Attacks on Random Number Generators

Felix Dörre

Karlsruhe Institute of Technology

Abstract. In this paper we will first show where random numbers are used in today's applications and look at the typical design of a random number generator (RNG) while considering problems that might arise from such a design. Based on this analysis, we will present several different high-profile bugs in applications using random numbers that are caused by improper use of RNGs and their impact on the applications. For example we will cover a problem that existed in Debian's OpenSSL package for years and that has caused RSA keys generated with that package to be extraordinarily weak and therefore unsuited for use in encryption.

Keywords: random number generator, attack, entropy, pseudo random

1 Introduction

In today's programming languages acquiring seemingly random numbers is as easy as a simple standard library call. But for some applications these random numbers are simply not good enough. There have been several major problems due to designing and implementing the random number generator (RNG) wrong or due to improper use of the RNG's output. In order to understand the attacks and problems that will be presented in this paper, we will start by giving an overview over typical uses of random numbers and what the risks of these applications might be. Then we will show, how hard it is to measure the amount of randomness in given data (and hence the quality of random data) and what approaches are taken to do so. Those observations will lead to the question of how to generate good pseudo random numbers and how todays RNGs work. Based on this overview of principles we will look at real-world problems to understand in which ways an application using random numbers may break.

2 Applications of Random Numbers

Random numbers are used in a broad range of applications. These range from cryptographic applications and Monte Carlo algorithms that are dependent on random data to games where events are triggered at random or a program that displays a random quote on startup. The magnitude of consequences of bad randomness varies with different kinds of uses of random data. This paper will focus on applications, where bad randomness has high impact on the operation of the program.

2.1 Cryptography

In cryptography, randomness is primarily used with two aims. The first one uses randomness as data that is not known to an attacker who knows how the application operates. The second one uses randomness to generate numbers that need to be unique randomly in order to make the reuse of such numbers extremely improbable.

One of the main uses of cryptography is establishing connections that are confidential and of integrity. Establishing confidentiality might be achieved by means of encryption. In order to encrypt data from an attacker, it is necessary to choose a secret key at random. This prevents the attacker from taking any advantage of knowing the encrypted data. Bad randomness leads to predictable keys and therefore breaking the confidentiality of the connection. Also initialization vectors for encryption algorithms need to be chosen unique, which is the second of the mentioned aims of using random data.

For establishing integrity, signatures might be used. But some signature algorithms depend unique numbers. The DSA algorithm that creates digital signatures to ensure integrity requires such unique numbers. Repeating numbers can result in losing the private key to an attacker who can then create signatures on his own. This breaks the integrity of the connection.

2.2 Monte Carlo Algorithms

Monte Carlo algorithms are, by definition, algorithms that require random data to operate. In order to see what happens when bad randomness is used, consider a Monte Carlo algorithm that calculates π. The algorithm repeatedly selects two random floating point numbers x, y in $[0, 1)$ and checks, if $x^2 + y^2 < 1$. The ratio of points in the circle to total points converges to $\frac{\pi}{4}$, if good random numbers are used. If the points are not evenly distributed but appear in some kind of pattern the expectation for the ratio might be different. Bad randomness used in Monte Carlo algorithms may result in slow convergence, no convergence or even convergence to another number.

2.3 Online Games

Even for online (multiplayer) games randomness may be of the essence. Information about how a card deck is shuffled or what a software dice will roll might be critical for a player to win a game. If the game is about money, having good randomness is directly about winning or losing money. An online poker game is an example for money that is directly protected by a good shuffled virtual deck.

