OCRA: OATH Challenge-Response Algorithms

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

Copyright Notice

Copyright (c) 2009 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents in effect on the date of publication of this document (http://trustee.ietf.org/license-info).

Please review these documents carefully, as they describe your rights and restrictions with respect to this document.
Abstract

This document describes the OATH algorithm for challenge-response authentication and signatures. This algorithm is based on the HOTP algorithm [RFC4226] that was introduced by OATH (initiative for Open AuTHentication) [OATH] and submitted as an individual draft to the IETF in 2006.

Table of Contents

1. Introduction.................................................. 3
2. Requirements Terminology..................................... 3
3. Algorithm Requirements........................................ 3
4. OCRA Background............................................... 4
4.1 HOTP Algorithm............................................... 4
5. Definition of OCRA............................................. 5
5.1 DataInput Parameters.......................................... 5
5.2 CryptoFunction............................................... 6
6. The OCRASuite................................................ 7
7. Algorithm Modes for Authentication.......................... 9
7.1 One way Challenge-Response................................ 9
7.2 Mutual Challenge-Response.................................. 10
8. Algorithm Modes for Signature............................... 12
8.1 Plain Signature............................................. 12
8.2 Signature with Server Authentication...................... 13
9. Security Considerations....................................... 14
9.1 Security Analysis of the OCRA algorithm................... 14
9.2 Implementation Considerations.............................. 15
10. IANA Considerations........................................... 16
11. Conclusion................................................... 16
12. Acknowledgements............................................ 17
13. References.................................................... 17
13.1 Normative.................................................. 17
13.2 Informative................................................ 17
Appendix A: Source Code......................................... 18
14. Authors’ Addresses........................................... 25
1. Introduction

OATH has identified several use cases and scenarios that require an asynchronous variant to accommodate users who do not want to maintain a synchronized authentication system. A commonly accepted method for this is to use a challenge-response scheme.

Such challenge response mode of authentication is widely adopted in the industry. Several vendors already offer software applications and hardware devices implementing challenge-response - but each of those uses vendor-specific proprietary algorithms. For the benefits of users there is a need for a standardized challenge-response algorithm which allows multi-sourcing of token purchases and validation systems to facilitate the democratization of strong authentication.

Additionally, this specification describes the means to create symmetric key based digital signatures. Such signatures are variants of challenge-response mode where the data to be signed becomes the challenge.

2. Requirements Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

3. Algorithm Requirements

This section presents the main requirements that drove this algorithm design. A lot of emphasis was placed on flexibility and usability, under the constraints and specificity of the HOTP algorithm and hardware token capabilities.

R1 - The algorithm MUST support asynchronous challenge-response based authentication.

R2 - The algorithm MUST be capable of supporting symmetric key based digital signatures. Essentially this is a variation of challenge-response where the challenge is derived from the data that need to be signed.

R3 - The algorithm MUST be capable of supporting server-authentication, whereby the user can verify that he/she is talking to a trusted server.

R4 - The algorithm SHOULD use HOTP [RFC4226] as a key building block.
R5 - The length and format of the input challenge SHOULD be configurable.

R6 - The output length and format of the generated response SHOULD be configurable.

R7 - The challenge MAY be generated with integrity checking (e.g., parity bits). This will allow tokens with pin pads to perform simple error checking when the user enters the challenge value into a token.

R8 - There MUST be a unique secret (key) for each token/soft token that is shared between the token and the authentication server. The keys MUST be randomly generated or derived using a key derivation algorithm.

R9 - The algorithm MAY enable additional data attributes such as a timestamp or session information to be included in the computation. These data inputs MAY be used individually or all together.

4. OCRA Background

OATH introduced the HOTP algorithm as a first open, freely available building block towards strengthening authentication for end-users in a variety of applications. One-time passwords are very efficient at solving specific security issues thanks to the dynamic nature of OTP computations.

After carefully analyzing different use cases, OATH came to the conclusion that providing for extensions to the HOTP algorithms was important. A very natural extension is to introduce a challenge mode for computing HOTP values based on random questions. Equally beneficial is being able to perform mutual authentication between two parties, or short-signature computation for authenticating transaction to improve the security of e-commerce applications.

4.1 HOTP Algorithm

The HOTP algorithm, as defined in [RFC4226] is based on an increasing counter value and a static symmetric key known only to the prover and verifier parties.

As a reminder:

\[
\text{HOTP}(K, C) = \text{Truncate}(\text{HMAC-SHA1}(K, C))
\]

Where \(\text{Truncate}\) represents the function that converts an \(\text{HMAC-SHA-1}\) value into an HOTP value.
We refer the reader to [RFC4226] for the full description and further details on the rationale and security analysis of HOTP.

The present draft describes the different variants based on similar constructions as HOTP.

5. Definition of OCRA

OCRA is a generalization of HOTP with variable data inputs not solely based on an incremented counter and secret key values.

The definition of OCRA requires a cryptographic function, a key K and a set of DataInput parameters. This section first formally introduces the OCRA algorithm and then introduces the definitions and default values recommended for all parameters.

