SIG02-C. Avoid using signals to implement normal functionality

Avoid using signals to implement normal functionality. Signal handlers are severely limited in the actions they can perform in a portably secure manner. Their use should be reserved for abnormal events that can be serviced by little more than logging.

Noncompliant Code Example

This noncompliant code example uses signals as a means to pass state changes around in a multithreaded environment:

```c
/* THREAD 1 */
int do_work(void) {
    /* ... */
    kill(THR2_PID, SIGUSR1);
}

/* THREAD 2 */
volatile sig_atomic_t flag;

void sigusr1_handler(int signum) {
    flag = 1;
}

int wait_and_work(void) {
    flag = 0;
    while (!flag) {}
    /* ... */
}
```

However, using signals for such functionality often leads to nonportable or otherwise complicated solutions.

This code illustrates one thread using a signal to wake up a second thread. Using an architecture’s native thread library usually allows for a more sophisticated means of sending messages between threads.

Compliant Solution (POSIX)

A better solution, in this case, is to use condition variables. This code example uses a condition variable from the POSIX pthread library [IEEE Std 1003.1:2013]:

```c
/* THREAD 1 */
int do_work(void) {
    /* ... */
    pthread_kill(THR2_PID, SIGUSR1);
}

/* THREAD 2 */
volatile sig_atomic_t flag;

void sigusr1_handler(int signum) {
    flag = 1;
}

int wait_and_work(void) {
    flag = 0;
    while (!flag) {}
    /* ... */
```
#include <pthread.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

/* THREAD 1 */
int do_work(void) {
    int result;
    /* ... */
    if ((result = pthread_mutex_lock(&mut)) != 0) {
        /* Handle error condition */
    }
    if ((result = pthread_cond_signal(&cond, &mut)) != 0) {
        /* Handle error condition */
    }
    if ((result = pthread_mutex_unlock(&mut)) != 0) {
        /* Handle error condition */
    }
}

/* THREAD 2 */
int wait_and_work(void) {
    if ((result = pthread_mutex_lock(&mut)) != 0) {
        /* Handle error condition */
    }
    while (/* Condition does not hold */) {
        if ((result = pthread_cond_wait(&cond, &mut)) != 0) {
            /* Handle error condition */
        }
        /* ... */
    }
    if ((result = pthread_mutex_unlock(&mut)) != 0) {
        /* Handle error condition */
    }
    /* ... */
}

Compliant Solution (Windows)

This compliant solution uses a condition variable from the Win32 API [MSDN]:

#include <windows.h>

/*
 * Note that the CRITICAL_SECTION must be initialized with
 * InitializeCriticalSection, and the CONDITION_VARIABLE must
 * be initialized with InitializeConditionVariable prior to
 * using them.
 */
CRITICAL_SECTION CritSection;
CONDITION_VARIABLE ConditionVar;

/* THREAD 1 */
int do_work(void) {
    /* ... */
    WakeConditionVariable(&ConditionVar);
}

/* THREAD 2 */
int wait_and_work(void) {
    EnterCriticalSection(&CritSection);
    SleepConditionVariableCS(&ConditionVar, &CritSection, INFINITE);
    LeaveCriticalSection(&CritSection);
    /* ... */
}
### Noncompliant Code Example

This noncompliant code example is from a signal race vulnerability in WU-FTPD v2.4 [Greenman 1997]:

```c
void dologout(status) {
    if (logged_in) {
        (void) seteuid((uid_t)0);
        logwtmp(ttyline, "", "");
        /* ... */
    }
    _exit(status);
}

static void lostconn(int signo) {
    if (debug)
        syslog(LOG_DEBUG, "lost connection");
    dologout(-1);
}

static void myoob(signo) {
    if (!transflag)
        return;
    /* ... */
    if (strcmp(cp, "ABOR\r\n") == 0) {
        tmpline[0] = '\0';
        reply(426, "Transfer aborted. Data connection closed.");
        reply(226, "Abort successful");
        longjmp(urgcatch, 1);
    }
    /* ... */
}

signal(SIGPIPE, lostconn);
signal(SIGURG, myoob);
```

A serious exploit can occur if SIGURG is caught immediately following the elevation of privileges in `dologout()`. If the `longjmp()` in the SIGURG handler `myoob()` is invoked, execution returns to the main processing loop with an effective UID of 0.

Please note that this code sample violates SIG30-C. Call only asynchronous-safe functions within signal handlers and SIG31-C. Do not access shared objects in signal handlers.

An immediate fix is to ensure that `dologout()` cannot be interrupted by a SIGURG:

```c
void dologout(status) {
    /*
    * Prevent reception of SIGURG from resulting in a resumption
    * back to the main program loop.
    */
    transflag = 0;
    if (logged_in) {
        (void) seteuid((uid_t)0);
        logwtmp(ttyline, "", "");
        /* ... */
    }
    _exit(status);
}
```

A better solution is for `myoob()` to set a failure flag of type `volatile sig_atomic_t` that is periodically checked within the main loop:
volatile sig_atomic_t xfer_aborted = 0;
static void myoob(signo) {
    /* ... */
    if (strcmp(cp, "ABOR\r\n") == 0) {
        xfer_aborted = 1;
    }
    /* ... */
}

This solution, however, still violates SIG30-C. Call only asynchronous-safe functions within signal handlers and SIG31-C. Do not access shared objects in signal handlers.

Compliant Solution

A compliant solution (not shown) is to not use signals to signify lost connections and to design the system to have a robust error-handling mechanism (see ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy).

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>SIG02-C</td>
<td>High</td>
<td>Probable</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>CodeSonar</td>
<td>5.2p0</td>
<td>BADFUNC.SIGNAL</td>
<td>Use of signal</td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>44 S</td>
<td>Enhanced Enforcement</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_C-SIG02-a</td>
<td>The signal handling facilities of &lt;signal.h&gt; shall not be used</td>
</tr>
</tbody>
</table>

Related Vulnerabilities

Using signals to implement normal functionality frequently results in the violation of one or more secure coding rules for signal handling.

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

Related Guidelines

- SEI CERT C++ Coding Standard: VOID SIG02-CPP. Avoid using signals to implement normal functionality

Bibliography

[Dowd 2006] Chapter 13, "Synchronization and State"
[Greenman 1997]
[IEEE Std 1003.1:2013] XBD, Headers, <pthread>
[MSDN] Using Condition Variables