3 Measurement of Randomness

In order to talk about good randomness and bad randomness, it is important to know what is meant by those terms. This quality of randomness can be measured

in bits by the means of Shannon entropy $H(X)$ of a random variable X:

$$H(X) := -\sum_{x \in X} \mathbb{P}[X = x] \cdot \log_2 \mathbb{P}[X = x]$$

The remaining question is, what exactly is the random variable to analyze. Consider this string of bits as being the output of a RNG:

$$S_1 := 11111111110101111001010001111111111011100\ldots$$

One observes very easily that 1s appear more frequently in S_1 than 0s. Therefore the entropy isn't maximal, when considering every bit as independent probe of the random variable. But now consider this other string:

$$S_2 := 00000001000000000101111111110011111111111\ldots$$

In S_2 1s and 0s appear approximately of equal count. So measuring entropy as for the previous string, by simply considering the frequency of the 1s and 0s, doesn't fit this example. Here the distribution of pairs of consecutive bits would yield more detail by showing that this would be another case of non-maximal entropy.

As one can see, determining the quality of random data can be interpreted as finding patterns in the data and using heuristics to detect common mistakes. Therefore several individual test suites have been developed that all aim to detect different patterns.

The big test suites out there are the DIEHARD(ER) Test Suite [4] and the NIST Test Suite [10]. They contain several creative tests that all aim to detect different patterns, such as:

Birthday spacing draws probes in a finite interval and checks if the distribution of the distances between them is close enough to the expected theoretical distribution.

Run lengths draws floats in $[0, 1)$ and counts the lengths of sequences of increasing and decreasing values (called "runs") and checks their distribution.

Pattern frequency looks for the distribution of patterns of longer length.

Binary matrix rank fills a binary matrix with the random stream and determines its rank.

3Dspheres test draws randomly positioned spheres in a cube and checks the distribution of their distances.

All those tests are executed repeatedly on various samples of outputs of RNGs. If some of them yield bad results in many samples it's very likely that the RNG has a systematic error. So all these tests can only be heuristics to identify data that has an observable pattern. A good definition for pseudo random numbers is therefore: "A generated stream is of good randomness, if an observer with limited resources can not distinguish it from a real random stream".

To understand those systematic errors or patterns better, we will have a look on how a random number generator works and where errors can arise.

4 Generator Principles

A RNG's task is always the same: Take a small amount of entropy and stretch it out in a longer byte stream. To achieve this, many RNGs use the same principle.

Those RNGs consist of an internal state called x_i and a permutation function f. The RNG then works in cycles, where a new internal state (x_{i+1}) is generated in each cycle with $x_{i+1} = f(x_i)$. This principle is illustrated in Figure 1.

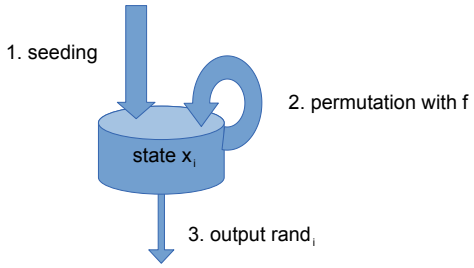

Fig. 1. Principle of a typical random number generator

When a new RNG is initialized, it starts at stage 1 where the random number generator is seeded. This describes the process when the generator gathers initial entropy. This part is required to make the output unpredictable even if an attacker knows the structure of the whole generator. Typically a random number generator initializes the whole internal state (x_0) with entropy. Real world sizes of the internal state are currently around 16-20 bytes.

Then stages 2 and 3 are repeated while more random numbers are needed. These two stages consist of updating the internal state by calculating $x_{i+1} = f(x_i)$ and then determining a part of the internal state x_{i+1}, that is being delivered to the application.

This is already a consideration for the designer of the RNG: If the RNG uses too much of its internal state it may risk becoming predictable. If the RNG uses too little of its internal state, the output is not random from the beginning. It is essential that there is maximal entropy inflow while seeding and good (not too much and not too little) entropy outflow while working.

To show how this balance can be achieved in reality we'll present two different types of pseudo random number generators.

4.1 Linear Congruence Generators (LCGs)

In this special type of RNGs the permutation function takes the form

$$x_{i+1} = (x_i \cdot a + c) \bmod m$$

where $a \in \mathbb{N}$, $c \in \mathbb{N}_0$ and $m \in \mathbb{N}$. In typical implementations, m is a power of 2 and the modulo operation is implemented implicitly by letting the corresponding variable overflow.

