Pseudo-random number generator in the Linux kernel

Dejan Benedik

February 7, 2017

1 Introduction

There are many programs, applications and protocols that depend on well chosen random numbers to operate safely. If such numbers are compromised, an adversary can break the encryption. The "quality" of the random numbers depends on their source — if the generated number sequences can't be predicted better than by a random chance, they are considered safe.

The sources of random numbers in computers are either hardware random number generators or pseudo-random number generators (PRNG). The former rely on measuring physical sources that are considered random, such as radiation decay, cosmic background radiation or thermal noise. On the other hand, PRNG are simply some algorithms that output appearing-to-be random sequences of numbers based on their own internal state. When the number of different internal states and outputs is big enough, PRNG is considered a cryptographically secure random number generator (CSPRNG).

In this work we describe and analyze the PRNG that's included in the Linux kernel. First we describe pseudo-random number generators and some criteria that ensure their safety. In the next chapter, structure of the Linux PRNG is analyzed in detail. Along with that, we explain some possible vulnerabilities and their impact. In the fourth chapter, there is an overview of challenges that arise with usage of PRNG in virtual machines.

2 Pseudo-random number generators

PRNG is an algorithm that takes a binary input sequence of length l and returns a binary output sequence of length n, where $l \leq n$, but usually $l << n$. It is deterministic, since for a given input, it will always return the same output. A PRNG algorithm f can be defined as:

$$f : \{0,1\}^l \to \{0,1\}^n$$

A random number generator is considered secure when there is no polynomial-time algorithm that would usually return the next RNG output bit, based on

its previous output. In more concrete terms, there would have to be an oracle $o : \{0,1\}^m \to \{0,1\}$, which should predict the $m+1$ bit (based on previous m bits) in more than $\frac{1}{2}$ of iterations. When there is no such oracle, PRNGs are called *cryptographically secure pseudo-random number generators* (CSPRNG). Examples include the Blum-Blum-Shub algorithm, Yarrow (which is used for `/dev/random` device in UNIX operating systems) and `/dev/random` in the Linux kernel.

One might wonder what are the advantages of PRNG as opposed to hardware random number generators (HRNG). First of all, PRNG usually aren't faster and sometimes they rely on hash functions with dubious security, as there is always a threat that they will be broken. Since HRNG is usually a black box[1], the user must trust both the HRNG implementation and its manufacturer. On the other hand, the Linux source code is available for anyone to inspect. Another reason is cost - especially in small embedded devices, adding a HRNG would be prohibitively expensive, both in size and money. Similar issues affect virtualized operating systems that run on servers. PRNG also offers reproducibility - for the given seed, it will always return the same output sequence.

2.1 Criteria for CSPRNG

There are some guidelines for writing cryptographically secure PRNG. First of all, it should be difficult or impossible to analyze its internal state. If an adversary could choose the input entropy, it should still be difficult to predict the PRNG output. The reverse also holds - given some output, an adversary must not be able to figure out the internal state of RNG. It is recommended that the source code is publicly available for scrunity and reviews - that way, some deficiencies and errors can be observed and fixed much sooner. The generator output should also be statistically tested, existence of any repeated patterns would imply that a PRNG is bad.

3 Linux PRNG driver

In this section, we explain the inner workings of Linux PRNG driver, first written in 1994. Instead of symmetric ciphers, the driver primarily used secure hashes because there were many patents and export restrictions imposed by various governments at the time of first implementation. Most of following anaysis is inferred from the source code (which is located in `drivers/char/random.c`) with the help of various emails and articles accessible on `https://lwn.net/`. There were some structural changes done to the random driver in October 2016, with version 4.8 of the kernel. As a result, the following description differs slightly from those in articles [4], [3], [2] and others.

In its essence, the random driver collects randomness (also called entropy) from various available sources and uses it as a seed for the chosen PRNG function. The benefit of such approach is the relative simplicity of changing the

[1] The user can only see the output of HRNG, but not its hardware and software.

PRNG function for another one when its performance or safety are not satisfactory any more.

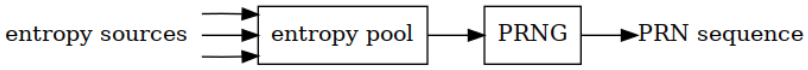

Figure 1: Simplified structure of Linux PRNG driver

First, we have a look at entropy pools, which hold the current entropy. Then there is a brief overview of entropy collection, how it's added to the pools and how it's counted to ensure appropriate level of randomness. Last part of this chapter presents the distinction between `/dev/random` and `/dev/urandom` PRNG devices and where the use of each is appropriate. In short, `/dev/random` is blocking, therefore it returns random sequences only when there is enough entropy.

