(In)Security of Java SecureRandom Implementations

M. Cornejo[1] S. Ruhault[2]

[1] École Normale Supérieure, INRIA, Paris, France

[2] DI/ENS, ENS-CNRS-INRIA and Oppida, France

Journées Codage et Cryptographie, 2014

Outline

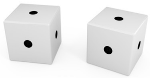
Motivations

Need for randomness

- key generation
- encryption (paddings, IV)
- signature (DSA)
- security protocols (nonces)

Recent vulnerabilities

- Mind your Ps and Qs
- OpenSSL PRNG bug on Debian
- Android PRNG bug



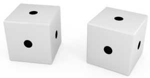
Motivations

Need for randomness

- key generation
- encryption (paddings, IV)
- signature (DSA)
- security protocols (nonces)

Recent vulnerabilities

- Mind your Ps and Qs
- OpenSSL PRNG bug on Debian
- Android PRNG bug



$\boxed{\text{Need for Randomness} \implies \text{Need for Security Analysis of PRNGs}}$

PRNG Security Model

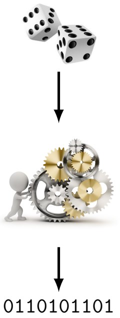
How to model a PRNG ?

Two operations

- input collection $I \rightarrow$ PRNG
- output generation PRNG $\rightarrow R$

Where

- R are contructed to be random
- I are not supposed random
- Operations are not synchronised



PRNG Security Model

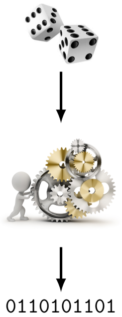
How to model a PRNG ?

Two operations

- input collection $I \rightarrow$ PRNG
- output generation PRNG $\rightarrow R$

Where

- R are contructed to be random
- I are not supposed random
- Operations are not synchronised



Need for an internal state S, s.t. $(I, S) \rightarrow S \rightarrow (R, S)$

Entropy is collected in S, Output is generated from S

PRNG Security Model

Dodis et al PRNG Model

A PRNG is a triple of algorithms (setup, refresh, next):

- setup, seed generation algorithm
- refresh, entropy collecting algorithm, $(S, I) \rightarrow S'$
- next, output algorithm, $S \rightarrow (R, S')$

Where :

- seed is a public parameter
- I is an input
- S and S' are values of the internal state
- R is the output of the PRNG

Reference

Security Analysis of PRNG With Input:`/dev/random` is not Robust. [DPRVW], ACM-CCS'13.

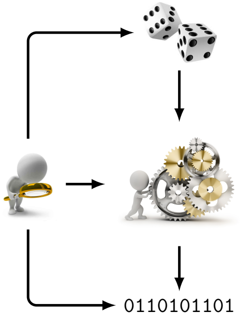
PRNG Security Model

Security properties ?

Attacker $\mathcal{A}$ can:

- ask for outputs: $S \rightarrow (R, S')$
- compromise inputs: $(S, I) \rightarrow S'$
- compromise internal state: $(S, I) \rightarrow S'$

$\mathcal{A}$ wants to distinguish R from random



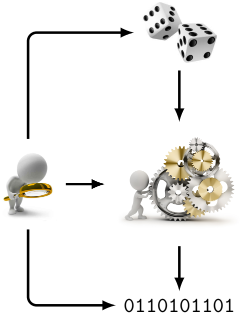
PRNG Security Model

Security properties ?

Attacker $\mathcal{A}$ can:

- ask for outputs: $S \rightarrow (R, S')$
- compromise inputs: $(S, I) \rightarrow S'$
- compromise internal state: $(S, I) \rightarrow S'$

$\mathcal{A}$ wants to distinguish R from random



How do we link this model with Java Implementations ?

Randomness in Java

Java Execution Model

- Java source code → compiled into Java bytecode.
- Java bytecode → executed in a Virtual Machine (JVM).

Java `SecureRandom` Class

- Part of the Java *Cryptographic Architecture*

Providers

Android → `SHA1PRNG`, SUN → `SHA1PRNG`, `NativePRNG`, Bouncycastle → `SHA1PRNG`, . . .

Previous Work

[MMS13] Randomly Failed! The state of randomness in current java implementations. In Topics in Cryptology, CT-RSA 2013

Randomness in Java

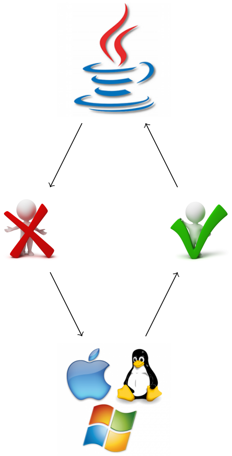
Java Security Model

The Java Security Model relies on:

- Protection of the environment from the Java application.
- But not protection of the Java application from the environment.

