Analisis Serangan Terhadap *Hashed-Password* Serta Implementasi Penambahan *Salt* Sebagai Teknik Pertahanannya

Akbar Gumbira 13508106
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
if18106@students.if.itb.ac.id

***Abstract*—Penggunaan fungsi hash terhadap password mungkin sudah dianggap cukup bagi para *programmer* untuk disimpan pada basis data. Setidaknya, penggunaan fungsi hash ini dapat mengatasi jumlah karakter yang dibutuhkan pada field password pada basis data yang dibuat dan seseorang yang memiliki akses terhadap basis data tersebut tidak dapat melihat langsung password dari daftar user. Namun, hal ini masih menjadi celah bagi *attacker* untuk mendapatkan password-password tersebut apabila *attacker* inilah yang memiliki akses terhadap basis data. Karena hasil dari fungsi hash memiliki panjang yang konstan tergantung pada algoritmanya, maka dengan melihat field password yang telah dihash, *attacker* bisa dengan mudah mengetahui algoritma apa yang digunakan. Oleh karena itu, pada makalah ini, penulis membahas mengeni serangan yang mungkin terhadap *hashed-password* serta melakukan pengujian sederhana dengan penggunaan *salt* sebagai salah satu cara untuk mempersulit *attacker*. Penulis membatasi beberapa hal pada makalah ini yaitu fungsi hash yang digunakan pada pengujian adalah fungsi hash SHA-1, *attacker* telah mengetahui bagaimana *salted-hashed-password* dibentuk serta *salt* yang digunakan adalah *random salt*.**

***Index Terms*— fungsi hash, password, attacker, hashed-password, salt, random salt, SHA-1.**

I. PENDAHULUAN

Pada tahun 1979, Morris dan Thompson melakukan percobaan autentikasi password di UNIX. Mereka membuat daftar password yang sering digunakan, yaitu nama pertama dan terakhir orang, nama jalan, nama kota, kata-kata yang diambil dari kamus, nomor lisensi mobil, dan string-string pendek yang dibentuk secara random dari karakter. Mereka kemudian membandingkan dengan daftar dari file password system untuk mengecek apakah ada kecocokan diantara kedua file tersebut. Hasil percobaan tersebut menunjukkan bahwa lebih dari 86% dari semua password yang ada terdapat pada daftar password yang dibentuk tadi. Hasil percobaan yang mirip juga diperoleh oleh Klein pada tahun 1990. Oleh karena itu, selain user harus peduli tentang masalah ini dan karena masih banyak user yang tidak memedulikan masalah password yang dimiliki, programmer dari sistem yang dibuat pun harus dapat menangani masalah tersebut.

Penggunaan *password* merupakan suatu cara untuk melakukan autentikasi terhadap pengguna yang akan masuk pada suatu sistem. Cara biasa untuk sistem melakukan autentikasi adalah dengan menyimpan *password* tersebut pada basis data untuk setiap pengguna. Pada beberapa sistem operasi terdahulu, sistem operasi menyimpan *password* untuk setiap pengguna ini dalam bentuk tidak terekripsi. Kelemahan dari cara ini yaitu apabila *password* disimpan begitu saja pada *database*, tentu hal ini akan mengizinkan siapapun yang memiliki akses ke basis data tersebut mengetahui *password* dari setiap pengguna. Suatu solusi yang lebih baik yaitu dengan menggunakan fungsi *hash*. Keuntungan dari skema ini yaitu tidak seorang pun (bahkan yang memiliki hak akses terhadap basis data) dapat melihat *password* pengguna karena *password* tersebut tersimpan dalam bentuk terenkripsi. Selain itu, karena hasil dari fungsi *hash* memiliki panjang bit yang sama, maka secara teknis seberapa panjang *password* pengguna dapat ditangani pada *field password* pada basis data.

Secara umum, pada protokol autentikasi *password*, autentikasi dengan menggunakan *password* serta fungsi *hash* dilakukan dengan cara sebagai berikut:

1. User yang ingin mendapatkan autentikasi mengirimkan *password* yang dimilikinya ke sistem.
2. Sistem melakukan *hashing password* dengan menggunakan suatu fungsi *hash*
3. Sistem membandingkan hasil dari fungsi *hash* dengan nilai yang yang disimpan sebelumnya pada tabel (basis data).
4. User akan terautentikasi apabila hasil perbandingan tersebut adalah sama.

Bagaimanapun, skema perbaikan diatas masih dapat mengalami serangan oleh *attacker*. Terdapat tiga serangan yang memungkinkan dari skema tersebut. Serangan tersebut yaitu *dictionary attacks*, *bruteforce attacks*, serta *rainbow table*. Pada makalah ini, penulis akan membahas mengenai ketiga serangan tersebut serta penulis akan membuat pengujian terhadap implementasi penggunaan *salt* pada *password* yang dihash dengan menggunakan SHA-1 sebagai salah satu cara untuk menangani serangan tersebut. Penulis akan fokus pada serangan dimana *attacker* memiliki akses terhadap basis data.

