Android Low Entropy Demystified

Yuanyuan Zhou
University of Michigan, Ann Arbor

Zhuo Peng
University of Michigan, Ann Arbor

Abstract

We look into the issue that the amount of entropy kept by the pseudorandom number generator (PRNG) of Android is constantly low. We find that the accusation against this issue of causing poor performance and low frame rate experienced by users is ungrounded. We also investigate possible security vulnerabilities resulting from this issue. We find that this issue does not affect the quality of random numbers that are generated by the PRNG and used in Android applications because recent Android devices do not lack entropy sources. However, we identify a vulnerability in which the stack canary for all future Android applications is generated earlier than the PRNG is properly setup. This vulnerbility makes stack overflow simpler and threats Android applications linked with native code (through NDK) as well as Dalvik VM instances. An attacker could nullify the stack protecting mechanism, given the knowledge of the time of boot or a malicious app running on the victim device. This vulnerability also affects the address space layout randomization (ALSR) mechanism on Android, and can turn it from a weak protection to void. We discuss in this paper several possiable attacks against this vulnerbility as well as ways of defending. As this vulnerbility is rooted in an essential Android design choice since the very first beginning, it is difficult to fix.

1 Introduction

1.1 Motivation

Our motivation roots in an issue reported to the Android Open Source Program [1] , which complained about the constantly low amount of entropy kept by the PRNG affected the overall performance and user experience, causing low frame rate in the UI layer. There was a solution proposed by the XDA community which periodically wrote into /dev/urandom. The idea behind the solution is that the lag is attributed to the blocking read of /dev/random; writing into /dev/urandom raises the amount of entropy and processes blocking on /dev/random then get unblocked.

We first want to investigate whether this accusation is grounded and whether the solution does any help, since the solution, as described by some users, resolved the long-lived lagging UI issue of Android.

Furthermore, this issue reminds us the identical or factorable RSA private keys issue prevalent among embedded devices which lack sources of entropy and have constantly low amount of entropy [6]. We want to know if there is a similar vulnerability on the Android devices, which could also lack entropy sources during boot, especially during their first boot: the applications which require high quality random numbers, such as a disk encryption application, get predictable or identical results from the PRNG.

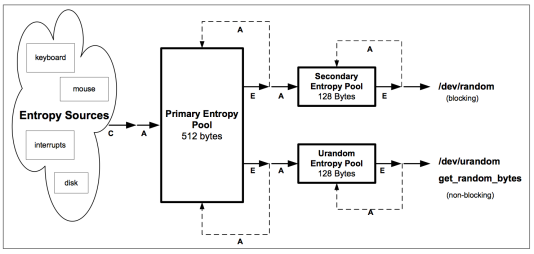

Figure 1: **General Structure of Linux PRNG**

1.2 Linux PRNG and Our Assumption

Android runs on a Linux kernel. The /dev/random and /dev/urandom devices are backed by the Linux PRNG. Figure 1 shows its general structure [5]. In this section, we briefly describe how it works and its properties related to our work.

The Linux PRNG consists of three entropy pools which act in a same manner. They have two major functions: mix entropy into the pool and extract entropy from the pool. The input pool (also known as the primary pool) gets entropy from three types of entropy sources: disk, interrupt and input. When an event happens in a source, the source feed the timing and the type of the event to the input pool. The two output pools (secondary pools) gets their entropy by extracting entropy from the input pool. The blocking pool blocks upon no entropy in its pool and the input pool while the non-blocking pool does not. An important property of the non-blocking pool is that it does not extract any entropy if the amount of entropy in the input pool is less than 192 (`random_read_wakeup_thresh * 3 / 8`) bits. When there is no entropy mixed into the non-blocking pool, its output is deterministic and predictable given any of its internal states between the last entropy mix and the time of present.

The three pools are initialized when the kernel initializes the random character device. On initialization, the pools are not wiped and the time and machine information are mixed in. Under traditional Linux environment, there is an init script which read the saved random seed from the disk and mix it into the nonblocking pool.

Our assumption on Linux ALSR is the following: after the first time when /dev/urandom extract enough entropy (for example, the minimum amount of extraction every time, 60 bits) from the input pool, /dev/urandom is secure. It makes no difference that the amount of entropy is low or high after this time point.

The rest of the paper is organized as follows. In section 2, we introduce existing work on analyzing Linux PRNG and the low entropy issue, as well as the work aiming at exploiting or defending buffer overflow. Section 3 describes our investigation methodology. Section 4 shows the result of our investigation, revailing several vulnerabilities. In Section 5 we discuss the effect of these vulnerabilities and possible attacks and defenses. Finally, in section 6 we describe the future work.

