Hasard Documentation

Release 1.5

Victor Stinner

Apr 18, 2017

Contents

CHAPTER 1

Hasard project

Hasard is a pseudo-random number generator (PRNG) library. It includes multiple engines (algorithms): Park-Miller, Mersenne Twister, Linux device (/dev/urandom or /dev/random), ... It has simple API but with strong code, eg. PRNG seed can be generated using strong entropy (hardware random number generator like /dev/random on Linux).

The library is written in C and a Python binding is available. The library is distributed under BSD license. The word "hasard" is the french name of "randomness" or "chance".

- Hasard website (this site)
- Hasard at Bitbucket: source code, bug tracker

CHAPTER 2

Features

Easy

- *Simple API* with high level functions and uniform distribution
- Generic functions to get/set generator seed and state, and to reseed the generator
- Reuse existing libraries: OpenSSL, gcrypt, GSL, GMP, glib, etc.
- Informations about the generator: name, min/max period, tick range, etc.

Safe

- Choose the best *generator* for your needs and your environment (Linux, Windows, FreeBSD, etc.) using *profiles* (eg. `@fast` or `@secure_nonblocking`)
- Automatically seed the engine using good quality source
- Thread safe: reentrant functions, support lock callbacks
- Reseed the engine at `fork()`

Tests

- Include many *unit tests*
- Reuse existing PRNG test programs like TestU01, dieharder and ENT

CHAPTER 3

Why using Hasard instead of just rand() or /dev/urandom?

It's hard to write algorithms with uniform distribution and most people write buggy functions. Hasard initializes the generator seed from a good entropy source (not the current time or process identifier). Hasard code is also tested by unit tests to ensure that the number distribution is correct.

Hasard always choose the best generator depending of the available generators, eg. use gcrypt_strong if openssl_secure is not available.

Read doc/common_errors.rst document for more information about common errors in pseudo-random number generators.

CHAPTER 4

Hasard User documentation

Install Hasard library

Download

Stable version:

- hasard-1.5.tar.bz2
- MD5: 8d785a177cf10dfd5840a9bfe0f7ba5a
- SHA-1: d91563d3a214d34ec2e9ad4c75dcd58d74c22659

Download development version using Mercurial:

```
hg clone http://bitbucket.org/haypo/hasard/
```

You can also browse the source code.

Dependencies

Compilation

- CMake 2.4+ or 2.6.2+ (only if you use Mercurial tarball) URL: http://www.cmake.org/
- GNU make.
- GCC (C compiler).
- Python 2.5+
- Sphinx (optional): used to compile the reST documentation to HTML http://sphinx.pocoo.org/

Command to run on Debian as root:

```
apt-get install cmake make gcc python
```

Note: TestBigEndian module of CMake 2.6.0 doesn't work.

Suggestions

- Python docutils (rst2html program) used to compile the reST documentation to HTML
- glib 2.0: RNG engine
- OpenSSL 0.9+: RNG engines
- GSL: RNG engine
- gmp: gmp_mt engine

Command to run on Debian as root:

    apt-get install python-docutils libglib2.0-0 openssl libgmp3-dev

Tests

- dieharder program: http://www.phy.duke.edu/~rgb/General/dieharder.php
- ENT program: http://www.fourmilab.ch/random/

Tools

- PIL (Python Imaging Library), need by draw_pil.py program Debian package: python-imaging

Installation

Hasard library

Quick installation (prefix=/usr):

```
./build.sh
(cd build; sudo make install)
```

After the installation, you can run the unit tests using:

```
./run_tests.sh
```

If you want to install Hasard is /opt/hasard instead of /usr, use:

```
mkdir build
cd build
cmake -D CMAKE_BUILD_TYPE=Release -D CMAKE_INSTALL_PREFIX=/opt/hasard ..
```

If you are a hacker, you may want to compile Hasard in debug mode (to include debugger symbols): use "-D CMAKE_BUILD_TYPE=Debug" option.

Python binding

To install the Python binding ("hasard" module), type:

```
cd python
sudo ./setup.py install
```

Windows

Compilation on MinGW with MSYS:

```
cmake -D CMAKE_BUILD_TYPE=Release -G "MSYS Makefiles" ..
```

Compilation on MinGW without MSYS:

```
cmake -D CMAKE_BUILD_TYPE=Release -G "MinGW Makefiles" ..
```

Status

Hasard 0.9 was tested on:

- Ubuntu Gutsy on i386
- Debian Etch on x86_64
- FreeBSD7 on i386
- Windows XP with MinGW on i386
- Debian Etch on ppc32

Hasard API

Public Hasard API is the header file `hasard.h`.

Example

Example written in C language:

```c
/**
 * "Hello World" example of Hasard library.
 * Show usage of common functions.
 *
 * Compile it using "gcc hello_world.c -lhasard -o hello_world".
 */

#include <hasard.h>
#include <stdio.h>

int main()
{
    struct hasard_t *rng;
    rng = hasard_new(HASARD_FAST);
```

```
    printf("Heads or Tails? %s!\n", hasard_bool(rng) ? "Heads" : "Tails");
    printf("Dice: %i\n", hasard_int(rng, 1, 6));
    printf("Integer in 0..999: %lu\n", hasard_ulong(rng, 0, 999));
    printf("Float in [0.0; 1.0(: %.3f\n", hasard_double(rng, 0.0, 1.0));

    hasard_destroy(rng);
    return 0;
}
```

Compile it with `gcc example.c -o example -lhasard`.

You don't have to initialize the random generator seed like the common `srand(time(NULL))`: this task is done by *hasard_new()*.

64-bit integers

`HASARD_INT64` is defined if 64-bit integer are available. Otherwise, some functions like *hasard_get_seed_uint64()* are not available.

Initialization

To create the generator, you have two functions:

struct hasard_t* **hasard_new** (const char* *profile*)

Create a new hasard object: allocate memory and call the constructor. See the *list of profiles* for *profile*. Use *hasard_new_full()* to choose the engines at startup.

Return new allocated object on success, or `NULL` on error.

Example:

```
struct hasard_t* rng = hasard_new(HASARD_FAST);
```

See also *hasard_set_error_callback()*, *hasard_setup_lock()*, *hasard_new_full()* and *hasard_destroy()*.

Changed in version 0.5: the function now has a *profile* parameter.

struct hasard_t* **hasard_new_full** (const char* *rng_engine*, const char* *seed_engine*)

Create a new hasard object: allocate memory and call the constructor:

- *rng*: *engine name* or *profile* of the random number generator
- *seed*: *engine name* or *profile* of the seed generator

Return new allocated object on success, and `NULL` on error.

See also *hasard_set_error_callback()*, *hasard_setup_lock()*, *hasard_new()* and *hasard_destroy()*.

Example:

```
struct hasard_t* rng = hasard_new_full("arcfour", "@secure_blocking");
```

Functions return a new Hasard object on success, or `NULL` on error. Don't forget to call *hasard_destroy()* at exit:

void **hasard_destroy** (struct hasard_t* *rng*)
Destroy a generator: free memory, unload libraries, close files, etc.

Clone a generator

struct hasard_t* **hasard_clone** (struct hasard_t **rng*)
Clone the generator *rng*.

Return `NULL` if the generator can not be cloned

New in version 0.9.

Generate random numbers

bool **hasard_bool** (struct hasard_t* *rng*)
Generate a random boolean: `true` or `false`.

int **hasard_int** (struct hasard_t **rng*, int *min*, int *max*)

int8_t **hasard_int8** (struct *hasard_t**, int8_t *min*, int8_t *max*)

int16_t **hasard_int16** (struct *hasard_t**, int16_t *min*, int16_t *max*)

int32_t **hasard_int32** (struct *hasard_t**, int32_t *min*, int32_t *max*)
Generate a random signed integer in the range [*min*; *max*].

unsigned long **hasard_ulong** (struct hasard_t* *rng*, unsigned long *min*, unsigned long *max*)

uint8_t **hasard_uint8** (struct hasard_t* *rng*, uint8_t *min*, uint8_t *max*)

uint16_t **hasard_uint16** (struct hasard_t* *rng*, uint16_t *min*, uint16_t *max*)

uint32_t **hasard_uint32** (struct hasard_t* *rng*, uint32_t *min*, uint32_t *max*)
Generate a random unsigned integer in the range [*min*; *max*].

double **hasard_double** (struct hasard_t* *rng*, double *min*, double *max*)
Generate a random floating point number in the range [*min*; *max*(.

Changed in version 0.9: the function now generates a number in [*min*; *max*(, instead of [*min*; *max*].

Changed in version 0.4: the function now has *min* and *max* parameters.

Generate an array of random numbers

For best performances, use:

void hasard_ulong_array(struct hasard_t* rng, unsigned long *array, size_t size, unsigned l
Generate *size* random unsigned numbers in the range [*min*; *max*].

void hasard_double_array(struct hasard_t* rng, double *array, size_t count, double min, dou
Generate *size* random floating point numbers in the range [*min*; *max*(.

New in version 0.9.7.

Generate other kind of random values

void **hasard_bytes** (struct hasard_t* *rng*, void* *dest*, size_t *size*)
Generate *size* random bytes: write them into *dest* byte string.

Do nothing if *size* is `0`.

int **hasard_uuid** (struct hasard_t* *rng*, char* *uuid*, size_t *size*)
Generate a random Universal Unique Identifier: UUID version 4. *size* must be at least 37 bytes (36 characters and the final null bytes).

Example of UUID: `"3fd88720-a695-4254-ae0c-55a638368d73"`.

Return `1` on error, `0` on success.

Shuffle an array

void **hasard_shuffle** (struct hasard_t* *rng*, void **base*, size_t *count*, size_t *size*)
Mix items in an array in a random order. Parameters:

- *base*: address of the first item
- *count*: number of elements
- *size*: size of one element in bytes

Implement Fisher–Yates shuffle algorithm.

Do nothing if *count* is less than 2 or if *size* is zero.

Set generator direction

int **hasard_set_forward** (struct hasard_t **rng*)
Set generator direction to forward.

Return `0` on success, or `1` on error.

New in version 1.5.

int **hasard_set_backward** (struct hasard_t **rng*)
Set generator direction to backward.

