<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

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

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<title>API Authentication and Security on the Community Platform</title>

<link href="../../../../resources/style/reset.css" rel="stylesheet" type="text/css" />

<link href="../../../../resources/style/base.css" rel="stylesheet" type="text/css" />

<link href="../../style/document.css" rel="stylesheet" type="text/css" />

</head>

<body>

<div class="documentation supportContent">

<!-----

layout: page

title: App Security

description: Learn about how to set up and manage app security.

product: cm

category: learn

subcategory: App Security

weight: 20

type: page

nav-title: App Security

----->

## <h2 id="top">App Security</h2>

<p>It's important to set up authentication for the messages your app sends to an API you&rsquo;re using. Authentication protects the sender, the receiver, and the message itself. It&rsquo;s important that the security is strong enough that a third party couldn&rsquo;t pose as the sender, pose as the receiver, and/or view or change the message content.</p>

<p>Authentication and security are particularly important when money and sensitive personally identifiable information are involved.</p>

<p>Once your app is registered, it's important to set up authentication so that the app can use the API to process messages securely.</p>

<p>This document provides information about the various security options available to you and the steps that you'll need to take to set up the option you choose.</p>

<h5>What do you want to read about?</h5>

<ul>

<li><a href="#appsecurityoverview">App Security Overview</a></li>

<li><a href="#token\_construction">Token Construction Approaches and Procedures</a></li>

<li><a href="#ref\_notes\_and\_procedures">Reference Notes and Procedures</a></li>

<li><a href="#errors\_all">Error Codes and Descriptions</a></li>

</ul>

<p class="apidoc\_note"><strong>Note</strong>: This topic provides information about setting up your app to authenticate with for any API hosted on the platform. For implementation information and examples for a specific API you're using, refer to the documentation for that API.</p>

<hr />

### <a name="appsecurityoverview"></a><h3>App Security Overview</h3>

<p>xxx <span class="apidoc\_red">general broad overview</span> here of what the security is doing in all cases. very general. </p>

<p>This section includes:</p>

<ul>

<li><a href="#security\_mechanisms\_available">Security Mechanisms Available</a></li>

<li><a href="#including\_token">Including Token in Request Message Structure</a></li>

<li><a href="#token\_structure">Token Structure</a></li>

<li><a href="#authentication\_realm">Authentication Realm</a></li>

</ul>

#### <a name="security\_mechanisms\_available"></a><h4>Security Mechanisms Available</h4>

<p>In the platform, as part of setting up your app, you&rsquo;ll need to choose the security mechanism you&rsquo;ll use for authentication of your messages. You can choose between these options:</p>

<ul>

<li>HMAC Signature with Shared Secret</li>

<li>HASH/Digest with Shared Secret</li>

<li>RSA Signature with PKI</li>

</ul>

<p>The first two use a Shared Secret value generated by the platform for your app, but use different algorithms for constructing the signature; the last mechanism uses RSA security with public/private key.</p>

<p>More information about each method, and instructions for constructing each, is given later in this document.</p>

#### <a name="including\_token"></a><h4>Including Token in Request Message Structure</h4>

<p>There are three ways you can include the token in the request message:</p>

<ul>

<li>In the authorization headers</li>

<li>In a cookie</li>

<li>As query/POST parameters</li>

</ul>

<p>{{<span class="apidoc\_red">What else do we need to say about this?</span>}}</p>

#### <a name="token\_structure"></a><h4>Token Structure</h4>

<p>In order for the platform to recognize your app, you must construct a token and send it with the request message.</p>

<p>The values that make up the token, and the way it's constructed, vary according to the security mechanism you're using. The table below lists all token values and indicates the security mechanisms for which each is valid. The specific values and their sequence depends on the specific security mechanism you choose; the procedures for setting up each security mechanism include exact instructions for constructing the token for that mechanism.</p>

<p class="apidoc\_note"><strong>Note</strong>: Each value includes a prefix, the value of which is determined by the Site Admin. In the table below, and everywhere in this document, we use a fictitious company name, <strong>acmecorp</strong>, for the prefix; for example, <strong>acmecorp\_app\_id</strong>. In your installation it will be some other value; for example, it might be the Tenant ID. If you're not sure what value to use, consult your site administrator.</p>

<table width="98%" cellpadding="2" cellspacing="0" class="tableparmsresponse" summary="table description">

<thead>

<tr>

<th width="25%">Name</th>

<th width="20%">Used By</th>

<th width="55%">Description</th>

</tr>

</thead>

<tbody>

<tr>

<td><a name="acmecorp\_app\_id"></a>acmecorp\_app\_id</td>

<td>All</td>

<td>The AppID assigned to the app in the platform.</td>

</tr>

<tr>

<td><a name="acmecorp\_nonce"></a>acmecorp\_nonce</td>

<td>All</td>

<td>A random string, uniquely generated for each request. The nonce allows the API providers to verify that a request has never been made before and helps prevent replay attacks when requests are made over a non-secure channel. For more information, see <a href="#generatingnonce">Generating a Nonce Value</a>.</td>

</tr>

<tr>

<td><a name="acmecorp\_secret\_digest"></a>acmecorp\_secret\_digest</td>

<td>Shared Secret {<span class="apidoc\_red">is it valid for HMAC?}</span></td>

<td>A value produced by concatenating the nonce, the timestamp, and the Shared Secret, hashing the combined value using SHA-1, and then Base-64 encoding the result. You can also URL-encode the result, but this isn't required.</td>

</tr>

<tr>

<td><a name="acmecorp\_signature\_method"></a>acmecorp\_signature\_method</td>

<td>All</td>

<td><p>A value indicating the signature method. Valid values:</p>

<ul class="apidoctable\_ul">

<li>For HMAC Signature with Shared Secret: <span class="apidoc\_red">xxx</span>.</li>