2 Related Work

Previous work was done to analysis the security of the Linux PRNG, especially the non-blocking pool, [5] from a cryptographic point of view. It showed that given the internal state not known to the attacker, predicting the output of the non-blocking pool is as hard as reversing the SHA-1 hash function, even if there is not any entropy mixed in. We base our work on this assumption and do not want to challenge it.

There was also work focusing on what could happen before /dev/urandom being properly seeded and its internal state being completely unpredictable and unknown to the attacker. [6] successfully caught such a vulnerability that on some embedded devices the ssh-keygen was invoked before /dev/urandom was ready, resulting in identical or factorable RSA private keys. Our goal is to identify a similar vulnerability, but on Android devices which are more complicated and not investigated. And our result and conclusion turns out to be different from that for embedded devices: we find that the user applications do not suffer from the low entropy issue while it is the anti-buffer overflow mechanisms, implemented in the kernel and other parts of Android framework that suffer from it.

Device	Kernel Version	Android Version
Nexus 4	3.0.31	4.2.2
Nexus 7	3.1.10	4.2.2
Galaxy Nexus	3.4.0	4.2.2

Table 1: **Experiment Devices and Their Software Versions**

Stack canary [4] is an protecting mechanism that can detect buffer overflow on the stack and terminate the program before attacker taking control of it. It works by putting several bytes in the stack frame, right before the return address and checking if the contents are changed before the function returns. Two major types of canaries are proposed: 1) terminator canary it contains terminator characters on which the string operations stops and 2) random canary. Android platform has random canary implemented since version 4.0. As shown in section 4, we identified a vulnerability that the attacker could know the canary value of all Android apps by a malicious app or simply guess the canary with the cost not depending on the length of the canary.

ALSR was proposed and researched in some work. [7] showed that on an 32-bit architecture ALSR could do little help. Our work confirms further this point: the ALSR implemented in Linux for the arm architecture provides only 8 bits of entropy for each mapped memory range and the base address could be leaked by a side channel.

3 Methodology

3.1 Performance

To establish link between the poor performance and the PRNG, we need to find out which processes would read from the random devices.

We investigate this issue by both statically searching references of /dev/urandom and /dev/random in the source code of Android framework and instrumenting the kernel to capture every read of these two devices and the

process who reads it. As showed in the next section, the result is sufficient to show that there is no causal relationship between reading from random devices and the poor performance.

3.2 Security

As mentioned in the introduction to Linux PRNG, the /dev/random device, although blocking, outputs absolutely secure random numbers that even its internal state compromises, the attacker still cannot predict its output, while the /dev/urandom device, once there is enough entropy mixed in, and its internal state is not known to the attacker, is also secure. Hence we focus only on /dev/urandom.

We want to know:

1. Whether /dev/urandom is ever properly seeded: not being properly seeded could result from the amount of entropy never crossing the threshold of 192 bit due to lack of entropy sources combined with the random seed not being properly saved and restored, or in the first boot.

2. If /dev/urandom is eventually properly seeded, then does the entropy accumulates rapidly enough so that when any application that requires high quality random numbers is invoked, /dev/urandom is ready.

To answer the above questions, we did the following:

1. Instrumented the kernel to capture every event that contributes entropy to the input pool as well as the amount of entropy before and after that event. We also captured the events that extract entropy from the input pool and the amount of entropy before and after them. The time of above events, as well as the time when the user can operate the device are recorded. With the above result, we collected the typical amount of contribution of the three event types.

2. With the result of 1), we investigated what processes read from /dev/urandom before its pool getting its first entropy mix.

3. After we found that virtually every process read from /dev/urandom in order to set up its stack canary, we manually read the related Linux Kernel and Android source to to figure out how and when the canaries are set. We also investigate the ASLR implementation on arm architecture by source code reading.

4. After we identified the predictable canary value issue, and found that given the same initial state, there could be different canary values due to different scheduling realization, we did experiment to see how much entropy does the canary value actually have.

Device	Full Boot Time (s)
Nexus 4	??
Nexus 7	24.8
Galaxy Nexus	??

Table 2: **Full Boot Time:** The time it takes from the kernel is loaded to the Android desktop launcher is brought up.

4 Results and Interpretation

4.1 Performance

We searched in the source code of Android framework since version 4.0 for string "/dev/random" and found no reference to it. We also captured all read operation on /dev/random in a daily use of a Nexus 7 tablet since boot, and found only once, process `wpa_supplicant` read from it. It was a WPA authenticator and it used /dev/random to initialize its own PRNG.

All Android applications who want high quality random number are recommended to use a Java class named SecureRandom, which on initialization read from /dev/urandom to seed its generator.

