Network Working Group
Request for Comments: 4226
Category: Informational

D. M'Raihi
VeriSign
M. Bellare
UCSD
F. Hoornaert
Vasco
D. Naccache
Gemplus
O. Ranen
Aladdin

December 2005

HOTP: An HMAC-Based One-Time Password Algorithm

Status of This Memo

Copyright Notice

Abstract

This document describes an algorithm to generate one-time password values, based on Hashed Message Authentication Code (HMAC). A security analysis of the algorithm is presented, and important parameters related to the secure deployment of the algorithm are discussed. The proposed algorithm can be used across a wide range of network applications ranging from remote Virtual Private Network (VPN) access, Wi-Fi network logon to transaction-oriented Web applications.

This work is a joint effort by the OATH (Open AuTHentication) membership to specify an algorithm that can be freely distributed to the technical community. 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.

Table of Contents

1. Overview

The document introduces first the context around an algorithm that generates one-time password values based on HMAC [BCK1] and, thus, is named the HMAC-Based One-Time Password (HOTP) algorithm. In Section 4, the algorithm requirements are listed and in Section 5, the HOTP algorithm is described. Sections 6 and 7 focus on the algorithm security. Section 8 proposes some extensions and improvements, and Section 10 concludes this document. In Appendix A, the interested reader will find a detailed, full-fledged analysis of the algorithm security: an idealized version of the algorithm is evaluated, and then the HOTP algorithm security is analyzed.

2. Introduction

Today, deployment of two-factor authentication remains extremely limited in scope and scale. Despite increasingly higher levels of threats and attacks, most Internet applications still rely on weak authentication schemes for policing user access. The lack of interoperability among hardware and software technology vendors has been a limiting factor in the adoption of two-factor authentication technology. In particular, the absence of open specifications has led to solutions where hardware and software components are tightly coupled through proprietary technology, resulting in high-cost solutions, poor adoption, and limited innovation.

In the last two years, the rapid rise of network threats has exposed the inadequacies of static passwords as the primary mean of authentication on the Internet. At the same time, the current approach that requires an end user to carry an expensive, single-function device that is only used to authenticate to the network is clearly not the right answer. For two-factor authentication to propagate on the Internet, it will have to be embedded in more flexible devices that can work across a wide range of applications.

The ability to embed this base technology while ensuring broad interoperability requires that it be made freely available to the broad technical community of hardware and software developers. Only an open-system approach will ensure that basic two-factor authentication primitives can be built into the next generation of consumer devices such as USB mass storage devices, IP phones, and personal digital assistants.

One-Time Password is certainly one of the simplest and most popular forms of two-factor authentication for securing network access. For example, in large enterprises, Virtual Private Network access often requires the use of One-Time Password tokens for remote user authentication. One-Time Passwords are often preferred to stronger

forms of authentication such as Public-Key Infrastructure (PKI) or biometrics because an air-gap device does not require the installation of any client desktop software on the user machine, therefore allowing them to roam across multiple machines including home computers, kiosks, and personal digital assistants.

This document proposes a simple One-Time Password algorithm that can be implemented by any hardware manufacturer or software developer to create interoperable authentication devices and software agents. The algorithm is event-based so that it can be embedded in high-volume devices such as Java smart cards, USB dongles, and GSM SIM cards. The presented algorithm is made freely available to the developer community under the terms and conditions of the IETF Intellectual Property Rights [RFC3979].

The authors of this document are members of the Open AuTHentication initiative [OATH]. The initiative was created in 2004 to facilitate collaboration among strong authentication technology providers.

3. 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 [RFC2119].

4. Algorithm Requirements

This section presents the main requirements that drove this algorithm design. A lot of emphasis was placed on end-consumer usability as well as the ability for the algorithm to be implemented by low-cost hardware that may provide minimal user interface capabilities. In particular, the ability to embed the algorithm into high-volume SIM and Java cards was a fundamental prerequisite.

R1 - The algorithm MUST be sequence- or counter-based: one of the goals is to have the HOTP algorithm embedded in high-volume devices such as Java smart cards, USB dongles, and GSM SIM cards.

R2 - The algorithm SHOULD be economical to implement in hardware by minimizing requirements on battery, number of buttons, computational horsepower, and size of LCD display.

R3 - The algorithm MUST work with tokens that do not support any numeric input, but MAY also be used with more sophisticated devices such as secure PIN-pads.

R4 - The value displayed on the token MUST be easily read and entered by the user: This requires the HOTP value to be of reasonable length.

The HOTP value must be at least a 6-digit value. It is also desirable that the HOTP value be 'numeric only' so that it can be easily entered on restricted devices such as phones.

R5 - There MUST be user-friendly mechanisms available to resynchronize the counter. Section 7.4 and Appendix E.4 details the resynchronization mechanism proposed in this document

R6 - The algorithm MUST use a strong shared secret. The length of the shared secret MUST be at least 128 bits. This document RECOMMENDs a shared secret length of 160 bits.

5. HOTP Algorithm

In this section, we introduce the notation and describe the HOTP algorithm basic blocks -- the base function to compute an HMAC-SHA-1 value and the truncation method to extract an HOTP value.

5.1. Notation and Symbols

A string always means a binary string, meaning a sequence of zeros and ones.

If s is a string, then |s| denotes its length.

If n is a number, then |n| denotes its absolute value.

If s is a string, then s[i] denotes its i-th bit. We start numbering the bits at 0, so s = s[0]s[1]...s[n-1] where n = |s| is the length of s.

Let StToNum (String to Number) denote the function that as input a string s returns the number whose binary representation is s. (For example, StToNum(110) = 6.)

Here is a list of symbols used in this document.

Symbol	Represents
C	8-byte counter value, the moving factor. This counter MUST be synchronized between the HOTP generator (client) and the HOTP validator (server).
K	shared secret between client and server; each HOTP generator has a different and unique secret K.
T	throttling parameter: the server will refuse connections from a user after T unsuccessful authentication attempts.

s resynchronization parameter: the server will attempt to verify a received authenticator across s consecutive counter values.

Digit number of digits in an HOTP value; system parameter.

5.2. Description

The HOTP algorithm is based on an increasing counter value and a static symmetric key known only to the token and the validation service. In order to create the HOTP value, we will use the HMAC-SHA-1 algorithm, as defined in RFC 2104 [BCK2].

As the output of the HMAC-SHA-1 calculation is 160 bits, we must truncate this value to something that can be easily entered by a user.