In a nutshell,

\[
\text{OCRA} = \text{CryptoFunction}(K, \text{DataInput})
\]

Where:

- K: a shared secret key known to both parties;
- DataInput: a structure that contains the concatenation of the various input data values defined in details in section 5.1;
- CryptoFunction: this is the function performing the OCRA computation from the secret key K and the DataInput material; CryptoFunction is described in details in section 5.2.

5.1 DataInput Parameters

This structure is the concatenation over byte array of the OCRASuite value as defined in section 6 with the different parameters used in the computation, save for the secret key K.

\[
\text{DataInput} = \{\text{OCRASuite} | 00 | C | Q | P | S | T\}
\]

where:

- OCRASuite is a value representing the suite of operations to compute an OCRA response;
- 00 is a byte value used as a separator;
- C is an unsigned 8-byte counter value processed high-order bit first, and MUST be synchronized between all parties; It loops around from "(Hex)0" to "(Hex)FFFFFFFFFFFF" and then starts over at "(Hex)0";
- Q, mandatory, is a 128-byte list of (concatenated) challenge question(s) generated by the parties; if Q is less than 128 bytes, then it should be padded with zeroes to the right;
- P is a hash (SHA1, SHA256 and SHA512 are supported) value of PIN/password that is known to all parties during the execution of OCRA.
of the algorithm; the length of P will depend on the hash function that is used;
. S is an UTF-8 encoded string of length up to 512 bytes that contains information about the current session; the length of S is defined in the OCRASuite string;
. T is an 8-byte unsigned integer in big endian (i.e. network byte order) representing the number of time-steps (seconds, minutes, hours or days depending on the specified granularity) since midnight UTC of January 1, 1970. More specifically, if the OCRA computation includes a timestamp T, you SHOULD first convert your current local time to UTC time (text form); you can then derive the UTC time in the proper format (i.e. seconds, minutes, hours or days elapsed from Epoch time); the size of the time-step is defined in the OCRASuite string.

When computing a response, the concatenation order is always the following:

```
C | OTHER-PARTY-GENERATED-CHALLENGE-QUESTION | YOUR-GENERATED-CHALLENGE-QUESTION | P | S | T
```

If a value is empty (i.e. a certain input is not used in the computation) then the value is simply not represented in the string.

The counter on the token or client MUST be incremented every time a new computation is requested by the user. The server’s counter value MUST only be incremented after a successful OCRA authentication.

5.2 CryptoFunction

The default CryptoFunction is HOTP-SHA1-6, i.e. the default mode of computation for OCRA is HOTP with the default 6-digit dynamic truncation and a combination of DataInput values as the message to compute the HMAC-SHA1 digest.

As indicated in section 5.1, we denote t as the length in digits of the truncation output. For instance, if t = 6, then the output of the truncation is a 6-digit value.

We define the HOTP family of functions as an extension to HOTP:
- HOTP-H-t: these are the different possible truncated versions of HOTP, using the dynamic truncation method for extracting an HOTP value from the HMAC output;
- We will denote HOTP-H-t as the realization of an HOTP function that uses an HMAC function with the hash function H, and the
dynamic truncation as described in [RFC 4226] to extract a t-digit value;
- t=0 means that no truncation is performed and the full HMAC value is used for authentication purpose.

We list the following preferred modes of computation, where * denotes the default CryptoFunction:
- HOTP-SHA1-4: HOTP with SHA-1 as the hash function for HMAC and a dynamic truncation to a 4-digit value; this mode is not recommended in the general case but can be useful when a very short authentication code is needed by an application;
- *HOTP-SHA1-6: HOTP with SHA-1 as the hash function for HMAC and a dynamic truncation to a 6-digit value;
- HOTP-SHA1-8: HOTP with SHA-1 as the hash function for HMAC and a dynamic truncation to an 8-digit value;
- HOTP-SHA256-6: HOTP with SHA-256 as the hash function for HMAC and a dynamic truncation to a 6-digit value;
- HOTP-SHA512-6: HOTP with SHA-512 as the hash function for HMAC and a dynamic truncation to a 6-digit value;

This table summarizes all possible values for the CryptoFunction:

<table>
<thead>
<tr>
<th>Name</th>
<th>HMAC Function Used</th>
<th>Size of Truncation (t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOTP-SHA1-t</td>
<td>HMAC-SHA1</td>
<td>0 (no truncation), 4-10</td>
</tr>
<tr>
<td>HOTP-SHA256-t</td>
<td>HMAC-SHA256</td>
<td>0 (no truncation), 4-10</td>
</tr>
<tr>
<td>HOTP-SHA512-t</td>
<td>HMAC-SHA512</td>
<td>0 (no truncation), 4-10</td>
</tr>
</tbody>
</table>

6. The OCRASuite

An OCRASuite value is a text string that captures one mode of operation for the OCRA algorithm, completely specifying the various options for that computation. An OCRASuite value is represented as follows:

Algorithm:CryptoFunction:DataInput

The client and server need to agree on one or two values of OCRASuite. These values may be agreed at time of token provisioning or for more sophisticated client-server interactions these values may be negotiated for every transaction.

