# CWE Detail – CWE-828

## Description

The product defines a signal handler that contains code sequences that are not asynchronous-safe, i.e., the functionality is not reentrant, or it can be interrupted.

## Extended Description

This can lead to an unexpected system state with a variety of potential consequences depending on context, including denial of service and code execution. Signal handlers are typically intended to interrupt normal functionality of a program, or even other signals, in order to notify the process of an event. When a signal handler uses global or static variables, or invokes functions that ultimately depend on such state or its associated metadata, then it could corrupt system state that is being used by normal functionality. This could subject the program to race conditions or other weaknesses that allow an attacker to cause the program state to be corrupted. While denial of service is frequently the consequence, in some cases this weakness could be leveraged for code execution. There are several different scenarios that introduce this issue: Invocation of non-reentrant functions from within the handler. One example is malloc(), which modifies internal global variables as it manages memory. Very few functions are actually reentrant. Code sequences (not necessarily function calls) contain non-atomic use of global variables, or associated metadata or structures, that can be accessed by other functionality of the program, including other signal handlers. Frequently, the same function is registered to handle multiple signals. The signal handler function is intended to run at most one time, but instead it can be invoked multiple times. This could happen by repeated delivery of the same signal, or by delivery of different signals that have the same handler function (CWE-831). Note that in some environments or contexts, it might be possible for the signal handler to be interrupted itself. If both a signal handler and the normal behavior of the product have to operate on the same set of state variables, and a signal is received in the middle of the normal execution's modifications of those variables, the variables may be in an incorrect or corrupt state during signal handler execution, and possibly still incorrect or corrupt upon return.

## Threat-Mapped Scoring

Score: 1.9

Priority: P3 - Important (Medium)

## Observed Examples (CVEs)

**•** CVE-2008-4109: Signal handler uses functions that ultimately call the unsafe syslog/malloc/s\*printf, leading to denial of service via multiple login attempts

**•** CVE-2006-5051: Chain: Signal handler contains too much functionality (CWE-828), introducing a race condition (CWE-362) that leads to a double free (CWE-415).

**•** CVE-2001-1349: unsafe calls to library functions from signal handler

**•** CVE-2004-0794: SIGURG can be used to remotely interrupt signal handler; other variants exist.

**•** CVE-2004-2259: SIGCHLD signal to FTP server can cause crash under heavy load while executing non-reentrant functions like malloc/free.

**•** CVE-2002-1563: SIGCHLD not blocked in a daemon loop while counter is modified, causing counter to get out of sync.

## Common Consequences

**•** Impact: DoS: Crash, Exit, or Restart, Execute Unauthorized Code or Commands — Notes: The most common consequence will be a corruption of the state of the product, possibly leading to a crash or exit. However, if the signal handler is operating on state variables for security relevant libraries or protection mechanisms, the consequences can be far more severe, including protection mechanism bypass, privilege escalation, or information exposure.

## Potential Mitigations

**•** Implementation: Eliminate the usage of non-reentrant functionality inside of signal handlers. This includes replacing all non-reentrant library calls with reentrant calls. Note: This will not always be possible and may require large portions of the product to be rewritten or even redesigned. Sometimes reentrant-safe library alternatives will not be available. Sometimes non-reentrant interaction between the state of the system and the signal handler will be required by design. (Effectiveness: High)

**•** Implementation: Where non-reentrant functionality must be leveraged within a signal handler, be sure to block or mask signals appropriately. This includes blocking other signals within the signal handler itself that may also leverage the functionality. It also includes blocking all signals reliant upon the functionality when it is being accessed or modified by the normal behaviors of the product. (Effectiveness: N/A)

## Demonstrative Examples

**•** The handler function uses global state (globalVar and logMessage), and it can be called by both the SIGHUP and SIGTERM signals. An attack scenario might follow these lines:

**•** However, the following sequence of events may result in a double-free (CWE-415):