Roberto Clapis (They/Them)

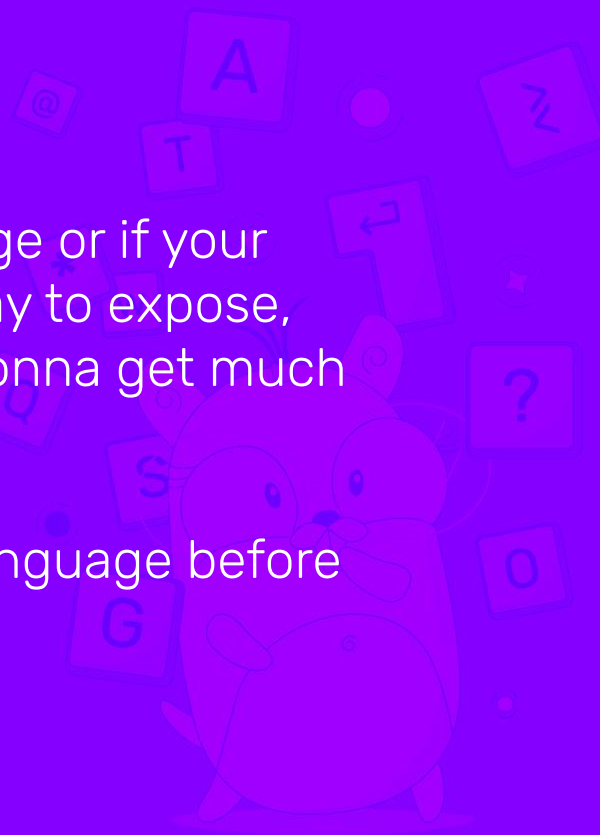
Funemployed

Safe by Construction

A word of warning

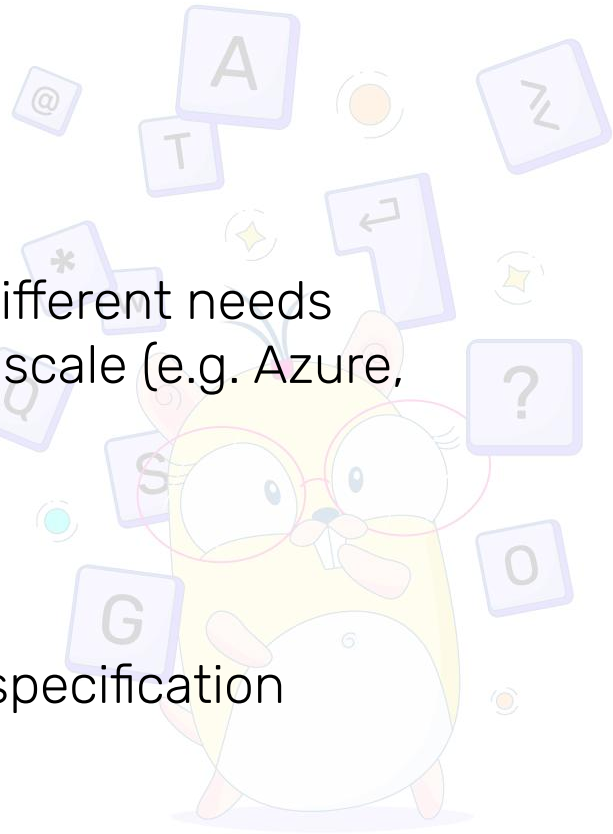
If you don't use a **memory safe** language or if your language/toolchain of choice has no way to expose, simulate or enforce **types**, you're not gonna get much value from this talk.

Please use types and a memory-safe language before applying anything that follows.



Why am I giving this talk?

- 3 years as a security consultant
  - Consulted for tens of companies with different needs
  - 9 months consulting for big projects at scale (e.g. Azure, GE…)
- 5 years at Google
  - Specializing in web security
  - Almost only defensive work
  - Got a chance to contribute to the web specification



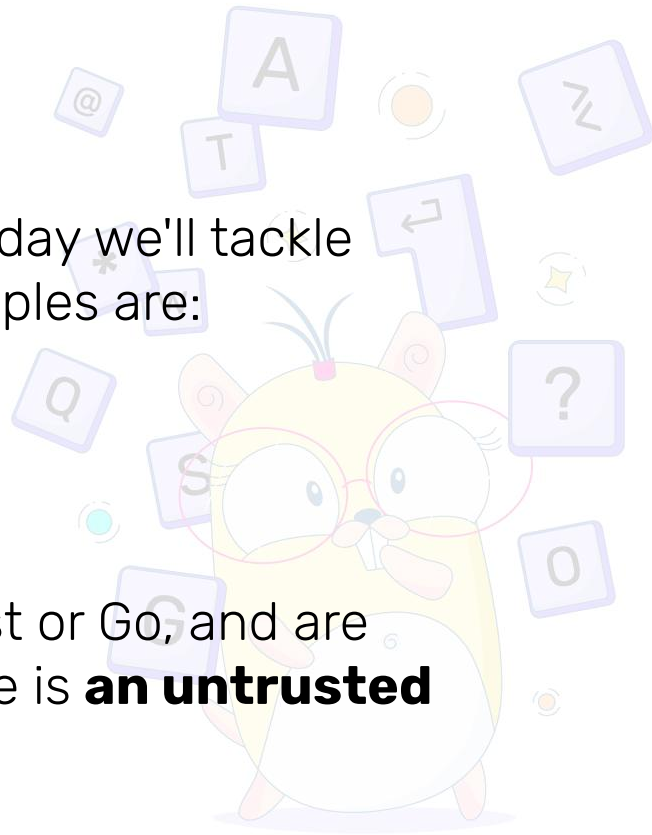
What is this talk about?

A big chunk of security is solved.

By the end of this talk you'll know how to replicate this result.