<li>For Hash/Digest with Shared Secret: <strong>Digest</strong>.</li>

<li>for RSA Signature with PKI: <strong>SHA1withRSA</strong> or <strong>NONE</strong>. The <strong>NONE</strong> signature method is used for APIs that don't require security. In this case, only the acmecorp\_app\_id is required in the HTTP Authorization headers.</li>

</ul></td>

</tr>

<tr>

<td><a name="acmecorp\_signature"></a>acmecorp\_signature</td>

<td>RSA Signature with PKI</td>

<td>The signature value. For information on how the signature value is calculated, see <a href="#composing">Composing the Signature</a>.</td>

</tr>

<tr>

<td><a name="acmecorp\_timestamp"></a>acmecorp\_timestamp</td>

<td>All</td>

<td>The timestamp of the request, expressed as the number of milliseconds since January 1, 1970 00:00:00 GMT. The timestamp must be a positive integer and must be greater than or equal to the timestamp used in previous requests.</td>

</tr>

<tr>

<td><a name="acmecorp\_version"></a>acmecorp\_version</td>

<td>All</td>

<td>The platform version. Optional. If present, the value must be 1.0.</td>

</tr>

</tbody>

</table>

#### <a name="authentication\_realm"></a><h4>Authentication Realm</h4>

<p>Every authentication mechanism offered by the platform includes a specific authentication parameter, <strong>realm</strong>, that identifies the fact that the message comes from the realm for your app.</p>

<p>In the platform, you can specify the value to be used for the realm parameter, but there are restrictions. You have two choices:</p>

<ul>

<li>The platform automatically defaults to the first part of the incoming request hostname.</li>

<li>You can change it, but the value you choose must be a valid part of the incoming request hostname.</li>

</ul>

<p>Let's say, for example, the hostname for your API requests is acmecorp.payments.com. The platform defaults to acmecorp. You can change it to payments, but you cannot choose acmecorporation. You must choose a valid part of the hostname.</p>

<p>When constructing the signature, do not include the realm in the <a href="#constructingsbs">signature base string</a>.</p>

<p>For more information about use of the realm parameter in authentication, refer to the <a href="http://tools.ietf.org/html/rfc2617#section-1.2" target="\_blank">Access Authentication Framework</a> section of the HTTP Authentication standard for Basic and Digest Access Authentication.</p>

<p><a href="#top">Back to top</a></p>

<hr />

### <a name="token\_construction"></a><h3>Token Construction Approaches and Procedures</h3>

<p>This section gives you the detailed information you need to choose which security mechanism you'll use and to implement it. It includes:</p>

<ul>

<li><a href="#whichtochoose">Which Security Mechanism Should I Choose?</a></li>

<li><a href="#authenticating\_hmac\_ss">HMAC Signature with Shared Secret</a></li>

<li><a href="#authenticating\_hash\_ss">HASH/Digest with Shared Secret</a></li>

<li><a href="#authenticating\_rsa\_pki">RSA Signature with PKI</a></li>

</ul>

#### <a name="whichtochoose"></a><h4>(A) Which Security Mechanism Should I Choose?</h4>

<p>Below are some points about the various security mechanisms to help you determine which approach is best for your app.</p>

<ul>

<li>All approaches, Shared Secret and RSA, Signature with PKI, are very secure. All use a secret key combined with hashing and encoding, and are sent via HTTPS.</li>

<li>The Shared Secret approaches are easier to implement since you don't have to generate public and private keys.</li>

<li>The PKI approach is a little more secure, although both approaches are very secure. The PKI approach is more complex to implement.</li>

<li>HMAC Signature with Shared Secret is compatible with the OAuth 1.0 standard. However, this standard is now deprecated (see <a href="http://oauth.net/core/1.0/" target="\_blank">OAuth Core 1.0 specification</a>).</li>

</ul>

<p class="apidoc\_note"><strong>Note</strong>: The platform also supports OAuth 1.0a and OAuth 2.0.</p>

<p><a href="#top">Back to top</a></p>

<hr />

#### <a name="authenticating\_hmac\_ss"></a><h4>HMAC Signature with Shared Secret</h4>

<p>{{<span class="apidoc\_red">We need to fill in this section with the below info</span>:}}</p>

<p>When you use the HMAC Signature with Shared Secret option, you do not need to send the Shared Secret. You just use it to create the signature.</p>

<p>This approach essentially follows the mechanism laid out in the OAuth 1.0 specification (deprecated; see <a href="http://oauth.net/core/1.0/" target="\_blank">OAuth Core 1.0 specification</a>).</p>

<p>This section includes:</p>

<ul>

<!-- <li>Authenticating with HMAC signature: Process Flow</li>-->

<li><a href="#headerexample\_hmac\_ss">HMAC Signature with Shared Secret: Header Example</a></li>

<li><a href="#signing\_hmac\_ss">Signing Requests with the Shared Secret</a></li>

<li><a href="#composingsignature\_hmac\_ss">Composing the Signature</a></li>

<li><a href="#oauth10\_hmac\_ss">Compatibility with OAuth 1.0 (a deprecated standard)</a></li>

</ul>

<h5><a name="headerexample\_hmac\_ss"></a>HMAC Signature with Shared Secret: Header Example</h5>

<p>xxx</p>

<a name="signing\_hmac\_ss"></a><h5>Signing Requests with the Shared Secret</h5>

<p>The Shared Secret is a binary hashed value, generated within the secure environment of the platform and known only to you and to the platform.For more information, see <a href="#def\_sharedsecret">What Is the Shared Secret?</a></p>

<p>With the HMAC Signature with Shared Secret security mechanism, you do the following:</p>

<ol>

<li>Combine the Shared Secret with two other values.</li>

<li>Hash the result using the SHA-1 secure hashing algorithm.</li>

<li>Encode the hashed value using the Base64 encoding scheme.</li>

<li>Optional: You can also URL-encode the result, but this isn't required.</li>

<li>Assign the result as the value for the acmecorp\_secret\_digest parameter.</li>

