# CWE Detail – CWE-787

## Description

The product writes data past the end, or before the beginning, of the intended buffer.

## Extended Description

N/A

## Threat-Mapped Scoring

Score: 0.0

Priority: Unclassified

## Observed Examples (CVEs)

**•** CVE-2025-27363: Font rendering library does not properly
 handle assigning a signed short value to an unsigned
 long (CWE-195), leading to an integer wraparound
 (CWE-190), causing too small of a buffer (CWE-131),
 leading to an out-of-bounds write
 (CWE-787). (KEV)

**•** CVE-2023-1017: The reference implementation code for a Trusted Platform Module does not implement length checks on data, allowing for an attacker to write 2 bytes past the end of a buffer.

**•** CVE-2021-21220: Chain: insufficient input validation (CWE-20) in browser allows heap corruption (CWE-787), as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2021-28664: GPU kernel driver allows memory corruption because a user can obtain read/write access to read-only pages, as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2020-17087: Chain: integer truncation (CWE-197) causes small buffer allocation (CWE-131) leading to out-of-bounds write (CWE-787) in kernel pool, as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2020-1054: Out-of-bounds write in kernel-mode driver, as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2020-0041: Escape from browser sandbox using out-of-bounds write due to incorrect bounds check, as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2020-0968: Memory corruption in web browser scripting engine, as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2020-0022: chain: mobile phone Bluetooth implementation does not include offset when calculating packet length (CWE-682), leading to out-of-bounds write (CWE-787)

**•** CVE-2019-1010006: Chain: compiler optimization (CWE-733) removes or modifies code used to detect integer overflow (CWE-190), allowing out-of-bounds write (CWE-787).

**•** CVE-2009-1532: malformed inputs cause accesses of uninitialized or previously-deleted objects, leading to memory corruption

**•** CVE-2009-0269: chain: -1 value from a function call was intended to indicate an error, but is used as an array index instead.

**•** CVE-2002-2227: Unchecked length of SSLv2 challenge value leads to buffer underflow.

**•** CVE-2007-4580: Buffer underflow from a small size value with a large buffer (length parameter inconsistency, CWE-130)

**•** CVE-2007-4268: Chain: integer signedness error (CWE-195) passes signed comparison, leading to heap overflow (CWE-122)

**•** CVE-2009-2550: Classic stack-based buffer overflow in media player using a long entry in a playlist

**•** CVE-2009-2403: Heap-based buffer overflow in media player using a long entry in a playlist

## Modes of Introduction

**•** Implementation: N/A

## Common Consequences

**•** Impact: Modify Memory, Execute Unauthorized Code or Commands — Notes: Write operations could cause memory corruption. In some cases, an adversary can modify control data such as return addresses in order to execute unexpected code.

**•** Impact: DoS: Crash, Exit, or Restart — Notes: Attempting to access out-of-range, invalid, or unauthorized memory could cause the product to crash.

**•** Impact: Unexpected State — Notes: Subsequent write operations can produce undefined or unexpected results.

## Potential Mitigations

**•** Requirements: Use a language that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. For example, many languages that perform their own memory management, such as Java and Perl, are not subject to buffer overflows. Other languages, such as Ada and C#, typically provide overflow protection, but the protection can be disabled by the programmer. Be wary that a language's interface to native code may still be subject to overflows, even if the language itself is theoretically safe. (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. Examples include the Safe C String Library (SafeStr) by Messier and Viega [REF-57], and the Strsafe.h library from Microsoft [REF-56]. These libraries provide safer versions of overflow-prone string-handling functions. (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: Consider adhering to the following rules when allocating and managing an application's memory: Double check that the buffer is as large as specified. 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 accessing the buffer in a loop and make sure there is no danger of writing past the allocated space. If necessary, 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)

**•** Implementation: Replace unbounded copy functions with analogous functions that support length arguments, such as strcpy with strncpy. Create these if they are not available. (Effectiveness: Moderate)

## Applicable Platforms

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

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

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

## Demonstrative Examples

**•** Since the array is only allocated to hold three elements, the valid indices are 0 to 2; so, the assignment to id\_sequence[3] is out of bounds.

**•** If returnChunkSize() happens to encounter an error it will return -1. Notice that the return value is not checked before the memcpy operation (CWE-252), so -1 can be passed as the size argument to memcpy() (CWE-805). Because memcpy() assumes that the value is unsigned, it will be interpreted as MAXINT-1 (CWE-195), and therefore will copy far more memory than is likely available to the destination buffer (CWE-787, CWE-788).

**•** This function allocates a buffer of 64 bytes to store the hostname. However, there is no guarantee that the hostname will not be larger than 64 bytes. If an attacker specifies an address which resolves to a very large hostname, then the function may overwrite sensitive data or even relinquish control flow to the attacker.

**•** The programmer attempts to encode the ampersand character in the user-controlled string. However, the length of the string is validated before the encoding procedure is applied. Furthermore, the programmer assumes encoding expansion will only expand a given character by a factor of 4, while the encoding of the ampersand expands by 5. As a result, when the encoding procedure expands the string it is possible to overflow the destination buffer if the attacker provides a string of many ampersands.

**•** However, this function can cause a buffer underwrite if the input character string contains all whitespace. On some systems the while statement will move backwards past the beginning of a character string and will call the isspace() function on an address outside of the bounds of the local buffer.

**•** However, this code contains an off-by-one calculation error (CWE-193). It allocates exactly enough space to contain the specified number of widgets, but it does not include the space for the NULL pointer. As a result, the allocated buffer is smaller than it is supposed to be (CWE-131). So if the user ever requests MAX\_NUM\_WIDGETS, there is an out-of-bounds write (CWE-787) when the NULL is assigned. Depending on the environment and compilation settings, this could cause memory corruption.

**•** In the case where the substring is not found in destBuf, strstr() will return NULL, causing the pointer arithmetic to be undefined, potentially setting the value of idx to a negative number. If idx is negative, this will result in a buffer underwrite of destBuf.