Security Best Practices

The Gigya service is designed to provide maximum security for your critical data. Please follow the guidelines provided in this document to make sure you are leveraging all the security features built into the Gigya service.

When using any REST API that transmits sensitive data, e.g., passwords or secrets, recommended best practice is to always use POST requests via an SSL secured connection. When using any Gigya APIs, attempting to send a secret over a non-SSL connection will result in an *errorCode: 403006 - Secret Sent Over Http*.

Protect Your Secret Key

Your "**Secret Key**" is provided in BASE64 encoding, at the bottom of the Site table in the Dashboard section on the Gigya's website (please make sure that you have logged in to Gigya's website and that you have completed the Gigya's Site Setup process).

The secret key is a cryptographic random number used as a shared secret between your application and Gigya. Anyone who gains access to this key may pretend to be you and perform actions on your users on your behalf, therefore it is crucial to protect the secret key. Take extra caution and never ever use the secret key on a client where malicious users could gain access to it.

Choose HTTPS

Gigya's Web SDK, REST API and SDKs API calls should be made over **HTTPS** (SSL). Using HTTPS ensures that no one can eavesdrop on the data passed between the server and your user's browser. You should always use HTTPS with Gigya APIs.
If your website uses a secure HTTPS domain, when loading the Gigya Web SDK on a web page, load it from our HTTPS domain (i.e., reference "CDNS" and not "CDN", as per the example below). This will not only load the SDK itself over HTTPS but will cause the SDK to perform all its communications with the Gigya server over HTTPS as well.
As a reminder, every page that uses the Gigya API must load Gigya's Web SDK in the <HEAD> section (refer to Getting Started for the complete guide). On secured pages, the line of code should be:

```
<script
src="https://cdns.gigya.com/js/gigya.js?apikey=INSERT-YOUR-APIKEY-HERE"
></script>
```

You can use a simple script that determines whether the page is being served over HTTP or HTTPS, so as to dynamically construct the script that loads Gigya's Web SDK.

```
//This places the Gigya Web SDK at the end of the page <head>
function addGigyaWebSDK(apiKey) {
    var apiKey, gig, gSrc;
    if (apiKey) {
        apiKey=apiKey;
        if ((window) && (typeof(window.location.protocol) !=='undefined'))
{
         gig = window.location.protocol == "https:" ?
"https://cdns.gigya.com/js/gigya.js?apiKey=" :
"http://cdn.gigya.com/js/gigya.js?apiKey=";
         gig = gig + apiKey;
         gSrc = document.createElement('script');
         gSrc.type="text/javascript";
         gSrc.src=gig;
         document.getElementsByTagName('head')[0].appendChild(gSrc);
        }
    } else {
        if (console) {
            console.warn("No Gigya API Key Found!");
        }
    }
}
addGigyaWebSDK('Your-API-Key');
```

Notes:

- In the above code example, note **https://cdns**.gigya.com, as opposed to the non-SSL SDK located at http://cdn.gigya.com.
- If you are using generated code or one of our code examples, the above line should **substitute** the equivalent line that loads Gigya's Web SDK from an HTTP domain.

Validate the UID Signature in the Social Login Process

The social login process requires the user to successfully login to one of the providers supported by Gigya. Obviously, this process has to happen on the client side, which means there is a risk that a malicious user will try to tamper with the data sent from the client to the server. The most important piece of data is the UID, which is used to authenticate the user and log him/her into your system. Gigya uses HMAC-SHA1 digital signatures to prevent tampering with the UID and therefore it is crucial to validate the signature before logging the user in based on the UID coming from the browser.

It is important to note that the **UIDSignature** and **signatureTimestamp** properties are only returned to client-side calls and **only** after a successful login. The two aforementioned properties are never returned when a user is still **pending registration**, as access to these fields may allow an unauthorized user interaction with your server.

Refer to the Social Login documentation, for a complete overview of the social login flow and the signature verification within it.

If you are using one of Gigya's SDKs, implementation is simple. Use the SigUtils.validateUserSignature() method provided in your SDK.

The method receives four parameters: the UID to be validated, a timestamp, the secret key and the signature. Obtain your partner's 'Secret Key' from Gigya's Site Setup page. The UID, timestamp and signature are provided in the UID, signatureTimestamp, UIDSignature fields of the User object respectively.

See further details in the reference documentation for your SDK.