A Java application

- runs in a dedicated process
- runs in user mode and is not protected by the kernel.
- can be interrupted (or analysed) by a concurrent process



Randomness in Java

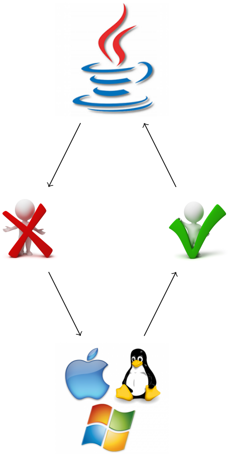
Java Security Model

The Java Security Model relies on:

- Protection of the environment from the Java application.
- But not protection of the Java application from the environment.

A Java application

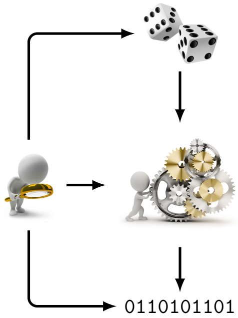
- runs in a dedicated process
- runs in user mode and is not protected by the kernel.
- can be interrupted (or analysed) by a concurrent process



The PRNG Internal State can be compromised !

Implementation Analysis

How do we link this model with Java Implementations ?



Source code

```
field1;
field2;
instruction1;
instruction2;
instruction3;
```

Security Model

Internal State S

setup $\rightarrow S$

refresh$(S, I) \rightarrow S'$

next$(S) \rightarrow (R, S')$

Internal state identification

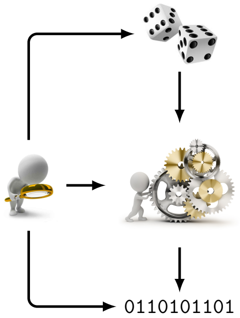
Internal State S	field1; field2;

Algorithms identification

setup	instruction1;
refresh	instruction2;
next	instruction3;

Implementation Analysis

How do we link this model with Java Implementations ?



Source code

```
field1;
field2;
instruction1;
instruction2;
instruction3;
```

Security Model

Internal State S

$\mathsf{setup} \rightarrow S$

$\mathsf{refresh}(S, I) \rightarrow S'$

$\mathsf{next}(S) \rightarrow (R, S')$

Internal state identification

Internal State S	field1; field2;

Algorithms identification

setup	instruction1;
refresh	instruction2;
next	instruction3;

Vulnerabilities can be identified in implementations !

Android SHA1PRNG case

- setup (H_0: SHA1 init vector)
- refresh ($S' = \mathrm{SHA1}(S||I)$)
- Implemented! $H_1 = \mathrm{C}(H_0, I)$
- next ($R = \mathrm{SHA1}(S||\mathrm{ctr})$)
- Implemented! $R = H_2 = \mathrm{C}(H_1, (0||1))$

S:	0	0	0	H_0
S:	I	0	0	H_1
S:	I	1	0	H_2
			R:	H_2

Android SHA1PRNG case

- setup (H_0: SHA1 init vector)
- refresh ($S' = \text{SHA1}(S||I)$)
- Implemented! $H_1 = \text{C}(H_0, I)$
- next ($R = \text{SHA1}(S||\text{ctr})$)
- Implemented! $R = H_2 = \text{C}(H_1, (0||1))$

S:	0	0	0	H_0
S:	I	0	0	H_1
S:	I	1	0	H_2
			R:	H_2

$\boxed{\text{What if } |I| = 512 \text{ ?}}$

Android SHA1PRNG case

- setup (H_0: SHA1 init vector)
- refresh ($S' = H_1 = \mathrm{C}(H_0, I)$)
- next ($R = H_2 = \mathrm{C}(H_1, (0||1))$)

S:	0	0	0	H_0
S:	0	0	0	H_1
S:	0	1	0	H_2
			R:	H_2

Android SHA1PRNG case

- setup (H_0: SHA1 init vector)
- refresh ($S' = H_1 = C(H_0, I)$)
- next ($R = H_2 = C(H_1, (0||1))$)
- next ($R = H_3$)

S:	0	0	0	H_0
S:	0	0	0	H_1
S:	0	1	0	H_2
R:				H_2
S:	0	2	0	H_3
R:				H_3

$$\boxed{R = H_3 = C(H_2, (0||2)) \text{ !!}}$$

Android SHA1PRNG not even pseudo-random (for version < 4.2.2) !

