# Time-Based One-Time Password using SHA1 Hash Function

Kevin Nathaniel Wijaya - 13517072[1]
*Program Studi Teknik Informatika*
*Sekolah Teknik Elektro dan Informatika*
*Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia*
[1]*kevin.wijaya@students.itb.ac.id*

***Abstract*—The paper reviews the use of the Standard Hash Algorithm 1 for time-based one-time password. The time-based one-time password is based on the use of the hash-based one-time password, which is based on the hash-based message authentication code. The first part of the text explains the basic theories that will be used such as cryptography and hash. The next part shows the implementation of the Standard Hash Algorithm 1 in hashing a certain key and message, with the use of time to generate a six-digit code known as the time-based one-time password.**

***Keywords*—Time-based one-time password, SHA1 hash function, Hash-based one-time password, Hash-based message authentication code .**

## I. Introduction

You might wonder, why you have to enter a six-digit number which you got from either text message or an app, just to check your mail. You have already entered your login credentials, there should be no need of more security, right?

Nowadays, companies are starting to implement another stage of verification using one-time passwords. This is what you receive through some way and have to put in to continue logging in.

A one-time password (OTP) is a set of characters that is only valid for logging into a single session or completing a transaction. This one-time password is normally not stored or kept in any way in the database or storage, and is normally discarded after its use and no longer valid. This is different from normal passwords we have because of its single use and uniqueness.

A time-based one-time password (TOTP) is an extension of the one-time password. It is generated based on the time, a key, and a hashing algorithm, which this paper will be using Secure Hash Algorithm 1 (SHA1) from the many hashing algorithms.

The time-based one-time password algorithm will generate normally a six-digit number which will be used as a code to continue the authentication process the client is doing. From a key and a time that is both agreed upon between the client and the user, the TOTP algorithm will generate a six-digit number with an interval of a set time, usually thirty seconds. This will set the previously generated code unusable and the current code will only be valid for thirty seconds. This creates a secure environment, as even if the six-digit code is stolen, it would be rendered useless after thirty seconds, and without the secret key, the six-digit code would change and permanent access would not be granted to people trying to hack in.

In this paper, the author will explain about the time-based one-time password which uses the SHA1 hashing algorithm.

## II. Number Theory

Numbers are divided into different parts such as real numbers, rational numbers, whole numbers, natural numbers, and so on, as shown in Fig. 1.

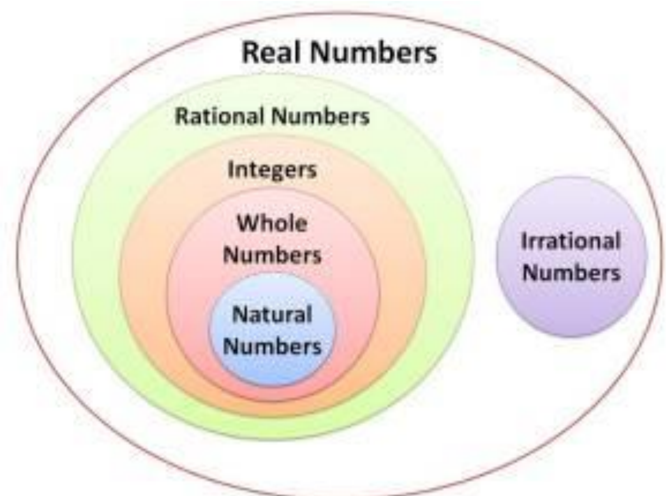


Fig. 1.Venn Diagram of Number Classification
(Source: *moreheadmathteacher.files.wordpress.com/2013/10/realnumbersvenn1.jpg*)

The number theory is a branch of mathematics which describes the property of whole numbers or integers which does not have decimal numbers. Examples of this would be 0, 22, -3, -12345, 324. Non-integer numbers, which have decimal numbers, would be for example 3.4, 2.9, -312.43, and so on.

These integers have division property which is written with the notation "a | b" meaning b is divisible by a without remainders, or that a is a multiple of b. For example, writing 4 |

12 would mean with the factor of 3, we can get 4 * 3 = 12, therefore 12 is divisible by 4, or that 4 is a multiple of 12 [1].

