Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: September 12, 2011

D. M'Raihi
Verisign, Inc.
J. Rydell
Portwise, Inc.
S. Bajaj
Symantec Corp.
S. Machani
Diversinet Corp.
D. Naccache
Ecole Normale Superieure

March 11, 2011

OCRA: OATH Challenge-Response Algorithms
draft-mraihi-mutual-oath-hotp-variants-14.txt

Abstract

This document describes an algorithm for challenge-response authentication developed by the "Initiative for Open AuTHentication" (OATH). The specified mechanisms leverage the HMAC-based One-Time Password algorithm (HOTP) [RFC4226] and offer one-way and mutual authentication, and digital signature capabilities.

Table of Contents

1. Introduction

OATH [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 short digital signatures. Such signatures are variants of challenge-response mode where the data to be signed becomes the challenge.

2. Notation and 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 [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 challenge-response based authentication.

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

```
HOTP(K,C) = Truncate(HMAC-SHA1(K,C))
```

Where Truncate represents the function that converts an 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,

```
                  OCRA = CryptoFunction(K, DataInput)
```

Where:

- o K: a shared secret key known to both parties
- o DataInput: a structure that contains the concatenation of the various input data values defined in details in section 5.1
- o 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 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.

DataInput = {OCRASuite | 00 | C | Q | P | S | T} where:

- o OCRASuite is a value representing the suite of operations to compute an OCRA response
- o 00 is a byte value used as a separator
- o 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}FFFFFFFFFFFFFFFF" and then starts over at "{Hex}0". Note that 'C' is optional for all OCRA modes described in this document.

- o 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
- o P is a hash (SHA1 [RFC3174], SHA256 and SHA512 [SHA2] are supported) value of PIN/password that is known to all parties during the execution of the algorithm; the length of P will depend on the hash function that is used
- o S is an UTF-8 [RFC2279] encoded string of length upto 512 bytes that contains information about the current session; the length of S is defined in the OCRASuite string
- o 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 specificatlly, if the OCRA computation includes a timestamp T, you should first convert your current local time to UTC time; 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.

We denote t as the length in decimal digits of the truncation output. For instance, if t = 6, then the output of the truncation is a 6-digit (decimal) value.

We define the HOTP family of functions as an extension to HOTP:

1. 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
2. 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 [RFC4226] to extract a t-digit value
3. 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:

Name	HMAC Function Used	Size of Truncation (t)
HOTP-SHA1-t	HMAC-SHA1	0 (no truncation), 4-10
HOTP-SHA256-t	HMAC-SHA256	0 (no truncation), 4-10
HOTP-SHA512-t	HMAC-SHA512	0 (no truncation), 4-10

Table 1: CryptoFunction 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 OCRASuite value is the concatenation of three sub-components that are described below. Some example OCRASuite strings are described in section 6.4.

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.

6.1. 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.

6.2. CryptoFunction

Description: Indicates the function used to compute OCRA values

Values: Permitted values are described in section 5.2

6.3. 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 | Snnn | TG] : Challenge-Response computation

[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:

Format (F)	Up to Length (xx)
A (alphanumeric)	04-64
N (numeric)	04-64
H (hexadecimal)	04-64

Table 2: Challenge Format Table

The default challenge format is N08, numeric and upto 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:

Time-step Size (G)	Examples
[1-59]S	number of seconds, e.g. 20S
[1-59]M	number of minutes, e.g. 5M
[0-48]H	number of hours, e.g. 24H

Table 3: Time-step Size 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.

6.4. OCRASuite Examples

Here are some examples of OCRASuite strings:

- o "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
- o "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
- o "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.

```
              CLIENT                                   SERVER
             (PROVER)                                 VERIFIER)
                |                                        |
                |   Verifier sends challenge to prover   |
                |   Challenge = Q                        |
                |<---------------------------------------|
                |                                        |
                |   Prover Computes Response             |
                |   R = OCRA(K, {[C] | Q | [P | S | T]}) |
                |   Prover sends Response = R            |
                |--------------------------------------->|
                |                                        |
                |  Verifier Validates Response           |
                |  If Response is valid, Server sends OK |
                |  If Response is not,  Server sends NOK |
                |<---------------------------------------|
                |                                        |
```

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 Q1 and Q2.