3.1 Entropy pools

There are 2 main entropy pools in the random driver: *input* and *blocking*. Collected entropy gets mixed in to the former one, while the latter is only used for the blocking `/dev/random` driver. Their sizes are different - *input* storage is 2^7 words (with $32b$ words, that is 4096 bits) large while *blocking* is 2^5 words or 1024 bits large. There used to be an additional entropy pool associated with `/dev/urandom`, but it was removed in Linux 4.8.

Each pool has an entropy counter, which keeps track of the available entropy[2]. When entropy is mixed in or accessed, the counter increases or decreases accordingly. After new entropy is collected, it gets mixed into the *input* pool using a mixing function `mix_pool_bytes`, which is also used for mixing and transferring the entropy to the *blocking* pool.

The function `mix_pool_bytes` implements a twisted generalized feedback shift register (TGFSR). In contrast with LSFR, a GSFR uses the XOR operation with all of the bits from previous state to generate the next state (LSFR uses only some of the bits). Twisted GFSR additionally scrambles the XOR'd state by multiplying it with matrix A, in the kernel this is done using a CRC32 hash function.

$$x_{l+n} = x_{l+m} \oplus x_l A, (l = 0, 1, \dots)$$

It is noted in the source that while this function is not cryptographically strong, it's very fast and good enough for modifying the entropy pool. Its speed is important because entropy gets collected and mixed in during interrupt handling.

3.2 Entropy collection

The kernel tries to collect as many good sources of randomness as possible. Entropy is collected from various events, such as user input (mouse and keyboard),

[2]This is important mostly for the `/dev/random` device.

disk activity and interrupts. Except for the interrupts, each event consists of 3 values: a unique event identifier, *jiffies*, which is a CPU cycle counter that only increments during interrupt handling, and a general CPU cycle counter.

Interrupts get collected in a separate entropy pool called *fast pool*. Its contents get mixed in the *input* pool either every second or every 64 interrupts.

The quality of entropy in the *input* pool can also be improved by mixing in output of available hardware random number generators. Since they are usually black box implementations, the entropy count is not increased.

3.3 Entropy estimation

Entropy estimation for incrementing entropy counters is based on timing differences between *jiffies* of consecutive events. The algorithm first calculates 3 levels of differences, which takes 6 consecutive events:

$$\delta_n^{[1]} = j_n - j_{n-1}$$

$$\delta_n^{[2]} = \delta_n^{[1]} - \delta_{n-1}^{[1]}$$

$$\delta_n^{[3]} = \delta_n^{[2]} - \delta_{n-1}^{[2]}$$

$$e = \log_2 \left(\lfloor min(|\delta_n^{[1]}|, |\delta_n^{[2]}|, |\delta_n^{[3]}|) \rfloor \right)$$

Entropy estimate e is then a logarithm of minimum difference from each level, rounded down by one bit. Additionally, the logarithm is truncated into an interval $[0, 11]$. When the entropy gets mixed into the pool, even if the estimate is zero, the entropy counter is credited with the calculated value.

Users are allowed to add entropy to the random driver by writing to `/dev/random` and `/dev/urandom` devices. In such case, no entropy is counted to prevent attacks on internal state of PRNG.

3.4 `/dev/random`, `/dev/urandom` and `get_random_bytes`

At last, we can describe the user-facing random devices and the `get_random_bytes` system call.

The `random` device is blocking - it returns output only when there is an appropriate amount of entropy in the *blocking* pool. Output is generated by the following algorithm:

1. Hash all bytes of the *blocking* pool using SHA-1. If there is less than The result consists of 5 4-byte words, which is immediately mixed back into the pool to prevent backtracking attacks (when an adversary knows the state of the pool and the current outputs, he could attempt finding previous outputs).

2. The 5 hashed words are then folded using XOR operation to hide any recognizable pattern:

$$output = \big(w_0 \oplus w_3 || w_1 \oplus w_4 || w_{2_{[0\ldots15]}} \oplus w_{2_{[16\ldots31]}} || w_{2_{[16\ldots31]}} || w_3 || w_4\big)$$

Output is 20 bytes large. If the user demands less data, it is truncated, if more, the procedure executes again. Entropy counter for *blocking* pool is then decreased by an appropriate amount.

