Effortless Bug Hunting with Differential Fuzzing

`$ whoami`

- 🧑‍🦱 Maciej Mionskowski
- 🔒 Offensive Security Engineer @ Form3
- prev. Platform Engineer, Software Engineer
- ⛵ Sailing, 🧗 Climbing, 🎲 Board Games

What we'll talk about

- Fuzzing
- Differential Fuzzing
- Bugs in the standard library
- Contributing to the standard library
- Fuzzing in CI pipelines

What we'll not talk about

- How fuzzing works under the hood.

Why should you care?

Fuzzing is very effective!

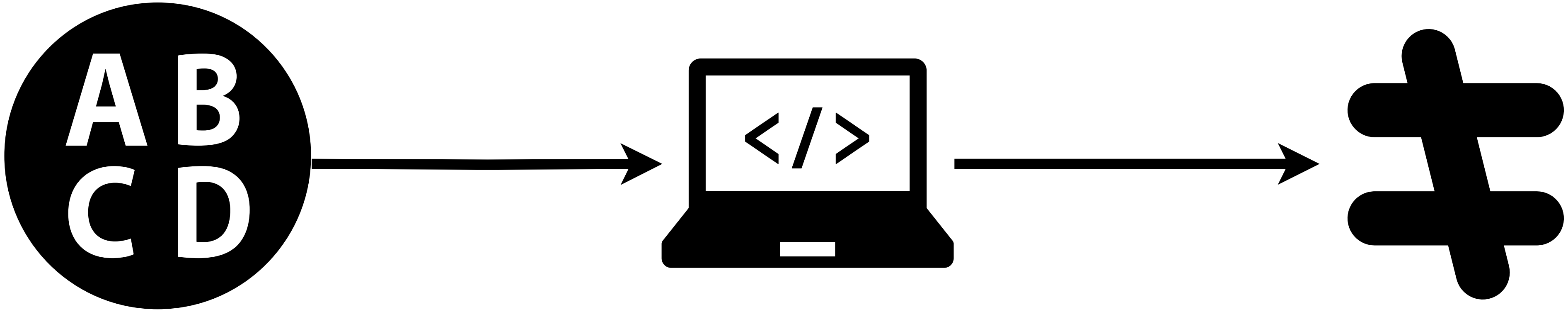
> As of August 2023, OSS-Fuzz has helped identify and fix over **10,000 vulnerabilities** and **36,000 bugs** across **1,000 projects.**

https://google.github.io/oss-fuzz/#trophies

```
func Rot13(in string) string
```

```
a -> n
b -> o
c -> p
```

Regular testing



devised input

function under test

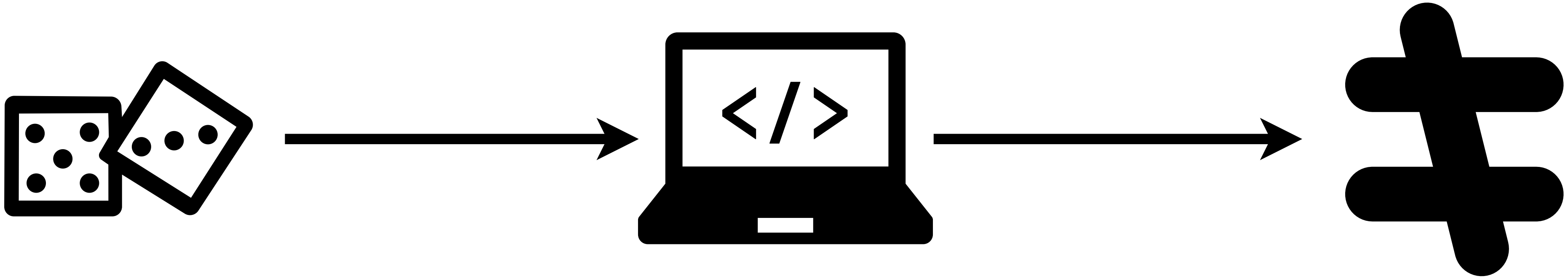
check assertions

Regular testing

```
func TestRot13(t *testing.T) {
    if Rot13("The quick brown fox") != "Gur dhvpx oebja sbk"  {
        t.Fail()
    }
}
```

```
go test -run=TestRot13 .
```