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
    (PROVER)                                           (VERIFIER)
       |                                                   |
       |   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                                   |
       |<--------------------------------------------------|
       |                                                   |
```

7.3. Algorithm Modes for Signature

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

7.3.1. 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.

```
          CLIENT                                   SERVER
         (PROVER)                                (VERIFIER)
            |                                        |
            |    Verifier sends signature-challenge  |
            |    Challenge = QS                      |
            |<---------------------------------------|
            |                                        |
            |    Client Computes Response            |
            |    SIGN = OCRA(K, [C] | QS | [P | T])  |
            |    Response = SIGN                     |
            |--------------------------------------->|
            |                                        |
            |    Verifier Validates Response         |
            |    Response = OK                       |
            |<---------------------------------------|
            |                                        |
```

7.3.2. 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.

```
        CLIENT                                              SERVER
       (PROVER)                                            VERIFIER)
          |                                                   |
          |    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                                  |
          |<--------------------------------------------------|
          |                                                   |
```

8. 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.

8.1. 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 [RFC3174] (or SHA-256 or SHA-512 [SHA2]) 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.

8.2. 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 [RFC4086] 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 - All the communications SHOULD take place over a secure channel e.g. SSL/TLS [RFC5246], IPsec connections.

IC7 - 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.

IC8 - 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 mitigate against 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. [RFC5056] contains a relevant discussion of using Channel Bindings to Secure Channels

IC9 - 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.

IC10 - 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.

9. IANA Considerations

This document has no actions for IANA.

10. 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.

11. 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.

12. References

12.1. Normative references

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997, <http://www.ietf.org/rfc/rfc2104.txt>.

[RFC2119] "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, <http://www.ietf.org/rfc/rfc2119.txt>.

[RFC2279] Yergeau, F., "UTF-8, a transformation format of ISO 10646", RFC 2279, January 1998, <http://www.ietf.org/rfc/rfc2279.txt>.

[RFC3174] Eastlake, D. and P. Jones, "US Secure Hash Algorithm 1 (SHA1)", RFC 3174, September 2001, <http://www.ietf.org/rfc/rfc3174.txt>.

[RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Recommendations for Security", RFC 4086, June 2005, <http://www.ietf.org/rfc/rfc4086.txt>.

[RFC4226] M'Raihi, D., Bellare, M., Hoornaert, F., Naccache, D., and O. Ranen, "HOTP: An HMAC-Based One-Time Password Algorithm", RFC 4226, December 2005, <http://www.ietf.org/rfc/rfc4226.txt>.

[SHA2] NIST, "FIPS PUB 180-3: Secure Hash Standard (SHS)", October 2008, <http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf>.

12.2. Informative References

[CN] Coron, J. and D. Naccache, "An accurate evaluation of Maurer's universal test", LNCS 1556, February 1999, <http://www.gemplus.com/smart/rd/publications/pdf/CN99maur.pdf>.

[OATH] Initiative for Open AuTHentication, "OATH Vision", <http://www.openauthentication.org/about>.

[RFC5056] Williams, N., "On the Use of Channel Bindings to Secure

Channels", RFC 5056, November 2007,
<http://www.ietf.org/rfc/rfc5056.txt>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008, <http://www.ietf.org/rfc/rfc5246.txt>.

[RFC5929] Altman, J., Williams, N., and L. Zhu, "Channel Bindings for TLS", RFC 5929, July 2010, <http://www.ietf.org/rfc/rfc5929.txt>.

[UT] Wikipedia, "Unix time", <http://en.wikipedia.org/wiki/Unix_time>.

Appendix A. Reference Implementation

<CODE BEGINS>

```
/**
  Copyright (c) 2011 IETF Trust and the persons identified as authors of
  the code. All rights reserved.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
  A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
  OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
  DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
  THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
  (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
  OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import java.math.BigInteger;