II. DASAR TEORI

A. *Fungsi Hash SHA-1*

SHA merupakan fungsi *hash* satu arah yang dibuat oleh NIST (National Institute of Standards and Technology). Sampai saat ini, terdapat enam jenis SHA, yaitu SHA-0, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512. Algorima SHA-1 sendiri menerima masukan berupa pesan dengan ukuran maksimum 2^{64} bit dan menghasilkan *message digest* yang memiliki panjang 160 bit.

Untuk mempermudah pemrosesan, SHA-1 dibagi menjadi dua proses utama, yaitu *Preprocessing* dan *Hash Computation*. Secara lebih rinci, proses tersebut yaitu :

1. Preprocessing

Prepocessing ini merupakan proses sebelum proses komputasi hash dilakukan. Proses ini terdiri dari tiga langkah, yaitu:

a. Padding message

Message yang akan di-*hash* sebelumnya perlu di-*padding* dengan tujuan *message* yang telah di-*padding* ini merupakan kelipatan 512 bit. Misakan saja panjang dari *message*, *M*, yaitu l bit. Tambahkan bit 1 setelah *message* ini, kemudian ditambahkan ssebanyak *k* bit 0, dimana *k* merupakan nilai terkecil bilangan cacah dari solusi persamaan $l + 1 + k \cong 448 \, mod \, 512$. Kemudian, di akhir ditambahkan pula 64 bit yang merupakan representasi biner dari *l*. Sebagai contoh, misalkan *message* "abc" memiliki panjang $8x3 = 24$ (8-bit ASCII), sehingga seteleh ditambahkan bit 1 sebanyak 1, kemudian ditambahkan bit 0 sebanyak $448 - (24 + 1) = 423$, dan diakhir ditambahkan representasi biner dari panjang *message* dalam 64 bit. Hasil akhir dapat dilihat sebagai berikut :

$$\underbrace{01100001}_{a}\ \underbrace{01100010}_{b}\ \underbrace{01100011}_{c}\ \overbrace{1}^{bit\ 1}\ \overbrace{00..00}^{423\ bit}\ \overbrace{00..011000}^{64\ bit\ (l=24)}$$

b. Parsing padding message menjadi block-block

Pada SHA-1, *message* yang telah di-*padding* di-*parsing* menjadi N buah 512-bit *block*, sebut saja $M^{(1)}, M^{(2)}, \ldots M^{(N)}$. Karena 512 bit dari *block* input dapat ditulis sebagai 16 bit dari 32-bit *words*, maka 32 bit pertama dari block *i* dinotasikan sebagai M_0^i, 32 bit selanjutnya M_1^i, dan seterusnya sampai M_{15}^i.

c. Setting nilai awal hash, $H^{(0)}$

Untuk SHA-1. Nilai awal *hash*, $H^{(0)}$, terdiri dari 5 buath 32 bit *words*. Dalam notasi heksadimal sebagai berikut:

$$H_0^{(0)} = 67452301$$
$$H_1^{(0)} = efcdab89$$
$$H_2^{(0)} = 98badcfe$$
$$H_3^{(0)} = 10325476$$
$$H_4^{(0)} = c3d2e1f0$$

2. Hash Computation

Setelah preprocessing selesai dilakukan, setiap block message, $M^{(1)}$, $M^{(2)}, \ldots M^{(N)}$, diproses berdasarkan urutan menggunakan langkah berikut (menggunakan metode alternatif SHA-1 dari Federal Information Processing Standards, dengan nilai $MASK = 0000000f$ (dalam hex)):

for *i*=1 to N :
{

1. For *t* = 0 to 15:
   {
   Wt = $M_t^{(i)}$
   }
2. Inisialisasi 5 variabel, a, b, c, d, e dengan nilai hash ke (i-1)st :
   $a = H_0^{(i-1)}$
   $b = H_1^{(i-1)}$
   $c = H_2^{(i-1)}$
   $d = H_3^{(i-1)}$
   $e = H_4^{(i-1)}$
3. For *t* = 0 to 79:
   {
   $s = t \wedge MASK$
   If $t \geq 16$ then
   {
   $W_s = ROTL^1(W_{(s+13)\wedge MASK} \oplus W_{(s+8)\wedge MASK} \oplus W_{(s+2)\wedge MASK} \oplus W_s)$
   }
   $T = ROTL^5(a) + f_t(b,c,d) + e + K_t + W_s$
   $e = d$
   $d = c$
   $c = ROTL^{30}(b)$
   $b = a$
   $a = T$
   }
4. Hitung nilai hash ke-*i* $H^{(i)}$:
   $H_0^{(i)} = a + H_0^{(i-1)}$
   $H_1^{(i)} = b + H_1^{(i-1)}$
   $H_2^{(i)} = c + H_2^{(i-1)}$
   $H_3^{(i)} = d + H_3^{(i-1)}$
   $H_4^{(i)} = e + H_4^{(i-1)}$

}

Setelah melakukan langkah dari 1 – 4 sebanyak N kali, hasil message digest 160-bit diperoleh dari :

$$H_0^{(N)} \,||\, H_1^{(N)} \,||\, H_2^{(N)} || H_3^{(N)} || H_4^{(N)}$$

dengan || merupakan operasi konkatenasi.

