
PVS-Studio

Go-Go-Gadg...Error?
Watching how Go developers make mistakes!

PVS-Studio

Gleb Aslamov

Senior Developer Advocate

Gleb Aslamov

Senior Developer Advocate

- C# and Go Developer
- Static analyzer developer
- I look for bugs in code and share my findings

PVS-Studio

Static analysis in the world of go

Static analysis in Go

Static analysis in Go

- Static analysis is the process of detecting flaws, bugs, and potential vulnerabilities in the source code of programs. Static analysis can be thought of as an automated code review process.

Static analysis in Go

- Static analysis is the process of detecting flaws, bugs, and potential vulnerabilities in the source code of programs. Static analysis can be thought of as an automated code review process.
- But I think you know this better than I do…
  Right?

A bit of the basics

- go vet *as* part of the toolchain

A bit of the basics

- go vet ***is*** part of the toolchain

  No, not yours
  The Go toolchain Go, it's not a separate linter

A bit of the basics

- go vet ***is*** part of the toolchain

  No, not yours
  The Go toolchain Go, it's not a separate linter

- Have you run go build? go test? You're a confident static analyzer user!

You're already using it

What we'll cover today

1. Where we got this from
2. Lets make mistakes just like everyone else
3. So why bother, if there's go vet
4. Lets make mistakes like a Go developer
5. And what to do about it?

Where did we get this from?

We recently started...

#Go Dec 29 2025

How to create your own Go static analyzer?

Artem Rovenskii

Go developers regularly deal with warnings from the built-in static analyzer. But what if you need more features or want to find something specific in your project? Go provides powerful tools...

...

#Go Mar 11 2026

PVS-Studio Go analyzer beta testing

Artem Rovenskii

After months of extensive work on a new static code analyzer for Go, our team announces that it's now ready for public testing. Join the Early Access Program to be among the first to test. Here's...

...

How we did it

- We've been building analyzers for over 18 years

How we did it

- We've been building analyzers for over 18 years
- Go analyzer starting with the first 40 diagnostic rules

How we did it

- We've been building analyzers for over 18 years
- Go analyzer starting with the first 40 diagnostic rules
- A tester with labeled findings from 33 projects

How we did it

- We've been building analyzers for over 18 years
- Go analyzer starting with the first 40 diagnostic rules
- A tester with labeled findings from 33 projects
- A stress tester with ~1000 projects for statistics

But not about that today...
Let's look at some bugs!