<li>Send it in the HTTP Authorization message header.</li>

</ol>

<a name="composingsignature\_hmac\_ss"></a><h5>HMAC Signature with Shared Secret: Composing the Signature</h5>

<p>xxx</p>

<a name="oauth10\_hmac\_ss"></a><h5>Compatibility with OAuth 1.0 (a deprecated standard)</h5>

<p>xxx</p>

<a name="authenticating\_hash\_ss"></a>

#### <h4>Hash/Digest with Shared Secret</h4>

<p>The Shared Secret is a binary hashed value, generated within the secure environment of the platform and known only to you and to the platform.For more information, see <a href="#def\_sharedsecret">What Is the Shared Secret?</a></p>

<p>With the Hash/Digest with Shared Secret security mechanism, you do the following:</p>

<ol>

<li>Combine the Shared Secret with two other values.</li>

<li>Hash the result using the SHA-1 secure hashing algorithm.</li>

<li>Encode the hashed value using the Base64 encoding scheme.</li>

<li>Optional: You can also URL-encode the result, but this isn't required.</li>

<li>Assign the result as the value for the acmecorp\_secret\_digest parameter.</li>

<li>Include this parameter in the HTTP Authorization message header.</li>

</ol>

<p>The API verifies each incoming request using the Shared Secret.</p>

<p>If you choose the Shared Secret security mechanism, your HTTP Authorization header must include the values shown below. For definitions of each value, refer to <a href="#tokenstructure">Token Structure</a> above.</p>

<ul>

<li><a href="#acmecorp\_app\_id">acmecorp\_app\_id</a></li>

<li><a href="#acmecorp\_nonce">acmecorp\_nonce</a></li>

<li><a href="#acmecorp\_secret\_digest">acmecorp\_secret\_digest</a></li>

<li><a href="#acmecorp\_signature\_method">acmecorp\_signature\_method</a></li>

<li><a href="#acmecorp\_timestamp">acmecorp\_timestamp</a></li>

<li><a href="#acmecorp\_version">acmecorp\_version</a></li>

</ul>

<p>More information about the shared secret approach is given in these sections:</p>

<ul>

<li><a href="#processflow\_ss">Authenticating with the Shared Secret: Process Flow</a></li>

<li><a href="#sampleheader">Sample Header With Authorization Parameters (Shared Secret Approach)</a></li>

<li><a href="#generatingnonce">Generating a Nonce Value</a></li>

<li><a href="#generatingtimestamp">Generating the Timestamp</a></li>

<li><a href="#hashing\_sha1">Hashing With the SHA-1 Secure Hash Algorithm</a></li>

<li><a href="#encodingbasestring">Encoding the base string into bytes</a></li>

<li><a href="#troubleshooting\_ss">Troubleshooting (Shared Secret Approach)</a></li>

<li><a href="#compromise\_ss">What If the Shared Secret Is Compromised?</a></li>

<li><a href="#compromise\_ss">Java Example: Creating a Shared Secret Hash</a></li>

</ul>

<h5><a name="processflow\_ss"></a>(C) Authenticating with the Shared Secret: Process Flow</h5>

<p> The process flow diagram below shows an overview of API request authentication with the shared secret security mechanism.</p>

<p class="apidoc\_image\_p"><img src="images/processflow\_apirequestauth\_ss.jpg" alt="Process flow: shared secret security mechanism" /></p>

<h5><a name="sampleheader"></a>Sample Header With Authorization Parameters (Shared Secret Approach)</h5>

<p>Below is an example of an Authorization header with app authentication parameters, when you use the shared secret approach. The sequence of the parameters isn't significant, but all parameters must be present with valid values.</p>

<p class="apidoc\_note"><strong>Note</strong>: The value for the realm is determined by the www-Authenticate header. Make sure the value you set in the Authorization header is correct for your installation. For more information about the <strong>realm</strong> value, see <a href="#authentication\_realm">Authentication Realm</a>.</p>

<pre>Authorization: Atmosphere realm="http://acmecorp",

acmecorp\_app\_id="development-AS0iTmhoGaE6Y9sWhUkvcL6T",

acmecorp\_nonce="1326409129918",

acmecorp\_secret\_digest="wOJIO9A2W5mFwDgiDvZbTSMK%2FPY%3D",

acmecorp\_digest\_method="SHA1",

acmecorp\_timestamp="1326755565940",

acmecorp\_version="1.0"

</pre>

<a name="hashing\_sha1"></a>

<h5>Hashing With the SHA-1 Secure Hash Algorithm</h5>

<p>SHA-1 is a cryptographic hash function, broadly used and trusted.</p>

<p>When you hash a value with SHA-1, the hash function returns a 160-bit string. This is the message digest. The value is hashed and sent with the message; at the receipt point, the value is hashed again, and the two hash values are compared. When the two hash values match, it is a secure, reliable indication that the message hasn't changed; the message at the receipt point is an accurate duplication of the message at the send point.</p>

<p>If you're using the Shared Secret security mechanism, you concatenate the nonce, the timestamp, and the Shared Secret, and then you hash that combined value using SHA-1. You then encode the result, and send it in the message header as the value for the acmecorp\_secret\_digest parameter.</p>

<p>The formula for computing the hash is:</p>

<pre>acmecorp\_secret\_digest = Base64 ( SHA-1 ( nonce + created + AppSecret ) ) </pre>

<p>Below is an example of a hashed base string:</p>

<pre>

1326409129918+1326755565940+2d9d42b42a4e2abc1fa5489d5081e03b95818ffd </pre>

<h5><a name="encodingbasestring"></a>Encoding the base string into bytes</h5>

<p>When you concatenate the nonce, timestamp, and app secret into a base string to be hashed, you'll need to make sure the concatenated value is a byte value before hashing. The hashing algorithm requires the input parameter to be in bytes. Depending on your app, you might need to use encoding to get a byte value. If so, use UTF-8 encoding.</p>