Note that for Mutual Challenge-Response or Signature with Server Authentication modes, the client and server will need to agree on two values of OCRASuite - one for server computation and another for client computation.
Algorithm
---------

Description: Indicates the version of OCRA algorithm.
Values: OCRA-v where v represents the version number (e.g. 1, 2 etc.). This document specifies version 1 of the OCRA algorithm.

CryptoFunction
--------------

Description: Indicates the function used to compute OCRA values
Values: Permitted values are described in section 5.2

DataInput
---------

Description: This component of the OCRASuite string captures the list of valid inputs for that computation; [] indicates a value is optional:
[C] | QFxx | [PH | TG] : Plain Signature computation

Each input that is used for the computation is represented by a single letter (except Q) and they are separated by a hyphen.

The input for challenge is further qualified by the formats supported by the client for challenge question(s). Supported values can be:

<table>
<thead>
<tr>
<th>Format (F)</th>
<th>Up To Length (xx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A (alphanumeric)</td>
<td>04-64</td>
</tr>
<tr>
<td>N (numeric)</td>
<td>04-64</td>
</tr>
<tr>
<td>H (hexadecimal)</td>
<td>04-64</td>
</tr>
</tbody>
</table>

The default challenge format is N08, numeric and up to 8 digits.

The input for P is further qualified by the hash function used for the PIN/password. Supported values for hash function can be:

- Hash function (H) - SHA1, SHA256, SHA512.

The default hash function for P is SHA1.

The input for S is further qualified by the length of the session data in bytes. The client and server could agree to any length but the typical values are:

- Length (nnn) - 064, 128, 256 and 512.

The default length is 064 bytes.
The input for timestamps is further qualified by $G$, size of the time-step. $G$ can be specified in number of seconds, minutes or hours:

<table>
<thead>
<tr>
<th>Time-step size (G)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1-59]S</td>
<td>number of seconds, e.g. 20S</td>
</tr>
<tr>
<td>[1-59]M</td>
<td>number of minutes, e.g. 5M</td>
</tr>
<tr>
<td>[0-48]H</td>
<td>number of hours, e.g. 24H</td>
</tr>
</tbody>
</table>

Default value for $G$ is 1M, i.e. time step size is one minute and the $T$ represents the number of minutes since Epoch time.

Here are some examples of OCRASuite strings:

- OCRA-1:HOTP-SHA512-8:C-QN08-PSHA1 means version 1 of the OCRA algorithm with HMAC-SHA512 function, truncated to an 8-digit value, using the counter, a random challenge and a SHA1 digest of the PIN/Password as parameters. It also indicates that the client supports only numeric challenge upto 8 digits in length;

- OCRA-1:HOTP-SHA256-6:QA10-T1M means version 1 of the OCRA algorithm with HMAC-SHA256 function, truncated to a 6-digit value, using a random alphanumeric challenge upto 10 characters in length and a timestamp in number of minutes since Epoch time;

- OCRA-1:HOTP-SHA1-4:QH8-S512 means version 1 of the OCRA algorithm with HMAC-SHA1 function, truncated to a 4-digit value, using a random hexadecimal challenge upto 8 nibbles and a session value of 512 bytes.

7. Algorithm Modes for Authentication

This section describes the typical modes in which the above defined computation can be used for authentication.

7.1 One way Challenge-Response

A challenge/response is a security mechanism in which the verifier presents a question (challenge) to the prover who must provide a valid answer (response) to be authenticated.

To use this algorithm for a one-way challenge-response, the verifier will communicate a challenge value (typically randomly generated) to the prover. The prover will use the challenge in the computation as described above. The prover then communicates the response to the verifier to authenticate.

Therefore in this mode, the typical data inputs will be:

- $C$ - Counter, optional.
- $Q$ - Challenge question, mandatory, supplied by the verifier.
P - Hashed version of PIN/password, optional.
S - Session information, optional
T - Timestamp, optional.

The diagram below shows the message exchange between the client (prover) and the server (verifier) to complete a one-way challenge-response authentication.

It is assumed that the client and server have a pre-shared key $K$ that is used for the computation.

<table>
<thead>
<tr>
<th>CLIENT (PROVER)</th>
<th>SERVER (VERIFIER)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verifier sends challenge to prover</td>
<td>Challenge = $Q$</td>
</tr>
<tr>
<td>&lt;------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Prover Computes Response</td>
<td>Prover sends Response $R = OCRA(K, {[C] \mid Q \mid [P \mid S \mid T]})$</td>
</tr>
<tr>
<td>Prover sends Response $= R$</td>
<td>&lt;------------------------------------------</td>
</tr>
<tr>
<td>Verifier Validates Response</td>
<td>If Response is valid, Server sends OK</td>
</tr>
<tr>
<td>If Response is not, Server sends NOK</td>
<td>&lt;-----------------------------</td>
</tr>
</tbody>
</table>

7.2 Mutual Challenge-Response

Mutual challenge-response is a variation of one-way challenge-response where both the client and server mutually authenticate each other.