Problem Definition

Problem definition



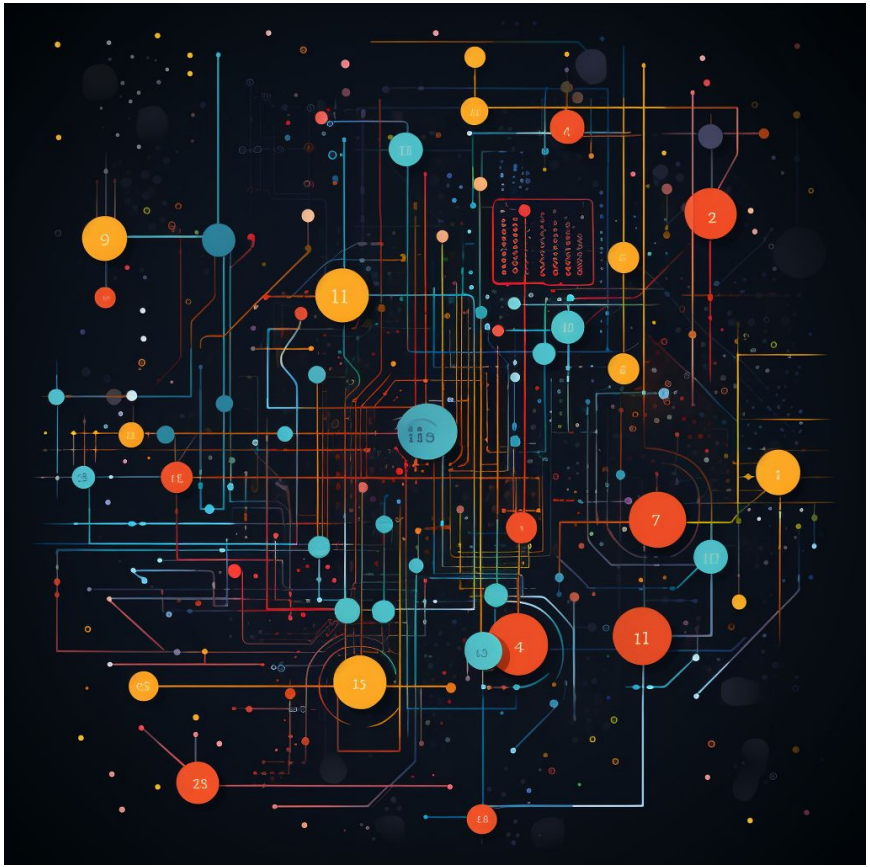
There are many security issues in the wild, today we'll tackle **injection-related** ones, some notable examples are:

- XSS
- Command injection
- SQL injections

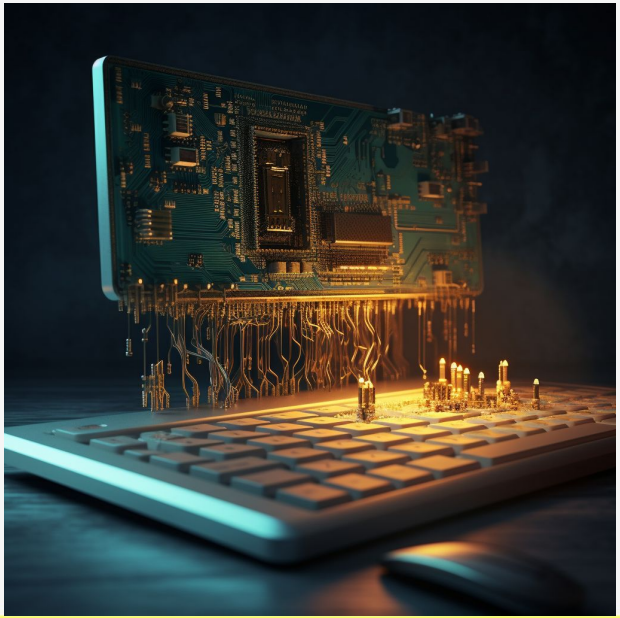
These affect memory safe languages like Rust or Go, and are common problems in all contexts where there is **an untrusted party** and an **authoritative one**.

A program

Any program can be abstracted as a function from inputs to outputs, and inputs should not become part of the function itself.



Code injections



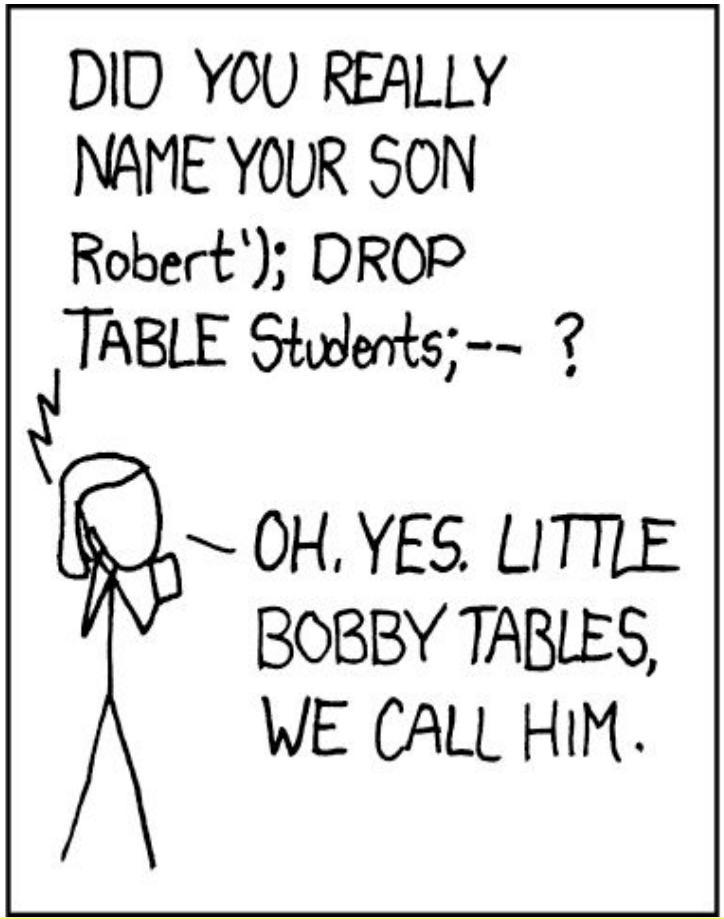
- They happen when data accidentally gets executed as code.
- This affects more or less all code that runs on a server of any sort, plus some client-side code too.
- This is specifically dangerous when more than one language or context of execution is at play, e.g:
  - SQL called by Go
  - HTML rendered by a node backend
  - A shell command ran by Rust

