

/dev/!random: Profiling Entropy Collection in the Linux Random Number Generator

Dept. of CIS – Senior Design 2015-2016

Justin MacIntosh
jmacin@seas.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

Richard Roberts
ricro@seas.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

Abstract

The primary source for random number generation on Linux devices is the Linux Random Number Generator (LRNG). The LRNG collects unpredictable events, such as keypresses or mouse movements, and mixes these events together in a pool. When random values are requested, they are then pulled from this pool.

Even though every Linux machine relies on the LRNG, research on its effectiveness has been almost exclusively theoretical. This can be attributed to the difficulty of profiling and auditing the LRNG, as interfering during its execution can result in mixing additional values into the pool. In an effort to enable and encourage the research community to conduct practical experiments on the LRNG's effectiveness, we have modified the Linux source code to safely produce relevant logs, and send these logs to a remote machine running an application we have created to parse and store these logs for future analysis. Finally, we have used these tools in several experiments to analyze the entropy addition habits during average computer use, and the randomness extraction habits of several security minded and non-security minded applications.

1. Introduction

Random numbers are necessary everywhere in computing. Applications that use cryptography are perhaps the most obvious consumers of random numbers. Any time an encrypted communication channel is utilized by two parties, random numbers must be generated to ensure that their communication

stays private and unalterable to malicious adversaries. Likewise, random numbers must be generated to encrypt files onto a hard drive, and ensure that the only person who can access those files is their owner, even if their computer falls into the wrong hands.

However, there are many non-cryptographic uses of random numbers in computing as well. Randomized algorithms have been developed to sort or process large amounts of data. These algorithms rely on the generation of random values to ensure that the route of execution is chosen with equal probability from the set of possible routes. Any unencrypted communication between two computers, such as connecting through an internet browser to a website's server, generates a random value to be used as an initial sequence number. In fact, every single time a program executes on a computer, the operating system must generate a random value. This value is used when determining where to put the program on the execution stack; by placing the program at a random offset, certain application attacks are mitigated. If the application were in fact malicious, knowledge of location on the stack could aid in attacks used to compromise the machine.

Software applications thus require random numbers to be generated for multiple purposes. They must obtain these random numbers from somewhere; they are unable to generate random numbers themselves, as they are inherently deterministic processes. As John von Neumann said, "*Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin,*" and software applications are arithmetical methods at their core. Even if one writes their own random number generator in software, they still must obtain some

random value to use as a seed to initialize their generator. The solution: applications ask the operating system to provide random numbers for them. However, operating systems are just software as well and run into the same issues with generating unpredictable random values. The operating system too must look somewhere else.

The operating system could use a seemingly random process. Hardware random number generators exist that take unpredictable processes, such as radioactive decay, and use those processes to create random values. One could purchase one of these machines and connect it to their computer, thus giving the operating system a source to collect random values. If this is outside of a user's price range, they could turn to Lavarand. Lavarand is a random number generator that generates its values by taking a picture of a lava lamp and converting the image of the lamp's internal state into a random number. In addition to issues of performance, both of these solutions require the addition of specialized hardware to function. A solution for random number generation must exist for *general purpose computers*, such as laptops and desktops, that does not rely the addition of additional hardware.

2. Technical Background

The Linux Random Number Generator, the random number generator distributed with the Linux kernel, gathers events from its environment. The events are converted to numerical values and mixed into several pools of bytes, from which random numbers are extracted. The amount of entropy, or unpredictability contributed to the internal pools, is estimated with each event that occurs.

2.1 Events

Entropy is harvested from events that happen in the machine's environment. The following list represents the types of events that the LRNG uses to add entropy to its internal state.

Input: Methods of user input, such as typing on keys or moving a mouse, contributes to the internal pool. While these events are not necessarily

uniform and may be somewhat predictable, it is assumed that a reasonable attacker is unable to sufficiently predict these events. Key presses are converted to numerical codes based on the key, and mouse movements are converted to a vector given the size and direction of the movement. The timing of the event is also included, to add additional entropy.

Disk: Every time the disk is accessed, whether reading or saving files, the timing of the access is mixed into the internal state.

Interrupt: Whenever an interrupt is issued from the kernel, its timing is mixed into a small, temporary pool called the "fast pool." Given the frequency of interrupts, it is a poor performance decision to mix the pool every time an interrupt occurs. Instead, at longer regular intervals, the fast pool is mixed with the other pools.

