MSC51-CPP. Ensure your random number generator is properly seeded

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 a constant 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.

Consequently, 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 a PRNG is always properly seeded with an initial seed value that will not be predictable or controllable by an attacker. 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 PRNGs that can be seeded.

Noncompliant Code Example

This noncompliant code example generates a sequence of 10 pseudorandom numbers using the Mersenne Twister engine. No matter how many times this code is executed, it always produces the same sequence because the default seed is used for the engine.

```
#include <random>
#include <iostream>

void f() {
  std::mt19937 engine;

  for (int i = 0; i < 10; ++i) {
    std::cout << engine() << ", ";
  }
}
```

The output of this example follows.

```
1st run: 3499211612, 581869302, 3890346734, 3586334585, 545404204, 4161255391, 3922919429, 949333985,
2715962298, 1323567403,
2nd run: 3499211612, 581869302, 3890346734, 3586334585, 545404204, 4161255391, 3922919429, 949333985,
2715962298, 1323567403,
...
nth run: 3499211612, 581869302, 3890346734, 3586334585, 545404204, 4161255391, 3922919429, 949333985,
2715962298, 1323567403,
```

Noncompliant Code Example

This noncompliant code example improves the previous noncompliant code example by seeding the random number generation engine with the current time. However, this approach is still unsuitable when an attacker can control the time at which the seeding is executed. Predictable seed values can result in exploits when the subverted PRNG is used.

```
#include <ctime>
#include <random>
#include <iostream>

void f() {
  std::time_t t;
  std::mt19937 engine(std::time(&t));

  for (int i = 0; i < 10; ++i) {
    std::cout << engine() << ", ";
  }
}
```

Compliant Solution

This compliant solution uses `std::random_device` to generate a random value for seeding the Mersenne Twister engine object. The values generated by `std::random_device` are nondeterministic random numbers when possible, relying on random number generation devices, such as `/dev/random`. When such a device is not available, `std::random_device` may employ a random number engine; however, the initial value generated should have sufficient randomness to serve as a seed value.

```
#include <random>
#include <iostream>

void f() {
  std::random_device dev;
  std::mt19937 engine(dev());

  for (int i = 0; i < 10; ++i) {
    std::cout << engine() << ", ";
  }
}
```

The output of this example follows.

```
1st run: 3921124303, 1253168518, 1183339582, 197772533, 83186419, 2599073270, 3238222340, 101548389, 296330365,
3335314032,
2nd run: 2392369099, 2509898672, 2135685437, 3733236524, 883966369, 2529945396, 764222328, 138530885,
4209173263, 1693483251,
3rd run: 914243768, 2191798381, 2961426773, 3791073717, 2222867426, 1092675429, 2202201605, 850375565,
3622398137, 422940882,
...
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC51-CPP	Medium	Likely	Low	**P18**	**L1**

Automated Detection

Tool	Version	Checker	Description
Axivion Bauhaus Suite	6.9.0	**CertC++-MSC51**	
Polyspace Bug Finder	R2019b	CERT C++: MSC51-CPP	Checks for: • Deterministic random output from constant seed • Predictable random output from predictable seed Rule partially covered.
Parasoft C/C++test	10.4.2	**CERT_CPP-MSC51-a**	Properly seed pseudorandom number generators

Related Vulnerabilities

Using a predictable seed value, such as the current time, result in numerous vulnerabilities, such as the one described by CVE-2008-1637.

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

Related Guidelines

SEI CERT C Coding Standard	MSC32-C. Properly seed pseudorandom number generators
MITRE CWE	CWE-327, Use of a Broken or Risky Cryptographic Algorithm CWE-330, Use of Insufficiently Random Values CWE-337, Predictable Seed in PRNG

Bibliography

[ISO/IEC 9899:2011]	Subclause 7.22.2, "Pseudo-random Sequence Generation Functions"
[ISO/IEC 14882-2014]	Subclause 26.5, "Random Number Generation"