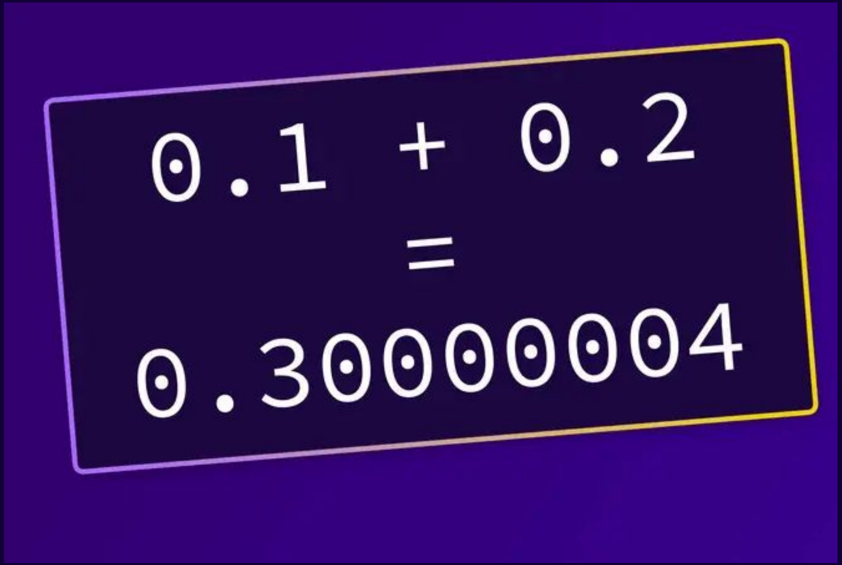
Floats: Pequenas imprecisões Grandes vulnerabilidades



Felipe Caon

15 Dezembro 2024

BUG BOUNTY BRAZIL

AGENDA

CONCEITOS

1 O QUE SÃO FLOATS?

2 IEEE 754

3 REPRESENTAÇÕES 32 E 64 BITS

4 FLOATS E ROUNDS

HACKING

5 VULNS ENCONTRADAS

O QUE SÃO FLOATS?

- Representação de números com casas decimais
- Utiliza **IEEE 754** para representação binária

```
2.12
8.98
3.14159265359
0.0000000000001
```

INTEIRO	FLOAT
42	3.141
=	≈
101010	???

IEEE 754

The Baleful Influence of Benchmarks:
Hardware and compilers are increasingly being rated entirely according to their performance in benchmarks that measure only speed. That is a mistake committed because speed is so much easier to measure than other qualities like reliability and convenience. Sacrificing them in order to run faster will compel us to run longer. By disregarding worthwhile qualities other than speed, current benchmarks penalize conscientious adherence to standards like IEEE 754; worse, attempts to take those qualities into account are thwarted by political constraints imposed upon programs that might otherwise qualify as benchmarks.

For example, a benchmark should compile and run on every commercially significant computer system. This rules out our programs for solving the differential equation and the eigenvalue problem described above under the Digression on Division-by-Zero. To qualify as benchmarks, programs must prevent exceptional events that might stop or badly slow some computers even if such prevention retards performance on computers that, by conforming conscientiously to IEEE 754, would not stop.

The Digression on Gradual Underflow offered an example of a benchmark that lent credibility to a misguided preference for Flush-to-Zero, in so far as it runs faster than Gradual Underflow on some computers, by disregarding accuracy. If Gradual Underflow's superior accuracy has no physical significance there, neither has the benchmark's data.

Accuracy poses tricky questions for benchmarks. One hazard is the ...

> Stopped Clock Paradox: Why is a mechanical clock more accurate stopped than running?
> A running clock is almost never exactly right, whereas a stopped clock is exactly right twice a day.
> (*But WHEN is it right? Alas, that was not the question.*)

The computational version of this paradox is a benchmark that penalizes superior computers, that produce merely excellent approximate answers, by making them seem less accurate than an inferior computer that gets exactly the right answer for the benchmark's problem accidentally. Other hazards exist too; some will be illustrated by the next example.

Quadratic equations like

$$p\,x^2 - 2\,q\,x + r = 0$$

arise often enough to justify tendering a program that solves it to serve as a benchmark. When the equation's roots x1 and x2 are known in advance both to be real, the simplest such program is the procedure `Qdrtc` exhibited on the next page.