The simplicity of this kind of generators has made them very easy to analyze completely. Theorem 1, proved by Hull and Dobell in 1962, shows how to choose the constants perfectly. On the other hand, the simplicity of its design makes LCGs also easy to attack.

Theorem 1 (Hull-Dobell Theorem [5]). *An LCG has a full period length of m, if and only if all of the following is true:*

1. *c and m are relatively prime.*
2. *$a - 1$ is divisible by all prime factors of m.*
3. *$a - 1$ is a multiple of 4 if m is a multiple of 4.*

Let's have a look at the output of an LCG in order to examine its weaknesses. In Figure 2 a very small LCG (13 bit of internal state) has been called 512 times to output a value in $[0, 512)$. The output of each call has been drawn as a single point. On the horizontal axis is the different call, on the vertical axis the outputs of the random number generator.

Although the output looks random at first glance, one can observe the very small sub-periods even though the random number generator has the maximal possible period length of 8192.

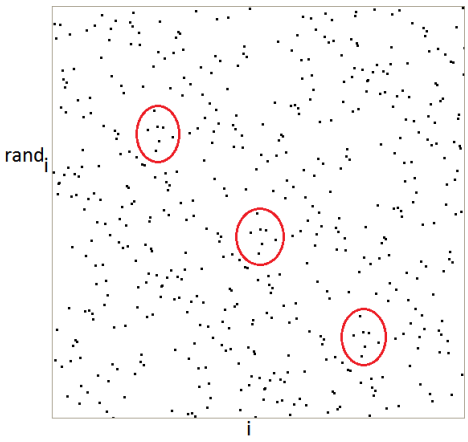

Fig. 2. 512 outputs (i, rand_i) where $\text{rand}_i := x_i \bmod 512$ and $x_{i+1} := x_i \cdot 4805 + 83 \bmod 8192$

Although this observation makes the output look bad, let's have a look at the correlation between two consecutive outputs of this LCG. Figure 3 shows this correlation. For this diagram an x and y value in $[0, 512)$ are randomly chosen

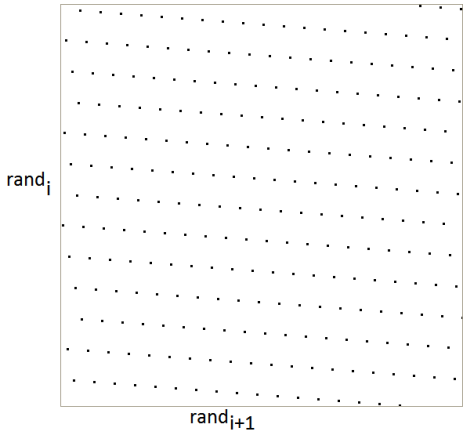

Fig. 3. 512 outputs $(\text{rand}_i, \text{rand}_{i+1})$ where $\text{rand}_i := x_i \mod 512$ and $x_{i+1} := x_i \cdot 4805 + 83 \mod 8192$

and the point with these coordinates is shown on the diagram. One can now see the output of the RNG rand_{i+1} depending on its previous value rand_i.

The reason why all points generated this way are on a finite set of hyperplanes is clear when you take a look at the generating equation. It's principally a planar equation with an additional modulus operation.

As one can see randomly selecting uniformly distributed points on a plane is a task where a LCG is a bad choice for.

The fact that there is a pattern in the output of the random number generator is inherent to the problem. An RNG has a finite amount of entropy and therefore can only produce a finite amount of different output streams. The best chance a random number generator has is to hide its pattern so well that it is hard to compute how the pattern will continue. This leads directly to another class of RNGs, namely hash-based ones.