In general, any integer divided by another integer could be written in the form of integers. Take m and n as integers with n greater than zero, if m is divided by n, then a quotient (q) and a remainder (r) could be acquired in the form of integers, with the remainder being less than n.

$$m = nq + r$$
$$n > 0, \qquad 0 \leq r < n$$

The theorem above was founded by Euclid at around 350 BC, and the theorem is known now to be the Euclidean theorem. The notation above could be written in another form, through the usage of the *modulus* operator (mod) and the *division* operator (div) [1]. Writing the results of the quotient (q) and remainder (r) from the equation above would be as so.

$$q = m \textbf{ div } n$$
$$r = m \textbf{ mod } n$$

For example, 30 divided by 7 would result 4 with the remainder of 2, as shown below.

$$30 = 7(4) + 2$$
$$30 \textbf{ mod } 7 = 2$$
$$30 \textbf{ div } 7 = 4$$

The remainder of the equation has to stay positive, as to comply with the theorem, which means that -33 divided by 5 has to have a positive remainder, as shown below.

$$-33 = 5(-7) + 2$$
$$-33 \textbf{ mod } 5 = 2$$
$$-33 \textbf{ div } 5 = -7$$

## III. CRYPTOGRAPHY

One of the applications of number theory is cryptography. Cryptography is widely used as a way to keep a message secret. The way cryptography works is it changes a message into something that does not have any meaning. The message that wants to be kept secret is called the plaintext, and the result of the process of cryptography, also called encryption, is called ciphertext. The opposite of encryption, the process of changing ciphertext back to the original plaintext, is called decryption. as shown below (see Fig. 2).

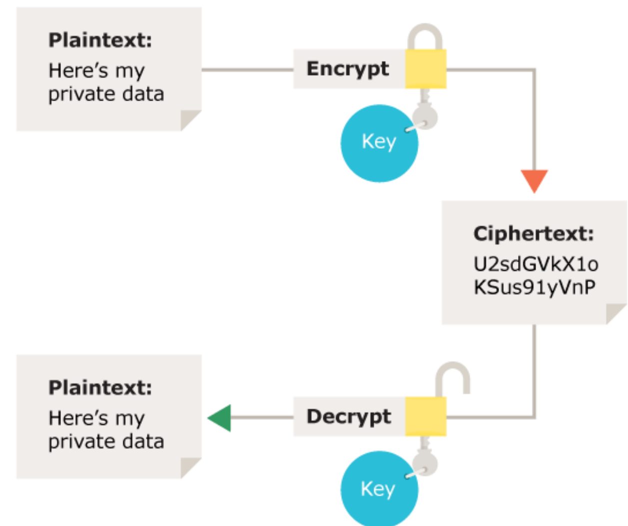


Fig 2. Symmetric-key Cryptosystem
(Source: *https://www.engadget.com/2017/03/07/a-beginners-guide-to-encryption/)*

The diagram above shows the process of encryption and decryption with the use of a same key, which is called the symmetric-key cryptosystem. The plaintext "Here's my private data" when encrypted becomes "U2sdGVkX1oKSus91yVnP", which has no meaning whatsoever to us. However, when you have the key, this piece of unknown combination can be changed back to "Here's my private data", and that is how simple encryption works. If the keys used to encrypt and decrypt are different, then the system is called the asymmetric cryptosystem. There are an abundant amount of cryptography methods, such as the Caesar cipher, which shifts each letter 3 letters forward, which changes 'A' to 'D', 'B' to 'E', and so forth [1]. In our case, cryptography is used in conjunction with the hash function SHA1 to later produce the one-time password.

## IV. HASH FUNCTION SHA1

A hash function in its very basis is used in data structuring. A hash function works by taking an input and changing it into an index for the data to be stored at. The very basic form of hashing uses the modulus operator, which generates a number based on the input, as shown below:

$$h(k) = k \textbf{ mod } m$$

where *m* is the memory provided, with an index from 0 to m-1. *h* is the hash function, with k as the input.

A more complicated hash function accepts a record, which when receives the input will generate hashes which is used to map out the location of how the data should be indexed, as shown in the illustration below.

