

# Linux 随机数生成器的原理及缺陷

曹润聪, 曹立明

(同济大学 电信学院, 上海 200092)

**摘要:**Linux 操作系统是目前最流行的开源项目之一。Linux 的随机数生成器是所有类 Linux 操作系统内核的重要组成部分,生成器的输入来自于操作系统中随机事件的熵值,输出几乎涵盖系统中的每一个安全协议,例如生成 TLS/SSL 的密钥、TCP 的序列号,以及用于对文件系统和电子邮件进行加密。尽管随机数生成器是开源项目的一部分,它的源代码(大约 2500 行)却没有很好的文档支持,并且分散于多个代码片段当中。文中将学习随机数生成器原理与应用。详细阐述了随机数生成器的算法,并指出了算法中所隐藏的安全漏洞。还展示了如何对生成器进行攻击从而让其计算出系统先前的状态与输出。最后指出了生成器在设计上的一些缺陷,并提出了在此缺陷上如何进行攻击以及如何防御攻击的方法。

**关键词:**Linux 操作系统;Linux 随机数生成器(LRNG);熵;安全漏洞

**中图分类号:**TP311

**文献标识码:**A

**文章编号:**1673-629X(2007)10-0109-04

## Theory and Flaw of Linux Random Number Generator

CAO Run-cong, CAO Li-ming

(School of Electronic Information, Tongji Univ., Shanghai 200092, China)

**Abstract:**Linux is the most popular open source project. The Linux random number generator is part of the kernel of all Linux distributions and is based on generating randomness from entropy of operating system events. The output of this generator is used for almost every security protocol, including TLS/SSL key generation, choosing TCP sequence numbers, and file system and email encryption. Although the generator is part of an open source project, its source code (about 2500 lines of code) is poorly documented, and patched with hundreds of code patches. Used dynamic and static reverse engineering to learn the operation of this generator. Presents a description of the underlying algorithms and exposes several security vulnerabilities. In particular, show an attack on the forward security of the generator which enables an adversary who exposes the state of the generator to compute previous states and outputs. In addition present a few cryptographic flaws in the design of the generator, as well as some advice and solutions for those flaws.

**Key words:**Linux OS; Linux random number generator; entropy; security vulnerabilities

### 1 随机数生成器的重要属性

随机数是系统进行加密(例如系统利用随机数生成密钥)的重要工具,因此随机数生成器是所有加密系统所必备的一个环节。一个随机数生成器必须具有足够安全的能力去防止来自外部或内部的攻击。下面列出了最基本的三条安全原则<sup>[1]</sup>:

- 1) 伪随机性,生成器的输出要具有随机性;
- 2) 前向安全性,即使知道生成器在某一特定时刻的状态,也不能由此推断出生成器在此刻以前的输出数;
- 3) 入侵恢复/后向安全性,即使知道生成器在某一

特定时刻的状态,也不能由此推断出生成器在此刻以后的输出数。

### 2 LRNG 的整体构架

首先要说明的是,我们的工作基于的是 Linux 2.6.10 版内核,操作系统是 RedHat Linux。

#### 2.1 一般结构

LRNG 的执行过程由 3 个相互串联(异步)的步骤共同完成。第一步是从操作系统所提供的大量随机事件中收集熵值;第二个步骤则利用 mixing 函数将上一步所收集到的熵放入一个叫做熵池的地方;每当系统或用户希望获得一个随机数的时候,第三步就自动地生成一个随机数,同时将熵池更新。这里,“熵”被当成了对系统中事件的不规则性和随机性进行度量的标准。

图 1 描述了 LRNG 的工作流程。LRNG 的内部状态被分别保存在 primary pool(512 字节), secondary

收稿日期:2007-01-10

作者简介:曹润聪(1983-),男,安徽蚌埠人,硕士研究生,研究方向为模式识别与智能系统;曹立明,教授,博士生导师,研究方向为智能分布式系统。

pool(128 字节)和 urandom pool(128 字节)这三个熵池中。每一个熵池都有一个属于自己的熵值计数器。这个计数器其实是一个整数,在 0 到熵池容量大小之间取值。如果有数据从 Entropy Source 流向 primary pool,内核会先测量流入数据的不确定性(即熵),然后增加计数器的值。其它情况下,三个熵池之间熵的流动仅仅会简单地增加或减少计数器的值。从图 1 中还可以看出,熵池在每次输出时都会用新的熵值对自身的状态进行更新(图中的虚线部分),并且这种更新在输出之前发生。

