Internet Engineering Task Force (IETF)
Request for Comments: 6238
Category: Informational
ISSN: 2070-1721

D. M’Raihi
Verisign, Inc.
S. Machani
Diversinet Corp.
M. Pei
Symantec
J. Rydell
Portwise, Inc.

May 2011

# TOTP: Time-Based One-Time Password Algorithm

## Abstract

This document describes an extension of the One-Time Password (OTP) algorithm, namely the HMAC-based One-Time Password (HOTP) algorithm, as defined in RFC 4226, to support the time-based moving factor. The HOTP algorithm specifies an event-based OTP algorithm, where the moving factor is an event counter. The present work bases the moving factor on a time value. A time-based variant of the OTP algorithm provides short-lived OTP values, which are desirable for enhanced security.

The proposed algorithm can be used across a wide range of network applications, from remote Virtual Private Network (VPN) access and Wi-Fi network logon to transaction-oriented Web applications. The authors believe that a common and shared algorithm will facilitate adoption of two-factor authentication on the Internet by enabling interoperability across commercial and open-source implementations.

## Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6238.

## Copyright Notice

## Table of Contents

# 1. Introduction

## 1.1. Scope

This document describes an extension of the One-Time Password (OTP) algorithm, namely the HMAC-based One-Time Password (HOTP) algorithm, as defined in [RFC4226], to support the time-based moving factor.

### 1.2. Background

As defined in [RFC4226], the HOTP algorithm is based on the HMAC-SHA-1 algorithm (as specified in [RFC2104]) and applied to an increasing counter value representing the message in the HMAC computation.

Basically, the output of the HMAC-SHA-1 calculation is truncated to obtain user-friendly values:

```
HOTP(K,C) = Truncate(HMAC-SHA-1(K,C))
```

where Truncate represents the function that can convert an HMAC-SHA-1 value into an HOTP value. K and C represent the shared secret and counter value; see [RFC4226] for detailed definitions.

TOTP is the time-based variant of this algorithm, where a value T, derived from a time reference and a time step, replaces the counter C in the HOTP computation.

TOTP implementations MAY use HMAC-SHA-256 or HMAC-SHA-512 functions, based on SHA-256 or SHA-512 [SHA2] hash functions, instead of the HMAC-SHA-1 function that has been specified for the HOTP computation in [RFC4226].

## 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 summarizes the requirements taken into account for designing the TOTP algorithm.

R1: The prover (e.g., token, soft token) and verifier (authentication or validation server) MUST know or be able to derive the current Unix time (i.e., the number of seconds elapsed since midnight UTC of January 1, 1970) for OTP generation. See [UT] for a more detailed definition of the commonly known "Unix time". The precision of the time used by the prover affects how often the clock synchronization should be done; see Section 6.

R2: The prover and verifier MUST either share the same secret or the knowledge of a secret transformation to generate a shared secret.

R3: The algorithm MUST use HOTP [RFC4226] as a key building block.

R4: The prover and verifier MUST use the same time-step value X.

R5: There MUST be a unique secret (key) for each prover.

R6: The keys SHOULD be randomly generated or derived using key derivation algorithms.

R7: The keys MAY be stored in a tamper-resistant device and SHOULD be protected against unauthorized access and usage.

## 4. TOTP Algorithm

This variant of the HOTP algorithm specifies the calculation of a one-time password value, based on a representation of the counter as a time factor.

### 4.1. Notations

- X represents the time step in seconds (default value X = 30 seconds) and is a system parameter.

- T0 is the Unix time to start counting time steps (default value is 0, i.e., the Unix epoch) and is also a system parameter.

### 4.2. Description

Basically, we define TOTP as TOTP = HOTP(K, T), where T is an integer and represents the number of time steps between the initial counter time T0 and the current Unix time.

More specifically, T = (Current Unix time - T0) / X, where the default floor function is used in the computation.

For example, with T0 = 0 and Time Step X = 30, T = 1 if the current Unix time is 59 seconds, and T = 2 if the current Unix time is 60 seconds.

The implementation of this algorithm MUST support a time value T larger than a 32-bit integer when it is beyond the year 2038. The value of the system parameters X and T0 are pre-established during the provisioning process and communicated between a prover and verifier as part of the provisioning step. The provisioning flow is out of scope of this document; refer to [RFC6030] for such provisioning container specifications.

## 5. Security Considerations

### 5.1. General

The security and strength of this algorithm depend 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 inputs are uniformly and independently distributed strings.