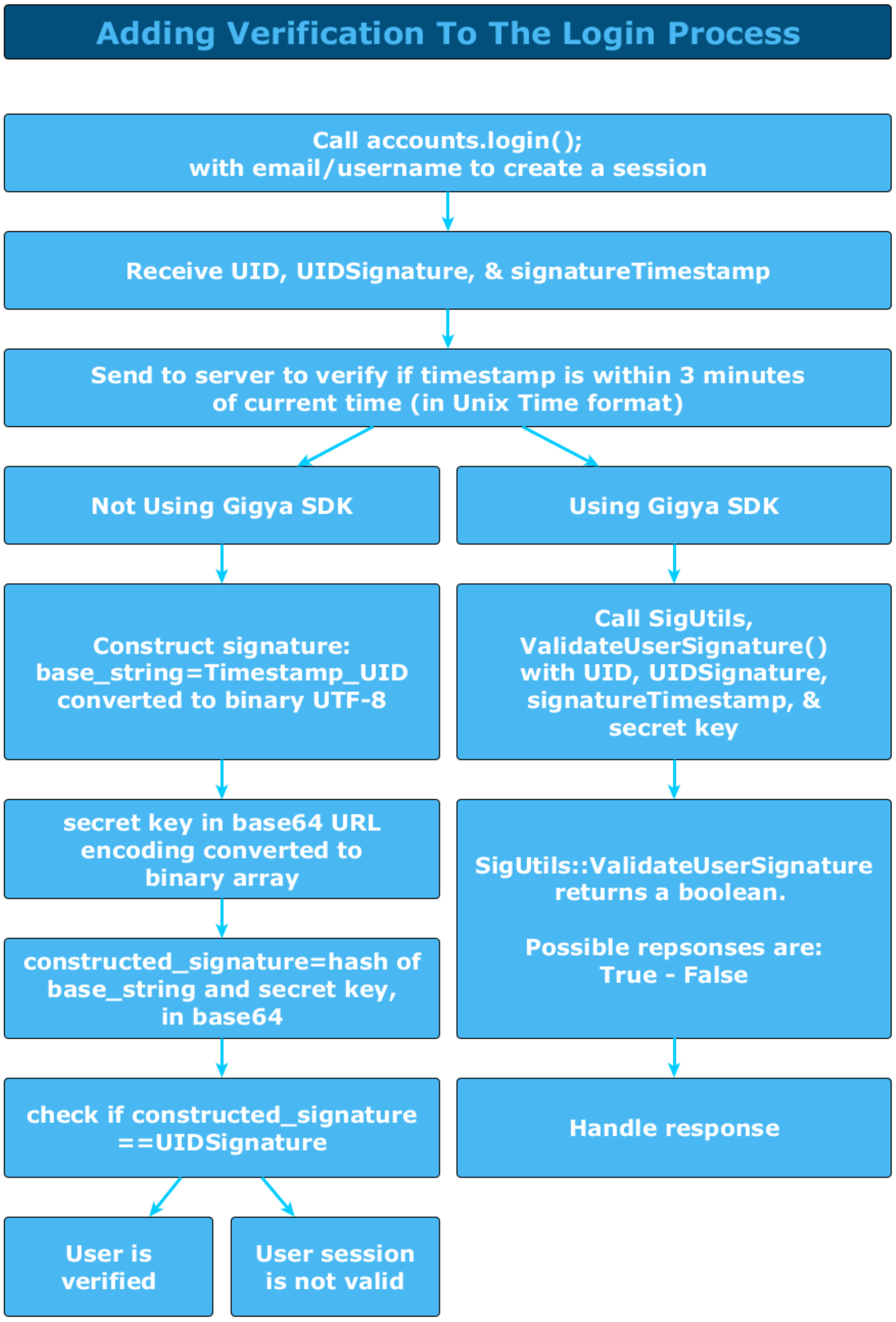
If there is no SDK available for your language and you are using our Web SDK you will need to implement the signature validation process. The following section gives the implementation steps.

Using A User Key or Application Key

If you are using a user key or application key and their corresponding secret (as opposed to the partner secret in your Dashboard) you can use accounts.exchangeUIDSignature to verify a UID signature with a userKey/secret instead of a Partner secret.