Universal bugs

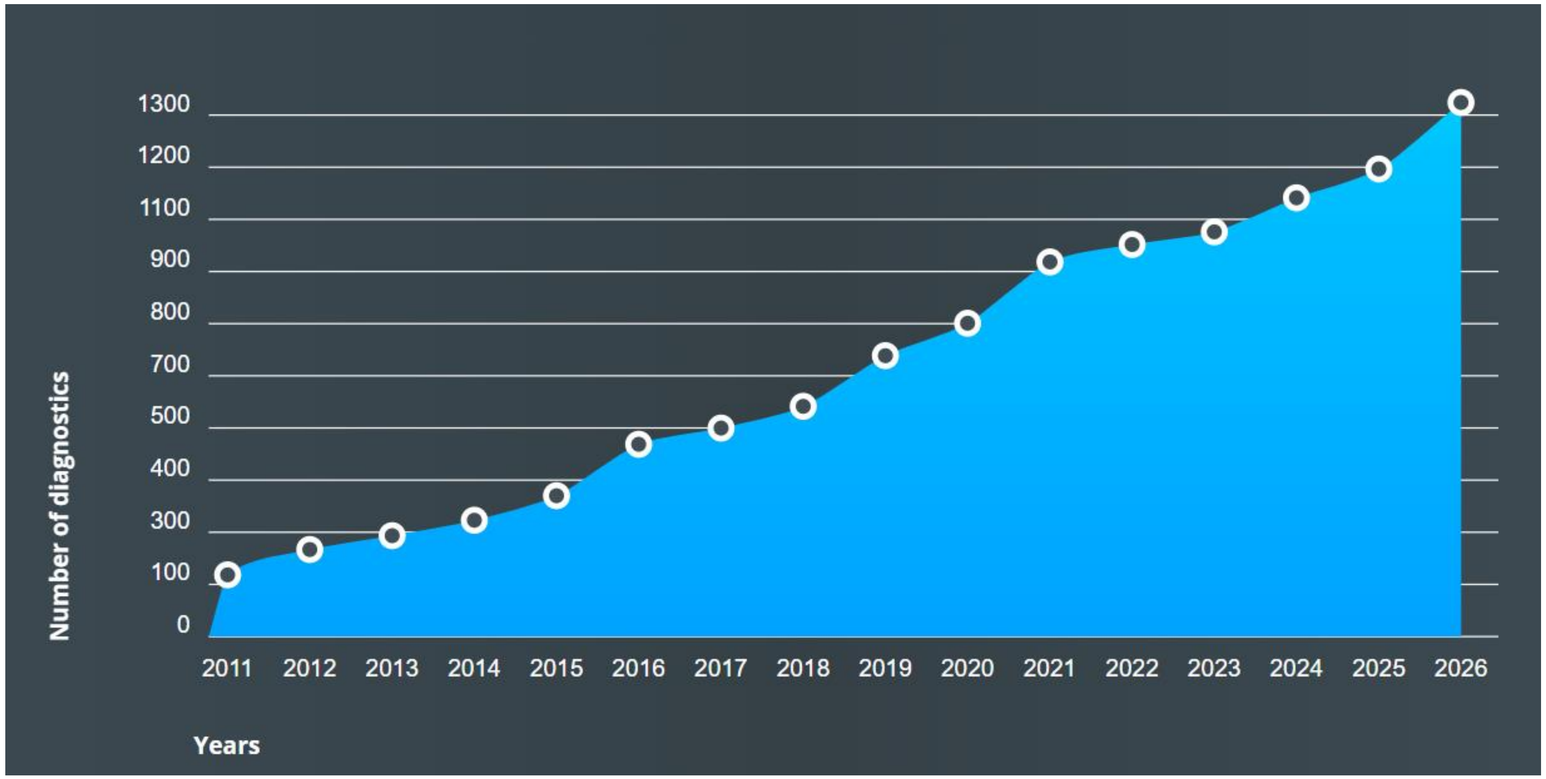
- Simple enough, but just as serious

Universal bugs

- Simple enough, but just as serious
- Anyone can make them, literally anywhere

Universal bugs

- Simple enough, but just as serious
- Anyone can make them, literally anywhere
- We have a lot of experience with this!



Example. Synthetic


```
if A == B {
   if A == B {
      ....
   }
}
```

Example. Synthetic


```
if A == B {
    if A == B {
        ....
    }
}
```

Message from PVS-Studio:

V8020. Recurring check. This condition was already verified on a previous line.

- Ha-ha, that bug is too simple! I'd never make a mistake like that!

Example. Synthetic


```
if A == B {
    if A == B {
        ....
    }
}
```

Message from PVS-Studio:

V8020. Recurring check. This condition was already verified on a previous line.

- «Ha-ha, that bug is too simple! I'd never make a mistake like that!»

...Right...?

