1 HMAC

A Hash-based Message Authentication Code (HMAC) can be used to determine whether a message sent over an insecure channel has been tampered with, provided that the sender and receiver share a secret key. The sender computes the hash value for the original data and sends both the original data and the HMAC as a single message. The receiver recomputes the hash value on the received message and checks that the computed hash value matches the transmitted hash value. HMAC can be used with any iterative cryptographic hash function, such as MD5 or SHA-1, in combination with a secret shared key. The cryptographic strength of HMAC depends on the properties of the underlying hash function. The real life example of HMAC is SSL.

\[ \text{HMAC}_K(m) = \text{NMAC}_{k_1,k_2}(m) \]

where
\( k1 = h(\overline{k} \oplus opad) \) and \\
\( k2 = h(\overline{k} \oplus ipad) \)

The default values for ipad is 0110110(0x36) and opad is 1011100(0x5c) repeated \( b/8 \) times where \( b \) is Block size.

So,

\[ HMAC_K(m) = H(\overline{k} \oplus opad, H(\overline{k} \oplus ipad, m)) \]

Here, \( \overline{k} = pad(k) \).

Note that HMAC is quite efficient as it only requires 3 more compression calls than \( H(x) \). This makes HMAC a very fast construction.

## 2 Public Key Cryptography

In case of Public Key Cryptography both the sender and receiver has a pair of secret key and public key \((S_{KA}, P_{KB})\) and it is assumed that everybody knows each other public key. Here A sends the encrypted message to B using B’s public key and B can decrypt the message by using his/her private key. The mechanism of Pubic Key encryption is opposite to that of Digital Signature, in case of Digital Signature A signs the message using his/her private key and sends the signed message along with the original message to B where B can verify using A’s public key. Here the public key is a function of Private key/secret key for every entity. If having the knowledge of public key it should be hard for someone to know its corresponding private key.
In order to understand Public Key Encryption we must have the knowledge of Basic Number Theory.

2.1 Number Theory

2.1.1 Groups

Definition 1 \((G, \cdot)\) (where \(G\) is a set and \(\cdot : G \times G \rightarrow G\)) is called a group if the following properties are satisfied:

1. **Closure**: \(\forall a,b \in G, \ a \cdot b \in G\)

2. **Associativity**: \(\forall a,b,c \in G, \ (a \cdot b) \cdot c = a \cdot (b \cdot c)\)

3. **Identity**: \(\exists\) an identity element \(e \in G\) such that \(\forall a \in G, \ a \cdot b = b \cdot a = a\)

4. **Inverse**: \(\exists\) an element \(a^{-1} \in G\) such that \(\forall a \in G, \ a \cdot a^{-1} = a^{-1} \cdot a = e\)

2.1.2 Group: Examples

We know

- \(Z\) = Set of all integers
- \(Z_m\) = Set of all whole numbers (modulo \(m\))
- \(Z_p\) = Set of all natural numbers (modulo \(p\)) which are relatively prime to \(p\).
- \(Z_n^*\) = Set of all numbers which are relatively prime to \(n\), where \(n=p\cdot q\).

Let us consider whether or not the following sets are groups:

1. \((Z, \text{“addition”})\) is a group.

2. \((Z, \text{“multiplication”})\) isn’t a group because not all integers have a multiplicative inverse.

3. \((Z_m, \text{“modular addition”})\) is a group.

4. \((Z_m, \text{“modular multiplication”})\) isn’t a group, again due to the lack of inverses for all integers in the set.

5. \((Z_p, \text{“modular multiplication”})\) is a group.
6. \((Z_n^*, \text{“modular addition”})\) isn’t a group because the addition of two integers that are relatively prime to \(n\) may produce an integer which is not relatively prime to \(n\). Thus the set lacks closure.

7. \((Z_n^*, \text{“modular multiplication”})\) is a group.