III. ANALISIS KEMUNGKINAN SERANGAN TERHADAP PASSWORD YANG TELAH DIHASH

Terdapat tiga jenis kemungkinan serangan yang biasa dilakukan oleh *attacker* untuk mendapatkan autentikasi berupa password yang telah dihash. Tiga kemungkinan tersebut yaitu:

A. *Bruteforce Attack*

Bruteforce attack merupakan serangan yang digunakan dengan cara mencoba satu persatu kemungkinan password dari seorang user. Terdapat dua bruteforce yang mungkin dilakukan. Bruteforce yang pertama yaitu bruteforce password melalui sistem. Bruteforce ini dapat diatasi dengan cara membatasi jumlah percobaan login ke sistem apabila user berturut-turut salah memasukkan password. Brutefoce kedua yaitu, apabila attacker telah memiliki daftar hashed password. Attacker dapat melakukan bruteforce dengan cara membentuk kemungkinan string dari karakter, kemudian menghitung nilai hashnya. Cara ini sangat tidak *feasible* dilakukan, kecuali attacker mengetahui informasi tambahan, misalnya informasi panjang password.

B. *Dictionary Attack*

Dictionary attack merupakan serangan yang digunakan dengan bantuan kata-kata yang umum digunakan sebagai password. Kata-kata tersebut yaitu password yang sering digunakan, bahasa natural (kata tunggal yang diambil dari kamus), nama orang, nama tempat, nama tim, judul buku, film, dll.

Dictionary attack dapat dilakukan untuk melakukan serangan terhadap satu user ataupun banyak user sekaligus. Serangan terhadap satu user biasanya dilakukan dengan cara bruteforce namun dengan kemungkinan password dari dictionary yang telah dibentuk oleh attacker. Sedanggkan serangan terhadap banyak user akan *feasible* dilakukan oleh *attacker* apabila *attacker* sudah memiliki semua nilai hash dari setiap user. Semakin banyak *attacker* memiliki hash dari *user*, maka peluang sukses dictionary attack yang dilakukan semakin besar pula.

Pada makalah ini, penulis akan lebih fokus membahas mengenai serangan dictionary attack yang kedua. Artinya *attacker* dianggap memiliki tabel user pada basisdata (*attacker* dapat melihat nilai hash untuk setiap user). Serangan ini dilakukan oleh attacker dengan cara membuat tabel prekomputasi dengan field pasangan password beserta nilai hashnya. Tabel prekomputasi ini dibentuk dengan cara *attacker* mengumpulkan kata-kata yang sering digunakan sebagai password, kemudian melakukan hashing password tersebut satu persatu. Karena *attacker* mengetahui hash password dari sistem, maka karena panjang hash adalah konstan bergantung pada algoritma hashnya, attacker dapat mengetahui algoritma apa yang digunakan untuk melakukan hashing password oleh sistem. Setelah membentuk tabel prekomputasi tersebut (diperoleh daftar pasangan (password, hash(password)), attacker melakukan lookup field password pada tabel basisdata. Jika terdapat field password yang sama dengan nilai hash(password) dari tabel prekomputasi maka *attacker* dapat mengetahui password dari user dengan password tersebut.

Bagaimanapun juga, cara ini belum tentu *feasible* dilakukan apabila password dari user memiliki ukurang yang panjang. Semakin panjang ukuran password user, maka *attacker* juga perlu memasukkan kata-kata dengan panjang sepanjang ukuran password user juga, ini artinya ukuran tabel prekomputasi yang dibuat pun membesar.

Misalkan saja, *attacker* telah mengetahui fungsi hash dari password, sebut saja H, dan kumpulan password, sebut saja P. Hal yang dilakukan oleh *attacker* adalah melakukan komputasi data, sehingga diberikan suatu ouput h dari fungsi hash, maka *attacker* harus bisa menentukan p di P sehingga memenuhi H(p) = h, atau tidak ada p di P yang memenuhi. Cara yang paling mudah dilakukan untuk melakukan hal tersebut adalah menghitung H(p) untuk setiap p di P, dan menyimpannya pada tabel prekomputasi. Bagaimanapun juga untuk menyimpan tabel tersebut artinya dibutuhkan sebanyak $|P|\ x\ n$ bit memori, dengan n adalah ukuran dari output fungsi Hash (dimana SHA-1 adalah 160 bit) dan untuk ukuran P yang sangat besar, hal ini menjadi halangan bagi *attacker* zaman dahulu. Namun, cara ini menjadi sangat *feasible* saat ini karena ukuran *storage* yang semakin membesar dengan didukung murahnya *storage* tersebut.

C. *Rainbow Table*

Sebelum dibahas mengenai rainbow table ini, penulis akan membahas dulu mengenai *precomputed hash chains*. Hash chain ini digunakan sebagai teknik untuk mengurangi memori yang terlalu banyak digunakan pada dictionary attack, namun melakukan komputasi lebih banyak dibandingkan dengan dictionary attack. Hal ini biasa terjadi karena akan selalu terjadi trade-off antara penggunaan memori dan komputasi pada dunia komputasi.