Fuzzing



random input

function under test

check assertions

Fuzzing

```
func FuzzRot13(f *testing.F) {
    f.Add("The quick brown fox")

    f.Fuzz(func(t *testing.T, in string) {
        if Rot13(in) != ????????? {
            t.Fail()
        }
    })
}
```

```
go test -fuzz=FuzzRot13 .
```

```
go test -fuzz=FuzzRot13 -run=^$
fuzz: elapsed: 0s, gathering baseline coverage: 0/11 completed
fuzz: elapsed: 0s, gathering baseline coverage: 11/11 completed
fuzz: elapsed: 1s, execs: 201040 (297903/sec), new interesting: 139 (total: 150)
fuzz: elapsed: 2s, execs: 401040 (268204/sec), new interesting: 130 (total: 250)
```

```
> 200 000 inputs/sec
```

It's easy to create fuzz tests

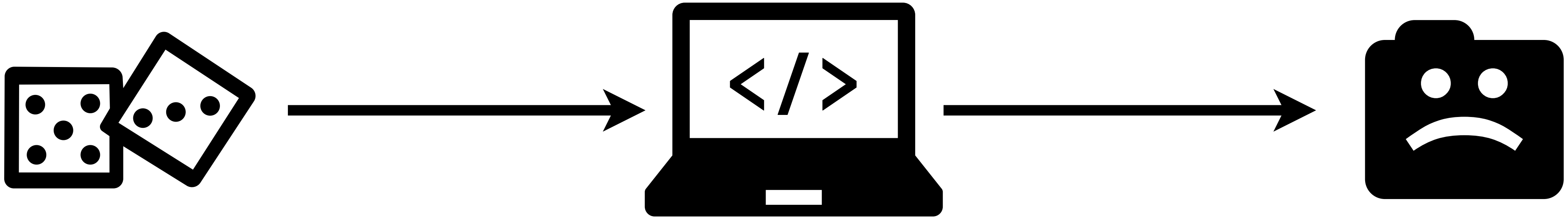
if you have unit tests in place!

The input/The corpus

```go
func FuzzRot13(f *testing.F) {
    f.Add("The quick brown fox")
```

Add inputs from unit tests to the corpus

Fuzzing (commonly)



randomize input

function under test

check if crashed

Fuzzing (commonly)

```go
func FuzzRot13(f *testing.F) {
    f.Fuzz(func(t *testing.T, in string) {

        Rot13(in)

    })
}
```

You can (should) assert on invariants

$$\mathrm{Rot13}(\mathrm{Rot13}(x)) = x$$

$$f^{-1}(f(x)) = x$$

Fuzzing (with an inversible function)

```go
func FuzzRot13(f *testing.F) {
    f.Fuzz(func(t *testing.T, in string) {

        if Rot13(Rot13(in)) != in {
            t.Fail()
        }

    })
}
```

