# CWE Detail – CWE-1339

## Description

The product processes a real number with an implementation in which the number's representation does not preserve required accuracy and precision in its fractional part, causing an incorrect result.

## Extended Description

When a security decision or calculation requires highly precise, accurate numbers such as financial calculations or prices, then small variations in the number could be exploited by an attacker. There are multiple ways to store the fractional part of a real number in a computer. In all of these cases, there is a limit to the accuracy of recording a fraction. If the fraction can be represented in a fixed number of digits (binary or decimal), there might not be enough digits assigned to represent the number. In other cases the number cannot be represented in a fixed number of digits due to repeating in decimal or binary notation (e.g. 0.333333...) or due to a transcendental number such as Π or √2. Rounding of numbers can lead to situations where the computer results do not adequately match the result of sufficiently accurate math.

## Threat-Mapped Scoring

Score: 0.0

Priority: Unclassified

## Observed Examples (CVEs)

**•** CVE-2018-16069: Chain: series of floating-point precision errors  
 (CWE-1339) in a web browser rendering engine causes out-of-bounds read  
 (CWE-125), giving access to cross-origin data

**•** CVE-2017-7619: Chain: rounding error in floating-point calculations  
 (CWE-1339) in image processor leads to infinite loop (CWE-835)

**•** 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-2008-2108: Chain: insufficient precision (CWE-1339) in  
 random-number generator causes some zero bits to be reliably  
 generated, reducing the amount of entropy (CWE-331)

**•** CVE-2006-6499: Chain: web browser crashes due to infinite loop - "bad  
 looping logic [that relies on] floating point math [CWE-1339] to exit  
 the loop [CWE-835]"

## Modes of Introduction

**•** Implementation: This weakness is introduced when the developer picks a method to represent a real number. The weakness may only be visible with very specific numeric inputs.

## Common Consequences

**•** Impact: DoS: Crash, Exit, or Restart — Notes: This weakness will generally lead to undefined results and therefore crashes. In some implementations the program will halt if the weakness causes an overflow during a calculation.

**•** Impact: Execute Unauthorized Code or Commands — Notes: The results of the math are not as expected. This could cause issues where a value would not be properly calculated and provide an incorrect answer.

**•** Impact: Read Application Data, Modify Application Data — Notes: This weakness can sometimes trigger buffer overflows which can be used to execute arbitrary code. This is usually outside the scope of a product's implicit security policy.

## Potential Mitigations

**•** Implementation: The developer or maintainer can move to a more accurate representation of real numbers. In extreme cases, the programmer can move to representations such as ratios of BigInts which can represent real numbers to extremely fine precision. The programmer can also use the concept of an Unum real. The memory and CPU tradeoffs of this change must be examined. Since floating point reals are used in many products and many locations, they are implemented in hardware and most format changes will cause the calculations to be moved into software resulting in slower products. (Effectiveness: N/A)

## Applicable Platforms

**•** None (Class: Not Language-Specific, Prevalence: Undetermined)

## Demonstrative Examples

**•** The chart below shows values for different data structures in the rust language when Muller's recurrence is executed to 80 iterations. The data structure f64 is a 64 bit float. The data structures I<number>F<number> are fixed representations 128 bits in length that use the first number as the size of the integer and the second size as the size of the fraction (e.g. I16F112 uses 16 bits for the integer and 112 bits for the fraction). The data structure of Ratio comes in three different implementations: i32 uses a ratio of 32 bit signed integers, i64 uses a ratio of 64 bit signed integers and BigInt uses a ratio of signed integer with up to 2^32 digits of base 256. Notice how even with 112 bits of fractions or ratios of 64bit unsigned integers, this math still does not converge to an expected value of 5.

**•** N/A

**•** N/A