4.2 Hash-based Generators

In this type of RNG the permutation function f is being chosen so that it contains a cryptographic hash function. In order to understand the benefit a random number generator can get from the hash function, it's essential to understand the properties of a hash function. Here we will use that a hash function is ...

... a one-way function It is hard to calculate $x = f^{-1}(y)$ for a given y
... collision resistant It is hard to calculate x for a given y with $f(x) = f(y)$

The property of being a one-way function ensures that no old outputs of the RNG can be calculated, if the RNG's internal state gets known to an attacker. In order to calculate an old value, the attacker has to derive an old internal state, which is exactly the problem of reversing the one-way function. The collision resistance

helps preventing an attacker from combining the knowledge that he gets from different observations of the RNGs output to a combined knowledge over the internal state of the generator.

Summing it up, a RNG based on a hash function is better suited for the use in an application where good randomness is important, because the properties of the hash function prevent the attacker from deriving statements about consecutive output of the RNG, if the hash function is secure.

5 Attacks

In this section we present the differences and similarities between different kinds of attacks against the scheme that most applications use when dealing with random numbers. This common scheme is shown in Figure 4. The figure shows

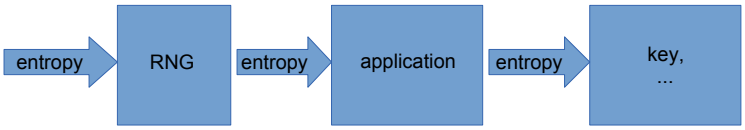

Fig. 4. Scheme of applications using RNGs and the flow of entropy through them

that in most systems entropy flows into the RNG from outside the system. This is the RNGs seeding process. The RNG then uses this entropy to distribute it over a longer byte stream in a way that is computationally hard to reverse. This byte stream is then used by the application that transforms it into a random object. This might be a random prime (for a random key), a random point on a plane (for Monte Carlo algorithms using such points), or a card deck for an online poker game. For analyzing problems in such programs, we will start at the beginning of the entropy flow: the RNGs seeding.

5.1 Attacks on Entropy Flow into the RNG

The main problem of entropy flow into the RNG, is that the inflow might be too low. It is important that enough entropy is gathered here, because entropy can not be increased by computation and therefore the amount of inflow is a limit of the entropy available to the application. Problems reducing the entropy inflow are typically caused by bugs in the application.

Such bugs may occur, because it is still default or fallback behavior to seed the RNG with low entropy. Often these default RNGs use the system time as entropy source. That source doesn't contain much entropy, because the current system time, which is probably close to the real time, is known to an attacker with some resolution. So the only remaining entropy of such a seed are its last few digits. Therefore the resolution of the clock used for seeding is important. Java 8 seeds its default RNG with `System.nanoTime()`, which guarantees no

specific resolution of the timer at all [6], so therefore we have no guarantees on the amount of entropy that is contained in the seed. Visual Basic's `Randomize()` also seeds with an arbitrarily-resolved system time [11] when called with no explicit seed. Such default implementations have been used in many old implementations such as the Poker game analyzed in [1]. Nevertheless both documentations (Java and Visual Basic) strongly recommend to use other RNGs than the default one when good quality of randomness is required.

A more important and recent example of reduced entropy inflow was a bug in Apache Harmony's implementation of a SHA1-based RNG. This incident is more significant than the problem mentioned above, because this RNG has served as the default secure RNG on all Android OS based devices. The bug causes the RNG to be seeded with 20 bytes of system entropy, of which 12 bytes are accidentally overwritten by the RNG before being used as seed. Thus, the remaining entropy is quite high with 8 byte compared to what is left when seeding with millisecond resolved system time, but it is still significantly lower than the promised 20 bytes.