Inversible examples

`Encode()`/`Decode()`

`Marshal()`/`Unmarshal()`

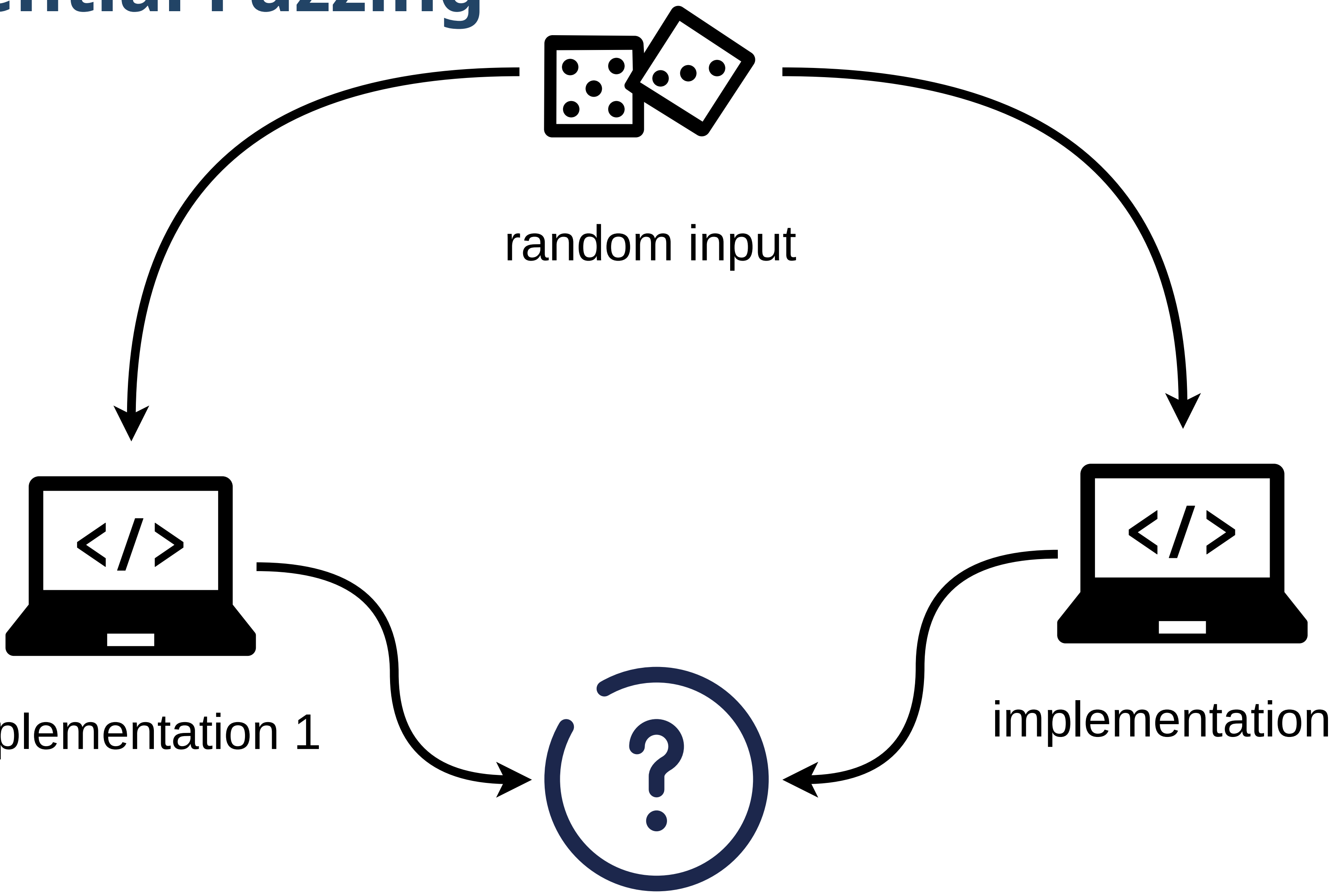
Other examples

```
len(SHA256(x)) == 32
```

Other examples

```go
func FuzzRot13(f *testing.F) {
    f.Fuzz(func(t *testing.T, in string) {

        if Rot13(in) != otherimpl.Rot13(in) {
            t.Fail()
        }

    })
}
```

Differential Fuzzing



random input

implementation 1

implementation 2

check for disagreement

2nd implementation?

Refactoring

If you're refactoring code.

You can keep your old implementation to verify the refactored one.

Performance

You're might be maintaining two implementations where:

- The 1st is written close to some spec, but might be inefficient
- 2nd one is fast, but the code is heavily optimized and unclear

There's a C library that does a similar thing

And you can use CGO to call it.

Case Study

`x/net/html.Tokenizer`

What does a HTML tokenizer do?

`<p>text</p><a>`

=>

```
StartTag - p
Text - text
EndTag - p
StartTag - a
```

How and where is it defined?

https://html.spec.whatwg.org/multipage/parsing.html#tokenization

- Well defined, high in detail
- **It's a state machine**

How is it implemented in Go?

https://github.com/golang/net/blob/master/html/token.go

- **It's not a state machine**
- Not quite easy to understand

```
func Tokenize(input string) (tokens []Token) {
    tok := html.NewTokenizer(input)
    for {
        tt := tok.Next()
        // ...
        tokens = append(tokens, tt.Token())
        // ...
    }
    return
}
```

Fuzzing x/net/html

```go
func FuzzTokenize(f *testing.F) {
    f.Add("<p>text</p><a>")
    f.Add("<!-- command --><body><script>alert(1)</script>")

    f.Fuzz(func(t *testing.T, input string) {
        Tokenize(input)
    })
}
```

No results

It doesn't crash.