In the absence of premature Over/Underflow, `Qdrtc` computes x1 and x2 at least about as accurately as they are determined by data { p, q, r } uncorrelatedly uncertain in their last digits stored. It should be tested first on trivial data to confirm that it has not been corrupted by a misprint nor by an ostensible correction like "x1 := (q+s)/p ; x2 := (q-s)/p " copied naively from some elementary programming text. Here are some trivial data:

{ p = Any nonzero, q = r = 0 }; x1 = x2 = 0 .

{ p = 2.0 , q = 5.0 , r = 12.0 }; x1 = 2.0 , x2 = 3.0 .

{ p = 2.0 E-37, q = 1.0 , r = 2.0 }; x1 ≈ 1.0 , x2 ≈ 1.0 E 37 .

Swapping p with r swaps { x1, x2 } with { 1/x2, 1/x1 } .

{μ*p, μ*q, μ*r} yields {x1, x2} independently of nonzero μ .

Lecture Notes on the Status of

IEEE Standard 754 for Binary Floating-Point Arithmetic

Prof. W. Kahan
Elect. Eng. & Computer Science
University of California
Berkeley CA 94720-1776

Introduction:
Twenty years ago anarchy threatened floating-point arithmetic. Over a dozen commercially significant arithmetics boasted diverse wordsizes, precisions, rounding procedures and over/underflow behaviors, and more were in the works. "Portable" software intended to reconcile that numerical diversity had become unbearably costly to develop.

Thirteen years ago, when IEEE 754 became official, major microprocessor manufacturers had already adopted it despite the challenge it posed to implementors. With unprecedented altruism, hardware designers had risen to its challenge in the belief that they would ease and encourage a vast burgeoning of numerical software. They did succeed to a considerable extent. Anyway, rounding anomalies that preoccupied all of us in the 1970s afflict only CRAY X-MPs — J90s now.

Now atrophy threatens features of IEEE 754 caught in a vicious circle:
Those features lack support in programming languages and compilers,
so those features are mishandled and/or practically unusable,
so those features are little known and less in demand, and so
those features lack support in programming languages and compilers.
To help break that circle, those features are discussed in these notes under the following headings:

Insofar as this is a status report, it is subject to change and supersedes versions with earlier dates. This version supersedes one distributed at a panel discussion of "Floating-Point Past, Present and Future" in a series of San Francisco Bay Area Computer History Perspectives sponsored by Sun Microsystems Inc. in May 1995. A Post-Script version is accessible electronically as http://http.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps .

The first example shows how Infinity eases the numerical solution of a differential equation that appears to have no divisions in it. The problem is to compute y(10) where y(t) satisfies the *Riccati* equation

$$dy/dt = t + y^2 \quad \text{for all } t \ge 0, \quad y(0) = 0 .$$

Let us pretend not to know that y(t) may be expressed in terms of Bessel functions J... , whence y(10) = -7.53121 10731 35425 34544 97349 58··· . Instead a numerical method will be used to solve the differential equation approximately and as accurately as desired if enough time is spent on it.

Q(θ, t, Y) will stand for an *Updating Formula* that advances from any estimate Y ≈ y(t) to a later estimate Q(θ, t, Y) ≈ y(t+θ) . Vastly many updating formulas exist; the simplest that might be applied to solve the given Riccati equation would be Euler's formula:

$$Q(\theta, t, Y) := Y + \theta\cdot(t + Y^2) .$$

This "First-Order" formula converges far too slowly as *stepsize* θ shrinks; a faster "Second-Order" formula, of Runge-Kutta type, is Heun's :

$$f := t + Y^2 ; \qquad q := Y + \theta\cdot f ;$$
$$Q(\theta, t, Y) := Y + (f + t{+}\theta + q^2)\cdot\theta/2 .$$

Formulas like these are used widely to solve practically all ordinary differential equations. Every updating formula is intended to be iterated with a sequence of stepsizes θ that add up to the distance to be covered; for instance, Q(...) may be iterated N times with constant stepsize θ := 10/N to produce Y(n·θ) ≈ y(n·θ) thus:

Y(0) := y(0) ;
for n = 1 to N do Y(n·θ) := Q(θ, (n-1)·θ, Y((n-1)·θ)) .

