MSC32-C. Properly seed pseudorandom number generators

A pseudorandom number generator (PRNG) is a deterministic algorithm capable of generating sequences of numbers that approximate the properties of random numbers. Each sequence is completely determined by the initial state of the PRNG and the algorithm for changing the state. Most PRNGs make it possible to set the initial state, also called the seed state. Setting the initial state is called seeding the PRNG.

Calling a PRNG in the same initial state, either without seeding it explicitly or by seeding it with the same value, results in generating the same sequence of random numbers in different runs of the program. Consider a PRNG function that is seeded with some initial seed value and is consecutively called to produce a sequence of random numbers. If the PRNG is subsequently seeded with the same initial seed value, then it will generate the same sequence.

As a result, after the first run of an improperly seeded PRNG, an attacker can predict the sequence of random numbers that will be generated in the future runs. Improperly seeding or failing to seed the PRNG can lead to vulnerabilities, especially in security protocols.

The solution is to ensure that the PRNG is always properly seeded. A properly seeded PRNG will generate a different sequence of random numbers each time it is run.

Not all random number generators can be seeded. True random number generators that rely on hardware to produce completely unpredictable results do not need to be and cannot be seeded. Some high-quality PRNGs, such as the /dev/random device on some UNIX systems, also cannot be seeded. This rule applies only to algorithmic pseudorandom number generators that can be seeded.

Noncompliant Code Example (POSIX)

This noncompliant code example generates a sequence of 10 pseudorandom numbers using the random() function. When random() is not seeded, it behaves like rand(), producing the same sequence of random numbers each time any program that uses it is run.

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

void func(void) {
    for (unsigned int i = 0; i < 10; ++i) {
        /* Always generates the same sequence */
        printf("%ld, ", random());
    }
}
```

The output is as follows:

1st run: 1804289383, 846930886, 1681692777, 1714636915, 1957747793, 424238335, 719885386, 1649760492, 596516649, 1189641421,
2nd run: 1804289383, 846930886, 1681692777, 1714636915, 1957747793, 424238335, 719885386, 1649760492, 596516649, 1189641421,
...  
nth run: 1804289383, 846930886, 1681692777, 1714636915, 1957747793, 424238335, 719885386, 1649760492, 596516649, 1189641421,

Compliant Solution (POSIX)

Call srand() before invoking random() to seed the random sequence generated by random(). This compliant solution produces different random number sequences each time the function is called, depending on the resolution of the system clock:
void func(void) {
    struct timespec ts;
    if (timespec_get(&ts, TIME_UTC) == 0) {
        /* Handle error */
    } else {
        srand(ts.tv_nsec ^ ts.tv_sec);
        for (unsigned int i = 0; i < 10; ++i) {
            /* Generates different sequences at different runs */
            printf("%ld, ", random());
        }
    }
}

The output is as follows:

1st run: 198682410, 2076262355, 910374899, 428635843, 2084827500, 1558698420, 4459146, 733695321, 2044378618,
1649046624,
2nd run: 1127071427, 252907983, 1358798372, 2101446505, 1514711759, 229790273, 954268511, 1116446419, 368192457,
1297948050,
3rd run: 2052868434, 1645663878, 731874735, 1624006793, 938447420, 1046134947, 1901136083, 418123888, 836428296,
2017467418,

This may not be sufficiently random for concurrent execution, which may lead to correlated generated series in different threads. Depending on the application and the desired level of security, a programmer may choose alternative ways to seed PRNGs. In general, hardware is more capable than software of generating real random numbers (for example, by sampling the thermal noise of a diode).

Compliant Solution (Windows)

The BCryptGenRandom() function does not run the risk of not being properly seeded because its arguments serve as seeders:

void func(void) {
    BCRYPT_ALG_HANDLE hAlgorithm = NULL;
    long rand_buf;
    UCHAR pbBuffer = (UCHAR) &rand_buf;
    ULONG cbBuffer = sizeof(rand_buf);
    ULONG dwFlags = BCRYPT_USE_SYSTEM_PREFERRED_RNG;
    NTSTATUS status;
    for (unsigned int i = 0; i < 10; ++i) {
        status = BCryptGenRandom(hAlgorithm, pbBuffer, cbBuffer, dwFlags);
        if (status == STATUS_SUCCESS) {
            printf("%ld, ", rand_buf);
        } else {
            /* Handle Error */
        }
    }
}

The output is as follows:
Risk Assessment

<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>MSC32-C</td>
<td>Medium</td>
<td>Likely</td>
<td>Low</td>
<td>P18</td>
<td>L1</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>CertC-MSC32</td>
<td>Supported, but no explicit checker</td>
</tr>
<tr>
<td>Axivion Bauhaus Suite</td>
<td>6.9.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2019b</td>
<td>CERT C: Rule MSC32-C</td>
<td>Checks for:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Deterministic random output from constant seed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Predictable random output from predictable seed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rule fully covered.</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_C-MSC32-a, CERT_C-MSC32-b, CERT_C-MSC32-c, CERT_C-MSC32-d</td>
<td>Avoid functions which use random numbers from standard C library</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Do not use the rand() function for generating pseudorandom numbers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Standard random number generators should not be used to generate randomness for security reasons</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Properly seed pseudorandom number generators</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>5031</td>
<td></td>
</tr>
</tbody>
</table>

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 Secure Coding Standard</td>
<td>MSC30-C. Do not use the rand() function for generating pseudorandom numbers</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CERT C</td>
<td>MSC51-CPP. Ensure your random number generator is properly seeded</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-331, Insufficient Entropy</td>
<td>2017-06-28: CERT: Exact</td>
</tr>
</tbody>
</table>

CERT-CWE Mapping Notes

Key here for mapping notes

CWE-327 and MSC32-C

- Intersection( MSC30-C, MSC32-C) = Ø
MSC32-C says to properly seed pseudorandom number generators. For example, if you call rand(), make sure to seed it properly by calling srand() first. So far, we haven't found any calls to rand().

Failure to seed a PRNG causes it to produce reproducible (hence insecure) series of random numbers.

CWE-327 = Union(MSC32-C, list) where list =

Invocation of broken/risky crypto algorithms that are not properly seeded

CWE-330 and MSC32-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.

Bibliography
[MSDN] "BCryptGenRandom() Function"