Differential fuzzing x/net/html

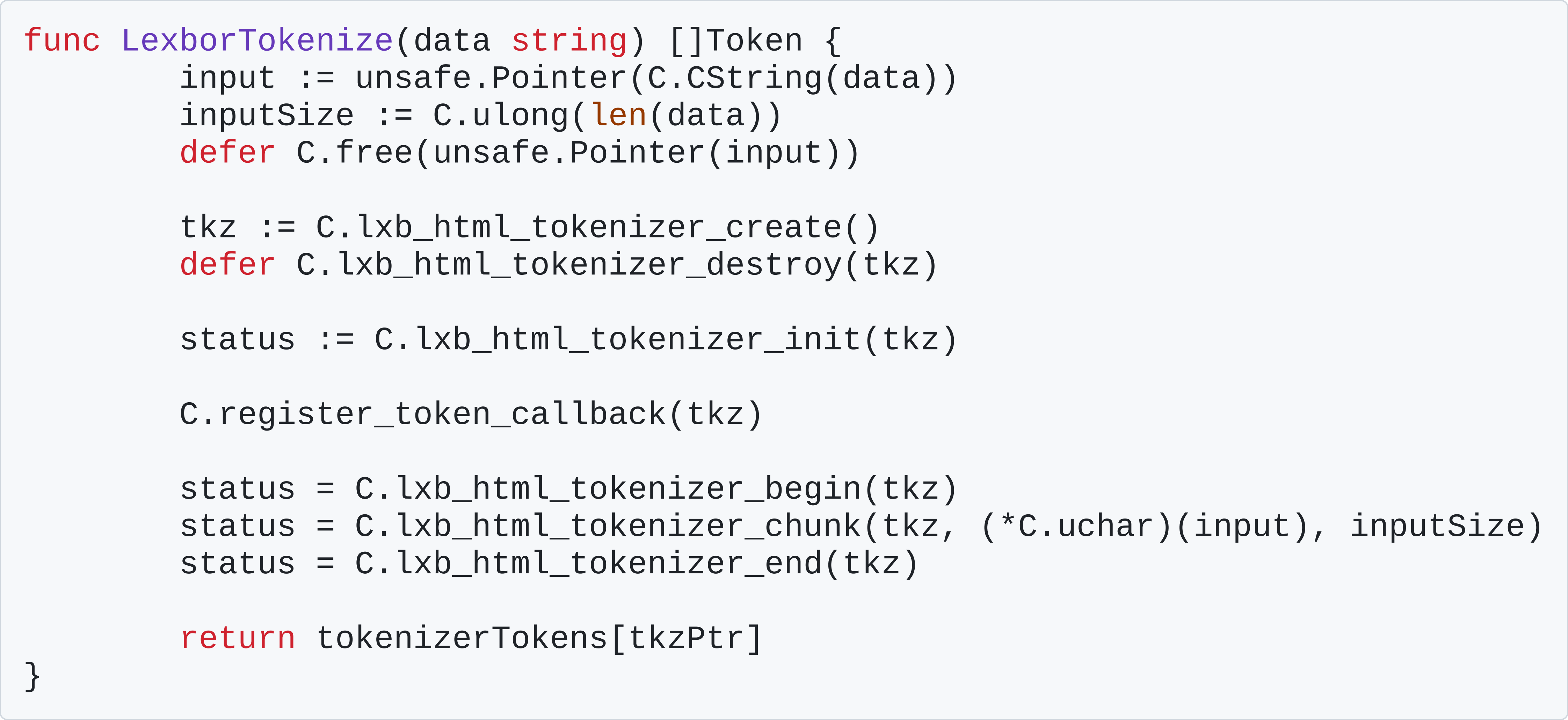
```go
func FuzzTokenize(f *testing.F) {
        f.Fuzz(func(t *testing.T, input string) {
                netTokens := Tokenize(input)
                altImplTokens := altimpl.Tokenize(input)

                if netTokens != altImplTokens {
                        t.Fail()
                        return
                }
        })
}
```

There's C library for that

Lexbor

> We build a web browser engine available as a software library; it ships under the Apache 2.0 license and has no extra dependencies.

