Implementasi Pembangkitan Bilangan Acak Untuk Menghasilkan Salt Menggunakan Hénon Map

Fawwaz Anugrah Wiradhika Dharmasatya - 13520086
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13520086@std.stei.itb.ac.id

***Abstrak*—*Hash* bersifat deterministik sehingga masukan yang sama akan menghasilkan nilai *hash* yang sama. Hal ini membuat penyimpanan *password* menggunakan *hash* rentan terhadap serangan berbasis tabel *hash*. *Salt* muncul sebagai solusi untuk masalah ini sehingga dua buah *password* yang sama memiliki nilai *hash* yang berbeda. Namun, *salt* harus dibangkitkan dengan cukup keacakan agar terhindari dari kolisi. Entropi sistem dapat memberikan sumber keacakan yang benar-benar acak. Namun, pengumpulan entropi membutuhkan waktu sehingga pembangkitan *salt* dalam jumlah banyak membutuhkan waktu yang lama. Salah satu alternatif yang dapat digunakan untuk membangkitkan *salt* yang acak dalam jangka waktu singkat adalah menggunakan *cryptographically secure pseudorandom number generator* (CSPRNG*)*. Pembangkitan bilangan acak menggunakan fungsi *chaos Hénon map* memenuhi syarat CSPRNG sehingga dapat dimanfaatkan untuk memecahkan masalah ini .**

Kata kunci—random number generator; rng; chaos system; hénon map ; salt; CSPRNG;

I. PENDAHULUAN

Pada penyimpanan data sensitif seperti *password*, umumnya *password* disimpan di basis data dalam bentuk *hash* dari *password* tersebut untuk mencegah penyalahgunaan *password* jika terjadi *data breach*. Namun, fungsi *hash* bersifat deterministik sehingga jika terdapat dua buat masukan yang sama, nilai *hash* dari kedua masukan akan sama. Hal ini memungkinkan penyerang membuat tabel berisi *password* dan nilai *hash password* tersebut. Penyerang dapat memanfaatkan tabel tersebut untuk mengakses akun lain yang memiliki nilai *hash password* yang sama. Hal ini akan mengancam keamanan pengguna.

Salah satu metode yang umum digunakan untuk mencegah serangan tersebut adalah menambahkan *salt* saat proses *hashing*. *Salt* adalah sebuah *string* yang dibangkitkan secara acak untuk ditambahkan ke *password* pada saat *hashing*. *Salt* bersifat unik untuk setiap *password* sehingga dua buah *password* yang sama akan memiliki nilai *hash* yang berbeda. Hal ini akan menyulitkan penyerang membuat tabel *hash password*.

Berdasarkan *guidelines* OWASP[1], *salt* harus *cryptographically strong*. *Cryptographically strong* didefinsiikan sebagai sistem kriptografi yang sangat tahan terhadap kriptoanalisis. Kriptoanalisis adalah upaya untuk men-*decipher* pola rahasia sebuah sistem. Untuk menghasilkan nilai yang *cryptographically strong*, umumnya digunakan *cryptographically secure pseudorandom number generator*(CSPRNG) atau mengumpulkan masukan acak dari sumber yang tidak bisa diamati, seperti API *Random Generator* di sistem operasi.

Dari kedua metode tersebut, membangkitkan nilai menggunakan API *Random Generator* dari sistem operasi, seperti **/dev/random** di Linux, menghasilkan nilai yang benar-benar acak. Hal ini dikarenakan API *Random Generator* mengumpulkan keacakan (entropi) sistem untuk membangkitkan nilai acak. Namun, hal ini berarti bila keacakan sistem belum cukup untuk membangkitkan nilai, program harus menunggu sampai keacakan sistem sudah cukup untuk mendapatlan nilai acak. Hal ini dapat menyebabkan masalah *availability* jika terdapat banyak *request* dalam waktu singkat yang membutuhkan bilangan acak untuk membangkitkan *salt*. Oleh karena itu, pembangkitan *salt* menggunakan CSPRNG menjadi alternatif yang bagus untuk membangkitkan *salt* dalam jumlah banyak dan waktu yang singkat.