<p>Below is an example using Java.</p>

<pre>

MessageDigest md = MessageDigest.getInstance("SHA1");

md.update(baseString.getBytes("UTF-8")); or

md.update(baseString.getBytes());

String digest = encodeAsBase64String(md.digest());</pre>

<p class="apidoc\_note"><strong>Note</strong>: The example above is part of a larger Java sample for a specific API. For more information, refer to the implementation examples for the API you're using.</p>

<a name="createsshash\_java"></a>

<h5>Java Example: Creating a Shared Secret Hash</h5>

<p>Below is an example of how to create the acmecorp\_secret\_digest value in Java.</p>

<pre> //Use the shared secret of app on file

String sharedSecret = "1008877afabf32efb31f9c974dbeaa688bed0769";

//Generate the nonce value

Import java.security.SecureRandom:

Random rand = SecureRandom.getInstance ("SHA1PRNG");

long nonce = rand.nextLong();

//Get the timestamp in milliseconds

long timestamp = System.currentTimeMillis();

// Concatenate the nonce, timestamp, and Shared Secret

String baseString = String.valueOf(nonce) + String.valueOf(timestamp) + sharedSecret;

// Hash the concatenated value into a message digest using SHA-1

MessageDigest md = MessageDigest.getInstance("SHA1");

md.update(baseString.getBytes());

// Encode the message digest using Base-64 encoding

String digest = encodeAsBase64String(md.digest());</pre>

<p><a href="#top">Back to top</a></p>

<hr />

#### <h4><a name="authenticating\_rsa\_pki"></a></h4>

#### <h4>RSA Signature with PKI</h4>

<p>When you register your app in the platform, you can set up app security in the <em>Security Credentials</em> section of your <em>App Details</em> page. </p>

<p>If you want to use Public Key Infrastructure (PKI) for secure message signing, click the button to choose Public Key Integration. You'll need to get a public/private key pair and upload the public key to the platform, as explained later in this document. In the custom app code you will sign the API requests using the private key. The API verifies each incoming request using the public key.</p>

<p>More information about authenticating with PKI is given in these sections:</p>

<ul>

<li><a href="#processflow\_pki">Authenticating with PKI: Process Flow</a></li>

<li><a href="#generatingkeypairwithopenssl">Generating a Public/Private Key Pair</a></li>

<li><a href="#java\_keystorejdk">Java Developers: Generating a Keystore and CSR with the JDK Keytool</a></li>

<li><a href="#signing\_pki">Signing Requests with Private Key</a></li>

<li><a href="#composing">Composing the Signature</a></li>

</ul>

<a name="processflow\_pki"></a><h5>(M) Authenticating with PKI: Process Flow</h5>

<p> The process flow diagram below shows an overview of API request authentication with PKI.</p>

<p class="apidoc\_image\_p"><img src="images/processflow\_apirequestauth\_pki.jpg" alt="Process flow: PKI security mechanism" /></p>

<a name="generatingkeypair"></a><h5>Generating a Public/Private Keypair</h5>

<p>The first step is to generate the public/private keypair for your app.</p>

<p>For a detailed example, refer to <a href="#generating\_keypair\_openssl">Generating PKI Key Pair with Certificate Using OpenSSL</a> later in this document.</p>

<h5>Sample header (RSA Signature with PKI)</h5>

<p>In the example below, the message header uses the PKI approach. The atmosphere-signature value is the signed value of the signature base string (see <a href="#constructingsbs">signature base string</a>).</p>

<pre>

Atmosphere realm="http://atmosphere",

atmosphere\_app\_id="Atmosphere-2f97rkSViLn6yd7syPtRiG7q&quot;,

atmosphere\_nonce=&quot;1328745832972&quot;,

atmosphere\_timestamp=&quot;1328745832972&quot;

atmosphere\_signature\_method="SHA1withRSA",

atmosphere\_signature="XKYtUsH7ta8I5mbZaZebm4b6NLjAVSGcK3op9lJxWC%2FlpNLxAm1GoHoMkKzLUh

%2BN7QdEBjR6ufgKMD%2BTWK4fPT2uXbHWNuOOTM4qUizrRymPxuAXEWhmECEB4WL7zr9UmWG7OGDsyuDUOt

93nAnVvfqinF0bC%2FnWBAH%2FelZlX%2FUMtO29%2BNrMq7ep0B%2BWKEosJmCFBtfgk72kRYl5rKsDqDm2m

gIFyHPWoFvC5WJiqtXTqGEaNxSbdWweaCaLqHNRnXowozOLMYYnnEd%2FTSFF0E5CrJBTzOq5Wvixnt6iJqWE

J0ywIc%2FPjVFs5muYOv30Lno44F0w6b3vjKTHWOJHEg%3D%3D",

atmosphere\_version="1.0"

</pre>

<a name="java\_keystorejdk"></a><h5>Java Developers: Generating a Keystore and CSR with the JDK Keytool</h5>

<p>If you are a Java developer and have worked with cryptography in Java, you might be familiar with the concept of a <strong><em>keystore</em></strong>. A keystore is a file used to store and deploy certificates and private keys in a single package. You can create a keystore using OpenSSL as explained above. However, as a Java developer, it is probably far more convenient to use <strong><em>keytool</em></strong>, a key and certificate management utility that's distributed as part of the Java Development Kit (JDK).</p>

<p>You can use keytool to create both the keystore and the CSR associated with the private key in the keystore. Once you've created your CSR, you upload it to the platform just like any other CSR. Your Java code reads your private key, used for signing the base string used in API request headers, from the keystore file you create.</p>

<p>Since the keytool utility is part of the JDK, you don't have to download anything to create your Java keystore. Just follow the steps below.</p>