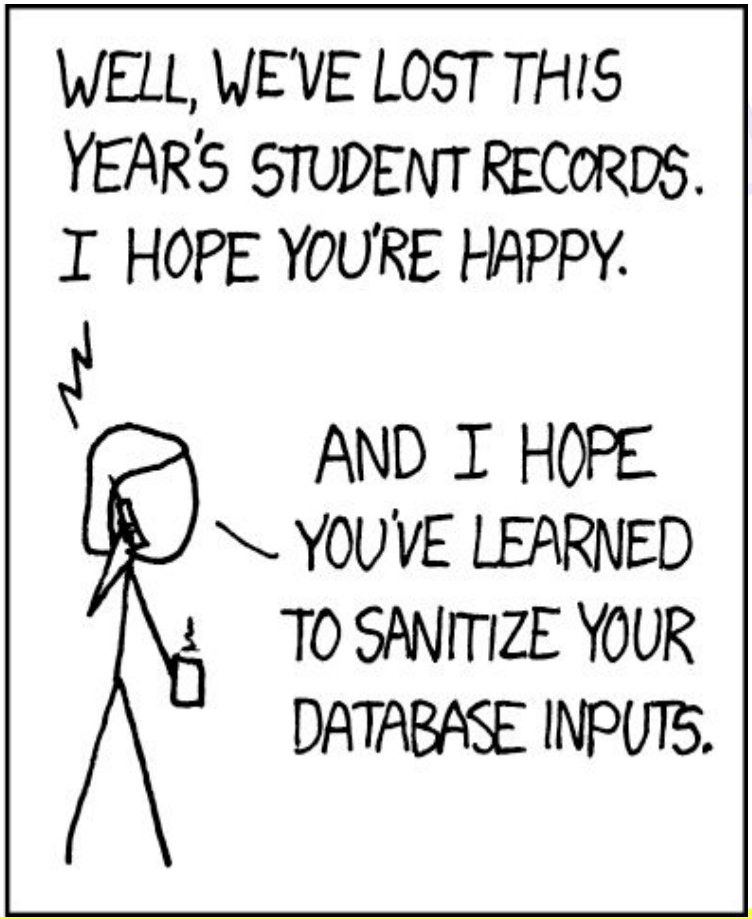
- Be careful approach

Command injections should clearly never happen

So… just be careful, right?

- 2007


DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.


WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Examples

This is bad if:

- The user inputting the highlighted string doesn't own the machine running this code
- The machine running this code contains any valuable information or has any value for its owner
- …

```
let mut cmd_exec =
    Command::new("sh");
cmd_exec.arg("-c")
        .arg(USER_INPUT);

cmd := exec.Command("sh",
            "-c",
            USER_INPUT)
```

A slightly more complex example

But this encapsulates the entire issue.

The SQL engine in the first example knows how to handle the string.

```
db.QueryRow("SELECT id from
db where name = ?", name)

db.QueryRow("SELECT id from
db where name =
'"+name+"'")

db.QueryRow("SELECT id from
db where " + stringVar)
```

A complex example

How many contexts are there here?

```
<img
onclick="alert('clicked')"

style="outline: 1px solid
red"

src="data:image/svg+xml;
   utf8,<svg>...</svg>"
>
```

A more complex example

How many contexts are there here?

Way too many.

```
<img
onclick="alert('clicked')"

style="outline: 1px solid
red"

src="data:image/svg+xml;
    utf8,<svg>...</svg>"
>
```

- But… how careful are we talking?

The solutions is to make sure no untrusted strings ever become part of a page without being properly **escaped** (e.g. using < instead of < when the context is HTML)

Manually escaped.

For the correct context.

EXACTLY ONCE.

(Who hasn't seen some "?)

- We cannot possibly be that careful

The standard approach requires all programmers:

- to be **security-savvy**
- to **never make mistakes**
- to **fully understand the dataflow** of the entire software infrastructure, any time they write code, even code they don't own.

This is a nice fantasy novel. We know what happened with manual memory management, there's a reason Go and Rust exist.

I would not trust myself to be able to consistently do this.

- There's even more

… well, sometimes **we need users to provide markup** for their text, for example in mail clients that run in web browsers, emails use HTML for style…

So we have to let **some** code execute.

So escape only dangerous markup… but not all of it?