https://github.com/lexbor/lexbor



```go
func LexborTokenize(data string) []Token {
        input := unsafe.Pointer(C.CString(data))
        inputSize := C.ulong(len(data))
        defer C.free(unsafe.Pointer(input))

        tkz := C.lxb_html_tokenizer_create()
        defer C.lxb_html_tokenizer_destroy(tkz)

        status := C.lxb_html_tokenizer_init(tkz)

        C.register_token_callback(tkz)

        status = C.lxb_html_tokenizer_begin(tkz)
        status = C.lxb_html_tokenizer_chunk(tkz, (*C.uchar)(input), inputSize)
        status = C.lxb_html_tokenizer_end(tkz)

        return tokenizerTokens[tkzPtr]
}
```

https://github.com/maciekmm/go-std-lib-fuzz/blob/master/lexbor.go

Differential fuzzing x/net/html

```go
func FuzzTokenize(f *testing.F) {
        f.Fuzz(func(t *testing.T, input string) {
                netTokens = Tokenize(input)
                lexborTokens := LexborTokenize(input)

                if !reflect.DeepEqual(netTokens, lexborTokens) {
                        t.Fail()
                        return
                }
        })
}
```

It found something!

```
go test -fuzz=FuzzTokenize -run=^$
fuzz: elapsed: 0s, gathering baseline coverage: 0/11 completed
fuzz: elapsed: 0s, gathering baseline coverage: 11/11 completed, now fuzzing with 20 workers
fuzz: elapsed: 1s, execs: 201040 (297903/sec), new interesting: 139 (total: 150)
--- FAIL: FuzzTokenize (0.68s)
    --- FAIL: FuzzTokenize (0.00s)
        lexbor_test.go:65: length mismatch:
            lexbor      =[{Name:a Type:StartTag}],
            net =[]
             not equal, input: <A =">
```

The finding

```
lexbor = [{Name:a Type:StartTag}],
net    = []
not equal, input: <A =">
```

How do browsers interpret this?

```
<a =">test</a>
```

```
=>
```

```
<script =">alert(1)</script>
```

file://

1

Could this be a security issue?

What if you made trust decisions based on the tokenizer?

```go
func IsSafe(content io.Reader) bool {
	tok := html.NewTokenizer(content)
	for {
		tt := tok.Next()
		switch tt {
		case html.StartTagToken:
			name, hasAttr := tok.TagName()
			if hasAttr || string(name) != "strong" {
				return false
			}
		case html.ErrorToken:
			if tok.Err() == io.EOF {
				return true
			}
			return false
		case html.TextToken, html.EndTagToken:
		default:
			return false
		}
	}
	return true
}
```

What if you made trust decisions based on the tokenizer?

```
case html.StartTagToken:
        name, hasAttr := tok.TagName()
        if hasAttr || string(name) != "strong" {
                return false
        }
case html.TextToken, html.EndTagToken:
        // text is allowed
default:
        return false
```

What if you made trust decisions based on the tokenizer?

```
IsSafe(`<script =">alert(1)</script>`) == true
```

What if you me trust decisions based on the tokenizer?

> Security Considerations
>
> Care should be taken [...] especially with regard to untrusted inputs.
>
> [...]
>
> If your use case requires semantically well-formed HTML, [...] **the parser should be used rather than the tokenizer.**
>
> pkg.go.dev/golang.org/x/...

The parser has the same problem

```
func IsSafe(content io.Reader) bool {
        parsed, err := html.ParseFragment(content, nil)
        if err != nil {
                return false
        }
        for _, el := range parsed {
                if !isNodeSafe(el) {
                        return false
                }
        }
        return true
}

func isNodeSafe(node *html.Node) bool {
        if node == nil {
                return true
        }
        if len(node.Attr) != 0 {
                return false
        }
        if node.Type == html.ElementNode {
                // Parse and ParseFragment will inject html, head, and body.
                // We'll allow these tags for the sake of simplicity, you'd normally want to filter them out.
                if node.Data != "strong" && node.Data != "html" && node.Data != "head" && node.Data != "body" {
                        return false
                }
        }
        return isNodeSafe(node.NextSibling) && isNodeSafe(node.FirstChild)
}
```

The parser has the same problem

```
IsSafe(`<script ="">alert(1)</script>`) == true
```