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

Where:

- Truncate represents the function that converts an HMAC-SHA-1 value into an HOTP value as defined in Section 5.3.

The Key (K), the Counter (C), and Data values are hashed high-order byte first.

The HOTP values generated by the HOTP generator are treated as big endian.

5.3. Generating an HOTP Value

We can describe the operations in 3 distinct steps:

Step 1: Generate an HMAC-SHA-1 value Let HS = HMAC-SHA-1(K,C) // HS is a 20-byte string

Step 2: Generate a 4-byte string (Dynamic Truncation)
Let Sbits = DT(HS) // DT, defined below,
// returns a 31-bit string

Step 3: Compute an HOTP value
Let Snum = StToNum(Sbits) // Convert S to a number in
0...2^{31}-1
Return D = Snum mod 10^Digit // D is a number in the range
0...10^{Digit}-1

The Truncate function performs Step 2 and Step 3, i.e., the dynamic truncation and then the reduction modulo 10^Digit. The purpose of the dynamic offset truncation technique is to extract a 4-byte dynamic binary code from a 160-bit (20-byte) HMAC-SHA-1 result.

```
 DT(String) // String = String[0]...String[19]
  Let OffsetBits be the low-order 4 bits of String[19]
  Offset = StToNum(OffsetBits) // 0 <= OffSet <= 15
  Let P = String[OffSet]...String[OffSet+3]
  Return the Last 31 bits of P
```

The reason for masking the most significant bit of P is to avoid confusion about signed vs. unsigned modulo computations. Different processors perform these operations differently, and masking out the signed bit removes all ambiguity.

Implementations MUST extract a 6-digit code at a minimum and possibly 7 and 8-digit code. Depending on security requirements, Digit = 7 or more SHOULD be considered in order to extract a longer HOTP value.

The following paragraph is an example of using this technique for Digit = 6, i.e., that a 6-digit HOTP value is calculated from the HMAC value.

5.4. Example of HOTP Computation for Digit = 6

The following code example describes the extraction of a dynamic binary code given that hmac_result is a byte array with the HMAC-SHA-1 result:

```
     int offset   =  hmac_result[19] & 0xf ;
     int bin_code = (hmac_result[offset]  & 0x7f) << 24
        | (hmac_result[offset+1] & 0xff) << 16
        | (hmac_result[offset+2] & 0xff) <<  8
        | (hmac_result[offset+3] & 0xff) ;
```

SHA-1 HMAC Bytes (Example)

Byte Number																	
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
Byte Value																	
1f	86	98	69	0e	02	ca	16	61	85	50	ef	7f	19	da	8e	94	5b	55	5a

* The last byte (byte 19) has the hex value 0x5a.
* The value of the lower 4 bits is 0xa (the offset value).
* The offset value is byte 10 (0xa).
* The value of the 4 bytes starting at byte 10 is 0x50ef7f19, which is the dynamic binary code DBC1.
* The MSB of DBC1 is 0x50 so DBC2 = DBC1 = 0x50ef7f19 .
* HOTP = DBC2 modulo 10^6 = 872921.

We treat the dynamic binary code as a 31-bit, unsigned, big-endian integer; the first byte is masked with a 0x7f.

We then take this number modulo 1,000,000 (10^6) to generate the 6-digit HOTP value 872921 decimal.

6. Security Considerations

The conclusion of the security analysis detailed in the Appendix is that, for all practical purposes, the outputs of the Dynamic Truncation (DT) on distinct counter inputs are uniformly and independently distributed 31-bit strings.

The security analysis then details the impact of the conversion from a string to an integer and the final reduction modulo 10^Digit, where Digit is the number of digits in an HOTP value.

The analysis demonstrates that these final steps introduce a negligible bias, which does not impact the security of the HOTP algorithm, in the sense that the best possible attack against the HOTP function is the brute force attack.

Assuming an adversary is able to observe numerous protocol exchanges and collect sequences of successful authentication values. This adversary, trying to build a function F to generate HOTP values based on his observations, will not have a significant advantage over a random guess.

The logical conclusion is simply that the best strategy will once again be to perform a brute force attack to enumerate and try all the possible values.

Considering the security analysis in the Appendix of this document, without loss of generality, we can approximate closely the security of the HOTP algorithm by the following formula:

$$Sec = sv/10^{Digit}$$

Where:
- Sec is the probability of success of the adversary;
- s is the look-ahead synchronization window size;
- v is the number of verification attempts;
- Digit is the number of digits in HOTP values.

Obviously, we can play with s, T (the Throttling parameter that would limit the number of attempts by an attacker), and Digit until achieving a certain level of security, still preserving the system usability.

7. Security Requirements

Any One-Time Password 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.

The parameters T and s discussed in this section have a significant impact on the security -- further details in Section 6 elaborate on the relations between these parameters and their impact on the system security.

It is also important to remark that the HOTP algorithm is not a substitute for encryption and does not provide for the privacy of data transmission. Other mechanisms should be used to defeat attacks aimed at breaking confidentiality and privacy of transactions.

7.1. Authentication Protocol Requirements

We introduce in this section some requirements for a protocol P implementing HOTP as the authentication method between a prover and a verifier.

RP1 - P 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 One-Time Password value for authentication purpose (two-factor authentication).

RP2 - P SHOULD NOT be vulnerable to brute force attacks. This implies that a throttling/lockout scheme is RECOMMENDED on the validation server side.

RP3 - P SHOULD be implemented over a secure channel in order to protect users' privacy and avoid replay attacks.

7.2. Validation of HOTP Values

The HOTP client (hardware or software token) increments its counter and then calculates the next HOTP value HOTP client. If the value received by the authentication server matches the value calculated by the client, then the HOTP value is validated. In this case, the server increments the counter value by one.

If the value received by the server does not match the value calculated by the client, the server initiate the resynch protocol (look-ahead window) before it requests another pass.

If the resynch fails, the server asks then for another authentication pass of the protocol to take place, until the maximum number of authorized attempts is reached.

If and when the maximum number of authorized attempts is reached, the server SHOULD lock out the account and initiate a procedure to inform the user.

7.3. Throttling at the Server

Truncating the HMAC-SHA-1 value to a shorter value makes a brute force attack possible. Therefore, the authentication server needs to detect and stop brute force attacks.