Return 0 on success, or 1 on error (if the generator cannot run backward for example).

New in version 1.5.

Get and set the state of a generator

size_t **hasard_get_state** (struct hasard_t* *rng*, unsigned char** *state*)
Get a copy of the generator state.

Return `0` if it is not possible to get the state of the generator, for example it is an hardware generator.

Example:

```
size_t size;
unsigned char* state;

size = hasard_get_state(rng, &state);
```

int **hasard_set_state**(struct hasard_t* *rng*, const unsigned char* *state*, size_t *size*)
Restore the state of a generator.

Return `0` on success, `1` on error.

Warning: A state is NOT portable (depends on an integer size, of the endian, etc.) and can only be reused on the same computer. Use hasard_set_seed_xxx() functions (eg. *hasard_set_seed_uint32()*) to generate the same numbers on different computers.

Use *hasard_clone()* to clone a generator.

Get and set the seed

int **hasard_reseed**(struct hasard_t **rng*)
Reseed the engine: regenerate the seed using the hasard seed engine. Return `0` on success, or `1` on error.

New in version 0.4.

To get current generator seed, you can use:

int **hasard_get_seed_uint32**(struct hasard_t* *rng*, uint32_t **seed*)
Get the random number generator seed as an 32 bits unsigned integer.

Return `0` on success, or `1` on error.

New in version 0.5.

int **hasard_get_seed_uint64**(struct hasard_t* *rng*, uint64_t **seed*)
Get the random number generator seed as an 64 bits unsigned integer.

Availability: *need 64-bit integers*.

New in version 0.5.

int **hasard_set_seed_uint32**(struct hasard_t* *rng*, uint32_t *seed*)
Set the random number generator seed from an 32 bits unsigned integer.

Return `0` on success, or `1` on error.

New in version 0.4.

int **hasard_set_seed_uint64**(struct hasard_t* *rng*, uint64_t *seed*)
Set the random number generator seed from an 64 bits unsigned integer.

Return `0` on success, or `1` on error.

Availability: *need 64-bit integers*.

New in version 0.5.

int **hasard_set_seed_bytes**(struct hasard_t* *rng*, const unsigned char* *bytes*, size_t *size*)
Set the random number generator seed from a bytes string.

Return `0` on success, or `1` on error.

New in version 0.5.

For some engines, it's possible to set the seed but not to get it because the seed is only used to initialize the state and it's not possible to retrieve the seed from the state. Get the *state* instead of getting the *seed*.

Get informations about the engines

Informations about the random number generator engine:

const char* **hasard_rng_name** (struct hasard_t* *rng*)
Get the name of the random number generator engine.

See the *list of engines*.

New in version 0.3.

const char* **hasard_seed_name** (struct hasard_t* *rng*)
Get the name of the seed engine.

See the *list of engines*.

New in version 0.3.

double **hasard_min_period_log2** (struct hasard_t* *rng*)
Base-2 logarithm of the minimum period of the generator.

Return a negative value (`-1.0`) if the maximum period is unknown.

double **hasard_max_period_log2** (struct hasard_t* *rng*)
Base-2 logarithm of the maximum period of the generator.

Return a negative value (`-1.0`) if the maximum period is unknown.

Low level functions

See the documentation of *engine tick*.

hasard_tick_t **hasard_tick** (struct hasard_t* *rng*)
Generate a random "tick". A tick is the raw output of a generator: a random unsigned integer in range [0; *tick_max*]. Use `hasard_tick_max()` to get *tick_max* value.

hasard_tick_t **hasard_tick_max** (struct hasard_t* *rng*)
Get the maximum value of a tick: `0` for byte only generators.

unsigned int **hasard_tick_bytes** (struct hasard_t* *rng*)
Get the number of bytes generated by `hasard_tick()`.

Return `0` if the engine has no tick callback or if `hasard_tick_max() + 1` is not a power of `256`.

New in version 0.9.

void **hasard_tick_array** (struct hasard_t* *rng*, hasard_tick_t **array*, size_t *size*)
Generate *count* random "ticks".

New in version 0.9.5.

int **hasard_skip_ticks** (struct hasard_t* *rng*, size_t *count*)
Skip the next *count* ticks.

Return `0` on success, or `1` on error.

New in version 0.7.

Changed in version 0.9: *count* type is now `size_t`, instead of `unsigned long`.

int **hasard_skip_bytes** (struct hasard_t* *rng*, size_t *count*)
Skip the next *count* bytes.

Return `0` on success, or `1` on error.

New in version 0.7.

Changed in version 0.9: *count* type is now `size_t`, instead of `unsigned long`.

Error handling

void **(*hasard_error_prototype)** (const char* *message*)
Error handler callback.

void **hasard_set_error_callback** (*hasard_error_prototype* *display_error*)
Set a custom error handler.

The default handler is *hasard_display_error()*.

If *display_error* is `NULL`, ignore all errors.

void **hasard_display_error** (const char* *message*)
Display an error message.

void **hasard_null_error** (const char* *message*)
"Null" error handler: ignore all errors (do nothing).

Multithreading

If you want to share an Hasard object between multiple threads, you have to protect the internal state using locks.

int **hasard_setup_lock** (hasard_lock_new_prototype *new_lock*, hasard_lock_lock_prototype *lock_lock*, hasard_lock_unlock_prototype *unlock_lock*, hasard_lock_destroy_prototype *destroy_lock*)
Setup lock callbacks:

- new: create a new mutex
- lock: lock the mutex
- unlock: unlock the mutex
- destroy: destroy the mutex

This function must be called before the first call to *hasard_new()* or *hasard_new_full()*.

Library version

const char* **hasard_version_string** (void)
Get the version of the Hasard library as a string. Example: `0.6.0`.

unsigned int **hasard_version** (unsigned int* *major*, unsigned int **minor*, unsigned int* *revision*)
Get the version of the Hasard library as an unsigned integer:

- bits `16..23`: major `((version >> 16) & 0xff)`
- bits `8..15`: minor `((version >> 8) & 0xff)`
- bits `0..7`: revision `(version & 0xff)`

If set, *major*, *minor* and/or *revision* are set to the major, minor and revision version.

int **hasard_compare_version** (unsigned int *major*, unsigned int *minor*, unsigned int *revision*)
Compare the version of the Hasard library with (*major*, *minor*, *revision*):

- return negative number if the library is older

- return zero if both versions are the same
- return positive number if the library is newer

Example:

```
if (hasard_compare_version(0, 6, 0) < 0) {
    printf("Your Hasard version (%s) is older than 0.6, please upgrade!\n",
        hasard_version_string());
    exit(1);
}
```

Other functions

char* **hasard_engine_list** (void)
Get the list of *available engines* as a comma-separated string.

Return a new allocated string on success, or `NULL` on error (unable to allocate memory).

Example: `randu,isaac,park_miller,windows,one`.

char* **hasard_profile_list** (void)
Get the list of *available profiles* as a comma-separated string.

Return a new allocated string on success, or `NULL` on error (unable to allocate memory).

Example: `@fast,@secure_nonblocking`.

Hasard profile list

Hasard library uses profiles to choose the best number generator and seed engines depending on your needs and the available engines. Each operating system provides differents engines, like devices like /dev/urandom for UNIX and BSD, and CryptGen for Windows. Hasard tries also to load dynamic libraries like OpenSSL or gcrypt to get strong, cryptographic, well tested, generators.

This document is a list of available profiles with the usage and the list of engines used by each profile. Some profiles have multiple engines: Hasard tries each engine one by one, until an engine can be used (ie. initialization success).

HASARD_FAST

Usage

HASARD_FAST uses the fastest number generator and a fast non-blocking seed engine. You can use this profile for games or simulations.

Don't use it for security applications!

Engines

- rng: mersenne_twister
- seed: cryptgen, dev_nonblocking, openssl_secure, gcrypt_strong or dev_blocking

HASARD_SECURE_NONBLOCKING

Usage

HASARD_SECURE_NONBLOCKING is a secure non-blocking random number generator. It can be used to generate:

- password
- initialization vector (IV)
- session identifier
- nonce

But don't use it for certificates.

Engines

- rng: openssl_secure, gcrypt_strong, cryptgen, dev_nonblocking
- seed: (same engines than rng)

hasard_new(HASARD_SECURE_NONBLOCKING) uses the engine "zero" for the seed because none of the RNG engines use the seed. It is not necessary to load an engine twice.

HASARD_SECURE_BLOCKING

Usage

HASARD_SECURE_BLOCKING: Secure blocking random number generator. It can also be non blocking depending on the available engines. It can be used to generate certificates (eg. RSA secret key). It may block until it gets enough entropy (eg. keyboard strokes, mouse movements, hardware interruptions, etc.).

Engines

- rng: gcrypt_very_strong, openssl_secure, dev_blocking
- seed: (same engines than rng)

hasard_new(HASARD_SECURE_BLOCKING) uses the engine "zero" for the seed because none of the RNG engines use the seed. It is not necessary to load an engine twice.

openssl_secure is non blocking, other engines are blocking.

HASARD_HARDWARE

Usage

HASARD_HARDWARE: True hardware random number generator, blocking device. Most secure generator, but also the slowest. You might use it to generate passwords or certificates, but it is better to use it to seed another faster cryptographic generator.

Engines

- rng: dev_hardware
- seed: (same engines than rng)

hasard_new(HASARD_HARDWARE) uses the engine "zero" for the seed because none of the RNG engines use the seed. It is not necessary to load an engine twice.

HASARD_TEST

Usage

HASARD_TEST is reserved for Hasard internal tests! It only generates zeros.

Engines

- rng: zero
- seed: zero

Engine list

List of the Pseudo-Random Number Genearators (PRNG), called "engines", of the Hasard library.

Good PRNG

All of these generators can by used for games or physical simulations. There are uniform. Except GSL and KISS, the seed is a least 128 bits.