### 2.1.3 Modular Arithmetic

1. **Modulo:**\((a \mod N)\) where

\[
a = (a_{p-1}, a_{p-2}, \ldots, a_1, a_0)
\]

\[
N = (N_{q-1}, N_{q-2}, \ldots, N_1, N_0)
\]

Thus,

\[
a = (a_{p-1} \cdot 2^{p-1} + a_{p-2} \cdot 2^{p-2} + \ldots + a_1 \cdot 2^1 + a_0 \cdot 2^0)
\]

\[
N = (N_{q-1} \cdot 2^{q-1} + N_{q-2} \cdot 2^{q-2} + \ldots + N_1 \cdot 2^1 + N_0 \cdot 2^0)
\]

So the runtime complexity of \((a \mod N)\) is \(O(|a| \cdot |N|)\).

2. **Modular Addition:** \((a + b) \mod N\) has time complexity of \(O(max(|a|, |b|)) + O(max(|a|, |b|) \cdot |N|))\); using addition and then modulo operation.

   In modular addition \(a < N\), \(b < N\) and thus \(a + b < 2N\). So,

   \[
   (a + b) \mod N = (a + b - N)
   \]

   Taking \(|a| = |b| = |N|\), runtime complexity of \((a + b) \mod N\) is \(O(|N|)\).

3. **Modular Multiplication:** \((ab \mod N)\) has running time complexity of \(O(|a| \cdot |b|) + O((|a| + |b|) \cdot |N|))\).

   In modular multiplication, \(a < N\), \(b < N\) and thus \(ab < N^2\). Taking \(|a| = |b| = |N|\), the runtime complexity of \((ab \mod N)\) is \(O(|N|^2)\).

4. **Modular Exponentiation:**

   \((a^n \mod N)\) has a runtime complexity of \(O(|n| \cdot |a| \cdot |N|)\)

   Taking \(|a| = |N|\), the runtime complexity of \((a^n \mod N)\) is \(O(|n| \cdot |N|^2)\)

   The usual approach to computing \(a^n \mod N\) is inefficient when \(n\) is large. Instead, \(n\) can be represented as a bit string \(n = [n_{k-1}, n_{k-2}, n_{k-3}, \ldots, n_1, n_0]\) and input to the square-and-multiply algorithm below to efficiently compute the result of a modular exponentiation operation.
\[ z = 1 \]
\[ \text{for } i = k - 1 \text{ down to } 0 \text{ do} \]
\[ z = z^2 \mod N \]
\[ \text{if } (n_i == 1) \]
\[ z = (z \cdot a \mod N) \]

This algorithm has a runtime complexity of \( O(|n| \cdot |a| \cdot |N|) \). Taking \( |a| = |N| \), the runtime complexity becomes \( O(|n| \cdot |N|^2) \) This can further be explained as:

\[ Z = a^{64} \mod N \]

\[ Z_1 = a \mod N \]

\[ Z_2 = Z_1 \cdot Z_1 \mod N = a^4 \]

\[ Z_3 = Z_2 \cdot Z_2 \mod N = a^8 \]

\[ Z_4 = Z_3 \cdot Z_3 \mod N = a^{16} \]

\[ Z_5 = Z_4 \cdot Z_4 \mod N = a^{32} \]

\[ Z_6 = Z_6 \mod N = a^{64} \]

5. **Greatest Common Divisor:**

Given \( a, b \in \mathbb{N} \), \( \gcd(a, \ b) \) represents the greatest common divisor that these two numbers share. The naive solution to this problem would be to simply try all values from 1 to \( \max(a/2, \ b/2) \). For numbers that may be hundreds of digits long, this is not feasible. Therefore, we will improve upon this algorithm by using the Euclidean Algorithm. Before we introduce this algorithm, we must prove a related fact.

Claim: \( \gcd(a, b) = \gcd(a - b, b) \)

Proof: To show this, we will break the proof into 2 parts. We show that

(1) \( \gcd(a, b) \leq \gcd(a-b, b) \)

(2) \( \gcd(a, b) \geq \gcd(a-b, b) \)

Proof of (1)

We define \( CD_{a, b} = (k \in \mathbb{N} : k | a \text{ and } k | b) \), this is the set of all common divisors of \( a \) and \( b \). Similarly, \( CD_{a-b, b} = (k \in \mathbb{N} : k | (a-b) \text{ and } k | b) \).

Consider some \( k \) in \( CD_{a,b} \).

