STUDI MENGENAI SALT

Tania Krisanty – 13504101

Laboratorium Ilmu Rekayasa dan Komputasi
Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung
Jalan. Ganesha 10, Bandung
e-mail : if14101@students.if.itb.ac.id

Abstrak

Makalah ini akan membahas tentang aplikasi *salt* dalam kriptografi. *Salt* adalah bit-bit acak yang ditambahkan ke input dalam proses derivasi kunci. *Salt* berfungsi untuk memperkuat keamanan data yang terenkripsi, terutama dari *dictionary-attack*. Secara garis besar, makalah ini akan membahas proses pembangkitan *salt* yang melibatkan pembangkitan bilangan acak semu, proses enkripsi data (umumnya *password*) yang dikonkatenasi dengan *salt*, contoh aplikasi *salt*, analisis dan pembahasan keuntungan dan/atau kerugian adanya *salt* dalam enkripsi proses berbasis password.

Kata kunci: *cryptography*, *dictionary-attack*, enkripsi, enkripsi, *password*, *pseudo-random number generator*, *salt*.

1. Pendahuluan

Seiring dengan perkembangan teknologi informasi dan komunikasi yang semakin mendukung pertukaran data dari dan ke mana saja melalui jaringan internet, usaha-usaha untuk mengusik pertukaran data tersebut juga semakin berkembang. Usaha tersebut sangat bervariasi, mulai dari menggagalkan proses transfer data, menyadap proses transfer data, sampai memodifikasi data tersebut selama proses.

Beberapa jenis usaha mungkin dilakukan untuk menguji keamanan proses pertukaran data, sehingga kelemahan yang ditemukan dapat menjadi bahan pembelajaran untuk meningkatkan kekuatan proses. Sebaliknya, beberapa jenis usaha lainnya mungkin dilakukan semata-mata untuk kepentingan pribadi si pelaku. Sangat disayangkan, sampai sekarang ini usaha negatif inilah yang sering dilakukan. Tujuan dari usaha ini dapat bermacam-macam, misalnya untuk sekedar kepuasan pribadi karena mengetahui data rahasia milik orang lain atau karena berhasil mematahkan kekokohan proses pengiriman data, maupun untuk dapat menyalahgunakan data rahasia yang didapat. Hal ini tentunya sangat mengganggu terutama bagi pihak yang memiliki wewenang atas data.

Berbagai upaya dilakukan untuk menghindari jatuhnya data ke tangan pihak yang tak berwenang agar tidak disalahgunakan, salah satunya dengan memodifikasi data dengan suatu kode yang hanya dimiliki pihak berwenang, seperti *digital signature* yang menambahkan data dengan hasil *hash*nya. Dengan adanya *digital signature*, pihak penerima dapat memeriksa autentikasi data, yaitu bahwa data tersebut benar-benar dibuat oleh pihak yang berwenang mengirimkannya tanpa diutak-atik oleh pihak lain.

Selain itu terdapat juga *salt* yang digunakan untuk memperkuat keamanan data terenkripsi. *Salt* merupakan sebuah string yang dibangkitkan secara acak untuk ditambahkan ke data sebelum proses enkripsi. Dengan adanya *salt*, serangan dengan mencocokkan potongan *ciphertext* yang diketahui ke sampel potongan *ciphertext* dapat dihindari. *Salt* terutama digunakan untuk menghindari *dictionary attack*, sebuah metode yang umum digunakan untuk mencuri *password*. Dalam metode ini, penyerang membuat daftar *password* yang umum digunakan lalu mengenkripsinya. Kemudian penyerang mencocokkannya dengan file berisi *password* terenkripsi yang disimpan di *database* atau *storage* milik pihak yang berwenang atas data. Jika terdapat kecocokan data yang terenkripsi, penyerang dapat mengetahui *password* dengan cara mendekripsinya kembali. Adanya *salt* akan menambahkan kesulitan pada pencocokan hasil enkripsi, karena *password* yang sama tidak selalu menghasilkan *ciphertext* yang sama, meskipun dienkripsi dengan algoritma enkripsi, kunci, dan *initialization vector* yang sama.

Dalam dunia nyata, terdapat banyak sekali contoh aplikasi yang memanfaatkan data rahasia yang dimiliki sebuah pihak, beberapa di antaranya adalah transaksi keuangan menggunakan kartu debit atau kredit, dan *login* ke *account e-mail*. Kedua contoh di atas melibatkan data yang sifatnya pribadi yaitu *PIN (Personal Identification Number)* dan *password*. Dengan mengetahui data pribadi tersebut, seseorang yang tidak berwenang dapat memanfaatkannya untuk melakukan transaksi atau *login* atas nama si pemilik data. Seseorang lain dapat melakukan pembelian produk secara online dengan menggunakan nomor kartu dan *PIN* tanpa diketahui si pemilik kartu, tiba-tiba si pemilik kartu telah mendapatkan data bukti transaksi yang tidak pernah ia lakukan. *Password e-mail* dapat disalahgunakan untuk memasuki *account e-mail* pribadi orang lain, bahkan dapat digunakan untuk mengirimkan *e-mail* lain atas nama si pemilik *account*.

Mengingat sangat pentingnya data-data tersebut, diperlukan suatu metode yang dapat memperkuat keamanan data, terutama dalam pengiriman dan penyimpanan. Salah satunya yang terbukti efektif adalah *salt*.

2. Pembangkitan SALT

Inti dari pembangkitan *salt* adalah bagaimana penambahan *salt* dapat mengenkripsi beberapa data yang sama menjadi beberapa data terenkripsi yang berbeda, terutama untuk mencegah *dictionary attack*. Oleh karena itu, *salt* haruslah dibangkitkan secara acak agar setiap data memperoleh *salt* yang berbeda, sehingga ketika terjadi kesamaan data, tidak terjadi pula kesamaan hasil enkripsinya, karena komponen *salt*nya berbeda.

Pembangkitan salt sepenuhnya bergantung kepada metode pembangkitan bilangan acak. Terdapat dua metode dasar yang digunakan untuk membangkitkan bilangan acak. Metode pertama dilakukan dengan mengukur suatu fenomena fisik yang acak lalu membangkitkan bilangan acak dengan memperhatikan bias yang mungkin terjadi selama proses pengukuran. Fenomena fisik adalah satu-satunya hal yang diakui sebagai *”true” random* dan selama ini tidak dapat diperkirakan. Fenomena yang dapat digunakan untuk pembangkitan bilangan acak merupakan fenomena yang berdasarkan pada ketidakseragaman atomik atau subatomik yang dapat diketahui dengan pendekatan mekanika kuantum, seperti peluruhan radioaktif, *thermal noise, shot noise,* dan *clock drift*.

Metode lainnya adalah algoritma komputasi yang menghasilkan satu seri bilangan ”acak” yang ditentukan oleh *seed* atau kunci pada awal pembangkitan. Metode yang memanfaatkan algoritma komputasi inilah yang dikenal sebagai *pseudo-random number generator*. Bilangan yang dihasilkan dengan metode ini bersifat periodik, akan muncul kembali setelah beberapa waktu.