To use this algorithm, the client will first send a random client-challenge to the server. The server computes the server-response and sends it to the client along with a server-challenge.

The client will first verify the server-response to be assured that it is talking to a valid server. It will then compute the client-response and send it to the server to authenticate. The server verifies the client-response to complete the two-way authentication process.

In this mode there are two computations: client-response and server-response. There are two separate challenge questions, generated by both parties. We denote these challenge questions $Q_1$ and $Q_2$. 
Typical data inputs for server-response computation will be:
C  - Counter, optional.
QC - Challenge question, mandatory, supplied by the client.
QS - Challenge question, mandatory, supplied by the server.
S  - Session information, optional.
T  - Timestamp, optional.

Typical data inputs for client-response computation will be:
C  - Counter, optional.
QS - Challenge question, mandatory, supplied by the server.
QC - Challenge question, mandatory, supplied by the client.
P  - Hashed version of PIN/password, optional.
S  - Session information, optional.
T  - Timestamp, optional.

The following picture shows the messages that are exchanged between the client and the server to complete a two-way mutual challenge-response authentication.

It is assumed that the client and server have a pre-shared key $K$ (or pair of keys if using dual-key mode of computation) that is used for the computation.

```
CLIENT                                SERVER

1. Client sends client-challenge       
   QC = Client-challenge               
--------------------------------------->

2. Server computes server-response    
   and sends server-challenge          
   RS = OCRA(K, [C] | QC | QS | [S | T]) 
   QS = Server-challenge               
   Response = RS, QS                   
<-------------------------------------

3. Client verifies server-response     
   and computes client-response        
   OCRA(K, [C] | QC | QS | [S | T]) != RS -> STOP 
   RC = OCRA(K, [C] | QS | QC | [P | S | T]) 
   Response = RC                       
--------------------------------------->

4. Server verifies client-response     
   OCRA(K, [C] | QS | QC | [P|S|T]) != RC -> STOP 
   Response = OK                        
<-------------------------------------
```
8. Algorithm Modes for Signature

In this section we describe the typical modes in which the above defined computation can be used for digital signatures.

8.1 Plain Signature

To use this algorithm in plain signature mode, the server will communicate a signature-challenge value to the client (signer). The signature-challenge is either the data to be signed or derived from the data to be signed using a hash function, for example.

The client will use the signature-challenge in the computation as described above. The client then communicates the signature value (response) to the server to authenticate.

Therefore in this mode, the data inputs will be:

- C - Counter, optional.
- QS - Signature-challenge, mandatory, supplied by the server.
- P - Hashed version of PIN/password, optional.
- T - Timestamp, optional.

The picture below shows the messages that are exchanged between the client (prover) and the server (verifier) to complete a plain signature operation.

It is assumed that the client and server have a pre-shared key K that is used for the computation.
8.2 Signature with Server Authentication

This mode is a variation of the plain signature mode where the client can first authenticates the server before generating a digital signature.

To use this algorithm, the client will first send a random client-challenge to the server. The server computes the server-response and sends it to the client along with a signature-challenge.

The client will first verify the server-response to authenticate that it is talking to a valid server. It will then compute the signature and send it to the server.

In this mode there are two computations: client-signature and server-response.

Typical data inputs for server-response computation will be:

- C - Counter, optional.
- QC - Challenge question, mandatory, supplied by the client.
- QS - Signature-challenge, mandatory, supplied by the server.
- T - Timestamp, optional.

Typical data inputs for client-signature computation will be:

- C - Counter, optional.
- QC - Challenge question, mandatory, supplied by the client.
- QS - Signature-challenge, mandatory, supplied by the server.
- P - Hashed version of PIN/password, optional.
- T - Timestamp, optional.

The diagram below shows the messages that are exchanged between the client and the server to complete a signature with server authentication transaction.

It is assumed that the client and server have a pre-shared key K (or pair of keys if using dual-key mode of computation) that is used for the computation.
1. Client sends client-challenge
   QC = Client-challenge

2. Server computes server-response
   and sends signature-challenge
   RS = OCRA(K, [C] | QC | QS | [T])
   QS = signature-challenge
   Response = RS, QS

3. Client verifies server-response
   and computes signature
   OCRA(K, [C] | QC | QS | [T]) != RS -> STOP
   SIGN = OCRA(K, [C] | QS | QC | [P | T])
   Response = SIGN

4. Server verifies Signature
   OCRA(K, [C] | QS | QC | [P | T]) != SIGN -> STOP
   Response = OK

9. Security Considerations

Any algorithm is only as secure as the application and the authentication protocols that implement it. Therefore, this section discusses the critical security requirements that our choice of algorithm imposes on the authentication protocol and validation software.

9.1 Security Analysis of the OCRA algorithm

The security and strength of this algorithm depends on the properties of the underlying building block HOTP, which is a construction based on HMAC [RFC2104] using SHA-1 as the hash function.

The conclusion of the security analysis detailed in [RFC4226] is that, for all practical purposes, the outputs of the dynamic truncation on distinct counter inputs are uniformly and independently distributed strings.