We know that: \( a = kA \)

\( b = kB \) for some \( A,B \in \mathbb{N} \). We therefore know that: \( a - b = kA-kB = k(A-B) \)

We can therefore conclude that \( k \) belongs to \( CD_{a-b, b} \). Therefore, \( CD_{a,b} \succeq CD_{a-b, b} \). We therefore see that the largest elements of \( CD_{a,b} \) must be at most as large as the largest element of \( CD_{a-b, b} \). We therefore conclude Equation 1 is true.

b) Proof of (2)
We now consider some element \( k \) belongs to \( CD_{a-b,b} \). We know that \( a-b = kA = kB \)

Therefore we can see that: \( a = kA + kB = k(A + B) \)

We therefore know that if some element \( k \) is in \( CD_{a-b,b} \), then it is also in \( CD_{a,b} \), meaning \( CD_{a-b,b} \) is a subset of \( CD_{a,b} \). We can therefore conclude Equation 2 is true.

From (1) and (2), we know that \( CD_{a,b} = CD_{a-b,b} \) and we also see that: \( gcd(a, b) = gcd(a-b, b) \)

We see a corollary of this result is: \( gcd(a, b) = gcd(a \mod b, b) \) for \( a > b \). In order to find the gcd of two arbitrary numbers, we will recursively use the above formula.

Example: We will use the Euclidean algorithm to find \( gcd(37, 15) \). We see that \( 37 \mod 15 = 7 \).

So, \( gcd(37, 15) = gcd(7, 15) \) And \( 15 \mod 7 = 1 \) hence,

\[
gcd(7, 15) = gcd(7, 1) = 1
\]

The running time of this algorithm will be \( O(N^2) \). We do not provide a complete analysis here, but we can see that we will be recursively cutting the size of the input in half at every step (one can easily show that \( a \mod b \leq a=2 \)).

And we see that we are doing \( O(N^2) \) work, because each step requires a gcd computation.

6. Modular Inverse:

We are given \( a \) (which belongs to \( N \)), and we want to find \( a^{-1} \mod N \). Because we trying to find \( Z = a^{-1} \mod N \) this is equivalent to saying \( Za = 1 \mod N \). Which can further be resolved as \( Z = kN + 1 \) also stated as \( Z \mod K = 1 \)

Note: Such an inverse only occurs when \( a \) and \( N \) are relatively prime, so \( gcd(a,N) = 1 \).

To find the inverse, we use the extended Euclidean algorithm, as we will demonstrate in the following example.

Example: \( 15^{-1} \mod 37 \)

We begin as we would with the standard Euclidean algorithm for \( gcd \): \( gcd(37,15) \)

(a) \( 37 = 2 * 15 + 7 \)

(b) \( 15 = 2 * 7 + 1 \) This is the step we are interested in, because the remainder is 1. \( 15 = 2*7+1 \implies 15-2*7=1 \implies 7 = 37-15*2 \) (from step 1) 1.

\( 5*15 - 2*37 = 1 \) This is the form we described before \( (xa - kN = 1) \), so we conclude that \( x = 15^{-1} \mod 37 = 5 \).

Example: \( 13^{-1} \mod 44 \)
(a) $44 = 3\cdot13 + 5$
(b) $13 = 2\cdot5 + 3$
(c) $5 = 1\cdot3 + 2$
(d) $3 = 1\cdot2 + 1$ Now we have got remainder 1 so
$3 - 1\cdot2 = 1$
$3 - 1\cdot(5 - 1\cdot3) = 1$ ($5 - 1\cdot3 = 2$ from c)
$2\cdot3 - 1\cdot5 = 1$
$2\cdot(13 - 2\cdot5) - 1\cdot5 = 1$ ($13 - 2\cdot5 = 3$ from b)
$2\cdot13 - 5\cdot5 = 1$
$2\cdot13 - 5\cdot(44 - 3\cdot13) = 1$ ($44 - 3\cdot13 = 5$ from a)
$17\cdot13 - 5\cdot44 = 1$

Now we can describe the above equation in the form: $xa - kN = 1$, so we can conclude that $x = 13^{-1} \mod 44 = 17$

The complexity of modular inverse is same as that of GCD algorithm as we are running GCD first and than backtracking it.