Definition

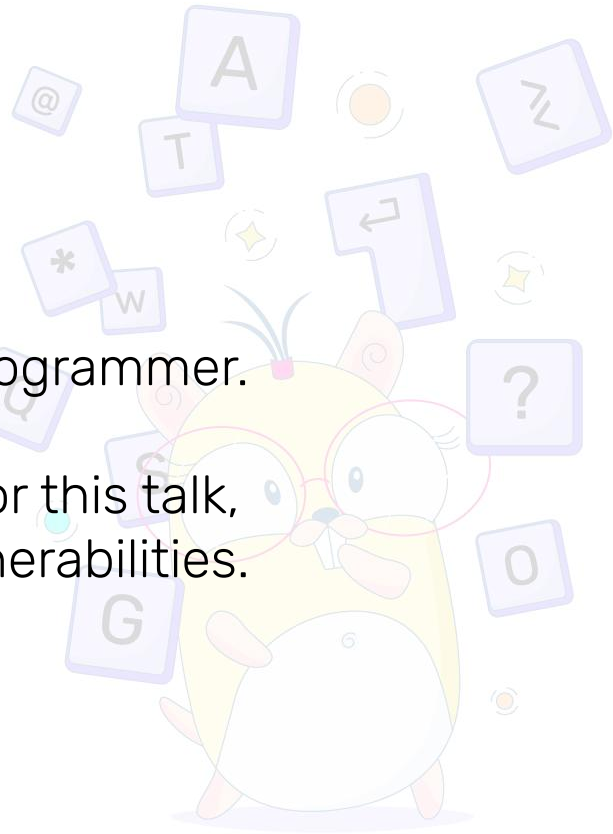
An absolute mess.
So, what can we do?

Solution Definition

Good ideas that don't work

- Tainting
- Linters
- Awareness courses
- Hope
- Penetration tests
- Prayers

What can we trust?



Code that has been written by the programmer.

We **do not** consider insider threats for this talk,
we try to prevent **unintentional** vulnerabilities.

Not all strings are born equal

There are strings that are as trustworthy as source code: **compile-time constants**.

- We should treat all other strings as unsafe
- We should have a way to have runtime-behavior for compile-time constants
- Later on, we should figure out how to "promote" an unsafe string to a safe one, or keep it unsafe and use it safely.

Just allow constants: Rust

The simplest approach, but already takes care of some important cases.

```
macro_rules! const_fn {
    ($a:literal) => {
        dynamic_fn($a)
    }
}

const_fn!("hello", 12)
const_fn!(s, x)
```

Just allow constants: Go

The simplest approach, but already takes care of some important cases.

```
package safe

type constantStr string
func ConstFn(s constantStr)


ConstFn("foo")
ConstFn(stringVar)
```

Just allow constants: TS

The simplest approach, but already takes care of some important cases.

```
function trusted(tpl:
TemplateStringsArray)...{

trusted`something`;
trusted("something");
```

Just allow constants: TS with JS interop

We can check some things at runtime when needed.

```
function trusted(tpl:
TemplateStringsArray)...{

if(!Array.isArray(tpl.raw)||
  !Object.isFrozen(tpl) ||
  !Object.isFrozen(tpl.raw)) {
throw new Error("plz stahp");
```

The one gadget we need

We can create **safe runtime strings** that can only be constructed with compile-time constants.

This trick allows us to define **functions that can only be called safely**. Their args are as trusty as code.

With this, we may decide to allow programmers to concatenate safe strings with other constants or with safe strings, and accept that the result is still safe.

A practical example

google/go-safeweb/safesql

A safe version of the standard sql package in 60 lines of code.

```
type stringConstant string

type TrustedSQLString struct {
    s string
}

func New(text stringConstant)
TrustedSQLString {
    return TrustedSQLString{
                string(text)
            }
}

db.QueryRow(Trusted, any)
```

How to deal with untrusted strings

- Most programs take user input and need to do something with it.
- Such input needs to flow through the program and be rendered/used.

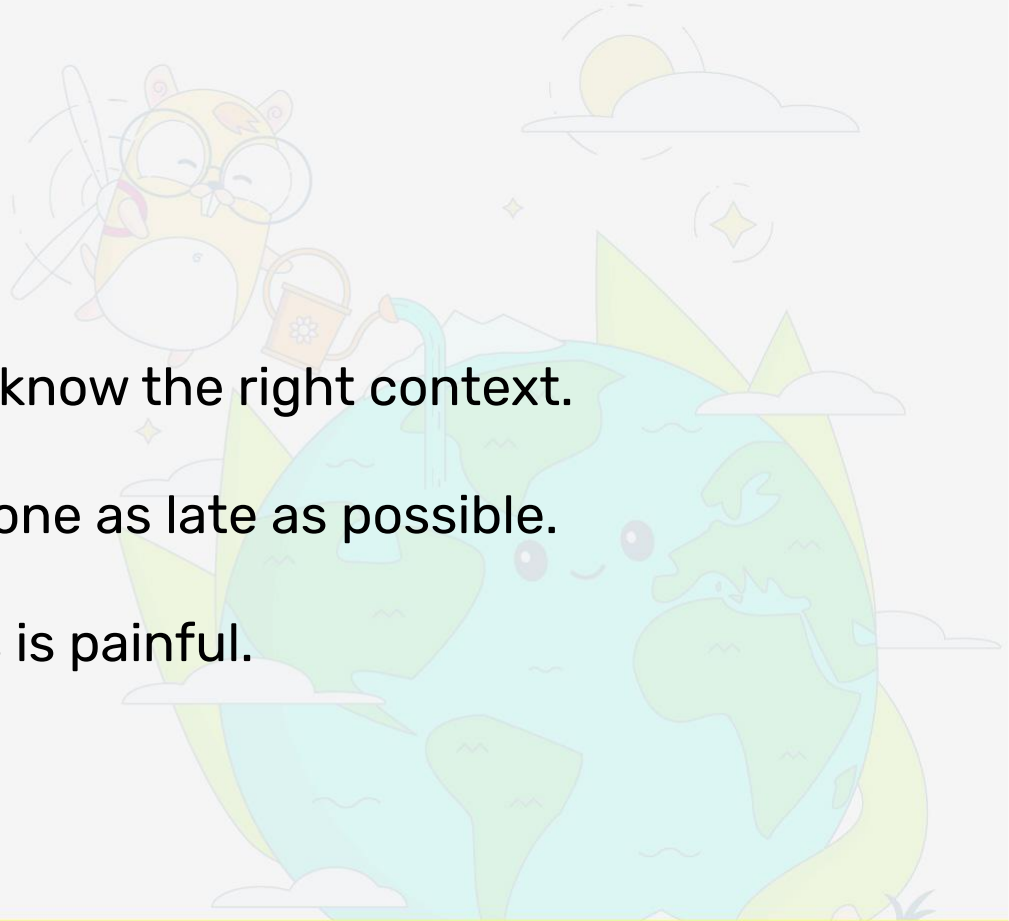
Are just strings

Any function that accepts just strings should be written to assume untrustworthy content.