Example. From the Incus


```
err = p.Save(pidPath)
if err != nil {
    err2 := p.Stop()
    if err != nil {
        return fmt.Errorf("...: %s: %s", err, err2)
    }
    ...
}
```

Example. From the Incus


```
err = p.Save(pidPath)
if err != nil {
    err2 := p.Stop()
    if err != nil {      // <= should be err2
        return fmt.Errorf("...: %s: %s", err, err2)
    }
    ...
}
```

Message from PVS-Studio:

V8020 Recurring check. The 'err != nil' condition was already verified on line 407

Example. From the Incus


```
err = p.Save(pidPath)
if err != nil {
    err2 := p.Stop()
    if err != nil {       // <= should be err2
        return fmt.Errorf("...: %s: %s", err, err2)
    }
    ...
}
```

Example. Synthetic


```
func Translate(value *T, numericValue int) {
    numericValue = reader.ReadInt32()
    ...
}
```

Example. Synthetic


```
func Translate(value *T, numericValue int) {
    numericValue = reader.ReadInt32() // <= the
    ...                               original value
}                                     of the
                                      parameter is
                                      never used
```

Message from PVS-Studio:

V8038. A parameter is always rewritten in the function body before being used.

Example. From the Kubernetes


```
func (c *FakeObjectConvertor)
   Convert(in, out, context interface{}) error {
      out = in
      return nil
}
```

Example. From the Kubernetes


```
func (c *FakeObjectConvertor)
   Convert(in, out, context interface{}) error {
      out = in
      return nil
}
```

Message from PVS-Studio:

V8038 The 'out' parameter is always rewritten in the function body before being used.

Example. Synthetic


```
if l >= 0x06C0 && l <= 0x06CE {
  return true
}
if l == 0x06D5 {
  return true
}
if l >= 0x0905 && l <= 0x0939 {
  return true
}
if l == 0x06D5 {
  return true
}
if l >= 0x0985 && l <= 0x098C {
  return true
}
```

Example. Synthetic


```
if l >= 0x06C0 && l <= 0x06CE {
  return true
}
if l == 0x06D5 { // <=
  return true
}
if l >= 0x0905 && l <= 0x0939 {
  return true
}
if l == 0x06D5 { // <=
  return true
}
if l >= 0x0985 && l <= 0x098C {
  return true
}
```

Example. From the vault


```
func (c *Core) PersistTOTPKey(...) error {
    val, err := jsonutil.EncodeJSON(ks)
    if err != nil {
        return err
    }
    if c.barrier.Put(ctx, &logical.StorageEntry{...}); err != nil {
        return err
    }
    return nil
}
```

Example. From the vault


```
func (c *Core) PersistTOTPKey(...) error {
    val, err := jsonutil.EncodeJSON(ks)
    if err != nil {
        return err
    }
    if c.barrier.Put(ctx, &logical.StorageEntry{...}); err != nil {
        return err                                      // ^= check the
    }                                                   old err
    return nil
}
```

Message from PVS-Studio:

V8014 Two 'if' statements have identical conditions. The first 'if' statement contains function return, making the second 'if' redundant, or the code contains a logical error.

Example. From the vault


```
func (c *Core) PersistTOTPKey(...) error {
    val, err := jsonutil.EncodeJSON(ks)
    if err != nil {
        return err
    }
    if c.barrier.Put(ctx, &logical.StorageEntry{...}); err != nil {
        return err
    }
    return nil
}
```

```
type Storage interface {
   ....
   Put(context.Context, *StorageEntry) error
   ....
}
```

Trojan Source


```
isAuthorized := false
/* verify before transfer */ if isAuthorized {
    TransferFunds(account, amount)
/* end verify */ }
```

Trojan Source


```
isAuthorized := false
/* verify before transfer */ if isAuthorized {
    TransferFunds(account, amount)
/* end verify */ }


isAuthorized := false
if isAuthorized {
    TransferFunds(account, amount)
```

Trojan Source


```
isAuthorized := false
/* verify before transfer */ if isAuthorized {
    TransferFunds(account, amount)
/* end verify */ }


isAuthorized := false
TransferFunds(account, amount)
```

Trojan Source


```
isAuthorized := false
/* verify before transfer */ if isAuthorized {
    TransferFunds(account, amount)
/* end verify */ }
```

```
isAuthorized := false
/*[RLO] } [LRI] if isAuthorized[PDI] [LRI] verify
before transfer */
    TransferFunds(account, amount)
/* end verify [RLO]{ [LRI]*/
```