These results indicate that the performance issue cannot be attributed to blocking read of /dev/random since there are nearly none.

There may be some links between the entropy collecting routine and the poor performance. We noticed that since Linux kernel 3.7, the input event entropy source handler in random.c source code has been restructured [3] to improve the performance on multitouch devices by batching the input events. Some developers commenting on Issue 42265 also suggested that the performance issue is due to suboptimal order of event process and entropy accumulation. We cannot tell if there would be any improvement since we did not successfully install a kernel later than 3.7 on our devices, but we do suggest users keep their software up-to-date since the changes are quite reasonable and promising.

4.2 Security

4.2.1 Entropy Sources

Figure 2 suggests that the Android devices have sufficient source of entropy. The entropy contributed merely

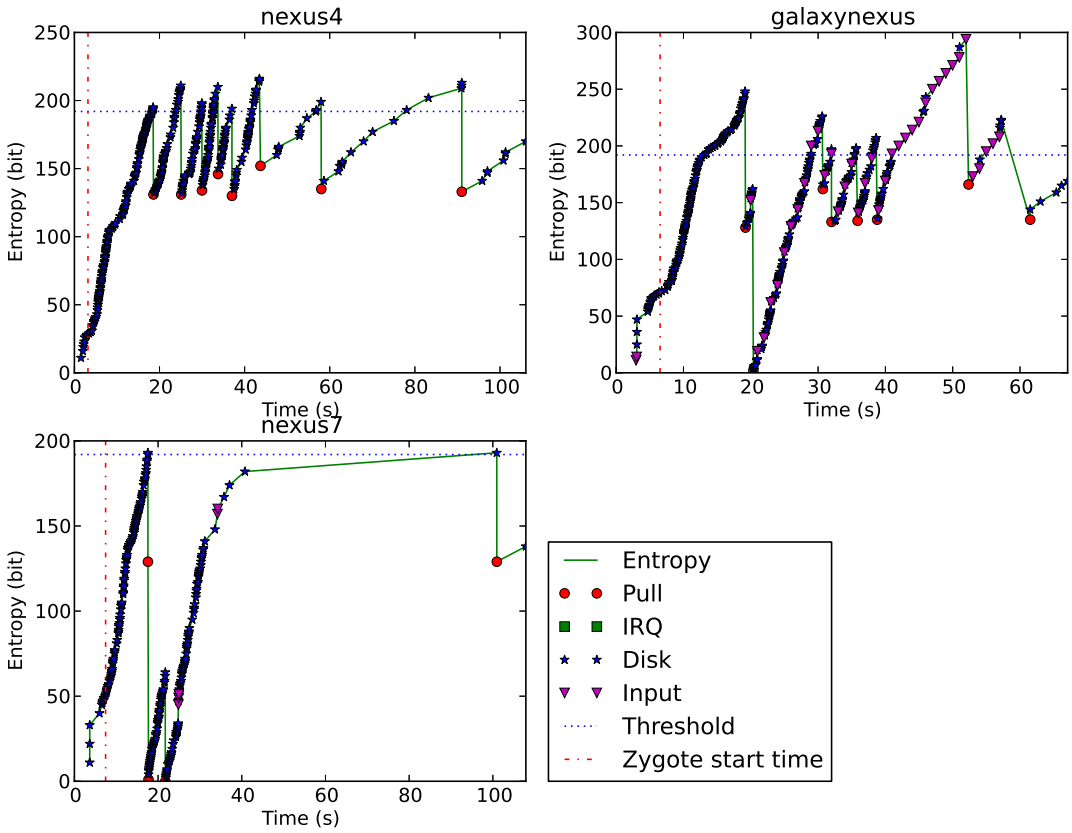

Figure 2: **Amount of Entropy Over Time:** The three devices shows different profiles on entropy accumulation, but their entropy crosses the 192 threshold quickly enough before the user interface is load up. Also note that when Zygote starts the entropy does not cross 192, thus no entropy is mixed into /dev/urandom

by disk events is enough to have /dev/urandom properly seeded before any user applications can run.

4.2.2 User Applications

Figure 2 and 2 show that the time when the user could start the first application is after the time when the amount of entropy first crossed the 192 threshold, which means the first read from /dev/urandom will cause the non-blocking pool extract at least 10 bytes (`EXTRACT_SIZE` defined in random.c). The 80 bits entropy is enough to prevent any attacker trying to predict the random number. There will be more entropy mixed in when user touches or slides on the screen as shown in table 3. If it is the first boot, users will be asked to setup the wireless network and enter Google account information, which cause a lot of input events and further secure the /dev/urandom device.

