# CWE Detail – CWE-131

## Description

The product does not correctly calculate the size to be used when allocating a buffer, which could lead to a buffer overflow.

## Extended Description

N/A

## Threat-Mapped Scoring

Score: 1.5

Priority: P4 - Informational (Low)

## 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-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-2004-1363: substitution overflow: buffer overflow using environment variables that are expanded after the length check is performed

**•** CVE-2004-0747: substitution overflow: buffer overflow using expansion of environment variables

**•** CVE-2005-2103: substitution overflow: buffer overflow using a large number of substitution strings

**•** CVE-2005-3120: transformation overflow: product adds extra escape characters to incoming data, but does not account for them in the buffer length

**•** CVE-2003-0899: transformation overflow: buffer overflow when expanding ">" to "&gt;", etc.

**•** CVE-2001-0334: expansion overflow: buffer overflow using wildcards

**•** CVE-2001-0248: expansion overflow: long pathname + glob = overflow

**•** CVE-2001-0249: expansion overflow: long pathname + glob = overflow

**•** CVE-2002-0184: special characters in argument are not properly expanded

**•** CVE-2004-0434: small length value leads to heap overflow

**•** CVE-2002-1347: multiple variants

**•** CVE-2005-0490: needs closer investigation, but probably expansion-based

**•** CVE-2004-0940: needs closer investigation, but probably expansion-based

**•** CVE-2008-0599: Chain: Language interpreter calculates wrong buffer size (CWE-131) by using "size = ptr ? X : Y" instead of "size = (ptr ? X : Y)" expression.

## Related Attack Patterns (CAPEC)

* CAPEC-100
* CAPEC-47

## Modes of Introduction

**•** Implementation: N/A

## Common Consequences

**•** Impact: DoS: Crash, Exit, or Restart, Execute Unauthorized Code or Commands, Read Memory, Modify Memory — Notes: If the incorrect calculation is used in the context of memory allocation, then the software may create a buffer that is smaller or larger than expected. If the allocated buffer is smaller than expected, this could lead to an out-of-bounds read or write (CWE-119), possibly causing a crash, allowing arbitrary code execution, or exposing sensitive data.

## Potential Mitigations

**•** Implementation: When allocating a buffer for the purpose of transforming, converting, or encoding an input, allocate enough memory to handle the largest possible encoding. For example, in a routine that converts "&" characters to "&amp;" for HTML entity encoding, the output buffer needs to be at least 5 times as large as the input buffer. (Effectiveness: N/A)

**•** Implementation: Understand the programming language's underlying representation and how it interacts with numeric calculation (CWE-681). Pay close attention to byte size discrepancies, precision, signed/unsigned distinctions, truncation, conversion and casting between types, "not-a-number" calculations, and how the language handles numbers that are too large or too small for its underlying representation. [REF-7] Also be careful to account for 32-bit, 64-bit, and other potential differences that may affect the numeric representation. (Effectiveness: N/A)

**•** Implementation: Perform input validation on any numeric input by ensuring that it is within the expected range. Enforce that the input meets both the minimum and maximum requirements for the expected range. (Effectiveness: N/A)

**•** Architecture and Design: For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server. (Effectiveness: N/A)

**•** Implementation: When processing structured incoming data containing a size field followed by raw data, identify and resolve any inconsistencies between the size field and the actual size of the data (CWE-130). (Effectiveness: N/A)

**•** Implementation: When allocating memory that uses sentinels to mark the end of a data structure - such as NUL bytes in strings - make sure you also include the sentinel in your calculation of the total amount of memory that must be allocated. (Effectiveness: N/A)

**•** 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)

**•** Implementation: Use sizeof() on the appropriate data type to avoid CWE-467. (Effectiveness: N/A)

**•** Implementation: Use the appropriate type for the desired action. For example, in C/C++, only use unsigned types for values that could never be negative, such as height, width, or other numbers related to quantity. This will simplify validation and will reduce surprises related to unexpected casting. (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. Use libraries or frameworks that make it easier to handle numbers without unexpected consequences, or buffer allocation routines that automatically track buffer size. Examples include safe integer handling packages such as SafeInt (C++) or IntegerLib (C or C++). [REF-106] (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)

**•** 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: Examine compiler warnings closely and eliminate problems with potential security implications, such as signed / unsigned mismatch in memory operations, or use of uninitialized variables. Even if the weakness is rarely exploitable, a single failure may lead to the compromise of the entire system. (Effectiveness: N/A)

**•** Architecture and Design: Run your code using the lowest privileges that are required to accomplish the necessary tasks [REF-76]. If possible, create isolated accounts with limited privileges that are only used for a single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations. (Effectiveness: N/A)

**•** Architecture and Design: Run the code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which files can be accessed in a particular directory or which commands can be executed by the software. OS-level examples include the Unix chroot jail, AppArmor, and SELinux. In general, managed code may provide some protection. For example, java.io.FilePermission in the Java SecurityManager allows the software to specify restrictions on file operations. This may not be a feasible solution, and it only limits the impact to the operating system; the rest of the application may still be subject to compromise. Be careful to avoid CWE-243 and other weaknesses related to jails. (Effectiveness: Limited)

## Applicable Platforms

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

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

## Demonstrative Examples

**•** 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.

**•** This code intends to allocate a table of size num\_imgs, however as num\_imgs grows large, the calculation determining the size of the list will eventually overflow (CWE-190). This will result in a very small list to be allocated instead. If the subsequent code operates on the list as if it were num\_imgs long, it may result in many types of out-of-bounds problems (CWE-119).

**•** 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.

**•** The code performs a check to make sure that the packet does not contain too many headers. However, numHeaders is defined as a signed int, so it could be negative. If the incoming packet specifies a value such as -3, then the malloc calculation will generate a negative number (say, -300 if each header can be a maximum of 100 bytes). When this result is provided to malloc(), it is first converted to a size\_t type. This conversion then produces a large value such as 4294966996, which may cause malloc() to fail or to allocate an extremely large amount of memory (CWE-195). With the appropriate negative numbers, an attacker could trick malloc() into using a very small positive number, which then allocates a buffer that is much smaller than expected, potentially leading to a buffer overflow.

**•** The problem with the code above is the value of the size parameter used during the malloc() call. It uses a value of '3' which by definition results in a buffer of three bytes to be created. However the intention was to create a buffer that holds three ints, and in C, each int requires 4 bytes worth of memory, so an array of 12 bytes is needed, 4 bytes for each int. Executing the above code could result in a buffer overflow as 12 bytes of data is being saved into 3 bytes worth of allocated space. The overflow would occur during the assignment of id\_sequence[0] and would continue with the assignment of id\_sequence[1] and id\_sequence[2].

## Notes

**•** Maintenance: This is a broad category. Some examples include: simple math errors, incorrectly updating parallel counters, not accounting for size differences when "transforming" one input to another format (e.g. URL canonicalization or other transformation that can generate a result that's larger than the original input, i.e. "expansion"). This level of detail is rarely available in public reports, so it is difficult to find good examples.

**•** Maintenance: This weakness may be a composite or a chain. It also may contain layering or perspective differences. This issue may be associated with many different types of incorrect calculations (CWE-682), although the integer overflow (CWE-190) is probably the most prevalent. This can be primary to resource consumption problems (CWE-400), including uncontrolled memory allocation (CWE-789). However, its relationship with out-of-bounds buffer access (CWE-119) must also be considered.