- `arcfour`
  - Arcfour (RC4)
  - http://en.wikipedia.org/wiki/RC4
  - Support set_seed(int) and set_seed(bytes)
  - Seed: 2048 bits
  - the period is larger than 10^100 (~2^332.2), see: "Stream Ciphers", RSA Laboratories Technical Report TR-701, July 1995
- `mersenne_twister`
  - Mersenne Twister
  - http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
  - http://en.wikipedia.org/wiki/Mersenne_Twister
  - Period: 2^19937 - 1
  - Support set_seed(int)
  - Seed: 128 or 256 bits (size of 4 unsigned long in bits)
  - Tick range: [0; 4294967295]

- `isaac`
  - Indirection, Shift, Accumulate, Add, and Count (ISAAC)
  - http://www.burtleburtle.net/bob/rand/isaacafa.html
  - http://en.wikipedia.org/wiki/ISAAC_(cipher)
  - Minimum period: 2^40
  - Maximum period: 2^8295
  - Seed: 8192 bits
  - Tick range: [0; 4294967295]
- `glib`
  - g_rand_int(), g_rand_double_range(), g_rand_set_seed() functions of the glib library.
  - http://www.gtk.org/
  - Implement Mersenne Twister
  - Seed: 128 bits
  - Tick range: [0; 4294967295]
- `dev_nonblocking`
  - UNIX/BSD device `/dev/urandom`
  - Non-blocking and unsafe
- `openssl_pseudo`
  - `RAND_pseudo_bytes()` function of the OpenSSL library
- `gcrypt_strong`:
  - `gcry_randomize(GCRY_STRONG_RANDOM)` function of the gcrypt library
  - for session keys
  - non blocking
- `gsl`:
  - `gsl_rng_get()`, `gsl_rng_uniform_int()`, `gsl_rng_uniform()` and `gsl_rng_set()` functions of the GSL library.
  - Use the default algorithm (`"mt19937"`)
  - Seed: 32 or 64 bits (size of an unsigned long in bits)
  - Tick range: [0; 4294967295]
- `gmp_mt`:
  - `mpz_urandomm()`, `mpf_urandomb()` and `gmp_randseed()` functions of the GMP library
  - Use Mersenne Twister (default) algorithm
  - Seed: 128 bits
  - Tick range: [0; 4294967295]
- `kiss`:
  - “KISS”, proposed by George Marsaglia

  - KISS combines:
    - LCG: x(n+1) = (69069*x(n) + 1327217885) mod (2^32), period=2^32
    - multiply-with-carry, period=597273182964842497 (>2^59)
    - xor-shift, period=2^32-1
  - http://www.math.uni-bielefeld.de/~sillke/ALGORITHMS/random/marsaglia-inline-c
  - Seed: ~125.4 bits
  - Period: ~2^123
  - Tick range: [0; 4294967295]
- `havege`
  - "Hardware Volatile Entropy Gathering and Expansion" (HAVEGE)
  - http://www.irisa.fr/caps/projects/hipsor/
  - ndrand() function of the HAVEGE library
- `nspr`
  - "NetScape Portable Runtime library"
  - `PR_GetRandomNoise()` function of the nspr library, get random noise from the host platform
  - The intent of `PR_GetRandomNoise()` is to provide a "seed" value for a another random number generator that may be suitable for cryptographic operations. This implies that the random value provided may not be, by itself, cryptographically secure. The value generated by `PR_GetRandomNoise()` is at best, extremely difficult to predict and is as non-deterministic as the underlying platform can provide.
  - Calls to `PR_GetRandomNoise()` may use a lot of CPU on some platforms. Some platforms may block for up to a few seconds while they accumulate some noise. Busy machines generate lots of noise, but care is advised when using `PR_GetRandomNoise()` frequently in your application.

Secure generators

A secure or "cryptographic" generator can be used to generate password or certificate. An attacker is not able to guess previous generated numbers, even if he knows the internal state.

- `dev_blocking`
  - `/dev/srandom` on OpenBSD, or `/dev/random` on other UNIX/BSD
  - Block if there is no more entropy, you have to wait next hardware interruption, keyboard stroke, mouse event, etc.
- `dev_hardware`
  - UNIX/BSD device `/dev/random`
  - Block if there is no more entropy, you have to wait next hardware interruption, keyboard stroke, mouse event, etc.
- `openssl_secure`
  - `RAND_bytes()` function of the OpenSSL library
- `cryptgen`
  - `CryptGenRandom()` function on the Windows API (advapi32.dll)
  - Windows API to generate pseudo-random numbers

  - Use RC4 and SHA-1
  - This generator is not cryptographic secure because it uses RC4, which can be run backward once its state is known
- `gcrypt_very_strong`
  - `gcry_randomize(GCRY_VERY_STRONG_RANDOM)` function of the `gcrypt` library
  - for key material
  - blocking
- `havege_crypto`
  - "Hardware Volatile Entropy Gathering and Expansion"
  - http://www.irisa.fr/caps/projects/hipsor/
  - `crypto_ndrand()` function of the HAVEGE library

Weak PRNG

- `gcrypt_nonce`
  - `gcry_create_nonce()` function of the `gcrypt` library
- `gcrypt_weak`
  - `gcry_randomize(GCRY_WEAK_RANDOM)` function of the `gcrypt` library

Weak PRNG of hasardweak library

Weak engines use a small internal state, a small seed and are highly predictable by an attacker. Most of them are not uniform (especially the lower bits). Don't use them!

LCG

- `libc_rand`
  - `rand()` function of the system standard library C
  - Tick range: [0; RAND_MAX]
  - Seed: 31 bits
  - WARNING: This engine is **not** reentrant
- `libc_rand_r`
  - rand_r() function of the system standard library C
  - Tick range: [0; RAND_MAX]
  - Seed: 31 bits
- `libc_rand48`
  - nrand48(), erand48() and seed48() functions of the standard library C
  - x(n+1) = (x(n) * 25214903917 + 11) % 2^48
  - tick(n) = (x(n) >> 17)

  - Tick range: [0; 2147483647]
  - Seed: 48 bits
  - WARNING: This engine is **not** reentrant
- `libc_rand48_r`
  - nrand48_r(), erand48_r() and seed48_r() functions of the standard library C
  - x(n+1) = (x(n) * 25214903917 + 11) % 2^48
  - tick(n) = (x(n) >> 17)
  - Seed: 48 bits
  - Tick range: [0; 2147483647]
- `libc_random`
  - random() function of the system standard library C
  - Tick range: [0; RAND_MAX]
  - Seed: 32 bits
  - WARNING: This engine is **not** reentrant
  - See also libc_random_r_* (the reentrant version)
- `libc_random_r_8`, `libc_random_r_32`, `libc_random_r_64`, `libc_random_r_128`, `libc_random_r_256`
  - random_r() function of the system standard library C with a state of 8, 32, 64, 128 or 256 bytes
  - Tick range: [0; RAND_MAX]
  - Seed: 32 bits
  - See also libc_random (non reentrant version)
- `zx_spectrum`
  - x(n+1) = (x(n) * 75) % 65537
  - tick(n) = x(n) - 1
  - Tick range: [0; 65536]
  - Period: 65536 (2^16)
  - Seed: ~16 bits
- `randu`
  - http://en.wikipedia.org/wiki/RANDU
  - x(n+1) = (x(n) * 65539) % 2147483648
  - tick(n) = x(n) - 1
  - Tick range: [0; 2147483646]
  - Minimum period: 536870912 (2^29)
  - Seed: 31 bits
  - RANDU is one of the worst PRNG!
- `minimum_standard`

  - "Minimum standard", ANSI C, Watcom, Digital Mars, CodeWarrior, ...
  - x(n+1) = (x(n) * 1103515245 + 12345) % 2147483648
  - Period: 2147483648 (2^31)
  - Seed: 31 bits
- `park_miller`
  - Park-Miller "Minimum Standard", first proposed by Lewis, Goodman, and Miller in 1969
  - x(n+1) = (x(n) * 16807) % 2147483647
  - tick(n) = x(n) - 1
  - Tick range: [0; 2147483645]
  - Period: 2147483646 (2^31 - 1)
  - Seed: 31 bits
- `windows`
  - x(n+1) = (x(n) * 214013 + 2531011) % 32768
  - tick(n) = (x(n) >> 16) & 32767
  - Tick range: [0; 32767]
  - Period: 2147483648 (2^31)
  - Seed: 32 bits
- `rand48`
  - x(n+1) = (x(n) * 25214903917 + 11) % 2^48
  - tick(n) = (x(n) >> 17)
  - Tick range: [0; 2147483647]
  - Seed: 48 bits

`RAND_MAX` is `32767` on Windows or `2147483647` on Linux.

Other

- `middle_square`
  - Middle-square algorithm, by John von Neumann (1946), using 5 decimal digits
  - x(n+1) = (x(n) * x(n) / 100) % 10^5
  - Minimum period: 1
  - Maximum period: 41
  - Seed: ~16.6 bits

Test generators

Generators reserved for testing purpose.

- `zero`
  - only generate nul bits

  - Period: 1
  - no seed
- `one`
  - only generate one bits
  - Period: 1
  - no seed
- `counter`
  - loop on the sequence 0, 1, 2, ..., 255
  - Period: 256
  - Seed: 8 bits

`zero`, `one` and `counter` support get/set seed, get/set state, reseed and skip tick.

Common errors in pseudo-random number generators

rand() % range or rand() & range

To generate an integer number in a range, some people use module operator, which is the worst idea to reduce the interval [0; RANDMAX] to [0; range-1]. The problem is that for some generators, lower bits are less random than higher bits.

Example

Example with RANDU algorithm:

```
x(0) = 42
x(1) = (65539 * x0) mod 2^31 = 2752638
x(2) = (65539 * x1) mod 2^31 = 16515450
x(3) = (65539 * x2) mod 2^31 = 74318958
x(4) = (65539 * x2) mod 2^31 = 297274698
...
```

And now, let's try to generate a number in range 0..10 with x(i) % 10:

```
2, 8, 0, 8, 8, ...
```

Examples with probability

Let's imagine that you have a *perfect* random source (hardware entropy) generating numbers in range [0; 255]. And you want to generate random numbers in range [0; 62]:

```
randint(): rand() % 63
```

So you get:

- input range [0; 62] => [0; 62]
- input range [63; 125] => [0; 62]

- input range [126; 188] => [0; 62]
- input range [189; 251] => [0; 62]
- input range [252; 255] => [0; 3]

The problem is in the last range: the output range is smaller than the input range, and so the values in range [0; 3] are more frequent than [4; 62]:

- output range [0; 3]: p(i) = 5 / 256 = 1.95%
- output range [4; 62]: p(i) = 4 / 256 = 1.56%

This can be even worse if the output range is closer to the input range. Eg. input=[0; 15], output=[0; 14]:

- output value 0: p(i) = 2 / 16 = 12.5%
- output range [1; 14]: p(i) = 1 / 16 = 6.3%

The number 0 is two times more frequence than the others.

