The wonderful applications of HMAC

Stellan Söderström

White Paper

Table of contents

Introduction

This paper describes the ”Hashed Message Authentication Code” or HMAC for short and a few examples of its applications. In many situations, the use of an HMAC, ensures a high level of security at the same simplifying otherwise complex solutions.

Background

HMAC construction was first published in 1996 by Mihir Bellare, Ran Canetti, and Hugo Krawczyk, Its structure is described thoroughly in RFC2104. In short, it defines a way of verifying both the integrity and authenticity of a message, building on a hash function in combination with a secret key.

A simplified way of defining a formula for HMAC can be written as:

```
HMAC = hashFunction(message + key)
```

The above shows the principle. The actual implementation requires some additional steps, such as padding and two passes of hashing the key, but what you see here is the general, elegantly simple idea.

The hash function can be any cryptographically secure algorithm and while the RFC suggests MD5 and SHA-1, it applies equally well to arbitrary algorithms. The key is nothing but a bunch of random or pseudo-random bytes, preferably the same length as the output of the selected hash algorithm.

Usage scenarios

The HMAC can be applied in a number of scenarios, for example:

- Sending out a password reset e-mail that is valid only for a certain time and can only be used once. The HMAC ”magic” allows for this without any server state.
- Verifying e-mail address in order to create or activate an account
- Authenticating form data that has been sent to the users’ web browser and then posted back.
- Authenticating data sent by external applications – typically any scenario where you provide a service that has the notion of an “API key”. In this case you share a secret key with each user of your service. The added benefit of this approach is that HMAC’s are computationally inexpensive and does not require much memory, thus very suitable for “Internet of Things” (IoT).

Code samples

This section contains a couple of code examples to illustrate the use of HMAC’s in Java and C#. The examples are simple and just illustrate the basic usage, but will give you a general idea on the language support. The goal of these examples is that they should produce the same result, as interoperability may be of interest for your particular use case.

Java code example

The following Java code example shows how to produce an HMAC using the standard Java security API functions:

```
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import java.util.Formatter;

public class Main {

  public static String toHexString(byte[] bytes) {
    Formatter formatter = new Formatter();
    for (byte b : bytes) {
      formatter.format("%02x", b);
    }
    return formatter.toString();
  }

  public static void main(String[] args) throws Exception {
    byte[] message = "Message to be processed".getBytes("UTF8");
    byte[] keybytes = "PoorKey".getBytes("UTF8");
    SecretKeySpec key = new SecretKeySpec(keybytes,"HmacSHA1");
    Mac hmac = Mac.getInstance("HmacSHA1");
    hmac.init(key);
    byte[] bytes = hmac.doFinal(message);
    System.out.println("HMAC: " + toHexString(bytes));
  }
}
```

The output produced when run is:

```
HMAC: 14510b1f7ec15554fbadcad358dfc2230eabfdc3
```

C# code example

The following C# code example shows how to produce an HMAC using the standard .Net security API functions:

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace HmacSample
{
  class MainClass
  {
    public static string ByteArrayToString(byte[] ba)
    {
      StringBuilder hex = new StringBuilder(ba.Length * 2);
      foreach (byte b in ba)
        hex.AppendFormat("{0:x2}", b);
      return hex.ToString();
    }

    public static void Main (string[] args)
    {
      string key ="PoorKey";
      string data= "Message to be processed";
```

```
        var hmac = new HMACSHA1(Encoding.UTF8.GetBytes(key));
        byte[] bytes = hmac.ComputeHash(Encoding.UTF8.GetBytes(data));
        Console.WriteLine("HMAC: " + ByteArrayToString(bytes));
      }
    }
}
```

The output produced when run is:

```
HMAC: 14510b1f7ec15554fbadcad358dfc2230eabfdc3
```

The standard algorithms are obviously implementing the same specification, since they both produce the same result.

Example applications

Password reset e-mail

Assume the following scenario:

- You are operating a site where users can log in with their e-mail address and a password of choice upon registration.
- You want to allow the users to select ”forgot password” and ensure that a reset password-link is sent to their e-mail address.
- The link must only be valid for one hour and may only be used once.
- You have created a secret key that is known only to you. Let’s call this K.

When the user requests a password reset-link:

1. Construct a string consisting of: The users e-mail address, the servers current time, the hash of the current user password. Let’s call this message M.
2. Calculate the HMAC of M, using K as the secret key.
3. Construct a URL containing the path to your password reset page, and as parameters, the user’s e-mail address, the current time and the HMAC produced in the previous step, e.g:

   ```
   https://www.example.com/forgotPassword?
   user=user%40example.com &time=20151205T131159Z
   &hmac=3902ed847ff28930b5f141abfa8b471681253673
   ```

   Note that you should make sure to URL-encode the parameters, as seen in the “user”-argument, where the “@”-sign is encoded as “%40”.

4. Include the computed URL in an e-mail message that you send to the end user.

When the user receives the e-mail, he or she follows the URL and gets to your forgotPassword page. In order to be allowed to set a new password, the following verification is performed on the server:

1. Find the GET-parameters for e-mail, current time and HMAC.

2. Lookup the hash of the user's password from the password database and the server's secret key, K.
3. Concatenate the e-mail, current time and user's password hash, producing M'.
4. Calculate HMAC of M' and K
5. If the calculated HMAC is equal to the HMAC supplied as a GET parameter, you can be sure that the timestamp, and e-mail parameters are not fiddled with. You also know that the link has not been used before, since the current password hash is unchanged.
6. If the timestamp is older than one hour, inform the user that the link is no longer valid. If the passed HMAC is different from the calculated HMAC, inform the user that the link can only be used once. If none of above apply, the link is valid and you should present the user with an input field that allows them to enter a new password.

Note: Rather than using a generic server key, the user's stored password hash should be possible to use as the key for the HMAC. I cannot think of any security flaws with this method (given that the password hashes are derived in a sound manner), but as always when dealing with security, the path is beset with pitfalls on all sides, so don't take my word for it.

Compare the above with the perhaps most intuitive (but less elegant) approach of solving the problem:

- Generate a long, random string and store it in the database together with a time stamp of when it was created.
- Append this string to a password reset-URL that you send to the user's e-mail.
- Wait for the URL to be called and check that the time is not over-due. If still valid, let the user change password.
- Introduce a timer that periodically deletes unused password reset tickets from the database.

Clearly the above approach involves both new database information as well as timer tasks for cleanup, which is quite unnecessary when the same result can be achieved by just using an HMAC.

Account activation e-mail

The scenario for e-mail activation is very similar to the password reset scenario. Assuming a new user has entererd their e-mail address and a password for the pending account, an HMAC is created containing timestamp and the e-mail address. These are all baked together in a URL that is attached to an e-mail, which is sent to the address supplied by the user.

The verification process pretty much follows the steps above, perhaps with a timeout of a day or so, and effectively making sure that the user is in possession of the supplied e-mail address.

Authenticated IoT service communication

In this scenario, picture yourself providing an online service for Internet of Things (IoT), where embedded devices are able to send all kinds of information (typically

sensor information, such as temperature readings or GPS coordinates) to be stored in a cloud storage service you provide. The connected devices have a limited amount of memory and computational power. They are able to connect to the Internet over Wi-Fi, but are usually not able to use SSL, or handle the memory-consuming big integer-calculations needed for public key computations. You still want to be able to authenticate valid users of your service and make sure no one is able to impersonate others or alter information in transit between the device and your service endpoint.

In order to provide the service, you start with sharing a randomly generated secret key (kind of an API-key) that you store locally together with other user information. The users of your service (i.e. the IoT device) then reports their data to your service by following the steps below:

1. Extract the data to report, say sensor readings for temperature and humidity
2. Compute an HMAC of the sensor readings and user-ID.
3. Construct a URL on the form:

   ```
   http://sensorservice.example.com?user=alice
   &arg0=25&arg1=65&hmac=b5f141abfa8b471681253673
   ```

   Where user is the unique user name, arg0 is the temperature reading and arg1 is the humidity reading. The HMAC is computed by the value of *user + arg0 + arg1* (plus sign meaning string concatenation) together with the shared secret key.

4. The HTTP request is made, and if everything works as expected, an HTTP status 200 OK response is returned.

As the provider of the service, you would act something like the following in order to validate the request:

1. You receive the HTTP request and look for the value of the request parameter ”user”, look up the shared secret key for that user (alice in this case).
2. Loop through the arguments (including the user name, but not the HMAC) and concatenate the values.
3. Compute the HMAC from the resulting string in the previous step using the secret key. Compare the result with the caller-supplied “hmac”-argument.
4. If the HMAC’s are the same, the user name is valid and the contents of the arguments cannot have been altered in transit – thus store them in the database and return HTTP status 200 OK.

Note: Make sure that you agree with the users of the service on the order of the concatenation of the arguments, as different order will produce different HMAC’s.

Compare the suggested approach with the traditional API-key use case, where the key is often just passed as an argument, allowing anyone that intercepts calls to start impersonate your users.

Conclusion

Hashed message authentication codes (HMAC's) are very useful when you wish to send data out to untrusted destinations and wish to be able to verify that whenever you get the data back, the information has not been altered, as the case with a password reset e-mail for example.

Many system interactions involve communication with just two parties, and if it is feasible to share a common secret, it is straightforward to use an HMAC to verify both the integrity of the information passed as well as authenticating the other party, as the case with the IoT cloud service for example.

About the author

Stellan Söderström is a senior software architect with strengths in solution-, security- and integration architectures. He is a certified Software Architecture Professional through Carnegie Mellon Software Engineering Institute and holds a Master's degree in Computer Science and Engineering from Luleå Techical University. Most of his professional work has centered around large, complex and transactionally intense systems with stringent requirements on availability, performance, security and maintainability.

Stellan can be reached at stellan.soderstrom@gmail.com
https://www.linkedin.com/in/stellan
https://steelmon.wordpress.com/