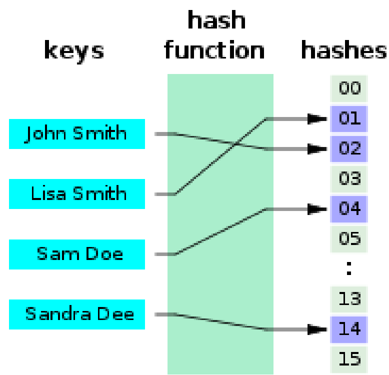


Fig. 3. Hashing using a hash function
(Source: *https://www.quora.com/What-is-the-Hashing-technique*)

This is all good and effective, until there comes the problem of a collision.

In the case of a collision, there are what is called the collision resolution policy. This policy states that if there is a collision, the record would be placed in the next available or free index, as shown below.

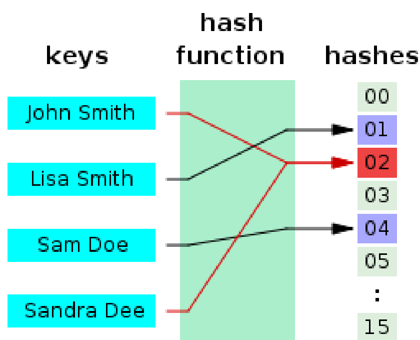


Fig. 4. Collision in a hash function
(Source: *https://en.wikipedia.org/wiki/File:Hash_table_4_1_1_0_0_1_0_LL.svg*)

In this case, the record "Sandara Dee" would instead be placed in the 03 index, because it is the next free index after 02.

The combination of cryptography and hash function would then be known as a cryptographic hash function. This type of hash function focuses on compressing an input so that the result would be shorter [2]. Also, this hash function is called cryptographic because this is a one-way function, meaning it is close to impossible to reverse the result of this hash function (also known as the "message digest"). This "message digest" would usually have a fixed length, as an example the SHA1 hash function would have a "message digest" of 160 bits, or 20 bytes. The Secure Hash Algorithm 1 or SHA1 is a type of hash function which utilizes the hashing process to create what is called the message digest. Below is the pseudocode for the SHA1 hash function.

*Note 1: All variables are unsigned 32-bit quantities and wrap modulo $2^{32}$ when calculating, except for*
*ml, the message length, which is a 64-bit quantity, and*
*hh, the message digest, which is a 160-bit quantity.*
*Note 2: All constants in this pseudo code are in big endian.*
*Within each word, the most significant byte is stored in the leftmost byte position*

*Initialize variables:*

h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
h4 = 0xC3D2E1F0

ml = message length in bits (always a multiple of the number of bits in a character).

*Pre-processing:*
append the bit '1' to the message e.g. by adding 0x80 if message length is a multiple of 8 bits.
append $0 \le k < 512$ bits '0', such that the resulting message length in *bits*
is congruent to $-64 \equiv 448 \pmod{512}$
append ml, the original message length, as a 64-bit big-endian integer.
Thus, the total length is a multiple of 512 bits.

*Process the message in successive 512-bit chunks:*
break message into 512-bit chunks
**for** each chunk
break chunk into sixteen 32-bit big-endian words w[i], $0 \le i \le 15$

*Extend the sixteen 32-bit words into eighty 32-bit words:*
**for** i **from** 16 to 79
w[i] = (w[i-3] **xor** w[i-8] **xor** w[i-14] **xor** w[i-16])
**leftrotate** 1

*Initialize hash value for this chunk:*
a = h0
b = h1
c = h2
d = h3
e = h4

*Main loop:*
**for** i **from** 0 **to** 79
**if** $0 \le i \le 19$ **then**
f = (b **and** c) **or** ((**not** b) **and** d)
k = 0x5A827999
**else if** $20 \le i \le 39$
f = b **xor** c **xor** d
k = 0x6ED9EBA1
**else if** $40 \le i \le 59$
f = (b **and** c) **or** (b **and** d) **or** (c **and** d)
k = 0x8F1BBCDC
**else if** $60 \le i \le 79$
f = b **xor** c **xor** d
k = 0xCA62C1D6

temp = (a **leftrotate** 5) + f + e + k + w[i]
e = d
d = c
c = b **leftrotate** 30
b = a
a = temp

*Add this chunk's hash to result so far:*
h0 = h0 + a
h1 = h1 + b
h2 = h2 + c
h3 = h3 + d
h4 = h4 + e

*Produce the final hash value (big-endian) as a 160-bit number:*
hh = (h0 **leftshift** 128) **or** (h1 **leftshift** 96) **or** (h2 **leftshift** 64) **or** (h3 **leftshift** 32) **or** h4

In the end, we will have the *hh* variable to be the 160 bits or 20 bytes of message digest that would be used as the result of the hash function. This is an example of the SHA1 hash function at work.