/**
 * This 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 (HmacSHA1, HmacSHA256,
     *                                   HmacSHA512)
     * @param keyBytes   the bytes to use for the HMAC key
     * @param text       the message or text to be authenticated.
     */

    private static byte[] hmac_sha1(String crypto,
                     byte[] keyBytes, byte[] text){
        Mac hmac = null;
        try {
            hmac = Mac.getInstance(crypto);
            SecretKeySpec macKey =
                new SecretKeySpec(keyBytes, "RAW");
            hmac.init(macKey);
            return hmac.doFinal(text);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }


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

    /**
     * This method converts HEX string to Byte[]
     *
     * @param hex   the HEX string
     *
     * @return      A byte array
     */
    private static byte[] hexStr2Bytes(String hex){
        // Adding one byte to get the right conversion
        // values starting with "0" can be converted
        byte[] bArray = new BigInteger("10" + hex,16).toByteArray();
```

```
        // Copy all the REAL bytes, not the "first"
        byte[] ret = new byte[bArray.length - 1];
        System.arraycopy(bArray, 1, ret, 0, ret.length);
        return ret;
    }


    /**
     * 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, HEX encoded
     * @param password     a password that can be used, HEX encoded
     * @param sessionInformation Static information that identifies
     *                     the current session, Hex encoded
     * @param timeStamp    a value that reflects a time
     *
     * @return A numeric String in base 10 that includes
     * {@link 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.getBytes()).length;
        int counterLength = 0;
        int questionLength = 0;
        int passwordLength = 0;
        int sessionInformationLength = 0;
        int timeStampLength = 0;

        // The OCRASuites components
        String CryptoFunction = ocraSuite.split(":")[1];
        String DataInput = ocraSuite.split(":")[2];


        if(CryptoFunction.toLowerCase().indexOf("sha1") > 1)
```

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

        // How many digits should we return
        codeDigits = Integer.decode(CryptoFunction.substring(
                CryptoFunction.lastIndexOf("-")+1));

        // The size of the byte array message to be encrypted
        // Counter
        if(DataInput.toLowerCase().startsWith("c")) {
            // Fix the length of the HEX string
            while(counter.length() < 16)
                counter = "0" + counter;
            counterLength=8;
        }
        // Question - always 128 bytes
        if(DataInput.toLowerCase().startsWith("q") ||
                (DataInput.toLowerCase().indexOf("-q") >= 0)) {
            while(question.length() < 256)
                question = question + "0";
            questionLength=128;
        }

        // Password - sha1
        if(DataInput.toLowerCase().indexOf("psha1") > 1){
            while(password.length() < 40)
                password = "0" + password;
            passwordLength=20;
        }

        // Password - sha256
        if(DataInput.toLowerCase().indexOf("psha256") > 1){
            while(password.length() < 64)
                password = "0" + password;
            passwordLength=32;
        }

        // Password - sha512
        if(DataInput.toLowerCase().indexOf("psha512") > 1){
            while(password.length() < 128)
                password = "0" + password;
            passwordLength=64;
        }

        // sessionInformation - s064
```

```
        if(DataInput.toLowerCase().indexOf("s064") > 1){
            while(sessionInformation.length() < 128)
                sessionInformation = "0" + sessionInformation;
            sessionInformationLength=64;
        }

        // sessionInformation - s128
        if(DataInput.toLowerCase().indexOf("s128") > 1){
            while(sessionInformation.length() < 256)
                sessionInformation = "0" + sessionInformation;
            sessionInformationLength=128;
        }

        // sessionInformation - s256
        if(DataInput.toLowerCase().indexOf("s256") > 1){
            while(sessionInformation.length() < 512)
                sessionInformation = "0" + sessionInformation;
            sessionInformationLength=256;
        }

        // sessionInformation - s512
        if(DataInput.toLowerCase().indexOf("s512") > 1){
            while(sessionInformation.length() < 1024)
                sessionInformation = "0" + sessionInformation;
            sessionInformationLength=512;
        }

        // TimeStamp
        if(DataInput.toLowerCase().startsWith("t") ||
                (DataInput.toLowerCase().indexOf("-t") > 1)){
            while(timeStamp.length() < 16)
                timeStamp = "0" + timeStamp;
            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();
        System.arraycopy(bArray, 0, msg, 0, bArray.length);
```

```
        // Delimiter
        msg[bArray.length] = 0x00;

        // Put the bytes of "Counter" to the message
        // Input is HEX encoded
        if(counterLength > 0 ){
            bArray = hexStr2Bytes(counter);
            System.arraycopy(bArray, 0, msg, ocraSuiteLength + 1,
                    bArray.length);
        }


        // Put the bytes of "question" to the message
        // Input is text encoded
        if(questionLength > 0 ){
            bArray = hexStr2Bytes(question);
            System.arraycopy(bArray, 0, msg, ocraSuiteLength + 1 +
                    counterLength, bArray.length);
        }

        // Put the bytes of "password" to the message
        // Input is HEX encoded
        if(passwordLength > 0){
            bArray = hexStr2Bytes(password);
            System.arraycopy(bArray, 0, msg, ocraSuiteLength + 1 +
                    counterLength +     questionLength, bArray.length);
        }

        // Put the bytes of "sessionInformation" to the message
        // Input is text encoded
        if(sessionInformationLength > 0 ){
            bArray = hexStr2Bytes(sessionInformation);
            System.arraycopy(bArray, 0, msg, ocraSuiteLength + 1 +
                    counterLength +     questionLength +
                    passwordLength, bArray.length);
        }

        // Put the bytes of "time" to the message
        // Input is text value of minutes
        if(timeStampLength > 0){
            bArray = hexStr2Bytes(timeStamp);
            System.arraycopy(bArray, 0, msg, ocraSuiteLength + 1 +
                    counterLength + questionLength +
                    passwordLength + sessionInformationLength,
                    bArray.length);
        }

        bArray = hexStr2Bytes(key);
```

```
        byte[] 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;
    }
}
```

