Predictably Random

James Roper

Atlassian

The Perils of Psuedorandom numbers in Web Security

Opinions on PRNG

the problem … was that it was predictable due to the seeding either not happening or the seed value being recoverable … So just using a better seed for srand() should help there, as I understand. Or am I missing something?

Comment from a Firefox developer

Opinions on PRNG

The important thing is that we choose a seed that is sufficiently hard to guess.

Myself, 2 years ago

Maybe this is you

- Exploiting random number generators is doable, but hard
- Securely generating random numbers is all about choosing a good seed
- There are simple techniques that can be used to choose a good seed
- You need a lot of maths to know anything about random number generators

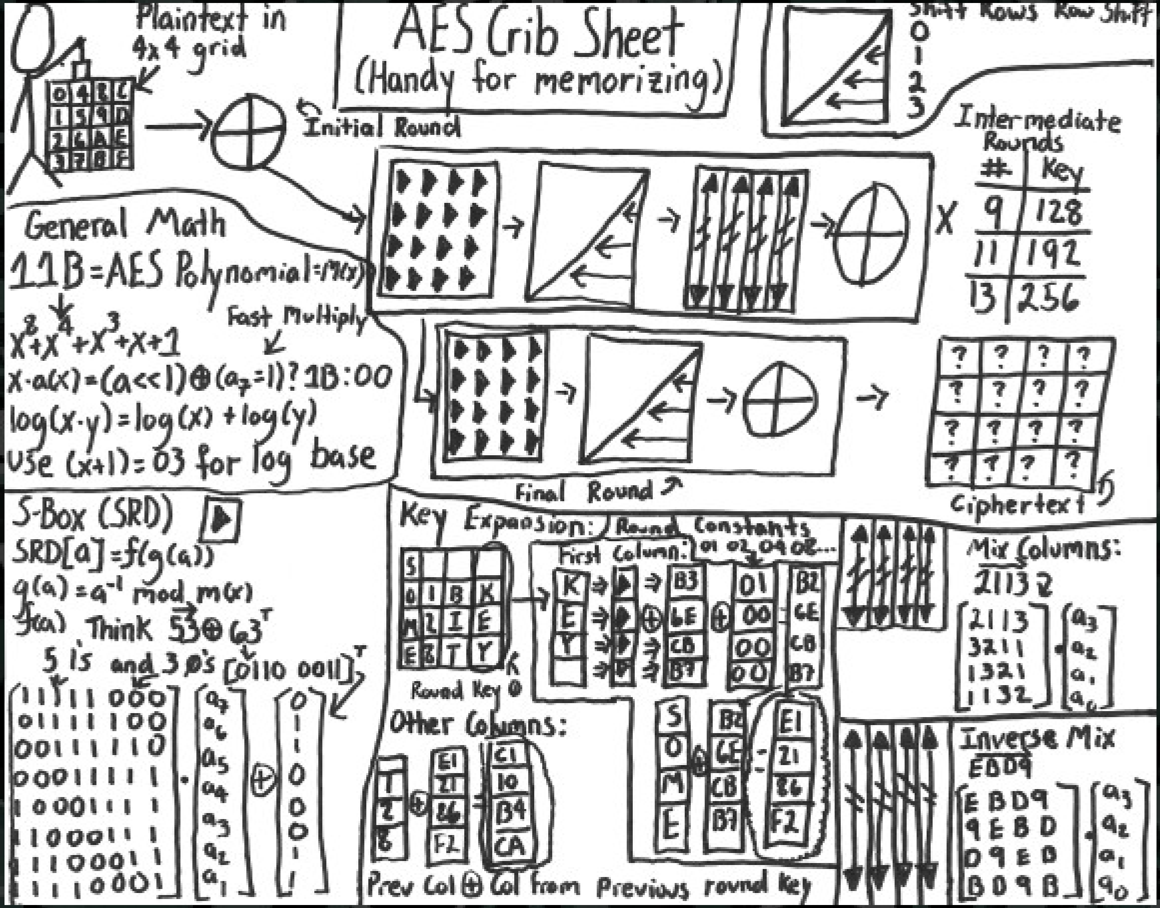
You will learn

- How easy it is to exploit applications that use random number generators badly
- A secure seed is not enough to generate secure random numbers
- Choosing a secure seed is difficult
- How to securely generate random numbers

Maths

- This presentation will have very little maths
    - Multiplication
    - Addition
    - Bit masking
    - Bit shifting
    - Some binary/hex

Cryptography



http://www.moserware.com/2009/09/stick-figure-guide-to-advanced.html

Web Developers

- They don't:
  - Understand cryptography
  - Want to understand cryptography
  - Need to understand cryptography
- Or do they?