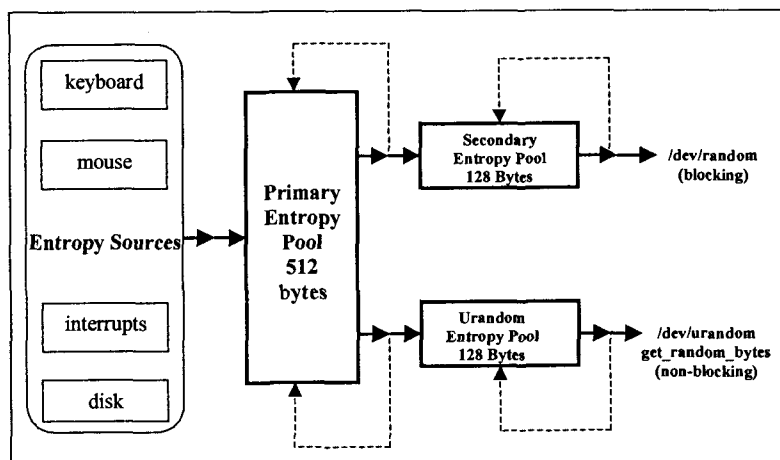


图 1 LRNG 的工作流程

## 2.2 初始化

当内核启动时,系统完成一系列几乎完全可以预测的动作,而且熵池每次初始化后的内容也都大致相同。所以攻击者在启动时可以较准确地推测出熵池的状态。为了解决这个问题,LRNG 在关机时会激活一个脚本文件,这个脚本从 `/dev/urandom` 中读取 512 个字节并将它们写到一个文件中保存起来(就好像保存了一个随机的种子)。当系统再次启动时,脚本被重新激活,从而把这些字节再写回到 `/dev/urandom` 中去。通过这个操作,这次关机和下次开机之间便好似存在了一种延续性。脚本文件的加入确保了 Linux 启动时的不可预测性<sup>[2]</sup>。

## 2.3 收集熵

熵从外部事件源被添加进 primary pool。在桌面计算机或服务器上有 4 种不同的事件源可以使用,它们分别是:鼠标或键盘事件,磁盘的 I/O 操作,以及特殊中断。任何一个事件的发生都会促使系统生成两个 32 位的字符串,并把它们送入熵池。其中一个字符串表示事件发生的时间(时间是自从系统启动以来所经过的毫秒数)。另一个字符串表示所发生事件的类型。

由于整个系统的异步性,从事件产生源处收集来的熵会被先放入 primary pool,若 primary pool 已满(当

primary pool 的熵值计数器的值为 4096 时),就将熵放到 secondary pool。如果这时 secondary pool 也满了,那就再将熵放回 primary pool。如此反复。记住,永远不要首先把熵放到 urandom pool。每一次将熵添加进熵池,都会增加相应熵池计数器的值。

## 2.4 熵的计算

LRNG 在计算事件熵值的时候,只会考虑这个事件的发生时间,不会关注事件的类型。具体定义如下<sup>[3]</sup>:

$$\delta_n = t_n - t_{n-1}; \delta_{2n} = \delta_n - \delta_{n-1}; \delta_{3n} = \delta_{2n} - \delta_{2n-1} \quad (\text{其中, } t_n \text{ 表示第 } n \text{ 个事件发生的时间,且 } t_n, \delta_n, \delta_{2n}, \delta_{3n} \text{ 长度皆为 32 字节}).$$

事件所能提供的熵被定义为  $\log_2(\min(|\delta_n|, |\delta_{2n}|, |\delta_{3n}|)_{[19-30]})$ ,  $S_{[a-b]}$  表示  $S$  的  $a$  到  $b$  位(位置 0 叫做 MSB)。如果  $\min(|\delta_n|, |\delta_{2n}|, |\delta_{3n}|)_{[19-30]} = 0$ , 则计算出的熵值为 0(但即使熵值为 0,事件也一样可以被用于更新 LRNG 的状态。但熵池计数器的值不会增加)。还有,只有当熵从事件源(Entropy Sources)传递到熵池的时候,熵的计算才会发生。