Seperti sebuah kutipan terkenal dari John von Neumann, “Anyone who uses arithmetic methods to produce random numbers is in a state of sin”, sebuah “*random number generator*” yang bergantung sepenuhnya pada komputasi deterministik tidak dapat dinyatakan sebagai “*true” random number generator*, karena hasil pembangkitan bilangannya dapat diperkirakan. Membedakan bilangan hasil keluaran dari *pseudo-random number generator* dengan *“true” random number* sangat sulit, namun pemilihan dan penggunaan *pseudo-random number generator* yang terbaik dapat mengatasi masalah pembangkitan bilangan acak dalam berbagai aplikasi. Analisis statistik yang tepat terhadap hasil keluaran *pseudo-random number generator* juga sering sekali digunakan untuk menguji ketangguhan algoritma pembangkitannya.

Banyak bahasa pemrograman yang menyediakan fungsi-fungsi atau *library* untuk membangkitkan bilangan acak. Fungsi-fungsi atau *library* tersebut didesain untuk membangkitkan *byte* atau *word* acak, atau bilangan real antara 0 sampai 1. *Library* tersebut sering kali dapat dibuktikan buruk oleh data statistik, misalnya hasil keluaran akan berulang setelah puluhan atau ribuan percobaan pembangkitan. Pembangkitan bilangan pada fungsi tersebut umumnya menggunakan waktu pada komputer sebagai *seed*. Fungsi bawaan bahasa pemrograman ini dapat memberikan performansi yang cukup baik pada beberapa *task* seperti video game sederhana, namun tidak pada *task* yang berkaitan dengan bilangan “true” *random* seperti aplikasi kriptografi, analisis statistik, maupun analisis numerik. Beberapa contoh sistem operasi yang telah menyediakan fungsi pembangkitan bilangan yang lebih mendekati “*true” random number,* seperti /dev/random pada berbagai jenis BSD, Linux, Mac OS-X dan Solaris, fungsi CryptGenRandom dari Cryptographic Application Programming Interface pada Microsoft Windows. Pada bahasa pemrograman Java juga terdapat kelas SecureRandom.

Berikut dua contoh kode pembangkitan password secara acak yang naif dalam bahasa C dan PHP:

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int length = 8;
    int r, i;
    char c;
    srand((unsigned int) time (0));

    for(i = 0; i < length; i++)
    {
        r = rand() + 33;
        c = (char) r;
        printf ("%c", c);
    }
}
```

Pada kode di atas, fungsi standar pada bahasa C, yang merupakan *pseudo-random number generator*, `rand` menggunakan *seed* dari fungsi `time`. Menurut standar ANSI C, `time` mengembalikan nilai bertipe `time t`, yang umumnya merupakan sebuah bilangan integer 32-bit berisi jumlah detik hingga saat ini sejak 1 Januari 1970. Terdapat sekitar 31 juta detik dalam satu tahun, sehingga penyerang yang telah mengetahui tahun berapa *password* tersebut dibangkitkan akan mendapatkan sejumlah bilangan yang relatif sedikit untuk diuji lebih lanjut. Pada kasus di mana si penyerang juga memiliki *password* yang terenkripsi, pengujian dapat dipersingkat dan pencurian *password* dapat terjadi lebih cepat.

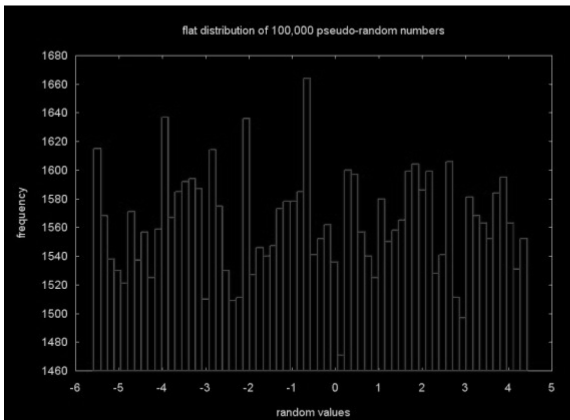
Selain itu, fungsi `rand` di atas, layaknya setiap *pseudo-random number generator,* memiliki kelemahan lain, yaitu adanya memori internal atau *state*. Ukuran *state* menentukan banyak maksimal bilangan berbeda yang dapat dibangkitkannya. Sebuah *state* *n*-bit dapat membangkitkan paling banyak 2^n> bilangan yang berbeda. Dalam banyak sistem, `rand` memiliki *state* 31-bit atau 32-bit. Hal ini yang semakin memperkecil faktor keamanan dengan menggunakan fungsi `rand`. Bahkan pada Microsoft Windows, `rand` memiliki *state* 15-bit, hanya mungkin mengeluarkan 32.767 bilangan yang berbeda.