<CODE ENDS>

Appendix B. Test Vectors Generation

```
<CODE BEGINS>

/**
  Copyright (c) 2011 IETF Trust and the persons identified as authors of
  the code. All rights reserved.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
  A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
  OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
  DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
  THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
  (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
  OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

import java.math.BigInteger;
import java.util.*;
import java.text.DateFormat;
import java.text.SimpleDateFormat;

public class TestOCRA {

public static String asHex (byte buf[]) {
    StringBuffer strbuf = new StringBuffer(buf.length * 2);
    int i;

    for (i = 0; i < buf.length; i++) {
        if (((int) buf[i] & 0xff) < 0x10)
            strbuf.append("0");
        strbuf.append(Long.toString((int) buf[i] & 0xff, 16));
    }
    return strbuf.toString();
}

/**
 * @param args
 */
public static void main(String[] args) {

    String ocra = "";
```

```
    String seed = "";
    String ocraSuite = "";
    String counter = "";
    String password = "";
    String sessionInformation = "";
    String question = "";
    String qHex = "";
    String timeStamp = "";

    // PASS1234 is SHA1 hash of "1234"
    String PASS1234 = "7110eda4d09e062aa5e4a390b0a572ac0d2c0220";

    String SEED = "3132333435363738393031323334353637383930";
    String SEED32 = "3132333435363738393031323334353637383939" +
        "3031323334353637383930313132";
    String SEED64 = "3132333435363738393031323334353637383939" +
        "3031323334353637383930313233343536373839" +
        "3031323334353637383930313233343536373839" +
        "3031323334";
    int STOP = 5;

    Date myDate = Calendar.getInstance().getTime();
    BigInteger b = new BigInteger("0");
    String sDate = "Mar 25 2008, 12:06:30 GMT";

    try{
        DateFormat df =
            new SimpleDateFormat("MMM dd yyyy, HH:mm:ss zzz");
        myDate = df.parse(sDate);
        b = new BigInteger("0" + myDate.getTime());
        b = b.divide(new BigInteger("60000"));


        System.out.println("Time of \"" + sDate + "\" is in");
        System.out.println("milli sec: " + myDate.getTime());
        System.out.println("minutes: " + b.toString());
        System.out.println("minutes (HEX encoded): "
            + b.toString(16).toUpperCase());
        System.out.println("Time of \"" + sDate
            + "\" is the same as this localized");
        System.out.println("time, \""
            + new Date(myDate.getTime()) + "\"");

        System.out.println();
        System.out.println("Standard 20Byte key: " +
            "3132333435363738393031323334353637383930");
        System.out.println("Standard 32Byte key: " +
            "3132333435363738393031323334353637383930");
```

```
        System.out.println("                     " +
            "3132333435363738393031 32");
        System.out.println("Standard 64Byte key: 313233343536373839"
            + "3031323334353637383930");
        System.out.println("                     313233343536373839"
            + "3031323334353637383930");
        System.out.println("                     313233343536373839"
            + "3031323334353637383930");
        System.out.println("                     31323334");

        System.out.println();
        System.out.println("Plain challenge response");
        System.out.println("========================");
        System.out.println();

        ocraSuite = "OCRA-1:HOTP-SHA1-6:QN08";
        System.out.println(ocraSuite);
        System.out.println("=======================");
        seed = SEED;
        counter = "";
        question = "";
        password = "";
        sessionInformation = "";
        timeStamp = "";
        for(int i=0; i < 10; i++){
            question = "" + i + i + i + i + i + i + i + i;
            qHex = new String((new BigInteger(question,10))
                       .toString(16)).toUpperCase();
            ocra = OCRA.generateOCRA(ocraSuite,seed,counter,
                           qHex,password,
                           sessionInformation,timeStamp);
             System.out.println("Key: Standard 20Byte  Q: "
                   + question + "  OCRA: " + ocra);
        }
