MSC30-C. Do not use the rand() function for generating pseudorandom numbers

Pseudorandom number generators use mathematical algorithms to produce a sequence of numbers with good statistical properties, but the numbers produced are not genuinely random.

The C Standard rand() function makes no guarantees as to the quality of the random sequence produced. The numbers generated by some implementations of rand() have a comparatively short cycle and the numbers can be predictable. Applications that have strong pseudorandom number requirements must use a generator that is known to be sufficient for their needs.

Noncompliant Code Example

The following noncompliant code generates an ID with a numeric part produced by calling the rand() function. The IDs produced are predictable and have limited randomness.

```c
#include <stdio.h>
#include <stdlib.h>
enum { len = 12 };
void func(void) {
    /*
    * id will hold the ID, starting with the characters
    *  "ID" followed by a random integer.
    */
    char id[len];
    int r;
    int num;
    /* ... */
    r = rand(); /* Generate a random integer */
    num = snprintf(id, len, "ID%-d", r); /* Generate the ID */
    /* ... */
}
```

Compliant Solution (POSIX)

This compliant solution replaces the rand() function with the POSIX random() function:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
enum { len = 12 };
void func(void) {
    /*
    * id will hold the ID, starting with the characters
    *  "ID" followed by a random integer.
    */
    char id[len];
    int r;
    int num;
    /* ... */
    struct timespec ts;
    if (timespec_get(&ts, TIME_UTC) == 0) {
        /* Handle error */
    }
    srand(random(ts.tv_nsec ^ ts.tv_sec)); /* Seed the PRNG */
    /* ... */
    r = random(); /* Generate a random integer */
    num = snprintf(id, len, "ID%-d", r); /* Generate the ID */
    /* ... */
}
```
The POSIX `random()` function is a better pseudorandom number generator. Although on some platforms the low dozen bits generated by `rand()` go through a cyclic pattern, all the bits generated by `random()` are usable. The rand48 family of functions provides another alternative for pseudorandom numbers.

Although not specified by POSIX, `arc4random()` is another possibility for systems that support it. The `arc4random(3)` manual page [OpenBSD] states... provides higher quality of data than those described in `rand(3)`, `random(3)`, and `drand48(3)`.

To achieve the best random numbers possible, an implementation-specific function must be used. When unpredictability is crucial and speed is not an issue, as in the creation of strong cryptographic keys, use a true entropy source, such as `/dev/random`, or a hardware device capable of generating random numbers. The `/dev/random` device can block for a long time if there are not enough events going on to generate sufficient entropy.

Compliant Solution (Windows)

On Windows platforms, the `BCryptGenRandom()` function can be used to generate cryptographically strong random numbers. The Microsoft Developer Network `BCryptGenRandom()` reference [MSDN] states:

```
#include <Windows.h>
#include <bcrypt.h>
#include <stdio.h>
#pragma comment(lib, "Bcrypt")

