Client-side encrypted API tokens

- Objectives
- Terminology
- REST API requests are stateless
- Server-side apps are secure
- Client-side apps are not secure
- API requests from client-side apps
  - Using proxy servers
  - Encrypting tokens for API requests
  - Making an API request with an encrypted token
- Sample code
  - PyCrypto
  - C#
  - node.js
  - Java
- Compromised authentication tokens
- Technical details
  - Validating your encrypted token
  - Java

Objectives

For both Xignite commercial reasons and data provider requirements, Xignite employs authentication for its APIs to prevent unauthorized access. Exchange agreements vary by exchange, as do agreements with other data vendors such as S&P, but all require some form of data access restrictions and detailed usage reporting. As such, we designed Xignite API authentication to achieve the following objectives:

- **Enforce only authorized access** - Xignite authenticates API requests to ensure that only Xignite customers have access to its APIs, and only to the APIs that each customer is entitled to. Xignite also ensures that each customer can access only the underlying data that each customer is entitled to, for example prices from specific exchanges (e.g., NYSE, NASDAQ), pricing freshness (e.g., real-time, delayed, or historical), and data values from specific vendors (e.g., S&P, LSE),
- **Optimize API response times** - Xignite APIs are designed to be called directly from a customer's application, so response time is critical. Xignite API authentication is designed to minimize overhead to optimize response time.
- **Minimize complexity** - Xignite APIs are designed to be simple to use, which also extends to Xignite API authentication.

Terminology

We use these terms in describing Xignite API authentication:

Term	Definition	Example
Customer	An Xignite customer. A company that has a business relationship with Xignite.	Personal Capital
End-user	An end-user of a customer. An end-user has a business relationship with a customer and not Xignite, which includes end-users of a customer's public website.	A Personal Capital end-user
Authentication token	An alphanumeric value assigned by Xignite that a customer uses to authenticate its Xignite API requests.	ABCDEFGHIJKLMNOPQRSTUV123456
Encryption key	An alphanumeric value that a customer uses to encrypt its authentication token.	HIGKLMN678901234
Encryption initialization vector	An additional alphanumeric value that a customer uses to encrypt its authentication token.	PQRSTUVWXYZ01234
User ID	A unique numeric ID that Xignite assigns to each customer.	112233

REST API requests are stateless

REST APIs are a major reason why Xignite can establish a highly scalable, cloud-based API platform for financial market data. Each REST API request is stateless, which means that Xignite handles each request that a customer makes independently of any prior request that the customer may have made earlier. As every request is independent, every request requires authentication. Xignite optimizes the authentication process through in-memory caching of valid tokens and their corresponding authorizations, which minimizes any overhead to ensure rapid response times.

Server-side apps are secure

API requests from a server that the customer controls are secure. Or more specifically, API requests that include a customer's authentication token are secure because the API request is neither publicly accessible nor accessible by any end-user. So from a server-side app, a customer can simply include its authentication token in an Xignite API request. For example, a customer that makes a REST API request for the latest price for Microsoft can call XigniteGlobalQuotes with this URL:

```
https://globalquotes.xignite.com/xglobalquotes.json/GetGlobalDelayedQuotes?IdentifierType=Symbol&Identifiers=MSFT&_token=ABCDEFGHIJKLMNOPQRSTUV123456
```

where `ABCDEFGHIJKLMNOPQRSTUV123456` is that customer's authentication token.

Xignite authenticates each request using the token that a customer provides, and verifies that that customer is authorized to access that API and market data. In the prior example, the request for the latest price of Microsoft would succeed assuming:

- The token is valid
- The customer is authorized to use XigniteGlobalQuotes
- The customer is authorized to access NASDAQ delayed prices

Client-side apps are not secure

Client-side apps include public websites, private websites (i.e., websites that require a login), and mobile apps. API requests from client-side apps are not secure, so these requests require that the customer use an encrypted token instead of its authentication token.

Do not use an authentication token within a client-side app. Doing so would enable anyone who navigates to your website or downloads your mobile app to gain access to your authentication token and make unauthorized Xignite API requests on your behalf. For example, websites are not secure, as any end-user that navigates to a website can also use the browser to examine the contents of that website. That would include examining any Xignite API requests made within a website, and therefore compromise any authentication token that may be included within those requests.