The parser has the same problem

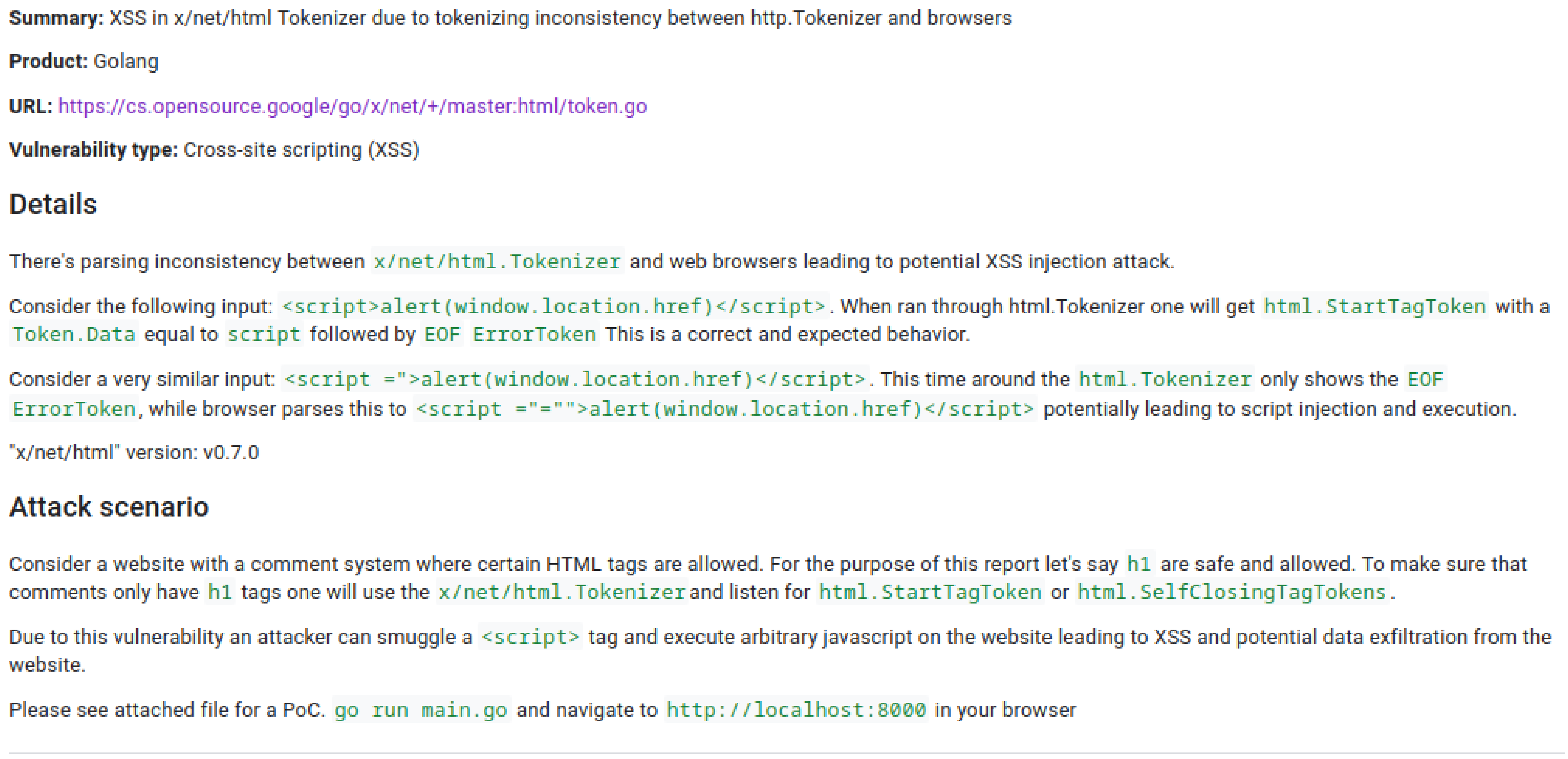
file://

1

1. **The documentation could be improved**

2. **There's a bug in the tokenizer**

Google Vulnerability Report Program



Summary: XSS in x/net/html Tokenizer due to tokenizing inconsistency between http.Tokenizer and browsers

Product: Golang

URL: https://cs.opensource.google/go/x/net/+/master:html/token.go

Vulnerability type: Cross-site scripting (XSS)

Details

There's parsing inconsistency between `x/net/html.Tokenizer` and web browsers leading to potential XSS injection attack.

Consider the following input: `<script>alert(window.location.href)</script>`. When ran through html.Tokenizer one will get `html.StartTagToken` with a `Token.Data` equal to `script` followed by `EOF` `ErrorToken` This is a correct and expected behavior.