Salah satu metode untuk mengimpementasikan CSPRNG adalah menggunakan fungsi *chaos*. Fungsi *chaos* adalah fungsi yang peka pada nilai awal. Jika nilai awal berubah sedikit, maka nilai fungsi *chaos* yang dihasilkan bisa berbeda jauh. Salah satu contoh fungsi *chaos* adalah *hénon map*.

Pada makalah ini, penulis akan menjelaskan teori dasar *hénon map*, implementasi *hénon map* untuk membangkitkan *salt*, beserta eksperimen dan analisis eksperimen untuk menunjukkan efektifitas *hénon map* dalam membangkitkan *salt.*

II. DASAR TEORI

A. *Salt*

Salt adalah *string* unik yang dibangkitkan secara acak untuk ditambahkan ke setiap *password* pada saat *hashing*. Karena nilai *salt* unik untuk setiap *password*, penyerang harus memecahkan *hash* satu per satu menggunakan *salt* masing-masing *hash*. Hal ini membuat memecahkan *hash* dalam jumlah besar menjadi lebih sulit. Selain itu, *salt* juga melindungi *password* dari *rainbow attack*, yang menggunakan daftar *hash* yang sudah dikomputasi oleh penyerang.

B. Bilangan Acak

Bilangan acak adalah bilangan yang tidak dapat diprediksi nilai dan kemunculannya. Bilangan acak bisa berupa *integer*, bilangan riil antara 0-1, atau *string* biner. Dalam dunia kriptografi, bilangan acak memegang peranan penting. Beberapa diantaranya adalah:

1. Pembangkitan nilai-nilai parameter kunci dalam algoritma kriptografi kunci publik.
2. Pembangkitan nilai acak k dalam algoritma enkripsi ElGamal.
3. Pembangkitan *initialization vector* (IV) dalam *block cipher.*
4. Pembangkitan *string* di dalam mekanisme *challenge and response* untuk otentikasi.
5. Pembangkitan kunci sesi oleh *client* di dalam SSL.

C. Pembangkitan Bilangan Acak

Bilangan acak bisa dibangkitkan dengan dua cara, mengumpulkan masukan acak dari sumber yang tidak bisa diamati, seperti API *Random Generator* di sistem operasi, atau dibangkitkan menggunakan komputasi. Namun, tidak ada prosedur komputasi yang menghasilkan deret bilangan acak yang benar-benar sempurna (*true random*). Bilangan acak yang dihasilkan dengan prosedur komputasi adalah bilangan acak semu (*pseudo-random*), karena pembangkitan bilangannya dapat diulang kembali. Pembangkit deret bilangan acak semacam itu disebut *pseudo-random number generator (PRNG*). PRNG bersifat deterministik, artinya bilangan acak bisa diulang kembali pembangkitannya asalkan kunci (umpan) yang digunakan sama.

D. CSPRNG

C*ryptographically secure pseudorandom number generator*(CSPRNG) adalah pembangkit bilangan acak yang aman secara kriptografi. Agar aman secara kriptografis, PRNG harus memenuhi beberapa syarat berikut:

1. Secara statistik lolos uji keacakan (*randomness test*).
2. Tahan terhadap serangan (*attack*) yang serius. Serangan ini bertujuan untuk memprediksi bilangan acak yang dihasilkan dari nilai-nilai sebelumnya.

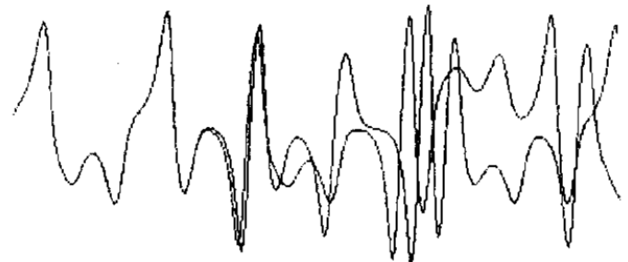
Beberapa persyaratan untuk memastikan bahwa syarat kedua terpenuhi antara lain:

1. Pembangkit bilang acak lulus pengujian *next-bit*. Sebuah pembangkit bit acak dikatakan lulus uji bit berikutnya (next-bit test) jika diberikan barisan k bit, maka tidak dapat diprediksi bit berikutnya 0 atau 1 dengan peluang lebih besar dari ½, sehingga dikatakan *unpredictable*.
2. Pembangkit bilangan acak harus tahan terhadap kompromi perluasan status (*state compromise extensions*). Maksudnya adalah jika penyerangan memperlajari sebagian atau seluruh status sekarang, tidak mungkin bagi penyerang untuk membentuk ulang aliran bilangan acak sebelumnya. Dalam praktiknya, ini berarti CSPRNG harus menggunakan fungsi banyak-ke-satu dalam prosesnya sehingga percobaan untuk membalikkan proses akan menaikkan jumlah kemungkinan aliran secara eksponensial dengan setiap langkah mundur. [5]

E. Teori Chaos

Teori *chaos* menggambarkan perilaku sistem dinamis nonlanjar yang menunjukkan fenomena *chaos*. Berdasarkan Wolfram Mathworld [6], suatu sistem disebut menunjukkan fenomena *chaos* apabila memiliki karakterisitik berikut:

1. Memiliki sekumpulan titik yang tebal (*dense collections of points)* dengan orbit periodik.
2. Sensitif terhadap keadaan awal sistem. Jika nilai awal berubah sedikit saja, maka nilai *chaos* yang dihasilkan akan berbeda signifikan. Properti ini kadang-kadang disebut sebagai efek kupu-kupu (*butterfly* effect).
3. Memiliki topologi yang transitif.



Gambar 1. Ilustrasi Grafik Nilai Pada Sistem *Chaos*. Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2023-2024/32-Pembangkit-bilangan-acak-2024.pdf

F. Hénon Map

H*énon* map adalah pemetaan 2 dimesnsi untuk merepresentasikan perilaku *chaotic*, yang merupakan bentuk sederhana dari sistem Lorentz. H*énon* map dapat diformulasikan dalam persamaan (1) dan (2).

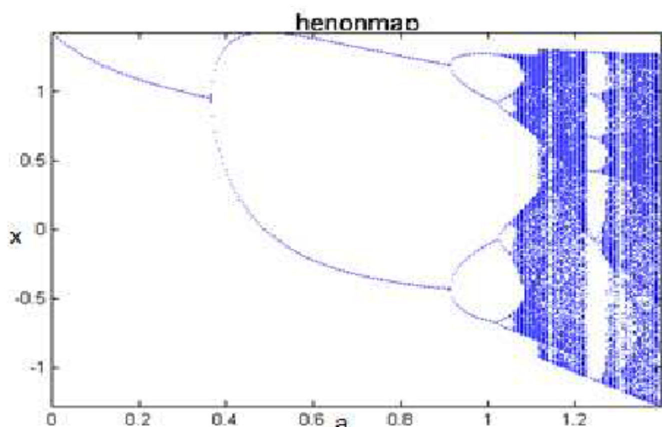
$$x_{i+1} = 1 - ax_i^2 + y_i \quad (1)$$

$$y_{i+1} = bx_i \quad (2)$$

Nilai x_i dan y_i merepresentasikan kordinat awal. Nilai x_{i+1} dan y_{i+1} merepresentasikan kordinat setelah transformasi. Nilai a dan b merepresentasikan nilai parameter *hénon map*. Jika persamaan *hénon map* direduksi menjadi 1 dimensi, maka persamaan *hénon map* akan menjadi seperti persamaan (3).

$$x_{i+1} = 1 - ax_i^2 + bx_i \quad (3)$$

Persamaan Hanon yang kanon menggunakan nilai a=1,4 dan b=0,3.



Gambar 2. *Bifurcation Diagram Hénon Map*.
Sumber: https://www.researchgate.net/figure/Bifurcation-diagram-for-Henon-map_fig1_224926941

III. IMPLEMENTASI

Bagian ini menjelaskan tentang langkah-langkah untuk membangkitkan bilangan acak menggunakan *hénon map* serta pemanfaatan bilangan acak tersebut untuk membangkitkan *salt*. Implementasi dari makalah ini berupa program untuk membangkitkan *salt* menggunakan *hénon map* dalam bahasa Python.

A. Antarmuka Program

Pada tahap pertama, program membaca argumen yang dimasukkan pengguna ketika menjalankan program. Semua argumen bersifat opsional. Jika pengguna tidak memasukkan nilai untuk suatu argumen, program akan menggunakan nilai *default* argumen tersebut. Berikut contoh *command* untuk menjalankan program beserta argumen program.

```
$ python main.py --x0 0.00096204 --base-iteration 100
```

Keterangan mengenai setiap argumen pada program terdapat pada tabel **III.1**.