As only an authentication token is required to validate an Xignite API request, it is important to prevent compromising these tokens. Xignite requires additional security precautions and disallows simply including a customer's authentication token due to the ease in which it could be compromised. From within a mobile app, an authentication may be slightly more difficult to retrieve than from within a browser. But it is still a security vulnerability because of the same underlying reason — the authentication token is distributed within the mobile app to an end-user, and therefore can be compromised.

API requests from client-side apps

The following options are available for making Xignite API requests from client-side apps:

- **Proxy server** - The customer's client-side app makes calls to a proxy server residing in the customer's domain; the proxy server makes Xignite API requests and returns the results back to the customer's client-side app.
- **Encrypted tokens** - The customer's client-side app makes direct Xignite API calls using an encrypted token that the customer generates.

Using proxy servers

Proxy servers secure authentication tokens simply by not exposing the API request (and therefore the authentication token included within an API request) to the public or any end-user. By making Xignite API requests from private proxy servers that a customer controls, end-users cannot examine these requests and therefore cannot compromise the customer's authentication

token within these requests. Using this approach does, however, require that the customer deploy sufficient proxy servers to relay all requests between its client-side apps and Xignite.

From Xignite's perspective, API requests from proxy servers are the same as any other server-side API requests, as they are made simply using the customer's authentication token. There is no distinction to Xignite between an API request from a proxy server as compared to an API request from any other server-side app.

Encrypting tokens for API requests

Xignite enables encrypting authentication tokens and using these encrypted tokens for API requests from client-side apps. In order to secure authentication tokens while maintaining high-performance for API requests, Xignite employs the following approach:

- AES-128 symmetric encryption
- Customer-definable encryption key
- Customer-definable encryption initialization vector
- Embedding a timestamp for expiring an encrypted token.

In the My Accounts page, each customer can define an encryption key, initialization vector, and expiration time (up to a maximum 30 minutes) for encrypting its authentication tokens. And when encrypting a token, a customer includes the current timestamp; the encrypted token is valid only for the duration of time as specified by the expiration time for that customer.

Do not use an encryption key or initialization vector within a client-side app. As with unencrypted authentication tokens, doing so would enable anyone who navigates to a website or downloads a mobile app to decrypt the encrypted token and gain access to an authentication token.

Xignite automatically detects API requests that originate from a website, and enforces the use of encrypted tokens in these requests. In other words, from a website, an API request that includes the customer's authentication token will fail.

Making an API request with an encrypted token

Xignite supports encrypting an authentication token with a 128-bit AES key and Cipher Feedback mode. Under My Accounts->Account->Settings, you define 16-character ASCII encryption keys and initialization vectors, which generates a 128-bit key. Cipher Feedback mode supports encrypting any length character string, so there is no need to pad out a timestamped token to a certain length. Note that Xignite uses a feedback size for Cipher Feedback mode of 8 bits; some implementations (such as in C#) allow you to set the feedback size, which you should ensure is set to 8 bits.

Here are the steps in making an Xignite API request with an encrypted token:

- Customer creates an encrypted token:
    - Retrieve current datetime using the UTC timezone
    - Format current datetime in ISO 8601 format, YYYY-MM-DDTHH24:MI:SS, for example “2015-08-15T16:01:30” for August 15, 2015 at 4:01:30 PM.
    - Concatenate your authentication token and current timestamp using the following format: “<authentication token>|<current timestamp>”. For example:

      ”ABCDEFGHIJKLMNOPQRSTUV123456|2015-08-15T16:01:30”

    - Use your encryption key and initialization vector for encryption.
    - Use any AES function to encrypt the string. For example:
        - .NET Framework Class Library System.Security.Cryptography
        - Java Cryptography Extension
        - Bouncy Castle Crypto Library (for Java and C#)
        - PyCrypto

- Includes the encrypted token and the user ID in the Xignite API request using these query string parameters:
    - _token -- The encrypted token
    - _token_userid -- The customer’s user ID
    - Using the earlier API request example with an encrypted token:

```
https://globalquotes.xignite.com/xglobalquotes.json/GetGlobalDelayedQuotes?IdentifierType=Symbol&Identifiers=MSFT&_token=A1B2C3D4E5F67890&_token_userid=112233
```

For example, for websites a customer can generate the API request prior to serving the webpage to an end-user, and re-generating the API request with a new encrypted token prior to the expiration of that encrypted token.

