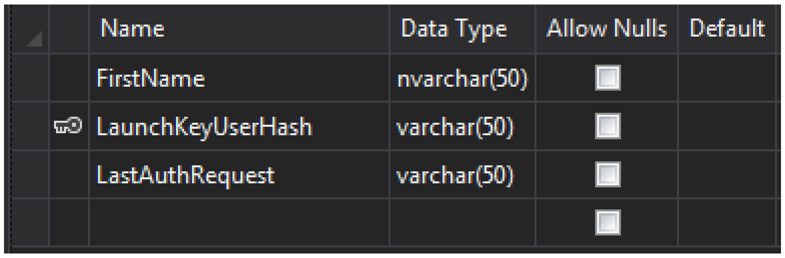
In the example application we store all of our LaunchKey configuration data in the root Web.config file:

```
<appSettings>
    <add key="lk_appKey" value="1234567890"/>
    <add key="lk_appSecret" value="my_secret_key"/>
    <add key="lk_appPrivateKey" value="my_private_rsa_key" />
</appSettings>
```

It is important that the lk_appSecret and lk_appPrivateKey values be kept private at all times.

Database

Included with the example is a very basic SQL Server Local Database file. The database has one table `User`, which is used to keep track of which LaunchKey users we've seen before, using the unique LaunchKey `UserHash` value as the primary key. The schema:



	Name	Data Type	Allow Nulls	Default
	FirstName	nvarchar(50)	☐	
🔑	LaunchKeyUserHash	varchar(50)	☐	
	LastAuthRequest	varchar(50)	☐	
			☐	

A more complete application would likely have several more columns for storing user data.

Account Controller

Our `Account` controller is pretty basic in this example. We don't implement some common actions you might be used to seeing in an `Account` controller: there is no password, so there are no password management actions; we do not register users explicitly, so there is no traditional `Register` action, and so on.

Authentication Flow

The basic login attempt works like this:

1) The user arrives at the `Login` action and is served the `Login.cshtm`l view.
2) The user enters their LaunchKey username. This is sent to our `LoginJson` action on the Accounts controller. This action sends a Launch Request to the LaunchKey API and returns that information to the browser as JSON.
3) If the launch request is successfully sent, the client side JS enters a polling loop where it will periodically make a request to `LoginPollJson` to check the status of the request.
4) If the launch request becomes approved, we redirect the user's browser to the `LoginConfirm` action.
5) If the user is new, they are served the `LoginNewUser.cshtml` view where they will provide a name for themselves within our test application. Once they successfully submit this form, they are logged in using our `SetAuthCookie()` function.
6) If the user is not new, then we already know their name and are logged in using our `SetAuthCookie()` function.

Account Actions

The following actions are implemented on our `Account` controller:

- `Login` – This is the action you're used to seeing. The user is redirected here any time they try to access a secured part of the site, or if they specifically click the Log In link.
- `LoginConfirm` – This is the second step of authentication. The user is redirected here when their LaunchKey launch request is successful. This action checks if they are an existing user. If they are, their session is created. If not, they are redirected to `LoginNewUser`
- `LoginNewUser` – This action is used when a user has successfully logged in using LaunchKey, but has never been on our site before. We collect their name, store it in the database, and complete their authentication.
- `LogOut` – This action is used when the user wishes to log out. We terminate their session and notify LaunchKey of the change.
- `LoginJson, LoginPollJson` – These two actions are used by our client-side JavaScript to initiate a LaunchKey login request, and return JSON

Account Views

- `Login.cshtml` – contains an input field for the LaunchKey username, as well as some JavaScript to make calls that reach back to the `Account` controller to perform LaunchKey API tasks.
- `LoginNewUser.cshtml` – Contains a form where a user provides information about themselves on their first visit to our site.

Authentication Flow in Detail

Our authentication flow begins in the same place it does for most ASP.NET MVC applications: on the Account controller in the Login action. When the user arrives here, we serve them the `Login.cshtml` view. Once the user enters their name, the form is submitted asynchronously via JSON to our `LoginJson` action.