We RECOMMEND setting a throttling parameter T, which defines the maximum number of possible attempts for One-Time Password validation. The validation server manages individual counters per HOTP device in order to take note of any failed attempt. We RECOMMEND T not to be too large, particularly if the resynchronization method used on the server is window-based, and the window size is large. T SHOULD be set as low as possible, while still ensuring that usability is not significantly impacted.

Another option would be to implement a delay scheme to avoid a brute force attack. After each failed attempt A, the authentication server would wait for an increased T*A number of seconds, e.g., say T = 5, then after 1 attempt, the server waits for 5 seconds, at the second failed attempt, it waits for 5*2 = 10 seconds, etc.

The delay or lockout schemes MUST be across login sessions to prevent attacks based on multiple parallel guessing techniques.

7.4. Resynchronization of the Counter

Although the server's counter value is only incremented after a successful HOTP authentication, the counter on the token is incremented every time a new HOTP is requested by the user. Because of this, the counter values on the server and on the token might be out of synchronization.

We RECOMMEND setting a look-ahead parameter s on the server, which defines the size of the look-ahead window. In a nutshell, the server can recalculate the next s HOTP-server values, and check them against the received HOTP client.

Synchronization of counters in this scenario simply requires the server to calculate the next HOTP values and determine if there is a match. Optionally, the system MAY require the user to send a sequence of (say, 2, 3) HOTP values for resynchronization purpose, since forging a sequence of consecutive HOTP values is even more difficult than guessing a single HOTP value.

The upper bound set by the parameter s ensures the server does not go on checking HOTP values forever (causing a denial-of-service attack) and also restricts the space of possible solutions for an attacker trying to manufacture HOTP values. s SHOULD be set as low as possible, while still ensuring that usability is not impacted.

7.5. Management of Shared Secrets

The operations dealing with the shared secrets used to generate and verify OTP values must be performed securely, in order to mitigate risks of any leakage of sensitive information. We describe in this section different modes of operations and techniques to perform these different operations with respect to the state of the art in data security.

We can consider two different avenues for generating and storing (securely) shared secrets in the Validation system:

* Deterministic Generation: secrets are derived from a master seed, both at provisioning and verification stages and generated on-the-fly whenever it is required.
* Random Generation: secrets are generated randomly at provisioning stage and must be stored immediately and kept secure during their life cycle.

Deterministic Generation

A possible strategy is to derive the shared secrets from a master secret. The master secret will be stored at the server only. A tamper-resistant device MUST be used to store the master key and derive the shared secrets from the master key and some public information. The main benefit would be to avoid the exposure of the shared secrets at any time and also avoid specific requirements on storage, since the shared secrets could be generated on-demand when needed at provisioning and validation time.

We distinguish two different cases:

- A single master key MK is used to derive the shared secrets; each HOTP device has a different secret, K_i = SHA-1 (MK,i) where i stands for a public piece of information that identifies uniquely the HOTP device such as a serial number, a token ID, etc. Obviously, this is in the context of an application or service -- different application or service providers will have different secrets and settings.
- Several master keys MK_i are used and each HOTP device stores a set of different derived secrets, {K_i,j = SHA-1(MK_i,j)} where j stands for a public piece of information identifying the device. The idea would be to store ONLY the active master key at the validation server, in the Hardware Security Module (HSM), and keep in a safe place, using secret sharing methods such as [Shamir] for instance. In this case, if a master secret MK_i is compromised, then it is possible to switch to another secret without replacing all the devices.

The drawback in the deterministic case is that the exposure of the master secret would obviously enable an attacker to rebuild any shared secret based on correct public information. The revocation of all secrets would be required, or switching to a new set of secrets in the case of multiple master keys.

On the other hand, the device used to store the master key(s) and generate the shared secrets MUST be tamper resistant. Furthermore, the HSM will not be exposed outside the security perimeter of the validation system, therefore reducing the risk of leakage.

Random Generation

The shared secrets are randomly generated. We RECOMMEND following the recommendations in [RFC4086] and selecting a good and secure random source for generating these secrets. A (true) random generator requires a naturally occurring source of randomness. Practically, there are two possible avenues to consider for the generation of the shared secrets:

* Hardware-based generators: they exploit the randomness that occurs in physical phenomena. A nice implementation can be based on oscillators and built in such ways that active attacks are more difficult to perform.

* Software-based generators: designing a good software random generator is not an easy task. A simple, but efficient, implementation should be based on various sources and apply to the sampled sequence a one-way function such as SHA-1.

We RECOMMEND selecting proven products, being hardware or software generators, for the computation of shared secrets.

We also RECOMMEND storing the shared secrets securely, and more specifically encrypting the shared secrets when stored using tamper-resistant hardware encryption and exposing them only when required: for example, the shared secret is decrypted when needed to verify an HOTP value, and re-encrypted immediately to limit exposure in the RAM for a short period of time. The data store holding the shared secrets 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 shared secrets should be limited to programs and processes required by the validation system only. We will not elaborate on the different security mechanisms to put in place, but obviously, the protection of shared secrets is of the uttermost importance.

8. Composite Shared Secrets

It may be desirable to include additional authentication factors in the shared secret K. These additional factors can consist of any data known at the token but not easily obtained by others. Examples of such data include:

* PIN or Password obtained as user input at the token
* Phone number
* Any unique identifier programmatically available at the token

In this scenario, the composite shared secret K is constructed during the provisioning process from a random seed value combined with one or more additional authentication factors. The server could either build on-demand or store composite secrets -- in any case, depending on implementation choice, the token only stores the seed value. When the token performs the HOTP calculation, it computes K from the seed value and the locally derived or input values of the other authentication factors.

The use of composite shared secrets can strengthen HOTP-based authentication systems through the inclusion of additional authentication factors at the token. To the extent that the token is a trusted device, this approach has the further benefit of not requiring exposure of the authentication factors (such as the user input PIN) to other devices.

9. Bi-Directional Authentication

Interestingly enough, the HOTP client could also be used to authenticate the validation server, claiming that it is a genuine entity knowing the shared secret.

Since the HOTP client and the server are synchronized and share the same secret (or a method to recompute it), a simple 3-pass protocol could be put in place:

1- The end user enter the TokenID and a first OTP value OTP1;
2- The server checks OTP1 and if correct, sends back OTP2;
3- The end user checks OTP2 using his HOTP device and if correct, uses the web site.

Obviously, as indicated previously, all the OTP communications have to take place over a secure channel, e.g., SSL/TLS, IPsec connections.

10. Conclusion

This document describes HOTP, a HMAC-based One-Time Password algorithm. It also recommends the preferred implementation and related modes of operations for deploying the algorithm.