Device: Device adds a string of character bytes to the pools. However, these strings are usually hard-coded values (such as a computer's MAC address), and if an attacker determines these values once, they are able to predict what they will be the next time the computer turns on. Device will therefore not actually contribute to the entropy estimation of the system, but the bytes will still be mixed with the pools. This is to ensure that the pool is properly initialized on startup, and prevent practical attacks seen in [3].

2.2 Entropy Estimation

Unpredictable events add entropy to the system, which in this case can be a measure of how unpredictable the internal state of the random number generator is. The amount of entropy contributed by an individual event is determined by the timing of the event in comparison to the timing of previous events; if a similar event occurs multiple times in rapid succession it is deemed predictable, and will contribute a lower value to the estimated entropy available. For an event occurring at time t_i (measured in jiffies, the number of cycles that have elapsed since the computer booted) we calculate first

calculate the delta between this event and its previous event:

$$\delta_i = t_i - t_{i-1}$$

The LRNG then calculates the delta between this delta and the previous delta:

$$\delta_i^2 = \delta_i - \delta_{i-1}$$

One final level of deltas is computed:

$$\delta_i^3 = \delta_i^2 - \delta_{i-1}^2$$

The value Δ_i is set to be the delta with the lowest magnitude. Finally, we use Δ_i to estimate how much entropy was added by this event. If $\Delta_i < 2$, we add 0 to the estimated entropy counter. If $\Delta_i > 2^{12}$, we add 11. Finally, for all other values of Δ_i , we add $\lceil \log_2(\Delta_i) \rceil$.

2.3 Random Number Output

Random numbers can be requested from one of three interfaces. When a random string of bytes is requested, the LRNG will pull from its internal state and apply a cryptographic hash to the value. A portion of the result will be mixed back into the internal state, and the rest will be folded together and output (in blocks of 10 bytes, that can later be truncated when filling the caller's buffer). The following describe these interfaces.

/dev/random: A special character device visible to userspace processes. */dev/random* will determine if the request being made is for a number of bytes larger than the current count of estimating entropy, and if so, will block the user process until enough entropy is available to fulfil their request.

/dev/urandom: Another special character device visible to userspace processes. Instead of blocking if not enough entropy is available, */dev/urandom* will extract values from the internal state anyway and return them. The maintainers of the LRNG claim that */dev/urandom* should not be used

when cryptographically secure random values are needed.

get_random_bytes(): A kernel-internal function that operates along the same principles as */dev/urandom*

2.4 Theoretical Vulnerabilities

There is much tension over whether or not the Linux Random Number Generator can offer sufficient security guarantees. Academics have discussed theoretical flaws in the design of the Linux Random Number Generator that may lead to a malicious adversary corrupting the internal state such that its output can become biased. One issue frequently discussed is the method of entropy estimation; the estimator described in section 2.2 has no theoretical backing, and parameters seem to have been chosen arbitrarily. The developers of the LRNG are reluctant to change the overall design as that would take a significant amount of resources, and these theoretical vulnerabilities have yet to be shown as feasible in practice.

In 2012, "Mining your Ps and Qs..." [3] demonstrated a practical attack on the LRNG was demonstrated in special circumstances. It was shown that during startup, the LRNG is "entropy starved," that is, not enough events have been collected to ensure that the internal state is unpredictable. This led to a catastrophic attack where the researchers were able to predict the values of keys that were generated soon after the boot process was complete. The LRNG was updated in the next version of the kernel that was released (v. 3.6) such that the *device* events described above were added. While this is seen to have fixed the issue of low entropy on startup, the overall design of the LRNG has not been altered, and thus it still retains the theoretical vulnerabilities described in [1].

3. System

The Linux Random Number Generator is implemented in the Linux Kernel. We feel that the inherent complexity of dealing with the Linux source code may result in researchers neglecting practical analysis of the Linux Random Number Generator in

favor of working on theoretical vulnerabilities. We have created a testing framework that will allow the community to profile entropy collection and random value generation in the LRNG without having to heavily modify kernel code. To do so, we have modified the Linux kernel to send logs to a machine that parses and collects the results.

3.1 Kernel Modifications

The source code for the LRNG is located almost entirely in the kernel source file `drivers/char/random.c`. We have made alterations to this file such that, after recompiling the kernel and installing the updated version, we are able to log when useful events occur. Such events include what functions are adding entropy, when and how many random bytes are being requested, what the running totals are for entropy addition since the computer begins its boot process, and what the state of the internal pools is at a particular moment in time. To determine which logs get sent, we created a struct that stores booleans representing whether or not certain information should be logged. Before sending a particular log, we check to see if it is enabled by this struct, and if not, discard the log. This allows users to enable or disable particular logs by editing out `log_option` struct, instead of searching through the code to determine where each type of log is actually sent.