- SQL engines usually have prepared statements that can be used to pass strings as just strings.
- HTML renderers… don't. At least not fully. Not in all languages.

(A safe HTML template for Rust would be a nice project, please contact me if you are interested, I have proposals)

The problem with escaping



Escaping requires to know the right context.

Escaping **must** be done as late as possible.

And this is painful.

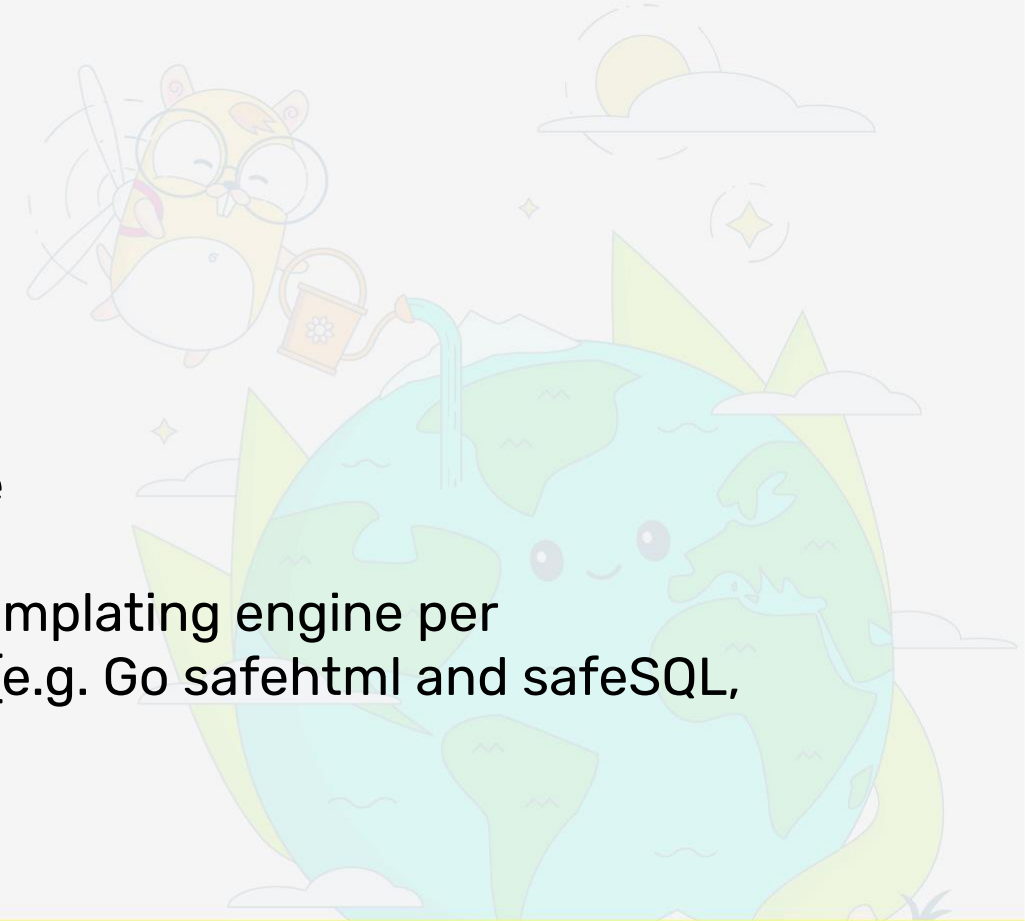
Writing an escaper

If the escaper just treats this as HTML, JS is still going to run.

We need context-aware automatic escaping templates.

```
<img onclick="{{.UserInput}}">

jsMaliciousFunc("malicious")

Will be escaped as:

jsMaliciousFunc(&quot;malicious&quot;)

Which the browser will decode and execute.
```

Contextual autoescaping

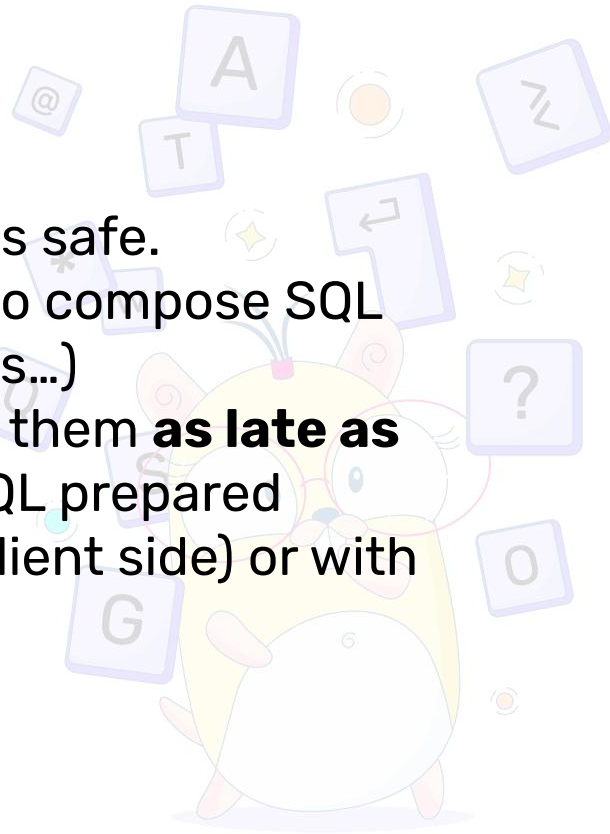


- Expensive to write
- Hard to write
- Requires maintenance
- It's the only way to be safe

We should have one trusted templating engine per language/script combination (e.g. Go safehtml and safeSQL, Rust safehtml and safeSQL…)

Short recap

- Use a way to mark all compile-time data as safe.
- Only use **safe data as part of code** (e.g. to compose SQL queries, to render HTML, to run commands…)
- If you need to use unsafe strings, **escape** them **as late as possible**, preferably within the engine (SQL prepared statements, use HTML .innerText on the client side) or with context-aware templates.



