MSC11-C. Incorporate diagnostic tests using assertions

Incorporate diagnostic tests into your program using, for example, the `assert()` macro.

The `assert` macro expands to a void expression:

```
#include <assert.h>
void assert(scalar expression);
```

When it is executed, if `expression` (which must have a scalar type) is false, the `assert` macro outputs information about the failed assertion (including the text of the argument, the name of the source file, the source line number, and the name of the enclosing function) on the standard error stream, in an implementation-defined format, and calls the `abort()` function.

In the following example, the test for integer wrap was omitted for the unsigned multiplication on the assumption that `MAX_TABLE_SIZE * sizeof (char *)` cannot exceed `SIZE_MAX`. Although we *know* this is true, it cannot do any harm to codify this assumption.

```
assert(size <= SIZE_MAX/sizeof(char *));
table_size = size * sizeof(char *);
```

Assertions are primarily intended for use during debugging and are generally turned off before code is deployed by defining the `NDEBUG` macro (typically as a flag passed to the compiler). Consequently, assertions should be used to protect against incorrect programmer assumptions and not for runtime error checking.

Assertions should never be used to verify the absence of runtime (as opposed to logic) errors, such as

- Invalid user input (including command-line arguments and environment variables)
- File errors (for example, errors opening, reading or writing files)
- Network errors (including network protocol errors)
- Out-of-memory conditions (for example, `malloc()` or similar failures)
- System resource exhaustion (for example, out-of-file descriptors, processes, threads)
- System call errors (for example, errors executing files, locking or unlocking mutexes)
- Invalid permissions (for example, file, memory, user)

Code that protects against a buffer overflow, for example, cannot be implemented as an assertion because this code must be presented in the deployed executable.

In particular, assertions are generally unsuitable for server programs or embedded systems in deployment. A failed assertion can lead to a denial-of-service attack if triggered by a malicious user, such as `size` being derived, in some way, from client input. In such situations, a soft failure mode, such as writing to a log file and rejecting the request, is more appropriate.

```
if (size > SIZE_MAX / sizeof(char *)) {
  fprintf(log_file, "%s: size %zu exceeds %zu bytes\n",
          __FILE__, size, SIZE_MAX / sizeof(char *));
  size = SIZE_MAX / sizeof(char *);
}
table_size = size * sizeof(char *);
```

Noncompliant Code Example (`malloc()`)

This noncompliant code example uses the `assert()` macro to verify that memory allocation succeeded. Because memory availability depends on the overall state of the system and can become exhausted at any point during a process lifetime, a robust program must be prepared to gracefully handle and recover from its exhaustion. Consequently, using the `assert()` macro to verify that a memory allocation succeeded would be inappropriate because doing so might lead to an abrupt termination of the process, opening the possibility of a denial-of-service attack. See also MEM11-C. Do not assume infinite heap space and void MEM32-C. Detect and handle memory allocation errors.

```
char *dupstring(const char *c_str) {
  size_t len;
  char *dup;

  len = strlen(c_str);
  dup = (char *)malloc(len + 1);
  assert(NULL != dup);

  memcpy(dup, c_str, len + 1);
  return dup;
}
```

Compliant Solution (`malloc()`)

This compliant solution demonstrates how to detect and handle possible memory exhaustion:

```
char *dupstring(const char *c_str) {
  size_t len;
  char *dup;

  len = strlen(c_str);
  dup = (char*)malloc(len + 1);
  /* Detect and handle memory allocation error */
  if (NULL == dup) {
      return NULL;
  }

  memcpy(dup, c_str, len + 1);
  return dup;
}
```

Risk Assessment

Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities. The absence of assertions, however, does not mean that code is incorrect.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MSC11-C	Low	Unlikely	High	**P1**	**L3**

Automated Detection

Tool	Version	Checker	Description
CodeSonar	5.1p0	**LANG.FUNCS. ASSERTS**	Not enough assertions
Coverity	2017.07	**ASSERT_SIDE_EFFECT**	Can detect the specific instance where assertion contains an operation/function call that may have a side effect
Parasoft C /C++test	10.4.2	**CERT_C-MSC11-a**	Assert liberally to document internal assumptions and invariants

Related Vulnerabilities

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

Related Guidelines

CERT C Secure Coding Standard	ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy
SEI CERT C++ Coding Standard	VOID MSC11-CPP. Incorporate diagnostic tests using assertions
MITRE CWE	CWE-190, Reachable assertion