# CWE Detail – CWE-364

## Description

The product uses a signal handler that introduces a race condition.

## Extended Description

Race conditions frequently occur in signal handlers, since signal handlers support asynchronous actions. These race conditions have a variety of root causes and symptoms. Attackers may be able to exploit a signal handler race condition to cause the product state to be corrupted, possibly leading to a denial of service or even code execution. These issues occur when non-reentrant functions, or state-sensitive actions occur in the signal handler, where they may be called at any time. These behaviors can violate assumptions being made by the "regular" code that is interrupted, or by other signal handlers that may also be invoked. If these functions are called at an inopportune moment - such as while a non-reentrant function is already running - memory corruption could occur that may be exploitable for code execution. Another signal race condition commonly found occurs when free is called within a signal handler, resulting in a double free and therefore a write-what-where condition. Even if a given pointer is set to NULL after it has been freed, a race condition still exists between the time the memory was freed and the pointer was set to NULL. This is especially problematic if the same signal handler has been set for more than one signal -- since it means that the signal handler itself may be reentered. There are several known behaviors related to signal handlers that have received the label of "signal handler race condition": Shared state (e.g. global data or static variables) that are accessible to both a signal handler and "regular" code Shared state between a signal handler and other signal handlers Use of non-reentrant functionality within a signal handler - which generally implies that shared state is being used. For example, malloc() and free() are non-reentrant because they may use global or static data structures for managing memory, and they are indirectly used by innocent-seeming functions such as syslog(); these functions could be exploited for memory corruption and, possibly, code execution. Association of the same signal handler function with multiple signals - which might imply shared state, since the same code and resources are accessed. For example, this can be a source of double-free and use-after-free weaknesses. Use of setjmp and longjmp, or other mechanisms that prevent a signal handler from returning control back to the original functionality While not technically a race condition, some signal handlers are designed to be called at most once, and being called more than once can introduce security problems, even when there are not any concurrent calls to the signal handler. This can be a source of double-free and use-after-free weaknesses. Signal handler vulnerabilities are often classified based on the absence of a specific protection mechanism, although this style of classification is discouraged in CWE because programmers often have a choice of several different mechanisms for addressing the weakness. Such protection mechanisms may preserve exclusivity of access to the shared resource, and behavioral atomicity for the relevant code: Avoiding shared state Using synchronization in the signal handler Using synchronization in the regular code Disabling or masking other signals, which provides atomicity (which effectively ensures exclusivity)

## Threat-Mapped Scoring

Score: 1.9

Priority: P3 - Important (Medium)

## Observed Examples (CVEs)

**•** CVE-1999-0035: Signal handler does not disable other signal handlers, allowing it to be interrupted, causing other functionality to access files/etc. with raised privileges

**•** CVE-2001-0905: Attacker can send a signal while another signal handler is already running, leading to crash or execution with root privileges

**•** 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.

## Modes of Introduction

**•** Implementation: N/A

## Common Consequences

**•** Impact: Modify Application Data, Modify Memory, DoS: Crash, Exit, or Restart, Execute Unauthorized Code or Commands — Notes: It may be possible to cause data corruption and possibly execute arbitrary code by modifying global variables or data structures at unexpected times, violating the assumptions of code that uses this global data.

**•** Impact: Gain Privileges or Assume Identity — Notes: If a signal handler interrupts code that is executing with privileges, it may be possible that the signal handler will also be executed with elevated privileges, possibly making subsequent exploits more severe.

## Potential Mitigations

**•** Requirements: Use a language that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. (Effectiveness: N/A)

**•** Architecture and Design: Design signal handlers to only set flags, rather than perform complex functionality. These flags can then be checked and acted upon within the main program loop. (Effectiveness: N/A)

**•** Implementation: Only use reentrant functions within signal handlers. Also, use validation to ensure that state is consistent while performing asynchronous actions that affect the state of execution. (Effectiveness: N/A)

## Applicable Platforms

**•** C (Class: None, Prevalence: Sometimes)

**•** C++ (Class: None, Prevalence: Sometimes)

## 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):