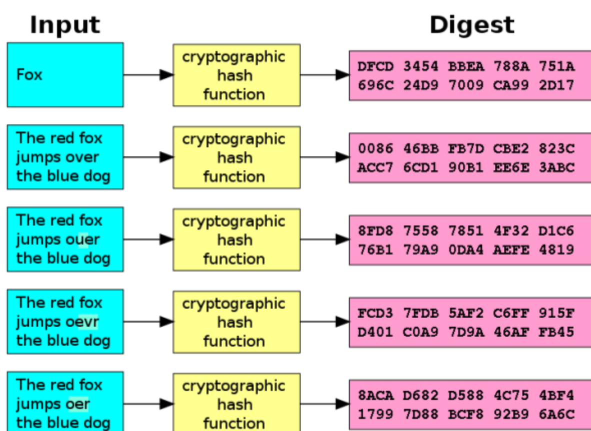


Fig. 5. SHA1 Hash Function
(Source: *https://commons.wikimedia.org/wiki/File:Cryptographic_Hash_Function.svg*)

Because of the SHA1 hash function's complexity, even a letter difference or switching would result in a quite different message digest, as seen from the second and fourth example, which has the word "over" misspelled. With the message digest, servers store them and when a password is entered, the hash function is applied to the password, and if the 160 bits string matches with the one on the server, access is granted to the user. Something that does this is also called an HMAC, or a hash-based message authentication code.

## V. Hash-Based Message Authentication Code

The hash-based message authentication code (HMAC) is a piece of information to authenticate a message, in this case the message will be the password. A secret key will be used, which has to be randomized for the sake of security, and is usually kept secret so people could not try brute-forcing because they need the key to be able to do that. Without the key, the HMAC function is simply useless. The definition of HMAC from RFC 2104 [4]:

$$HMAC(K,m) = H\Big((K' \oplus opad) \parallel H\big((K' \oplus ipad) \parallel m\big)\Big)$$

$$K' = \begin{cases} H(K) & K \text{ is larger than blocksize} \\ K & Otherwise \end{cases}$$

where
H is a cryptographic hash function, in our case the SHA1,
*m* is the message to be authenticated,
*K* is the secret key,
*K'* is a block-sized key derived from the secret key, *K*; either by padding to the right with 0s, up to the block size, or by hashing down to the block size,
∥ denotes concatenation,
⊕ denotes bitwise exclusive or (XOR),
*opad* is the outer padding, consisting of repeated bytes, valued 0x5c, up to the block size, and
*ipad* is the inner padding, consisting of repeated bytes, valued 0x36, up to the block size.

This results in a pseudocode as such:

```
Function hmac
  Inputs:
    key:        Bytes     //array of bytes
    message:    Bytes     //array of bytes to be hashed
    hash:       Function  //the hash function to use (e.g. SHA-1)
    blockSize:  Integer   //the block size of the underlying hash function (e.g. 64 bytes for SHA-1)
    outputSize: Integer   //the output size of the underlying hash function (e.g. 20 bytes for SHA-1)

  //Keys longer than blockSize are shortened by hashing them
  if (length(key) > blockSize) then
    key ← hash(key) //Key becomes outputSize bytes long

  //Keys shorter than blockSize are padded to blockSize by padding with zeros on the right
  if (length(key) < blockSize) then
    key ← Pad(key, blockSize) //pad key with zeros to make it blockSize bytes long

  o_key_pad = key xor [0x5c * blockSize]   //Outer padded key
  i_key_pad = key xor [0x36 * blockSize]   //Inner padded key

   return hash(o_key_pad ∥ hash(i_key_pad ∥ message))
//Where ∥ is concatenation
```

To better understand the HMAC algorithm, here is a diagram explaining how it works.

