MSC22-C. Use the setjmp(), longjmp() facility securely

The `setjmp()` macro should be invoked from only one of the contexts listed in subclause 7.13.1.1 of the C Standard [ISO/IEC 9899:2011]. Invoking `setjmp()` outside of one of these contexts results in undefined behavior. (See undefined behavior 125.)

After invoking `longjmp()`, non-volatile-qualified local objects should not be accessed if their values could have changed since the invocation of `setjmp()`. Their value in this case is considered indeterminate, and accessing them is undefined behavior. (See undefined behaviors 127 and 10.)

The `longjmp()` function should never be used to return control to a function that has terminated execution. (See undefined behavior 126.)

Signal masks, floating-point status flags, and the state of open files are *not* saved by the `setjmp()` function. If signal masks need to be saved, the POSIX `sigsetjmp()` function should be used.

This recommendation is related to SIG30-C. Call only asynchronous-safe functions within signal handlers and ENV32-C. All exit handlers must return normally.

Noncompliant Code Example

This noncompliant code example calls `setjmp()` in an assignment statement, resulting in undefined behavior:

```
jmp_buf buf;

void f(void) {
  int i = setjmp(buf);
  if (i == 0) {
    g();
  } else {
    /* longjmp was invoked */
  }
}

void g(void) {
  /* ... */
  longjmp(buf, 1);
}
```

Compliant Solution

Placing the call to `setjmp()` in the `if` statement and, optionally, comparing it with a constant integer removes the undefined behavior, as shown in this compliant solution:

```
jmp_buf buf;

void f(void) {
  if (setjmp(buf) == 0) {
    g();
  } else {
    /* longjmp was invoked */
  }
}

void g(void) {
  /* ... */
  longjmp(buf, 1);
}
```

Noncompliant Code Example

Any attempt to invoke the `longjmp()` function to transfer control to a function that has completed execution results in undefined behavior:

```
jmp_buf buf;
unsigned char b[] = {0xe5, 0x06, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00};

int main(void) {
  setup();
  do_stuff();
  return 0;
}

void setup(void) {
  f();
}

void f(void) {
  g();
}

void g(void) {
  if (setjmp(buf) == 0) {
    printf("setjmp() invoked\n");
  } else {
    printf("longjmp() invoked\n");
  }
}

void do_stuff(void) {
  char a[8];
  memcpy(a, b, 8);
  /* ... */
  longjmp(buf, 1);
}

void bad(void) {
  printf("Should not be called!\n");
  exit(1);
}
```

Implementation Details

Compiled for x86-64 using GCC 4.1.2 on Linux, the preceding example outputs the following when run:

```
setjmp() invoked
longjmp() invoked
Should not be called!
```

Because `g()` has finished executing at the time `longjmp()` is called, it is no longer on the stack. When `do_stuff()` is invoked, its stack frame occupies the same memory as the old stack frame of `g()`. In this case, a was located in the same location as the return address of function `g()`. The call to `memcpy()` overwrites the return address, so when `longjmp()` sends control back to function `g()`, the function returns to the wrong address (in this case, to function `bad()`).

If the array b were user specified, the user would be able to set the return address of function `g()` to any location.

Compliant Solution

The `longjmp()` function should be used only when the function containing the corresponding `setjmp()` is guaranteed not to have completed execution, as in the following example:

```
jmp_buf buf;
unsigned char b[] = {0xe5, 0x06, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00};

int main(void) {
  if (setjmp(buf) == 0) {
    printf("setjmp() invoked\n");
  } else {
    printf("longjmp() invoked\n");
  }
  do_stuff();
  return 0;
}

void do_stuff(void) {
  char a[8];
  memcpy(a, b, 8);
  /* ... */
  longjmp(buf, 1);
}

void bad(void) {
  printf("Should not be called!\n");
  exit(1);
}
```

There is no risk of overwriting a return address because the stack frame of `main()` (the function that invoked `setjmp()`) is still on the stack; so when `do_stuff()` is invoked, the two stack frames will not overlap.

Noncompliant Code Example

In this noncompliant example, non-volatile-qualified objects local to the function that invoked the corresponding `setjmp()` have indeterminate values after `longjmp()` is executed if their value has been changed since the invocation of `setjmp()`:

```
jmp_buf buf;

void f(void) {
  int i = 0;
  if (setjmp(buf) != 0) {
    printf("%i\n", i);
    /* ... */
  }
  i = 2;
  g();
}

void g(void) {
  /* ... */
  longjmp(buf, 1);
}
```

Compliant Solution

If an object local to the function that invoked `setjmp()` needs to be accessed after `longjmp()` returns control to the function, the object should be volatile-qualified:

```
jmp_buf buf;

void f(void) {
  volatile int i = 0;
  if (setjmp(buf) != 0) {
    printf("%i\n", i);
    /* ... */
  }
  i = 2;
  g();
}

void g(void) {
  /* ... */
  longjmp(buf, 1);
}
```

Risk Assessment

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
MSC22-C	Low	Probable	Medium	**P4**	**L3**

Automated Detection

Tool	Version	Checker	Description
CodeSonar	5.0p0	**BADFUNC.LONGJMP** **BADFUNC.SETJMP**	Use of longjmp Use of setjmp
LDRA tool suite	9.7.1	**43 S**	Enhanced enforcement
Parasoft C/C++test	10.4.2	**CERT_C-MSC22-a**	The setjmp macro and the longjmp function shall not be used
Polyspace Bug Finder	R2018a	Use of setjmp/longjmp	`setjmp` and `longjmp` cause deviation from normal control flow
SonarQube C/C++ Plugin	3.11	**S982**