Sebagai contoh dari hash chain ini, misalkan P merupakan kumpulan password yang terdiri dari 6 karakter, dan nilai hash yang dihasilkan memiliki panjang 32 bit. Pada hash chain digunakan fungsi reduksi R. Fungsi reduksi R ini digunakan untuk memetakan nilai hash menjadi suatu nilai pada P (Fungsi R ini bukan berarti memetakan hasil fungsi satu arah menjadi nilai awal, karena hal ini mustahil dilakukan. Selain itu, jika memang bisa dilakukan fungsi hash satu arah tidak melakukan pemetaan satu-satu, artinya terdapat collision yaitu untuk nilai masukan yang berbeda dimungkinkan menghasilkan nilai hash yang sama). Hash chain akan terlihat seperti ini:

$$\boldsymbol{kripto} \xrightarrow{H} AC1D40F3 \xrightarrow{R} \boldsymbol{dodidu} \xrightarrow{H} 3F54AE69 \xrightarrow{R} \boldsymbol{syandl}$$

Tabel hash chain ini dibuat dengan cara memilih kumpulan password awal secara random dari P, hitung rantai dengan panjang tetap untuk setiap rantai. Nilai yang disimpan pada tabel hanyalah nilai awal dan akhir password untuk setiap rantai. Password pertama dinamakan *starting point* dan password terakhir dinamakan *endpoint*. Pada contoh diatas, ***kripto*** merupakan *starting point*, sedangkan ***syandl*** merupakan *endpoint*. Password-password lainnya pada rantai tidak disimpan pada tabel, pada contoh diatas ***dodidu*** tidak disimpan pada tabel. Jadi, tabel yang dibentuk sebagai berikut :

Starting Point	End Point
kripto	**syandl**
.....	**.....**

Tabel 1Tabel Precomputed Hash Chain

Cara untuk mendapatkan password dari suatu nilai hash *h* yaitu dengan membuat rantai dengan menggunakan fungsi R pada *h*, kemudian hasilnya dilakukan fungsi H, hasil dari fungsi H ini dilakukan fungsi R, begitu seterusnya sampai pada suatu saat terdapat password yang terdapat pada tabel *endpoint* yang telah dibentuk. Karena *endpoint* berkorespondensi dengan *startingpoint*, maka dari *starting point* ini dibentuk kembali rantai. Sebagai contoh, misal diberikan nilai hash $3F54AE69$, kemudian dibentuk rantai, misakan saja pada pembentukan rantai kita menemukan password ***syandl***, dimana ***syandl*** ini disimpan pada tabel. Proses pencarian *endpoint* tabel sebagai berikut:

$$3F54AE69 \xrightarrow{R} \boldsymbol{syandl}$$

Karena pada tabel hash chain yang dibentuk *starting point* dari ***syandl*** adalah ***kripto***, maka perlu ditelusuri rantai dari ***kripto*** ini sampai $3F54AE69$ dicapai:

$$\boldsymbol{kripto} \xrightarrow{H} AC1D40F3 \xrightarrow{R} \boldsymbol{dodidu} \xrightarrow{H} 3F54AE69$$

Sehingga didapatkan string password yang dicari adalah ***dodidu***.

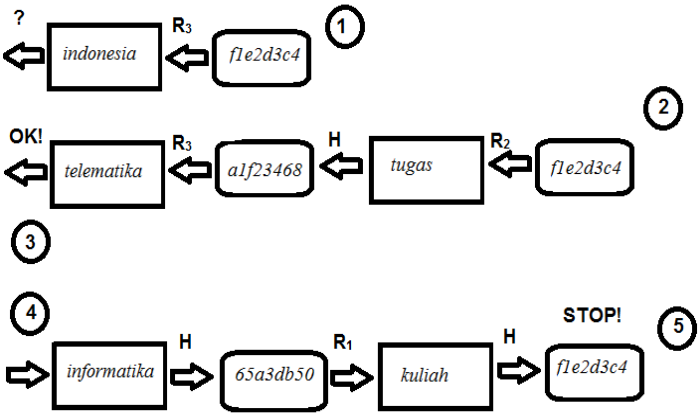
Kelemahan dari hash chain ini yaitu jika antara dua rantai yang dibentuk terdapat suatu titik yang menghasilkan password/nilai hash yang sama, maka tabel yang dibentuk tidak akan mengcover sebanyak mungkin password. Dari kelemahan ini, dibentuklah teknik baru, yaitu Rainbow Table.