Here the number N of *timesteps* is chosen with a view to the desired accuracy since the error Y(10) - y(10) normally approaches 0 as N increases to Infinity. Were Euler's formula used, the error in its final estimate Y(10) would normally decline as fast as 1/N ; were Heun's, ... $1/N^2$. But the Riccati differential equation is not normal; no matter how big the number N of steps, those formulas' estimates Y(10) turn out to be huge positive numbers or overflows instead of -7.53··· . Conventional updating formulas do not work here.

The simplest unconventional updating formula Q available turns out to be this rational formula:

$$Q(\theta, t, Y) := Y + (t + \tfrac{1}{2}\theta + Y^2)\cdot\theta/(1 - \theta\cdot Y) \qquad \text{if } |\theta\cdot Y| < \tfrac{1}{2} ,$$
$$:= (1/\theta + (t + \tfrac{1}{2}\theta)\cdot\theta)/(1 - \theta\cdot Y) - 1/\theta \qquad \text{otherwise.}$$

The two algebraically equivalent forms are distinguished to curb rounding errors. Like Heun's, this Q is a second-order formula. (It can be compounded into a formula of arbitrarily high order by means that lie beyond the scope of these notes.) Iterating it N times with stepsize θ := 10/N yields a final estimate Y(10) in error by roughly $(105/N)^2$ even if Division-by-Zero insinuates an Infinity among the iterates Y(n·θ) . Disallowing Infinity and Division-by-Zero would at least somewhat complicate the estimation of y(10) because y(t) has to pass through Infinity seven times as t increases from 0 to 10 . (See the graph on the next page.)

What becomes complicated is not the program so much as the process of developing and verifying a program that can dispense with Infinity. First, find a very tiny number ε barely small enough that 1 + 10 √ε rounds off to 1 . Next, modify the foregoing rational formula for Q by replacing the divisor (1 - θ·Y) in the "otherwise" case by ((1 - θ·Y) + ε) . Do not omit any of these parentheses; they prevent divisions by zero. Then perform an error-analysis to confirm that iterating this formula produces the same values Y(n·θ) as would be produced without ε except for replacing infinite values Y by huge finite values.

Survival without Infinity is always possible since "Infinity" is just a short word for a lengthy explanation. The price paid for survival without Infinity is lengthy cogitation to find a not-too-lengthy substitute, if it exists.

https://ieeexplore.ieee.org/document/8766229

RESUMÃO IEEE 754

→ **Existem cálculos para representação de 32 bits e 64 bits**

ARREDONDAMENTO

Se último número for:

> 5, arredonda para cima

< 5, arredonda para baixo

= 5, arredonda para o par mais próximo

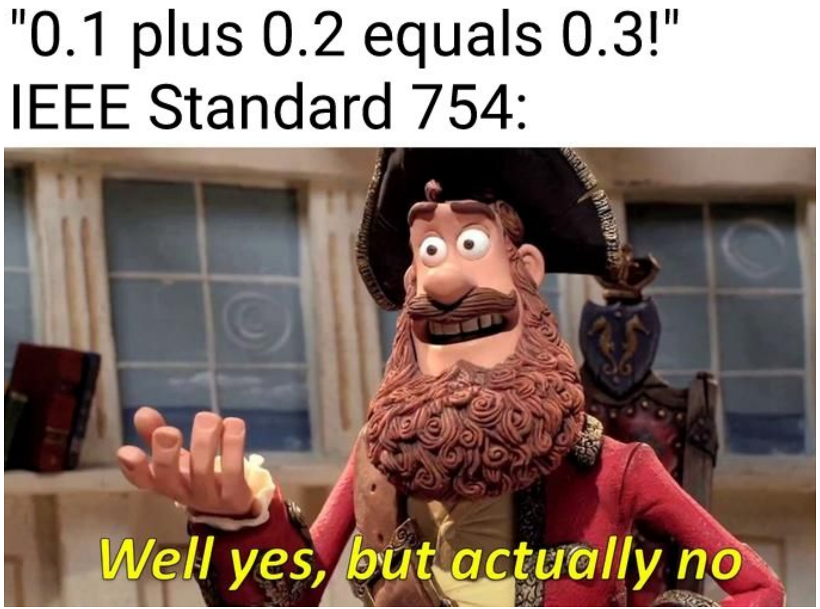
2.5 → 2 (porque 2 é par)
3.5 → 4 (porque 4 é par)
4.5 → 4 (porque 4 é par)
5.5 → 6 (porque 6 é par)