<p class="apidoc\_note"><strong>Note</strong>: The procedure below assumes that your JDK <strong>bin</strong> directory is in your Windows system path.</p>

<h5>To create a Java keystore using the keytool utility</h5>

<ol class="apidoc\_ol">

<li>Open a command prompt.</li>

<li>Create the keystore file by typing the following:

<p><strong>keytool -genkey -alias &lt;keystorealias&gt; -keyalg RSA -keystore &lt;keystore\_filename&gt;.p12 -storetype PKCS12 -keypass &lt;password&gt; -keysize 2048</strong></p></li>

<li>Create the CSR associated with the keystore created in step 2 by typing the following:

<p><strong>keytool -certreq -alias &lt;keystorealias&gt; -keystore &lt;keystore\_filename&gt;.p12 -file &lt;csr\_filename&gt;.csr -storetype PKCS12</strong></p></li>

</ol>

<p>The illustration below shows the steps to create a keystore called <span class="apidoc\_code">keystore.p12</span> with an alias of <strong>mykeystore</strong> and the password <strong>changeme</strong>.</p>

<p class="apidoc\_image\_p"><img src="images/java\_keytool\_createkeystore.png" width="1040" alt="Creating a keystore with the Java keytool utility" /></p>

<p>The illustration below shows how to create a CSR called <strong>csr.csr</strong> from <strong>keystore.p12</strong>.</p>

<p class="apidoc\_image\_p"><img src="images/java\_keytool\_createcsr.png" width="811" height="106" alt="Creating a CSR with the Java keytool utility" /></p>

<p class="apidoc\_note"><strong>Note</strong>: For implementation examples, check the authentication documentation for the specific API you're using.</p>

<p class="apidoc\_note"><strong>IMPORTANT :</strong> Upload the file <strong>CSR.csr</strong> to the portal.</p>

<h5>Sample headers (PKI approach)</h5>

<p>This section gives some sample headers, including:</p>

<ul>

<li><a href="#sample\_pki\_authparms">Sample header with authorization parameters</a></li>

<li><a href="#sample\_pki\_noauthparms">Sample header with authorization parameters</a></li>

</ul>

<a name="sample\_pki\_authparms"></a><h5>Sample header with authorization parameters</h5>

<p>Below is an example of an Authorization header with app authentication parameters. Make sure you list the parameters in the sequence shown below.</p>

<p class="apidoc\_note"><strong>Note</strong>: The value for the realm is determined by the www-Authenticate header. Make sure the value you set in the Authorization header is correct for your installation. For more information about the <strong>realm</strong> value, see <a href="#authentication\_realm">Authentication Realm</a>.</p>

<pre>Authorization: realm="http://acmecorp",

acmecorp\_app\_id="acmecorp-7FSXeNRkVRJ8XtAurgaea65R",

acmecorp\_nonce="1323732744354",

acmecorp\_signature\_method="SHA1withRSA",

acmecorp\_signature="wOJIO9A2W5mFwDgiDvZbTSMK%2FPY%3D",

acmecorp\_timestamp="1323732744354",

acmecorp\_version="1.0"</pre>

<a name="sample\_pki\_noauthparms"></a><h5>Sample header without authorization parameters</h5>

<p> Below is an example of an Authorization header for an API that doesn't require authentication:</p>

<pre>

Authorization: realm=&quot;http://acmecorp&quot;,

acmecorp\_app\_id=&quot;http://www.open.com/app/101&quot;,

acmecorp\_signature\_method=&quot;NONE&quot;</pre>

<p class="apidoc\_note"><strong>Note</strong>: In case the client, such as an Ajax client in the browser or an Android app, cannot set the HTTP Authorization header, you can set the required authorization header parameters in the HTTP query string.</p>

<a name="signing\_pki"></a><h5>Signing Requests with Private Key</h5>

<p>The signature process for signing the request with private key follows a similar approach to the <a href="http://tools.ietf.org/html/rfc5849" target="\_blank">OAuth 1.0 standard</a>. The platform uses an HTTP Extension header to support its app authentication scheme. This Extension header is named Atmosphere (not case-sensitive). It uses the standard HTTP Authorization header and WWW-Authenticate response header to pass app-related authentication parameters. The HTTP query string can also be used to pass authentication data for cases where the app cannot directly manipulate HTTP headers.</p>

<h5>WWW-Authenticate header</h5>

<p> API providers can indicate their support for the extension by returning the HTTP WWW-Authenticate header in response to app requests for protected resources. Below is an example:</p>

<pre>

WWW-Authenticate: Atmosphere realm=&quot;http://acmecorp</pre>

<h5>Authorization header</h5>

<p>Apps should send their credentials in the HTTP Authorization header. If you choose the Public Key Integration approach for your app security, your HTTP Authorization header must include the values shown below.</p>

<p class="apidoc\_note"><strong>Important</strong>: The sequence of the header attributes is critical and must exactly match the order below.</p>

<ul>

<li><a href="#acmecorp\_app\_id">acmecorp\_app\_id</a></li>

<li><a href="#acmecorp\_nonce">acmecorp\_nonce</a></li>

<li><a href="#acmecorp\_signature\_method">acmecorp\_signature\_method</a></li>

<li><a href="#acmecorp\_signature">acmecorp\_signature</a></li>

<li><a href="#acmecorp\_timestamp">acmecorp\_timestamp</a></li>

<li><a href="#acmecorp\_version">acmecorp\_version</a></li>

</ul>

<a name="composing"></a><h5>(S) Composing the Signature</h5>

<p>There are several steps to composing the signature:</p>

<ul>

<li><a href="#constructingsbs">Creating the Signature Base String</a></li>

<li><a href="#signingandencoding">Signing the Signature Base String and Encoding the Signature</a></li>