Therefore we claim that the user applications are able to enjoy high quality random numbers from the beginning.

4.2.3 Saving and Restoring the Random Seed

Android framework has a system service named EntropyMixer which replaces the traditional init script to restore the random seed at boot. There are two major differences between them that may cause security issue:

- EntropyMixer is an Android service which starts after the Android framework is initialized. Any processes starting before are not protected by the saved entropy. /dev/urandom is merely seeded by the time of boot and machine information before that.

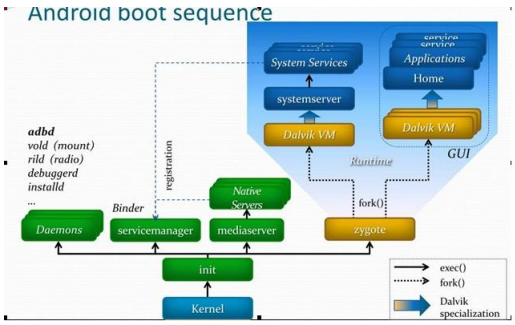

Figure 3: **Android boot sequence**

- EntropyMixer reads /dev/urandom to save the random seed on boot and then saves every 3 hours instead of before rebooting or halting. This may be a better choice since cell phones are not likely to reboot or halt normally, but it also means the amount of entropy of the saved seed is less than or equal to the amount of entropy of the time of first boot, if the device is rebooted within three hours of its first boot.

4.2.4 Shared Stack Canary

We found that virtually all processes need to read from /dev/urandom at their start. We looked into why and found that recent Android versions since 4.0 adopted the stack protector mechanism, as known as stack canaries to protect the Android framework processes as well as native code bridged by Java Native Interface in user applications.

The stack protector works with the help of two components:

- Android NDK compiler (gcc): insert putting and checking code in function prologues and epilogues. The canary value is read from a global variable.

- The standard C library (bionic): initialize the canary value by reading 4 bytes from /dev/urandom and put it in a global variable. The initialization routine is a library constructor which is invoked by the dynamic linker after mapping the shared library into the process? memory space, typically when the process is exec()ed.

However, the way in which the canary value is initialized could be problematic on Android platform. All system services and user applications running on the Dalvik VM are forked from a process called Zygote. By leveraging the Copy-on-Write fork on Linux, Android apps do not need to load the Dalvik VM execution image or the shared library repeatedly on start, leading to better performance. But it also means that there is no chance for an app to invoke the library constructor which setup the canary value. Therefore all Android processes share one stack canary.

We made an app [2] to show its canary value on the screen and compared this value to what `app_process` (which then forked Zygote) read from /dev/urandom on its start. The result verified this vulnerability.

4.2.5 Predictable Stack Canary

Combining section 2.2 and 2.3, we found that it is possible to predict the canary value shared by all Android apps and Zygote. Note that EntropyMixer is also a system service running on Dalvik VM and thus is going to be forked from Zygote. Therefore, when Zygote sets up its canary value, the /dev/urandom is only initialized by its internal routine (`std_initialise`) with the system information and boot time.

However in our experiment, we found the canary value was not a constant when the initial state of /dev/urandom pool was fixed.

We conjectured that it was due to multitasking and different scheduling realization, because /dev/urandom adopts a fine grained locking scheme that maximizes parallelism. It is possible that the internal state changes during the current reading process is descheduled. To verify our conjecture, we modified the /dev/urandom interface to enforce a coarse grained locking scheme so that other process cannot change the internal state before the current process finishing reading. It then showed a fixed canary value that depend only on the initial state of /dev/urandom.

Given no entropy mixed into the pool of /dev/urandom, the internal state depends only on the number of extractions from the pool (this is different from the number of bytes read from /dev/urandom, because the minimum number of bytes to extract is 10). With the non-determinism of scheduling, we can model the canary value as a function $f(B,N)$ where B is the boot time and N is the number of extractions from /dev/urandom that happened before Zygote read from it.

B is acquired by `getnstimeofday()` function which supports resolution of nanosecond, but we observed only microsecond precision on our devices. B may not confirm to a specific distribution and the attacker may have knowledge about it in various degrees, we can only make some simple assumption, for example, we can assume that the attacker knows the boot time with precision of second, then the attacker has to guess the microsecond out of 10^6 values, equivalent to 20 bits of entropy.

N may confirm to some distribution because it can be seen as a value affected by a set of independent events.