## 2.5 更新熵池和读取随机数

LRNG 的熵池被定义成一个  $m$  ( $m = 32$  或  $m = 128$ ) 字长的数组。内核通过使用 `add(pool, j, g)` 来实现熵增并同时更新熵池的索引  $j$ 。其中, `pool` 表示哪个熵池将会有新的熵加入,  $j$  是熵池中当前位置的索引,  $g$  则表示要加进(熵池)的熵。

下面将以 urandom pool 或 secondary pool 为例,描述读取随机数的算法。简化起见,示例算法中不包括减少熵值计数器的值和更新熵池的步骤。在 Algorithm Extract 算法中<sup>[4]</sup>, `pool` 表示 urandom pool 或 secondary pool(它们的长度都是 32 个字)。  $n$  bytes 表示希望读出的随机数的长度,  $j$  是熵池中当前位置的索引。具体算法步骤如图 2 所示。

算法首先将数组 `pool` 的 0 到 15 位传递给 SHA-1 函数,并将 SHA-1 返回的结果(5 个字长)添加到 `pool` 的  $j$  处;再将 `pool` 的 16 到 31 位传递给 SHA-1' 函数并把结果赋给数组 `tmp`, `tmp[2]` 和 `tmp[4]` 会被分别添加到 `pool` 的  $j-1$  和  $j-2$  处;最后将 `pool` 的  $j-2-15$  位至  $j-2$  位传递给 SHA-1', SHA-1' 的计算结果(5 个字长,即 20 字节)传递给 folding 函数, folding 的返回值会被当成 output 函数(`output(tmp, min(nbytes, 10))`)的参数进行计算, output 的计算结果被复制到用

户或内核缓冲(buffer)。以上的步骤会被重复执行,直到有  $n$  bytes 长度的随机数输出时才会结束。

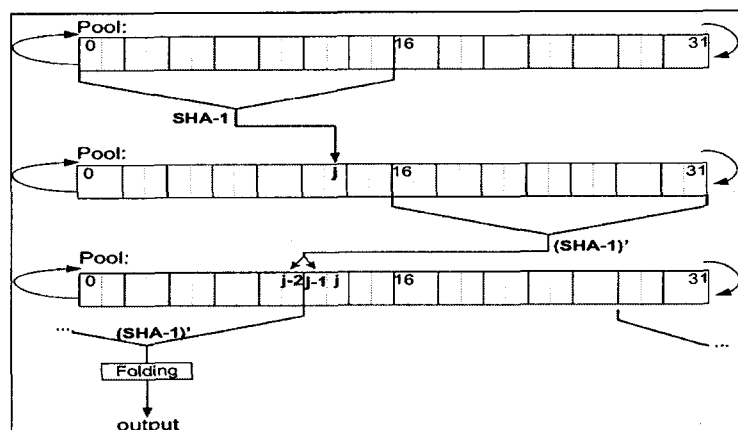


图 2 Algorithm Extract(pool,  $n$  bytes,  $j$ )

folding() 会把 5 个字长(160 位)的输入转化为 2.5 字长(80 位)的输出。算法如下所示<sup>[5]</sup>:

输入:  $W_0, W_1, W_2, W_3, W_4$ ; 输出:  $W_0 \oplus W_3, W_1 \oplus W_4, W_2[0-15] \oplus W_2[16-31]$

(其中,  $W_i$  表示第  $i$  个字,  $W_{i[l-m]}$  表示第  $i$  个字的  $l$  至  $m$  位)

内核通过调用 `get_random_bytes(*buf, nbytes)` 来获取随机数。有两个设备能够生成随机数(\*buf 指向这两个设备中任意一个的设备驱动名), 它们是 `/dev/random` 和 `/dev/urandom`。虽然 `/dev/random` 和 `/dev/urandom` 这两个设备都允许用户从中读取一个伪随机数, 但是它们在所能读取到的随机数的安全性和延迟性上却有不同。`/dev/random` 从 secondary pool 获得输入, 它的输出的是一个“绝对安全”的随机数。因为 secondary pool 在每次输出前, 都会去检查它的熵值计数器的值, 看看能否满足生成随机数的请求, 并且在得到肯定答案之前, 还会不断尝试从 primary pool 获得输入并同时阻塞 `/dev/random` 的请求。相比之下, `/dev/urandom` 从 urandom pool 获得输入, 它的输出永远不会被阻塞, 具有及时性和较弱的安全性。