</ul>

<a name="constructingsbs"></a><h5>Creating the Signature Base String</h5>

<p>The first step in composing the digital signature is to create the content. Once that's done, the data is signed using the signature algorithm.</p>

<p class="apidoc\_note"><strong>Note</strong>: The acmecorp\_signature parameter <em>must</em> be excluded from the Signature Base String.</p>

<p>Below is an example of the Signature Base String.</p>

<pre>

POST&amp;https://api.sandbox.yoursandbox.com/APIName/Payment/v1/MethodName

&amp;acmecorp\_app\_id=acmecorp-7FSXeNRkVRJ8XtAurgaea65R

&amp;acmecorp\_nonce=1323732744354&amp;acmecorp\_signature\_method=SHA1withRSA

&amp;acmecorp\_timestamp=1323732744354&amp;acmecorp\_version=1.0</pre>

<a name="signingandencoding"></a><h5>Signing the Signature Base String and Encoding the Signature</h5>

<p>Once the Signature Base String is constructed, the next step is signing:</p>

<ol class="apidoc\_ol">

<li>Use the SHA-1 algorithm to generate the hash of the Signature Base String.</li>

<li>Sign the hashed value using the private key of the app.</li>

<li>Base64-encode the signature value.

<span class="apidoc\_note"><strong>Note:</strong> Base64 encoding should <em>not</em> include the CRLF (carriage return/line feed) every 72 characters which is part of strict Base64 encoding. Instead, the whole Base64 encoded string should be without line breaks.</span>

</li>

<li>Optionally, URL-encode the result into a string.</li>

<li>Set the string as the value for the acmecorp\_signature parameter.</li>

</ol>

<p><a href="#top">Back to top</a></p>

<hr />

### <a name="ref\_notes\_and\_procedures"></a><h3>Reference Notes and Procedures</h3>

<p>This section includes overview information about the authentication process and specific details and procedures about aspects of the process of composing the signature. It includes:</p>

<ul>

<li><a href="#def\_sharedsecret">What Is the Shared Secret?</a></li>

<li><a href="#generatingtimestamp">Generating the Timestamp</a></li>

<li><a href="#generatingnonce">Generating a Nonce Value</a></li>

<li><a href="#generating\_keypair\_openssl">Generating PKI Key Pair with Certificate Using OpenSSL</a></li>

<li><a href="#encoding\_base64">Encoding with the Base64 Encoding Scheme</a></li>

<li><a href="#compromise">What If the Private Key or Shared Secret Is Compromised?</a></li>

<!-- <li><a href="#differences">Differences between PKI-Based (Asymmetric) and Symmetric Key-Based Solutions</a></li>

-->

</ul>

#### <a name="def\_sharedsecret"></a><h4>What Is the Shared Secret?</h4>

<p>When you register your app, the AppID and Shared Secret are automatically generated. The Shared Secret is a binary hashed value, generated within the secure environment of the platform and known only to you and to the platform. In the <em>Security Credentials</em> section of the App Details page you'll see the Shared Secret option.</p>

<p>The Shared Secret approach follows the WS-Security digest authentication mechanism.</p>

<p>The Shared Secret, generated by the platform for the app, is a 40-character alphanumeric value that acts as the key. An example is shown below.</p>

<pre><a name="example\_sharedsecret"></a>1008877afabf32efb31f9c974dbeaa688bed0769</pre>

#### <a name="generatingtimestamp"></a><h4>Generating the Timestamp</h4>

<p>The timestamp of the request is sent in the message header, for both Shared Secret and PKI security mechanisms. In the Shared Secret mechanism it is also used in creating the Secret Digest.</p>

<p> The timestamp must be in Unix epoch time, expressed as the number of milliseconds since January 1, 1970 00:00:00 GMT. The timestamp must be a positive integer and must be greater than or equal to the timestamp used in previous requests. In most implementations, the timestamp is taken from the host server. It's important that the timestamp in the message is accurate; if the timestamp is off, the message might be rejected.</p>

<p>For more information on Unix epoch time, and examples of implementation of the timestamp in different programming languages, see <a href="http://www.epochconverter.com" target="\_blank">http://www.epochconverter.com</a> (scroll down).</p>

#### <a name="generatingnonce"></a><h4>(E) Generating a Nonce Value</h4>

<p>A nonce is a random string that is uniquely generated for each request. The nonce allows the API providers to verify that a request has never been made before. A nonce is sent in the message header for both Shared Secret and PKI security mechanisms. In the Shared Secret mechanism, the same nonce value sent in the message header is also used in creating the Secret Digest.</p>

<p>The way you create the nonce will depend on your development environment. Most programming languages include a method for creating a nonce. Below is an example of generating a nonce value in Java:</p>

<pre>Import java.security.SecureRandom:

Random rand = SecureRandom.getInstance ("SHA1PRNG");

long nonce = rand.nextLong();</pre>

#### <a name="generating\_keypair\_openssl"></a><h4>Generating PKI Key Pair with Certificate Using OpenSSL</h4>

<p>You can use the OpenSSL open source product to produce a public/private key pair. For more information on OpenSSL, refer to <a href="http://www.openssl.org" target="\_blank">www.openssl.org</a>.</p>

<p>If you are developing the application using the Microsoft Windows OS you can get an adaptation of OpenSSL for the Windows platform from the following location: <a href="http://code.google.com/p/openssl-for-windows/downloads/list" target="\_blank">http://code.google.com/p/openssl-for-windows/downloads/list</a>.</p>

<h5><a name="issuecert"></a>To create a public/private key pair with OpenSSL</h5>

<ol class="apidoc\_ol">

<li>From the URL given above, download the applicable <strong>openssl</strong> archive file for your operating system.</li>

<li>Unzip the contents to a folder; for example, <strong>C:\openssl</strong>. </li>

<li>Launch the command-line utility <strong>cmd.exe</strong>.</li>

<li>Change location to the folder referenced in step 2; for example, <strong>cd c:\openssl</strong>.</li>