Consider a very similar input: `<script =">alert(window.location.href)</script>`. This time around the `html.Tokenizer` only shows the `EOF` `ErrorToken`, while browser parses this to `<script ="="">alert(window.location.href)</script>` potentially leading to script injection and execution.

"x/net/html" version: v0.7.0

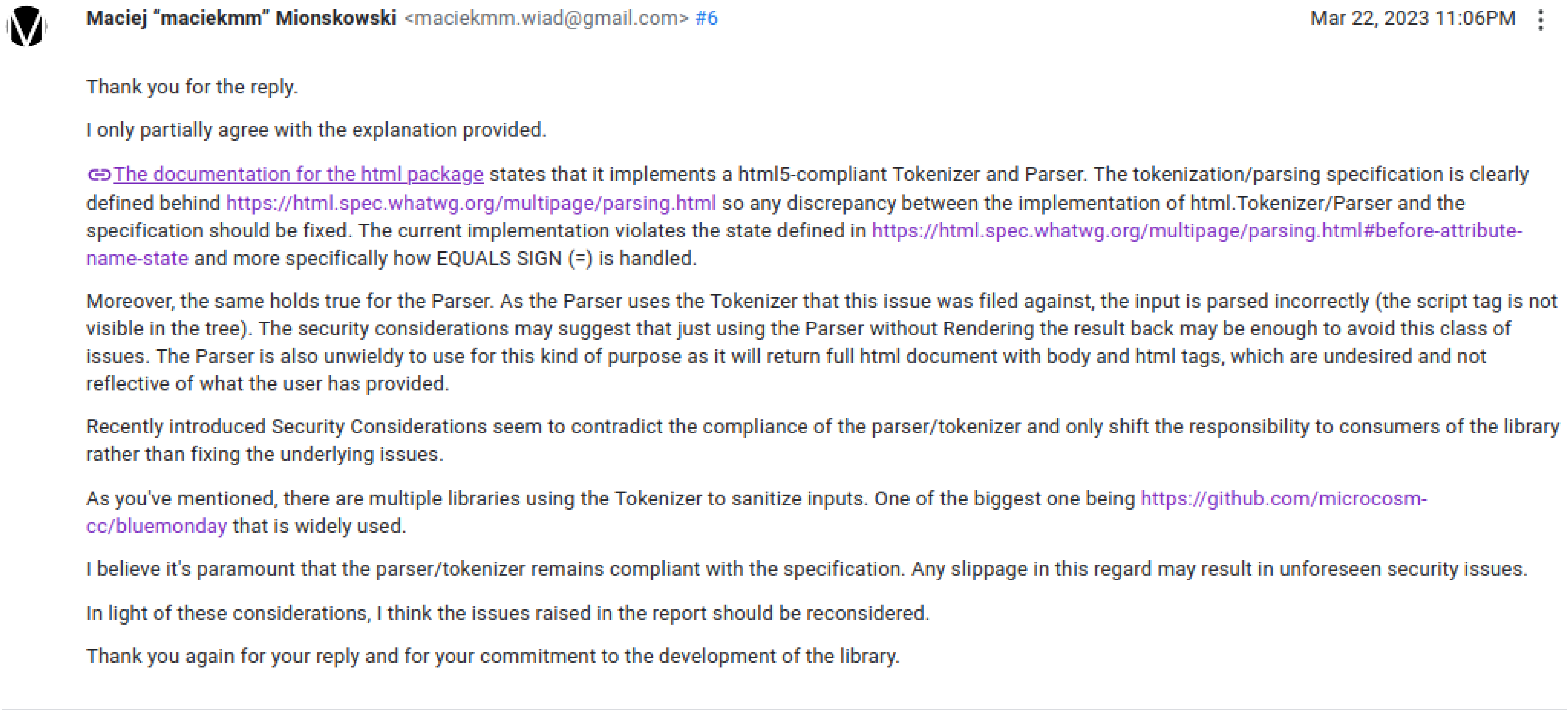
Attack scenario

Consider a website with a comment system where certain HTML tags are allowed. For the purpose of this report let's say `h1` are safe and allowed. To make sure that comments only have `h1` tags one will use the `x/net/html.Tokenizer` and listen for `html.StartTagToken` or `html.SelfClosingTagTokens`.