Trojan Source

- `[LRI] if isAuthorized[PDI]`
  the fragment between `LRI` and `PDI` is isolated and displayed left-to-right:
  `if isAuthorized`

- `[LRI] verify before transfer */`
  to the end of the line, also an isolated block:
  `` `verify before transfer */` ``

- `[RLO]`
  reverses everything right-to-left

Trojan Source

- `[LRI] if isAuthorized[PDI]`
  the fragment between `LRI` and `PDI` is isolated and displayed left-to-right:
  `if isAuthorized`

- `[LRI]`
  to the
  `` `verify before transfer */` ``

```
'verify before transfer */', 'if isAuthorized',
{a space}, '{', {a space}
```

- `[RLO]`
  reverses everything right-to-left

Trojan Source

- `[LRI] if isAuthorized[PDI]`
  the fragment between `LRI` and `PDI` is isolated and displayed left-to-right:
  `if isAuthorized`

- `[LRI]`
  to the
  `` `verify before transfer */` ``

```
'verify before transfer */', 'if isAuthorized',
{a space}, '{', {a space}
```

Message from PVS-Studio:

- **V5901. OWASP. Code contains invisible characters that may alter its logic. Consider enabling the display of invisible characters in the code editor.**

So it will catch it, go vet right!

Is that enough...?


```
func rgb1(r float32, g float32, b float32) {
  if r > 1 || g > 1 || r > 1 {
....
  }
}
```

Is that enough...?


```
func rgb1(r float32, g float32, b float32) {
  if r > 1 || g > 1 || r > 1 {
....
  }
}
```

Of course it's enough!

go vet: `redundant or: r > 1 || r > 1`

Is that enough...?


```
func rgb1(r float32, g float32, b float32) {
  if r > 1 || g > 1 || 1 < r {
....
  }
}
```

What if it's like this...

Is that enough...?


```
func rgb1(r float32, g float32, b float32) {
  if r > 1 || g > 1 || 1 < r {
....
  }
}
```

What if it's like this...

Of course it's enough!

Is that enough...?


```
func rgb1(r float32, g float32, b float32) {
  if r > 1 || g > 1 || 1 < r {
....
  }
}
```

What if it's like this...

Of course it's enough!

Right...?


go vet can't go

Is that enough...?


```
func rgb1(r float32, g float32, b float32) {
  if r > 1 || g > 1 || 1 < r {
....
  }
}
```

What if it's like this...

Of course **not** enough!

go vet: All good! 👍

What if it's not synthetic? Nuclei


```
func NewEntityParser(dir string) (*EntityParser, error) {
  cfg := &packages.Config{
    Mode:
                  packages.NeedName   | packages.NeedFiles |
                  packages.NeedImports| packages.NeedTypes |
                  packages.NeedSyntax | packages.NeedTypes |
                  packages.NeedModule | packages.NeedTypesInfo,
    ....
  }
}
```

What if it's not synthetic? Nuclei


```
func NewEntityParser(dir string) (*EntityParser, error) {
  cfg := &packages.Config{
    Mode:
                  packages.NeedName   | packages.NeedFiles |
                  packages.NeedImports| packages.NeedTypes |
                  packages.NeedSyntax | packages.NeedTypes |
                  packages.NeedModule | packages.NeedTypesInfo,
    ....
  }
}
```

go vet: All good! 

Well, that's probably a one-off case...