For downloaded client-side applications, a customer can implement a server-side function that generates the encrypted token (or the entire API request that includes the encrypted token). The client-side application can call the customer’s server-side function to retrieve the encrypted token (or entire API request that includes the encrypted token).

Note that in all cases, the unencrypted authentication token, the encryption key, and the initialization vector are not present on the client-side application.

Sample code

PyCrypto

```
import urllib2
```

```
from time import gmtime, strftime
from Crypto.Cipher import AES

TOKEN="ABCDEFGHIJKLMNOPQRSTUV123456"
KEY="HIGKLMN678901234"
IV="PQRSTUVWXYZ01234"

# Get current UTC time
current_time = strftime("%Y-%m-%d %H:%M:%S", gmtime())

# Concatenate authentication token with current UTC time
timestamped_token = TOKEN + "|" + current_time

# Create an AES-128 cipher using Cipher Feedback
cipher = AES.new(KEY, AES.MODE_CFB, IV)

# Encrypt the timestamped token
binary_encrypted_token = cipher.encrypt(timestamped_token)

# Convert the binary encrypted token into a hex string
hex_encrypted_token = binary_encrypted_token.encode('hex')

# Call an Xignite API
u = urllib2.urlopen("http://globalquotes.xignite.com/xGlobalQuotes.xml/" + \
                    "GetGlobalDelayedQuotes" + \
                    "&_token=" + hex_encypted_token + \
                    "&_token_userid=112233")
```

C#

```
// Get current UTC time
var time = DateTime.UtcNow.ToString("s");

// Concatenate authentication token with current UTC time
var token = string.Format("{0}|{1}", "ABCDEFGHIJKLMNOPQRSTUV123456", time);

// Create an AES-128 cipher using Cipher Feedback
var algo = new RijndaelManaged
  {
    Mode = System.Security.Cryptography.CipherMode.CFB, KeySize = 128,
      Padding = PaddingMode.None, FeedbackSize = 8
  };

// Encrypt the timestamped token
const string key = "HIGKLMN678901234";
const string iv = "PQRSTUVWXYZ01234";

using (var encryptor = algo.CreateEncryptor(Encoding.ASCII.GetBytes(key),
```

```
                                            Encoding.ASCII.GetBytes(iv)))
{
  using (var msEncrypt = new MemoryStream())
  {
    // Create the streams used for encryption
    using (var csEncrypt = new CryptoStream(msEncrypt, encryptor,
                                            CryptoStreamMode.Write))
    {
      var buffer = Encoding.ASCII.GetBytes(token);
      csEncrypt.Write(buffer, 0, buffer.Length)
    }

    var binaryEncryptedToken = msEncrypt.ToArray();

    // Convert to a hex string
    var hexEncryptedToken = new StringBuilder(binaryEncryptedToken.Length);
    for (int i = 0; i < binaryEncryptedToken.Length; i++)
    {
      HexEncryptedToken.Append(binaryEncryptedToken[i].ToString("X2"));
    }

    // Call an Xignite API
    var url = "http://globalquotes.xignite.com/xGlobalQuotes.xml/" +
              "GetGlobalDelayedQuotes?Identifiers=MSFT&IdentifierType=Symbol" +
              "&_token=" + hexEncryptedToken +
              "&_token_userid=112233";

    var response = new WebClient().DownloadString(url);
  }
}
```

node.js

```
var crypto = require('crypto');

var TOKEN="ABCDEFGHIJKLMNOPQRSTUV123456";
var KEY="HIGKLMN678901234";
var IV="PQRSTUVWXYZ01234";

function createToken(){
   // Concatenate authentication token with current UTC time
  var toEncrypt=TOKEN+"|"+new Date().toISOString().substr(0,19);

  // Create an AES128 cipher using Cipher Feedback
  var cipher=crypto.createCipheriv("aes-128-cfb8",KEY,IV);

  // Encrypt the timestamped token into a hex string
  var token=cipher.update(toEncrypt,"utf8","hex")+cipher.final("hex");
```

```
  return token;
}

var encryptedToken=createToken();
```