The analysis demonstrates that the best possible attack against the HOTP function is the brute force attack.
9.2 Implementation Considerations

IC1 - In the authentication mode, the client MUST support two-factor authentication, i.e., the communication and verification of something you know (secret code such as a Password, Pass phrase, PIN code, etc.) and something you have (token). The secret code is known only to the user and usually entered with the Response value for authentication purpose (two-factor authentication). Alternatively, instead of sending something you know to the server, the client may use a hash of the Password or PIN code in the computation itself, thus implicitly enabling two-factor authentication.

IC2 - Keys should be of the length of the CryptoFunction output to facilitate interoperability.

IC3 - Keys SHOULD be chosen at random or using a cryptographically strong pseudo-random generator properly seeded with a random value. We RECOMMEND following the recommendations in [RFC1750] for all pseudo-random and random generations. The pseudo-random numbers used for generating the keys SHOULD successfully pass the randomness test specified in [CN].

IC4 - Challenge questions SHOULD be 20-byte values and MUST be at least t-byte values where t stands for the digit-length of the OCRA truncation output.

IC5 - On the client side, the keys SHOULD be embedded in a tamper resistant device or securely implemented in a software application. Additionally, by embedding the keys in a hardware device, you also have the advantage of improving the flexibility (mobility) of the authentication system.

IC6 - We RECOMMEND following the recommendations in [RFC1750] for all pseudo-random and random challenge generations.

IC7 - All the communications SHOULD take place over a secure channel e.g. SSL/TLS, IPsec connections.

IC8 - The OCRA algorithm when used in mutual authentication mode or in signature with server authentication mode MAY use dual key mode - i.e. there are two keys that are shared between the client and the server. One shared key is used to generate the server response on the server side and to verify it on the client side. The other key is used to create the response or signature on the client side and to verify it on the server side.
IC9 - We recommend that implementations MAY use the session information, S as an additional input in the computation. For example, S could be the session identifier from the TLS session. This will enable you to counter certain types of man-in-the-middle attacks. However, this will introduce the additional dependency that first of all the prover needs to have access to the session identifier to compute the response and the verifier will need access to the session identifier to verify the response.

IC10 - In the signature mode, whenever the counter or time (defined as optional elements) are not used in the computation, there might be a risk of replay attack and the implementers should carefully consider this issue in the light of their specific application requirements and security guidelines. The server SHOULD also provide whenever possible a mean for the client (if able) to verify the validity of the signature challenge.

IC11 - We also RECOMMEND storing the keys securely in the validation system, and more specifically encrypting them using tamper-resistant hardware encryption and exposing them only when required: for example, the key is decrypted when needed to verify an OCRA response, and re-encrypted immediately to limit exposure in the RAM for a short period of time. The key store MUST be in a secure area, to avoid as much as possible direct attack on the validation system and secrets database. Particularly, access to the key material should be limited to programs and processes required by the validation system only.

10. IANA Considerations

This document has no actions for IANA.

11. Conclusion

This draft introduced several variants of HOTP for challenge-response based authentication and short signature-like computations.

The OCRASuite provides for an easy integration and support of different flavors within an authentication and validation system.

Finally, OCRA should enable mutual authentication both in connected and off-line modes, with the support of different response sizes and mode of operations.
12. Acknowledgements

We would like to thank Jeff Burstein, Shuh Chang, Oanh Hoang, Philip Hoyer, Jon Martinsson, Frederik Mennes, Mingliang Pei, Jonathan Tuliani, Stu Vaeth, Enrique Rodriguez and Robert Zuccherato for their comments and suggestions to improve this draft document.

13. References

13.1 Normative


13.2 Informative


[OATH] Initiative for Open AuTHenticaion
http://www.openauthentication.org

import java.lang.reflect.UndeclaredThrowableException;
import java.security.GeneralSecurityException;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import java.math.BigInteger;

/**
 * This is an example implementation of the OATH OCRA algorithm.
 * Visit www.openauthentication.org for more information.
 *
 * @author Johan Rydell, PortWise
 */

public class OCRA {

    private OCRA() {} 

    /**
     * This method uses the JCE to provide the crypto
     * algorithm.
     * HMAC computes a Hashed Message Authentication Code with the
     * crypto hash algorithm as a parameter.
     * @param crypto the crypto algorithm
     * @param keyBytes the bytes to use for the HMAC key
     * @param text the message or text to be authenticated.
     */
    public static byte[] hmac_sha1(String crypto,
                       byte[] keyBytes,
                       byte[] text){

        try {
            Mac hmac;
            hmac = Mac.getInstance(crypto);
            SecretKeySpec macKey =
                    new SecretKeySpec(keyBytes, "RAW");
            hmac.init(macKey);
            return hmac.doFinal(text);
        }
        catch (GeneralSecurityException gse) {
            throw new UndeclaredThrowableException(gse);
        }
    }

    private static final int[] DIGITS_POWER
    // 0 1 2 3 4 5 6 7 8
    = {1,10,100,1000,10000,100000,1000000,10000000,100000000};