Tokens

- Small amount of random data
- Used for identification
- Must be hard to guess

Tokens on the web

- Session tracking
- RPC authentication
- Initial password
- Password reset
- Remember me
- Email address verification
- OAuth
- CAPTCHA
- SSO
- Two factor authentication
- OpenID
- XSRF protection

… and the list goes on

A simple web app

The token generator

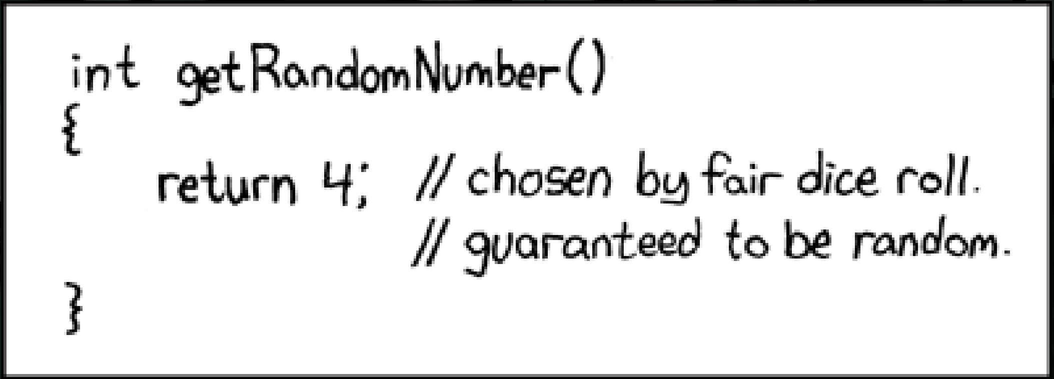
```java
import java.util.Random;

public class TokenGenerator {

    private final Random random = new Random();

    public String generateToken() {
        return Long.toHexString(random.nextLong());
    }
}
```

The token generator

- Generates 64 bit hex encoded tokens
- At first glance, appears to generate 2^{64} possible tokens
- Would take millenia to brute force, right?

java.util.Random

- Linear congruential PRNG
- Uses 48 bit seed
- Is it bad?



```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

- That all depends on what you want to use it for

Linear Congruential PRNG

- Mantains a seed or state with n bits
- On each call to next:
    - Multiply seed by some prime number
    - Add some other prime number
    - Trim back down to n bits
    - … and now you have your next seed
- If you choose the right numbers to multiply and add, you get an even spread of random numbers

In Binary…

```
Seed:              111111100110011100110101110110110100110100011011
Multiplier:                     10111011110111011001110011001101101 *
   ----------------------------------------------------------------
   1111101000011111011000001110010001110100100010100001011001111111
Addend:                                                        1011 +
   ----------------------------------------------------------------
   1111101000011111011000001110010001110100100010100001011010001010
Bit mask:          111111111111111111111111111111111111111111111111 &
   ----------------------------------------------------------------
New seed:          011000001110010001110100100010100001011010001010
```

But we wanted a long?

- java.util.Random generates two 32 bit ints, and puts them next to each other
- So, a long actually contains two tokens
- To generate an int, it bitshifts the seed to the right by 16 bits

```
Seed One:          111111100110011100110101110110110100110100011011
Seed Two:          011000001110010001110100100010100001011010001010
Next Long:
   1111111001100111001101011101101101100000111001000111010010001010
```

Exploiting our app

- Given a single token, can we predict the next token?
- If we can guess the seed, yes!
- But we only have 32 bits of the 48 bit seed
- The bit shift discarded 16 bits
- That means we only have to try 65536 possible seeds

Pseudocode