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

As indicated in the algorithm requirement section, keys SHOULD be chosen at random or using a cryptographically strong pseudorandom generator properly seeded with a random value.

Keys SHOULD be of the length of the HMAC output to facilitate interoperability.

We RECOMMEND following the recommendations in [RFC4086] for all pseudorandom and random number generations. The pseudorandom numbers used for generating the keys SHOULD successfully pass the randomness test specified in [CN], or a similar well-recognized test.

All the communications SHOULD take place over a secure channel, e.g., Secure Socket Layer/Transport Layer Security (SSL/TLS) [RFC5246] or IPsec connections [RFC4301].

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 OTP value, and re-encrypted immediately to limit exposure in the RAM to 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.

### 5.2. Validation and Time-Step Size

An OTP generated within the same time step will be the same. When an OTP is received at a validation system, it doesn’t know a client’s exact timestamp when an OTP was generated. The validation system may typically use the timestamp when an OTP is received for OTP comparison. Due to network latency, the gap (as measured by T, that is, the number of time steps since T0) between the time that the OTP was generated and the time that the OTP arrives at the receiving system may be large. The receiving time at the validation system and the actual OTP generation may not fall within the same time-step window that produced the same OTP. When an OTP is generated at the end of a time-step window, the receiving time most likely falls into the next time-step window. A validation system SHOULD typically set a policy for an acceptable OTP transmission delay window for validation. The validation system should compare OTPs not only with the receiving timestamp but also the past timestamps that are within the transmission delay. A larger acceptable delay window would expose a larger window for attacks. We RECOMMEND that at most one time step is allowed as the network delay.

The time-step size has an impact on both security and usability. A larger time-step size means a larger validity window for an OTP to be accepted by a validation system. There are implications for using a larger time-step size, as follows:

First, a larger time-step size exposes a larger window to attack. When an OTP is generated and exposed to a third party before it is consumed, the third party can consume the OTP within the time-step window.

We RECOMMEND a default time-step size of 30 seconds. This default value of 30 seconds is selected as a balance between security and usability.

Second, the next different OTP must be generated in the next time-step window. A user must wait until the clock moves to the next time-step window from the last submission. The waiting time may not be exactly the length of the time step, depending on when the last OTP was generated. For example, if the last OTP was generated at the halfway point in a time-step window, the waiting time for the next OTP is half the length of the time step. In general, a larger time-step window means a longer waiting time for a user to get the next valid OTP after the last successful OTP validation. A too-large window (for example, 10 minutes) most probably won’t be suitable for typical Internet login use cases; a user may not be able to get the next OTP within 10 minutes and therefore will have to re-login to the same site in 10 minutes.

Note that a prover may send the same OTP inside a given time-step window multiple times to a verifier. The verifier MUST NOT accept the second attempt of the OTP after the successful validation has been issued for the first OTP, which ensures one-time only use of an OTP.

## 6. Resynchronization

Because of possible clock drifts between a client and a validation server, we RECOMMEND that the validator be set with a specific limit to the number of time steps a prover can be "out of synch" before being rejected.

This limit can be set both forward and backward from the calculated time step on receipt of the OTP value. If the time step is 30 seconds as recommended, and the validator is set to only accept two time steps backward, then the maximum elapsed time drift would be around 89 seconds, i.e., 29 seconds in the calculated time step and 60 seconds for two backward time steps.

This would mean the validator could perform a validation against the current time and then two further validations for each backward step (for a total of 3 validations). Upon successful validation, the validation server can record the detected clock drift for the token in terms of the number of time steps. When a new OTP is received after this step, the validator can validate the OTP with the current timestamp adjusted with the recorded number of time-step clock drifts for the token.

Also, it is important to note that the longer a prover has not sent an OTP to a validation system, the longer (potentially) the accumulated clock drift between the prover and the verifier. In such cases, the automatic resynchronization described above may not work if the drift exceeds the allowed threshold. Additional authentication measures should be used to safely authenticate the prover and explicitly resynchronize the clock drift between the prover and the validator.

## 7. Acknowledgements

The authors of this document would like to thank the following people for their contributions and support to make this a better specification: Hannes Tschofenig, Jonathan Tuliani, David Dix, Siddharth Bajaj, Stu Veath, Shuh Chang, Oanh Hoang, John Huang, and Siddhartha Mohapatra.

## 8. References

### 8.1. Normative References

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Recommendations for Security", BCP 106, RFC 4086, June 2005.

[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.