How to overdo it and underdo it at the same time

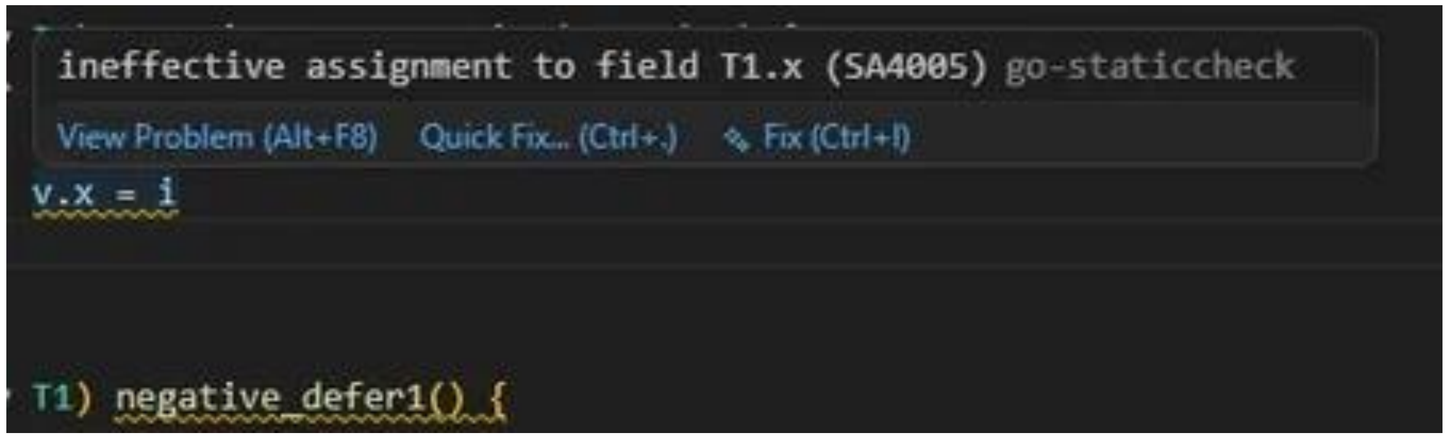
- Staticchek

```
func (v T1) negative_range_string(s string) {
    for i := range s {
        fmt.Println(v.x)
        v.x = i
    }
}
```

How to overdo it and underdo it at the same time

- Staticchek

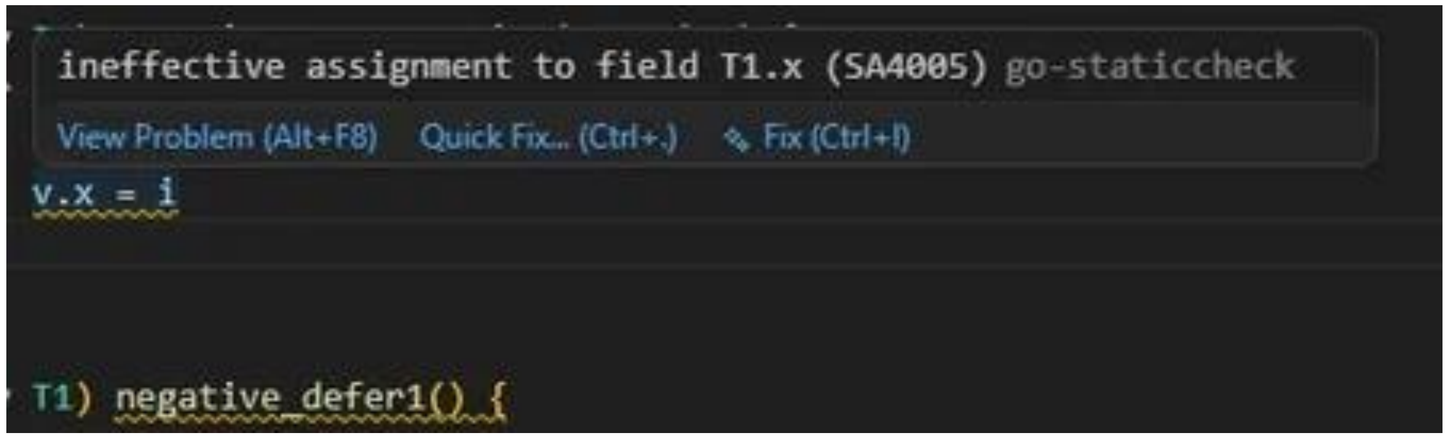
```
func (v T1) negative_range_string(s string) {
    for i := range s {
        fmt.Println(v.x)
        v.x = i
    }
}
```



How to overdo it and underdo it at the same time

- Staticchek

```
func (v T1) negative_range_string(s string) {
    for i := range s {
        fmt.Println(v.x)
        v.x = i
    }
}
```



But is it even a bug?

