NAME

arc4random, arc4random_uniform, arc4random_buf, arc4random_stir, arc4random_addrandom — random number generator

LIBRARY

Utility functions from BSD systems (libbsd, −lbsd)

SYNOPSIS

```
#include <stdlib.h>
```

(See *libbsd*(7) for include usage.)

uint32_t
arc4random(*void*);

uint32_t
arc4random_uniform(*uint32_t bound*);

void
arc4random_buf(*void *buf*, *size_t len*);

void
arc4random_stir(*void*);

void
arc4random_addrandom(*unsigned char *buf*, *int len*);

DESCRIPTION

The **arc4random** family of functions provides a cryptographic pseudorandom number generator automatically seeded from the system entropy pool and safe to use from multiple threads. **arc4random** is designed to prevent an adversary from guessing outputs, unlike *rand*(3) and *random*(3), and is faster and more convenient than reading from */dev/urandom* directly.

arc4random() returns an integer in [0, 2^32) chosen independently with uniform distribution.

arc4random_uniform() returns an integer in [0, *bound*) chosen independently with uniform distribution.

arc4random_buf() stores *len* bytes into the memory pointed to by *buf*, each byte chosen independently from [0, 256) with uniform distribution.

arc4random_stir() draws entropy from the operating system and incorporates it into the library's PRNG state to influence future outputs.

arc4random_addrandom() incorporates *len* bytes, which must be nonnegative, from the buffer *buf*, into the library's PRNG state to influence future outputs.

It is not necessary for an application to call **arc4random_stir**() or **arc4random_addrandom**() before calling other **arc4random** functions. The first call to any **arc4random** function will initialize the PRNG state unpredictably from the system entropy pool.

SECURITY MODEL

The **arc4random** functions provide the following security properties against three different classes of attackers, assuming enough entropy is provided by the operating system:

1. An attacker who has seen some outputs of any of the **arc4random** functions cannot predict past or future unseen outputs.
2. An attacker who has seen the library's PRNG state in memory cannot predict past outputs.
3. An attacker who has seen one process's PRNG state cannot predict past or future outputs in other processes, particularly its parent or siblings.

One 'output' means the result of any single request to an **arc4random** function, no matter how short it is.

The second property is sometimes called ‘forward secrecy’, ‘backtracking resistance’, or ‘key erasure after each output’.

IMPLEMENTATION NOTES

The **arc4random** functions are currently implemented using the ChaCha20 pseudorandom function family. For any 32-byte string *s*, ChaCha20_*s* is a function from 16-byte strings to 64-byte strings. It is conjectured that if *s* is chosen with uniform distribution, then the distribution on ChaCha20_*s* is indistinguishable to a computationally bounded adversary from a uniform distribution on all functions from 16-byte strings to 64-byte strings.

The PRNG state is a 32-byte ChaCha20 key *s*. Each request to an**arc4random** function

- computes the 64-byte quantity *x* = ChaCha20_*s*(0),
- splits *x* into two 32-byte quantities *s′* and *k*,
- replaces *s* by *s′*, and
- uses *k* as output.

arc4random() yields the first four bytes of *k* as output directly. **arc4random_buf**() either yields up to 32 bytes of *k* as output directly, or, for longer requests, uses *k* as a ChaCha20 key and yields the concatenation ChaCha20_*k*(0) || ChaCha20_*k*(1) || ... as output. **arc4random_uniform**() repeats **arc4random**() until it obtains an integer in [2^32 % *bound*, 2^32), and reduces that modulo *bound*.

The PRNG state is per-thread, unless memory allocation fails inside the library, in which case some threads may share global PRNG state with a mutex. The global PRNG state is zeroed on fork in the parent via *pthread_atfork*(3), and the per-thread PRNG state is zeroed on fork in the child via *minherit*(2) with `MAP_INHERIT_ZERO`, so that the child cannot reuse or see the parent’s PRNG state. The PRNG state is reseeded automatically from the system entropy pool on the first use of an **arc4random** function after zeroing.

The first use of an **arc4random** function may abort the process in the highly unlikely event that library initialization necessary to implement the security model fails. Additionally, **arc4random_stir**() and **arc4random_addrandom**() may abort the process in the highly unlikely event that the operating system fails to provide entropy.

SEE ALSO

rand(3), *random*(3), *rnd*(4), *cprng*(9)

Daniel J. Bernstein, *ChaCha, a variant of Salsa20*, http://cr.yp.to/papers.html#chacha, 2008-01-28, Document ID: 4027b5256e17b9796842e6d0f68b0b5e.

HISTORY

These functions first appeared in OpenBSD 2.1, FreeBSD 3.0, NetBSD 1.6, and DragonFly 1.0. The functions **arc4random**(), **arc4random_buf**() and **arc4random_uniform**() appeared in glibc 2.36.

BUGS

There is no way to get deterministic, reproducible results out of **arc4random** for testing purposes.

The name ‘arc4random’ was chosen for hysterical raisins -- it was originally implemented using the RC4 stream cipher, which has been known since shortly after it was published in 1994 to have observable biases in the output, and is now known to be broken badly enough to admit practical attacks in the real world. Unfortunately, the library found widespread adoption and the name stuck before anyone recognized that it was silly.

The signature of **arc4random_addrandom**() is silly. There is no reason to require casts or accept negative lengths: it should take a *void* * buffer and a *size_t* length. But it’s too late to change that now.

arc4random_uniform() does not help to choose integers in [0, *n*) uniformly at random when *n* > 2^32.

The security model of **arc4random** is stronger than many applications need, and stronger than other operating systems provide. For example, applications encrypting messages with random, but not secret, initialization vectors need only prevent an adversary from guessing future outputs, since past outputs will have been published already.

On the one hand, **arc4random** could be marginally faster if it were not necessary to prevent an adversary who sees the state from predicting past outputs. On the other hand, there are applications in the wild that use **arc4random** to generate key material, such as OpenSSH, so for the sake of NetBSD users it would be imprudent to weaken the security model. On the third hand, relying on the security model of **arc4random** in NetBSD may lead you to an unpleasant surprise on another operating system whose implementation of **arc4random** has a weaker security model.

One may be tempted to create new APIs to accommodate different security models and performance constraints without unpleasant surprises on different operating systems. This should not be done lightly, though, because there are already too many different choices, and too many opportunities for programmers to reach for one and pick the wrong one.