REPRESENTAÇÃO 32 E 64 BITS

```
print(0.3)
Output 32: 0.30000001192092896
Output 64: 0.30000000000000004

a = 0.1 + 0.2
b = 0.3
print(a == b)
Output 32: False
Output 64: False

print(1.0 - 0.9)
Output 32: 0.09999999403953552
Output 64: 0.09999999999999998
```



IMPLEMENTAÇÃO FLOAT

32 BITS	AMBOS	64 BITS
0.30000001192092896		0.30000000000000004

FLOATS E ROUNDS

Se **número "cabe"** na representação binária, **segue arredondamento IEEE 754**

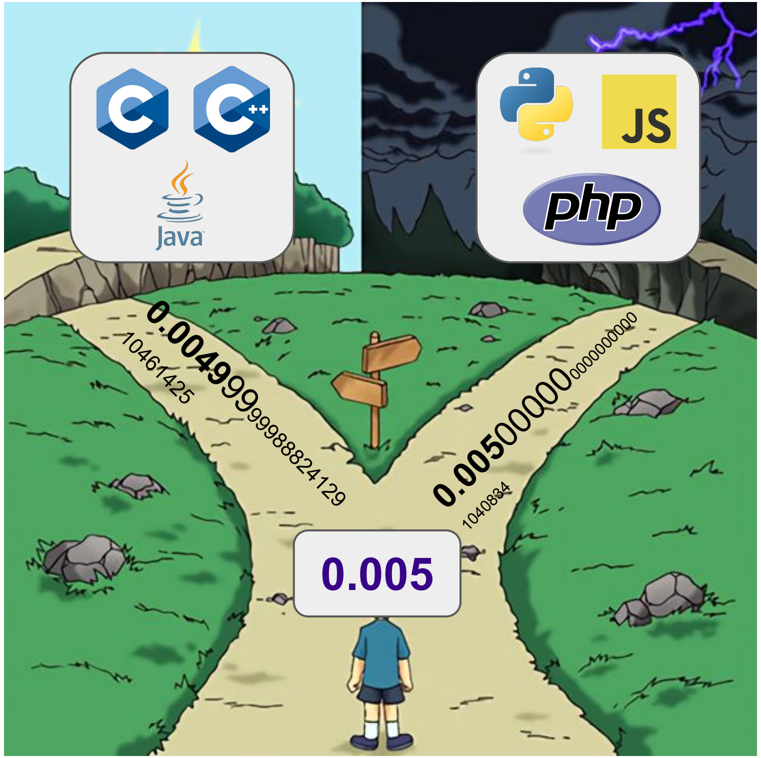
```
0.75
= 01000000010010010000111111011011
= 0.75
```

FLOATS E ROUNDS

Se **número não "cabe"** na representação binária, **arredonda** para o valor **mais próximo**

```
0.3
= 00111110100110011001100110011010
= 0.30000000000000004
```

REPRESENTAÇÃO 32 E 64 BITS



0.005

64 bits

0.005000000000000000001040834…

32 bits

0.0049999999882412910461425…

FLOAT64 E ROUNDS

1

amount = 0.002000094994902...

`amount.Round(2)`

amount = 0.00

2

amount = 0.00800000037997...

`amount.Round(2)`

amount = 0.01

3

amount = 0.005000001040834...

`amount.Round(2)`

amount = ?

FLOAT64 E ROUNDS

1

amount = 0.002000094994902...

`amount.Round(2)`

amount = 0.00

2

amount = 0.00800000037997...

`amount.Round(2)`

amount = 0.01

3

amount = 0.0050000001040834...