```
function pass_gen($len)
{
    $pass = '';
    srand((float) microtime() *
    10000000);
    for($i=0; $i<$len; $i++)
    {
        $pass .= chr(rand(33, 126));
    }
    return $pass;
}
```

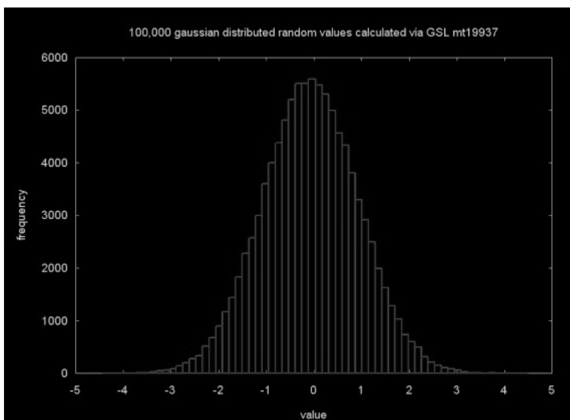
Pada contoh ke dua, digunakan fungsi `microtime` dari PHP yang mengembalikan *timestamp* Unix saat ini dalam *microseconds*. Hal ini meningkatkan jumlah kemungkinan, namun data mengenai waktu *password* tersebut dibangkitkan, misalnya *password* selalu dibuat bersamaan dengan hari seorang karyawan mulai bekerja di suatu instansi, dapat membantu memperkecil lingkup pencarian. Selain itu, beberapa sistem operasi tidak menyediakan tipe waktu dalam *microsecond*, sehingga semakin mengurangi cakupan pencarian. Fungsi `rand` yang digunakan juga menggunakan fungsi `rand` pada C, sehingga kelemahan-kelemahan pada contoh pertama juga terjadi pada contoh ke dua ini.

Selain metode-metode di atas, Bilangan *random* yang terdistribusi merata antara 0 dan 1 dapat digunakan untuk membangkitkan bilangan random lain dengan melakukan fungsi invers distribusi kumulatif dari distribusi yang diinginkan. Untuk membangkitkan sepasang *standard normally distributed random number* yang saling independen (x, y), pertama-tama perlu dibangkitkan koordinar polar (r, θ), di mana $r\sim\chi_2^2$ dan $\theta\sim$UNIFORM$(0,2\pi)$. Hasil keluaran dari beberapa *random number generator* dapat dikombinasikan, misalnya dengan *bit-wise* XOR.

Fungsi *pseudo-random number generator* yang terdapat di banyak *math libraries* pada komputer standar menghasilkan nilai yang distribusinya *flat.* Jika banyak bilangan *random* yang dibangkitkan dalam suatu wilayah (misalnya [0,1), di mana nilainya yaitu v berada dalam *range* 0 <= v <= 1) lalu digambarkan dalam sebuah histogram, histogram tersebut akan membentuk blok seperti histogram di bawah ini, di mana bilangan dibangkitkan dalam *range* [5.5, 4.5):



Sejumlah aplikasi seperti pembangkitan *Brownian random walks* memerlukan bilangan *random* yang berada dalam distribusi Gaussian (misalnya sebuah kurva lonceng). Sebuah contoh dari bilangan random yang tersebar oleh distribusi Gaussian (di mana terdapat mean 0 dan standar deviasi 1) digambarkan di bawah ini:



Untuk mengkonversi suatu bilangan *random* dalam distribusi flat ke dalam distribusi Gaussian, dinyatakan sebuah persamaan yang terdapat dalam buku *Chaos and Fractals* oleh Peitgen, berikut penjelasannya:

Gaussian distribusi terjadi di mana seluruh kejadian *random* yang independen dan terdistribusi identik dirata-rata atau dijumlahkan. Hal ini adalah konteks dari teorema matematik yang dikenal dengan nama *the central limit theorem.*

Sebuah contoh *"random event"* atau kejadian *random* adalah pelemparan enam buah dadu bersisi enam. Seluruh nilai dadu ditambahkan menghasilkan nilai dengan jangkauan 6 sampai 36. Jika terjadi 10.000 lemparan, distribusi nilai akan berada dalam kurva Gaussian. Prinsip dari menambahkan sejumlah *random events* digunakan oleh Peitgen untuk mengkonversi nilai dari *uniform random number generator* secara kasar ke dalam distribusi Gaussian menggunakan rumus di bawah ini:

$$D = \frac{1}{A}\sqrt{\frac{12}{n}}\left(\sum_{i=1}^{n} Y_i\right) - \sqrt{3n}$$

D: bilangan random Gaussian

A: batas atas dari random number generator, yang mengembalikan 0, 1, ... *A*

n: jumlah *event* yang independen, misalnya dadu

Y_1, Y_2, ... Y_n: hasil dari *event* yang independen, misalnya lemparan dadu

Berikut kode header file yang berfungsi membangkitkan bilangan random dalam distribusi flat:

```
#ifndef RAND_FLAT_H
#define RAND_FALT_H

#include "rand_base.h"

#include "gsl/gsl_randist.h"


/**
 * Membangkitkan bilangan random
 * dalam distribusi flat.
 * Konstruktor kelas memiliki
 * parameter nilai seed dan batas
 * atas dan bawah untuk distribusi
 * flat. Bilangan random yang
 * dibangkitkan akan berada dalam
 * jangkauan:
 * lower <= randVal < upper
 */
class rand_flat : public rand_base
{
 private:
  double lower_, upper_;

 public:
  rand_flat( int seedVal,
             double lower,
             double upper ) :
  rand_base( seedVal ),
  lower_(lower),
  upper_(upper) {}
```

```
  double nextRandVal()
  {
   return gsl_ran_flat
   ( state(), lower_, upper_ );
  }
}; // rand_gauss

#endif
```

Berikut kode header file yang berfungsi membangkitkan bilangan random dalam distribusi Gaussian:

```
#ifndef RAND_GAUSS_H
#define RAND_GAUSS_H

#include "rand_base.h"

#include "gsl/gsl_randist.h"

/**
 * Membangkitkan bilangan random
 * dalam distribusi Gaussian.
 * Konstruktor kelas memiliki
 * parameter nilai seed untuk random
 * number generator.
 */
class rand_gauss : public rand_base
{
public:
 rand_gauss( int seedVal ) :
 rand_base( seedVal )
 {

 }

 double nextRandVal()
 {
  return gsl_ran_ugaussian(state());
 }
}; // rand_gauss

#endif
```

Berikut file yang digunakan oleh kedua file di atas:

```
#ifndef RAND_BASE_H
#define RAND_BASE_H

//
// GNU Scientific Library includes
//
#include "gsl/gsl_rng.h"

class rand_base
{
 private:
  gsl_rng *rStatePtr_;

 private:
  rand_base( const rand_base &rhs );

 protected:
  gsl_rng *state()
  {
   return rStatePtr_;
  }

 public:
  rand_base( int seedVal )
  {
    const gsl_rng_type *T;

    T = gsl_rng_env_setup();

    rStatePtr_ = gsl_rng_alloc( T );

    gsl_rng_set( rStatePtr_, 127 );
  } // rand_base constructor

  ~rand_base()
  {
    gsl_rng_free( rStatePtr_ );
  } // rand_base destructor

  virtual double nextRandVal() = 0;