Signature Validation Process

The following graphic illustrates the login and validation process:


Adding Verification To The Login Process
Call accounts.login(); with email/username to create a session
Receive UID, UIDSignature, & signatureTimestamp
Send to server to verify if timestamp is within 3 minutes of current time (in Unix Time format)
Not Using Gigya SDK
Using Gigya SDK
Construct signature: base_string=Timestamp_UID converted to binary UTF-8
Call SigUtils, ValidateUserSignature() with UID, UIDSignature, signatureTimestamp, & secret key
secret key in base64 URL encoding converted to binary array
SigUtils::ValidateUserSignature returns a boolean.
Possible repsonses are: True - False
constructed_signature=hash of base_string and secret key, in base64
check if constructed_signature ==UIDSignature
Handle response
User is verified
User session is not valid

Please follow these steps to implement signature validation:

1. Validate that the timestamp is within 3 minutes of your current server time. Note that the timestamp is in Unix time format, meaning the

number of seconds since Jan. 1st 1970 in the GMT/UTC timezone. The timestamp is provided in the signatureTimestamp field of the User object.
2. Construct a signature from the UID, signatureTimestamp and your secret key. Follow the instructions in the Constructing a Signature section below.
3. Compare the signature you have calculated to the one generated by Gigya. If the UID signature is valid the two would be exactly the same. The Gigya signature is provided in the UIDSignature field of the User object.

The following is the validation process in pseudo-code:

```
boolean isValidSignature(timestamp, UID, secretKey, signature) {
    If abs(now-timestamp)>180 then return false;      // Validate that the
timestamp is within 3 minutes of your current server time
    mySignature = constructSignature(timestamp, UID, secretKey);  // See
the code of this method in "Constructing a Signature" section below
    if (mySignature<>signature) return false;        // Compare the
signature you calculated to the one received
    return true;
}
```

The following is a PHP example of validating a signature using our PHP SDK.

```
<?php
require_once('GSSDK.php');
require_once('secret.inc.php');
require_once('apiKey.inc.php');
$UID = <UID>;
$timestamp = <signatureTimestamp>;
$secret = <secret>;
$signature = <UIDSignature>;
$isValidSession =
SigUtils::validateUserSignature($UID,$timestamp,$secret,$signature);
if ($isValidSession === true) {
    echo 'Validation passed!<br />';
} else {
    echo 'Validation failed!<br />';
}
?>
```

The following are Perl and Ruby implementations of the validation process not using an SDK:

Perl

```
use Digest::HMAC_SHA1 qw(hmac_sha1);
use MIME::Base64;

# construct signature
sub constructSignature {
       my ($key, $uid, $timestamp) = @_;
       return (encode_base64(hmac_sha1($timestamp.'_'.$uid,
decode_base64($key)), ''));
}

# compare incoming signature with constructed signature
sub isValidSignature {
       my ($key, $uid, $timestamp, $signature) = @_;
       my $constructedSignature = constructSignature($key, $uid,
$timestamp);
       return($signature eq $constructedSignature);
}
```

Constructing a Signature

Follow these steps to produce a signature:

Your partner's "**Secret Key**" is provided in BASE64 encoding, at the bottom of the Sites Table in the Dashboard section in Gigya website (please make sure that you are logged in to Gigya's website and that you have completed the Gigya's Site Setup process).

1. Construct a "base string" for signing: "%Timestamp%_%UID%" replacing %Timestamp% and %UID% with the corresponding values.
2. Convert the base string into a binary array using UTF-8 encoding.
3. Optional - If you have a mechanism for storing and verifying cryptographic nonces it is recommend that you store the base string as a nonce and verify that you only get the same base string once.
4. Convert your "**Secret Key**" from its BASE64 encoding to a binary array.
5. Use the HMAC-SHA1 algorithm to calculate the cryptographic signature of the "base string" constructed in step 1, with your binary "**Secret Key**" calculated in step 4 as the key. The HMAC-SHA1 algorithm is implemented in many standard libraries and is readily available in any web development environment. The HMAC-SHA1 method usually receives two parameters: a *binary key*, and a *buffer* to be signed. It returns a binary array containing the signature.
6. Convert the signature to a BASE64 string.

Following is a ***pseudo*** code of the signature construction:

```
string constructSignature(string timestamp, string UID, string secretKey) {
    baseString = timestamp + "_" + UID;                          //
Construct a "base string" for signing
    binaryBaseString = ConvertUTF8ToBytes(baseString);          // Convert
the base string into a binary array
    binaryKey = ConvertFromBase64ToBytes(secretKey);            // Convert
secretKey from BASE64 to a binary array
    binarySignature = hmacsha1(binaryKey, binaryBaseString);    // Use the
HMAC-SHA1 algorithm to calculate the signature
    signature = ConvertToBase64(binarySignature);               // Convert
the signature to a BASE64
    return signature;
}
```

Notes:

1. Signature construction should be implemented on the server side.
2. When sending the parameters from your client to your server, please make sure you encode the **UID** parameter using the encodeURIComponent() function, before sending it to your server, and then decode it on your server before calculating the signature.
3. This algorithm implementation applies when using Gigya's Web SDK or a Mobile SDK. When using our REST API, please sign the HTTP requests using the algorithm that is described under the Signing requests section of the Using the REST API guide.

