# CWE Detail – CWE-119

## Description

The product performs operations on a memory buffer, but it reads from or writes to a memory location outside the buffer's intended boundary. This may result in read or write operations on unexpected memory locations that could be linked to other variables, data structures, or internal program data.

## Extended Description

N/A

## Threat-Mapped Scoring

Score: 0.0

Priority: Unclassified

## Observed Examples (CVEs)

**•** CVE-2021-22991: Incorrect URI normalization in application traffic product leads to buffer overflow, as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2020-29557: Buffer overflow in Wi-Fi router web interface, as exploited in the wild per CISA KEV. (KEV)

**•** 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

**•** CVE-2009-0689: large precision value in a format string triggers overflow

**•** CVE-2009-0690: negative offset value leads to out-of-bounds read

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

**•** CVE-2009-1528: chain: lack of synchronization leads to memory corruption

**•** 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-2009-0558: attacker-controlled array index leads to code execution

**•** 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-2009-0566: chain: incorrect calculations lead to incorrect pointer dereference and memory corruption

**•** CVE-2009-1350: product accepts crafted messages that lead to a dereference of an arbitrary pointer

**•** CVE-2009-0191: chain: malformed input causes dereference of uninitialized memory

**•** CVE-2008-4113: OS kernel trusts userland-supplied length value, allowing reading of sensitive information

**•** CVE-2005-1513: Chain: integer overflow in securely-coded mail program leads to buffer overflow. In 2005, this was regarded as unrealistic to exploit, but in 2020, it was rediscovered to be easier to exploit due to evolutions of the technology.

**•** CVE-2003-0542: buffer overflow involving a regular expression with a large number of captures

**•** CVE-2017-1000121: chain: unchecked message size metadata allows integer overflow (CWE-190) leading to buffer overflow (CWE-119).

## Related Attack Patterns (CAPEC)

* CAPEC-10
* CAPEC-100
* CAPEC-123
* CAPEC-14
* CAPEC-24
* CAPEC-42
* CAPEC-44
* CAPEC-45
* CAPEC-46
* CAPEC-47
* CAPEC-8
* CAPEC-9

## Modes of Introduction

**•** Implementation: N/A

## Common Consequences

**•** Impact: Execute Unauthorized Code or Commands, Modify Memory — Notes: If the memory accessible by the attacker can be effectively controlled, it may be possible to execute arbitrary code, as with a standard buffer overflow. If the attacker can overwrite a pointer's worth of memory (usually 32 or 64 bits), they can alter the intended control flow by redirecting a function pointer to their own malicious code. Even when the attacker can only modify a single byte arbitrary code execution can be possible. Sometimes this is because the same problem can be exploited repeatedly to the same effect. Other times it is because the attacker can overwrite security-critical application-specific data -- such as a flag indicating whether the user is an administrator.

**•** Impact: Read Memory, DoS: Crash, Exit, or Restart, DoS: Resource Consumption (CPU), DoS: Resource Consumption (Memory) — Notes: Out of bounds memory access will very likely result in the corruption of relevant memory, and perhaps instructions, possibly leading to a crash. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.

**•** Impact: Read Memory — Notes: In the case of an out-of-bounds read, the attacker may have access to sensitive information. If the sensitive information contains system details, such as the current buffer's position in memory, this knowledge can be used to craft further attacks, possibly with more severe consequences.

## 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

**•** 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.

**•** The programmer allows the user to specify which element in the list to select, however an attacker can provide an out-of-bounds offset, resulting in a buffer over-read (CWE-126).

**•** However, this method only verifies that the given array index is less than the maximum length of the array but does not check for the minimum value (CWE-839). This will allow a negative value to be accepted as the input array index, which will result in a out of bounds read (CWE-125) and may allow access to sensitive memory. The input array index should be checked to verify that is within the maximum and minimum range required for the array (CWE-129). In this example the if statement should be modified to include a minimum range check, as shown below.

**•** N/A

## Notes

**•** Applicable Platform: It is possible in any programming languages without memory management support to attempt an operation outside of the bounds of a memory buffer, but the consequences will vary widely depending on the language, platform, and chip architecture.