Pada dasarnya Rainbow Table sama seperti hash chain, namun untuk mengatasi masalah *collision* tersebut fungsi reduksi *R* pada hash chain digantikan dengan barisan fungsi reduksi R_1 sampai R_k. Proses lookup pada tabel yang dilakukan yaitu, rantai pertama misalkan diasumsikan bahwa nilai hash berada pada posisi terakhir, sehingga kita hanya perlu menggunakan R_k. Rantai berikutnya diasumsikan nilai hash berada pada posisi kedua dari terakhir dan kita hanya menggunakan R_{k-1}, kemudian dilakukan H, dan terakhir dilakukan R_k. Begitu seterusnya sampai rantai terakhir, dimana di rantai terakhir ini digunakan semua fungsi reduksi. Sebagai contoh, berikut tabel hash chain yang dibentuk:

Starting Point	End Point
makalah	**kriptografi**
keamanan	**informasi**
informatika	**telematika**

Tabel 2 Rainbow Table

Misal diberikan nilah hash *f1e2d3c4* dan perlu dicari password yang menghasilkan nilai hash tersebut. Langkah dari pencarian tersebut yaitu sebagai berikut:



Gambar 1 Proses Pencarian Password Pada Rainbow Table

Penjelasan dari Gambar 1 yaitu sebagai berikut:

1. Dimulai dari nilai hash *f1e2d3c4*, dihitung nilai hasil fungsi reduksi dengan menggunakan fungsi reduksi terakhir. Kemudian dicek apakah password hasil fungsi reduksi terdapat pada kolom endpoint pada tabel.
2. Jika langkah 1 gagal, artinya ***indonesia*** tidak terdapat pada tabel, maka dengan menggunakan dua fungsi terakhir dibuat rantai hash. Apabila tidak terdapat password pada tahap ini, dilakukan pembuatan rantai dengan menggunakan tiga reduksi, empat reduksi, dan seterusnya sampai password ditemukan. Jika tidak terdapat password pada rantai, maka password gagal didapat.
3. Jika langkah 2 berhasil (terdapat password pada rantai), maka password didapat dengan cara membentuk rantai dimulai dari starting point dari end point yang bersangkutan. Dalam contoh ini, starting point yang dimaksud adalah ***telematika***.

4. Pada langkah ini, dimulai dari *starting point*, dalam hal ini ***informatika***, dibuat rantai. Pembuatan rantai dilakukan dengan cara membandingkan nilai hash dari starting point dengan nilai hash yang dimaksud (*f1e2d3c4)*. Jika tidak sesuai, lakukan fungsi reduksi dari nilai hash ini. Begitu seterusnya sampai didapat nilai hash yang dimaksud. Password yang bersesuaian adalah password sebelum dilakukan hash sehingga menghasilkan nilai hash yang dimaksud.

Telah disebutkan sebelumnya bahwa rainbow table merupakan teknik trade-off antara memori dan komputasi yang menyempurnakan dictionary attack. Dari fakta yang penulis peroleh dari daftar referensi, berikut perbandingan waktu untuk mendapatkan password antara brute force dan rainbow table :

Karakteristik Password	Contoh	Waktu (Bruteforce)	Waktu (Rainbow Table)
8-digit password (semua huruf)	abcdefgh	1,6 hari	28 menit
9-digit password (huruf dan angka)	AbC4E8Gh	378 tahun	28 menit
10-digit password	Ab4C7EfGh2	23481 tahun	28 menit
14-digit password	1A2*3&def456G$	$6.09x10^{12}$ tahun	28 menit

Tabel 3 Tabel Maksimum Waktu Pemecahan Password

IV. SOLUSI UNTUK MENGATASI SERANGAN

Dari berbagai kemungkinan serangan yang telah dipaparkan oleh penulis sebelumnya, terdapat beberapa cara untuk mengatasi serangan tersebut, yaitu :

1. Untuk serangan yang melakukan bruteforce langsung terhadap sistem, kita dapat mengatasinya dengan cara memberikan batas maksimum percobaan login.
2. Untuk mengatasi serangan terhadap dictionary attack serta rainbow table, kita dapat mempersulit attacker dengan cara menambahkan salt pada password. Pada makalah ini, penulis akan membahas lebih fokus bagaimana penggunaan salt tersebut.

Salt merupakan penambahan bit pada password sehingga attacker tidak dapat secara langsung melakukan hashing dari daftar password yang telah dibuatnya. Penambahan ini pada umumnya dilakukan dengan cara:

```
saltedhash(password) = hash (password+salt)
```

atau

```
saltedhash(password) = hash (hash(password)+salt)
```

Terdapat dua jenis salt yang umumnya digunakan, yaitu Constant Salt dan Random Salt.

A. *Constant Salt*

Constant salt merupakan salt yang ditambahkan pada password dimana untuk setiap password yang ada, sistem akan menambahkan salt yang sama. Oleh karena itu, constant salt tidak perlu ditambahkan pada basis data, cukup ditambahkan pada program. Notasi algoritma dari penambahan constant salt yaitu sebagai berikut (misal constant salt yang ditambahkan adalah 'rahasia' dan fungsi hash yang digunakan adalah SHA1):

```
function String HashConstantSalted(password : string)
{
        String salt = "rahasia";
        String newpassword = password+salt;
        return SHA1(newpassword);
}
```

Dengan menggunakan constant salt, apabila *attacker* memiliki akses terhadap basis data, maka *attacker* dapat dengan mudah mengetahui bahwa sistem menggunakan constant salt. Cara yang dilakukan yaitu:

1. Buat akun baru dengan menggunakan aplikasi target (target yang dimaksud adalah sistem yang diattack)
2. Lakukan dumping basis data (*attacker* memiliki akses terhadap basis data. Lihat kembali fokus penulis di bagian Pendahuluan). Dari basis data tersebut dapat dilihat berapa bit password yang telah dihash. Dari panjang bit ini, *attacker* bisa mengetahui fungsi hash apa yang digunakan karena panjang hasil fungsi hash adalah konstan.
3. Karena *attacker* mengetahui fungsi hash yang digunakan, maka *attacker* bisa melakukan hash sendiri dari password akun yang baru dibuatnya. Dengan membandingkan dengan field password pada basis data, *attacker* bisa mengetahui bahwa salt ditambahkan pada password dan salt constant yang ditambahkan apabila tidak terdapat bit lain yang ditambahkan pada final hash di basis data.

Dengan menggunakan constant salt, skema pemikiran dari *attacker* adalah sebagai berikut:

1. *Attacker* mengetahui bahwa hash(password) = A
2. Pada basis data misalkan hasil hashnya adalah B
3. Maka *attacker* tinggal mencari X, dimana hash(password+X) = B (Jika *attacker* mengetahui bahwa X dikonkatenasi setelah password).

Dari penjelasan diatas dapat diambil kesimpulan bahwa penggunaan constant salt tidak begitu aman.

B. *Random Salt*

Random salt merupakan salt yang ditambahkan pada password dimana untuk setiap password yang ada, sistem akan menambahkan salt yang berbeda. Oleh karena itu, random salt perlu ditambahkan pada basis data atau dikonkatenasi pada akhir hasil hash password yang juga telah ditambahkan salt. Notasi algoritma dari penambahan Random Salt ini sebagai berikut (misal jumlah byte salt

adalah *n* byte dan fungsi hash menggunakan SHA1):

```
Function                                    String
HashRandomSalted(password:String)
{
        byte[] saltBytes = new byte[n];

        // isi salt byte secara random
        RandomBytes(saltBytes)

        String            saltString            =
        saltByte.ToString();

        //New Password (pass+salt)
        String            newPassword           =
        password+saltString;

        //Hash New Password:
        String resultHash = SHA1(newPassword);

        //Tambah salt di belakang hasil:
        resultHash+=saltString;

        return resultHash;
}
```

Kelebihan dari metode Random Salt adalah bahwa attacker tidak bisa secara langsung membuat daftar kemungkinan password (dictionary), mengenkripsinya dan menyimpannya, sehingga password yang telah dihash dapat dengan mudah di-*lookup*. Karena untuk setiap password penggunaan salt yang digunakan mungkin berbeda, maka jika *attacker* masih ingin menggunakan dictionary attack, maka *attacker* perlu membuat dictionary table dimana untuk setiap kemungkinan password diperlukan sebanyak 2^n kemungkinan penambahan salt (apabila salt terdiri dari *n* bit). Artinya, jika *atacker* memiliki sebanyak *p* password pada daftar kemungkinan, maka total banyak password yang dibentuk adalah $p \; x \; 2^n$. Komputasi hash yang diperlukan pun sebanyak itu pula. Jadi, semakin besar *n*, semakin mustahil *attacker* dapat membentuk dictionary tabel tersebut. Perlu diperhatikan bahwa penggunaan salt ini mengatasi serangan attacker yang ingin mendapatkan banyak password, bukan hanya ingin mendapatkan hanya satu password.

Misalkan saja, *attacker* lebih pintar dari yang kita duga, maka karena field hash password dapat ia lihat, ia dapat mengetahui *salt* untuk setiap field hash password. Kemudian ia membuat daftar *salt* yang berbeda, dimana jumlah dari *salt* yang berbeda yaitu dalam rentang [1, 2^n]. Jika beruntung, mungkin attacker bisa saja mendapatkan pengali pada rentang tersebut yang kecil. Untuk mengatasi hal ini, programmer sistem bisa saja membuat *salt* pada akun user baru sehingga jumlah *salt* yang digunakan adalah maksimum dengan cara mengecek terlebih dahulu apakah random salt yang akan disisipkan sudah ada pada daftar *salt* yang telah digunakan.

Keuntungan lain dari Random Salt adalah apabila terdapat dua atau beberapa user yang memiliki password sama, maka apabila tanpa menggunakan random salt ini, password-password tersebut akan tersimpan dengan nilai hash yang sama pada basis data. Hal ini tidak menutup fakta bahwa apabila terdapat dua akun yang memiliki password sama, maka seseorang yang mengetahui salah satu akun tersebut juga akan memiliki akses ke akun lain. Dengan menggunakan salt, string password yang sama untuk akun yang berbeda mungkin menghasilkan nilai hash yang berbeda.