Validate Friendship Signatures when Required

In some scenarios it is required to validate friendship signatures to make sure the user is really a friend of another user. For example, suppose you have a news web site and you keep track of your articles read by your users. When a user logs in to your site you want to show her what articles her friends have recently read on your site. Without signature verification a malicious user may tamper with the friends list and pretend to be a friend of users he is not really friends with, by that he would gain access to the articles read by those users and violate their privacy. To prevent this scenario Gigya provides an optional "friendship signature" that verifies the two users are really friends.

In order to enable the friendship signature you must set the signIDs field to '*true*' in the global configuration object passed to socialize.getFriendsInfo method. The result of this action is that all the Friend objects returned by these methods will be signed by Gigya. Now your job is to validate the signature received within each Friend object.

If you are using one of Gigya's SDKs, implementation is simple. Use the ***SigUtils.validateFriendSignature()*** method provided in your SDK.

The method receives five parameters:

1. UID - the UID of the current user.
2. timestamp - use the ***signatureTimestamp*** field of the Friend object.
3. friendUID - the UID of the friend to be validated. Please use the ***UID*** field of the Friend object.
4. secret - use your partner's 'secret key', obtained from Gigya's Dashboard page.
5. signature - use the ***friendshipSignature*** fields of the Friend object.

If there is no SDK available for your language and you are using our Web SDK you will need to implement the signature validation process. The process is identical to the process described in the Signature Validation Process section, except the base string is constructed as "%timestamp%_%friendUID%_%UID%". Where %timestamp% is provided in the signatureTimestamp field of the Friend object, %friendUID% is provided in the UID field of the Friend object, and %UID% is the current user's UID.

Signed UIDs Passed to Gigya's Web SDK Methods

For the same reasons mentioned above, some client side API methods, such as socialize.notifyRegistration, socialize.setUID and socialize.notifyLogin require you to sign UIDs passed to them in order to validate the authenticity of those UIDs. It is vital that you construct these signatures on your server side and pass them to your client for making the API call. The **UIDSig** parameter of the socialize.notifyRegistration, socialize.setUID and socialize.notifyLogin methods is defined for this objective, and is a required parameter. Gigya will verify the authenticity of this signature to prove that it is in fact coming from your site and not from somewhere else.

Follow the instructions below to set the signature parameter of the method, and make the API call as soon as possible after that to prevent the signature from expiring.

If you are using one of Gigya's SDKs, implementation is simple. Use the ***SigUtils.calcSignature()*** method provided in your SDK. The method returns a signature as expected by Gigya. The method receives two parameters:

1. baseString - construct the base string according to the specification in each respective method reference documentation.
2. key - use your partner's 'secret key', obtained from Gigya's Dashboard page.

If there is no SDK available for your language and you are using our Web SDK you will need to implement the signature calculation process. Please follow the implementation steps described in the Constructing a Signature section.

Use REST API Calls when Verified User Data is Required

The signature mechanism described above will prevent any tampering with users' and friends' IDs but it is not meant to protect other data fields, like the user name or email address. If your application requires this data to be authenticated with the data received from the provider, you should make a REST API call to Gigya, directly from your server to retrieve that data. Learn more in the Using the REST API guide.

Signing REST API Calls

When making REST API calls over HTTP the call has to be signed using the OAuth 1.0 signature calculation method. For more information see the Signing requests section of the Using the REST API guide.

General Note about Signatures

Remember that signatures contain binary data encoded using the BASE64 format. BASE64 format (as well as email addresses, etc.) uses characters that must be properly encoded when passed between the browser and the server or the signature verification will fail. Be sure to use **encodeURIComponent()** on all pertinent strings.

Example

```
userKey='LV/24EGI7T+xl';
userKey=encodeURIComponent(userKey); // >> LV%2F24EGI7T%2Bxl
```

Additional Information

- Managing Session Expiration
- Application Keys

Special Information - Data Controllers

The Gigya platform does not include any technical measures that support the processing of Special Categories of Personal Data, which includes but is not limited to:

- racial or ethnic origin
- political opinions
- religious or philosophical beliefs
- trade-union membership
- health or sex life
- personal data concerning bank and credit accounts
- genetic data
- biometric data for the purpose of uniquely identifying a natural person