Internal State Compromise

Java Platform Debugger Architecture

- JPDA
  - A standardized infrastructure for third-party debuggers
  - Defines a set of instructions to control the application execution and memory managment
- Debug a running application remotely or locally
- From a different process it is possible to modify the memory

Internal State Compromise

Java Platform Debugger Architecture

- JPDA
  - A standardized infrastructure for third-party debuggers
  - Defines a set of instructions to control the application execution and memory managment
- Debug a running application remotely or locally
- From a different process it is possible to modify the memory

Attack Idea

- Force the JVM in debug mode
- `JAVA_OPTIONS='-Xdebug -Xrunjdwp:transport=dt_socket,address=8998,server=y,suspend=n'`
- Wait for the execution and modify the memory

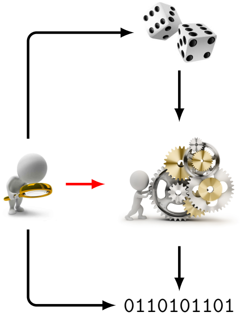
Malicious Code Implementation

Concrete Implementation !

Malicious code can:

- ask for outputs: $S \rightarrow (R, S')$
- compromise inputs: $(S, I) \rightarrow S'$
- compromise internal state: $(S, I) \rightarrow S'$

Malicious code can compromise PRNG !



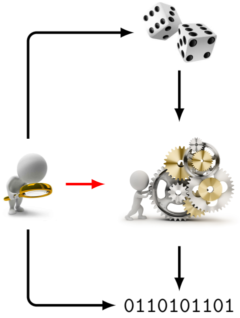
Malicious Code Implementation

Concrete Implementation !

Malicious code can:

- ask for outputs: $S \rightarrow (R, S')$
- compromise inputs: $(S, I) \rightarrow S'$
- compromise internal state: $(S, I) \rightarrow S'$

Malicious code can compromise PRNG !



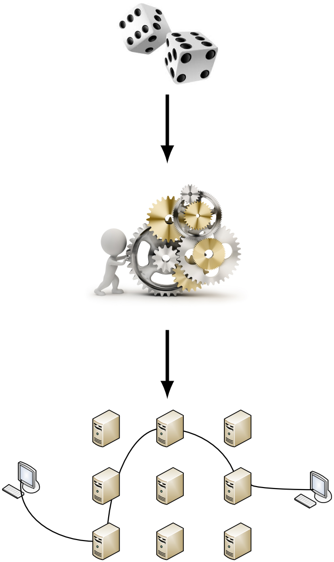
Concrete Attack !

- Only a small part of the internal state needs to be compromised !
- e.g. SUN `SHA1PRNG`: only 32 compromised bits (out of 352) are necessary to compromise the PRNG !
- No remote communication is required !

Attack against a full Java Tor Client

The Tor Network

- Tor is a anonymous and resistant to censorship network.
- Each node encrypts the traffic and send it through a random path.
- Full Open Source Java implementation : Orchid
- Relies on SUN `SHA1PRNG`



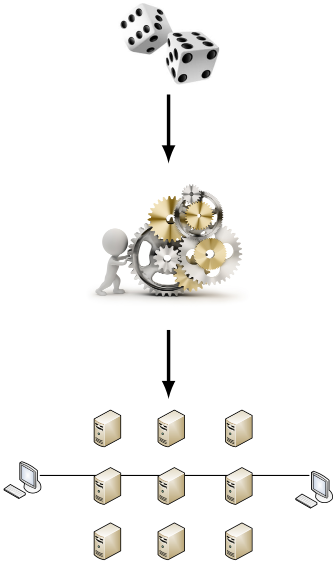
Attack against a full Java Tor Client

The Tor Network

- Tor is a anonymous and resistant to censorship network.
- Each node encrypts the traffic and send it through a random path.
- Full Open Source Java implementation : Orchid
- Relies on SUN `SHA1PRNG`

Attack

- Connect to the application with the JDPA.
- Wait for random path generation.
- Compromise 32 bits (of 352) of S.
- Always use the same path !



Conclusion

Java `SecureRandom` Analysis

- First analysis with a strong security model.
- Concrete implementation of attacks in the security model.
- New vulnerabilities.
- Concrete attack on security applications.

Recommandations

- Update Android !
- Ensure that memory can't be corrupted.
- Rely on system PRNG: e.g. use `NativePRNG`.