# CWE Detail – CWE-122

## Description

A heap overflow condition is a buffer overflow, where the buffer that can be overwritten is allocated in the heap portion of memory, generally meaning that the buffer was allocated using a routine such as malloc().

## Extended Description

N/A

## Threat-Mapped Scoring

Score: 1.5

Priority: P4 - Informational (Low)

## Observed Examples (CVEs)

**•** CVE-2021-43537: Chain: in a web browser, an unsigned 64-bit integer is forcibly cast to a 32-bit integer (CWE-681) and potentially leading to an integer overflow (CWE-190). If an integer overflow occurs, this can cause heap memory corruption (CWE-122)

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

**•** CVE-2009-2523: Chain: product does not handle when an input string is not NULL terminated (CWE-170), leading to buffer over-read (CWE-125) or heap-based buffer overflow (CWE-122).

**•** CVE-2021-29529: Chain: machine-learning product can have a heap-based
 buffer overflow (CWE-122) when some integer-oriented bounds are
 calculated by using ceiling() and floor() on floating point values
 (CWE-1339)

**•** CVE-2010-1866: Chain: integer overflow (CWE-190) causes a negative signed value, which later bypasses a maximum-only check (CWE-839), leading to heap-based buffer overflow (CWE-122).

## Related Attack Patterns (CAPEC)

* CAPEC-92

## Modes of Introduction

**•** Implementation: N/A

## Common Consequences

**•** Impact: DoS: Crash, Exit, or Restart, DoS: Resource Consumption (CPU), DoS: Resource Consumption (Memory) — 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: Execute Unauthorized Code or Commands, Bypass Protection Mechanism, Modify Memory — Notes: Buffer overflows often can be used to execute arbitrary code, which is usually outside the scope of a program's implicit security policy. Besides important user data, heap-based overflows can be used to overwrite function pointers that may be living in memory, pointing it to the attacker's code. Even in applications that do not explicitly use function pointers, the run-time will usually leave many in memory. For example, object methods in C++ are generally implemented using function pointers. Even in C programs, there is often a global offset table used by the underlying runtime.

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

## Potential Mitigations

**•** N/A: Pre-design: Use a language or compiler that performs automatic bounds checking. (Effectiveness: N/A)

**•** Architecture and Design: Use an abstraction library to abstract away risky APIs. Not a complete solution. (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)

**•** Implementation: Implement and perform bounds checking on input. (Effectiveness: N/A)

**•** Implementation: Do not use dangerous functions such as gets. Look for their safe equivalent, which checks for the boundary. (Effectiveness: N/A)

**•** Operation: Use OS-level preventative functionality. This is not a complete solution, but it provides some defense in depth. (Effectiveness: N/A)

## Applicable Platforms

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

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

## Demonstrative Examples

**•** The buffer is allocated heap memory with a fixed size, but there is no guarantee the string in argv[1] will not exceed this size and cause an overflow.

**•** 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.

## Notes

**•** Relationship: Heap-based buffer overflows are usually just as dangerous as stack-based buffer overflows.