We wish to send many types of logs when their respective event occurs. There are some logs, such as running totals, that we would like to control manually. Since keyboard key presses add entropy and every key has a unique combination of values passed to its entropy addition function, we are able to designate certain keys as “triggers” for sending logs. When pressed, these keys will send their designated logs without adding any additional entropy. For our experiments, we used number pad keys, as they were never used in any of our scenarios. If one wishes to designate other keys as trigger keys, they may do so provided they know the appropriate values to represent that key (values can be determined by enabling all logging when keys are pressed, pressing the desired key, and manually inspecting the log). We note that pressing a key down, holding a key, and releasing a key are all separate events; we have

enabled our trigger keys to send logs when the key is considered held.

3.2 Sending Logs

Our ultimate goal is to generate logs that can be stored persistently for future analysis; in order to gain meaningful results, one may need to process a large number of logs at the same time, or one may wish to inspect logs over a long period of time. One might first imagine a scenario where the machine running the modified kernel saved its logs to a special file located on disk, that could then be read later. However, we have already established that the LRNG considers disk access an unpredictable event. Writing logs to disk would result in the generation of more logs, that would then generate more logs themselves. This is undesirable.

Instead, we send logs from the modified machine to a client machine running custom software described in section 4. Using the kernel’s `netpoll` interface, we are able to send logs over ethernet without adding additional entropy to the system. This has the downside that the source and destination ip addresses and ports must be known at compile time; changing addresses requires recompilation and installation of the kernel. Logs can be sent out over the internet, or by using an ethernet crossover cable, we can connect machines directly to one another

4. Client

Given that we cannot store logs to disk on the modified machine and must now send them to a secondary computer for collection and storage, we needed to create a schema that represented the structure of the logs and a client program that would parse and store them.

4.1 Schema

In order for the storage machine to receive and store the logs, a system needed to be developed. With XML, a schema would be developed that would allow a script to parse the logs one section at a time. The XML schema contains a name for each section of the log, which would be used later to store the log

in a database. The XML schema also contains branching paths, that determine different paths of logging that the log has followed. An example of this would be branching to Read, Write, or State. In addition, the XML schema contains sections where data could be present, and a type that allows for easy entry of the logs into a database.

```
DATA = ""C0001RBrj\0""
```

```
import socket
import json

IP = "127.0.0.1"
PORT = 5005
DATA = ""C0001RBrj\0""
END = ""end""

print("IP:", IP)
print("Port:", PORT)
print("Data:", DATA)

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
#sock.sendto(bytes(DATA, 'UTF-8'), (IP, PORT))
sock.sendto(bytes(DATA, 'UTF-8'), (IP, PORT))
```

```
<decipher>
  <children type='Continuation'>
    <n name='Cont' trigger='C' size='4'>
      <children type='Function'>
        <n name='Write' trigger='W' size='0'>
          <n name='Read' trigger='R' size='0'>
            <children type='Data'>
              <n name='Bytes' trigger='B' size='3'>/>
            </children>
          </n>
        </children>
      </n>
    </children>
  </decipher>
```

Above is an example of one such schema. An example of a simple log being sent is shown. The bytes that constitute the log are 'C0001RBrj\0'. Following the schema, we recurse through the list of requirements. The first byte relates to the continuation of the log. This section will determine if the given log is part of a greater whole log, or is meant to be taken alone. There is only one child of the <children> tag, meaning that this log can only

follow one path at this point. In order to follow the one path provided to it, the log must meet the next requirement. The requirement is given by 'trigger', and in this case the trigger is 'C'. The log meets this requirement, since its first byte is 'C'. If this byte was different, the log would be malformed, since there are no other paths to take. Finally, we see that the size of this block is 4, which means that the Continuation of this log is comprised of the next 4 bytes. The Continuation of this log is '0001'.

Next, this schema requests the function of the log. In this case, due to the next byte being 'R', the function is 'Read'. No data is given, and therefore size is 0. The Read path continues on, whereas in this schema (but not any real schema) the Write function would be the last part of the log.