Tabel **III.1**: Daftar Argumen Program

Argumen	**Keterangan**	**Nilai *default***
x0	Nilai awal *hénon map*	0.0016408
a	Konstanta a pada *hénon map*	1.4
b	Konstanta b pada *hénon map*	-0.3
size	Ukuran **minimal** *salt* yang dihasilkan (dalam satuan *byte*)	16
base-iteration	Iterasi minimal untuk pembangkitan *salt*	1000

Setelah membaca argumen, program membuat objek dari kelas **HanonMapRNG** yang berfungsi untuk membangkitkan *salt*. Penjelasan lebih lengkap tentang isi kelas tersebut akan dibahas pada bagian berikutnya. Terakhir, program menampilkan *salt* yang sudah dibangkitkan. Kode implementasi antarmuka program dapat dilihat pada **Gambar 3**.

```python
import argparse
from lib.HanonMapRNG import HanonMapRNG


if __name__=="__main__":
    parser = argparse.ArgumentParser(prog="HaonMapRNG",description="Pembangkit Bilangan Acak Berbasis Hanon Map")
    parser.add_argument("--x0",type=float,help="Nilai Awal x0",default='0.0016408') # Just a random number buat nilai default
    parser.add_argument("--a",type=float,help="Nilai konstanta a untuk Hanon Map",default='1.4')
    parser.add_argument("--b",type=float,help="Nilai konstanta b untuk Hanon Map",default='-0.3')
    parser.add_argument("-size",type=int,help="Ukuran minimal (bytes) salt yang dihasilkan",default='16')
    parser.add_argument("-base-iteration",type=int,help="Minimal iterasi yang dilakukan untuk membangkitkan salt",default='1000')

    args = parser.parse_args()
    hanon_rng = HanonMapRNG(args.x0,args.a,args.b,args.size,args.base_iteration)
    salt = hanon_rng.get_salt()
    print("Your salt :",salt)
```

Gambar 3. Kode Sumber Antarmuka Program
Sumber: Dokumentasi Penulis

*B. Kelas **HanonMapRNG***

Seluruh fungsionalitas untuk membangkitkan salt berada pada kelas **HanonMapRNG**. Kelas ini menerima argumen **x**, **a**, **b**, **min_size**, dan **base_iteration** yang masing-masing berkorespondensi dengan argumen masukan pengguna pada saat menjalankan program. Kelas ini juga memiliki dua atribut kelas, yakni:

a. **SIGNIFICANT_DIGIT.** Atribut ini digunakan dalam proses normalisasi pada penentuan jumlah iterasi.
b. **DIGIT_MODULO**. Atribut ini menentukan berapa banyak digit di belakang koma yang diambil untuk menentukan jumlah iterasi.

Pustaka pembangkitan *salt* diimplementasikan dalam bentuk kelas dikarenakan nilai **x** terakhir *hénon map* pada pembangkitan *salt* sebelumnya akan dijadikan nilai awal *hénon map* untuk pembangkitan *salt* berikutnya. Kode implementasi kelas **HanonMapRNG** dapat dilihat pada **Gambar 4**.

```
You, 5 hours ago | 1 author (You)
class HanonMapRNG:
    SIGNIFICANT_DIGIT = 10 ** 7
    DIGIT_MODULO = 10 ** 4
    def __init__(self,x:float,a:float=1.4,b:float=-0.3,min_size:int=16,base_iteration:int=1000) →
    None:
        self.x:float = x
        self.a:float = a
        self.b:float = b
        self.min_size = min_size
        self.base_iteration = base_iteration

>   def get_salt(self)→bytes:    You, 5 hours ago • feat: encode salt to base64 string ···

>   def _get_hanon_map_value(self,n:int)→float:···
```