`amount.Round(2)`

amount = 0.01


Talk is cheap. Show
me the IMPACT

 ∞

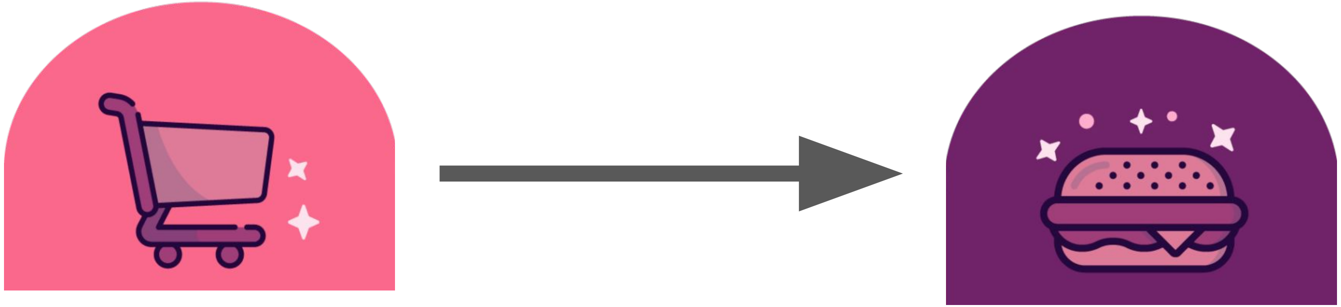
CASO #1: BENEFÍCIOS CORPORATIVOS

Vale-alimentação x **Vale-refeição**

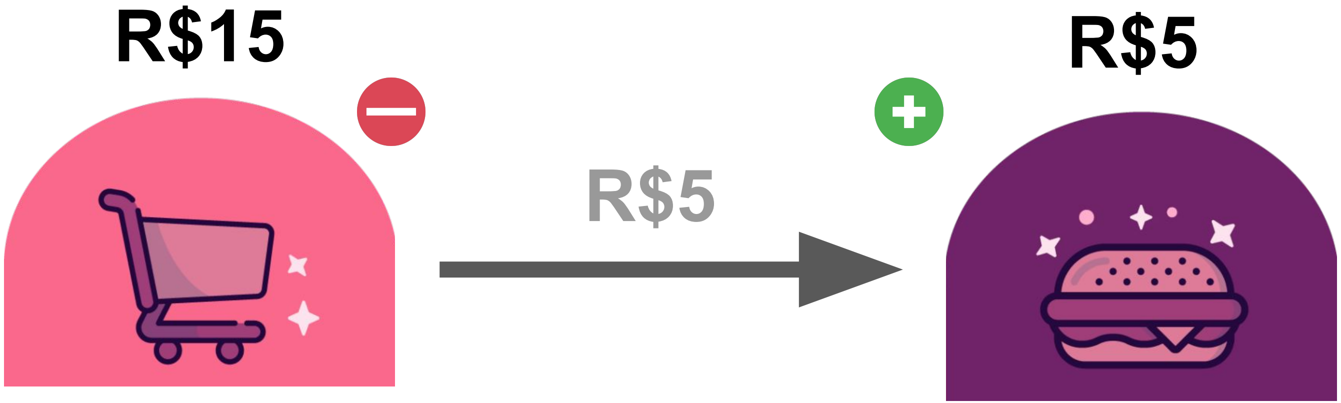
Vale-alimentação	Vale-refeição
Substitui a cesta básica	Substitui as refeições na empresa
Para usar em mercados, açougues, peixarias, feiras e etc	Para usar em restaurantes, padarias, lanchonetes, deliverys e etc
Utilizado para a aquisição de alimentos para preparo em casa	Utilizado para a aquisição de refeições prontas dentro ou fora do horário de trabalho

BENEFÍCIOS CORPORATIVOS

R$20 **R$0**



BENEFÍCIOS CORPORATIVOS



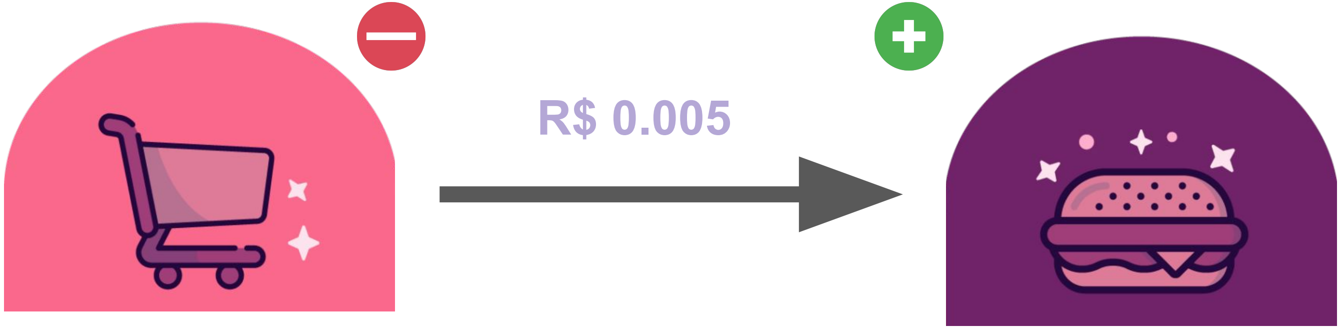
VULNERABILIDADE

R$20

R$20 - R$0.005 = R$19.995

R$??