    /**
     * (HmacSHA1, HmacSHA256, HmacSHA512)
     */
* This method generates an OCRA HOTP value for the given
* set of parameters.
* 
* @param ocraSuite  the OCRA Suite
* @param key         the shared secret, HEX encoded
* @param counter    the counter that changes on a per use basis,
*                   HEX encoded
* @param question   the challenge question
* @param password   a password that can be used
* @param sessionInformation  Static information that identifies
*                            the current session
* @param timeStamp   a value that reflects a time
* 
* @return A numeric String in base 10 that includes
*        (truncationDigits) digits
* /
static public String generateOCRA(String ocraSuite,
String key,
String counter,
String question,
String password,
String sessionInformation,
String timeStamp){

    int codeDigits = 0;
    String crypto = "";
    String result = null;
    int ocraSuiteLength = ocraSuite.length();
    int counterLength = 0;
    int questionLength = 0;
    int passwordLength = 0;
    int sessionInformationLength = 0;
    int timeStampLength = 0;

    if(ocraSuite.toLowerCase().indexOf("sha1") > 1)
        crypto = "HmacSHA1";
    if(ocraSuite.toLowerCase().indexOf("sha256") > 1)
        crypto = "HmacSHA256";
    if(ocraSuite.toLowerCase().indexOf("sha512") > 1)
        crypto = "HmacSHA512";

    // How many digits should we return
    String oS = ocraSuite.substring(ocraSuite.indexOf(":"),
        ocraSuite.indexOf(":" , ocraSuite.indexOf(":" ) + 1));
    codeDigits = Integer.decode(oS.substring(oS.lastIndexOf("-")+1,
        oS.length()));

    // The size of the byte array message to be encrypted
    // Counter
    if(ocraSuite.toLowerCase().indexOf(":c") > 1) {
        counterLength=8;
    }
// Question
if((ocraSuite.toLowerCase().indexOf(":q") > 1) ||
  (ocraSuite.toLowerCase().indexOf("-q") > 1)) {
    questionLength=128;
}

// Password - sha1 supported
if((ocraSuite.toLowerCase().indexOf(":psha1") > 1) ||
  (ocraSuite.toLowerCase().indexOf("-psha1") > 1)) {
    passwordLength=20;
}

// sessionInformation
if((ocraSuite.toLowerCase().indexOf(":s") > 1) ||
  (ocraSuite.toLowerCase().indexOf("-s",
    ocraSuite.indexOf(":", ocraSuite.indexOf(":" + 1)) > 1)) {
    sessionInformationLength=64;
}

// TimeStamp
if((ocraSuite.toLowerCase().indexOf(":t") > 1) ||
  (ocraSuite.toLowerCase().indexOf("-t") > 1)) {
    timeStampLength=8;
}

// Remember to add "1" for the "00" byte delimiter
byte[] msg = new byte[ocraSuiteLength +
  counterLength +
  questionLength +
  passwordLength +
  sessionInformationLength +
  timeStampLength +
  1];

// Put the bytes of "ocraSuite" parameters into the message
byte[] bArray = ocraSuite.getBytes();
for(int i = 0; i < bArray.length; i++) {
  msg[i] = bArray[i];
}

// Put the bytes of "Counter" to the message
// Input is HEX encoded
if(counter.length() > 0) {
  bArray = new BigInteger(counter,16).toByteArray();
  if(bArray.length == 9) {
    // First byte is the "sign" byte
    for (int i = 0; i < 8 && i < bArray.length ; i++) {
      msg[i + 8 - bArray.length + ocraSuiteLength + 1] =
        bArray[i+1];
    }
  }
  else {
  
  
  }
for (int i = 0; i < 8 && i < bArray.length; i++) {
    msg[i + 8 - bArray.length + ocraSuiteLength + 1] = bArray[i];
}

// Put the bytes of "question" to the message
// Input is text encoded
if(question.length() > 0) {
    bArray = question.getBytes();
    for (int i = 0; i < 128 && i < bArray.length; i++) {
        msg[i + ocraSuiteLength + 1 + counterLength] = bArray[i];
    }
}

// Put the bytes of "password" to the message
// Input is HEX encoded
if(password.length() > 0) {
    bArray = new BigInteger(password, 16).toByteArray();
    if(bArray.length == 21) {
        // First byte is the "sign" byte
        for (int i = 0; i < 20 && i < bArray.length; i++) {
            msg[i + ocraSuiteLength + 1 + counterLength + questionLength] = bArray[i+1];
        }
    } else {
        for (int i = 0; i < 20 && i < bArray.length; i++) {
            msg[i + ocraSuiteLength + 1 + counterLength + questionLength] = bArray[i];
        }
    }
}

// Put the bytes of "sessionInformation" to the message
// Input is text encoded
if(sessionInformation.length() > 0) {
    bArray = sessionInformation.getBytes();
    for (int i = 0; i < 128 && i < bArray.length; i++) {
        msg[i + ocraSuiteLength + 1 + counterLength + questionLength + passwordLength] = bArray[i];
    }
}

