# CWE Detail – CWE-195

## Description

The product uses a signed primitive and performs a cast to an unsigned primitive, which can produce an unexpected value if the value of the signed primitive can not be represented using an unsigned primitive.

## Extended Description

It is dangerous to rely on implicit casts between signed and unsigned numbers because the result can take on an unexpected value and violate assumptions made by the program. Often, functions will return negative values to indicate a failure. When the result of a function is to be used as a size parameter, using these negative return values can have unexpected results. For example, if negative size values are passed to the standard memory copy or allocation functions they will be implicitly cast to a large unsigned value. This may lead to an exploitable buffer overflow or underflow condition.

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

## Modes of Introduction

**•** Implementation: N/A

## Common Consequences

**•** Impact: Unexpected State — Notes: Conversion between signed and unsigned values can lead to a variety of errors, but from a security standpoint is most commonly associated with integer overflow and buffer overflow vulnerabilities.

## Applicable Platforms

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

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

## Demonstrative Examples

**•** If the error condition in the code above is met, then the return value of readdata() will be 4,294,967,295 on a system that uses 32-bit integers.

**•** If the return value of accessmainframe() is -1, then the return value of readdata() will be 4,294,967,295 on a system that uses 32-bit integers.

**•** 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 programmer has set an upper bound on the structure size: if it is larger than 512, the input will not be processed. The problem is that len is a signed short, so the check against the maximum structure length is done with signed values, but len is converted to an unsigned integer for the call to memcpy() and the negative bit will be extended to result in a huge value for the unsigned integer. If len is negative, then it will appear that the structure has an appropriate size (the if branch will be taken), but the amount of memory copied by memcpy() will be quite large, and the attacker will be able to overflow the stack with data in strm.

**•** 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).

**•** The buffer length ends up being -1, resulting in a blown out stack. The space character after the colon is included in the function calculation, but not in the caller's calculation. This, unfortunately, is not usually so obvious but exists in an obtuse series of calculations.