The document also exhibits elements of security and demonstrates that the HOTP algorithm is practical and sound, the best possible attack being a brute force attack that can be prevented by careful implementation of countermeasures in the validation server.

Eventually, several enhancements have been proposed, in order to improve security if needed for specific applications.

11. Acknowledgements

The authors would like to thank Siddharth Bajaj, Alex Deacon, Loren Hart, and Nico Popp for their help during the conception and redaction of this document.

12. Contributors

The authors of this document would like to emphasize the role of three persons who have made a key contribution to this document:

- Laszlo Elteto is system architect with SafeNet, Inc.

- Ernesto Frutos is director of Engineering with Authenex, Inc.

- Fred McClain is Founder and CTO with Boojum Mobile, Inc.

Without their advice and valuable inputs, this document would not be the same.

13. References

13.1. Normative References

[BCK1] M. Bellare, R. Canetti and H. Krawczyk, "Keyed Hash Functions and Message Authentication", Proceedings of Crypto'96, LNCS Vol. 1109, pp. 1-15.

[BCK2] 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.

[RFC3979] Bradner, S., "Intellectual Property Rights in IETF Technology", BCP 79, RFC 3979, March 2005.

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

13.2. Informative References

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

[PrOo] B. Preneel and P. van Oorschot, "MD-x MAC and building fast MACs from hash functions", Advances in Cryptology CRYPTO '95, Lecture Notes in Computer Science Vol. 963, D. Coppersmith ed., Springer-Verlag, 1995.

[Crack] Crack in SHA-1 code 'stuns' security gurus
http://www.eetimes.com/showArticle.jhtml?articleID=60402150

[Sha1] Bruce Schneier. SHA-1 broken. February 15, 2005.
http://www.schneier.com/blog/archives/2005/02/sha1_broken.html

[Res] Researchers: Digital encryption standard flawed
http://news.com.com/Researchers+Digital+encryption+standard+flawed/2100-1002-5579881.html?part=dht&tag=ntop&tag=nl.e703

[Shamir] How to Share a Secret, by Adi Shamir. In Communications of the ACM, Vol. 22, No. 11, pp. 612-613, November, 1979.

Appendix A - HOTP Algorithm Security: Detailed Analysis

The security analysis of the HOTP algorithm is summarized in this section. We first detail the best attack strategies, and then elaborate on the security under various assumptions and the impact of the truncation and make some recommendations regarding the number of digits.

We focus this analysis on the case where Digit = 6, i.e., an HOTP function that produces 6-digit values, which is the bare minimum recommended in this document.

A.1. Definitions and Notations

We denote by {0,1}^l the set of all strings of length l.

Let Z_{n} = {0,.., n - 1}.

Let IntDiv(a,b) denote the integer division algorithm that takes input integers a, b where a >= b >= 1 and returns integers (q,r)

the quotient and remainder, respectively, of the division of a by b. (Thus, a = bq + r and 0 <= r < b.)

Let H: {0,1}^k x {0,1}^c --> {0,1}^n be the base function that takes a k-bit key K and c-bit counter C and returns an n-bit output H(K,C). (In the case of HOTP, H is HMAC-SHA-1; we use this formal definition for generalizing our proof of security.)

A.2. The Idealized Algorithm: HOTP-IDEAL

We now define an idealized counterpart of the HOTP algorithm. In this algorithm, the role of H is played by a random function that forms the key.

To be more precise, let Maps(c,n) denote the set of all functions mapping from {0,1}^c to {0,1}^n. The idealized algorithm has key space Maps(c,n), so that a "key" for such an algorithm is a function h from {0,1}^c to {0,1}^n. We imagine this key (function) to be drawn at random. It is not feasible to implement this idealized algorithm, since the key, being a function from {0,1}^c to {0,1}^n, is way too large to even store. So why consider it?

Our security analysis will show that as long as H satisfies a certain well-accepted assumption, the security of the actual and idealized algorithms is for all practical purposes the same. The task that really faces us, then, is to assess the security of the idealized algorithm.

In analyzing the idealized algorithm, we are concentrating on assessing the quality of the design of the algorithm itself, independently of HMAC-SHA-1. This is in fact the important issue.

A.3. Model of Security

The model exhibits the type of threats or attacks that are being considered and enables one to assess the security of HOTP and HOTP-IDEAL. We denote ALG as either HOTP or HOTP-IDEAL for the purpose of this security analysis.

The scenario we are considering is that a user and server share a key K for ALG. Both maintain a counter C, initially zero, and the user authenticates itself by sending ALG(K,C) to the server. The latter accepts if this value is correct.

In order to protect against accidental increment of the user counter, the server, upon receiving a value z, will accept as long as z equals ALG(K,i) for some i in the range C,...,C + s-1, where s is the resynchronization parameter and C is the server counter. If it accepts with some value of i, it then increments its counter to i+1. If it does not accept, it does not change its counter value.

The model we specify captures what an adversary can do and what it needs to achieve in order to "win". First, the adversary is assumed to be able to eavesdrop, meaning, to see the authenticator transmitted by the user. Second, the adversary wins if it can get the server to accept an authenticator relative to a counter value for which the user has never transmitted an authenticator.

The formal adversary, which we denote by B, starts out knowing which algorithm ALG is being used, knowing the system design, and knowing all system parameters. The one and only thing it is not given a priori is the key K shared between the user and the server.

The model gives B full control of the scheduling of events. It has access to an authenticator oracle representing the user. By calling this oracle, the adversary can ask the user to authenticate itself and get back the authenticator in return. It can call this oracle as often as it wants and when it wants, using the authenticators it accumulates to perhaps "learn" how to make authenticators itself. At any time, it may also call a verification oracle, supplying the latter with a candidate authenticator of its choice. It wins if the server accepts this accumulator.

Consider the following game involving an adversary B that is attempting to compromise the security of an authentication algorithm ALG: K x {0,1}^c --> R.

Initializations - A key K is selected at random from K, a counter C is initialized to 0, and the Boolean value win is set to false.

Game execution - Adversary B is provided with the two following oracles:

```
Oracle AuthO()
--------------
   A = ALG(K,C)
   C = C + 1
   Return O to B

Oracle VerO(A)
--------------
   i = C
   While (i <= C + s - 1 and Win == FALSE) do
      If A == ALG(K,i) then Win = TRUE; C = i + 1
      Else i = i + 1
   Return Win to B
```

AuthO() is the authenticator oracle and VerO(A) is the verification oracle.