```
a = first 32 bits of token
b = second 32 bits of token
for i = 0 to 65535:
    seed = (a << 16) + i
    if (nextInt(seed) == b):
        // We've found the seed
        print seed

function nextInt(seed):
    return ((seed * multiplier + addend) & mask)
           >>> 16
```

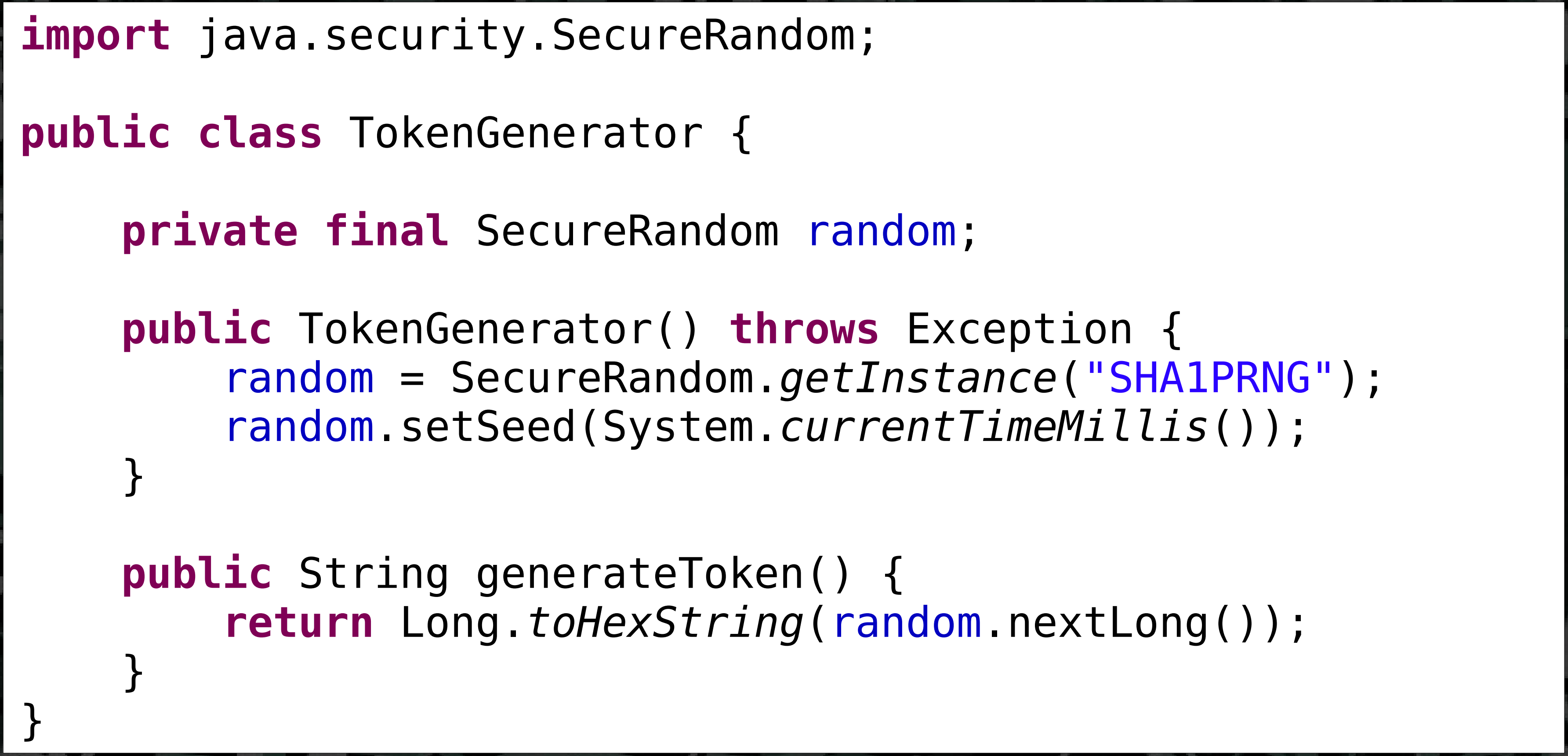
This runs in less than 10ms!

Demo

Rule #1

- Don't use a PRNG for which the internal state can be guessed based on its output
    - This means looking for a PRNG that is labelled 'cryptographically secure'
    - Or, use an entropy based RNG

Second attempt