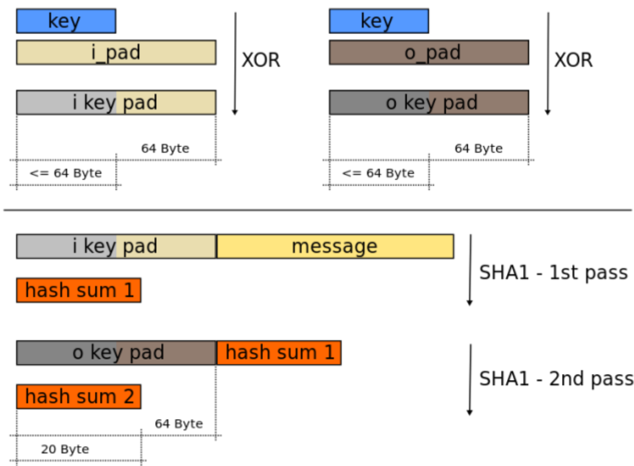


Fig. 6. HMAC based on SHA1 hash function (Source: *https://commons.wikimedia.org/wiki/File:SHAhmac.svg*)

First of all, we have the concatenation of the repeated byte 0x36 by 64 times, also known as the *ipad*, xor the key, and also the concatenation of the repeated byte 0x5c by 64 times, also known as the *opad*, xor the key. This ipad xor key will be concatenated by the message and will be used to create the first hash sum. The opad xor key will then be concatenated with the first hash sum and will be used to create the second hash sum, which will be the result of the HMAC algorithm. This HMAC algorithm will be the basis of the time-based one-time password.

## VI. Time-Based One-Time Password Algorithm

The time-based one-time password algorithm (TOTP) is based on the usage of the HMAC-based one-time password algorithm (HOTP). To understand TOTP, we must first understand HOTP and how it works.

The HOTP is an algorithm to provide authentication and generate human-readable values so as to ease the use of the one-time authentication attempt [5]. Using the HMAC algorithm from the previous section, we can essentially simplify the HOTP process into a single equation:

$$HOTP(K, C) = Truncate(HMAC(K, C))$$

where

K is the secret key, which is generated at random and only the server and the client has,

C is a counter that has to be kept in synchronized between the client and the server.

This truncate function serves as a conversion method from the 160 bit message digest to become an HOTP value, usually in the form of a six-digit number, readable for humans. This 160-bit message digest will be converted to a 4-byte string, then will be converted again to a digit somewhere between 0 and (2^31) – 1, which will then be applied the modulo operator by 10 ^ digits, which would be 10 ^ 6, resulting in a six-digit number. Here is the pseudocode for the conversion:

```
Let Sbits = DT(message digest) //  DT, defined below,
    // returns a 31-bit string

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

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
```

For example, the ASCII string "12345678901234567890" when put in the HMAC algorithm with a count of 1 will result in a hexadecimal like so, “75a48a19d4cbe100644e8ac1397eea747a2d33ab”. This will then be truncated to “41397eea”, which will be turned into a decimal form “1094287082”, and then mod by 10^6 which will result in the number “287082”, which the user will use to authenticate. As soon as the authentication process is done, the count will increase and a new HOTP will be generated based on the new count.

TOTP is based on HOTP. The only difference between TOTP and HOTP is the count variable [6]. With HOTP, the count variable has to be kept in sync with the user, and has to manually increment based on its use. On the other hand, with TOTP, the count variable is now dynamic not based on manual count, but based on Unix time, a type of time which is the number of seconds that have elapsed since 00:00:00 (UTC) of Thursday, 1 January 1970. As soon as a key is set and a handshake is created, the client and server agrees on a few things, such as the interval of the counter (X), which defaults to 30 seconds, and the Unix time to start counting ($T_0$), which defaults to 0. The TOTP value is calculated by:

$$TOTP(K) = HOTP(K, T)$$

$$T = \frac{(Current\ Unix\ time - T_0)}{X}$$

Here are the examples of TOTP which has its counter based on time.

| Time (s) | UTC Time | T (hex) | TOTP |
|---|---|---|---|
| 59 | 1970-01-01 00:00:59 | 0x1 | 287082 |
| 1111111109 | 2005-03-18 01:58:29 | 0x23523EC | 081804 |
| 20000000000 | 2603-10-11 11:33:20 | 0x27BC86AA | 353130 |

Table 1. Example of TOTP [6]

With this kind of implementation, there are benefits and compromises when compared to its predecessor, HOTP. The benefits of TOTP are first, they are very dynamic and is harder

to break through because of its 30-second changing time. It also eliminates the need of syncing count, because it is synced through time. Although, the compromise of this implementation is that once there is a significant time drift between the client and server, there needs to be a solution to resync the time, which could be done by connecting to the internet and syncing time. For minor cases, a small margin of error could be added, but not too big or it could compromise the system. This TOTP method is becoming more and more common, as we see popular sites like Google enforcing users to use their authenticator app, which is based on TOTP.