Next, the Data is requested. 'B' is the 'trigger' and the log fulfills this requirement. The size dictates that 3 bytes must be read from this section. However, with only 2 real bytes to give, the log leaves the last byte as the null byte. This is important, as the logger expects exactly the number of bytes requested in 'size'. With all of this processing done, the logger would print as below.

```
SUCCESS
Function: ('Read', None)
Continuation: ('Cont', b'0001')
Data: ('Bytes', b'rj\x00')
```

Furthermore, these sections are easily placed into the SQLite Database, as the columns and values required are certain to be present. The described system can be expanded and manipulated so long as the database is changed alongside it, to support the changes and remain current with the latest columns and values that need to be stored.

4.2 Python Listener/Recursive Handler

The previous example of a log being processed and recorded has given a general overview of the program that handled the logs. However, in order to receive them a small listener was created.

The logger was a python script, which looped constantly, listening on a created socket. Whenever packets would be received, whether the packet's data read only 'end' was checked. In this case the server ceases to loop.

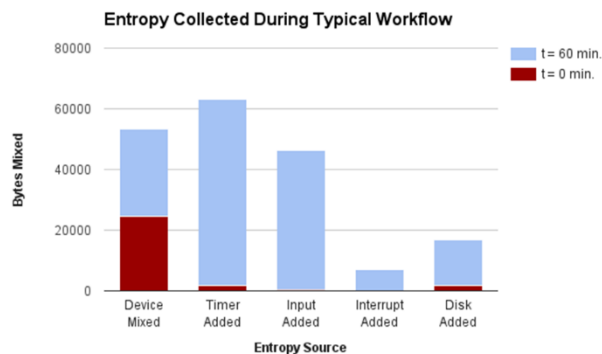
Otherwise, the data is given to the recursive function, which operates as described above. The function returns a dictionary, which should be populated with the results of the recursive calls. In the event that a log could not be parsed, an empty dictionary is returned. This allows the main loop to recognize easily that the log was of an incorrect format. No proper XML schema should permit this function to return an empty dictionary as a legitimate result. With this result in place, mapping the dictionary output to a SQLite schema is the next step, which is relatively simple given that the database schema will be custom modeled to the XML schema. With the data logged in this way, further analysis can take place through queries to the database.

5. Analytical Results

We ran several experiments to ensure that our framework was practical and straightforward to use. Unless noted otherwise, these experiments were executed on version 4.2 of the Linux kernel.

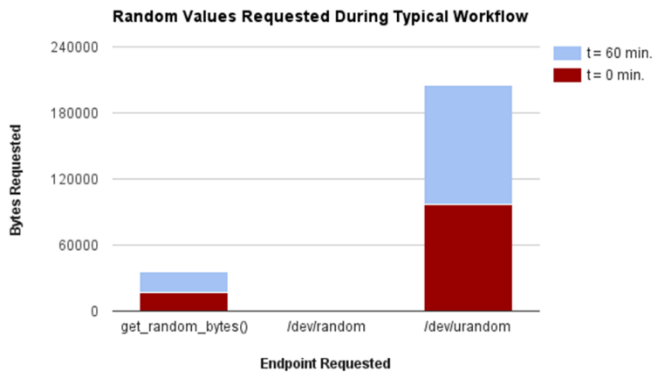
5.1 One-hour Usage Experiment

Our first experiment was to gather data over the course of one hour. During this hour, we engaged in what we deemed to be typical desktop-user activities: web browsing, opening and saving local files, and running local desktop applications. We collected how much entropy was added and how many random bytes were requested from each input and output channel, both immediately after the boot processes (when a user is first prompted to log in) and one hour from that point.



The red bars in the graphs represent events that occurred during the boot process, and the blue bars represent events that occurred during the hour of typical usage. Note that the second column represents entropy added from the timing of events, which includes both input and disk events; timer is the sum of the input and disk columns. Thus, the only events that add entropy during boot are from adding devices (which mixes values into the pool but does not increase the estimated entropy total) and reading/writing to the disk. Most entropy during typical usage comes from user input; in a remote server or workstation scenario where one is not using a keyboard or mouse connected to the machine, the entropy pool is likely to fill at a significantly slower rate.

[2] is the only study we found that provided experimental figures for entropy collection. During their experiments, no entropy was added from interrupts, possibly due to the fact that their experiment operated on a virtual machine. Our data shows that interrupts do contribute to the estimated entropy count by mixing bytes, although the contributions from interrupts are overshadowed by those from input and disk events.



Two interesting conclusions can be drawn from the number of random bytes during this scenario. First, across both `get_random_bytes()` and `/dev/urandom`, approximately the same number of bytes were requested during both startup and the hour of usage. Second, `/dev/random` was never called once throughout the entire experiment. We reflect on this in section 5.2.