```
import java.security.SecureRandom;

public class TokenGenerator {

    private final SecureRandom random;

    public TokenGenerator() throws Exception {
        random = SecureRandom.getInstance("SHA1PRNG");
        random.setSeed(System.currentTimeMillis());
    }

    public String generateToken() {
        return Long.toHexString(random.nextLong());
    }
}
```

java.security.SecureRandom

- Platform dependent, default on Windows is SHA1PRNG
- Uses 160 bit seed
- Uses the SHA1 hashing algorithm to update the seed on each call to next
- Is considered to be cryptographically secure
- The algorithm is only as strong as the seed seeding it

Exploiting our app

- The initial seed is the time at which the app started
- There may have been a few tokens generated since we generated ours
- If we can guess the time at which the app started, and guess the maximum tokens generated, we can brute force the initial seed

Pseudo code

```
a = first 32 bits of token
b = second 32 bits of token
t = earliest possible application start time
while true:
    r = SecureRandom.getInstance(”SHA1PRNG”);
    r.setSeed(t)
    for i = 1 to 100:
        if (random.nextInt() == a and
                random.nextInt() == b):
            // We've found the seed
            print t
    t++
```

May take minutes/hours/days depending on how accurate our start time estimate is

Demo

Rule #2

- Don't use a seed that can be guessed
  - The seed should be entropy based
  - Use an entropy source written by the experts
  - Always read the docs on how a CSPRNG should be used

Best practices

- Never use a home brewed random number generator for anything to do with security
- Always read up on what CSPRNG are available
- Always make sure that you are using a CSPRNG as intended to be used – for SecureRandom, that means not calling setSeed(), it will seed itself securely.

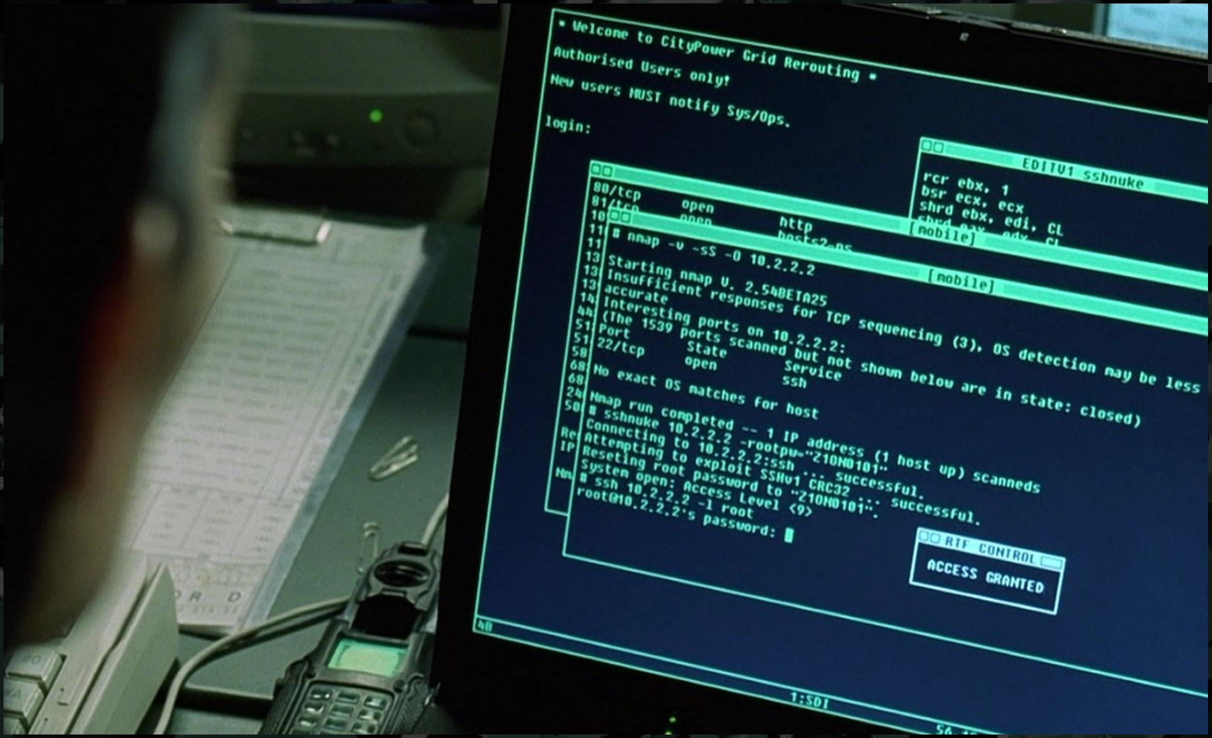
Best practices

- Use automated tools such as checkstyle to ensure insecure generators are not used
- Incorporate code reviews into your development process
- Educate developers frequently on security topics, for example, run brown bag sessions

CSPRNG for your language

Language	Insecure	CSPRNG
Java	java.util.Random – Linear Congruential	java.security.SecureRandom - /dev/urandom, SHA1PRNG
Ruby	rand() - Mersenne Twister	ActiveSupport::SecureRandom – openssl, /dev/urandom/, Win32 CryptGenRandom
Python	random() - Mersenne Twister	os.urandom() - /dev/urandom, Win32 CryptGenRandom

Questions?