// Put the bytes of "time" to the message
// Input is text value of minutes
if(timeStamp.length() > 0) {
    bArray = new BigInteger(timeStamp, 16).toByteArray();
    if(bArray.length == 9) {
// First byte is the "sign" byte
for (int i = 0; i < 8 && i < bArray.length ; i++) {
    msg[i + 8 - bArray.length + ocraSuiteLength + 1 +
        counterLength + questionLength +
        passwordLength + sessionInformationLength] =
        bArray[i+1];
}

else {
    for (int i = 0; i < 8 && i < bArray.length ; i++) {
        msg[i + 8 - bArray.length + ocraSuiteLength + 1 +
            counterLength + questionLength +
            passwordLength + sessionInformationLength] =
            bArray[i];
    }
}

byte[] hash;
byte[] bArray = new BigInteger(key,16).toByteArray();
if(bArray[0] == 0){
    byte[] b = new byte[bArray.length - 1];
    for(int i = 0 ; i < b.length; i++)
        b[i]=bArray[i+1];
    hash = hmac_sha1(crypto, b, msg);
}
else{
    // compute hmac hash
    hash = hmac_sha1(crypto, bArray, msg);
}

// put selected bytes into result int
int offset = hash[hash.length - 1] & 0xf;

int binary =
    ((hash[offset] & 0x7f) << 24) |
    ((hash[offset + 1] & 0xff) << 16) |
    ((hash[offset + 2] & 0xff) << 8) |
    (hash[offset + 3] & 0xff);

int otp = binary % DIGITS_POWER[codeDigits];

result = Integer.toString(otp);
while (result.length() < codeDigits) {
    result = "0" + result;
}
return result;
Appendix B: Test Vectors

Time of (Mar 25 2008, 12:06:30 GMT) is in
(millis): 120646790000 (min): 20107446 (HEX): 132d0b6
Time of (Mar 25 2008, 12:06:30 GMT) is the same as this localized
time: Tue Mar 25 05:06:30 PDT 2008

Standard 20Byte key: 3132333435363738393031323334353637383930
Standard 32Byte key: 3132333435363738393031323334353637383930
           3132333435363738393031323334353637383930
Standard 64Byte key: 3132333435363738393031323334353637383930
           3132333435363738393031323334353637383930
           3132333435363738393031323334353637383930
           3132333435363738393031323334353637383930

Plain challenge response
==================================

OCRA-1:HOTP-SHA1-6:QN08
==================================
Key: Standard 20Byte  Q: 00000000OCRA: 713673
Key: Standard 20Byte  Q: 11111111OCRA: 640542
Key: Standard 20Byte  Q: 22222222OCRA: 434144
Key: Standard 20Byte  Q: 33333333OCRA: 024883
Key: Standard 20Byte  Q: 44444444OCRA: 473006
Key: Standard 20Byte  Q: 55555555OCRA: 911781
Key: Standard 20Byte  Q: 66666666OCRA: 059218
Key: Standard 20Byte  Q: 77777777OCRA: 175339
Key: Standard 20Byte  Q: 88888888OCRA: 478461
Key: Standard 20Byte  Q: 99999999OCRA: 681743

OCRA-1:HOTP-SHA256-8:QN08-PSHA1
==================================
Key: Standard 32Byte  Q: 00000000PIN (1234): 7110eda4d09e062aa5e4a390b0a572ac0d2c0220 OCRA: 40675653
Key: Standard 32Byte  Q: 11111111PIN (1234): 7110eda4d09e062aa5e4a390b0a572ac0d2c0220 OCRA: 14928254
Key: Standard 32Byte  Q: 22222222PIN (1234): 7110eda4d09e062aa5e4a390b0a572ac0d2c0220 OCRA: 09120993
Key: Standard 32Byte  Q: 33333333PIN (1234): 7110eda4d09e062aa5e4a390b0a572ac0d2c0220 OCRA: 50886787
Key: Standard 32Byte  Q: 44444444PIN (1234): 7110eda4d09e062aa5e4a390b0a572ac0d2c0220 OCRA: 23934759
OCRA-1:HOTP-SHA512-8:C-QN08
===========================
Key: Standard 64Byte C: 00000 Q: 00000000OCRA: 81947120
Key: Standard 64Byte C: 00001 Q: 11111111OCRA: 46439675
Key: Standard 64Byte C: 00002 Q: 22222222OCRA: 44178142
Key: Standard 64Byte C: 00003 Q: 33333333OCRA: 33562866
Key: Standard 64Byte C: 00004 Q: 44444444OCRA: 99699620
Key: Standard 64Byte C: 00005 Q: 55555555OCRA: 73476531
Key: Standard 64Byte C: 00006 Q: 66666666OCRA: 44853974
Key: Standard 64Byte C: 00007 Q: 77777777OCRA: 99378156
Key: Standard 64Byte C: 00008 Q: 88888888OCRA: 87993791
Key: Standard 64Byte C: 00009 Q: 99999999OCRA: 56984649