Solution using float

Use floating point number:

```
int randint(int min, int max)
{
    double range  = (double)max - (double)min + 1.0;
    return min + (int)( (double)rand() * range / (RAND_MAX + 1.0) );
}
```

The result is converted to integer and C language truncated digits. so 0.9 is converted to 0. That's why it uses a wider ranges:

- max-min+1 and not not max-min
- RAND_MAX+1.0 and not RAND_MAX

Warning: This solution doesn't work with 64 bits integer because double has smaller precision (52 bits) and so you will loose the least significant bits.

Solution using integers

Hasard implements a solution using only integers working on 32 and 64 bits CPU. See lib/randint.c.

Non initialized generator

If a generator is not initialized, it's easy to guess its internal state and to guess previous and next generated numbers.

Example

Example with non initialized generator:

```
#include <stdio.h>

int main()
{
    int i;
    for (i=0; i<4; i++) printf("%u\n", rand());
    return 0;
}
```

First program run:

```
1804289383
846930886
1681692777
1714636915
```

Second program run:

```
1804289383
846930886
1681692777
1714636915
```

Solution

Set the seed (eg. `srand()` function) with good entropy source.

Generator reseed at each tick

Some buggy generators are reseed (especially using `time()` and `getpid()`) at each tick which is bad! If many numbers numbers are generated during the same seconds, they will be all equal!

Example: old versions of PHP and ClamAV have this bug.

Poor seed

There are two distincts problems about the seed:

- some seed values leads the smaller period than the maximum period
- to guarantee the confidentiality, the attacker should not be able to guess the PRNG internal state

Small period

Some generators have smaller period if the seed is not correctly choosen. In case of Park-Miller generator, the seed have to be coprime with the modulus (ensure that gcd(seed, modulus)=1)

Example with RANDU:

```
x(n+1) = (x(n) * 65539) % 2147483648
```

With the worst seed, 1073741824 (modulus / 2), the period is only one:

```
x(0) = 1073741824
x(1) = 1073741824
x(2) = 1073741824
...
```

Another bad seed, 536870912 (modulus / 4), the period is two:

```
x(0) = 536870912
x(1) = 1610612736
x(2) = 536870912
x(3) = 1610612736
...
```

Confidentiality and PRNG state

In simple PRNG, it's easy to recover the PRNG state: it's very close to the random tick. But even if the attacker has no access to the random numbers, he can guess next numbers if he knows the initial seed.

Example: if the seed is only based `time() + getpid()`, the attacker can guess this value using bruteforce. Most computers share the same time() value (or close values) and `getpid()` value is in [2; 32767] in most cases.

So to make attacker job harder, use a True Random Number Generator like an Hardware Random Number Generator.

Integer overflow

In most langagues, integers have physical limits like `[-2^31; 2^31-1]` or `[0; 2^64-1]`. If an algorithm uses the wrong types, the result can be biased in favor of some numbers.

Example with integer overflow:

```
#define RAND_MAX 2147483648
uint32_t min, max, tick, number;
number = (double)tick * (max - min + 1) / (RAND_MAX + 1.0);
```

This code works correctly for most values, but if you try with min=0 and max=UINT32_MAX, the result doesn't depend on tick anymore, the result is always zero! It's because max-min+1 = UINT32_MAX+1 doesn't fit in uint32_t type.

To fix this code, use the right types. Fixed example:

```
number = (double)tick * ((double)max - min + 1) / (RAND_MAX + 1.0);
```

Security

Hasard library

File descriptor: dev_nonblocking tries to set FD_CLOEXEC flag on the file descriptor, whereas dev_blocking and dev_hardware fail if FD_CLOEXEC flag can not be set.

Mersenne Twister

Why is Mersenne Twister non-secure? http://groups.google.com/group/sci.crypt/browse_thread/thread/305c507efbe85be4

```
MT was not designed as a cryptographic generator.  There's a
straightforward way (described in the MT paper) to reconstruct the
internal state.  Remember where it says something like it's
equidistributed in 623 dimensions?  That means that if you start
counting in 624 or more dimensions, all bets are off.  That's
different from a cryptographic generator, which can't be distinguished
from uniform no matter how many dimensions you use.
```

```
There can be no "cryptographic generator" in that sense
which uses a computer program.  This includes BBS, which
to be sure has more dimensions, and is not linear.
```

```
As has been noted, it's a linear shift register.

It is because there is a simple relationship between the successive
32-bit words of pseudorandom data that the Mersenne Twister produces.
The relationship will allow those words to be in a sequence that does
not exactly repeat itself for a very long time, but it only takes a
short time to determine all the information you need to duplicate the
entire sequence.

The successive words of data give the entire internal state; in a
secure keystream generator, the internal state is nearly impossible to
derive from the outputs.
```

```
Is it possible to predict forthcoming values,
given a string of previous values? (...)
Mersenne Twister, which needs < 650
```

```
In the simplest possible terms there is a difference between something
producing a random looking distrubtion of bytes and producing a stream
of bytes that is indistinguishable from a random source.

The MT is an algorithm that passes the first test but not the second.
It is easy to distinguish MT from a random source and this fact alone
is considered a break of the MT. Fufilling the indistinguishability
criteria is neccessary and sufficient for producing a secure
stream-cipher.
```

CHAPTER 5

Random number generators documentation

Random number generators

Hardware entropy generators

Hardware:

- Sound: audio entropy daemon
- Video: video entropy daemon
- Lava Lamp! lavarnd.org
- True random numbers from Wi-Fi background noise

Daemons:

- EGD: Entropy Gathering Daemon
- PRNGD: Pseudo Random Number Generator Daemon
- aed: an "additional entropy" daemon for Linux

Other:

- Solaris /dev/random device (not maintained since 2002)
- Hurd /dev/random device
- rng-tools (part of gkernel project): inject entropy from hardware generator to /dev/random, but test also the entropy quality using FIPS 140-2

Download entropy from websites

- random.org
- HotBits: Genuine random numbers, generated by radioactive decay

- Quantum Random Bit Generator Service
- EntropyPool and Entropy Filter

Seed collision

Size of the internal state

If the size of the RNG internal state is too small, you may generate duplicate seeds. For example, most LCG RNG uses a 32 bits unsigned integer as state.

Using the birthday problem, we can compute the number of seed that have to be tested to get the first collision with a probability of 50%:

```
>>> from math import sqrt, log
>>> def n(p, d): return sqrt(2*d*log(1/(1.0-p)))
...
>>> n(0.5, 2.0**32)
77162.743235574148
```

where 2^32 is the number of different possible seeds.

So with an internal state of 32 bits, you have to try 77163 seeds to create the same RNG with a probability of 50%. If we want to be sure, you can try with a probability of 99%:

```
>>> n(0.99, 2.0**32)
198892.20870276989
```

Even if you have a perfect random seed generator, you have a risk to generate duplicate generators if the internal state is too small. If you want to avoid collisions, use a geneartor using larger internal state, eg. at least 128 bits:

```
>>> n(0.5, 2.0**128)
2.1719381355163562e+19
```

Quality of the seed generator

In the first section, we supposed that the seed generator was perfect. But in real life, the seed is usually created using predictable informations like process identifier or current time in seconds. Because of a bug introduced in OpenSSL package, the seed was only created using the process identifier, which is -most of the time- a number of... 15 bits.

```
>>> n(0.5, 2.0**15)
213.13398045637061
```

Try 213 different keys was enough to break Debian OpenSSL security...

Good entropy sources

- `/dev/random`: best entropy source on Linux, but blocks if there is no more entropy. It uses physical events like: keyboard strokes, mouse clicks and moves, hard drive timing, etc.
- `/dev/urandom`: non blocking generator
- (External) Hardware RNG

Cryptographic secure PRNG (CSPRNG)

Properties:

- Protect against previous / next numbers prediction, even if the attacker knows the last results
- Protect internal state: even if the attacker knows the RNG output, he shouldn't be able to guess the internal state. Common trick is to hash the output (eg. MD5, SHA-1, or using a cryptographic hash function)
- Protect against entropy injection

LCG RNGs don't feet these requierements because knowing one number is enough to guess all previous and next numbers. Mersenne Twister is not a CSPRNG because it's not uniform in dimension 624.

Linear congruential generators

Formula

The generator is defined by the recurrence relation:

x(n+1) = (x(n) * multiply + add) % modulo

Known LCG

Generators sorted by modulo.

