# CWE Detail – CWE-762

## Description

The product attempts to return a memory resource to the system, but it calls a release function that is not compatible with the function that was originally used to allocate that resource.

## Extended Description

This weakness can be generally described as mismatching memory management routines, such as: The memory was allocated on the stack (automatically), but it was deallocated using the memory management routine free() (CWE-590), which is intended for explicitly allocated heap memory. The memory was allocated explicitly using one set of memory management functions, and deallocated using a different set. For example, memory might be allocated with malloc() in C++ instead of the new operator, and then deallocated with the delete operator. When the memory management functions are mismatched, the consequences may be as severe as code execution, memory corruption, or program crash. Consequences and ease of exploit will vary depending on the implementation of the routines and the object being managed.

## Threat-Mapped Scoring

Score: 1.8

Priority: P4 - Informational (Low)

## Modes of Introduction

**•** Implementation: N/A

## Common Consequences

**•** Impact: Modify Memory, DoS: Crash, Exit, or Restart, Execute Unauthorized Code or Commands — Notes:

## Potential Mitigations

**•** Implementation: Only call matching memory management functions. Do not mix and match routines. For example, when you allocate a buffer with malloc(), dispose of the original pointer with free(). (Effectiveness: N/A)

**•** Implementation: Choose a language or tool that provides automatic memory management, or makes manual memory management less error-prone. For example, glibc in Linux provides protection against free of invalid pointers. When using Xcode to target OS X or iOS, enable automatic reference counting (ARC) [REF-391]. To help correctly and consistently manage memory when programming in C++, consider using a smart pointer class such as std::auto\_ptr (defined by ISO/IEC ISO/IEC 14882:2003), std::shared\_ptr and std::unique\_ptr (specified by an upcoming revision of the C++ standard, informally referred to as C++ 1x), or equivalent solutions such as Boost. (Effectiveness: N/A)

**•** Architecture and Design: Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. For example, glibc in Linux provides protection against free of invalid pointers. (Effectiveness: N/A)

**•** Architecture and Design: Use a language that provides abstractions for memory allocation and deallocation. (Effectiveness: N/A)

**•** Testing: Use a tool that dynamically detects memory management problems, such as valgrind. (Effectiveness: N/A)

## Applicable Platforms

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

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

## Demonstrative Examples

**•** Instead, the programmer should have either created the object with one of the malloc family functions, or else deleted the object with the delete operator.

**•** N/A

**•** N/A

## Notes

**•** Applicable Platform: This weakness is possible in any programming language that allows manual management of memory.