ENV32-C. All exit handlers must return normally

The C Standard provides three functions that cause an application to terminate normally: _Exit(), exit(), and quick_exit(). These are collectively called exit functions. When the exit() function is called, or control transfers out of the main() entry point function, functions registered with atexit() are called (but not at_quick_exit()). When the quick_exit() function is called, functions registered with at_quick_exit() (but not atexit()) are called. These functions are collectively called exit handlers. When the _Exit() function is called, no exit handlers or signal handlers are called.

Exit handlers must terminate by returning. It is important and potentially safety-critical for all exit handlers to be allowed to perform their cleanup actions. This is particularly true because the application programmer does not always know about handlers that may have been installed by support libraries. Two specific issues include nested calls to an exit function and terminating a call to an exit handler by invoking longjmp.

A nested call to an exit function is undefined behavior. (See undefined behavior 182.) This behavior can occur only when an exit function is invoked from an exit handler or when an exit function is called from within a signal handler. (See SIG30-C. Call only asynchronous-safe functions within signal handlers.)

If a call to the longjmp() function is made that would terminate the call to a function registered with atexit(), the behavior is undefined.

Noncompliant Code Example

In this noncompliant code example, the exit1() and exit2() functions are registered by atexit() to perform required cleanup upon program termination. However, if some_condition evaluates to true, exit() is called a second time, resulting in undefined behavior.

```c
#include <stdlib.h>

void exit1(void) {
    /* ... Cleanup code ... */
    return;
}

void exit2(void) {
    extern int some_condition;
    if (some_condition) {
        /* ... More cleanup code ... */
        exit(0);
    }
    return;
}

int main(void) {
    if (atexit(exit1) != 0) {
        /* Handle error */
    }
    if (atexit(exit2) != 0) {
        /* Handle error */
    }
    /* ... Program code ... */
    return 0;
}
```

Functions registered by the atexit() function are called in the reverse order from which they were registered. Consequently, if exit2() exits in any way other than by returning, exit1() will not be executed. The same may also be true for atexit() handlers installed by support libraries.

Compliant Solution

A function that is registered as an exit handler by atexit() must exit by returning, as in this compliant solution:
Noncompliant Code Example

In this noncompliant code example, `exit1()` is registered by `atexit()` so that upon program termination, `exit1()` is called. The `exit1()` function jumps back to `main()` to return, with undefined results.

```c
#include <stdlib.h>
#include <setjmp.h>
jmp_buf env;
int val;

void exit1(void) {
    longjmp(env, 1);
}

int main(void) {
    if (atexit(exit1) != 0) {
        /* Handle error */
    }
    if (setjmp(env) == 0) {
        exit(0);
    } else {
        return 0;
    }
}
```

Compliant Solution

This compliant solution does not call `longjmp()` but instead returns from the exit handler normally:

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

void exit1(void) {
    /* ... Cleanup code ... */
    return;
}

void exit2(void) {
    extern int some_condition;
    if (some_condition) {
        /* ... More cleanup code ... */
    }
    return;
}

int main(void) {
    if (atexit(exit1) != 0) {
        /* Handle error */
    }
    if (atexit(exit2) != 0) {
        /* Handle error */
    }
    /* ... Program code ... */
    return 0;
}
```
Risk Assessment

Terminating a call to an exit handler in any way other than by returning is undefined behavior and may result in abnormal program termination or other unpredictable behavior. It may also prevent other registered handlers from being invoked.

<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>ENV32-C</td>
<td>Medium</td>
<td>Likely</td>
<td>Medium</td>
<td>P12</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>Axivion Bauhaus Suite</td>
<td>6.9.0</td>
<td>CertC-ENV32</td>
<td>Can detect violations of this rule. In particular, it ensures that all functions registered with <code>atexit()</code> do not call functions such as <code>exit()</code></td>
</tr>
<tr>
<td>Compass/ROSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>122 S 7</td>
<td>Enhanced enforcement</td>
</tr>
<tr>
<td>Parasoft C /C++test</td>
<td>10.4.2</td>
<td>CERT_C-ENV32-a</td>
<td>Properly define exit handlers</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2019b</td>
<td>CERT C: Rule ENV32-C</td>
<td>Checks for abnormal termination of exit handler (rule fully covered)</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>SIG30-C. Call only asynchronous-safe functions within signal handlers</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CWE 2.11</td>
<td>CWE-705, Incorrect Control Flow Scoping</td>
<td>2017-07-10: CERT: Rule subset of CWE</td>
</tr>
</tbody>
</table>

CERT-CWE Mapping Notes

Key here for mapping notes

CWE-705 and ENV32-C
CWE-705 = Union(ENV32-C, list) where list =

- Improper control flow besides a non-returning exit handler