# CWE Detail – CWE-121

## Description

A stack-based buffer overflow condition is a condition where the buffer being overwritten is allocated on the stack (i.e., is a local variable or, rarely, a parameter to a function).

## Extended Description

N/A

## Threat-Mapped Scoring

Score: 1.5

Priority: P4 - Informational (Low)

## Observed Examples (CVEs)

**•** CVE-2021-35395: Stack-based buffer overflows in SFK for wifi chipset used for IoT/embedded devices, as exploited in the wild per CISA KEV. (KEV)

## Modes of Introduction

**•** Implementation: N/A

## Common Consequences

**•** Impact: Modify Memory, 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: Modify Memory, Execute Unauthorized Code or Commands, Bypass Protection Mechanism — Notes: Buffer overflows often can be used to execute arbitrary code, which is usually outside the scope of a program's implicit security policy.

**•** Impact: Modify Memory, 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

**•** 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)

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

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

**•** Implementation: Do not use dangerous functions such as gets. Use safer, equivalent functions which check for boundary errors. (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)

## Applicable Platforms

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

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

## Demonstrative Examples

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

**•** 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.

## Notes

**•** Other: Stack-based buffer overflows can instantiate in return address overwrites, stack pointer overwrites or frame pointer overwrites. They can also be considered function pointer overwrites, array indexer overwrites or write-what-where condition, etc.