An even more significant bug of this category has been discovered in the Debian-maintained version of the OpenSSL package. A developer had asked on the OpenSSL mailing list, whether he could remove accesses to uninitialized memory in OpenSSL's RNG. As uninitialized memory was used as a secondary source of entropy in OpenSSL's RNG he was told that he could safely remove the mentioned reference. Accidentally the Debian maintainer removed a second access to uninitialized memory that was used to handle the main entropy inflow into the RNG. Consequently all entropy inflow has been cut off, except the current process' identifier that holds at most 16 bit of entropy. As this RNG is often used to generate cryptographic keys for server applications, many server administrators were required to regenerate all keys that were generated while the RNG was vulnerable.

5.2 Attacks on Entropy Flow out of the RNG

This category of problems covers the part where the applications transfers the randomness into its required random object. In the case of the first example, this random object is a deck of playing cards that should be shuffled for the online poker game that has been analyzed in [1]. The code in Listing 1.1 shows how the original algorithm worked.

This algorithm suffered from two problems: The first problem is an off-by-one error. When trying to select a random number in $[1, 52]$ the author of the algorithm accidentally selected only a number in $[1, 52)$, because the function $random(x)$ returns a random number in $[0, x)$. This finally yields in preventing all shuffles that have card 52 at place 52 from occurring. This bug is not so severe that it is clearly visible in a random output, but it decreases the randomness of the output.

The other problem arises from the way how the cards are shuffled. Because any card may be randomly swapped with any other card some decks can be generated with different random input. In the case of this shuffle function, this

Listing 1.1. How to not shuffle cards

```
// Fill the deck with unique cards
for(int ctr = 1; ctr <= 52; ctr++){
    card[ctr] = ctr;
}
    // Randomly rearrange each card
for(int ctr = 1; ctr <= 52; ctr++) {
    int random_number = random(51) + 1;
    int tmp = card[random_number];
    card[random_number] = card[ctr];
    card[ctr] = tmp;
}
```

flaw results in a non-uniform distribution of the decks that even further reduces the remaining entropy in the generated deck.

This shows the general 2 problems, that such a transformation function might have that result in the reduced randomness: Leaving some results out (loss of surjectivity), and retuning some results for multiple different inputs (loss of injectivity).

The second example of what can go wrong with the entropy outflow of the RNG is that the RNG might not able to hold onto parts of its entropy long enough, because there is too much entropy outflow. Figure 5 shows the entropy remaining in the given 2 LCGs (the internal state is 13 bits) and how this entropy is reduced by observing the LCGs' selection of a number in $[1, 52]$. Although both LCGs fulfill Theorem 1, one of them is able to hold onto some fractions of its entropy longer than the other one. So still with knowing of how to achieve maximal period length, it is still a question how much information two consecutive outputs should have in common, and how much entropy should be preserved in the RNG for later use.

5.3 Attacks due to Misperceived Resource Bounds

This third kind of problem focuses on the incident with the Dual EC [2] RNG. The Dual EC RNG is basically a hash-based RNG, where the hash function is based on elliptic curve mathematics. Its one-way property however depends on the attacker not knowing numbers that help to invert this function easily.

In the case of the curves used for the Dual EC RNG standard, it is not clear, how the curves were selected, and if anyone knows the numbers that would help reversing the hash function for which they are used. This could be a backdoor in the RNG definition itself.

The current state of Dual EC in different SSL implementations in 2013 has been analyzed in [2]. In the BSAFE library, dual EC is correctly implemented and could therefore be attacked if the secret constants are known. In OpenSSL,

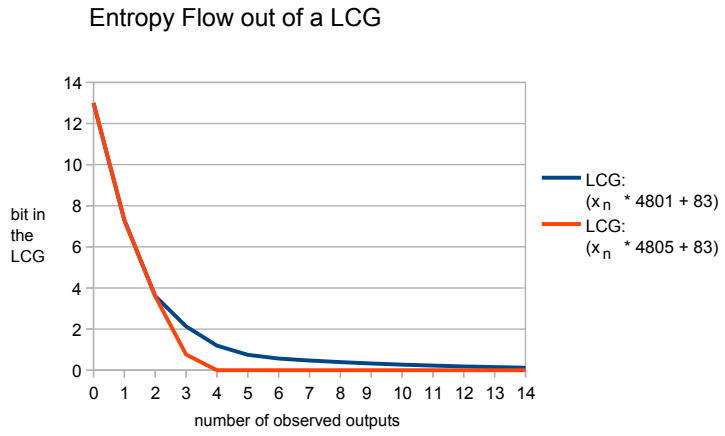

Fig. 5. The entropy remaining in the given LCGs after different counts of output of the LCG (numbers in $[1, 52]$) have been observed. Measured with CHIMP [3]