The `LoginJson` action initializes our LaunchKey client, and sends the launch request to LaunchKey, returning this data to the JavaScript directly:

```
[AllowAnonymous]
[NoCache]
public ActionResult LoginJson(string username)
{
          var lkClient = LaunchKeyClientFactory.GetInstanceFromConfig();
          var authsResponse = lkClient.Authenticate(username, AuthenticationType.Session);
          return Json(authsResponse, JsonRequestBehavior.AllowGet);
}
```

Once the client-side JavaScript receives this data it will either error out, indicating to the user that their launch request failed for some reason, or enter a poll state where it will periodically call our `LoginPollJson` action to see if the launch request has been approved.

Our `LoginPollJson` action is a bit more complicated than our `LoginJson` action, and returns different data to the client-side code based on the current status of the `Poll` call. A successful `Poll` attempt might contain a great deal of secretive data, and so we do not return this data directly to the browser. If we did, this could create a significant security vulnerability.

Instead, we process the data and provide the client-side code with only the information it needs: Is the launch request approved or not and were there any errors?

```
[AllowAnonymous]
[NoCache]
public ActionResult LoginPollJson(string authRequest)
{
          var lkClient = LaunchKeyClientFactory.GetInstanceFromConfig();
          var pollResponse = lkClient.Poll(authRequest);

          // request failed for some reason, let client error out
          if (!pollResponse.Successful)
          {
                    return Json(new { Successful = false, Waiting = false, ErrorCode = pollResponse.MessageCode,
ErrorMessage = pollResponse.Message }, JsonRequestBehavior.AllowGet);
          }

          // request succeeded but still waiting
          if (pollResponse.UserHash == null)
          {
                    return Json(new { Successful = true, Waiting = true }, JsonRequestBehavior.AllowGet);
          }

          // request succeeded, device responded with an OK
          if (pollResponse.DecryptedAuth.Response)
          {
                    return Json(new { Successful = true, Waiting = false, Accepted = true, RedirectUrl =
Url.Action("LoginConfirm", new { authRequest = authRequest }) }, JsonRequestBehavior.AllowGet);
          }
          // request succeeded, device rejected
          else
          {
                    lkClient.Logs(LogsAction.Authenticate, LogsStatus.Denied, authRequest);
                    return Json(new { Successful = true, Waiting = false, Accepted = false },
JsonRequestBehavior.AllowGet);
          }

}
```

Once the request succeeds, we redirect the user to our LoginConfirm action, passing along the unique `authRequest` identifier provided to us by LaunchKey. Several things happen here.

First, we verify the `authRequest` being passed to us is a valid session by calling Poll once more.

If we have a valid login, we check the `UserHash` value on our `PollResponse` object to see if we know this user. The `UserHash` value is a unique value given to our application by the LaunchKey API, and is a permanently unique way to identify this user within the context of our application.

If the user is new, we serve the `LoginNewUser.cshtml` view, collecting information to create their user in the database.

If the user is existing, we pull their information from our database.

Once the user has been verified as existing or has entered registration information as a new user, we do the dirty work of logging them in. This is achieved using .NET's built-in `FormsAuthentication` class. The basic idea is that we provide the user's browser with an encrypted cookie, and that cookie will be passed back to us on subsequent requests to our web application. The encrypted cookie is created by constructing a `FormsAuthenticationTicket` object then securely serializing that data using the `FormsAuthentication.Encrypt()` helper method. This is all carried out in the `SetAuthCookie()` function on our Account controller:

```
private void SetAuthCookie(User user)
{
          FormsAuthenticationTicket ticket = new FormsAuthenticationTicket(
                    2,
                    user.LaunchKeyUserHash,
                    DateTime.Now,
                    DateTime.Now.AddMinutes(15),
                    false,
                    string.Format("{0};;{1}", user.LastAuthRequest, user.FirstName)
          );
          string encryptedTicket = FormsAuthentication.Encrypt(ticket);
          HttpCookie cookie = new HttpCookie(FormsAuthentication.FormsCookieName, encryptedTicket);
          cookie.Expires = DateTime.Now.AddMinutes(15);
          Response.Cookies.Add(cookie);
}
```

Once this cookie has been sent to our user, they are considered logged in.