Name	multiply	add	modulo	bits
ZX Spectrum	75		2^16 + 1	
Park-Miller (1)	16807		2^31 - 1	
ANSI C (2)	1103515245	12345	2^31	30..16
IBM RANDU	65539		2^31	
Another Park-Miller	279470273		2^32 - 4	
Numerical Recipes	1664525	1013904223	2^32	
VAX	69069	1	2^32	
GCC	69069	5	2^32	30..16
Microsoft (3)	214013	2531011	2^32	30..16
Borland Delphi (4)	134775813	1	2^32	63..32
Borland C/C++ (5)	22695477	1	2^32	30..16
CRAY	44485709377909		2^48	
GNU libc rand48	25214903917	11	2^48	48..17

Full names:

- 1. Park-Miller and Apple CarbonLib
- 2. ANSI C: Watcom, Digital Mars, CodeWarrior, IBM VisualAge C/C++
- 3. Microsoft Visual, Microsoft Quick C/C++, PHP (on Windows)
- 4. Borland Delphi and Virtual Pascal
- 5. Borland C/C++: rand() uses bits 30..16 whereas lrand() uses bits 30..0

Notes:

- `2^16 + 1` (65537) is a Fermat prime F4, and 75 is a primitive root modulo F4
- To get bits `30..16`, use `rand() = (x(n) >> 16) & 0x7fff`

Park-Miller

A Park-Miller generator is a special case of a LCG: add=0. The seed have to be correctly generated to get the maximal period:

- if seed=0, all numbers will all zero
- if gcd(seed, modulo) != 1, the period will be smaller than the maximum

Linux kernel devices

Linux provides two devices:

- `/dev/random`: RNG, block when there is no more true random bytes
- `/dev/urandom`: PRNG, fast

Kernel configuration

Options:

```
CONFIG_HW_RANDOM=y
CONFIG_HW_RANDOM_INTEL=y
CONFIG_HW_RANDOM_AMD=y
CONFIG_HW_RANDOM_GEODE=y
CONFIG_HW_RANDOM_VIA=y
```

Feed the device

- Move your mouse
- Use your keyboard (write random text to /dev/null)

Example of commands:

```
find / &>/dev/null            # 50 bytes/seconde
dd if=/dev/hda of=/dev/null   # 15 bytes/seconde
hdparm -t /dev/hda            # 10 bytes/seconde
```

Control devices with /proc

Files of the directory `/proc/sys/kernel/random`:

- `poolsize`: Entropy pool size in bytes
- `entropy_avail`: Estimation of available entropy... bits or bytes?
- `read_wakeup_threshold`: number of bits
- `write_wakeup_threshold`: number of bits
- `boot_id`: UUID generated once at boot
- `uuid`: generate an UUID at each read

PRNG

Files: crypto/ansi_cprng.c, crypto/krng.c, crypto/rng.c

Neil Horman implemented ANSI X9.31 (Appendix A.2.4) using AES 128 cipher in Linux 2.6.28.

See also

- Manual page: RANDOM(4)
- Analysis of the Linux Random Number Generator (2006) by Zvi Gutterman, Benny Pinkas and Tzachy Reinman

Operating system services

Linux

- `/dev/random`: blocking. Based on timing of keyword, mouse, interruption and hard drive.
- `/dev/urandom`: non blocking. PRNG using /dev/random.

OpenBSD

- `/dev/random`: use hardware random number generator
- `/dev/srandom` and `/dev/urandom`: entropy pool data is converted in to output data using MD5. srandom blocks if entropy pool is empty, whereas urandom is non blocking.
- `/dev/prandom`: Simple pseudo-random generator (?)
- `/dev/arandom`: As required, entropy pool data re-seeds an ARC4 generator, which then generates high-quality pseudo-random output data.

Manual page: RANDOM(4).

FreeBSD

Sources of randomness from the environment include inter-keyboard timings, inter-interrupt timings from some interrupts, and other events which are both (a) non-deterministic and (b) hard for an outside observer to measure. Randomness from these sources are added to an "entropy pool", which is periodically mixed using the MD5 compression function in CBC mode.

- `/dev/random`: blocking
- `/dev/urandom`: not blocking

Manual page: RANDOM(4).

Usage of RNG in applications and libraries

GNU libc

Project website: http://www.gnu.org/software/libc/

Functions

- rand(), rand_r(), srand()
- random(), srandom(), initstate(), setstate()
- random_r(), srandom_r(), initstate_r(), setstate_r()
- drand48(), erand48(), lrand48(), nrand48(), mrand48(), jrand48(), srand48(), seed48(), lcong48()
- drand48_r(), erand48_r(), lrand48_r(), nrand48_r(), mrand48_r(), jrand48_r(), srand48_r(), seed48_r(), lcong48_r()
- strfry(): reuse __initstate_r() and __random_r()

Seed

- strfry() uses __initstate_r(time(NULL) ^ getpid())

Bugs

- mrand48 and it's friends return values outside of [-2^31, 2^31] on 64-bit platforms http://sourceware.org/bugzilla/show_bug.cgi?id=3747
- strfry() gives skewed distributions: http://sourceware.org/bugzilla/show_bug.cgi?id=4403
- use of long int in srandom_r() changes results under 64bit http://sourceware.org/bugzilla/show_bug.cgi?id=9920

Python

Project website: http://www.python.org/

Informations about Python 2.5

Files: Modules/_randommodule.c, Modules/posixmodule.c, Lib/random.py, Lib/os.py

Python 2.5 uses Mersenne Twister engine with os.urandom() as seed. If seed is None, it uses time(&now).

WichmannHill class implements Wichman-Hill random number generator. It was used as Random before Mersenne Twister was implemented (december 2002).

os.urandom()

urandom() has different implementations:

- Windows: win32_urandom(), in Modules/posixmodule.c, uses CryptGenRandom()
- vms_urandom(), in Modules/posixmodule.c, uses RAND_pseudo_bytes() from OpenSSL
- otherwise: urandom(), in Lib/os.py, reads /dev/urandom

Bugs in old Python versions

- (Jun 2010) Non-uniformity in randrange for large arguments. http://bugs.python.org/issue9025
- (Dec 2006) random.randrange don't return correct value for big number http://bugs.python.org/issue1590891 http://svn.python.org/view?view=rev&revision=53099
- (Oct 2003) randint is always even (fixed in revision 34364). http://bugs.python.org/issue812202 http://svn.python.org/view?view=rev&revision=34364
- (Aug 2003) Bad seed in python 2.3 random. The default seed is time.time(). Multiplied by 256 before truncating so that fractional seconds are used. http://bugs.python.org/issue778964 http://svn.python.org/view?view=rev&revision=33822
- (Feb 2001) Change random.seed() so that it can get at the full range of possible internal states. Put the old .seed() (which could only get at about the square root of the # of possibilities) under the new name .whseed(), for bit-level compatibility with older versions. Bug related to the old implementation using Wichman-Hill, not Mersenne Twister. http://svn.python.org/view?view=rev&revision=19241

PHP

Project website: http://www.php.net/

Informations about PHP5

Files: main/reentrance.c, ext/standard/php_rand.h, ext/standard/rand.c, ext/standard/lcg.c

PHP has two main functions: php_rand()->long and php_srand(long). It has four different methods to generate numbers depending on PHP compilation options:

- (ZTS mode) php_rand_r() + internal seed: The standard LCG (m=1103515245, a=12345)
- or random() + srandom(): libc best LCG
- or lrand48() + srand48(): libc 48 bits LCG
- or rand() + srand(): libc 32 bits LCG

The first method (Standard LCG) is used if PHP is compiled in Thread Safe mode (ZTS).

php_rand() setups the seed at the first call if php_srand() was not called, using GENERATE_SEED(). This macro uses:

- on Windows: time(), GetCurrentProcessId(), php_combined_lcg()
- on Linux: time(), getpid(), php_combined_lcg()

And php_combined_lcg() is seeded using: gettimeofday() and getpid() (or thread id in ZTS mode).

rand() has two prototypes: rand() or rand(min, max). If the range is specified, it uses: min + (long)((double)x * (max - min + 1.0) / (randmax + 1.0))

But PHP also includes Mersenne Twister! See functions: mt_rand(), mt_srand() and mt_getrandmax(). If mt_rand() is not seeded, it uses the same entropy than php_rand().

Bugs in old PHP versions

- (Jan 2010) Worked with Samy Kamkar to improve LCG entropy http://svn.php.net/viewvc?view=revision&revision=293253

- (Mar 2008) GENERATE_SEED() in PHP < 5.2.5 has bias http://svn.php.net/viewvc?view=revision&revision=254458

  Change:

```
# PHP 5.2.4 (bug)
(long)(time(0) * getpid() * 1000000 * php_combined_lcg(TSRMLS_C))

# PHP 5.2.5 (fixed)
((long)(time(0) * getpid())) ^ ((long) (1000000.0 * php_combined_lcg(TSRMLS_C)))
```

- (Jan 2004) rand(min, max) always returns min when ZTS enabled. RAND_MAX constant was invalid (2147483647 instead of 32767). http://bugs.php.net/bug.php?id=26949 http://svn.php.net/viewvc?view=revision&revision=149174
- (Aug 2003) rand() & mt_rand() seed RNG at each call (call srand()/mt_srand()). http://bugs.php.net/bug.php?id=25007 http://svn.php.net/viewvc?view=revision&revision=137294
- (Aug 2003) Problem with generation of random numbers on solaris http://bugs.php.net/bug.php?id=25170 http://svn.php.net/viewvc?view=revision&revision=138352
- (Aug 2003) rand function with range always returns low value of range (on Solaris) http://bugs.php.net/bug.php?id=24909 http://svn.php.net/viewvc?view=revision&revision=136996

Older versions didn't call srand() for rand(), shuffle(), etc.

Parrot

- Parrot 2.2: RNG non-randomness fixes http://www.parrot.org/news/2010/parrot-2.2.0
- Ticket #64: Parrot needs a source of entropy http://trac.parrot.org/parrot/ticket/64
- Ticket #392: Make rand/srand work the same for 32bit and 64bit int http://trac.parrot.org/parrot/changeset/37087 http://trac.parrot.org/parrot/ticket/392
- Ticket #923: Make RNG algorithm used by rand dynop pluggable http://trac.parrot.org/parrot/ticket/923

BIND

Project website: http://www.isc.org/software/bind

Files

- lib/dst/prandom.c
- lib/cylink/rand.c
- lib/dst/dst_api.c
- bin/named/ns_main.c

PRNG

BIND uses multiple PRNG depending on the program, and the PRNG depends on the version of BIND.