}; // rand_base

#endif
```

Keacakan suatu bilangan yang dihasilkan dengan suatu *random number generator* dapat diujikan dengan *theoretical tests*, atau metode lainnya yaitu *empirical ("statistical") tests*. Tes yang ke dua inilah yang lebih sesuai untuk aplikasi kriptografi.

Pada pembangkitan bilangan *random*, algoritma pembangkitannya dapat dianalisis secara teoritis. Untuk melakukannya digunakan konsep *discrepancy* atau *spectral test*. *Discrepancy* tidak dapat dikomputasi dalam dimensi yang lebih tinggi, misalnya di atas 2, karena kompleksitas komputasinya terlalu tinggi.

Berikut salah satu contoh tabel hasil perbandingan *theoretical tests* dengan *empirical tests*.

Theoretical Tests	LCG	ICG	EICG
Discrepancy	-,+	+	-,+

Spectral Test	-,+	not def.	not def.
Weighted Spectral Test	-,+	+	+

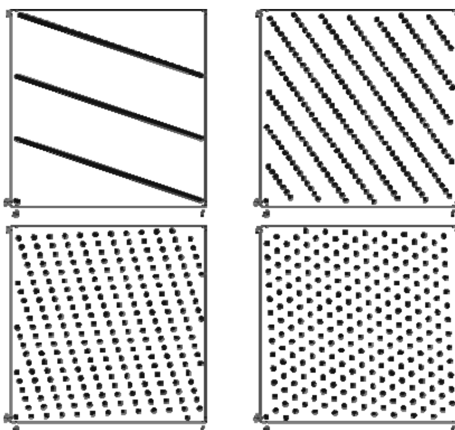
Empirical Tests	LCG	ICG	EICG
Serial Test	-	+	+
Overlapping Serial Test	-	+	+
Run Test	-,+	+	+

Estimasi *Discrepancy* untuk sebuah *linear congruential generator (LCG)* tidak diketahui akan menghasilkan nilai yang tidak diketahui pula. Fakta-fakta tersebut dilambangkan dengan tanda -,+ dan + pada tabel. Jika diketahui *inversive congruential generators (ICG)* dan *explicit-inversive congruential generators (EICG)*, *discrepancy* dapat diestimasi.

Spectral test merupakan tes yang paling penting dalam memilih parameter untuk *linear type* dari *random number generator* seperti *LCG* dan *multiple recursive generators (MRG)*. *Spectral test* sangat efisien, namun memerlukan *random number generator* yang membangkitkan struktur *lattice* dalam dimensi yang lebih tinggi. Oleh karena itu, *spectral test* tidak didefinisikan untuk *nonlinear generators* seperti *ICG* dan *EICG*.

Contoh *spectral test*:

Terdapat *"`baby"'* *LCG*(256,a,1,0) dengan a = 85, 101, 61, 237. *LCG* menunjukkan hasil *spectral tests* d_2 = 0.3162, 0.1162, 0.0790, 0.3162, 0.1162, 0.0790, 0.0632 dan *spectral tests* yang telah dinormalisasi dalam dua dimensi s_2 = 0.1839, 0.5003, 0.7357, 0.9196, set L_2 akan membangkitkan struktur *lattice* seperti di bawah ini:



Berikut beberapa algoritma *pseudo-random number generator*:

- Blum-Blum-Shub *pseudo-random number generator*
- ISAAC
- Lagged Fibonacci *generator*
- *Linear congruential generator*. Algoritma ini paling umum digunakan dalam pemrograman komputer.
- *Linear feedback shift register*
- Mersenne *twister*

Berikut beberapa algoritma *cryptographic pseudo-random number generator*:

Algoritma enkripsi dan *fungsi* hash dapat digunakan untuk juga sebagai *pseudo-random number generator*, antara lain:

- *Block ciphers* dalam *counter mode*
- *Cryptographic hashes* dalam *counter mode*
- *Stream ciphers*

Berikut beberapa algoritma *true random number generator*:

Beberapa di antara algoritma di bawah ini mengandung *information entropy*, sehingga dapat dikatakan sebagai *true random number generator*.

- CryptGenRandom - Microsoft Windows
- Fortuna
- kelas SecureRandom dalam bahasa pemrograman
- Yarrow - Mac OS X and FreeBSD
- /dev/random - Linux and Unix

Bagaimanapun, random number generator pun dapat mengalami serangan. Beberapa jenis serangan yang umum dipakai antara lain:

Serangan terhadap perangkat lunak *random number generator*. Jika penyerang mendapatkan sebagian besar bit acak dari *stream*, akan sangat membantu baginya untuk menghitung atau mencari bit-bit sisanya. Jika penyerang mengamati *state* internal dari *random number generator*, penyerang tersebut akan dapat bekerja *backward* lalu mendeduksi beberapa nilai *random* selama proses. Jika penyerang mengamati *state* internal dari *random number generator*, akan memungkinkan baginya untuk memprediksi keluaran sampai entropi tertentu

diperoleh. Bagaimanapun, jika entropi ditambahkan secara inkremental, penyerang mungkin dapat mendeduksi nilai bit *random* yang ditambahkan lalu memperoleh state internal yang baru dari *random number generator (a state compromise extension attack)*. Jika seorang penyerang dapat mengendalikan masukan *random* ke *generator*, maka ia dapat mem-*flush* seluruh entropi yang tersisa dari sistem dan menempatkan di *state* yang diketahui. Ketika *generator* mulai menyala, seringkali *generator* tersebut memiliki sedikit entropi (atau bahkan tidak memiliki entropi sama sekali, terutama jika komputer baru saja melalui proses *booting*), sehingga si penyerang dapat memperoleh tebakan awal pada suatu *state*.

Serangan pada perangkat keras *random number generator* sangatlah mungkin terjadi, di antaranya dengan menangkap emisi pada frekuensi radio dari komputer (misalnya mendapatkan *hard drive interrupt times* dari *motor noise*), atau memberi umpan pada signal yang dikontrol menjadi sumber bilangan *random* (misalnya dengan mematikan lampu pada lampu lava, atau memasukkan suatu sinyal yang kuat ke dalam *sound card*).

Subversi *random number generator*. Subversi *random number generator* dapat diciptakan dengan menggunakan *cryptographically secure pseudo-random number generator* dengan nilai *seed* yang diketahui penyerang namun ditutupi di dalam perangkat lunak. Sebuah bagian dari *seed* yang relatif pendek, misalnya dari 24 - 40 bit, dapat benar-benar acak untuk mencegah *tell-tale repetitions*, namun tidak cukup menunda si penyerang untuk memulihkan hasil randomisasi.

Pertahanan terhadap serangan. Pertahanan terhadap serangan-serangan di atas dapat dilakukan dengan menggabungkan (misalnya dengan operasi XOR) *random number* yang dibangkitkan oleh perangkat keras dengan keluaran dari suatu *stream cipher*. Kunci *stream cipher* atau *seed* sebaiknya dapat diaudit atau diturunkan dari sumber yang terpercaya, misalnya lemparan dadu. Fortuna *random number generator* adalah contoh algoritma yang menggunakan mekanisme ini. Selain itu pertahanan juga dapat dilakukan dengan membangkitkan *password* dan *passphrases* menggunakan *true random generator*. Beberapa sistem bahkan menetapkan *password* acak untuk *user* daripada membiarkan user memilih sendiri passwordnya. Beberapa pendapat mengatakan bahwa penggunaan perangkat lunak yang *open source*s sangat baik untuk sistem keamanan.

3. Enkripsi dengan SALT

Berikut kode dalam bahasa C# yang menampilkan penggunaan SALT untuk membangkitkan berbagai ciphertext berbeda dari sebuah plaintext. Berbagai proses enkripsi yang menggunakan algoritma Rijndael ini menggunakan kunci dan *Initialization Vector* yang sama.

```
using System;
using System.IO;
using System.Text;
using System.Security.Cryptography;

public class RijndaelEnhanced
{
 #region Private members

 // Variabel untuk menyimpan panjang
 // minimal dan maksimal salt.
 private int minSaltLen = -1;
 private int maxSaltLen = -1;

 #endregion

 #region Constructors

 // Konstruktor untuk melakukan
 // enkripsi atau dekripsi dengan
 // kunci 256-bit, satu kali
 // iterasi, hashing tanpa salt,
 // tanpa Initialization Vector,
 // metode Cipher Block Chaining
 // (CBC), algoritma hashing
 // SHA-1, dan SALT 4 - 8 Byte.