void func(void) {
  BCRYPT_ALG_HANDLE Prov;
  int Buffer;
  if (!BCRYPT_SUCCESS(BCryptOpenAlgorithmProvider(&Prov, BCRYPT_RNG_ALGORITHM, NULL, 0))) {
    /* handle error */
  }
  if (!BCRYPT_SUCCESS(BCryptGenRandom(Prov, (PUCHAR) (&Buffer), sizeof(Buffer), 0))) {
    /* handle error */
  }
  printf("Random number: %d\n", Buffer);
  B CryptCloseAlgorithmProvider(Prov, 0);
}
```

Risk Assessment

The use of the `rand()` function can result in predictable random numbers.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSC30-C</td>
<td>Medium</td>
<td>Unlikely</td>
<td>Low</td>
<td>P6</td>
<td>L2</td>
</tr>
</tbody>
</table>

Automated Detection

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Astrée</td>
<td>19.04</td>
<td>stdlib-use-rand</td>
<td>Fully checked</td>
</tr>
<tr>
<td>Axivion Bauhaus Suite</td>
<td>6.9.0</td>
<td>Cert-MSC30</td>
<td></td>
</tr>
<tr>
<td>Clang</td>
<td>4.0 (prerelease)</td>
<td>cert-msc30-c</td>
<td>Checked by clang-tidy</td>
</tr>
<tr>
<td>CodeSonar</td>
<td>5.2p0</td>
<td>BADFUNC.RANDOM.RAND</td>
<td></td>
</tr>
<tr>
<td>Compass/ROSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coverity</td>
<td>2017.07</td>
<td>DONTCALL</td>
<td>Implemented - weak support</td>
</tr>
<tr>
<td>ECLAIR</td>
<td>1.2</td>
<td>CC2.MSC30</td>
<td>Fully implemented</td>
</tr>
</tbody>
</table>
LDRA tool suite 9.7.1 44 S Enhanced enforcement
Parasoft C/C++test 10.4.2 CERT_C-MSC30-a Do not use the rand() function for generating pseudorandom numbers
Polyspace Bug Finder R2019b CERT C: Rule MSC30-C Checks for vulnerable pseudo-random number generator (rule fully covered)
PRQA QA-C 9.7 5022 Fully implemented
PRQA QA-C++ 4.4 5029
RuleChecker 19.04 stdlib-use-rand Fully checked

Related Vulnerabilities
Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Related Guidelines
Key here (explains table format and definitions)

<table>
<thead>
<tr>
<th>Taxonomy</th>
<th>Taxonomy item</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERT C</td>
<td>MSC50-CPP. Do not use std::rand() for generating pseudorandom numbers</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CERT Oracle Secure Coding Standard for Java</td>
<td>MSC02-J. Generate strong random numbers</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CWE 2.11</td>
<td>CWE-327, Use of a Broken or Risky Cryptographic Algorithm</td>
<td>2017-05-16: CERT: Rule subset of CWE</td>
</tr>
<tr>
<td>CWE 2.11</td>
<td>CWE-330, Use of Insufficiently Random Values</td>
<td>2017-06-28: CERT: Rule subset of CWE</td>
</tr>
<tr>
<td>CWE 2.11</td>
<td>CWE-338, Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)</td>
<td>2017-06-28: CERT: Rule subset of CWE</td>
</tr>
<tr>
<td>CWE 2.11</td>
<td>CWE-676</td>
<td>2017-05-18: CERT: Rule subset of CWE</td>
</tr>
</tbody>
</table>

CERT-CWE Mapping Notes
Key here for mapping notes

CWE-327 and MSC30-C

- CWE-327 forbids “broken or risky cryptographic algorithms” but does not specify what constitutes such an algo.

- Per CERT judgement, rand() qualifies, so:

- CWE-327 = Union( MSC30-C, list) where list =

- Invocation of broken/risky crypto algorithms besides rand()

CWE-338 and MSC30-C
CWE-338 = Union( MSC30-C, list) where list =

- Use of a weak PRNG besides standard C rand().
CWE-330 and MSC30-C

Independent( MSC30-C, MSC32-C, CON33-C)

CWE-330 = Union( MSC30-C, MSC32-C, CON33-C, list) where list = other improper use or creation of random values. (EG the would qualify)

MSC30-C, MSC32-C and CON33-C are independent, they have no intersections. They each specify distinct errors regarding PRNGs.

CWE-676 and MSC30-C

- Independent( ENV33-C, CON33-C, STR31-C, EXP33-C, MSC30-C, ERR34-C)

- MSC30-C implies that rand() is dangerous.

- CWE-676 = Union( MSC30-C, list) where list =

  - Invocation of other dangerous functions, besides rand().

Bibliography

<table>
<thead>
<tr>
<th>[MSDN]</th>
<th>&quot;BCryptGenRandom() Function&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>[OpenBSD]</td>
<td>arc4random()</td>
</tr>
</tbody>
</table>