R$0 + R$0.005 = R$0.005



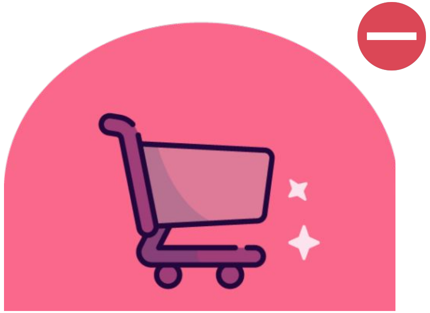
VULNERABILIDADE

R$20

R$20 - R$0.005 = R$19.995
round(19.995, 2)

R$??

R$0 + R$0.005 = R$0.005
round(0.005, 2)



R$ 0.005

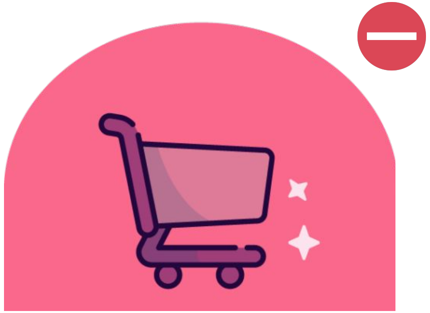
VULNERABILIDADE

R$20

R$20 - R$0.005 = R$19.995
round(19.995, 2)
Output: R$20

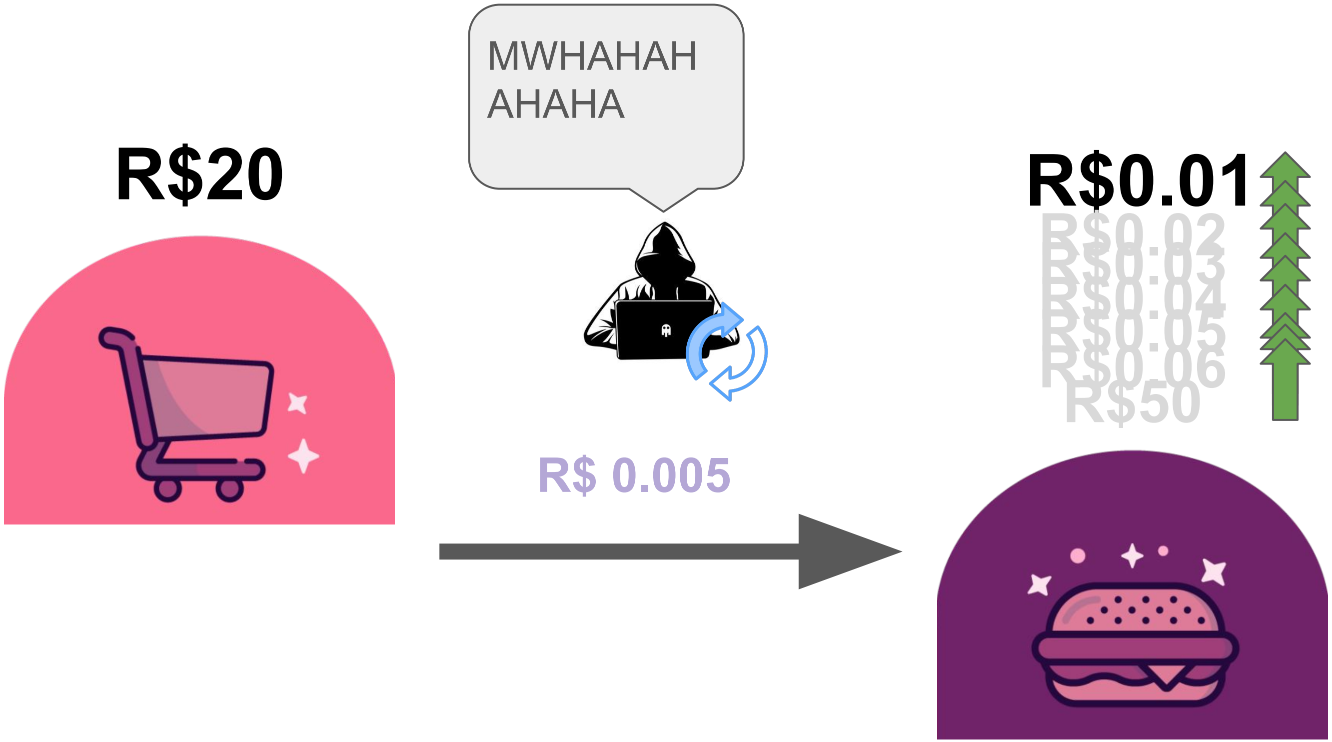
R$0.01

R$0 + R$0.005 = R$0.005
round(0.005, 2)
Output: R$0.01



R$ 0.005

VULNERABILIDADE



CASO #2: CASSINO

A CASA SEMPRE GANHA?