 // Initialization Vector (IV).
 // IV diperlukan untuk mengenkripsi
 // blok pertama dari plaintext.
 // Nilai IV tidak perlu
 // dirahasiakan.
 public RijndaelEnhanced(string
 passPhrase, string initVector) :
 this(passPhrase, initVector, -1)
 {
 }

 #endregion

 #region Encryption routines

 // Enkripsi sebuah string yang
 // menghasilkan string base64-
 // encoded (base64: konversi
```

```
// karakter 8-bit menjadi karakter
// 7-bit).
public string Encrypt(string
plainText)
{
 return
 Encrypt(Encoding.UTF8.GetBytes
 (plainText));
}

// Enkripsi array of Byte
// menghasilkan string base64
// encoded.
public string Encrypt(byte[]
plainTextBytes)
{
 return Convert.ToBase64String
 (EncryptToBytes(plainTextBytes));
}

// Enkripsi sebuah string
// menghasilkan ciphertext dalam
// array of Byte
public byte[] EncryptToBytes(string
plainText)
{
 return EncryptToBytes
 (Encoding.UTF8.GetBytes
 (plainText));
}

// Enkripsi sebuah array of Byte
// menghasilkan ciphertext dalam
// array of Byte.
public byte[] EncryptToBytes(byte[]
plainTextBytes)
{
 // Menambahkan salt pada awal
 // plaintext.
 byte[] plainTextBytesWithSalt =
 AddSalt(plainTextBytes);

 // Enkripsi dilakukan dengan
 //menggunakan stream memori
 MemoryStream memoryStream =
 new MemoryStream();

 // Membuat operasi kriptografi
 // yang bebas thread.
 lock (this)
 {
  // Untuk melakukan enkripsi
  // digunakan Write mode.
  CryptoStream cryptoStream =
  new CryptoStream(memoryStream,
  encryptor,
  CryptoStreamMode.Write);

  // Mulai mengenkripsi data.
  cryptoStream.Write(
  plainTextBytesWithSalt, 0,
  plainTextBytesWithSalt.Length);

  // Menyelesaikan enkripsi.
  cryptoStream.FlushFinalBlock();

  // Memindahkan data terenkripsi
  // dari memori ke array of Byte.
  byte[] cipherTextBytes =
  memoryStream.ToArray();

  // Menutup stream memori.
  memoryStream.Close();
  cryptoStream.Close();

  // Mengembalikan data
  // terenkripsi.
  return cipherTextBytes;
 }
}

#endregion

#region Decryption routines

// Dekripsi ciphertext base64-
// encoded menghasilkan string.
public string Decrypt(string
cipherText)
{
 return Decrypt
 (Convert.FromBase64String
 (cipherText));
}

// Dekripsi ciphertext dalam array
// of Byte menghasilkan string.
public string Decrypt(byte[]
cipherTextBytes)
{
 returnEncoding.UTF8.GetString
 (DecryptToBytes(cipherTextBytes));
}

// Dekripsi ciphertext dalam
// base64-encoded menjadi array of
// Byte
public byte[] DecryptToBytes(string
cipherText)
{
 return DecryptToBytes
 (Convert.FromBase64String
 (cipherText));
}
```

```
// Dekripsi ciphertext dalam
// base64-encoded menghasilkan
// array of Byte.
public byte[] DecryptToBytes(byte[]
cipherTextBytes)
{
 byte[] decryptedBytes = null;
 byte[] plainTextBytes = null;
 int decryptedByteCount = 0;
 int saltLen = 0;

 MemoryStream memoryStream =
 new MemoryStream(cipherTextBytes);

 decryptedBytes =
 new byte[cipherTextBytes.Length];

 // Membuat operasi kriptografi
 // yang bebas thread.
 lock (this)
 {
  // Untuk melakukan dekripsi
  // digunakan Read mode.
  CryptoStream cryptoStream =
  new CryptoStream(memoryStream,
  decryptor,
  CryptoStreamMode.Read);

  // Dekripsi data dan mendapatkan
  // jumlah Bytes pada
  // plaintext.
  decryptedByteCount =
  cryptoStream.Read(decryptedBytes,
  0, decryptedBytes.Length);

  // Menutup stream memori.
  memoryStream.Close();
  cryptoStream.Close();
 }

 // Mendapatkan salt dari 4 Byte
 // pertama plaintext
 if (maxSaltLen > 0 &&
 maxSaltLen >= minSaltLen)
 {
  saltLen =
  (decryptedBytes[0] & 0x03) |
  (decryptedBytes[1] & 0x0c) |
  (decryptedBytes[2] & 0x30) |
  (decryptedBytes[3] & 0xc0);
 }

 // Alokasi array of Byte untuk
 // menyimpan plaintext tanpa
 // salt.
 plainTextBytes = new
 byte[decryptedByteCount-saltLen];

 // Menyalin plaintext tanpa salt.
 Array.Copy(decryptedBytes,
 saltLen, plainTextBytes, 0,
 decryptedByteCount - saltLen);

 // Mengembalikan plaintext.
 return plainTextBytes;
}

#endregion

#region Helper functions

// Menambahkan array of Byte yang
// dibangkitkan secara acak di awal
// array yang menyimpan
// plaintext.
private byte[] AddSalt(byte[]
plainTextBytes)
{
 // Validasi salt
 if (maxSaltLen == 0 ||
 maxSaltLen < minSaltLen)
 {
  return plainTextBytes;
 }

 // Membangkitkan salt.
 byte[] saltBytes = GenerateSalt();

 // Alokasi array untuk menyimpan
 // salt dan Byte plaintext.
 byte[] plainTextBytesWithSalt =
 new byte[plainTextBytes.Length +
 saltBytes.Length];

 // Menyalin Byte salt.
 Array.Copy(saltBytes,
 plainTextBytesWithSalt,
 saltBytes.Length);

 // Menambahkan plaintext ke salt.
 Array.Copy( plainTextBytes, 0,
 plainTextBytesWithSalt,
 saltBytes.Length,
 plainTextBytes.Length);

 return plainTextBytesWithSalt;
}

// Membangkitkan array yang
// menyimpan Byte yang kokoh secara
// kriptografis.
private byte[] GenerateSalt()
{
```

```
 // Inisialisasi panjang salt.
 int saltLen = 0;

 // Jika panjang minimal dan
 // maksimal salt sama, tidak perlu
 // membangkitkan panjang salt
 // secara random.
 if (minSaltLen == maxSaltLen)
 {
  saltLen = minSaltLen;
 }
 // Menggunakan random number
 // generator untuk menghitung
 // panjang salt.
 else
 {
  saltLen =
  GenerateRandomNumber(minSaltLen,
  maxSaltLen);
 }

 // Alokasi array of Byte untuk
 // menyimpan salt.
 byte[] salt = new byte[saltLen];

 // Mengisi salt dengan Byte yang
 // kokoh.
 RNGCryptoServiceProvider rng =
 new RNGCryptoServiceProvider();

 rng.GetNonZeroBytes(salt);

 // Memisahkan panjang salt.
 salt[0] = (byte)((salt[0] & 0xfc)
 | (saltLen & 0x03));
 salt[1] = (byte)((salt[1] & 0xf3)
 | (saltLen & 0x0c));
 salt[2] = (byte)((salt[2] & 0xcf)
 | (saltLen & 0x30));
 salt[3] = (byte)((salt[3] & 0x3f)
 | (saltLen & 0xc0));

 return salt;
}

// Membangkitkan integer random.
private int GenerateRandomNumber
(int minValue, int maxValue)
{
 // Membangkitkan seed integer dari
 // 4 Byte dari array.
 byte[] randomBytes = new byte[4];