Upon execution, B queries the two oracles at will. Let Adv(B) be the probability that win gets set to true in the above game. This is the probability that the adversary successfully impersonates the user.

Our goal is to assess how large this value can be as a function of the number v of verification queries made by B, the number a of authenticator oracle queries made by B, and the running time t of B. This will tell us how to set the throttle, which effectively upper bounds v.

A.4. Security of the Ideal Authentication Algorithm

This section summarizes the security analysis of HOTP-IDEAL, starting with the impact of the conversion modulo 10^Digit and then focusing on the different possible attacks.

A.4.1. From Bits to Digits

The dynamic offset truncation of a random n-bit string yields a random 31-bit string. What happens to the distribution when it is taken modulo m = 10^Digit, as done in HOTP?

The following lemma estimates the biases in the outputs in this case.

Lemma 1

Let N >= m >= 1 be integers, and let (q,r) = IntDiv(N,m). For z in Z_{m} let:

$$P_{N,m}(z) = Pr [x \bmod m = z : x \text{ randomly pick in } Z_{n}]$$

Then for any z in Z_{m}

$$P_{N,m}(z) = \begin{cases} (q + 1) / N & \text{if } 0 \le z < r \\ q / N & \text{if } r \le z < m \end{cases}$$

Proof of Lemma 1

Let the random variable X be uniformly distributed over Z_{N}. Then:

$$P_{N,m}(z) = Pr [X \bmod m = z]$$

$$= Pr [X < mq] * Pr [X \bmod m = z | X < mq] + Pr [mq <= X < N] * Pr [X \bmod m = z | mq <= X < N]$$

$$= mq/N * 1/m + \begin{cases} (N - mq)/N * 1 / (N - mq) & \text{if } 0 \le z < N - mq \\ 0 & \text{if } N - mq \le z \le m \end{cases}$$

$$= q/N + \begin{cases} r/N * 1 / r & \text{if } 0 \le z < N - mq \\ 0 & \text{if } r \le z \le m \end{cases}$$

Simplifying yields the claimed equation.

Let N = 2^31, d = 6, and m = 10^d. If x is chosen at random from Z_{N} (meaning, is a random 31-bit string), then reducing it to a 6-digit number by taking x mod m does not yield a random 6-digit number.

Rather, x mod m is distributed as shown in the following table:

Values	Probability that each appears as output
0,1,...,483647	2148/2^31 roughly equals to 1.00024045/10^6
483648,...,999999	2147/2^31 roughly equals to 0.99977478/10^6

If X is uniformly distributed over Z_{2^31} (meaning, is a random 31-bit string), then the above shows the probabilities for different outputs of X mod 10^6. The first set of values appears with

probability slightly greater than 10^-6, the rest with probability slightly less, meaning that the distribution is slightly non-uniform.

However, as the table above indicates, the bias is small, and as we will see later, negligible: the probabilities are very close to 10^-6.

A.4.2. Brute Force Attacks

If the authenticator consisted of d random digits, then a brute force attack using v verification attempts would succeed with probability sv/10^Digit.

However, an adversary can exploit the bias in the outputs of HOTP-IDEAL, predicted by Lemma 1, to mount a slightly better attack.

Namely, it makes authentication attempts with authenticators that are the most likely values, meaning the ones in the range 0,...,r - 1, where (q,r) = IntDiv(2^31,10^Digit).

The following specifies an adversary in our model of security that mounts the attack. It estimates the success probability as a function of the number of verification queries.

For simplicity, we assume that the number of verification queries is at most r. With N = 2^31 and m = 10^6, we have r = 483,648, and the throttle value is certainly less than this, so this assumption is not much of a restriction.

Proposition 1

Suppose m = 10^Digit < 2^31, and let (q,r) = IntDiv(2^31,m). Assume s <= m. The brute-force-attack adversary B-bf attacks HOTP using v <= r verification oracle queries. This adversary makes no authenticator oracle queries, and succeeds with probability

$$Adv(B\text{-}bf) = 1 - (1 - v(q+1)/2^{31})^s$$

which is roughly equal to

$$sv * (q+1)/2^{31}$$

With m = 10^6 we get q = 2,147. In that case, the brute force attack using v verification attempts succeeds with probability

$$Adv(B\text{-}bf) \text{ roughly} = sv * 2148/2^{31} = sv * 1.00024045/10^6$$

As this equation shows, the resynchronization parameter s has a significant impact in that the adversary's success probability is proportional to s. This means that s cannot be made too large without compromising security.

A.4.3. Brute force attacks are the best possible attacks.

A central question is whether there are attacks any better than the brute force one. In particular, the brute force attack did not attempt to collect authenticators sent by the user and try to cryptanalyze them in an attempt to learn how to better construct authenticators. Would doing this help? Is there some way to "learn" how to build authenticators that result in a higher success rate than given by the brute-force attack?

The following says the answer to these questions is no. No matter what strategy the adversary uses, and even if it sees, and tries to exploit, the authenticators from authentication attempts of the user, its success probability will not be above that of the brute force attack -- this is true as long as the number of authentications it observes is not incredibly large. This is valuable information regarding the security of the scheme.

Proposition 2 ---------- Suppose m = 10^Digit < 2^31, and let (q,r) = IntDiv(2^31,m). Let B be any adversary attacking HOTP-IDEAL using v verification oracle queries and a <= 2^c - s authenticator oracle queries. Then

$$Adv(B) <= sv * (q+1)/ 2^{31}$$

Note: This result is conditional on the adversary not seeing more than 2^c - s authentications performed by the user, which is hardly restrictive as long as c is large enough.

With m = 10^6, we get q = 2,147. In that case, Proposition 2 says that any adversary B attacking HOTP-IDEAL and making v verification attempts succeeds with probability at most

Equation 1

$$sv * 2148/2^{31} \text{ roughly } = sv * 1.00024045/10^6$$

Meaning, B's success rate is not more than that achieved by the brute force attack.

A.5. Security Analysis of HOTP

We have analyzed, in the previous sections, the security of the idealized counterparts HOTP-IDEAL of the actual authentication algorithm HOTP. We now show that, under appropriate and well-believed assumption on H, the security of the actual algorithms is essentially the same as that of its idealized counterpart.

The assumption in question is that H is a secure pseudorandom function, or PRF, meaning that its input-output values are indistinguishable from those of a random function in practice.