Saldo: R$20

```
POST /api/v1/createWithdrawal
{
  "sessionId":"123456",
  "action":"withdraw",
  "amount":
}
```

A CASA SEMPRE GANHA?

Saldo: R$20 - R$0.005 = R$19.995

```
POST /api/v1/createWithdrawal
{
  "sessionId":"123456",
  "action":"withdraw",
  "amount": 0.005
}
```

A CASA SEMPRE GANHA?

Saldo: R$20 - R$0.005 = R$19.995 = **R$20**

```
POST /api/v1/createWithdrawal
{
  "sessionId":"123456",
  "action":"withdraw",
  "amount": 0.005
}
```

R$0.01

A CASA SEMPRE GANHA?

Saldo: R$20 - R$0.005 = R$19.995 = **R$20**

```
POST /api/v1/createWithdrawal
{
  "sessionId":"123456",
  "action":"withdraw",
  "amount": 0.005
}
```

R$0.01

R$0.01 * 86400 (segundos no dia) = **R$864** por dia
R$864 * 30 = **R$25.920** por mês

O QUE PODEMOS LEVAR DISSO?

→ Existem instituições que tratam dinheiro como float

→ **Testar valores como 0.005**, 1.99, 2.35

→ Testar notação científica, **5e-3 = 0.005**

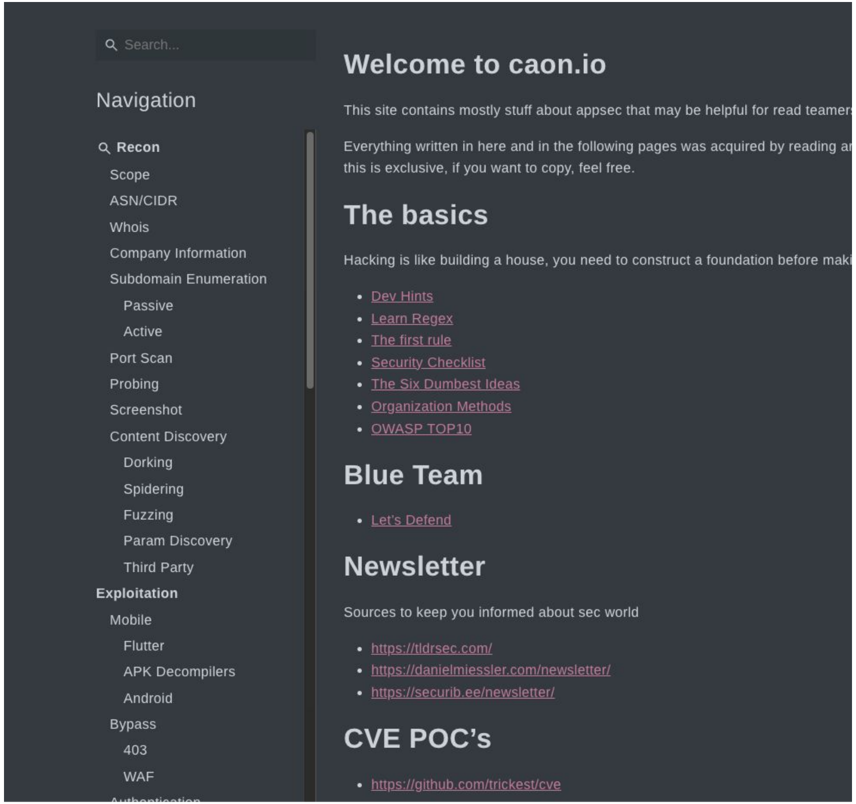
FIX??????

→ Inteiro ✅ 120 -> 120/100 -> R$1.20

→ Decimal ✅ 120,-2 -> R$1.20

→ Long para reais, int para centavos 🤨

OBRIGADO!



caon.io

linkedin.com/in/felipe-caon/

@caonio