 // Membangkitkan 4 Byte random
 RNGCryptoServiceProvider rng =
 new RNGCryptoServiceProvider();
 rng.GetBytes(randomBytes);

 // Konversi empat Byte random ke
 // dalam nilai integer positif.
 int seed =
 ((randomBytes[0] &0x7f) << 24) |
 (randomBytes[1]         << 16) |
 (randomBytes[2]         << 8 ) |
 (randomBytes[3]);

 // Melakukan randomisasi.
 Random random = new Random(seed);

 // Mengkalkulasi bilangan random.
 return random.Next(minValue,
 maxValue + 1);
}

#endregion

}

// Kelas RijndaelEnhanced untuk
// enkripsi dan dekripsi data dengan
// nilai salt yang random.
public class RijndaelEnhancedTest
{
 [STAThread]
 static void Main(string[] args)
 {
  string plainText =
  "Hello, World!"; // plaintext
  string cipherText =
  ""; // ciphertext
  string passPhrase =
  "Pas5pr@se"; // string sembarang
  string  initVector =
  "@1B2c3D4e5F6g7H8"; // 16 Byte

  // Menambahkan plaintext ke nilai
  // salt yang random dan panjangnya
  // 4 dan 8 Byte.
  RijndaelEnhanced rijndaelKey =
  new RijndaelEnhanced(passPhrase,
  initVector);

  Console.WriteLine
  (String.Format("Plaintext   :
  {0}\n", plainText));

  // Enkripsi plaintext yang sama
  // sebanyak 10 kali dengan kunci
  // dan IV yang sama.
  for (int i = 0; i < 10; i++)
  {
   cipherText =
   rijndaelKey.Encrypt(plainText);
```

```
   Console.WriteLine
   (String.Format("Encrypted #{0}:
   {1}", i, cipherText));

   plaintext =
   rijndaelKey.Decrypt(cipherText);
  }

  // Menuliskan hasil dekripsi.
  Console.WriteLine
  (String.Format("\nDecrypted
  :{0}", plainText));
 }
}
```

Hasil eksekusi kode:

```
Plaintext   : Hello, World!

Encrypted #0:
aZrQkBMKK98tnjddY/AUHxDeNlutylhcZ/AD
WpNrJ38=
Encrypted #1:
HhN01vcEEtMmwdNFliM8QYg+Y89xzBOJJG+B
H/ARC7g=
Encrypted #2:
uabeD7m8GdB9Kqm8tLI62zvkGN6Jf2+E0VtO
JfRQoRU=
Encrypted #3:
rtgB+F7ol9fEKRu/nm91qX3ZZSeHqKUFyK7r
LiLDpAs=
Encrypted #4:
MRuYU16m41MElUKPs4LZBxE5Mw8G9VlSf2Q0
EKSXmT4=
Encrypted #5:
UAHWYyVmdwQj4NHK5y0r6KdRe/98OlEmJnTL
b3Gn3yQ=
Encrypted #6:
TUpHFCiE6NYkp+NIwW9b/hGEEgv8V5IfI8i0
yip9tEo=
Encrypted #7:
1B9x2aQ8aoQgSnxv6561fPr7x36RooNFtZJD
7xhnVm0=
Encrypted #8:
SM+mAj0AWHfgBNdPM4lm7wV5t/PfJxxNvkWk
cb1WS5g=
Encrypted #9:
GGKnT87PFOCXALr5rX4wiU3xkhFZym1sLQqG
ZZTANzE=

Decrypted   : Hello, World!
```

Dari hasil eksekusi kode di atas didapatkan bahwa satu buah *plaintext* dapat dienkripsi bersama dengan *salt* menghasilkan berbagai *ciphertext* yang berbeda. Adanya tambahan *salt* pada proses enkripsi ini dapat mengurangi risiko *dictionary attack* tanpa bergantung pada kunci, *Initialization Vector*, dan parameter enkripsi lainnya. *Salt* dapat mengurangi akses program ke *database* atau *storage* tempat penyimpanan data terenkripsi, sehingga aplikasi dapat bekerja dengan lebih efisien.

Salt seringkali digunakan sebelum mengkalkulasi nilai *hash* dari suatu data. Berikut kode yang menunjukkan perbandingan proses *hashing* biasa dengan proses *hashing* yang menggunakan *salt* dalam .NET *framework*.

Hashing adalah proses yang sederhana. Berikut tahapan dalam proses *hashing*. Pertama-tama perlu diciptakan sebuah obyek *crypto service*. Selanjutnya, sebuah *array of* Byte dikirimkan sebagai parameter ke *method* `ComputeHash` yang ada pada obyek *crypto service*. Proses ini akan mengembalikan *array of* Byte yang mengandung data terenkripsi. Data terenkripsi tersebut harus dikonversi ke dalam string untuk disimpan atau ditampilkan. Contoh di bawah ini menggambarkan langkah-langkah *hashing* menggunakan algoritma SHA1:

```
string test = "Builder.com";
byte[] result =
new byte[test.Length];

try
{
 SHA1 sha =
 new SHA1CryptoServiceProvider();

 result =
 sha.ComputeHash
 (System.Text.Encoding.UTF8.GetBytes
 (test));

 Console.WriteLine
 (Convert.ToBase64String(result));
}
catch (CryptographicException ce)
{
 Console.WriteLine("Error: " +
 ce.Message);
}
```

Aplikasi yang umum menggunakan *hashing* antara lain pada penyimpanan data sederhana seperti *password*. Penyimpanan *password* setelah di-*hash*

sama seperti penyimpanan *password* biasa. Setiap kali terjadi *login user*, *password* yang dimasukkan di-*hash* dengan algoritma yang sama lalu hasilnya dibandingkan dengan yang tersimpan.

Salah satu kelemahan *hashing* adalah selalu menghasilkan nilai yang sama dari masukan yang sama. Misalnya dua *user* memiliki *password* yang identik akan memiliki nilai *hashing* yang sama untuk *password* mereka. Jika suatu saat *database* atau *storage* penyimpanan data *password* yang telah di-*hash* tersebut berhasil dimasuki oleh *hacker*, *hacker* tersebut dapat mengenali adanya "*trend*" dan mungkin dapat menebak nilainya. Dengan menambahkan *salt* ke dalam algoritma *hashing*, kemungkinan munculnya "*trend*" tersebut dapat diatasi.

Kelas RNGCryptoServiceProvider mengimplementasikan *random-number generator* yang kriptografis yang disediakan oleh *cryptographic service provider*. Pada kode di bawah ini terjadi penambahan *salt* sebelum terjadi *hashing*:

```
string test = "Builder.com";
byte[] salt = new byte[8];
string intermediate = null;

try
{
 SHA1 sha =
 new SHA1CryptoServiceProvider();

 RNGCryptoServiceProvider rng =
 new RNGCryptoServiceProvider();

 rng.GetBytes(salt);

 intermediate =
 Convert.ToBase64String(salt) +
 test;

 byte[] result = new
 byte[intermediate.Length];

 result =
 sha.ComputeHash
 (System.Text.Encoding.UTF8.GetBytes
 (intermediate));

 Console.WriteLine
 (Convert.ToBase64String(result));
}
catch (CryptographicException ce)
{
 Console.WriteLine("Error: " +
 ce.Message);
}
```

Untuk memperkuat keamanan, setelah menambahkan *salt* dapat juga dilakukan iterasi penghitungan nilai *hash* beberapa kali, dan hanya hasil iterasi terakhirlah yang akan diambil. Pada contoh di bawah ini, terdapat *password* "abc" yang menggunakan *salt* berukuran 8 Byte, berikut hasil penggabungannya:

```
78 57 8E 5A 5D 63 CB 06
```

Iterasi akan dilakukan sebanyak 1000 kali. Hasil *hash* dari penggabungan *password* dan *salt* adalah sebagai berikut:

```
P || S = 61 62 63 78 57 8E 5A 5D 63
CB 06
H(1) = MD5(P || S) =
0E8BAAEB3CED73CBC9BF4964F321824A
```

Proses ini diulang sampai 1000 kali.