Java

```
import javax.crypto.Cipher;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import javax.xml.bind.DatatypeConverter;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.TimeZone;

// FROM SETTINGS PAGE ON CLIENT PORTAL
String iv = "PQRSTUVWXYZ01234";
String key = "HIGKLMN678901234";

// YOUR API TOKEN HERE
String APIToken = "ABCDEFGHIJKLMNOPQRSTUV123456";

SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss");
sdf.setTimeZone(TimeZone.getTimeZone("UTC"));

String formattedTime =
sdf.format(Calendar.getInstance(TimeZone.getTimeZone("UTC")).getTime());

// Concatenate authentication token with current UTC time
String token = String.format("%s|%s", APIToken, formattedTime);

// function call to encrypt the token
String encryptedToken = encrypt(key, iv, token);

public static String encrypt(String key, String initVector, String value)
{
      try {
            IvParameterSpec iv = new IvParameterSpec(initVector.getBytes());
            SecretKeySpec skeySpec = new SecretKeySpec(key.getBytes(), "AES");

            Cipher cipher = Cipher.getInstance("AES/CFB8/NOPADDING");
            cipher.init(Cipher.ENCRYPT_MODE, skeySpec, iv);

            byte[] encrypted = cipher.doFinal(value.getBytes());

            return DatatypeConverter.printHexBinary(encrypted);
```

```
        } catch (Exception ex)
            {
                ex.printStackTrace();
                return null;
            }
}
```

Compromised authentication tokens

Xignite monitors API usage for unusually high traffic patterns and will contact customers to determine whether increased traffic is expected. If the increased traffic cannot be accounted for, it may indicate that the customer's authentication token and/or encryption key has been compromised. Xignite may elect to re-issue a new authentication token for that customer.

In extreme circumstances, for example, if a sudden, unexpected increase in traffic can potentially have an impact on Xignite production operations, Xignite may elect to immediately disable an authentication token while attempting to contact that customer.

Technical details

In order to properly enforce encrypted token expirations, Xignite validates timestamps to ensure that they are not in the future. Xignite rejects encrypted tokens with timestamps in the future as invalid.

For API request originating from a web page, Xignite uses the Cross-Origin Resource Sharing (CORS) standard to enforce the use of an encrypted token. In the HTTP header, Xignite sets Access-Control-Allow-Origin to the caller if that API request passes an encrypted token and Xignite can successfully validate the token. Xignite sets Access-Control-Allow-Origin for any API call in which Xignite can validate the token. If an API request passes an unencrypted token, or Xignite cannot validate the token, Access-Control-Allow-Origin is disabled, which causes the client's browser to trigger an XMLHttpRequest error.

Validating your encrypted token

When you encrypt tokens for an API call, it is useful to verify that you have correctly encrypted your token. Here is a sample python script for decrypting an encrypted token to confirm that what you are providing is properly encrypted. The inputs to this python script are the encrypted token, your account's key, and your account's initialization vector.

```
#!/usr/bin/python

import codecs
import binascii
import sys
from Crypto.Cipher import AES
```

```
# Decrypt an encrypted token to validate that it was encrypted correctly

def main():

  args = sys.argv[1:]
  if not args:
    print 'usage: key iv encrypted-token'
    sys.exit(1)

  AES_KEY = args[0]
  AES_IV = args[1]
  TOKEN = args[2]

  cipher = AES.new(AES_KEY, AES.MODE_CFB, AES_IV)
  binary_encrypted_token = codecs.decode(TOKEN,”hex”)
  decrypted_token = cipher.decrypt(binary_encrypted_token)

  print "Decrypted token: ",decrypted_token

  return

if __name__ == '__main__':
  main()
```

Java

```
public static String decrypt(String key, String initVector, String encrypted)
{
      try {
            IvParameterSpec iv = new IvParameterSpec(initVector.getBytes());
            SecretKeySpec skeySpec = new SecretKeySpec(key.getBytes(), "AES");

            Cipher cipher = Cipher.getInstance("AES/CFB8/NOPADDING");
            cipher.init(Cipher.DECRYPT_MODE, skeySpec, iv);

            byte[] original =
            cipher.doFinal(DatatypeConverter.parseHexBinary(encrypted));

            return new String(original);
          } catch (Exception ex)
            {
                 ex.printStackTrace();
            }
      return null;
 }
```