Consider an adversary A that is given an oracle for a function f: {0,1}^c --> {0, 1}^n and eventually outputs a bit. We denote Adv(A) as the prf-advantage of A, which represents how well the adversary does at distinguishing the case where its oracle is H(K,.) from the case where its oracle is a random function of {0,1}^c to {0,1}^n.

One possible attack is based on exhaustive search for the key K. If A runs for t steps and T denotes the time to perform one computation of H, its prf-advantage from this attack turns out to be (t/T)2^-k. Another possible attack is a birthday one [PrOo], whereby A can attain advantage p^2/2^n in p oracle queries and running time about pT.

Our assumption is that these are the best possible attacks. This translates into the following.

Assumption 1

Let T denotes the time to perform one computation of H. Then if A is any adversary with running time at most t and making at most p oracle queries,

$$Adv(A) <= (t/T)/2^k + p^2/2^n$$

In practice, this assumption means that H is very secure as PRF. For example, given that k = n = 160, an attacker with running time 2^60 and making 2^40 oracle queries has advantage at most (about) 2^-80.

Theorem 1

Suppose m = 10^Digit < 2^31, and let (q,r) = IntDiv(2^31,m). Let B be any adversary attacking HOTP using v verification oracle queries,

a <= 2^c - s authenticator oracle queries, and running time t. Let T denote the time to perform one computation of H. If Assumption 1 is true, then

$$Adv(B) <= sv * (q + 1)/2^{31} + (t/T)/2^k + ((sv + a)^2)/2^n$$

In practice, the (t/T)2^-k + ((sv + a)^2)2^-n term is much smaller than the sv(q + 1)/2^n term, so that the above says that for all practical purposes the success rate of an adversary attacking HOTP is sv(q + 1)/2^n, just as for HOTP-IDEAL, meaning the HOTP algorithm is in practice essentially as good as its idealized counterpart.

In the case m = 10^6 of a 6-digit output, this means that an adversary making v authentication attempts will have a success rate that is at most that of Equation 1.

For example, consider an adversary with running time at most 2^60 that sees at most 2^40 authentication attempts of the user. Both these choices are very generous to the adversary, who will typically not have these resources, but we are saying that even such a powerful adversary will not have more success than indicated by Equation 1.

We can safely assume sv <= 2^40 due to the throttling and bounds on s. So:

$$(t/T)/2^k + ((sv + a)^2)/2^n <= 2^{60}/2^{160} + (2^{41})^2/2^{160}$$
$$\text{roughly} <= 2^{-78}$$

which is much smaller than the success probability of Equation 1 and negligible compared to it.

Appendix B - SHA-1 Attacks

This sections addresses the impact of the recent attacks on SHA-1 on the security of the HMAC-SHA-1-based HOTP. We begin with some discussion of the situation of SHA-1 and then discuss the relevance to HMAC-SHA-1 and HOTP. Cited references are in Section 13.

B.1. SHA-1 Status

A collision for a hash function h means a pair x,y of different inputs such that h(x)=h(y). Since SHA-1 outputs 160 bits, a birthday attack finds a collision in 2^{80} trials. (A trial means one computation of the function.) This was thought to be the best possible until Wang, Yin, and Yu announced on February 15, 2005, that they had an attack finding collisions in 2^{69} trials.

Is SHA-1 broken? For most practical purposes, we would say probably not, since the resources needed to mount the attack are huge. Here is one way to get a sense of it: we can estimate it is about the same as the time we would need to factor a 760-bit RSA modulus, and this is currently considered out of reach.

Burr of NIST is quoted in [Crack] as saying "Large national intelligence agencies could do this in a reasonable amount of time with a few million dollars in computer time". However, the computation may be out of reach of all but such well-funded agencies.

One should also ask what impact finding SHA-1 collisions actually has on security of real applications such as signatures. To exploit a collision x,y to forge signatures, you need to somehow obtain a signature of x and then you can forge a signature of y. How damaging this is depends on the content of y: the y created by the attack may not be meaningful in the application context. Also, one needs a chosen-message attack to get the signature of x. This seems possible in some contexts, but not others. Overall, it is not clear that the impact on the security of signatures is significant.

Indeed, one can read in the press that SHA-1 is "broken" [Sha1] and that encryption and SSL are "broken" [Res]. The media have a tendency to magnify events: it would hardly be interesting to announce in the news that a team of cryptanalysts did very interesting theoretical work in attacking SHA-1.

Cryptographers are excited too. But mainly because this is an important theoretical breakthrough. Attacks can only get better with time: it is therefore important to monitor any progress in hash functions cryptanalysis and be prepared for any really practical break with a sound migration plan for the future.

B.2. HMAC-SHA-1 Status

The new attacks on SHA-1 have no impact on the security of HMAC-SHA-1. The best attack on the latter remains one needing a sender to authenticate 2^{80} messages before an adversary can create a forgery. Why?

HMAC is not a hash function. It is a message authentication code (MAC) that uses a hash function internally. A MAC depends on a secret key, while hash functions don't. What one needs to worry about with a MAC is forgery, not collisions. HMAC was designed so that collisions in the hash function (here SHA-1) do not yield forgeries for HMAC.

Recall that HMAC-SHA-1(K,x) = SHA-1(K_o,SHA-1(K_i,x)) where the keys K_o,K_i are derived from K. Suppose the attacker finds a pair x,y such that SHA-1(K_i,x) = SHA-1(K_i,y). (Call this a hidden-key collision.) Then if it can obtain the MAC of x (itself a tall order), it can forge the MAC of y. (These values are the same.) But finding hidden-key collisions is harder than finding collisions, because the attacker does not know the hidden key K_i. All it may have is some outputs of HMAC-SHA-1 with key K. To date, there are no claims or evidence that the recent attacks on SHA-1 extend to find hidden-key collisions.

Historically, the HMAC design has already proven itself in this regard. MD5 is considered broken in that collisions in this hash function can be found relatively easily. But there is still no attack on HMAC-MD5 better than the trivial 2^{64} time birthday one. (MD5 outputs 128 bits, not 160.) We are seeing this strength of HMAC coming into play again in the SHA-1 context.

B.3. HOTP Status

Since no new weakness has surfaced in HMAC-SHA-1, there is no impact on HOTP. The best attacks on HOTP remain those described in the document, namely, to try to guess output values.

The security proof of HOTP requires that HMAC-SHA-1 behave like a pseudorandom function. The quality of HMAC-SHA-1 as a pseudorandom function is not impacted by the new attacks on SHA-1, and so neither is this proven guarantee.

