Predictably Random

James Roper
Atlassian

The Perils of Pseudorandom 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
This presentation will have very little maths
- Multiplication
- Addition
- Bit masking
- Bit shifting
- Some binary/hex
Cryptography

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?

That all depends on what you want to use it for

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

- Maintains 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...

<table>
<thead>
<tr>
<th>Seed:</th>
<th>11111110011011011011011010111011011011010011001101111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier:</td>
<td>10111011110111011001110011001101101</td>
</tr>
<tr>
<td>Addend:</td>
<td>11111010000111101100000111001000110100100010100001011001111111</td>
</tr>
<tr>
<td>Bit mask:</td>
<td>111111111111111111111111111111111111111111</td>
</tr>
<tr>
<td>New seed:</td>
<td>011000001110010011101001001010001011010011010</td>
</tr>
</tbody>
</table>
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: 11111110011001110011010111011011
Seed Two: 011000001111001000111010010001010
Next Long: 11111110110011110011010111011011
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

```plaintext
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!
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
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

```plaintext
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
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

<table>
<thead>
<tr>
<th>Language</th>
<th>Insecure</th>
<th>CSPRNG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Java</strong></td>
<td>java.util.Random - Linear Congruential</td>
<td>java.security.SecureRandom - /dev/urandom, SHA1PRNG</td>
</tr>
<tr>
<td><strong>Ruby</strong></td>
<td>rand() - Mersenne Twister</td>
<td>ActiveSupport::SecureRandom - openssl, /dev/urandom/, Win32 CryptGenRandom</td>
</tr>
<tr>
<td><strong>Python</strong></td>
<td>random() - Mersenne Twister</td>
<td>os.urandom() - /dev/urandom, Win32 CryptGenRandom</td>
</tr>
</tbody>
</table>
Questions?
Supplement: The Mersenne Twister

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

- Uses bit shifting, bit masking and xor operators

```c
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:

```c
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
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

- \( \text{tmp} \mathbin{\^=} \text{tmp} \gg 18 \)
- In binary:

\[
\begin{align*}
\text{tmp} &= 1011011101011110011111001110010 \\
\text{tmp} \gg 18 &= 0000000000000000001011011010111001110010 \\
\text{tmp} \mathbin{\^} (\text{tmp} \gg 18) &= 10110111010111100101001110100101
\end{align*}
\]
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.
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) & 0xeefc60000
- This is similar to undoing the right bitshift, except we need to apply the mask each time we unapply
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

```c
int value = output;
value = unBitshiftRightXor(value, 18);
value = unBitshiftLeftXor(value, 15, 0xeefc60000);
value = unBitshiftLeftXor(value, 7, 0x9d2c5680);
value = unBitshiftRightXor(value, 11);
```
Questions?

For more information, please visit

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