5.2 Application Experiments

During our second set of experiments, we took snapshots before and after using particular applications, and gauged their consumption of random value consumption on the difference. One such experiment involved operating the same workflow (opening a browser, navigating to `www.facebook.com`, and logging in to an account) on both the Firefox browser and the Tor browser. We assumed that Tor, being privacy minded, would consume a significantly larger number of bytes, while Firefox would act as a control that represented an average user's browser. During the process, Firefox requested approximately 1500 bytes from `/dev/urandom`, and Tor requested approximately 1600. This difference was significantly smaller than expected. Upon inspection of Tor source code, we learned that Tor only uses `/dev/random` to seed its own pseudorandom number generator, which then becomes the browser's primary method of random number generation.

Out of all the applications we tested, the only one to ever make a call to `/dev/random` that was

not initiated explicitly by the user was key generation through GPG. Even `ssh-keygen` will default to `/dev/urandom` unless told otherwise. We can see that application developers are hesitant to use `/dev/random` due to its tendency to block frequently when entropy is low. GPG key generation took on the order of tens of minutes to generate a 4096-bit key, as it needed to collect 1024 bytes from `/dev/random`. Poor performance may be tolerable when generating a long-term key, but is unacceptable for encryption of live communications such as encrypted web browsing. `Ssh-keygen`'s defaulting to `/dev/urandom` shows that even developers of security-minded applications eschew the supposedly stronger security guarantees.

5.3 Version Experiments

Since all of our changes to the kernel lie in `random.c`, in order to test different versions of the kernel, we only need to port our changes to the appropriate locations in the other versions `random.c` file. This makes porting to older or newer versions of the kernel simple, so long as these versions keep the same high level design of the version we tested on. Since the overall design of the LRNG has not changed significantly since its inception, we were able to test older versions with ease.

Two notable versions were version 3.5 and version 3.6. [3] was published between these two versions, and by running our framework, we can see how the developers responded to the paper. 3.5 did not include entropy from devices; the only entropy available at startup came from disk accesses (which, as mentioned in the paper, was predictable to a reasonable degree). Version 3.6 corrected this by including device entropy during boot, which resulted in a graph similar to the red bars from section 5.1.

5. Future Work

The scope of this project was to create a framework and demonstrate its feasibility. In sections 3 and 4, we describe how our framework operates, and demonstrate it in section 5. In the future, we plan to use the framework to demonstrate a practical

attack on the LRNG, such as the theoretical attacks described in [1]. To mount such an attack, we plan on providing a large number of predictable events (keypresses) in an attempt to bring the internal state of the LRNG to a predictable point that may bias its outputs.

We also plan on open sourcing our framework to the research and development community. Our goal is to enable researchers to conduct analysis on the LRNG in practice, analysis that has largely been absent from the community until now. In addition to releasing documentation and our client code, we plan on releasing modified copies of the random.c files from popular versions of the Linux kernel, as many Linux machines are running older versions of the kernel that may be more susceptible to practical attacks. We hope that if such practical attacks on the LRNG are feasible, then the research community will be the first to know, and that the maintainers of the LRNG will move forward with a new, more secure design before such attacks are seen in the wild.

6. Acknowledgements

We would like to thank the lead research advisor for this project, Dr. Nadia Heninger, for her continuous advice and support. We thank the members of the University of Pennsylvania's SECLAB for their assistance, both conceptual and in providing a workspace and hardware to experiment with. We thank Senior Design advisors Dr. Ani Nenkova and Dr. Jonathan Smith for providing direction in how to effectively present and communicate our research to audiences of multiple levels of technical understanding. Finally, we would like to thank the University of Pennsylvania's School of Engineering and Applied Science and the Department of Computer and Information Science for providing a curriculum that enables us to perform novel research while pursuing our undergraduate education.

7. Citations

- [1] Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergniaud, D. Wichs.. Security analysis of pseudo-random number generators with input: /dev/random is not robust. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security, CCS 2013*, November 2013.
- [2] F. Goichon, C. Lauradoux, G. Salagnac, T. Vuillemin. Entropy transfers in the Linux Random Number Generator. In *HAL Archives*, October 2012.
- [3] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Security Symposium*, August 2012.
- [4] Z. Gutterman, B. Pinkas, T. Reinman. Analysis of the Linux Random Number Generator. In *Proceedings of the 2006 Symposium on Security and Privacy*, May 2006.
- [5] P. Lacharme, A. Röck, V. Strubel, M. Videau. The Linux Pseudorandom Number Generator Revisited. In *Cryptology ePrint Archive*, May 2012.