Seed:

- the best seed is in prandom.c: it uses /dev/random or a LCG

- some generators use getpid() and gettimeofday()

PRNG:

- one LCG in prandom.c
- two mixed LCG in ns_main.c

Bugs

- CVE-2007-2926: Weak random number generator (LFSR), by Amit Klein, extract the internal state from 13-15 consecutive transaction IDs http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-2926 http://www.trusteer.com/bind9dns
- April 22, 1997: The usage of predictable IDs. Patch creating the first res_randomid() function. http://www.openbsd.org/advisories/res_random.txt

glib

Project website: http://www.gtk.org/

File: glib/grand.c

Use Mersenne Twister. glib 2.0 uses the following code to seed with an 32 bits unsigned integer:

```
/* setting initial seeds to mt[N] using         */
/* the generator Line 25 of Table 1 in          */
/* [KNUTH 1981, The Art of Computer Programming */
/*    Vol. 2 (2nd Ed.), pp102]                  */
if (seed == 0) /* This would make the PRNG procude only zeros */
    seed = 0x6b842128; /* Just set it to another number */
mt[0]= seed;
for (mti=1; mti<N; mti++)
    mt[mti] = (69069 * mt[mti-1]);
```

whereas glib 2.2+ uses:

```
/* See Knuth TAOCP Vol2. 3rd Ed. P.106 for multiplier. */
/* In the previous version (see above), MSBs of the    */
/* seed affect only MSBs of the array mt[].            */
mt[0]= seed;
for (mti=1; mti<N; mti++)
    mt[mti] = 1812433253UL * (mt[mti-1] ^ (mt[mti-1] >> 30)) + mti;
```

To create the seed, g_rand_new() reads 128 bits (4 uint32) from /dev/urandom on UNIX. If the device can not be used or not available (eg. on Windows), it uses:

```
g_get_current_time(&now);
seed[0] = now.tv_sec;
seed[1] = now.tv_usec;
seed[2] = getpid();
#ifdef G_OS_UNIX
seed[3] = getppid();
#else
seed[3] = 0;
#endif
```

where g_get_current_time() is GetSystemTimeAsFileTime() on Windows and gettimeofday() on other OS.

g_rand_double() uses two Mersenne Twister ticks but a strange hack:

```
/* The following might happen due to very bad rounding luck, but
 * actually this should be more than rare, we just try again then */
if (retval >= 1.0)
    return g_rand_double (rand);
```

g_rand_int_range() has two versions: 20 (glib 2.0) and 22 (glib 2.2). Read the source code for the details...

pw_gen

File: randnum.c

Seed: use /dev/urandom, or open /dev/random in non blocking mode on failure, or drand48 on failure. drand48 is replaced by random() if drand48() is not available.

pw_random_number(max_num): read bytes into an unsigned int from /dev/(u)random and use (rand_num % max_num). If reading bytes fails, it uses:

```
#ifdef HAVE_DRAND48
    return ((int) ((drand48() * max_num)));
#else
    return ((int) (random() / ((float) RAND_MAX) * max_num));
#endif
```

libgcrypt

Project website: http://www.gnupg.org/

Files: cipher/random.c, cipher/rnd_*.c

Support:

- EGD: try in non blocking mode and then in blocking mode
- Linux: use /dev/random (level >= 2) or /dev/urandom
- UNIX: hash output of many programs like "vmstat", "w", "netstat", etc.
- W32: hash output of many variables (processes, threads, modules, netstat, disk performance, performance data, etc.

gcry_create_nonce()

- Initialization using time(NULL), getpid(), and a "private key" of 8 bytes generated by gcry_randomize(GCRY_WEAK_RANDOM)
- hash the internal state using SHA1() for each block of 20 bytes

OpenSSL

Project website: http://www.openssl.org/

Files

- crypto/rand/md_rand.c
- crypto/rand/rand_*.c

Functions

- RAND_bytes(), RAND_pseudo_bytes(): get random bytes
- RAND_add(), RAND_seed(), RAND_screen(), RAND_event(), RAND_egd(), RAND_egd_bytes(), RAND_query_egd_bytes(): add entropy
- RAND_status()
- RAND_file_name(), RAND_load_file(), RAND_write_file()
- RAND_cleanup()

By default, RAND_bytes() uses ssleay_rand_bytes().

ssleay_rand_bytes()

- call RAND_poll() at first call
- use SHA-1 hash
- use getpid() entropy (maybe to generate different numbers for each process)

RAND_poll()

- NetWare
  - GetProcessSwitchCount()
  - RunningProcess: current process address
  - RDTSC
  - GetSuperHighResolutionTimer()
- OpenBSD:
  - arc4random(): call it 8 times (to get 256 bits)
- OS/2:
  - DosTmrQueryTime()
  - DosQuerySysInfo()
  - DosPerfSysCall()
  - DosQuerySysState()
- UNIX
  - read 32 bytes from /dev/urandom, /dev/random, /dev/srandom, or EGD
  - getpid()
  - getuid()
  - time(NULL)

- VMS
  - sys$getjpiw(): processes
  - sys$gettim(): time
  - VxWorks:
  - nothing
- Windows:
  - CryptGenRandom(): read 64 bytes
  - GetForegroundWindow(): window handle
  - GetCursorInfo(): cursor position
  - GetQueueStatus(): message queue status
  - Heap32ListFirst()
  - Heap32First()
  - Process32First(), Process32Next(): processes
  - Thread32First(), Thread32Next(): threads
  - Module32First(), Module32Next(): modules
  - GlobalMemoryStatus()
  - GetCurrentProcessId()
- Windows CE (replace CryptGenRandom()):
  - netstatget(L"LanmanWorkstation")
  - netstatget(L"LanmanServer")
  - CryptGenRandom(): read 64 bytes
  - CryptGenRandom(), poll the Pentium PRG: read 64 bytes

RAND_bytes() and fork()

After many fork(), if two processes get the same identifier, RAND_bytes() generates the same bytes.

Bugs

- CVE-2008-0166: OpenSSL 0.9.8c-1 up to versions before 0.9.8g-9 on Debian-based operating systems uses a random number generator that generates predictable numbers, which makes it easier for remote attackers to conduct brute force guessing attacks against cryptographic keys. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166

GSL

Project website: http://www.gnu.org/software/gsl/

Files: rng/*.c, rng/gsl_rng.h

Implement many many PRNG.

gsl_rng_uniform_int(n) with n in [1; tick_max]:

```
scale = tick_max / n;
do {
    result = tick() / scale;
} while (result >= n);
```

Worst case: n = (tick_max/2 + 1), the while will fails 50% of the time.

gsl_rng_uniform_pos():

```
do {
    x = get_double();
} while (x == 0);
```

libcrypto++

Project website: http://www.cryptopp.com/

Files: osrng.cpp, rng.cpp

Classes: AutoSeededX917RNG, MicrosoftCryptoProvider, AutoSeededRandomPool

Engines:

- MicrosoftCryptoProvider: use CryptGenRandom() from Windows API
- NonblockingRng: /dev/urandom or MicrosoftCryptoProvider
- X917RNG: ANSI X9.17 Appendix C. Based on time(NULL), clock(), XOR and a block cipher (eg. DES-EDE3)
- LC_RNG: Linear Congruential Generator with m=2147483647 and a=16807, or m=2147483647 and a=48271 (original numbers)
- AutoSeededX917RNG: initialize X917RNG using SHA-256 of OS random (default: non blocking OS device)

Other:

- MaurerRandomnessTest: the class implements Maurer's Universal Statistical Test for Random Bit Generators it is intended for measuring the randomness of *PHYSICAL* RNGs. For more details see his paper in Journal of Cryptology, 1992.

Notes:

- X917RNG raises an exception if two sequences of bytes are identical

GMP

Project website: http://gmplib.org/

Files:

- randmt.c, randmt.h: Mersenne Twister (no seed)
- randmts.c: Mersenne Twister (with seed)
- randlc2s.c: linear congruential generator (2^32..2^256)

SPRNG

- SRC/makeseed.c: make_new_seed() uses time() + localtime()

libuuid

Files: gen_uuid.c

Funtions: get_random_bytes(), get_random_fd()

get_random_fd()

- try to open /dev/urandom or /dev/random
- on success, set FD_CLOEXEC flag on the file descriptor
- initialize srand() using:
  - getpid()
  - getuid()
  - gettimeofday() (use seconds and microseconds)
- initialize jrand_seed (array of 3 unsigned short) using:

```
jrand_seed[0] = getpid() ^ (tv.tv_sec & 0xFFFF);
jrand_seed[1] = getppid() ^ (tv.tv_usec & 0xFFFF);
jrand_seed[2] = (tv.tv_sec ^ tv.tv_usec) >> 16;
```

- calls rand() a random numbers of times: 0..31 times. Use gettimeofday() to generate the random number

rand() and jrand48() are initialized using the same timestamp.

jrand is used on Linux if gettid system call and jrand48() function are available.

get_random_bytes()

- read bytes from /dev/urandom or /dev/random, if the file is available. Stop after 16 consecutive read() errors.
- always mix the buffer with:

```
for (cp = buf, i = 0; i < nbytes; i++)
    *cp++ ^= (rand() >> 7) & 0xFF;
```

- if jrand is enabled, mix also the buffer with:

```
#ifdef DO_JRAND_MIX
    memcpy(tmp_seed, jrand_seed, sizeof(tmp_seed));
    jrand_seed[2] = jrand_seed[2] ^ syscall(__NR_gettid);
    for (cp = buf, i = 0; i < nbytes; i++)
        *cp++ ^= (jrand48(tmp_seed) >> 7) & 0xFF;
    memcpy(jrand_seed, tmp_seed,
           sizeof(jrand_seed)-sizeof(unsigned short));
#endif
```

get_clock()

Use:

```
THREAD_LOCAL uint16_t clock_seq;
get_random_bytes(&clock_seq, sizeof(clock_seq));
clock_seq &= 0x3FFF;
```

uuid__generate_time()

Create node_id once using:

```
static unsigned char node_id[6];
get_random_bytes(node_id, 6);
node_id[0] |= 0x01;   /* Set multicast bit */
```

uuid__generate_random()

Use:

```
get_random_bytes(buf, sizeof(buf));
... and set fixed bits (version, variant) ...
```

uuid_generate()

Use random UUID if `/dev/urandom` or `/dev/random` was opened correclty. Otherwise, generates an time based UUID.