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

### 8.2. 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>.

[RFC4301] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, December 2005.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC6030] Hoyer, P., Pei, M., and S. Machani, "Portable Symmetric Key Container (PSKC)", RFC 6030, October 2010.

[UT] Wikipedia, "Unix time", February 2011, <http://en.wikipedia.org/wiki/Unix_time>.

## Appendix A.  TOTP Algorithm: Reference Implementation

<CODE BEGINS>

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

 Redistribution and use in source and binary forms, with or without
 modification, is permitted pursuant to, and subject to the license
 terms contained in, the Simplified BSD License set forth in Section
 4.c of the IETF Trust’s Legal Provisions Relating to IETF Documents
 (http://trustee.ietf.org/license-info).
 */

 import java.lang.reflect.UndeclaredThrowableException;
 import java.security.GeneralSecurityException;
 import java.text.DateFormat;
 import java.text.SimpleDateFormat;
 import java.util.Date;
 import javax.crypto.Mac;
 import javax.crypto.spec.SecretKeySpec;
 import java.math.BigInteger;
 import java.util.TimeZone;


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

 public class TOTP {

     private TOTP() {}

     /**
      * 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_sha(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);
         }
     }


     /**
      * This method converts a 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];
         for (int i = 0; i < ret.length; i++)
             ret[i] = bArray[i+1];
         return ret;
     }

     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 generates a TOTP value for the given
      * set of parameters.
      *
      * @param key: the shared secret, HEX encoded
      * @param time: a value that reflects a time
      * @param returnDigits: number of digits to return
      *
      * @return: a numeric String in base 10 that includes
      *              {@link truncationDigits} digits
      */

     public static String generateTOTP(String key,
             String time,
             String returnDigits){
         return generateTOTP(key, time, returnDigits, "HmacSHA1");
     }


     /**
      * This method generates a TOTP value for the given
      * set of parameters.
      *
      * @param key: the shared secret, HEX encoded
      * @param time: a value that reflects a time
      * @param returnDigits: number of digits to return
      *
      * @return: a numeric String in base 10 that includes
      *              {@link truncationDigits} digits
      */

     public static String generateTOTP256(String key,
             String time,
             String returnDigits){
         return generateTOTP(key, time, returnDigits, "HmacSHA256");
     }
     /**
      * This method generates a TOTP value for the given
      * set of parameters.
      *
      * @param key: the shared secret, HEX encoded
      * @param time: a value that reflects a time
      * @param returnDigits: number of digits to return
      *
      * @return: a numeric String in base 10 that includes
      *              {@link truncationDigits} digits
      */

     public static String generateTOTP512(String key,
             String time,
             String returnDigits){
         return generateTOTP(key, time, returnDigits, "HmacSHA512");
     }


     /**
      * This method generates a TOTP value for the given
      * set of parameters.
      *
      * @param key: the shared secret, HEX encoded
      * @param time: a value that reflects a time
      * @param returnDigits: number of digits to return
      * @param crypto: the crypto function to use
      *
      * @return: a numeric String in base 10 that includes
      *              {@link truncationDigits} digits
      */

     public static String generateTOTP(String key,
             String time,
             String returnDigits,
             String crypto){
         int codeDigits = Integer.decode(returnDigits).intValue();
         String result = null;

         // Using the counter
         // First 8 bytes are for the movingFactor
         // Compliant with base RFC 4226 (HOTP)
         while (time.length() < 16 )
             time = "0" + time;

         // Get the HEX in a Byte[]
         byte[] msg = hexStr2Bytes(time);
         byte[] k = hexStr2Bytes(key);
         byte[] hash = hmac_sha(crypto, k, 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;
     }

     public static void main(String[] args) {
         // Seed for HMAC-SHA1 - 20 bytes
         String seed = "3132333435363738393031323334353637383930";
         // Seed for HMAC-SHA256 - 32 bytes
         String seed32 = "3132333435363738393031323334353637383930" +
         "313233343536373839303132";
         // Seed for HMAC-SHA512 - 64 bytes
         String seed64 = "3132333435363738393031323334353637383930" +
         "3132333435363738393031323334353637383930" +
         "3132333435363738393031323334353637383930" +
         "31323334";
         long T0 = 0;
         long X = 30;
         long testTime[] = {59L, 1111111109L, 1111111111L,
                 1234567890L, 2000000000L, 20000000000L};