### 3 安全问题以及解决方案

在对 LRNG 进行安全分析的时候, 总是假设攻击者掌握 LRNG 的源代码, 并且可以通过学习 LRNG 的内部状态去预测输出。所以对任何系统而言, 防止入侵的关键就是: 必须让黑客无从知晓随机数生成器的内部状态。但是当前, 黑客们总是可以通过一些特殊手段来窥测生成器的内部状态, 例如, 黑客可以通过攻击系统、读取机器内存的方法来获取随机数生成器的相关信息。因此 LRNG 的设计中通过调用 `refresh()` 函数实现了一个输出反馈的更新机制, 即熵池每次向外

输出, 总会有新的熵值被重新填入, 从而达到更新内部状态的目的。这样一来, 即使黑客知道了生成器在输出前的状态, 也无法去预知输出后的更新状态。

但现在的一个重要问题就是: LRNG 应该在何时调用 `refresh()` 函数。有两点需要考虑: 一方面, 频繁地调用 `refresh()` 会导致用于更新的数据不再具有足够的熵值, 这样一来, 黑客就可以通过获取生成器先前的内部状态来推测最新的状态(通常, 黑客都会用穷举的方法去搜索状态空间内所有可能的值); 而另一方面, 很少调用 `refresh()` 又意味着在某个较长的时间内, 生成器面临着巨大的安全风险。LRNG 采用的方法是: 每有一个输出就会

会有一个反应用于对熵池的状态进行更新。

但事实上, `refresh()` 的调用只有在黑客能够危及到(获得)生成器的先前状态(注意不是当前状态)时才会起作用。这是因为只要黑客不掌握生成器的先前状态, 那么即使用零熵(或更甚, 用黑客所恶意强加的数据)去更新熵池, 也不会有任何的安全问题。例如, 系统被病毒入侵, 泄漏了先前状态, 当病毒被清除后, 就需要调用 `refresh()` 以更新系统状态。事实上, 在大部分现实系统中, 黑客入侵这样的事件不会频繁地发生。而且, 如果需要人为地防御黑客, 那么就同样需要人为地产生足够的熵来对系统进行更新(最简单的方法就是在一小段时间内不停地敲击键盘)。所以在现实的系统中, 对熵池的更新频率应该维持在一个比较低的水平, 例如, 每 5 分钟更新一次。

正如上面所说, 系统应该以一个较低的频率(比如每 5 分钟更新一次)或用一个较低的熵值(例如反馈熵值的  $1/2$ )来维持对熵池的更新。而且, 在连续两次更新之间应该设定一个最小的时间间隔(例如不要高于一分钟一次)。这个最小的时间间隔应该能够避免黑客在此期间产生具有零熵值的随机事件。此外, 为了能够尽快修复被入侵的系统, 内核应该增添一个可以允许用户提供数据以更新熵池的接口函数, 就像上面所说的那样, 只要生成器的先前状态不被泄漏, 就算黑客恶意使用了这个接口, 也同样无济于事。不过, 为了安全起见, 最好可以对使用这个接口的用户做出一个权限上的限制。正如后面为了防御拒绝服务攻击所作的那样, 权限设置这个思想应当被 LRNG 的设计者所采纳。

此外, 我们已经知道, 当对一个熵池进行读取的时候, LRNG 总是先对熵池的状态进行更新, 然后才计算熵池的输出。这就导致了黑客可以通过学习 LRNG

当前的状态从而得知 LRNG 的先前状态。这样一来, LRNG 的前向安全性便遭到了破坏。所以 LRNG 的设计者必须调整“更新熵池”和“读取熵池”之间的顺序,即先计算熵池的输出然后再更新熵池的状态。

在对熵池更新进行了较为详尽的分析之后,仍然有一点需要说明,那就是在现有的设计下,即使黑客不知道生成器的先前状态,他仍然可以用攻击去影响 LRNG 的输出。因为在 LRNG 接受输入的过程中,当 primary pool 已满,从事件源处收集到的随机事件会被直接加入 secondary pool 中,而 secondary pool 正是/dev/random 读取输入的来源(这样看来,/dev/random 也不是那样地安全)。黑客正可以利用上面的这种情况人为地制造随机事件(称这种人为制造的事件为噪声“noise”)去影响 LRNG 的输出。所以笔者建议:总是要将新进的熵放入 primary pool。如果 primary pool 已满,就阻塞,直到 primary pool 又有了新的空间为止。