Gambar 4. Kode Sumber Kelas **HanonMapRNG**
Sumber: Dokumentasi Penulis

C. *Pembangkitan Bilangan Acak*

Pembangkitan bilangan acak menggunakan *hénon map* diimplementasikan dalam metode **_get_hanon_map_value** di kelas **HanonMapRNG**. Metode ini menerima masukan berupa jumlah iterasi untuk pembangkitan bilangan acak. Untuk setiap iterasi, program menghitung nilai **x** selanjutnya menggunakan persamaan (3). Setelah proses iterasi selesai, metode mengembalikan nilai **x** yang terakhir dihitung. Berikut adalah kode implementasi pembangkitan bilangan acak menggunakan *hénon map*.

```
def _get_hanon_map_value(self,n:int)->float:
  for _ in range(n):
    self.x = 1-self.a * (self.x * self.x) +self.b * self.x
  return self.x
```

D. *Pembangkitan Salt*

Pembangkitan *salt* diimplementasikan dalam metode **get_salt** di kelas **HanonMapRNG**. Berikut langkah-langkah yang dilakukan metode **get_salt** untuk membangkitkan *salt*:

1. Inisialisasi variabel **salt** dengan sebuah *bytearray* kosong.
2. Selama panjang *bytearray* di variabel **salt** masih lebih kecil dari nilai atribut **min_size** yang dimasukkan pengguna, lakukan langkah 3 hingga 5.
3. Hitung jumlah iterasi yang akan dilakukan dan simpan hasilnya pada variabel **n**. Jumlah iterasi dihitung dengan mengambil waktu saat ini dalam detik menggunakan fungsi **time()** pada pustaka **time**. Pemanfaatan waktu saat ini dalam penentuan jumlah iterasi dilakukan agar *salt* yang dihasilkan berbeda-beda meski nilai **x** awal *hénon map* sama. Dikarenakan fungsi **time()** mengembalikan waktu saat ini hingga 7 digit dibelakang koma, normalisasikan nilai yang dikembalikan fungsi **time()** dengan mengalikan nilai tersebut dengan atribut kelas bernama **SIGNIFICANT_DIGIT**. Atribut ini memiliki nilai *default* sebesar 10^7. Setelah berhasil dinormalkan, akan diambil sekian digit terbawah nilai tersebut. Jumlah digit yang diambil ditentukan oleh nilai pangkat dari atribut **DIGIT_MODULO. Atribut ini** memiliki nilai *default* 10^4. Dengan kata lain, secara *default* akan diambil 4 digit terbawah nilai dengan cara memodulokan nilai tersebut dengan atribut **DIGIT_MODULO**. Hasil akhirnya kemudian dikonversi menjadi *integer* dan dijumlahkan dengan atribut **base_iteration.**
4. Setelah nilai **n** berhasil didapatkan, panggil metode **_get_hanon_map_value** dengan argumen **n** dan simpan hasilnya ke dalam variabel **hanon_value**.
5. Ubah variabel **hanon_value** ke dalam representasi *byte* dengan menggunakan fungsi **pack()** dari pustaka **struct**, kemudian ubah menjadi *bytearray*. Konkatenasi hasilnya ke dalam variabel **salt**.
6. Setelah panjang **salt** sudah lebih besar atau sama dengan nilai atribut **min_size**, *encode* **salt** ke dalam representasi **base64** dengan menggunakan fungsi **b64encode()** dari pustaka **base64**. Hasil *encoding* merupakan nilai *salt* yang dibangkitkan.

Berikut adalah kode implementasi pembangkitan *salt*:

```
def get_salt(self)->bytes:
  salt = bytearray()
    while len(salt)<self.min_size:
      n = self.base_iteration + int(time.time() * HanonMapRNG.SIGNIFICANT_DIGIT) % HanonMapRNG.DIGIT_MODULO
      hanon_value = self._get_hanon_map_value(n)
      salt += bytearray(struct.pack("f",hanon_value))
    return base64.b64encode(salt)
```

IV. PENGUJIAN DAN ANALISIS

Bagian ini berisi pengujian program berserta analisis hasil pengujian. Pengujian dilakukan dengan mebangkitkan 1.000 *salt*, lalu dilakukan analisis apakah terjadi *collision*, waktu rata-rata untuk membangkitkan satu buat *salt* , serta waktu total pembangkitan 1.000 *salt*.

A. *Pengujian*

Pengujian dilakukan dengan menggunakan konfigurasi seperti pada tabel **1V.I**.