V. IMPLEMENTASI PENAMBAHAN SALT SERTA PENGUJIAN KEKUATAN

Untuk menguji kekuatan penambahan salt ini dibandingkan dengan tanpa penambahan salt, penulis melakukan pengujian sederhana. Skenario pengujian adalah sebagai berikut:

1 Penulis membuat tabel user yang berisi 100000 username, password, nilai SHA1 dari password tanpa ditambahkan salt. Kemudian, penulis membuat dictionary password yang mungkin (terdiri dari 150 password dan 3 diantaranya terdapat pada tabel user). Dari 150 password ini penulis membuat daftar nilai hashnya dan dilakukan lookup dari tabel user ke dictionary tersebut dengan menggunakan struktur data hash table, menghasilkan daftar user beserta password yang sama dengan yang ada pada dictionary password. Kemudian dihitung waktu mulai dari melakukan pembuatan pasangan (password, hashedpassword) sampai sistem mengeluarkan daftar user yang passwordnya terdapat pada dictionary.

2 Penulis membuat tabel user yang berisi 100000 username, password, nilai SHA1 dari password dengan penambahan salt sebanyak 8 bit. Kemudian, penulis membuat dictionary password yang mungkin (terdiri dari 150 password dan 3 diantaranya terdapat pada tabel user). Dari 150 password ini, karena terdapat 2^8 bit kemungkinan salt, maka untuk setiap password tersebut dilakukan penghitungan sebanyak 2^8 salt untuk ditambahkan pada password. Sehingga total dictionary yang ada yaitu $150 \; x \; 2^8 = 38400$ kemungkinan. Dictionary pada program menggunakan struktur data hash table. Sistem akan menelusuri satu persatu tabel user dan mengecek apakah hashed password terdapat pada dictionary. Jika iya, maka ditambahkan daftar user dengan password yang sama dengan password pada dictionary. Kemudian dihitung waktu mulai dari melakukan pembuatan pasangan (password, hashedpassword) sampai sistem mengeluarkan daftar user yang passwordnya terdapat pada dictionary.

3 Percobaan kedua diulangi, namun jumlah bit salt diganti menjadi 16 bit. Artinya terdapat $150 \; x \; 2^{16} = 9830400$ kemungkinan isi dictionary.

Sebagai gambaran, berikut tabel user yang penulis buat (tabel ini tidak dibuat pada basis data, namun pada program):

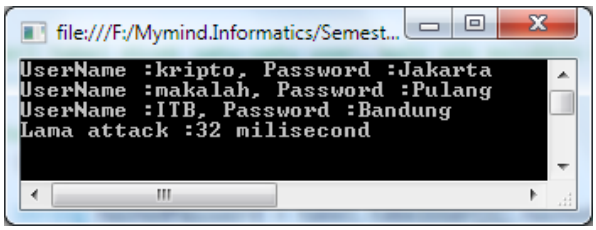
User	Password	Hashed Password
Akbar0	AkbarPass0	SHA1(AkbarPass0)
Akbar1	AkbarPass1	SHA1(AkbarPass1)
Akbar2	AkbarPass2	SHA1(AkbarPass2)
…….	…….	…….
Akbar99997	AkbarPass99997	SHA1(AkbarPas99997)
kripto	Jakarta	SHA1(Jakarta)
makalah	Pulang	SHA1(Pulang)
ITB	Bandung	SHA1(Bandung)

Tabel 4 Tabel User

Kemudian, daftar password yang mungkin, yaitu sebanyak 150 kata, penulis simpan dalam struktur data array. Isi dari daftar password yaitu sebagai berikut:

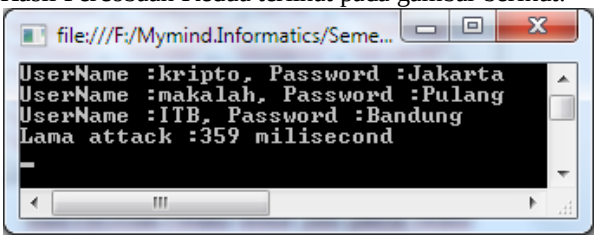
```
String[] dictionary = {"Akbar", "Bandung",
"Cianjur", "Depok", "Edinburgh", "Fajar",
"Garut", "Hotel", "Ikan", "Jakarta","Koala",
"Lumpur","Makan","Norak", "Operasi",
"Pulang", "Queue", "Rabu", "Sambal",
"Tulang", "Ulama", "Vaksin", "Waditra", "X-
Ray","Yamien", "Zahira","Aku", "Adalah",
"Anak", "Ayam", "Buku", "Coba", "Dahulu",
"Mesin", "Mandi","Nenek", "Nona", "Nasi",
"Odol", "Prancis", "Peluru", "Kutai",
"Kuta", "Burung","Bebek", "Bintang",
"Cibubur", "Dendam", "Esai", "Subuh",
"Akhir", "Banda", "Cendol", "Domba",
"Estonia", "Fikir","Gua", "Handuk", "Imba",
"Janda", "Kodrat", "Luluh","Malu", "Nuduh",
"Otak", "Patung", "Qed", "Rombak",
"Sandal","Tubuh", "Udin", "Vandalisme",
"Wayang", "Xenia","Yamaha", "Zombie","Asik",
"Abidal", "Adonan","Ambang","Bakti","Cinta",
"Duka", "Musik", "Malam","Nitip", "Ninja",
"Nobita", "Opah", "Pusaka", "Panji",
"Kentara", "Kubangan", "Busuk","Bidang",
"Bogor", "Ciater", "Dandanan", "Ebay",
"Sunda","Google", "Sering","Efektif",
"Sesuatu", "Tampilan", "Kemampuan","Biasa",
"Petunjuk","Kalkulator", "Contoh", "Hitung",
"Konstanta","Sekolah", "Guru", "Murid",
"Ilmu", "Manfaat", "Nilai","Konversi",
"Meter", "Uang", "Rupiah", "Kode", "Negara",
"Definisi", "Komputer","Merah","Kuning",
"Hijau", "Biru","Tugas", "File", "Penting",
"Artikel", "Format", "Ekstensi","Karyawan",
"Inggris", "Kelompok", "Grup", "Terjemahan",
"Belukar","Kotak", "Persegi", "Lingkaran",
"Sumber", "Baku", "Khawatir","Pilot",
"Amerika"
};
```