上面这个例子分析了/dev/random 设备存在的某种隐患。接下来,笔者还将给出两种通过/dev/random 设备发起攻击的方法。这种攻击叫做拒绝服务攻击。详述如下:

在任一单位时间里,用户都可以不受限制地从/dev/random 和/dev/urandom 这两个设备上读取任意数量的随机数。可是当熵池内熵的估计值小于某个特定阈值的时候,/dev/random 的输出就会被自动阻塞,直到有新的事件被添加进熵池中为止。以上的特性为两种拒绝服务攻击打开了方便之门,这两种攻击都试图阻止用户从/dev/random 中读取随机数。

第一种攻击操作起来很简单,只需要用 dd if=/dev/random 这个命令就可以实现。这种攻击的基本原理就是:不停地从/dev/random 设备上读取随机数。因为这个读取随机数的操作,在数量上没有限制,在权限上也不存在要求,所以就会导致其它用户的合理要求被阻塞(有可能是长时间地被阻塞)。

更进一步,还可以通过调用 get\_random\_bytes(\*buf, nbytes)来实现远程的拒绝服务攻击。因为非阻塞的 urandom pool 是从 primary pool 中获得输入,一旦攻击发起,primary pool 和 secondary pool 必然要陷入瘫痪。实现这种远程攻击的一个最简单的方法就是不断地发送 TCP 连接请求。对于每一个连接请求,都会产

生一个 TCP-syn-cookie,这个 TCP-syn-cookie 会要求从 urandom pool 中得到 128 个字节的随机数,因此造成了熵估计值的下降。

为了解决以上这个问题,有必要为每个用户增加一个权限设置。这个权限代表了用户的级别,不同的级别用户在对/dev/urandom 设备的使用权限,以及单位时间内从/dev/urandom 设备中读取随机数的数量上都有不同。最高级别的用户可以从/dev/urandom 上读取任意数量的随机数;最低级别的用户没有使用/dev/urandom 的权限;中间级别的用户可以使用/dev/urandom,但在读取随机数的数量上又依据级别存在不同的限制。

## 4 结 论

LRNG 是 Linux 内核以及 Linux 操作系统的重要组成部分,它在系统加密等方面发挥着极其重要的作用。分析了 Linux 随机数生成器的工作原理和设计思路,并指出了设计上存在的缺陷以及由此产生的风险,最后针对各种风险给出了安全建议。由于 LRNG 到目前为止都缺乏最新的相关文档以及分析说明,所以以上的工作也正是文中目的所在。

## 参考文献:

- [1] Castejon - Amenado J, McCue R, Simov B. Extracting randomness from external interrupts[C]//In The IASTED International Conference on Communication, Network, and Information Security. USA: [s. n.], 2003:141 - 146.
- [2] Kelsey J, Schneier B, Wagner D, et al. Cryptanalytic attacks on pseudorandom number generators[J]. In Fast Software Encryption, 1998, 1372:168 - 188.
- [3] de Raadt T, Hallqvist N, Grabowski A, et al. Cryptography in openBSD: An overview[C]//In USENIX Annual Technical Conference, FREENIX Track, USENIX. [s. l.]: [s. n.], 1999:93 - 101.
- [4] Kelsey J, Schneier B, Ferguson N. Yarrow - 160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator[J]. In Selected Areas in Cryptography, 1999, 1758:13 - 33.
- [5] Murray M R V. An implementation of the Yarrow PRNG for FreeBSD[C]//In Leffler S J. BSDCon, USENIX. [s. l.]: [s. n.], 2002:47 - 53.

(上接第 108 页)

- [8] Li Min - Quan, Kou Ji - Song, Lin Dan, et al. Primary Theory and Application of Genetic - Algorithms[M]. Beijing: Science Press, 2002.
- [9] Dai Xiao - ming, Chen Chang - ling, Shao Hui - he, et al. Con-

vergence Analysis of Coarse - Gains Parallel Genetic Algorithm and Its Application to Optimization[J]. Journal of Shanghai Jiaotong University, 2003, 37(4):499 - 502.