Appendix C - HOTP Algorithm: Reference Implementation

```
/*
 * OneTimePasswordAlgorithm.java
 * OATH Initiative,
 * HOTP one-time password algorithm
 *
 */

/* Copyright (C) 2004, OATH.  All rights reserved.
 *
 * License to copy and use this software is granted provided that it
 * is identified as the "OATH HOTP Algorithm" in all material
 * mentioning or referencing this software or this function.
 *
 * License is also granted to make and use derivative works provided
 * that such works are identified as
 *  "derived from OATH HOTP algorithm"
 * in all material mentioning or referencing the derived work.
 *
 * OATH (Open AuTHentication) and its members make no
 * representations concerning either the merchantability of this
 * software or the suitability of this software for any particular
 * purpose.
 *
 * It is provided "as is" without express or implied warranty
 * of any kind and OATH AND ITS MEMBERS EXPRESSaLY DISCLAIMS
 * ANY WARRANTY OR LIABILITY OF ANY KIND relating to this software.
 *
 * These notices must be retained in any copies of any part of this
 * documentation and/or software.
 */

package org.openauthentication.otp;

import java.io.IOException;
import java.io.File;
import java.io.DataInputStream;
import java.io.FileInputStream ;
import java.lang.reflect.UndeclaredThrowableException;

import java.security.GeneralSecurityException;
import java.security.NoSuchAlgorithmException;
import java.security.InvalidKeyException;

import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
```

```
/**
 * This class contains static methods that are used to calculate the
 * One-Time Password (OTP) using
 * JCE to provide the HMAC-SHA-1.
 *
 * @author Loren Hart
 * @version 1.0
 */
public class OneTimePasswordAlgorithm {
    private OneTimePasswordAlgorithm() {}

    // These are used to calculate the check-sum digits.
    //                                0  1  2  3  4  5  6  7  8  9
    private static final int[] doubleDigits =
                    { 0, 2, 4, 6, 8, 1, 3, 5, 7, 9 };

    /**
     * Calculates the checksum using the credit card algorithm.
     * This algorithm has the advantage that it detects any single
     * mistyped digit and any single transposition of
     * adjacent digits.
     *
     * @param num the number to calculate the checksum for
     * @param digits number of significant places in the number
     *
     * @return the checksum of num
     */
    public static int calcChecksum(long num, int digits) {
        boolean doubleDigit = true;
        int     total = 0;
        while (0 < digits--) {
            int digit = (int) (num % 10);
            num /= 10;
            if (doubleDigit) {
                digit = doubleDigits[digit];
            }
            total += digit;
            doubleDigit = !doubleDigit;
        }
        int result = total % 10;
        if (result > 0) {
            result = 10 - result;
        }
        return result;
    }

    /**
     * This method uses the JCE to provide the HMAC-SHA-1
```

```
        * algorithm.
        * HMAC computes a Hashed Message Authentication Code and
        * in this case SHA1 is the hash algorithm used.
        *
        * @param keyBytes   the bytes to use for the HMAC-SHA-1 key
        * @param text       the message or text to be authenticated.
        *
        * @throws NoSuchAlgorithmException if no provider makes
        *       either HmacSHA1 or HMAC-SHA-1
        *       digest algorithms available.
        * @throws InvalidKeyException
        *       The secret provided was not a valid HMAC-SHA-1 key.
        *
        */

       public static byte[] hmac_sha1(byte[] keyBytes, byte[] text)
           throws NoSuchAlgorithmException, InvalidKeyException
       {
   //         try {
               Mac hmacSha1;
               try {
                   hmacSha1 = Mac.getInstance("HmacSHA1");
               } catch (NoSuchAlgorithmException nsae) {
                   hmacSha1 = Mac.getInstance("HMAC-SHA-1");
               }
               SecretKeySpec macKey =
           new SecretKeySpec(keyBytes, "RAW");
               hmacSha1.init(macKey);
               return hmacSha1.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};

       /**
        * This method generates an OTP value for the given
        * set of parameters.
        *
        * @param secret       the shared secret
        * @param movingFactor the counter, time, or other value that
        *                     changes on a per use basis.
        * @param codeDigits   the number of digits in the OTP, not
        *                     including the checksum, if any.
        * @param addChecksum  a flag that indicates if a checksum digit
```

```
     *                      should be appended to the OTP.
     * @param truncationOffset the offset into the MAC result to
     *                      begin truncation.  If this value is out of
     *                      the range of 0 ... 15, then dynamic
     *                      truncation  will be used.
     *                      Dynamic truncation is when the last 4
     *                      bits of the last byte of the MAC are
     *                      used to determine the start offset.
     * @throws NoSuchAlgorithmException if no provider makes
     *                      either HmacSHA1 or HMAC-SHA-1
     *                      digest algorithms available.
     * @throws InvalidKeyException
     *                      The secret provided was not
     *                      a valid HMAC-SHA-1 key.
     *
     * @return A numeric String in base 10 that includes
     * {@link codeDigits} digits plus the optional checksum
     * digit if requested.
     */
    static public String generateOTP(byte[] secret,
                long movingFactor,
            int codeDigits,
                boolean addChecksum,
            int truncationOffset)
        throws NoSuchAlgorithmException, InvalidKeyException
    {
        // put movingFactor value into text byte array
String result = null;
int digits = addChecksum ? (codeDigits + 1) : codeDigits;
        byte[] text = new byte[8];
        for (int i = text.length - 1; i >= 0; i--) {
            text[i] = (byte) (movingFactor & 0xff);
            movingFactor >>= 8;
        }

        // compute hmac hash
        byte[] hash = hmac_sha1(secret, text);

        // put selected bytes into result int
        int offset = hash[hash.length - 1] & 0xf;
if ( (0<=truncationOffset) &&
       (truncationOffset<(hash.length-4)) ) {
    offset = truncationOffset;
}
        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];
      if (addChecksum) {
          otp =  (otp * 10) + calcChecksum(otp, codeDigits);
      }
      result = Integer.toString(otp);
      while (result.length() < digits) {
          result = "0" + result;
      }
      return result;
        }
    }
```

Appendix D - HOTP Algorithm: Test Values

The following test data uses the ASCII string "12345678901234567890" for the secret:

Secret = 0x3132333435363738393031323334353637383930

Table 1 details for each count, the intermediate HMAC value.