No!

How to overdo it and underdo it at the same time

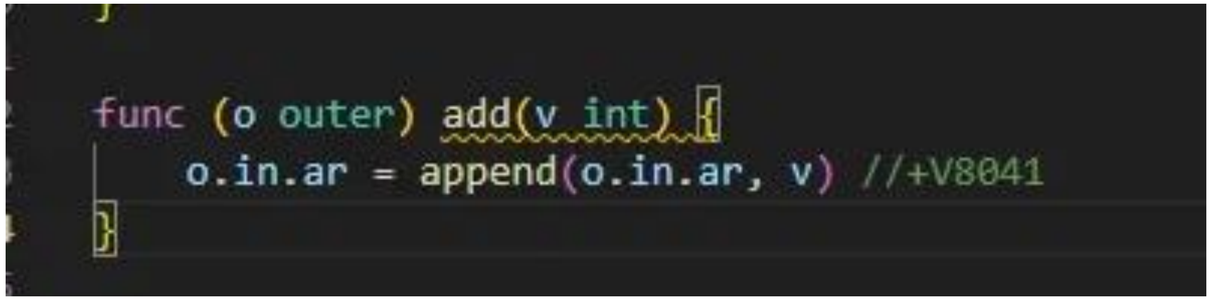
- Another Staticchek

```
func (o outer) add(v int) {
    o.in.ar = append(o.in.ar, v)
}
```

How to overdo it and underdo it at the same time

- Another Staticchek

```
func (o outer) add(v int) {
    o.in.ar = append(o.in.ar, v)
}
```



But is it even a bug?

Yes!

Some managed to spot it, some didn't

- One more example

```
var v interface {
    Read()
}

switch v.(type) {
case io.ReadCloser: //+V8035
    fmt.Println(1)
default:
    fmt.Println(2)
}
```

Language-specific bugs

XOR not what it seems

- `^(XOR)` in Lua/VB.NET/R everything is classic and it's exponentiation
- But in Go `^(XOR)` has nothing to do with exponentiation

XOR not what it seems

- `^(XOR)` in Lua/VB.NET/R everything is classic and it's exponentiation
- But in Go `^(XOR)` has nothing to do with exponentiation

For example:

```
x := 2 ^ 16
```

Expectation

```
x:= 65636 (2^16)
```

Reality

```
x:=18
```

Example. From the Lancet


```
func (q *ArrayQueue[T]) Enqueue(item T) bool {
  if q.tail < q.capacity {
    q.data = append(q.data, item)
    // q.tail++
    q.data[q.tail] = item
  } else {
    //upgrade
    if q.head > 0 {
      ....
    } else {
      if q.capacity < 65536 {
        if q.capacity == 0 {
          q.capacity = 1
        }
        q.capacity *= 2
      } else {
        q.capacity += 2 ^ 16 // <=
      }
      tmp := make([]T, q.capacity, q.capacity)
      copy(tmp, q.data)
      q.data = tmp
    }
    q.data[q.tail] = item
  }
  q.tail++
  q.size++
  return true
}
```

Message from PVS-Studio:

V8015 Suspicious use of the bitwise XOR operator '^'. The exponentiation operation may have been intended here

Example. From the Lancet


```
if q.capacity < 65536 {
        if q.capacity == 0 {
          q.capacity = 1
        }
        q.capacity *= 2
      } else {
        q.capacity += 2 ^ 16 // <=
      }
```