OCRA-1:HOTP-SHA512-8:QN08-T1M
===========================
Key: Standard 64Byte Q: 00000000 T: 132d0b6 OCRA: 66401302
Key: Standard 64Byte Q: 11111111 T: 132d0b6 OCRA: 23050616
Key: Standard 64Byte Q: 22222222 T: 132d0b6 OCRA: 39524082
Key: Standard 64Byte Q: 33333333 T: 132d0b6 OCRA: 97622335
Key: Standard 64Byte Q: 44444444 T: 132d0b6 OCRA: 16392830

Mutual Challenge Response
=========================

OCRASuite (server computation) = OCRA-1:HOTP-SHA256-8:QA08
OCRASuite (client computation) = OCRA-1:HOTP-SHA256-8:QA08
==========================================================
(server)Key: Standard 32Byte Q: CLI22220SRV11110 OCRA: 28247970
(client)Key: Standard 32Byte Q: SRV11110CLI22220 OCRA: 15510767
(server)Key: Standard 32Byte Q: CLI22221SRV11111 OCRA: 34479341
(client)Key: Standard 32Byte Q: SRV11111CLI22221 OCRA: 59062808
(server)Key: Standard 32Byte Q: CLI22222SRV11112 OCRA: 06200517
(client)Key: Standard 32Byte Q: SRV11112CLI22222 OCRA: 31050785
(server)Key: Standard 32Byte Q: CLI22223SRV11113 OCRA: 44568292
(client)Key: Standard 32Byte Q: SRV11113CLI22223 OCRA: 67628600
(server)Key: Standard 32Byte Q: CLI22224SRV11114 OCRA: 69375821
(client)Key: Standard 32Byte Q: SRV11114CLI22224 OCRA: 27094909

OCRASuite (server computation) = OCRA-1:HOTP-SHA512-8:QA08
OCRASuite (client computation) = OCRA-1:HOTP-SHA512-8:QA08-PSHA1
============================================================
(server)Key: Standard 64Byte Q: CLI22220SRV11110 OCRA: 79496648
(client)Key: Standard 64Byte Q: SRV11110CLI22220
P: 7110eda4d09e062aa5e4a390b0a572ac0d2c0220 OCRA: 8806276
(server)Key: Standard 64Byte Q: CLI22221SRV11111 OCRA: 76831980
(client)Key: Standard 64Byte Q: SRV11111CLI22221
P: 7110eda4d09e062aa5e4a390b0a572ac0d2c0220 OCRA: 70020315
(server)Key: Standard 64Byte Q: CLI22222SRV11112 OCRA: 12250499
(client)Key: Standard 64Byte Q: SRV11112CLI22222

OATH-HOTP-VARIANTS Expires - January 2010 [Page 24]
Plain Signature
===============

OCRA-1:HOTP-SHA256-8:QA08
=========================
Key: Standard 32Byte  Q(Signature challenge): SIG10000
OCRA: 53095496
Key: Standard 32Byte  Q(Signature challenge): SIG11000
OCRA: 04110475
Key: Standard 32Byte  Q(Signature challenge): SIG12000
OCRA: 31331128
Key: Standard 32Byte  Q(Signature challenge): SIG13000
OCRA: 76028668
Key: Standard 32Byte  Q(Signature challenge): SIG14000
OCRA: 46554205

OCRA-1:HOTP-SHA512-8:QA10-T1M
=============================  
Key: Standard 64Byte  Q(Signature challenge): SIG1000000
T: 132d0b6  OCRA: 77537423
Key: Standard 64Byte  Q(Signature challenge): SIG1100000
T: 132d0b6  OCRA: 31970405
Key: Standard 64Byte  Q(Signature challenge): SIG1200000
T: 132d0b6  OCRA: 10235557
Key: Standard 64Byte  Q(Signature challenge): SIG1300000
T: 132d0b6  OCRA: 95213541
Key: Standard 64Byte  Q(Signature challenge): SIG1400000
T: 132d0b6  OCRA: 65360607

14. Authors’ Addresses

Primary point of contact (for sending comments and question):

David M’Raihi
VeriSign, Inc.
685 E. Middlefield Road          Phone: 1-650-426-3832
Mountain View, CA 94043 USA      Email: dmraihi@verisign.com
Other Authors’ contact information:

Johan Rydell  
Portwise, Inc.  
275 Hawthorne Ave, Suite 119  
Palo Alto, CA 94301 USA  
Phone: 1-650-515-3569  
Email: johan.rydell@portwise.com

David Naccache  
ENS, DI  
45 rue d’Ulm  
75005, Paris France  
Phone: +33 6 16 59 83 49  
Email: david.naccache@ens.fr

Salah Machani  
Diversinet Corp.  
2225 Sheppard Avenue East  
Suite 1801  
Toronto, Ontario M2J 5C2  
Phone: 1-416-756-2324 Ext. 321  
Email: smachani@diversinet.com

Siddharth Bajaj  
VeriSign, Inc.  
487 E. Middlefield Road  
Mountain View, CA 94043 USA  
Phone: 1-650-426-3458  
Email: sbajaj@verisign.com