## VII. Implementation of TOTP

One of the famous apps that uses TOTP is *Google Authenticator*. For the authenticator to work, there will be a key sent by the server for the client to input, which could be in the form of a QR code or an alphanumeric string which is generated at random. The server would then ask for the six-digit code that is generated by the app to synchronize the time used. Here is an example of the app in use.

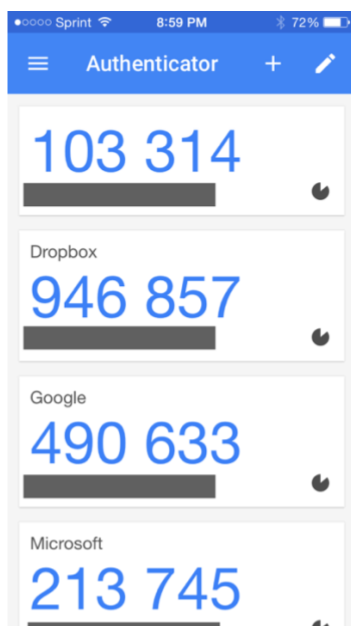


Fig. 7. Example of Google Authenticator (Source: *https://www.joshmoulin.com/wp-content/uploads/2015/07/Google-Authenticator.png*)

This image shows the six-digit one-time passwords which comes from multiple servers. These passwords also have 30-second countdowns on the right-hand corner, which when it ends, another six-digit password will appear.

## VIII. Conclusion

From the explanation above, we can now see the use of the SHA1 hash function to generate one-time passwords which are based on both time and a secret key that is agreed upon by both the client and the server. This type of authentication is used as a supplementary protection for logging in or completing a transaction.

Although time-based one-time passwords raises the security of accounts by a significant amount, clients are still susceptible to attacks like phishing, because phishing does not concern the algorithm, but instead gains access of the keys by faking the client's identity.

As for the future, with the advancement of technology and also computing power, it might be a good idea to start to use these two-factor authentication methods which uses the TOTP as it creates another barrier for people who are trying to break in to the digital privacy.

## VIII. Acknowledgment


The author would like to thank first of all God as the author was able to finish writing this paper well. The author would also like to thank lecturer Dr. Judhi Santoso, M.Sc. from the Discrete Mathematics IF2120 class for his lectures and support. Also, the author would like to express his gratitude for his family and friends for their constant support.


## References


[1] Munir, Rinaldi. *Matematika Diskrit*. 3rd ed., Penerbit INFORMATIKA Bandung, 2010.

[2] Christensson, Per. "Hash Definition." *TechTerms*. Sharpened Productions, 21 April 2018. Web, https://techterms.com/definition/hash. Accessed 8 Dec. 2018.

[3] Mihir, Bellare. "Hash Functions." University of California, 2018, cseweb.ucsd.edu/~mihir/cse207/w-hash.pdf. Accessed 8 Dec. 2018.

[4] Krawczyk, H, et al. "HMAC: Keyed-Hashing for Message Authentication." *IETF Tools*, Feb. 1997, tools.ietf.org/html/rfc2104. Accessed 8 Dec. 2018.

[5] M'Raihi, D, et al. "HOTP: An HMAC-Based One-Time Password Algorithm." *IETF Tools*, Dec. 2005, tools.ietf.org/html/rfc4226. Accessed 8 Dec. 2018.

[6] M'Raihi, D, et al. "TOTP: Time-Based One-Time Password Algorithm." *IETF Tools*, May 2011, tools.ietf.org/html/rfc6238. Accessed 8 Dec. 2018.


## Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 9 Desember 2018

Kevin Nathaniel Wijaya
13517072