Example. From the Lancet


```
func (q *ArrayQueue[T]) Enqueue(item T) bool {
  if q.tail < q.capacity {
    q.data = append(q.data, item)
    // q.tail++
    q.data[q.tail] = item
  } else {
    //upgrade
    if q.head > 0 {
      ....
    } else {
      if q.capacity < 65536 {
        if q.capacity == 0 {
          q.capacity = 1
        }
        q.capacity *= 2
      } else {
        q.capacity += 2 ^ 16 // <=
      }
      tmp := make([]T, q.capacity, q.capacity)
      copy(tmp, q.data)
      q.data = tmp
    }
    q.data[q.tail] = item
  }
  q.tail++
  q.size++
  return true
}
```

Message from PVS-Studio:

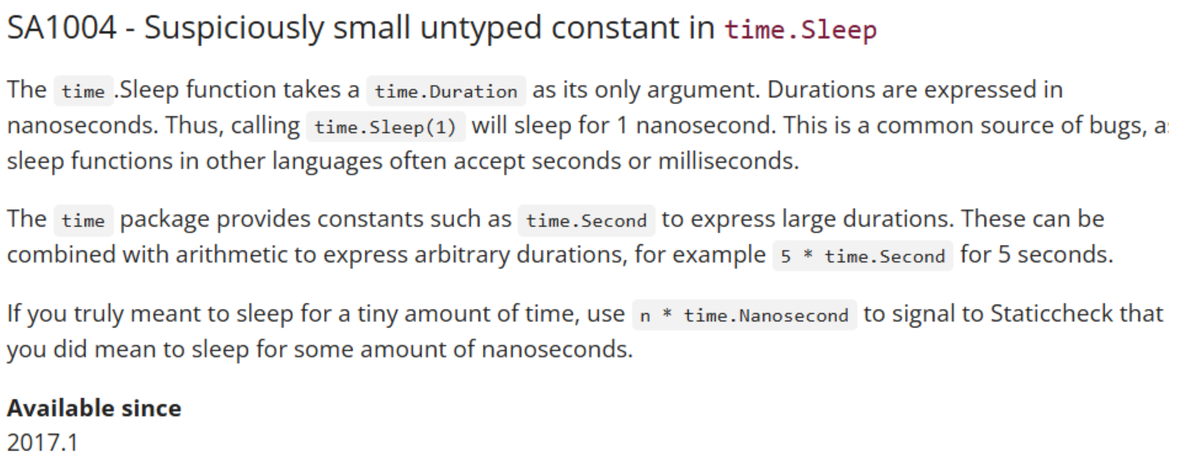
V8015 Suspicious use of the bitwise XOR operator '^'. The exponentiation operation may have been intended here

Documentation

- Description (why, where from, official sources and docs)
- Example of the bug
- Example of the fix (with an explanation of how)
- Mapping to standards
- Examples from real projects

Documentation

- Description (why, where from, official sources and docs)
- Example of the bug
- Example of the fix (with an explanation of how)
- Mapping to standards
- Examples from real projects



SA1004 - Suspiciously small untyped constant in `time.Sleep`

The `time`.Sleep function takes a `time.Duration` as its only argument. Durations are expressed in nanoseconds. Thus, calling `time.Sleep(1)` will sleep for 1 nanosecond. This is a common source of bugs, a sleep functions in other languages often accept seconds or milliseconds.

The `time` package provides constants such as `time.Second` to express large durations. These can be combined with arithmetic to express arbitrary durations, for example `5 * time.Second` for 5 seconds.

If you truly meant to sleep for a tiny amount of time, use `n * time.Nanosecond` to signal to Staticcheck that you did mean to sleep for some amount of nanoseconds.

Available since
2017.1

So what do we do?

PVS-Studio 8.0

- Support for analyzing Go, Js, Ts
  (besides the C\C++\C#\Java)
- 40 new diagnostics for each language
  (on top of the existing 1000+)
- New IDE and updates to the classics
  (a large-scale rework of VS Code)

Q/A