        System.out.println();

        ocraSuite = "OCRA-1:HOTP-SHA256-8:C-QN08-PSHA1";
        System.out.println(ocraSuite);
        System.out.println("=================================");
        seed = SEED32;
        counter = "";
        question = "12345678";
        password = PASS1234;
        sessionInformation = "";
        timeStamp = "";
        for(int i=0; i < 10; i++){
            counter = "" + i;
            qHex = new String((new BigInteger(question,10))
```

```
                  .toString(16)).toUpperCase();
            ocra = OCRA.generateOCRA(ocraSuite,seed,counter,
                        qHex,password,sessionInformation,timeStamp);
            System.out.println("Key: Standard 32Byte  C: "
                          + counter + "  Q: "
                          + question + "  PIN(1234): ");
            System.out.println(password + "  OCRA: " + ocra);
        }
        System.out.println();

        ocraSuite = "OCRA-1:HOTP-SHA256-8:QN08-PSHA1";
        System.out.println(ocraSuite);
        System.out.println("===============================");
        seed = SEED32;
        counter = "";
        question = "";
        password = PASS1234;
        sessionInformation = "";
        timeStamp = "";
        for(int i=0; i < STOP; i++){
            question = "" + i + i + i + i + i + i + i + i;

            qHex = new String((new BigInteger(question,10))
                         .toString(16)).toUpperCase();
            ocra = OCRA.generateOCRA(ocraSuite,seed,counter,
                      qHex,password,sessionInformation,timeStamp);
            System.out.println("Key: Standard 32Byte  Q: "
                         + question + "  PIN(1234): ");
            System.out.println(password + "  OCRA: " + ocra);
        }
        System.out.println();

        ocraSuite = "OCRA-1:HOTP-SHA512-8:C-QN08";
        System.out.println(ocraSuite);
        System.out.println("===========================");
        seed = SEED64;
        counter = "";
        question = "";
        password = "";
        sessionInformation = "";
        timeStamp = "";
        for(int i=0; i < 10; i++){
            question = "" + i + i + i + i + i + i + i + i;
            qHex = new String((new BigInteger(question,10))
                         .toString(16)).toUpperCase();
            counter = "0000" + i;
            ocra = OCRA.generateOCRA(ocraSuite,seed,counter,
                      qHex,password,sessionInformation,timeStamp);
```

```
            System.out.println("Key: Standard 64Byte  C: "
                     + counter + "  Q: "
                     + question + "  OCRA: " + ocra);
        }
        System.out.println();

        ocraSuite = "OCRA-1:HOTP-SHA512-8:QN08-T1M";
        System.out.println(ocraSuite);
        System.out.println("=============================");
        seed = SEED64;
        counter = "";
        question = "";
        password = "";
        sessionInformation = "";
        timeStamp = b.toString(16);
        for(int i=0; i < STOP; i++){
            question = "" + i + i + i + i + i + i + i + i;
            counter = "";
            qHex = new String((new BigInteger(question,10))
                        .toString(16)).toUpperCase();
            ocra = OCRA.generateOCRA(ocraSuite,seed,counter,
                     qHex,password,sessionInformation,timeStamp);

            System.out.println("Key: Standard 64Byte  Q: "
                        + question +"  T: "
                          + timeStamp.toUpperCase()
                        + "  OCRA: " + ocra);
        }
        System.out.println();

        System.out.println();
        System.out.println("Mutual Challenge Response");
        System.out.println("=========================");
        System.out.println();

        ocraSuite = "OCRA-1:HOTP-SHA256-8:QA08";
        System.out.println("OCRASuite (server computation) = "
                           + ocraSuite);
        System.out.println("OCRASuite (client computation) = "
                           + ocraSuite);
        System.out.println("===============================" +
            "===========================");
        seed = SEED32;
        counter = "";
        question = "";
        password = "";
        sessionInformation = "";
        timeStamp = "";
```

```
        for(int i=0; i < STOP; i++){
            question = "CLI2222" + i + "SRV1111" + i;