The `urandom` device used to be quite similar to the `random` device - it had its own *non-blocking* entropy pool that was used in the same way as the *blocking* pool, except that `urandom` didn't stop outputting pseudo-random data when there was not enough entropy. If entropy was available, it was mixed into the *non-blocking* pool. Since Linux 4.8, `/dev/urandom` doesn't have an entropy pool and it uses a modified ChaCha20 stream cipher, which is reseeded either every 5 minutes. The seed data comes either from the *input* pool (if enough entropy is available) or from the internal state of ChaCha20.

The `get_random_bytes` syscall exposes the same function that generates pseudo-random data for `/dev/urandom`.

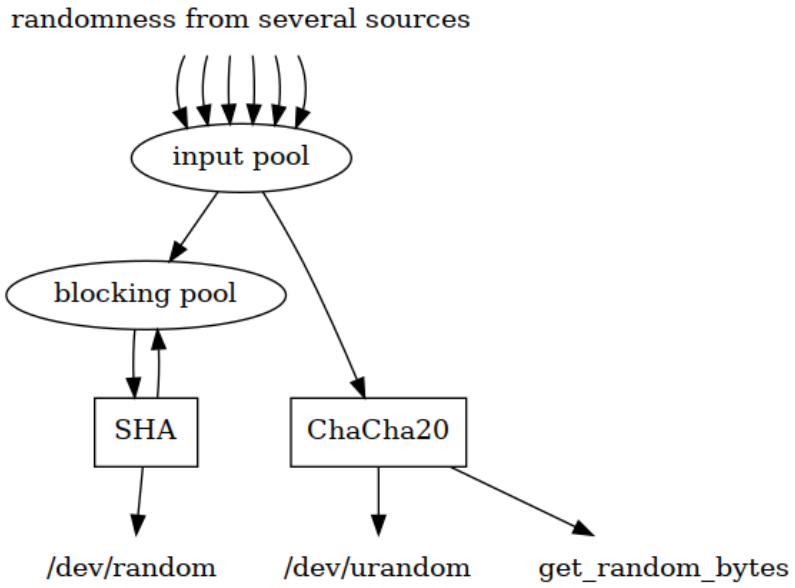

Figure 2: Simple overview of Linux PRNG since the version 4.8

In figure 2 is a summary of PRNG driver in the current Linux kernel. Entropy counters are not shown, as they would overcrowd the figure.

Since it is blocking and as a consequence rather slow, the `/dev/random` device is only recommended when randomness is critical for the security of user's application and the user can afford to wait - examples include one-time pads and generation of various long-lived cryptographic keys. The design of this device comes from a time when cryptographers weren't exactly sure about the safety of hash functions. Consequentially, this device operates only on gathered entropy.

For this reason, it could be considered a true random number generator. On the other hand, `/dev/urandom` is a PRNG and the recommended source of random numbers for any other use due to its speed.

3.5 Overview of possible vulnerabilities

In this section, we look at some vulnerabilities that might affect the random driver due to its architecture.

3.5.1 Entropy starvation and denial of service

First issue is related to the non-blocking nature of `/dev/urandom` - if there is not enough entropy in the *input* pool, the ChaCha20 simply gets reseeded with its own internal state. Draining the *input* pool is easily done by continuously reading from `/dev/random`. As a result, no new randomness is introduced into the `urandom` device. Overall this is not a serious issue because an adversary still has to break the ChaCha20 stream cipher.

There used to be another problem possible in the old `/dev/urandom` implementation. When data was read from that device, its *non-blocking* pool was reseeded with any available entropy from the *input* pool. Constant and continuous accessing of `urandom` device could therefore drain all entropy from the *input* pool. As a result, `/dev/random` would block indefinitely. This is not true anymore with the new design, as the ChaCha20 algorithm reseeds only in predefined intervals of time.

3.5.2 Direct cryptoanalytic attack

The Linux RNG relies on SHA-1 and ChaCha20, either of them would need to be broken for Linux RNG to be compromised, the former for `/dev/random` and the latter for the `/dev/urandom` device. At least in case of `/dev/random`, such attack would be extremely hard to execute due to constant mixing of the entropy pool.

3.5.3 Low entropy after initialization

When the operating system starts, it executes a sequence of actions that might be known by an adversary, especially if every action is done automatically. As a consequence, the internal state of entropy pools might be predictable. In the end, this is not an issue, because the kernel allows carrying the state of entropy pools across reboots. As a result, even with knowledge of all startup activities, an adversary would need to know the history of previous sessions, which is only possible on an already compromised system.