Changes to Global.asax

Since we are implementing a custom Forms Authentication process, there are additional steps that must be taken. We have supplied additional information to the `FormsAuthenticationTicket` which are relevant to the LaunchKey authentication process, and we're going to want to make those parameters available on subsequent requests. To do this, we create a method in `Global.asax`: `FormsAuthentcation_OnAuthenticate`. This will be automatically called by ASP.NET on each request. This is where we will retrieve the information we placed in the cookie.

We do this by implementing our own `IIdentity` type (`LaunchKeyIdentity`), and putting that in a `GenericPrincipal` object. Once we have constructed a `LaunchKeyIdentity` by decrypting the incoming `FormsAuthenticationTicket` cookie, we make a call to the LaunchKey API to determine if this particular user has not de-orbited our session.

If the session is no longer valid, we kill the cookie, inform LaunchKey that we have complied with the deorbit, and redirect the user to the home page.

If the session is still viable, we place the `LaunchKeyIdentity` in a `GenericPrincipal` and assign this to the `FormsAuthenticationEventArgs.User` property, which will make our custom Identity available throughout the application on this request. The code for this is below.

```
public void FormsAuthentication_OnAuthenticate(object sender, FormsAuthenticationEventArgs args)
{
          HttpCookie authCookie = Context.Request.Cookies[FormsAuthentication.FormsCookieName];
          if (authCookie != null)
          {
                    FormsAuthenticationTicket ticket = FormsAuthentication.Decrypt(authCookie.Value);
                    string[] userDataTokens = ticket.UserData.Split(new string[] { ";;" },
StringSplitOptions.RemoveEmptyEntries);
                    LaunchKeyIdentity identity = new LaunchKeyIdentity(ticket.Name, userDataTokens[0],
userDataTokens[1]);
                    // verify user hasn't de-orbited
                    var lkClient = LaunchKeyClientFactory.GetInstanceFromConfig();
                    var pollResponse = lkClient.Poll(identity.AuthRequest);
                    if (pollResponse.Successful)
                    {
                              args.User = new GenericPrincipal(identity, null);
                    }
                    else
                    {
                              // unset cookie
                              FormsAuthentication.SignOut();
                              lkClient.Logs(SDK.Rest.LogsAction.Revoke, SDK.Rest.LogsStatus.Granted,
identity.AuthRequest);
                              Context.Response.Redirect("~");
                    }
          }
}
```

Conclusion

This example application is very basic but demonstrates how one might incorporate LaunchKey into an ASP.NET MVC 4 application. A more complete example would have better error handling, more secure areas of the website, and probably some type of Roles implementation.

This is but one of several possible approaches. You might want to implement LaunchKey as one factor in a multifactor authentication scheme, use LaunchKey's OpenAuth endpoint, or just implement this in an overall different way.

Related Reading

It is worth investing some time in related reading:

1) http://msdn.microsoft.com/en-us/library/9wff0kyh(v=vs.100).aspx – Details the Forms Authentication Provider (MSDN)
2) http://msdn.microsoft.com/en-us/library/330a99hc(v=vs.100).aspx – ASP.NET Web Application Security (MSDN)
3) http://www.codeproject.com/Articles/578374/AplusBeginner-27splusTutorialplusonplusCustomplusF -- A Beginner's Tutorial on Custom Forms Authentication in ASP.NET MVC Application (Rahul Rajat Singh via CodeProject)

Other Examples

Included in the SDK package are two other example applications:

- **LaunchKey.Examples.ConsoleApplication** – extremely basic API usage. Allows you to send launch requests and de-orbit from the command line.
- **LaunchKey.Examples.WinFormsApplication** – a simple WinForms application which requires launch key authentication to show you a picture.