Linux kernel

Project website: http://www.kernel.org/

Files

- drivers/hw_random/*.c
- drivers/char/random.c

Hardware RNG drivers

- AMD chipsets
- AMD Geode LX CPUs
- Intel chipsets
- Intel IXP4xx family of NPUs
- TI OMAP CPU family
- PA Semi processor
- VIA chipsets

Bugs

- "random: make get_random_int() more random" by Linus Torvalds, 5 May 2009 http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=8a0a9bd4db63bc45e3017bedeafbd88d0eb84d02
- "random: fix error in entropy extraction", by Matt Mackall, 30 May 2007 http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=602b6aeefe8932dd8bb15014e8fe6bb25d736361

Gnome keyring

Project website: http://live.gnome.org/GnomeKeyring

Files: /daemon/keyrings/gkr-keyring-binary.c

Functions: init_salt(), gkr_keyring_binary_generate()

Code to generate the hash iterations:

```
keyring->hash_iterations = 1000 + (int) (1000.0 * rand() / (RAND_MAX + 1.0));
```

Code to generate the salt:

```
static void
init_salt (guchar salt[8])
{
    gboolean got_random;
    int i, fd;

    got_random = FALSE;
#ifdef HAVE_DEVRANDOM
    fd = open ("/dev/random", O_RDONLY);
    if (fd != -1) {
        struct stat st;
        /* Make sure it's a character device */
        if ((fstat (fd, &st) == 0) && S_ISCHR (st.st_mode)) {
            if (read (fd, salt, 8) == 8) {
                got_random = TRUE;
            }
        }
        close (fd);
    }
#endif

    if (!got_random) {
        for (i=0; i < 8; i++) {
            salt[i] = (int) (256.0*rand()/(RAND_MAX+1.0));
        }
    }

}
```

OpenBSD

A vulnerability was introduced in the random device the 13 April 2000. It was discovered and fixed the 23 january 2001:

- http://www.openbsd.org/cgi-bin/cvsweb/src/sys/dev/rnd.c.diff?r1=1.35;r2=1.36

- http://ftp.openbsd.org/pub/OpenBSD/patches/2.8/common/017_rnd.patch

Most important change of the commit:

```
- for (; n--; buf++) {
+ while (n--) {
```

Because of this change, only 8 bits (the first byte of the buffer) was used, instead of the whole buffer.

CHAPTER 6

Hasard hacker documentation

Tests

Hasard contains many tests to detect implementation bugs. To run all tests, type "./run_tests.sh". The most important test program is python/test_hasard.py.

You can also use the following programs to check random number generators quality: dieharder, ENT, TestU01.

Tests

- Test entropy: python/file_info.py
- Use ENT program: python/ent.py
- Benchmark: benchmark.c
- Manual visual check: `python/gnuplot.py, python/draw_pil.py`

test_hasard.py

Options

test_hasard.py has many test suites applied on each generator. The script have command line options, the most useful are:

- –slow / –very-slow: Slower tests, but test more cases
- -r REPEAT: repeaet each test suite REPEAT times
- -v: verbose

You can also test only one generator using its name, eg. "test_hasard.py arcfour". Or you can run only some test suites using -k option, eg. "test_haasrd.py -k fork" runs all fork tests.

Tests

"init" test ensures that if two differents generators are created, the state is different. "reseed" is similar: it checks also for duplicate state, but after a "reseed" (call hasard_reseed() function). Both tests use the options –slow and –very-slow to choose how much tests are used:

- Default: 10
- –slow: 2^10
- –very-slow: 2^20

dieharder and ENT

python/dieharder.py and python/ent.py require an Hasard data file. Use python/gen_files.py program to create such file. Examples:

```
./python/gen_files.py --rng=@fast --count=2**20 fast_ticks.dat
./python/gen_files.py --rng=@fast --op=ulong --min=0 --max=2**16-1 fast_ulong16.dat
```

Dieharder requires at least 2^24 numbers and is very slow (can take more than one hour).

ENT output is not really revelant, use Dieharder, or better TestU01.

TestU01

TestU01 is a test suite written by Pierre l'Ecuyer: http://www.iro.umontreal.ca/~simardr/testu01/tu01.html

Example: "./tools/testu01 @fast SmallCrush".

Except SmallCrush and FIPS_140_2, most test suites are very slow (takes more than 10 hours).

Test RNG quality

Programs:

- TUT01 by Pierre l'Écuyer and Richard Simard
- DieHarder (2007) by Robert G. Brown
- ENT
- Diehard, Battery of Tests of Randomness (1995) by George Marsaglia, dedicated to RNG using 32 bits unsigned integer
- rngtest from the rng-tools package FIPS 140-2 (2001-10-10)

Documents:

- NIST 800-90 (2007): Recommendation for Random Number Generation Using Deterministic Random Bit Generators
  - Be careful with Dual_EC_DRBG: read The Strange Story of Dual_EC_DRBG by Bruce Schneier
- NIST 800-22 (2001): A statistical test suite for random and pseudorandom number generators for cryptographic applications
- Software Generation of Practically Strong Random Numbers (1998, updated in 2000) by Peter Gutmann

- [[ftp://ftp.isi.edu/in-notes/rfc1750.txt|RFC 1750: Randomness Recommendations for Security]] (1994) by D. Eastlake, S. Crocker and J. Schiller
- Testing random number generators (1992) by Pierre L'Ecuyer

Other:

- Random Number Generation Technical Working Group from the NIST

Hasard engine API

Warning: This documentation is outdated (Hasard version 0.6). Ask Hasard author to update it, or read directly Hasard source code.

Hasard library supports multiple "engines". An engine is an algorithm responsible to generate pseudo-random numbers.

Engine API is the private header engine.h (see also internals.h and engines.c).

Initialization

An engine is initialized using hasard_engine_init() function. This function calls the engine constructor found in the hasard_engines list. The constructor, eg. hasard_mersenne_twister_init(), allocates memory to store the engine internal state, setup attributes (eg. tick_max) and, the most important point, set the pseudo-random generator seed. The seed is created using hasard->seed engine: an hardware generator (eg. /dev/random).

On error, the constructor returns the number 1. Otherwise, it returns zero. You can use hasard->error(...) to display an error message.

Destruction

On exit or when the user changes the current engine, the engine is destroyed using hasard_engine_deinit). It calls the engine destructor if the engine has a destructor (eg. hasard_linux_dev_deinit). And then the engine internal state memory is freed.

Pseudo-random number generation

An engine have to implement at least one method to generate pseudo-random numbers: "tick" or "bytes". Both it's possible to use both, hasard will choose the faster generator for each user request.

Generation using ticks

The engine have to setup three attributes:

- ticks: the function callback
- tick_max: the maximum value of a tick (minimum is always zero)
- tick_bits: the minimum number of random bits in a tick

A tick is an unsigned integer in range [0; tick_max].

If your engine is generating numbers in [a; b] where a is not nul (eg. Park-Miller), substract the value "a" to get the output range [0; b-a].

Generation using bytes

An engine can also choose to generate unsigned bytes. The engine just have to setup the "bytes" function callback. The engine have to fill all requested bytes.

Random file format

Hasard use plain text file format for the different test scripts. The file has two sections: [header] contains the metadata (maximum value, value format, value count, etc.) and [data] contains the values. Example of a file:

```
[header]
version=0.6
format=decimal
count=4
minimum=0
maximum=2147483646
rng=park_miller
operation=tick
seed_engine=linux_urandom
seed=550135504

[data]
16807
282475249
1622647863
947787490
```

Header

Header contains informations about data. The format is "key=value".

Mandatory keys:

- version: file format version (0.6)
- format: Value format: "decimal", "bytes" or "float"
- library: Name the library
- rng: Name of the random number generator
- operation: Name of the operation used to generate the data
- minimum: Minimum value (integer or float, depends on the operation)
- maximum: Maximum value (integer or float, depends on the operation)
- count (int): Number of values in [data] sections
- seed_engine: Name of the hardware engine used to create the seed value

Optional keys:

- os: Name of the operating system
- seed: Value of the seed (integer)

Data

The data contains one value per line. The format depends on "format" header value:

- decimal: Decimal integer, eg. "1977746726"
- bytes: Long hexadecimal string, eg. "0xb27cff1e8a5393b790e613040b61b62e". It's better for text editors to use less then 80 characters per line (eg. 16 bytes per line).
- float: Decimal floating point number, eg. "0.00659407751671". There is no digit number minimal or maximal.

Compression

Since plain text uses is very verbose (files are huge), you can compress them using gzip or bzip2.

Comments

You can write comments anywhere using # character: the text after this character is removed. Empty line are also allowed anywhere. Example:

```
# This is a comment

[header]
maximum=255    # 2^8
```

Visualise random numbers using images

See python/gnuplot.py and python/draw_pil.py programs.

draw_pil.py

Lowest bit of libc_rand ticks:

```
./gen_files.py --rng=libc_rand --op=tick --mask=1 --count=256**2|./draw_pil.py
```

Lower (8) bits of RANDU ticks:

```
./gen_files.py --rng=randu --op=tick --mask=2**8-1 --count=256**2|./draw_pil.py
```

Lower (15) bits of minimum standard ticks:

```
./gen_files.py --op=tick --rng=minimum_standard --mask=2**15-1 --count=512**2|./draw_
↪pil.py --binary
```

Lower (8) bits of rand48 ticks:

```
./gen_files.py --rng=rand48 --op=tick --mask=255 --count=2048*100|./draw_pil.py --
↪width=2048
```

gnuplot.py

RANDU lower bits:

```
./gen_files.py --rng=randu --op=tick --count=256**2 --mask=255|./gnuplot.py --point=1
```

Counter:

```
./gen_files.py --rng=counter --op=tick --count=256**2|./gnuplot.py --point=0.25
```

Implement of randint()

BIND

PRNG engine: arcfour, with tick in [0; 0xffff]

Code:

```
static isc_uint16_t
dispatch_arc4uniformrandom(dns_dispatchmgr_t *mgr, isc_uint16_t upper_bound) {
        isc_uint16_t min, r;
        /* The caller must hold the manager lock. */

        if (upper_bound < 2)
                return (0);

        /*
         * Ensure the range of random numbers [min, 0xffff] be a multiple of
         * upper_bound and contain at least a half of the 16 bit range.
         */

        if (upper_bound > 0x8000)
                min = 1 + ~upper_bound; /* 0x8000 - upper_bound */
        else
                min = (isc_uint16_t)(0x10000 % (isc_uint32_t)upper_bound);

        /*
         * This could theoretically loop forever but each retry has
         * p > 0.5 (worst case, usually far better) of selecting a
         * number inside the range we need, so it should rarely need
         * to re-roll.
         */
        for (;;) {
                r = dispatch_arc4random(mgr);
                if (r >= min)
                        break;
        }

        return (r % upper_bound);
}
```