Tabel **IV.1**: Konfigurasi Pengujian

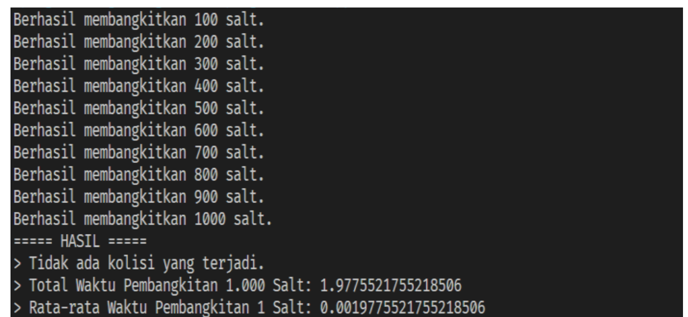
Argumen	**NIlai**
x0	0,009260419
a	1,4

b	-0,3
size	16
base-iteration	100

Dikarenakan memeriksa 1.000 *salt* secara manual sangat memakan waktu, pengujian dilakukan dengan membuat program Python yang memanggil kelas **HanonMapRNG**. Berikut adalah kode program untuk pengujian:

```
x0 = 0.009260419
a = 1.4
b = -0.3
size = 16
base_iteration = 100
# Inisialisasi RNG
rng = HanonMapRNG(x0,a,b,size,base_iteration)
# Melakukan Pengujian
avg_time = 0
total_time = 0
salts = []
for i in range(1_000):
    # Bangkitkan salt
    start_time = time.time()
    salt = rng.get_salt()
    end_time = time.time()
    salts.append(salt)
    # Hitung waktu yang dibutuhkan
    elapsed_time = end_time - start_time
    total_time += elapsed_time
    avg_time = total_time / (i+1)
    if (i+1) % 100 == 0:
        print(f"Berhasil membangkitkan {i+1} salt.")
print("="*5,"HASIL","="*5)
# Tes Kolisi
salts_set = set(salts)
if len(salts) == len(salts_set):
    print("> Tidak ada kolisi yang terjadi.")
else:
    print(f"> Terdapat {len(salt)-len(salts_set)} kasus kolisi.")
# Statistik Waktu
print("> Total Waktu Pembangkitan 1.000 Salt:",total_time)
print("> Rata-rata Waktu Pembangkitan 1 Salt:",avg_time)
```

Setelah program dijalankan, didapatkan hasil pengujian seperti terlampir pada **Gambar 7**.



Gambar 5. Hasil Pengujian
Sumber: Dokumentasi Penulis

B. *Analisis Kolisi*

Pengecekan kolisi dilakukan dengan membuat himpunan yang berisi *salt* unik. Hal ini dapat dilakukan dengan membuat variabel bertipe *set* pada Python. Jika terdapat kolisi, isi *set* yang berisi *salt* unik akan lebih kecil dibandingkan *list* yang berisi semua *salt* yang sudah dibangkitkan. Dari hasil pengujian muncul pesan bertuliskan “Tidak ada kolisi yang terjadi.”. Hal ini berarti isi *set* yang berisi elemen *salt* unik sama dengan *list* yang berisi semua *salt* yang sudah dibangkitkan. Dengan demikian, telah dibuktikan bahwa tidak terjadi kolisi pada pembangkitan 1.000 *salt* menggunakan pembangkit bilangan acak berbasis *hénon map*. Hal ini juga membuktikan efektifitas *hénon map* dalam membangkitkan *salt* yang aman dari kolisi.

C. Waktu Eksekusi

Dari pengujian didapatkan informasi waktu eksekusi seperti berikut:

```
> Total Waktu Pembangkitan 1.000 Salt: 1.9775521755218506
> Rata-rata Waktu Pembangkitan 1 Salt: 0.0019775521755218506
```

Untuk membangkitkan 1.000 *salt* program membutuhkan waktu sekitar 1,98 detik dengan rata-rata waktu pembangkitan 1 *salt* adalah 1,98 milidetik. Dari data ini, terlihat bahwa pembangkitan *salt* menggunakan pembangkit bilangan acak yang memanfaatkan *hénon map* memiliki kinerja yang bagus.

