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:

```
(1)  int offset   =  hmac_result[19] & 0xf ;
(2)  int bin_code = (hmac_result[offset]  & 0x7f) << 24
(3)     | (hmac_result[offset+1] & 0xff) << 16
 :      | (hmac_result[offset+2] & 0xff) <<  8
(5)     | (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

(2) ... (5)

0x50ef7f19 % 1000000 (6 dígitos) = 872921 Código TOTP

(1) → 0x5a & 15 = 10

Usando struct.pack('>I', ...) es bastante más cómodo y legible.

* 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:

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