```
H(2) = MD5(H(1)) =
1F0554E6F8810739258C9ABC60A782D5
H(3) = MD5(H(2)) =
ABA6FEDB4AD3EFAE8180364E617D9D79
...
H(1000) = MD5(H(999)) =
8FD6158BFE81ADD961241D8E4169D411
```

Hasil akhirnya yaitu 8FD6158BFE81ADD961241D8E4169D411 akan digunakan sebagai kunci untuk mengenkripsi data.

Untuk pengiriman pesan berikutnya, dengan password yang sama dan salt yang berbeda, misalnya:

```
7D 60 43 5F 02 E9 E0 AE
```

Iterasi akan dilakukan sebanyak 2048 kali. Dalam kasus ini, sama seperti sebelumnya, hasilnya yaitu CC584D1EE305FB7EF65926F62E88DFE3 akan digunakan untuk kunci enkripsi.

4. Analisis dan Pembahasan

Beberapa masalah utama yang memicu timbulnya salt antara lain keinginan pemilik data untuk mempunyai data pribadi seperti *PIN*, *password* yang mudah diingat, walaupun di sisi lain kemudahan untuk diinat

tersebut bisa memberikan dampak negatif seperti penyalahgunaan data. Selain itu juga untuk mencegah sembarang akses ke *database* atau *storage* yang menyimpan data-data penting dan rahasia, sebisa mungkin data tersebut disimpan dalam bentuk yang tidak dikenali atau tidak mudah diingat, karena jika *password* hanya disimpan dalam bentuk biasa, sekali terjadi akses dari pihak tak berwenang data *password* tersebut telah dapat dimanfaatkan.

Sebaliknya jika digunakan *hashing*, data yang tersimpan bukanlah *password* itu sendiri melainkan hasil *hashing* dari *password* tersebut. Terlebih lagi jika algoritma tersebut sangat kuat, tidak ada jalan lain bagi si penyerang selain menghitung nilai *hash* setiap data dalam media penyimpan lalu mencocokkannya dengan arsip *password* yang telah mengalami *hashing*. Jika terjadi kecocokan, maka *password* telah ditemukan. Namun waktu yang dibutuhkan untuk menghitung banyak nilai *hash* juga sangat besar.

Pada *dictionary attack* tidak diperlukan untuk mencari seluruh kombinasi yang mungkin, melainkan hanya mencari sekumpulan kata-kata yang mamiliki kemungkinan besar untuk dijadikan string *password*. *Salt* membantu mempersulit kerja pencarian dan pencocokan *password*. Selain *salt*, peraturan yang melarang kata-kata umum yang terdapat dalam suatu bahasa sebagai penggunaan *password* juga akan memberikan dampak positif untuk memperlambat pemecahan *password*.

Pada proses *hashing* dengan *salt* pada enkripsi berbasis *password*, untuk setiap *user* akan dibangkitkan string unik *random* dengan panjang tertentu. String tersebutlah yang dikatakan sebagai *salt*. Data yang akan disimpan ke dalam *database* atau *storage* tidak lagi data *password* itu sendiri, melainkan hasil *hashing* dari konkatenasi *password* dan *salt*, seakan-akan "*password"* (sebenarnya adalah *password* dan *salt*) itu sendiri memiliki bentuk yang lebih kompleks dan sangat unik.

Dalam dunia nyata, sering kali terjadi beberapa *user* memiliki data *password* yang sama, misalnya tanggal lahir mereka. Tanpa adanya penambahan *salt* pada *hashing,* beberapa *password* tersebut akan muncul sebagai hasil *hash* yang identik pada *database* atau *storage.* Semakin banyak terjadi penyimpanan hasil *hash* dari *password* yang sama, semakin besar kemungkinan terjadi "*trend*" yang dapat mempermudah pelacakan. Untuk menghindari terjadinya "*trend*" tersebut, dilakukan *hash* sedemikian rupa sehingga *password* yang identik tidak selalu menghasilkan nilai *hash* yang identik pula. Oleh karena itu dilakukan konkatenasi *password* dengan *salt* sebelum mengalami *hashing*. Dengan ini *dictionary attack* dapat dipatahkan, si penyerang tidak dapat hanya menghitung nilai *hash* dari setiap kata, lalu memeriksanya pada tabel hasil nilai *hash*, melainkan harus menghitung semua kemungkinan hasil *hash* untuk setiap kata, untuk setiap kemungkinan *salt*. *Salt* itu sendiri adalah bilangan yang dibangkitkan secara acak, sehingga pencarian kemungkinan *salt* akan sangat merepotkan karena memperkirakan sesuatu yang acak bukan hal yang mudah.

Salting secara signifikan menambahkan kesulitan bagi penyerang meskipun setelah data *password* berhasil didapatkan. Namun, dengan waktu yang sangat banyak, ditambah dengan pemilihan *password* yang lemah, pada akhirnya penyerang akan berhasil menerka *password* tersebut.

Aplikasi *salt* banyak ditemui dalam kehidupan sehari-hari, misalnya dalam sistem di mana klien mengirimkan *username* dan *password* ke *server*, *server* akan menambahkan *password*nya dengan *salt* mengkalkulasi nilai *hash*nya, lalu membandingkan hasilnya dengan nilai pada tabel data. Jika terjadi kecocokan, maka diasumsikan pengguna *username* tersebut adalah orang yang memang memiliki wewenang, sebaliknya jika tidak ditemukan kecocokan, akses akan ditolak untuk mencegah penyalahgunaan oleh pihak tak bertanggung jawab.

Melihat prosedur aplikasi *salt* di atas, dapat diambil kesimpulan bahwa *salt* boleh bersifat publik, karena *salt* adalah string yang acak. Idealnya, baik *salt* maupun nilai *hash* setelah penambahan *salt* sebaiknya disimpan secara privat agar tidak terjadi *dictionary attack* terhadap *salt*. Bagaimanapun juga, sangatlah sulit untuk mendeduksi informasi hanya dari *salt*.

Cara yang paling ampuh menghindari *dictionary attack* tentunya adalah dengan tidak mengijinkan penyerang mendapat akses ke *database* atau *storage* yang menyimpan data penting dan rahasia. Namun sistem keamanan yang baik adalah yang tidak bergantung pada sistem keamanan lain, jadi sistem keamanan tidak bisa sepenuhnya bergantung pada kekokohan jaringan sehingga tidak bisa ditembus pihak penyerang, melainkan harus memiliki kekokohan sendiri pula, salah satunya dengan penggunaan *salt.* Ide ini umum dikenali dengan istilah *"defense in depth"*.