Hasil Percobaan Pertama terlihat pada gambar berikut:



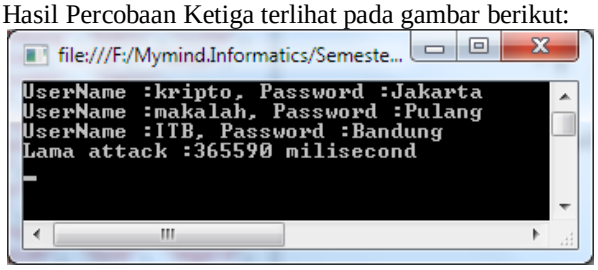
Gambar 2 Hasil Percobaan Pertama

Hasil Percobaan Kedua terlihat pada gambar berikut:



Gambar 3 Hasil Percobaan Kedua

Hasil Percobaan Ketiga terlihat pada gambar berikut:



Gambar 4 Hasil Percobaan Ketiga

Dari percobaan pertama, didapatkan waktu yang diperlukan untuk mendapatkan password yang telah dihash dengan SHA-1 tanpa menggunakan salt yaitu 32 milisecond. Sedangkan untuk percobaan kedua, yaitu dengan menggunakan salt sebanyak 8 bit diperlukan waktu 359 milisecond. Dan untuk percobaan ketiga, yaitu dengan menggunakan salt sebanyak 16 bit diperlukan waktu 365990 milisecond.

Dari ketiga percobaan ini, didapat kesimpulan bahwa dengan menggunakan salt, hashed-password yang di attack akan lebih sulit untuk diungkap. Hal ini dikarenakan dengan penambahan salt, *attacker* perlu membuat semua kemungkinan salt untuk dikonkatenasi dengan password. Selain itu, pada percobaan ini program *attacker* dianggap telah mengetahui bahwa salt ditambahkan di akhir password dan fungsi penambahan salt yang dilakukan yaitu:

saltedhash(password, salt) = SHA1(password+salt)+salt

Apabila kedua hal diatas tidak diketahui, maka akan sulit bagi *attacker* untuk mendapatkan password tersebut.

VI. KESIMPULAN

Dari semua yang telah penulis paparkan sebelumnya, dapat ditarik beberapa kesimpulan, yaitu:

1. Terdapat tiga kemungkinan serangan terhadap hashed-password, yaitu bruteforce, dictionary attack, serta rainbow table.

2. Penambahan random salt terhadap password akan meningkatkan kekuatan dari serangan attacker. Hal ini dikarenakan untuk setiap password, random salt yang dihasilkan pun berbeda. Sehingga untuk setiap password pada daftar password yang telah dibuat, attacker perlu menambahkan random salt sebanyak kemungkinan dari banyaknya random salt tersebut.
3. Keuntungan lain dari penggunaan random salt adalah, jika dua akun memiliki password sama, maka pada basis data hashed-password yang terlihat mungkin berbeda. Sehingga, kemungkinan pemilik akun yang memiliki akses terhadap basis data yang memiliki password yang sama dengan pemilik akun lain tidak akan melihat nilai hash yang sama pada basis data.
4. Masih banyak user yang menggunakan password yang mudah didapat dari dictionary. Oleh karena itu, programmer sistem pun harus dapat mengatasi hal ini.

REFERENSI

[1] A.Menezes, P.Van Oorschot, S. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1997, ch.9.

[2] Andrew S Tanenbaum, *Modern Operating System 2nd Edition*. Prentice Hall PTR, ch.9.

[3] Federal Information Processing Standars (FIPS) Publication 180-2, *Secure Hash Standard*, August 1, 2002.

[4] Rinaldi Munir. Slide kuliah "Protokol Kriptografi".

[5] Rinaldi Munir. Slide kuliah "SHA".

[6] http://s3.amazonaws.com/ppt-download/rainbowtables. Diakses tanggal 25 Apil 2011.

[7] http://s3.amazonaws.com/ppt-download/nullpresentationcrackingsaltedhashes. Diakses tanggal 25 April 2011.

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, 7 Mei 2011

ttd

Akbar Gumbira (13508106)