```
* Welcome to CityPower Grid Rerouting *
Authorised Users only!
New users MUST notify Sys/Ops.

login:
```

```
rcr ebx, 1
bsr ecx, ecx
shrd ebx, edi, CL
```

```
# nmap -v -sS -O 10.2.2.2
Starting nmap V. 2.54BETA25
Insufficient responses for TCP sequencing (3), OS detection may be less
accurate
Interesting ports on 10.2.2.2:
(The 1539 ports scanned but not shown below are in state: closed)
Port       State       Service
22/tcp     open        ssh

No exact OS matches for host

Nmap run completed -- 1 IP address (1 host up) scanneds
# sshnuke 10.2.2.2 -rootpw="Z1ON0101"
Connecting to 10.2.2.2:ssh ... successful.
Attempting to exploit SSHv1 CRC32 ... successful.
Reseting root password to "Z1ON0101".
System open: Access Level <9>
# ssh 10.2.2.2 -l root
root@10.2.2.2's password: █
```

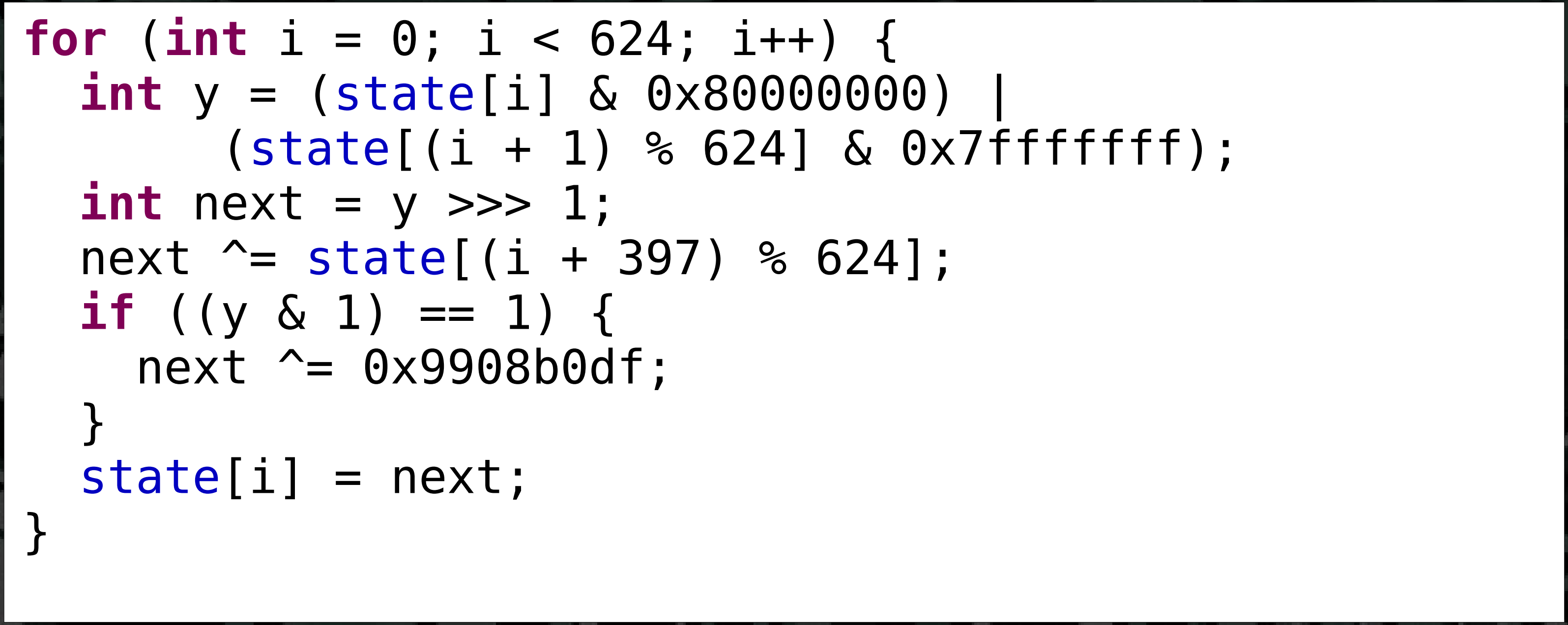
ACCESS GRANTED

Supplement: The Mersenne Twister

- Uses an internal state of 624 32 bit integers
- Hands each integer out sequentially, applying a fuction to even out distribution
- After handing out all 624 integers, applys a function to the internal state to get the next 624 integers

Generating the next state

- Uses bit shifting, bit masking and xor operators