         String steps = "0";
         DateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
         df.setTimeZone(TimeZone.getTimeZone("UTC"));
         try {
             System.out.println(
                     "+---------------+-----------------------+" +
             "------------------+--------+--------+");
             System.out.println(
                     "|  Time(sec)    |   Time (UTC format)   " +
             "| Value of T(Hex)  |  TOTP  | Mode   |");
             System.out.println(
                     "+---------------+-----------------------+" +
             "------------------+--------+--------+");

             for (int i=0; i<testTime.length; i++) {
                 long T = (testTime[i] - T0)/X;
                 steps = Long.toHexString(T).toUpperCase();
                 while (steps.length() < 16) steps = "0" + steps;
                 String fmtTime = String.format("%1$-11s", testTime[i]);
                 String utcTime = df.format(new Date(testTime[i]*1000));
                 System.out.print("|  " + fmtTime + "  |  " + utcTime +
                         "  | " + steps + " |");
                 System.out.println(generateTOTP(seed, steps, "8",
                 "HmacSHA1") + "| SHA1   |");
                 System.out.print("|  " + fmtTime + "  |  " + utcTime +
                         "  | " + steps + " |");
                 System.out.println(generateTOTP(seed32, steps, "8",
                 "HmacSHA256") + "| SHA256 |");
                 System.out.print("|  " + fmtTime + "  |  " + utcTime +
                         "  | " + steps + " |");
                 System.out.println(generateTOTP(seed64, steps, "8",
                 "HmacSHA512") + "| SHA512 |");

                 System.out.println(
                         "+---------------+-----------------------+" +
                 "------------------+--------+--------+");
             }
         }catch (final Exception e){
             System.out.println("Error : " + e);
         }
     }
 }

 <CODE ENDS>
```

## Appendix B.  Test Vectors

This section provides test values that can be used for the HOTP time-based variant algorithm interoperability test.

The test token shared secret uses the ASCII string value "12345678901234567890". With Time Step X = 30, and the Unix epoch as the initial value to count time steps, where T0 = 0, the TOTP algorithm will display the following values for specified modes and timestamps.

| Time (sec) | UTC Time | Value of T (hex) | TOTP | Mode |
|---|---|---|---|---|
| 59 | 1970-01-01 00:00:59 | 0000000000000001 | 94287082 | SHA1 |
| 59 | 1970-01-01 00:00:59 | 0000000000000001 | 46119246 | SHA256 |
| 59 | 1970-01-01 00:00:59 | 0000000000000001 | 90693936 | SHA512 |
| 1111111109 | 2005-03-18 01:58:29 | 00000000023523EC | 07081804 | SHA1 |
| 1111111109 | 2005-03-18 01:58:29 | 00000000023523EC | 68084774 | SHA256 |
| 1111111109 | 2005-03-18 01:58:29 | 00000000023523EC | 25091201 | SHA512 |
| 1111111111 | 2005-03-18 01:58:31 | 00000000023523ED | 14050471 | SHA1 |
| 1111111111 | 2005-03-18 01:58:31 | 00000000023523ED | 67062674 | SHA256 |
| 1111111111 | 2005-03-18 01:58:31 | 00000000023523ED | 99943326 | SHA512 |
| 1234567890 | 2009-02-13 23:31:30 | 000000000273EF07 | 89005924 | SHA1 |
| 1234567890 | 2009-02-13 23:31:30 | 000000000273EF07 | 91819424 | SHA256 |
| 1234567890 | 2009-02-13 23:31:30 | 000000000273EF07 | 93441116 | SHA512 |
| 2000000000 | 2033-05-18 03:33:20 | 0000000003F940AA | 69279037 | SHA1 |
| 2000000000 | 2033-05-18 03:33:20 | 0000000003F940AA | 90698825 | SHA256 |
| 2000000000 | 2033-05-18 03:33:20 | 0000000003F940AA | 38618901 | SHA512 |
| 20000000000 | 2603-10-11 11:33:20 | 0000000027BC86AA | 65353130 | SHA1 |
| 20000000000 | 2603-10-11 11:33:20 | 0000000027BC86AA | 77737706 | SHA256 |
| 20000000000 | 2603-10-11 11:33:20 | 0000000027BC86AA | 47863826 | SHA512 |

Table 1: TOTP Table

## Authors' Addresses

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

EMail: davidietf@gmail.com

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

EMail: smachani@diversinet.com

Mingliang Pei
Symantec
510 E. Middlefield Road
Mountain View, CA  94043
USA

EMail: Mingliang_Pei@symantec.com

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

EMail: johanietf@gmail.com