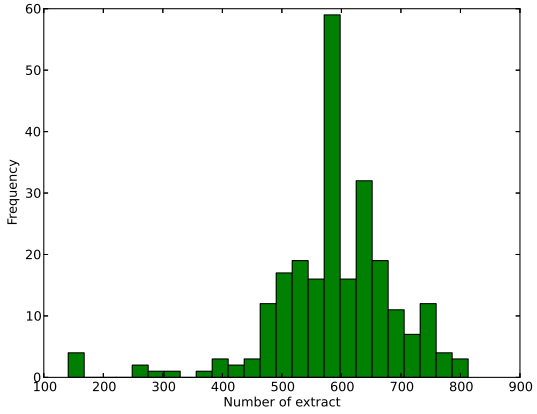

Figure 4: **Histogram of number of extractions from /dev/urandom before Zygote runs:** we repeatedly reboot a Nexus 7 for 244 times to record the number of extractions. The result suggests a bell curve distribution of the data which could help the attacker guess the canary value more effciently

In order to characterize its distribution, we collected its realization from 244 times of boot; the result is shown in figure 4.

If the attacker has prior knowledge of this distribution of the number of extractions before, he could guess the ranges in a higher probability first order, which gives an expected number of guess of 133 in our case, equivalent to about 7 bits of entropy.

With the above model, we can estimate that B and N together will provide 27 bits of entropy, assuming the attacker knows the boot time in second.

Although the computation needed to predict the value of canary may be comparable to simply guessing the 32-bit canary, this vulnerability nullifies the additional security brought by extending the length of the canary. Furthermore, the system time is not a secret number and can be leaked via various ways.

4.2.6 Nullified ASLR

Since 4.0, Android has turned on address space layout randomization with the support of the kernel. Despite of only 8 bits of entropy is added for each mapped region, due to the fork nature of Zygote, all Android apps share one address layout, including the same base addresses of the stack, the heap and the standard C library.

5 Effect and Possible Attacks and Defense

The sharing values and predictable canary vulnerability affects the following:

1. All Android applications using NDK: the simplest stack overflow attacks are made possible.
2. Dalvik VM: any buffer overflow vulnerability in DVM can be exploited with only weak defense mechanism.
3. Zygote: similar to 2), but Zygote runs as root and thus is more profitable to exploit.

5.1 Possible Attacks

Possible Attack 1 (side channel / canary collector) As all applications are sharing the stack canary and the base addresses and these values do not change until the next boot, a malicious application could read these values and use them to build payload to overflow other apps running on the device. We made an example of such an attack [2]. The malicious application could also collect these values and send to a server. If any buffer overflow vulnerability is found on a popular application, the attacker could compromise many devices as there is no stack protecting mechanism.

Possible Attack 2: guessing canary Depending on the resolution of the hardware timer and the knowledge the attacker has, guessing the input of function canary(boot_time, extract_times) may take less time than simply guessing the canary value.

Possible Attack 3: heap overflow Heap overflow vulnerabilities are more dangerous because it can circumvent overwriting the canary. A possible heap overflow attack only need to guess 16 bits, including the base addresses of the stack and the heap.

5.2 Defense

It is relatively easy to address the predictable canary issue by having Zygote write a new canary value after forking a new instance of Dalvik VM. This may bring an overhead of one page of memory due to modification on a CoW page.

The weak ASLR protection is rather hard to fix because:

- The shared base addresses is due to fork implementation. The Android developers choose to only `fork` instead of `exec` after `fork` to avoid repeatedly mapping same files in to the new process' address space,

since all Android apps runs on Dalvik VM and thus need a same set of shared libraries. There seems to be no better way to achieve different address layouts for each forked applications without sacrificing the performance.

- The low entropy provided by ASLR is due to the 32-bit architecture. It is hard to provide enough entropy that is preventive for attackers to guess with brute force without great modification to the Linux Kernel It is hard to provide enough entropy that is preventive for attackers to guess with brute force without great modification to the Linux Kernel.

6 Future Work

References

[1] `http://code.google.com/p/android/issues/detail?id=42265`.

[2] `https://github.com/brills/jni_overflow1`.

[3] Linux kernel 3.7 commit 4369c64c: Input: Send events one packet at a time. `https://github.com/torvalds/linux/commit/4369c64c79a22b98d3b7eff9d089196cd878a10a`.

[4] COWAN, C., PU, C., MAIER, D., HINTON, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *In Proceedings of the 7th USENIX Security Symposium* (1998), pp. 63–78.

[5] GUTTERMAN, Z., PINKAS, B., AND REINMAN, T. Analysis of the linux random number generator. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2006), SP '06, IEEE Computer Society, pp. 371–385.

[6] HENINGER, N., DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Security Symposium* (Aug. 2012).

[7] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security* (New York, NY, USA, 2004), CCS '04, ACM, pp. 298–307.