Pada arsip passwd dalam sistem Unix yang klasik, *password* disimpan sebagai nilai *hash* dengan penambahan dua karakter *salt* di depan *password.* Arsip passwd bersifat publik, dapat dibaca oleh seluruh user di dalam sistem tersebut. Arsip tersebut harus dapat dibaca agar perangkat lunak yang dimiliki *user* dapat menemukan *username* dan informasi lainnya.

Penggunaan utama *salt* dilakukan karena adanya kemungkinan *user* memilih string yang sama sebagai *password* mereka. Tanpa adanya *salt*, password tersebut akan tersimpan sebagai nilai *hash* yang sama pada arsip *password*. Dengan menambahkan *salt* sebelum melakukan *hashing*, *password* yang identik pun akan tersimpan sebagai data yang berbeda dalam arsip *password.*

Sistem *shadow password* yang modern, di mana terjadi *hashing* pada *password* dan seluruh informasi lainnya disimpan dalam arsip non publik dapat membantu mengurangi keberhasilan serangan-serangan terhadap data. Bagaimanapun, pada instalasi *multiserver* informasi-informasi ini sifatnya umum diketahui karena penggunaan sistem manajemen *password* yang tersentralisasi. Pada instalasi seperti ini, *account "root"* pada masing-masing sistem dapat dikatakan kurang dapat dipercaya dibandingkan dengan administrator dari sistem *password* tersentralisasinya, sehingga tetap diperlukan algoritma *hashing* yang melibatkan nilai salt yang unik.

Salt dapat juga digunakan sebagai perlindungan terhadap *rainbow tables,* karena *salt* memperluas panjang dan kompleksitas dari *password* itu sendiri. Jika *rainbow tables* tidak memiliki *password* dengan panjang tertentu (misalnya sebuah *password* 8 Byte, dan *salt* 2 Byte, secara efektif merupakan sebuah *password* 10 Byte) dan kompleksitas (jika *salt* bukan alfanumerik, namun pada *database* hanya terdapat *password* dalam alfanumerik), maka *password* tersebut tidak akan berhasil dipecahkan. Kalaupun si penyerang berhasil memecahkannya, ia harus mampu memisahkan *salt* dari *password* sebelum dapat.

Salt mampu secara signifikan membuat *dictionary attacks* dan *brute-force attacks* dalam memecahkan sejumlah *password* menjadi lebih lambat. Tanpa *salt*, proses pemecahan berbagai password dalam satu waktu tertentu dapat dilakukan dengan cara berikut: untuk setiap tebakan *password* diperlukan satu kali penghitungan nilai *hash*nya lalu membandingkannya dengan seluruh nilai *hash* pada tabel data. Dengan penambahan *salt*, untuk setiap tebakan *password* terdapat sejumlah kemungkinan nilai *salt*, sehingga untuk menghitung nilai *hash* dari sebuah *password* diperlukan berkali-kali *hashing* dengan nilai *salt* yang berbeda samapi ditemukan kecocokan dengan data yang didapat. Proses ini memakan lebih banyak waktu dan biaya dibandingkan dengan pemecahan *password* pada *hashing* tanpa *password*.

Pada pembangkitan *password* secara *random*, yang secara garis besar mirip dengan pembangkitan bilangan random, *generator* umumnya membangkitkan suatu string berisi simbol dengan jumlah panjang yang telah ditentukan. *Random password generators* umumnya menghasilkan sebuah string berisi simbol sejumlah panjang yang telah ditentukan. Simbol tersebut dapat berupa beberapa karakter individual dari set karakter, suku-suku kata yang didesain untuk membentuk *password* yang dapat diucapkan, atau kata-kata dari daftar kata yang membentuk *passphrase*.

Kekokohan *random password* dapat dikalkulasi dengan menghitung *information entropy* dari proses randomisasi yang membangkitkannya. Jika setiap simbol dibangkitkan secara independen, berlaku entropi sesuai rumus berikut:

$$H = L\log_2 N = L\frac{\log N}{\log 2}$$

di mana N adalah jumlah simbol yang mungkin dan L adalah jumlah simbol yang terdapat dalam password. H dinyatakan dalam bit.

Set simbol	N	Entropi/simbol
Digits only (0-9) (e.g. PIN)	10	3.32 bits
Single case letters (a-z)	26	4.7 bits
Single case letters and digits (a-z, 0-9)	36	5.17 bits
Mixed case letters and digits (a-z, A-Z, 0-9)	62	5.95 bits
All standard U.S. keyboard characters	94	6.55 bits
Diceware word list	7776	12.9 bits

5. Kesimpulan

Kesimpulan yang dapat diambil dari studi tentang *salt* ini adalah:

1. *Salting* adalah metode yang sangat ampuh untuk menghindari *dictionary attack* dan *rainbow tables*.

2. *Salt* sangat penting untuk ditambahkan pada data penting dan rahasia, terutama untuk data yang kecil dan sifatnya umum.

3. *Salt* meningkatkan kekokohan data dengan meningkatkan kompleksitas data, sehingga diperlukan usaha, biaya, dan waktu yang lebih besar untuk mendapatkan isi data tersebut dibandingkan dengan metode yang tidak menggunakan *salt*.

4. Kelamahan pada *salt* adalah kelemahan pada *random number generator.* Hal ini dapat diatasi dengan melakukan pemilihan *random number generator* yang baik dan berkualitas secara kriptografis.

DAFTAR PUSTAKA

[1] http://blogs.msdn.com/
Tanggal akses: 10 Desember 2006.

[2] http://builder.com.com/
Tanggal akses: 5 Desember 2006.

[3] http://cyphersbyritter.com/
Tanggal akses: 10 Desember 2006.

[4] http://en.wikipedia.org/
Tanggal akses: 14 November 2006.

[5] http://fp.gladman.plus.com/
Tanggal akses: 5 Desember 2006.

[6] http://msdn2.microsoft.com/
Tanggal akses: 2 Desember 2006.

[7] http://www.algorithmic-solution.info/
Tanggal akses: 14 November 2006.

[8] http://www.di-mgt.com.au/
Tanggal akses: 5 Desember 2006.

[9] http://www.eggheadcafe.com/
Tanggal akses: 5 Desember 2006.

[10] http://www.obviex.com/
Tanggal akses: 10 Desember 2006.

[11] http://www.oracle.com/pls
Tanggal akses: 5 Desember 2006.

[12] http://www.rsasecurity.com/products/bsafe/
Tanggal akses: 14 November 2006.

[13] http://www.webnet77.com/
Tanggal akses: 12 Desember 2006.

[14] http://www.zvon.org/tmRFC/
Tanggal akses: 12 Desember 2006.