The End?

I've watched and seen a lot of talks and documentation on the topic end on this note.

But this is not enough.

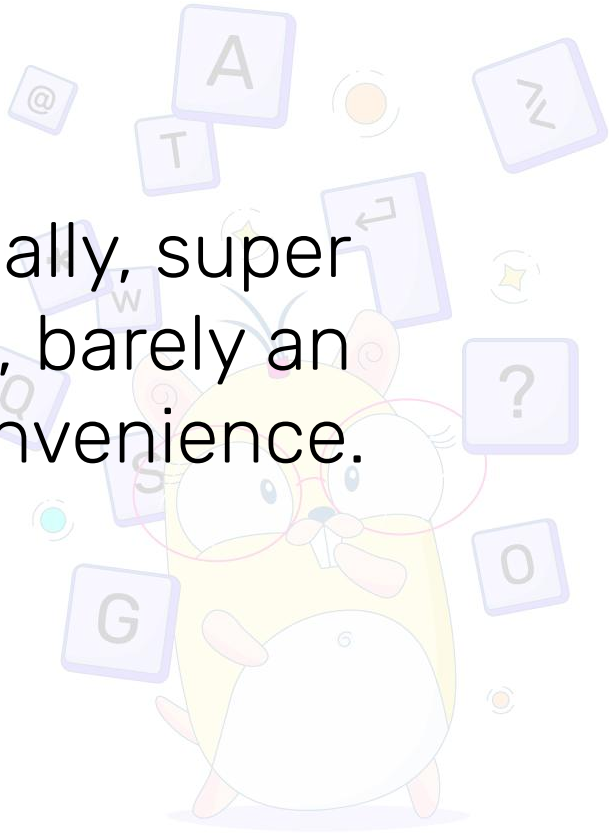
Solution usage

Sounds hard

This sounds like it's gonna be hard to adopt

But it isn't

Actually, super easy, barely an inconvenience.

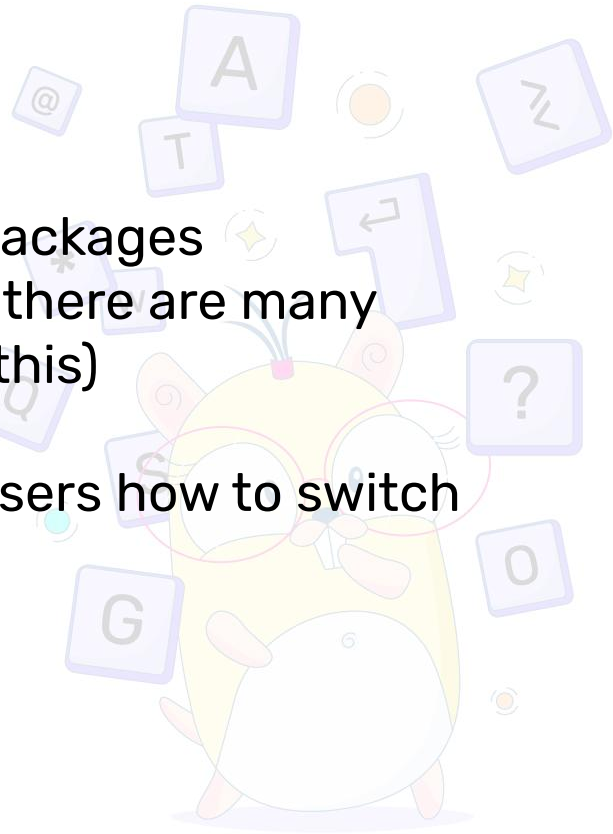


How to use it

Adoption is simpler than it sounds.
Really.
Pinky promise.

- For new code, just use the wrappers that force safe code, and ban the other packages.
- For old code:
  - Promote existing uses as safe
  - Force new code to be safe
  - Refactor old code to be safe
- This is not a Big Tech Energy approach

New code



The hardest part is to ban use of dangerous packages

- CI/CD could block new code that doesn't, there are many tools for that (we even wrote one just for this)
- Said tools can even run locally
- Just make sure that they clearly tell the users how to switch to the safe versions

Old code

Step 1

- Promote it as safe
- Ban all new uses of unsafe code

This should be done as atomically as possible.

Old code

Step 2

- Fix old code
- Take all the time you need
- Or don't, I'm not a cop

While you do, keep in mind that
usually the harder the refactor is,
the more likely it is to remove vulns.