Count	Hexadecimal HMAC-SHA-1(secret, count)
0	cc93cf18508d94934c64b65d8ba7667fb7cde4b0
1	75a48a19d4cbe100644e8ac1397eea747a2d33ab
2	0bacb7fa082fef30782211938bc1c5e70416ff44
3	66c28227d03a2d5529262ff016a1e6ef76557ece
4	a904c900a64b35909874b33e61c5938a8e15ed1c
5	a37e783d7b7233c083d4f62926c7a25f238d0316
6	bc9cd28561042c83f219324d3c607256c03272ae
7	a4fb960c0bc06e1eabb804e5b397cdc4b45596fa
8	1b3c89f65e6c9e883012052823443f048b4332db
9	1637409809a679dc698207310c8c7fc07290d9e5

Table 2 details for each count the truncated values (both in hexadecimal and decimal) and then the HOTP value.

	Truncated		
Count	Hexadecimal	Decimal	HOTP
0	4c93cf18	1284755224	755224
1	41397eea	1094287082	287082
2	82fef30	137359152	359152
3	66ef7655	1726969429	969429
4	61c5938a	1640338314	338314
5	33c083d4	868254676	254676
6	7256c032	1918287922	287922
7	4e5b397	82162583	162583
8	2823443f	673399871	399871
9	2679dc69	645520489	520489

Appendix E - Extensions

We introduce in this section several enhancements to the HOTP algorithm. These are not recommended extensions or part of the standard algorithm, but merely variations that could be used for customized implementations.

E.1. Number of Digits

A simple enhancement in terms of security would be to extract more digits from the HMAC-SHA-1 value.

For instance, calculating the HOTP value modulo 10^8 to build an 8-digit HOTP value would reduce the probability of success of the adversary from sv/10^6 to sv/10^8.

This could give the opportunity to improve usability, e.g., by increasing T and/or s, while still achieving a better security overall. For instance, s = 10 and 10v/10^8 = v/10^7 < v/10^6 which is the theoretical optimum for 6-digit code when s = 1.

E.2. Alphanumeric Values

Another option is to use A-Z and 0-9 values; or rather a subset of 32 symbols taken from the alphanumerical alphabet in order to avoid any confusion between characters: 0, O, and Q as well as l, 1, and I are very similar, and can look the same on a small display.

The immediate consequence is that the security is now in the order of sv/32^6 for a 6-digit HOTP value and sv/32^8 for an 8-digit HOTP value.

32^6 > 10^9 so the security of a 6-alphanumeric HOTP code is slightly better than a 9-digit HOTP value, which is the maximum length of an HOTP code supported by the proposed algorithm.

32^8 > 10^12 so the security of an 8-alphanumeric HOTP code is significantly better than a 9-digit HOTP value.

Depending on the application and token/interface used for displaying and entering the HOTP value, the choice of alphanumeric values could be a simple and efficient way to improve security at a reduced cost and impact on users.

E.3. Sequence of HOTP Values

As we suggested for the resynchronization to enter a short sequence (say, 2 or 3) of HOTP values, we could generalize the concept to the protocol, and add a parameter L that would define the length of the HOTP sequence to enter.

Per default, the value L SHOULD be set to 1, but if security needs to be increased, users might be asked (possibly for a short period of time, or a specific operation) to enter L HOTP values.

This is another way, without increasing the HOTP length or using alphanumeric values to tighten security.

Note: The system MAY also be programmed to request synchronization on a regular basis (e.g., every night, twice a week, etc.) and to achieve this purpose, ask for a sequence of L HOTP values.

E.4. A Counter-Based Resynchronization Method

In this case, we assume that the client can access and send not only the HOTP value but also other information, more specifically, the counter value.

A more efficient and secure method for resynchronization is possible in this case. The client application will not send the HOTP-client value only, but the HOTP-client and the related C-client counter value, the HOTP value acting as a message authentication code of the counter.

Resynchronization Counter-based Protocol (RCP)

The server accepts if the following are all true, where C-server is its own current counter value:

1) C-client >= C-server
2) C-client - C-server <= s
3) Check that HOTP client is valid HOTP(K,C-Client)
4) If true, the server sets C to C-client + 1 and client is authenticated

In this case, there is no need for managing a look-ahead window anymore. The probability of success of the adversary is only v/10^6 or roughly v in one million. A side benefit is obviously to be able to increase s "infinitely" and therefore improve the system usability without impacting the security.

This resynchronization protocol SHOULD be used whenever the related impact on the client and server applications is deemed acceptable.

E.5. Data Field

Another interesting option is the introduction of a Data field, which would be used for generating the One-Time Password values: HOTP (K, C, [Data]) where Data is an optional field that can be the concatenation of various pieces of identity-related information, e.g., Data = Address | PIN.

We could also use a Timer, either as the only moving factor or in combination with the Counter -- in this case, e.g., Data = Timer, where Timer could be the UNIX-time (GMT seconds since 1/1/1970) divided by some factor (8, 16, 32, etc.) in order to give a specific time step. The time window for the One-Time Password is then equal to the time step multiplied by the resynchronization parameter as defined before. For example, if we take 64 seconds as the time step and 7 for the resynchronization parameter, we obtain an acceptance window of +/- 3 minutes.

Using a Data field opens for more flexibility in the algorithm implementation, provided that the Data field is clearly specified.

Authors' Addresses

David M'Raihi (primary contact for sending comments and questions)
VeriSign, Inc.
685 E. Middlefield Road
Mountain View, CA 94043 USA

Phone: 1-650-426-3832
EMail: dmraihi@verisign.com

Mihir Bellare
Dept of Computer Science and Engineering, Mail Code 0114
University of California at San Diego
9500 Gilman Drive
La Jolla, CA 92093, USA

EMail: mihir@cs.ucsd.edu

Frank Hoornaert
VASCO Data Security, Inc.
Koningin Astridlaan 164
1780 Wemmel, Belgium

EMail: frh@vasco.com

David Naccache
Gemplus Innovation
34 rue Guynemer, 92447,
Issy les Moulineaux, France
and
Information Security Group,
Royal Holloway,
University of London, Egham,
Surrey TW20 0EX, UK

EMail: david.naccache@gemplus.com, david.naccache@rhul.ac.uk

Ohad Ranen
Aladdin Knowledge Systems Ltd.
15 Beit Oved Street
Tel Aviv, Israel 61110

EMail: Ohad.Ranen@ealaddin.com

Full Copyright Statement

Intellectual Property

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.