            qHex = asHex(question.getBytes());
            ocra = OCRA.generateOCRA(ocraSuite,seed,counter,qHex,
                         password,sessionInformation,timeStamp);
            System.out.println(
                     "(server)Key: Standard 32Byte  Q: "
                     + question + "  OCRA: "
                     + ocra);
            question = "SRV1111" + i + "CLI2222" + i;
            qHex = asHex(question.getBytes());
            ocra = OCRA.generateOCRA(ocraSuite,seed,counter,qHex,
                         password,sessionInformation,timeStamp);
            System.out.println(
                     "(client)Key: Standard 32Byte  Q: "
                     + question + "  OCRA: "
                     + ocra);
        }
        System.out.println();

        String ocraSuite1 = "OCRA-1:HOTP-SHA512-8:QA08";
        String ocraSuite2 = "OCRA-1:HOTP-SHA512-8:QA08-PSHA1";
        System.out.println("OCRASuite (server computation) = "
                           + ocraSuite1);
        System.out.println("OCRASuite (client computation) = "
                           + ocraSuite2);
        System.out.println("===============================" +
            "=================================");
        ocraSuite = "";
        seed = SEED64;
        counter = "";
        question = "";
        password = "";
        sessionInformation = "";
        timeStamp = "";
        for(int i=0; i < STOP; i++){
            ocraSuite = ocraSuite1;
            question = "CLI2222" + i + "SRV1111" + i;
            qHex = asHex(question.getBytes());
            password = "";
            ocra = OCRA.generateOCRA(ocraSuite,seed,counter,qHex,
                         password,sessionInformation,timeStamp);
            System.out.println(
                        "(server)Key: Standard 64Byte  Q: "
                        + question + "  OCRA: "
                        + ocra);
            ocraSuite = ocraSuite2;
            question = "SRV1111" + i + "CLI2222" + i;
```

```
            qHex = asHex(question.getBytes());
            password = PASS1234;
            ocra = OCRA.generateOCRA(ocraSuite,seed,counter,qHex,
                         password,sessionInformation,timeStamp);
            System.out.println("(client)Key: Standard 64Byte  Q: "
                         + question);
            System.out.println("P: " + password.toUpperCase()
                         + "  OCRA: " + ocra);
        }
        System.out.println();

        System.out.println();
        System.out.println("Plain Signature");
        System.out.println("===============");
        System.out.println();
        ocraSuite = "OCRA-1:HOTP-SHA256-8:QA08";
        System.out.println(ocraSuite);
        System.out.println("=========================");
        seed = SEED32;
        counter = "";
        question = "";
        password = "";
        sessionInformation = "";
        timeStamp = "";
        for(int i=0; i < STOP; i++){
            question = "SIG1" + i + "000";
            qHex = asHex(question.getBytes());
            ocra = OCRA.generateOCRA(ocraSuite,seed,counter,qHex,
                         password,sessionInformation,timeStamp);
            System.out.println(
                    "Key: Standard 32Byte  Q(Signature challenge): "
                    + question);
            System.out.println("   OCRA: " + ocra);
        }
        System.out.println();

        ocraSuite = "OCRA-1:HOTP-SHA512-8:QA10-T1M";
        System.out.println(ocraSuite);
        System.out.println("=============================");
        seed = SEED64;
        counter = "";
        question = "";
        password = "";
        sessionInformation = "";
        timeStamp = b.toString(16);
        for(int i=0; i < STOP; i++){
            question = "SIG1" + i + "00000";
            qHex = asHex(question.getBytes());
```

```
            ocra = OCRA.generateOCRA(ocraSuite,seed,counter,
                         qHex,password,sessionInformation,timeStamp);
            System.out.println(
                    "Key: Standard 64Byte  Q(Signature challenge): "
                    + question);
            System.out.println("   T: "
                    + timeStamp.toUpperCase() + "  OCRA: "
                    + ocra);
        }

    }catch (Exception e){
              System.out.println("Error : " + e);
    }
}
}
<CODE ENDS>
```