a small bug prevents usage of Dual EC, if that bug is not fixed locally. In Microsoft's SChannel library, the implementation is not conforming to the standard because it leaves one computation out. However this bug does not prevent attacks but facilitates them.

For a successful attack on the Dual EC RNG an attacker is required to see approximately 30 bytes of consecutive random data to calculate the internal state of the Dual EC RNG. Therefore a proposed TLS-Extension called Extended Random is interesting in this context, because it basically increases the number of visible random bytes in the connection and therefore facilitates the usage of the backdoor, if it exists.

6 General Prevention Techniques

To generally detect bugs like the ones mentioned above, it is essential to know where how much information in the application should flow and is flowing.

Several programs exist that measure entropy flow in programs (e.g., CHIMP [3], leakWatch [3], QIF/SAT [9], QIF/Poly [8]). Some of them measure the entropy flow exactly by considering all execution paths (e.g., CHIMP), but are therefore not usable for bigger applications. Some others sample the target program and therefore are not able to output an upper bound for entropy flow, but only a lower bound (e.g., leakWatch).

Where correct entropy flow through a transformation function is essential (as the shuffling function for the card deck), such functions may also be proved by KeY [7] to be bijective, when seeing the RNG's output as input to the function.

7 Conclusion

Summing it up, it is hard to detect good randomness, because there is principally no way to definitely tell from any data (and therefore the output of any RNG) if its random. Consequently it is hard to acquire good randomness: Real entropy sources are often too slow and the RNG's task to distribute the randomness well enough over its output stream can be very complicated for the RNG's designer and implementor. On top of that, it is hard to use random numbers correctly. All the problems that we showed in this paper are various pitfalls that developers of applications that use random data should be aware of.

References

1. B. Arkin, F. Hill, S. Marks, M. Schmidt, T. Walls, and G. Mcgraw. How we learned to cheat at online poker: A study in software security. 1999.
2. Stephen Checkoway, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, Hovav Shacham, and Matthew Fredrikson. On the practical exploitability of Dual EC in TLS implementations. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 319–335, San Diego, CA, August 2014. USENIX Association.
3. T. Chothia, Y. Kawamoto, C. Novakovic, and D. Parker. Probabilistic point-to-point information leakage. In *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*, pages 193–205, June 2013.
4. Pseudo random number generators. `https://www.ee.oulu.fi/research/ouspg/Frontier_Whitepaper-prng`.
5. T. E. Hull and A. R. Dobell. Random number generators. 4(3):230–254, July 1962.
6. Java 8 Javadoc on java.lang.System.nanoTime(). `http://docs.oracle.com/javase/8/docs/api/java/lang/System.html#nanoTime--`.
7. The KeY Project. `http://key-project.org/`.
8. Vladimir Klebanov. Precise quantitative information flow analysis – a symbolic approach. *Theoretical Computer Science*, 538(0):124–139, 2014.
9. Vladimir Klebanov, Norbert Manthey, and Christian Muise. SAT-based analysis and quantification of information flow in programs. In *Proceedings, International Conference on Quantitative Evaluation of Systems*, volume 8054, pages 156–171. Springer, 2013.
10. A statistical test suite for random und pseudorandom number generators for cryptographic applications. `http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf`.
11. Visual Basic's documentation on Randomize(). `https://msdn.microsoft.com/en-us/library/8zedbtdt%28v=vs.90%29.aspx`.