<li>To get a certificate, do one of the following:

<ul>

<li>Submit the certificate request (csr.csr) generated in the previous step to a certificate authority. The certificate authority will issue you an X.509 certificate. See <a href="#issuecert\_ca">To get a certificate issued by a certificate authority</a>.</li>

<li>Generate a self-signed certificate along with a public/private key pair. See <a href="#issuecert\_selfsign">To generate a self-signed certificate</a>.</li>

</ul>

</li>

<li>Upload the certificate file, the result of either of the procedures above, to the Community Manager platform.</li>

</ol>

<p>The keypair and certificate generation process includes the files shown below.</p>

<table width="98%" cellpadding="2" cellspacing="0" class="tableparmsresponse" summary="table description">

<thead>

<tr>

<th width="18%">File name</th>

<th width="14%">Key type</th>

<th width="68%">Comments</th>

</tr>

</thead>

<tbody>

<tr>

<td>privatekey.key</td>

<td>private key</td>

<td><p>Make sure you save a copy of the private key in a safe place. If you lose this key you will need to generate a new key pair. </p>

<p>You will sign your request using the private key. For more information, see <a href="#composing">Composing the Signature</a>.</p></td>

</tr>

<tr>

<td>csr.csr</td>

<td>certificate signing request</td>

<td>Submit this to a Certificate Authority. When you get a certificate in response (see <strong>mycert.cer</strong> below), upload it to the Community Manager platform.</td>

</tr>

<tr>

<td>mycert.cer</td>

<td>X.509 cert (public key)</td>

<td>Upload this file to the Community Manager platform. The API uses this file for signature verification on your app's API calls.</td>

</tr>

</tbody>

</table>

<h5><a name="issuecert\_ca"></a>To get a certificate issued by a certificate authority</h5>

<ol class="apidoc\_ol">

<li>Follow steps 1-4 of <a href="#issuecert">To create a public/private key pair with OpenSSL</a>.</li>

<li>At the command prompt, cut and paste or type the following:

<pre>C:\openssl>bin\openssl req -config openssl.cnf -out csr.csr -newkey rsa:2048 -keyout privatekey.key</pre>

</li>

<li>Enter data for the prompts that appear.

<p class="apidoc\_note"><strong>Note</strong>: Anything with an empty bracket set [] is not required. </p>

<p>Below is an example of what you might see in the command-line window.</p>

<p class="apidoc\_image\_p"><img src="images/generating\_csr2.png" alt="Generating the CSR file" /></p>

<p>This process creates your public/private key pair and a certificate signing request (<strong>csr.csr</strong> file) in the <strong>c:\openssl</strong> folder.</p>

</li>

<li>Submit the <strong>csr.csr</strong> file to a certificate authority. The certificate authority will issue you an X.509 certificate.</li>

<li>Upload the certificate file to the Community Manager platform.</li>

</ol>

<h5><a name="issuecert\_selfsign"></a>To generate a self-signed certificate</h5>

<ol class="apidoc\_ol">

<li>Follow steps 1-4 of <a href="#issuecert">To create a public/private key pair with OpenSSL</a>.</li>

<li>At the command prompt, cut and paste or type the following:

<pre>C:\openssl>bin\openssl req -config openssl.cnf -out mycert.cer -x509 -newkey rsa:2048 -keyout privatekey.key</pre>

</li>

<li>Enter data for the prompts that appear. This process generates a self-signed X.509 certificate (<strong>mycert.cer</strong>).

<p>Below is an example of what you might see in the command-line window.</p>

<p class="apidoc\_image\_p"><img src="images/generating\_csr\_ssl\_02.jpg" alt="Generating a self-signed certificate" /></p>

<p class="apidoc\_note"><strong>Note</strong>: Anything with an empty bracket set [] is not required.</p>

</li>

<li>Upload the <strong>mycert.cer</strong> file to the Community Manager platform.</li>

</ol>

#### <a name="encoding\_base64"></a><h4>Encoding with the Base64 Encoding Scheme</h4>

<p>The Base64 encoding scheme converts binary data into an ASCII string. Encoding turns it into ASCII string format. It's common practice to encode data to make sure it remains intact without modification during transport by HTTP or HTTPS.</p>

<p>As part of setting up your app security, you'll need to encode the values shown below. </p>

<p>If you're using the Shared Secret method with hash/digest, you must encode the binary hash value that you created using the SHA-1 secure hash algorithm; iIf you're using HMAC or RSA signature, you'll need to encode the signature.</p>

<p>Whether you are sending a hash or a signature, you'll need to convert the binary data to ASCII to be sent in the request message, using the Base64 encoding scheme.</p>

<table width="98%" cellpadding="2" cellspacing="0" class="tableparmsresponse" summary="table description">

<thead>

<tr>

<th width="60%">If you are using this security mechanism...</th>

<th width="40%">Encode these parameters...</th>

</tr>

</thead>

<tbody>

<tr>

<td>HMAC Signature with Shared Secret</td>

<td><p>acmecorp\_signature</p></td>

</tr>

<tr>

<td>HASH/Digest with Shared Secret</td>

<td><p>acmecorp\_secret\_digest</p></td>

</tr>

<tr>

<td>RSA Signature with PKI</td>

<td><p>acmecorp\_signature</p></td>

</tr>

</tbody>

</table>

<p>Once the hash i<span class="apidoc\_red">s encoded, the next step is to send the encoded value in the message header as the value for the acmecorp\_secret\_digest par</span>ameter.</p>

#### <a name="compromise"></a><h4>What If the Private Key or Shared Secret Is Compromised?</h4>

<p>If for any reason you know or suspect that your app's Shared Secret or private key might have been compromised, you must replace with a new one immediately. You can generate a new Shared Secret in the platform; you must get a new private key from the <span class="apidoc\_red">issuing authority.</span> Follow the applicable procedure below.</p>