V. Kesimpulan dan Saran

Dari implementasi dan pengujian yang sudah dilakukan, dapat disimpulkan bahwa pembangkitan bilangan acak dengan memanfaatkan *hénon map* dapat digunakan dalam pembangkitan *salt*. Pembangkit bilangan acak mampu membangkitkan *salt* secara efektif dan efisien. Hal ini terlihat dari tidak adanya kolisi dalam pembangkitan 1,000 *salt* serta waktu pembangkitan satu buah *salt* yang terhitung kecil, yakni hanya 1,98 milidetik. Ketiadaan kolisi sekaligus membuktikan sifat *chaotic* dari *hénon map* yang membuat *hénon map* sangat berguna dalam pembangkitan bilangan acak. Waktu pembangkitan *salt* yang singkat juga membuat metode ini dapat dijadikan alternatif untuk membangkitkan *salt* yang banyak dalam waktu singkat namun tetap *secure.*

Secara teori, program untuk membangkitan *salt* menggunakan *hénon map* dapat dijalankan secara multiproses atau multi-*thread* sehingga bisa menangani banyak permintaan pembangkitan *salt* secara bersamaan. Namun, ada kemungkinan terdapat kasus dua buah *thread* atau proses memanggil fungsi **time()** secara bersamaan. Hal ini berpotensi membuat kedua *thread* atau proses tersebut membangkitkan *salt* yang sama. Perlu ada pengembangan lebih lanjut agar pembangkitan bilangan acak tetap tahan terhadap kolisi sekalipun dijalankan secara multiproses atau multi-*thread.* Sebagai penutup, penulis berharap makalah ini dapat berkontribusi terhadap kemajuan Indonesia dan ilmu kriptografi.

Repositori Kode Sumber

Kode sumber untuk implementasi disimpan pada GitHub dan dapat diakses menggunakan tautan berikut:

https://github.com/Wiradhika6051/Tugas-5-Kriptografi

Ucapan Terima Kasih

Puji syukur penulis panjatkan kepada Allah Swt. Karena atas rahmat dan karunia-Nya penulis dapat menyelesaikan makalah ini. Penulis juga ingin mengucapkan terima kasih kepada Bapak Dr. Ir. Rinaldi Munir, selaku dosen pengampu mata kuliah IF4020 Kriptografi atas ilmu yang telah diberikan selama berjalannya mata kuliah IF4020. Ilmu yang diberikan beliau sangat membantu penulis dalam menyelesaikan makalah ini. Terakhir, penulis juga ingin mengucapkan terimaksih sebesar-besarnya kepada keluarga dan kawan penulis atas dukungannya selama penulis mengerjakan makalah ini.

Referensi

[1] OWASP. Methods for Enhancing Password Storage. Diakses pada 05 Juni 2024, dari https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#methods-for-enhancing-password-storage

[2] Arias, D. Adding Salt to Hashing: A Better Way to Store Passwords . Diakses pada 05 Juni 2024, dari https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords

[3] Sargent, W. The Right Way to Use SecureRandom. Diakses pada 05 Juni 2024, dari https://tersesystems.com/blog/2015/12/17/the-right-way-to-use-securerandom/

[4] Munir, R. (2024). *Pembangkit Bilangan Acak*. Diakses pada 05 Juni 2024

[5] B. Williams, R. E. Hiromoto and A. Carlson, "A Design for a Cryptographically Secure Pseudo Random Number Generator," 2019 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), Metz, France, 2019, pp. 864-869, doi: 10.1109/IDAACS.2019.8924431.

[6] Weisstein, Eric W. "Chaos." From MathWorld--A Wolfram Web Resource. https://mathworld.wolfram.com/Chaos.html**.** Diakses pada 09 Juni 2024.

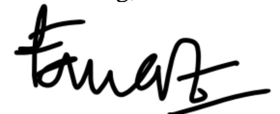
[7] Siswanto, A. , Katuk, N. , dan Ku-Mahamud, K. R. "Chaotic-Based Encryption Algorithm using Henon and Logistic Maps for Fingerprint Template Protection". International Journal of Communication Networks and Information Security (IJCNIS), vol.12, no.1, 2020.

[8] Text Steganography Using Lsb Insertion Method Along With Chaos Theory - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Bifurcation-diagram-for-Henon-map_fig1_224926941 [accessed 12 Jun, 2024]

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, 12 Juni 2024



Fawwaz Anugrah Wiradhika Dharmasatya
13520086