Appendix C. Test Vectors

This section provides test values that can be used for OCRA algorithm interoperability test.

Standard 20Byte key:

3132333435363738393031323334353637383930

Standard 32Byte key:

3132333435363738393031323334353637383930313233343536373839303132

Standard 64Byte key:

31323334353637383930313233343536373839303132333435363738393031323334
35363738393031323334353637383930313233343536373839303132333435363738393031323334

PIN (1234) SHA1 hash value:

7110eda4d09e062aa5e4a390b0a572ac0d2c0220

C.1. Plain challenge response

Key	Q	OCRA Value
Standard 20Byte	00000000	237653
Standard 20Byte	11111111	243178
Standard 20Byte	22222222	653583
Standard 20Byte	33333333	740991
Standard 20Byte	44444444	608993
Standard 20Byte	55555555	388898
Standard 20Byte	66666666	816933
Standard 20Byte	77777777	224598
Standard 20Byte	88888888	750600
Standard 20Byte	99999999	294470

OCRA-1:HOTP-SHA1-6:QN08

Key	C	Q	OCRA Value
Standard 32Byte	0	12345678	65347737
Standard 32Byte	1	12345678	86775851
Standard 32Byte	2	12345678	78192410
Standard 32Byte	3	12345678	71565254
Standard 32Byte	4	12345678	10104329
Standard 32Byte	5	12345678	65983500
Standard 32Byte	6	12345678	70069104
Standard 32Byte	7	12345678	91771096
Standard 32Byte	8	12345678	75011558
Standard 32Byte	9	12345678	08522129

OCRA-1:HOTP-SHA256-8:C-QN08-PSHA1

Key	Q	OCRA Value
Standard 32Byte	00000000	83238735
Standard 32Byte	11111111	01501458
Standard 32Byte	22222222	17957585
Standard 32Byte	33333333	86776967
Standard 32Byte	44444444	86807031

OCRA-1:HOTP-SHA256-8:QN08-PSHA1

Key	C	Q	OCRA Value
Standard 64Byte	00000	00000000	07016083
Standard 64Byte	00001	11111111	63947962
Standard 64Byte	00002	22222222	70123924
Standard 64Byte	00003	33333333	25341727
Standard 64Byte	00004	44444444	33203315
Standard 64Byte	00005	55555555	34205738
Standard 64Byte	00006	66666666	44343969
Standard 64Byte	00007	77777777	51946085
Standard 64Byte	00008	88888888	20403879
Standard 64Byte	00009	99999999	31409299