<h5>To generate a new Shared Secret</h5>

<ol class="apidoc\_ol">

<li>Navigate to App Details &gt; Security Credentials.</li>

<li>Click <strong>Regenerate Key</strong>.</li>

<li>Copy the new key and use it in your app in place of the old one.</li>

</ol>

<p class="apidoc\_note"><strong>Note</strong>: It might take up to five minutes for the new key to take effect. During that time, messages sent using the new key will fail. It's best to wait for five minutes before sending messages with the new key.</p>

<h5>To update your Private Key</h5>

<ol class="apidoc\_ol">

<li>Generate a new keypair.</li>

<li>Update:

<ul>

<li>Upload the new key to the platform.</li>

<li>Update your app to use the new key.</li>

</ul>

</li>

<li>Test.</li>

</ol>

<!--<a name="differences"></a>

<h5>Differences between PKI-Based (Asymmetric) and Symmetric Key-Based Solutions</a></h5>

<p>Note: got this section from Mujeeb's page: https://pdolinux1/wiki/index.php/Application\_Runtime\_Security\_%28Shared\_Secret%29#Differences\_between\_PKI\_based\_and\_Symmetric\_key\_based\_solutions</p>

<p>However some of the content is on the WEb.</p>

<p>Need to understand it and write something simple on this topic.</p>

<p>One resource that came up on a search for a snippet of the text below: http://pic.dhe.ibm.com/infocenter/tpfhelp/current/index.jsp?topic=%2Fcom.ibm.ztpf-ztpfdf.doc\_put.cur%2Fgtps7%2Fs7symm.html.</p>

<p>With most symmetic-key algorithms, the same key is used for both encryption and decryption.</p>

<p>With symmetric-key encryption, the encryption key can be calculated from the decryption key and vice versa. </p>

<p>With Symmetric-key encryption, information encrypted with one symmetric key cannot be decrypted with any other symmetric key. The same key must be used.</p>

<p>PKI based encryption (also called asymmetric encryption) involves a pair of keys&#8212;a public key and a private key&#8212;associated with an entity that needs to authenticate its identity electronically or needs to sign or encrypt data. Data encrypted with your private key can be decrypted only with your public key and vice versa. </p>

<p>Compared with symmetric-key encryption, public-key encryption requires more computation. Implementations of symmetric-key encryption can be highly efficient, so that users do not experience any significant time delay as a result of the encryption and decryption. Symmetric-key encryption is effective only if the symmetric key is kept secret by the two parties involved. If anyone else discovers the key, it affects both confidentiality and authentication. A person with an unauthorized symmetric key not only can decrypt messages sent with that key, but can encrypt new messages and send them as if they came from one of the two parties who were originally using the key. </p>

<p>With public-key encryption, instead of encrypting the data itself, the signing software creates a one-way hash of the data, then uses the private key to encrypt the hash. The encrypted hash, along with other information, such as the hashing algorithm, is known as a digital signature. To validate the integrity of the data, the receiving software first uses the signer's public key to decrypt the hash. It then uses the same hashing algorithm that generated the original hash to generate a new one-way hash of the same data. If the two hashes match, the recipient can be certain that the public key used to decrypt the digital signature corresponds to the private key used to create the digital signature. The significance of a digital signature is comparable to the significance of a handwritten signature. Once you have signed some data, it is difficult to deny doing so later-assuming that the private key has not been compromised or out of the owner's control. This quality of digital signatures provides a high degree of nonrepudiation-that is, digital signatures make it difficult for the signer to deny having signed the data. In some situations, a digital signature may be as legally binding as a handwritten signature. </p>

-->

<p><a href="#top">Back to top</a></p>

<hr />

### <a name="errors\_all"></a><h3>Error Codes and Descriptions</h3>

<p>If there's a problem with your HTTPS header, you might see one of the error messages shown below.</p>

<!--<p>{{Queries and suggested text changes on error messages.}}</p>-->

<table width="84%" cellpadding="2" cellspacing="0" class="tableparmsresponse" summary="table of error messages">

<thead>

<tr>

<th width="10%">Code</th>

<th width="66%">Description</th>

</tr>

</thead>

<tbody>

<tr>

<td>1010701</td>

<td>Required HTTP header parameter missing. [<strong>{0}</strong>]</td>

</tr>

<tr>

<td>1010702</td>

<td>One or more invalid HTTP header parameters.</td>

</tr>

<tr>

<td>1010703</td>

<td>Invalid <u>Nonce</u>. The value of the acmecorp\_nonce field has already been used.</td>

</tr>

<tr>

<td>1010704</td>

<td>Invalid timestamp. The value of the acmecorp\_timestamp field is out of range.</td>

</tr>

<tr>

<td>1010705</td>

<td>Signature or digest algorithm is not supported. [<strong>{0}</strong>]</td>

</tr>

<tr>

<td>1010706</td>

<td>Signature or digest verification failed.</td>

</tr>

<tr>

<td>1010707</td>

<td>Missing nonce. The acmecorp\_nonce field value is required.</td>

</tr>

<tr>

<td>1010708</td>

<td>Unable to verify signature. There is no public key associated with the app.</td>

</tr>

<tr>

<td>1010709</td>

<td>Authentication scheme is invalid or missing.</td>

</tr>

<tr>

<td>1010710</td>

<td>Invalid AppID. The value [{0}] in the acmecorp\_app\_id field is invalid or missing.</td>

</tr>

<tr>

<td>1010711</td>

<td>Unable to verify signature. There is no shared secret associated with the <u>app</u>.</td>

</tr>

<tr>

<td>1010712</td>

<td>Invalid timestamp. Timestamp must be Unix epoch time in milliseconds.</td>

</tr>

</tbody>

</table>

<p><a href="#top">Back to top</a></p>

<hr />

</div>

</body>

</html>