# CWE Detail – CWE-806

## Description

The product uses the size of a source buffer when reading from or writing to a destination buffer, which may cause it to access memory that is outside of the bounds of the buffer.

## Extended Description

When the size of the destination is smaller than the size of the source, a buffer overflow could occur.

## Threat-Mapped Scoring

Score: 1.9

Priority: P3 - Important (Medium)

## Modes of Introduction

**•** Implementation: N/A

## Common Consequences

**•** Impact: Modify Memory, DoS: Crash, Exit, or Restart, DoS: Resource Consumption (CPU) — Notes: Buffer overflows generally lead to crashes. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.

**•** Impact: Read Memory, Modify Memory, Execute Unauthorized Code or Commands — Notes: Buffer overflows often can be used to execute arbitrary code, which is usually outside the scope of a program's implicit security policy.

**•** Impact: Bypass Protection Mechanism — Notes: When the consequence is arbitrary code execution, this can often be used to subvert any other security service.

## Potential Mitigations

**•** Architecture and Design: Use an abstraction library to abstract away risky APIs. Examples include the Safe C String Library (SafeStr) by Viega, and the Strsafe.h library from Microsoft. This is not a complete solution, since many buffer overflows are not related to strings. (Effectiveness: N/A)

**•** Operation: Use automatic buffer overflow detection mechanisms that are offered by certain compilers or compiler extensions. Examples include: the Microsoft Visual Studio /GS flag, Fedora/Red Hat FORTIFY\_SOURCE GCC flag, StackGuard, and ProPolice, which provide various mechanisms including canary-based detection and range/index checking. D3-SFCV (Stack Frame Canary Validation) from D3FEND [REF-1334] discusses canary-based detection in detail. (Effectiveness: Defense in Depth)

**•** Implementation: Programmers should adhere to the following rules when allocating and managing their applications memory: Double check that your buffer is as large as you specify. When using functions that accept a number of bytes to copy, such as strncpy(), be aware that if the destination buffer size is equal to the source buffer size, it may not NULL-terminate the string. Check buffer boundaries if calling this function in a loop and make sure there is no danger of writing past the allocated space. Truncate all input strings to a reasonable length before passing them to the copy and concatenation functions. (Effectiveness: N/A)

**•** Operation: Run or compile the software using features or extensions that randomly arrange the positions of a program's executable and libraries in memory. Because this makes the addresses unpredictable, it can prevent an attacker from reliably jumping to exploitable code. Examples include Address Space Layout Randomization (ASLR) [REF-58] [REF-60] and Position-Independent Executables (PIE) [REF-64]. Imported modules may be similarly realigned if their default memory addresses conflict with other modules, in a process known as "rebasing" (for Windows) and "prelinking" (for Linux) [REF-1332] using randomly generated addresses. ASLR for libraries cannot be used in conjunction with prelink since it would require relocating the libraries at run-time, defeating the whole purpose of prelinking. For more information on these techniques see D3-SAOR (Segment Address Offset Randomization) from D3FEND [REF-1335]. (Effectiveness: Defense in Depth)

**•** Operation: Use a CPU and operating system that offers Data Execution Protection (using hardware NX or XD bits) or the equivalent techniques that simulate this feature in software, such as PaX [REF-60] [REF-61]. These techniques ensure that any instruction executed is exclusively at a memory address that is part of the code segment. For more information on these techniques see D3-PSEP (Process Segment Execution Prevention) from D3FEND [REF-1336]. (Effectiveness: Defense in Depth)

**•** Build and Compilation: Most mitigating technologies at the compiler or OS level to date address only a subset of buffer overflow problems and rarely provide complete protection against even that subset. It is good practice to implement strategies to increase the workload of an attacker, such as leaving the attacker to guess an unknown value that changes every program execution. (Effectiveness: N/A)

## Applicable Platforms

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

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

## Demonstrative Examples

**•** However, in the call to strncpy the source character string is used within the sizeof call to determine the number of characters to copy. This will create a buffer overflow as the size of the source character string is greater than the dest character string. The dest character string should be used within the sizeof call to ensure that the correct number of characters are copied, as shown below.

**•** However, in this case the string copy method, strncpy, mistakenly uses the length method argument to determine the number of characters to copy rather than using the size of the local character string, buf. This can lead to a buffer overflow if the number of characters contained in character string pointed to by filename is larger then the number of characters allowed for the local character string. The string copy method should use the buf character string within a sizeof call to ensure that only characters up to the size of the buf array are copied to avoid a buffer overflow, as shown below.