```
for (int i = 0; i < 624; i++) {
  int y = (state[i] & 0x80000000) |
       (state[(i + 1) % 624] & 0x7fffffff);
  int next = y >>> 1;
  next ^= state[(i + 397) % 624];
  if ((y & 1) == 1) {
    next ^= 0x9908b0df;
  }
  state[i] = next;
}
```

Getting the next int

- Obtaining the next int involves applying the following algorithm to the integer:

```
int tmp = state[current];
tmp ^= tmp >>> 11;
tmp ^= (tmp << 7) & 0x9d2c5680;
tmp ^= (tmp << 15) & 0xefc60000;
tmp ^= tmp >>> 18;
```

Determining the internal state

- Obtain 624 consecutive integers
- Reverse the transformation applied to each

Reversing the transformation

- The reverse of an xor operation is applying it again: X ^ Y ^ Y = X
- Take each of the four xors in order, and see if we can unapply them

Transformation Step 4

- tmp ^= tmp >>> 18
- In binary:

```
10110111010111100111111001110010
                    tmp
00000000000000000010110111010111100111111001110010
                    tmp >>> 18
10110111010111100101001110100101
                    tmp ^ (tmp >>> 18)
```

Transformation Step 4

- The first 18 bits of the result is the first 18 bits of the original number
- The next 14 bits can be obtained by xoring the result with the first 18 bits bitshifted to the right
- We can generalise this for any number of bits, and so solve for step 1 too

Undoing Right Bitshift

```
int unBitshiftRightXor(int value, int shift) {
  int i = 0;
  int result = 0;
  while (i * shift < 32) {
  int partMask = (-1 << (32 - shift)) >>> (shift * i);
    int part = value & partMask;
    value ^= part >>> shift;
    result |= part;
    i++;
  }
  return result;
}
```

Undoing Left Bitshift

- tmp ^= (tmp << 15) & 0xefc60000
- This is similar to undoing the right bitshift, except we need to apply the mask each time we unapply

Undoing Left Bitshift

```
int unBitshiftLeftXor(int value, int shift, int mask) {
  int i = 0;
  int result = 0;
  while (i * shift < 32) {
    int partMask = (-1 >>> (32 - shift)) << (shift * i);
    int part = value & partMask;
    value ^= (part << shift) & mask;
    result |= part;
    i++;
  }
  return result;
}
```

Putting it all together

```
int value = output;
value = unBitshiftRightXor(value, 18);
value = unBitshiftLeftXor(value, 15, 0xefc60000);
value = unBitshiftLeftXor(value, 7, 0x9d2c5680);
value = unBitshiftRightXor(value, 11);
```

Questions?

For more information, please visit

http://jazzy.id.au/default/tags/prng