3.5.4 Entropy collection from rogue device

Another proposed attack could be done by a device that's able to monitor the hardware - a USB device, for example, or even a CPU. In the latter case, the

CPU knows the state of entropy pools and can therefore contribute malicious "random" data that alters pool state in a certain way. As a result, the actual randomness of the entropy pool decreases. This example shows us that we must (be able to) trust the hardware vendors.

4 Random number generation in virtual environments

Virtualization technologies allow better utilization of all the available computing power by running multiple virtual instances on the same hardware. In the context of (pseudo) random number generation, this presents new challenges. Virtualized environments often lack many of the key entropy sources, such as hard drives, keyboards and mouses. Consequentially, entropy is gathered more slowly, which might impact some applications [2]. Another aspect is the ability of virtualization host (or provider) to measure and alter the entropy sources, as discussed in [5]. Lastly, [1] presented a new kind of vulnerability that might happen when virtual instances are stopped and then started again. Called the reset vulnerability, it causes multiple virtual machines to have equal RNG state, which consequentially generate same pseudo-random data.

4.1 Virtualization host influence on RNG

Article [5] analyzes the safety of running virtualized applications on systems, provided by cloud computing vendors. The authors were able to approximate a few bits of new entropy mix-ins by observing all sources of entropy. There was still some randomness involved and after a few iterations of entropy gathering, the entropy pool was sufficiently scrambled that they couldn't predict PRNG output. In the end, they demonstrated that a small reduction of entropy was possible, but ultimately not enough to break anything.

4.2 RNG performance

As the authors of [2] found out, the rate of entropy generation is on average at least 1.5 times greater on the host, compared to the virtualized guest system. It could be problematic if virtualized applications had to use `/dev/random` extensively. This isn't the case, because `/dev/urandom`, which needs much less entropy, is the preferred source of pseudo-random data. The impact from host on guest RNG was minimal, but they found out that the virtualized system could influence some entropy sources of the host system. Again, even though some entropy could be controlled by the guest machine, it would be unable to predict or influence the final state of the PRNG driver.

4.3 Reset vulnerability

This is perhaps the most threatening vulnerability, because it happens during routine management operations of multiple virtual machines. After the execution of a virtual system is paused and a snapshot[3] is made, the operator then resumes execution. For some time, state of the PRNG will be the same as it was before the pause. This becomes problematic when for scaling reasons, multiple instances of the same snapshot are run. They will have same PRNG state and for some time, they will even output same "random" data. In [1], the authors managed to demonstrate 3 situations that led to reset vulnerabilities with `/dev/urandom`. First two problems were related to the *non-blocking* entropy pool and were thus solved with its removal in Linux 4.8. The last one seems obvious - when *input* pool doesn't have enough entropy, the algorithm behind `/dev/urandom` will reseed by using its internal state. Multiple virtual instances could then have same PRNG state indefinitely, or as long as there was not enough entropy in the *input* pool. The authors then demonstrated reset vulnerability by creating RSA keys using OpenSSL. The `/dev/urandom` output was the same and 2 out of 27 resumed snapshots created identical private keys. Right now, this seems the most major deficiency in the Linux PRNG.

5 Conclusions

In this article, we defined pseudo-random number generators, reasons for their usage and some criteria for their safety. Afterwards, we described and briefly analyzed the details of pseudo-random number generator in the Linux kernel. We also looked at various aspects of PRNG in virtual environments, where the biggest threat is the reset vulnerability. Aside from that, there are no other obvious deficiencies that could be solved by modifying the generator - the rogue CPU threat can not be solved using software. To conclude, the Linux PRNG seems safe to use.

References

[1] A. Everspaugh, Y. Zhai, R. Jellinek, T. Ristenpart, and M. Swift. Not-so-random numbers in virtualized linux and the whirlwind rng. In *2014 IEEE Symposium on Security and Privacy*, pages 559–574, May 2014.

[2] Rashmi Kumari, Mohsen Alimomeni, and Reihaneh Safavi-Naini. Performance analysis of linux rng in virtualized environments. In *Proceedings of the 2015 ACM Workshop on Cloud Computing Security Workshop*, CCSW '15, pages 29–39, New York, NY, USA, 2015. ACM.

[3] Ž. Mahkovec. Analiza generatorjev psevdo-naključnih števil v operacijskem sistemu linux. 2004.

[3] A snapshot is a copy of the entire state of a virtual machine.

[4] D. Munda. Analiza generatorjev psevdo-naključnih števil v operacijskem sistemu linux. 2008.

[5] Christopher J. Thompson, Ian J. De Silva, Marie D. Manner, Michael T. Foley, and Paul E. Baxter. Randomness exposed – an attack on hosted virtual machines.