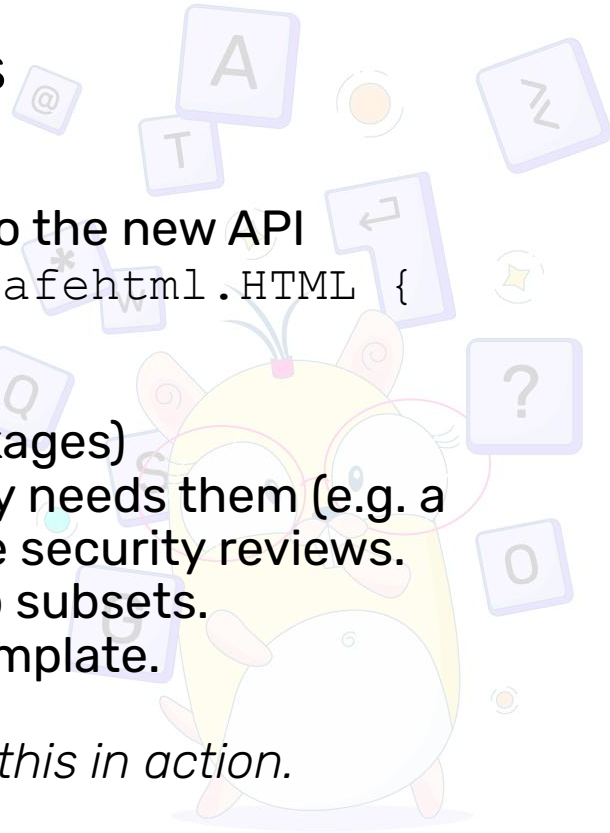
To do so, we use different conversions

- Legacy conversions to promote all old code to the new API

```
func LegacyRiskyAssumeHTML(s string) safehtml.HTML {
    return html(s)
}
```

- Testconversions for tests (use test-only packages)
- Unchecked conversions for stuff that actually needs them (e.g. a local SQL interpreter). They will all still require security reviews.
- Sanitizers, if you need, to create safe markup subsets.
- Raw strings will be handled by the engine/template.

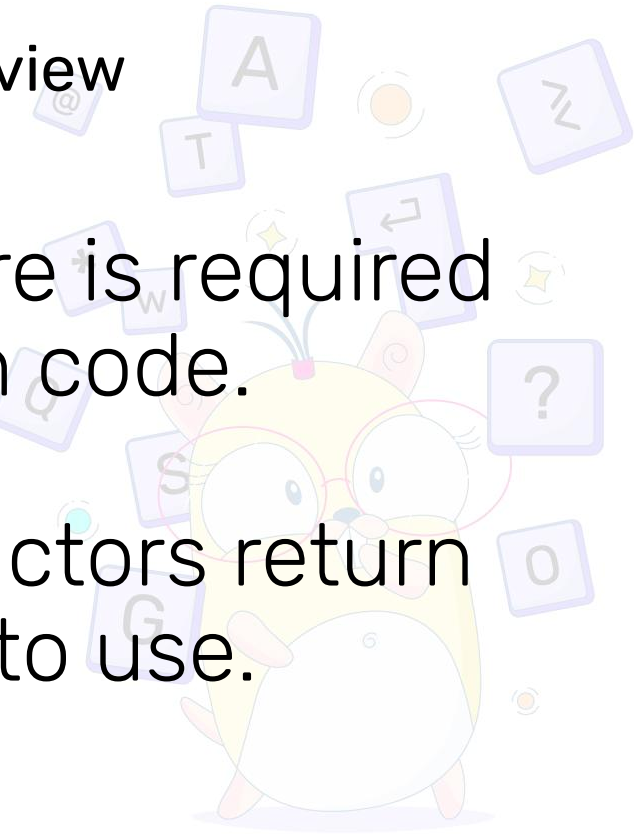
Take a look at github.com/google/safehtml to see this in action.



- Safe and intended uses require no review

No special interaction or care is required for normal application code.

Sanitizers and safe constructors return objects that are easy to use.



Final Notes

In case of failure there are additional safety nets.

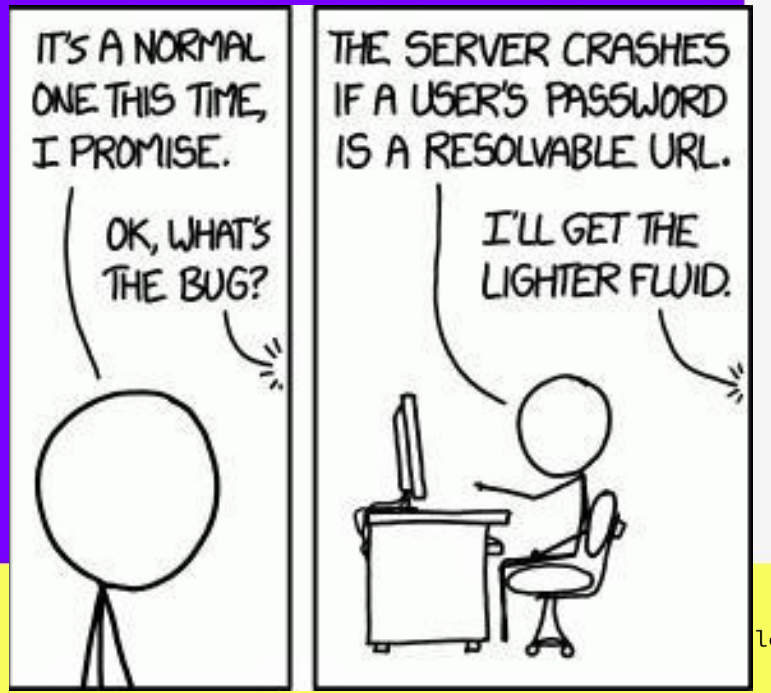
- CSP
- Sandboxes
- Fine grained privileges

The fact that you have compile time guarantees doesn't mean you should stop using other defense mechanisms.

Don't forget UX

- Ux must be part of security
- Frustrating the user can ruin your entire work.
- Errors must be telling.

Benefits



- Early adoptions guarantee safety
- Partial adoptions still give strong guarantees for new code
- Late adoptions can be done gradually
- Low cognitive load, just use a package instead of another
- **Security reviews only need to affect a super small package.**

Many adopters

- Angular
- React
- Go safehtml
- Python Security Manager
- Ruby safe active records
- Most Object-relational mapping libraries
- Closure templates

Piutost che nient, l'è mei piutost.