GSL

Code:

```
extern inline unsigned long int
gsl_rng_uniform_int (const gsl_rng * r, unsigned long int n)
{
  unsigned long int offset = r->type->min;
  unsigned long int range = r->type->max - offset;
  unsigned long int scale;
  unsigned long int k;

  if (n > range || n == 0)
    {
      GSL_ERROR_VAL ("invalid n, either 0 or exceeds maximum value of generator",
                     GSL_EINVAL, 0) ;
    }

  scale = range / n;

  do
    {
      k = (((r->type->get) (r->state)) - offset) / scale;
    }
  while (k >= n);

  return k;
}
```

glib

PRNG engine: Mersenne Twister, with tick in [0; 0xffffffff]

Code:

```
/**
 * g_rand_int_range:
 * @rand_: a #GRand.
 * @begin: lower closed bound of the interval.
 * @end: upper open bound of the interval.
 *
 * Returns the next random #gint32 from @rand_ equally distributed over
 * the range [@begin..@end-1].
 *
 * Return value: A random number.
 **/
gint32
g_rand_int_range (GRand* rand, gint32 begin, gint32 end)
{
  guint32 dist = end - begin;
  guint32 random;

  g_return_val_if_fail (rand != NULL, begin);
  g_return_val_if_fail (end > begin, begin);

  switch (get_random_version ())
    {
    case 20:
      if (dist <= 0x10000L) /* 2^16 */
        {
          /* This method, which only calls g_rand_int once is only good
```

```
           * for (end - begin) <= 2^16, because we only have 32 bits set
           * from the one call to g_rand_int (). */

          /* we are using (trans + trans * trans), because g_rand_int only
           * covers [0..2^32-1] and thus g_rand_int * trans only covers
           * [0..1-2^-32], but the biggest double < 1 is 1-2^-52.
           */

          gdouble double_rand = g_rand_int (rand) *
            (G_RAND_DOUBLE_TRANSFORM +
             G_RAND_DOUBLE_TRANSFORM * G_RAND_DOUBLE_TRANSFORM);

          random = (gint32) (double_rand * dist);
        }
      else
        {
          /* Now we use g_rand_double_range (), which will set 52 bits for
             us, so that it is safe to round and still get a decent
             distribution */
          random = (gint32) g_rand_double_range (rand, 0, dist);
        }
      break;
    case 22:
      if (dist == 0)
        random = 0;
      else
        {
          /* maxvalue is set to the predecessor of the greatest
           * multiple of dist less or equal 2^32. */
          guint32 maxvalue;
          if (dist <= 0x80000000u) /* 2^31 */
            {
              /* maxvalue = 2^32 - 1 - (2^32 % dist) */
              guint32 leftover = (0x80000000u % dist) * 2;
              if (leftover >= dist) leftover -= dist;
              maxvalue = 0xffffffffu - leftover;
            }
          else
            maxvalue = dist - 1;

          do
            random = g_rand_int (rand);
          while (random > maxvalue);

          random %= dist;
        }
      break;
    default:
      random = 0;                   /* Quiet GCC */
      g_assert_not_reached ();
    }

  return begin + random;
}
```

with:

```
/* transform [0..2^32] -> [0..1] */
#define G_RAND_DOUBLE_TRANSFORM 2.32830643653869628906e-10

/**
 * g_rand_double:
 * @rand_: a #GRand.
 *
 * Returns the next random #gdouble from @rand_ equally distributed over
 * the range [0..1).
 *
 * Return value: A random number.
 **/
gdouble
g_rand_double (GRand* rand)
{
  /* We set all 52 bits after the point for this, not only the first
     32. Thats why we need two calls to g_rand_int */
  gdouble retval = g_rand_int (rand) * G_RAND_DOUBLE_TRANSFORM;
  retval = (retval + g_rand_int (rand)) * G_RAND_DOUBLE_TRANSFORM;

  /* The following might happen due to very bad rounding luck, but
   * actually this should be more than rare, we just try again then */
  if (retval >= 1.0)
    return g_rand_double (rand);

  return retval;
}

/**
 * g_rand_double_range:
 * @rand_: a #GRand.
 * @begin: lower closed bound of the interval.
 * @end: upper open bound of the interval.
 *
 * Returns the next random #gdouble from @rand_ equally distributed over
 * the range [@begin..@end).
 *
 * Return value: A random number.
 **/
gdouble
g_rand_double_range (GRand* rand, gdouble begin, gdouble end)
{
  return g_rand_double (rand) * (end - begin) + begin;
}
```

Shuffle

Python trunk

random.py:

```
def shuffle(self, x, random=None, int=int):
    """x, random=random.random -> shuffle list x in place; return None.

    Optional arg random is a 0-argument function returning a random
```

```
    float in [0.0, 1.0); by default, the standard random.random.
    """

    if random is None:
        random = self.random
    for i in reversed(xrange(1, len(x))):
        # pick an element in x[:i+1] with which to exchange x[i]
        j = int(random() * (i+1))
        x[i], x[j] = x[j], x[i]
```

glibc 2.10

string/strfry.c:

```
char *
strfry (char *string)
{
  static int init;
  static struct random_data rdata;

  if (!init)
    {
      static char state[32];
      rdata.state = NULL;
      __initstate_r (time ((time_t *) NULL) ^ getpid (),
                     state, sizeof (state), &rdata);
      init = 1;
    }

  size_t len = strlen (string);
  if (len > 0)
    for (size_t i = 0; i < len - 1; ++i)
      {
        int32_t j;
        __random_r (&rdata, &j);
        j = j % (len - i) + i;

        char c = string[i];
        string[i] = string[j];
        string[j] = c;
      }

  return string;
}
```

glibc 2.6.1

string/strfry.c:

```
char *
strfry (char *string)
{
  static int init;
  static struct random_data rdata;
  size_t len, i;
```

```
  if (!init)
    {
      static char state[32];
      rdata.state = NULL;
      __initstate_r (time ((time_t *) NULL) ^ getpid (),
                     state, sizeof (state), &rdata);
      init = 1;
    }

  len = strlen (string) - 1;
  for (i = 0; i < len; ++i)
    {
      int32_t j;
      __random_r (&rdata, &j);
      j = j % len + 1;

      char c = string[i];
      string[i] = string[j];
      string[j] = c;
    }

  return string;
}
```

PHP 5.2.3

ext/standard/array.c:

```
static void array_data_shuffle(zval *array TSRMLS_DC)
{
    Bucket **elems, *temp;
    HashTable *hash;
    int j, n_elems, rnd_idx, n_left;

    n_elems = zend_hash_num_elements(Z_ARRVAL_P(array));

    if (n_elems < 1) {
            return;
    }

    elems = (Bucket **)safe_emalloc(n_elems, sizeof(Bucket *), 0);
    hash = Z_ARRVAL_P(array);
    n_left = n_elems;

    for (j = 0, temp = hash->pListHead; temp; temp = temp->pListNext)
            elems[j++] = temp;
    while (--n_left) {
            rnd_idx = php_rand(TSRMLS_C);
            RAND_RANGE(rnd_idx, 0, n_left, PHP_RAND_MAX);
            if (rnd_idx != n_left) {
                    temp = elems[n_left];
                    elems[n_left] = elems[rnd_idx];
                    elems[rnd_idx] = temp;
            }
    }
```

```
    HANDLE_BLOCK_INTERRUPTIONS();
    hash->pListHead = elems[0];
    hash->pListTail = NULL;
    hash->pInternalPointer = hash->pListHead;

    for (j = 0; j < n_elems; j++) {
            if (hash->pListTail) {
                    hash->pListTail->pListNext = elems[j];
            }
            elems[j]->pListLast = hash->pListTail;
            elems[j]->pListNext = NULL;
            hash->pListTail = elems[j];
    }
    temp = hash->pListHead;
    j = 0;
    while (temp != NULL) {
            temp->nKeyLength = 0;
            temp->h = j++;
            temp = temp->pListNext;
    }
    hash->nNextFreeElement = n_elems;
    zend_hash_rehash(hash);
    HANDLE_UNBLOCK_INTERRUPTIONS();

    efree(elems);
}
```

CHAPTER 7

News

- 2013-12-17: Hasard 1.5, 4 years later!
- 2009-08-22: Hasard 1.0
- 2009-07-23: Hasard 0.9.7
- 2009-07-09: Hasard 0.9.6
- 2009-07-07: Hasard 0.9.5
- 2009-06-13: Hasard 0.9
- 2009-05-12: Hasard 0.8

Read also the ChangeLog.

CHAPTER 8

Similar projects

- OpenSSL includes PRNG
- libgcrypt includes PRNG
- Pseudo-Random Number Generator (PRNG) by Otmar Lendl and Josef Leydold
- Python language includes its own random library written in Python (random.py)
- GNU Scientific Library (GSL)
- Boost Random Number Library
- The Scalable Parallel Random Number Generators Library (SPRNG)
- pLab-package: The pLab-package includes portable high-performance implementations of the linear congruential, quadratic congruential, inversive congruential, and explicit inversive congruential random number generators (LCG, QCG, ICG, and EICG, respectively), which were designed and implemented by Otmar Lendl.

CHAPTER 9

Random links

- The eSTREAM Project: portfolio of promising new stream ciphers
- Computer Random vs. True Random (PHP, Windows and random.org)
- PHP rand(0,1) on Windows < OpenSSL rand() on Debian
- (fr) Comment générer des nombres aléatoires avec un ordinateur (HSC)
- Hasard library on Freshmeat
- (fr) Sortie de la bibliothèque Hasard version 0.2 (may 2008) on linuxfr.org
- (fr) Des Trappes dans les Clés (2003) by Eric Wegrzynowski
- (fr) Développement de la bibliothèque Hasard (june 2008)
- Entropy and Random Numbers (Henric Jungheim)
- Random Number Results (Bob Jenkins)
- mt_srand and not so random numbers
- (fr) Cryptographie et reverse engineering en environnement Win32 by Kostya Kortchinsky: Weak cryptography because of weak PRNG (even of correct RSA implementation)
- Chapiter 6 by Peter Gutmann (2001)
- Cryptanalysis of the Random Number Generator of the Windows Operating System, Dorrendorf, Leo; Zvi Gutterman, Benny Pinkas

Index

H