OCRA-1:HOTP-SHA512-8:C-QN08

Key	Q	T	OCRA Value
Standard 64Byte	00000000	132d0b6	95209754
Standard 64Byte	11111111	132d0b6	55907591
Standard 64Byte	22222222	132d0b6	22048402
Standard 64Byte	33333333	132d0b6	24218844
Standard 64Byte	44444444	132d0b6	36209546

OCRA-1:HOTP-SHA512-8:QN08-T1M

C.2. Mutual Challenge Response

OCRASuite (server computation) = OCRA-1:HOTP-SHA256-8:QA08

OCRASuite (client computation) = OCRA-1:HOTP-SHA256-8:QA08

Key	Q	OCRA Value
Standard 32Byte	CLI22220SRV11110	28247970
Standard 32Byte	CLI22221SRV11111	01984843
Standard 32Byte	CLI22222SRV11112	65387857
Standard 32Byte	CLI22223SRV11113	03351211
Standard 32Byte	CLI22224SRV11114	83412541

Server -- OCRA-1:HOTP-SHA256-8:QA08

Key	Q	OCRA Value
Standard 32Byte	SRV11110CLI22220	15510767
Standard 32Byte	SRV11111CLI22221	90175646
Standard 32Byte	SRV11112CLI22222	33777207
Standard 32Byte	SRV11113CLI22223	95285278
Standard 32Byte	SRV11114CLI22224	28934924

Client -- OCRA-1:HOTP-SHA256-8:QA08

OCRASuite (server computation) = OCRA-1:HOTP-SHA512-8:QA08

OCRASuite (client computation) = OCRA-1:HOTP-SHA512-8:QA08-PSHA1

Key	Q	OCRA Value
Standard 64Byte	CLI22220SRV11110	79496648
Standard 64Byte	CLI22221SRV11111	76831980
Standard 64Byte	CLI22222SRV11112	12250499
Standard 64Byte	CLI22223SRV11113	90856481
Standard 64Byte	CLI22224SRV11114	12761449

Server -- OCRA-1:HOTP-SHA512-8:QA08

Key	Q	OCRA Value
Standard 64Byte	SRV11110CLI22220	18806276
Standard 64Byte	SRV11111CLI22221	70020315
Standard 64Byte	SRV11112CLI22222	01600026
Standard 64Byte	SRV11113CLI22223	18951020
Standard 64Byte	SRV11114CLI22224	32528969

Client -- OCRA-1:HOTP-SHA512-8:QA08-PSHA1

C.3. Plain Signature

In this mode of operation, Q represents the signature challenge.

Key	Q	OCRA Value
Standard 32Byte	SIG10000	53095496
Standard 32Byte	SIG11000	04110475
Standard 32Byte	SIG12000	31331128
Standard 32Byte	SIG13000	76028668
Standard 32Byte	SIG14000	46554205

OCRA-1:HOTP-SHA256-8:QA08

Key	Q	T	OCRA Value
Standard 64Byte	SIG1000000	132d0b6	77537423
Standard 64Byte	SIG1100000	132d0b6	31970405
Standard 64Byte	SIG1200000	132d0b6	10235557
Standard 64Byte	SIG1300000	132d0b6	95213541
Standard 64Byte	SIG1400000	132d0b6	65360607

OCRA-1:HOTP-SHA512-8:QA10-T1M

Authors' Addresses

David M'Raihi
Verisign, Inc.
487 E. Middlefield Road
Mountain View, CA 94043
USA

Email: davidietf@gmail.com

Johan Rydell
Portwise, Inc.
275 Hawthorne Ave, Suite 119
Palo Alto, CA 94301
USA

Email: johanietf@gmail.com

Siddharth Bajaj
Symantec Corp.
350 Ellis Street
Mountain View, CA 94043
USA

Email: siddharthietf@gmail.com

Salah Machani
Diversinet Corp.
2225 Sheppard Avenue East, Suite 1801
Toronto, Ontario M2J 5C2
Canada

Email: smachani@diversinet.com

David Naccache
Ecole Normale Superieure
ENS DI, 45 rue d'Ulm
Paris,   75005
France

Email: david.naccache@ens.fr