Honorable mentions

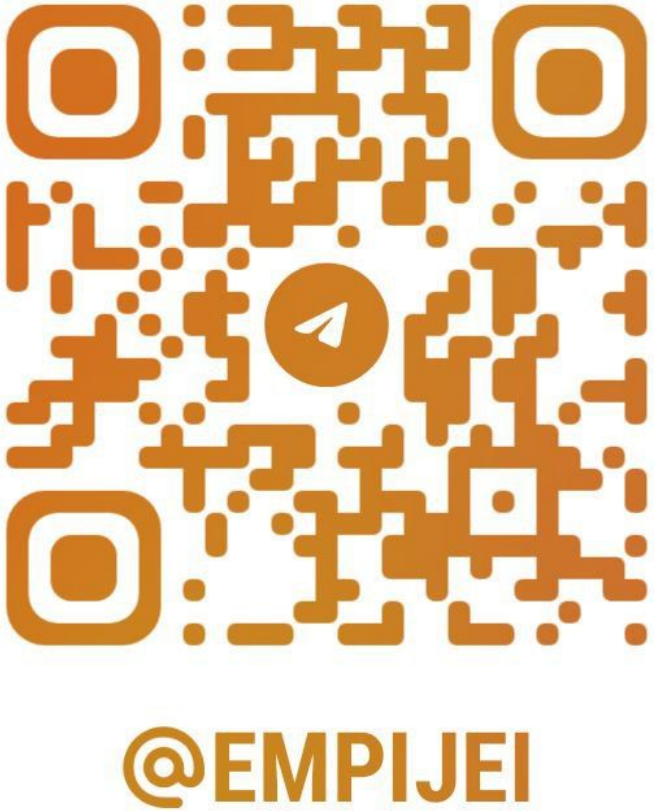
^(?!(xx+)\1+$)xx+$

"0 0\n\n"

- Regular expressions are potential sources of danger
- Parsers should be trustworthy, don't pick them just because they have "fast" in the name
- Don't parse things twice, once parsed, transform data into structs and pass those around.
- Don't copy paste from stackoverflow

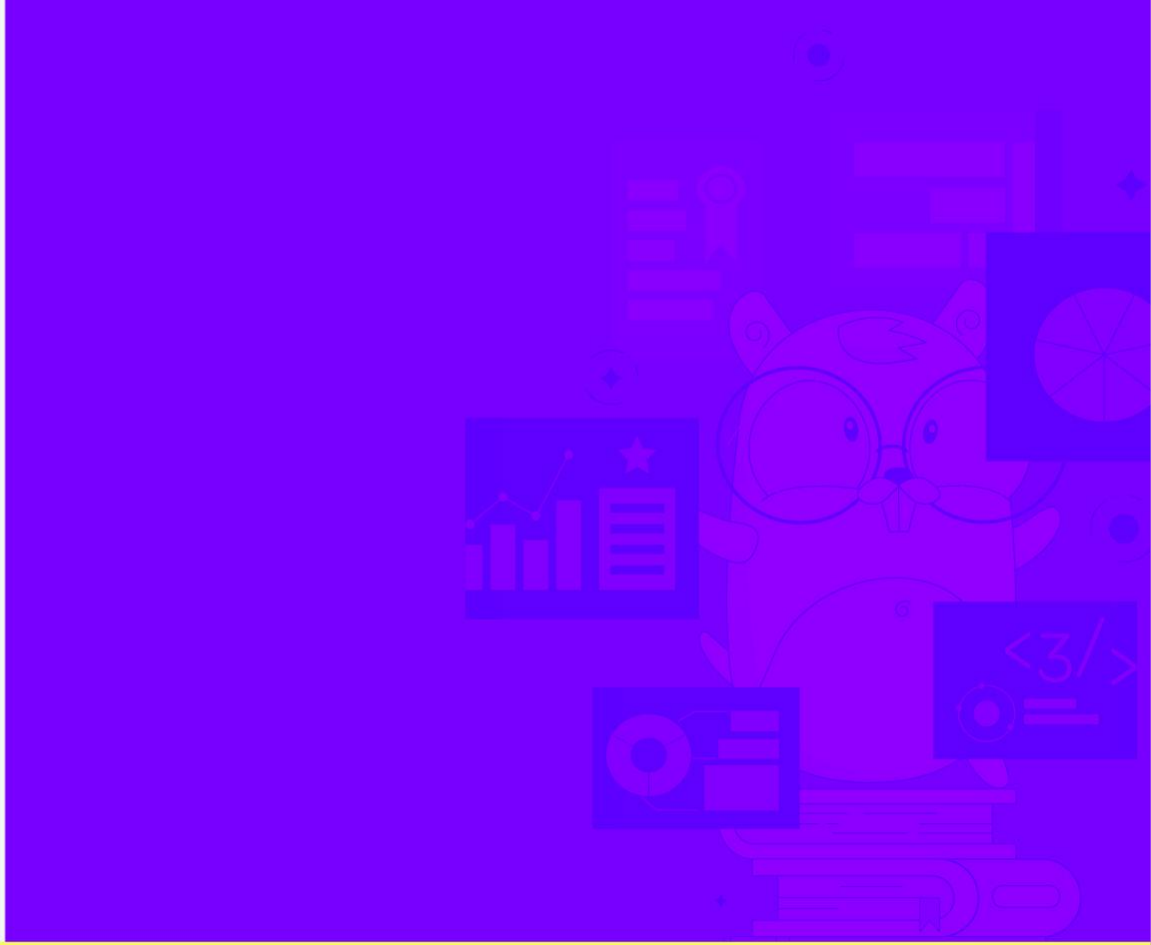
Thank you for listening

- Find me as **empijei** anywhere you like, preferably gmail or telegram
  - empijei@gmail.com
  - t.me/empijei
- **Feel free to contact me** for anything
- I also do freelance **consultancy** if you think your company might be interested
- Link to these slides: **https://t.ly/ECxzE**



Annoyance

Accept language headers



Tricks

Exceptions
The raw package