Due to this vulnerability an attacker can smuggle a `<script>` tag and execute arbitrary javascript on the website leading to XSS and potential data exfiltration from the website.

Please see attached file for a PoC. `go run main.go` and navigate to `http://localhost:8000` in your browser

Google Vulnerability Report Program



Maciej “maciekmm” Mionskowski <maciekmm.wiad@gmail.com> #6

Mar 22, 2023 11:06PM

Thank you for the reply.

I only partially agree with the explanation provided.

The documentation for the html package states that it implements a html5-compliant Tokenizer and Parser. The tokenization/parsing specification is clearly defined behind https://html.spec.whatwg.org/multipage/parsing.html so any discrepancy between the implementation of html.Tokenizer/Parser and the specification should be fixed. The current implementation violates the state defined in https://html.spec.whatwg.org/multipage/parsing.html#before-attribute-name-state and more specifically how EQUALS SIGN (=) is handled.

Moreover, the same holds true for the Parser. As the Parser uses the Tokenizer that this issue was filed against, the input is parsed incorrectly (the script tag is not visible in the tree). The security considerations may suggest that just using the Parser without Rendering the result back may be enough to avoid this class of issues. The Parser is also unwieldy to use for this kind of purpose as it will return full html document with body and html tags, which are undesired and not reflective of what the user has provided.

Recently introduced Security Considerations seem to contradict the compliance of the parser/tokenizer and only shift the responsibility to consumers of the library rather than fixing the underlying issues.

As you've mentioned, there are multiple libraries using the Tokenizer to sanitize inputs. One of the biggest one being https://github.com/microcosm-cc/bluemonday that is widely used.

I believe it's paramount that the parser/tokenizer remains compliant with the specification. Any slippage in this regard may result in unforeseen security issues.

In light of these considerations, I think the issues raised in the report should be reconsidered.

Thank you again for your reply and for your commitment to the development of the library.

The documentation update

```
+ In security contexts, if trust decisions are being made using the tokenized or
+ parsed content, the input must be re-serialized (for instance by using Render or
+ Token.String) in order for those trust decisions to hold, as the process of
+ tokenization or parsing may alter the content.
```

> **the input must be re-serialized (for instance by using `Render` or `Token.String`)**

A few months pass

Maintainers fix the bug

> **html: handle equals sign before attribute**
>
> **Apply the correct normalization when an equals sign appears before an attribute name (e.g. ‘<tag =>’ -> ‘<tag =="">’), per WHATWG 13.2.5.32.**

https://github.com/golang/net/commit/4050002696905e240612ce01211f8ff46cc35afa

Maintainers fix the bug

```
IsSafe(`<script =">alert(1)</script>`) == false
```

Let's run the fuzz test again

```
$ go test -fuzz=FuzzTokenize -run=^$
fuzz: elapsed: 0s, gathering baseline coverage: 0/434 completed
fuzz: elapsed: 0s, gathering baseline coverage: 434/434 completed, now fuzzing with 20 workers
fuzz: minimizing 37-byte failing input file
fuzz: elapsed: 0s, minimizing
--- FAIL: FuzzTokenize (0.30s)
    --- FAIL: FuzzTokenize (0.00s)
        lexbor_test.go:65: length mismatch:
            lexbor      =[{Name:a Type:StartTag}],
            net =[]
             not equal, input: <A/=">
```

Let's run the fuzz test again

```
lexbor      =[{Name:a Type:StartTag}],
net =[]
 not equal, input: <A/=">
```

I decided to learn the codebase myself

```diff
-       case ' ', '\n', '\r', '\t', '\f', '/':
-           z.pendingAttr[0].end = z.raw.end - 1
-           return
        case '=':
            if z.pendingAttr[0].start+1 == z.raw.end {
                // WHATWG 13.2.5.32, if we see an equals sign before the attribute name
                // begins, we treat it as a character in the attribute name and continue.
                continue
            }
            fallthrough
-       case '>':
+       case ' ', '\n', '\r', '\t', '\f', '/', '>':
+           // WHATWG 13.2.5.33 Attribute name state
+           // We need to reconsume the char in the after attribute name state to support the / character
```

https://go-review.googlesource.com/c/net/+/533518#message-73a79f71c04dc5c1fb75b37f218314bf803cbeaf

No more findings :)

Fuzzing is effective

Differential fuzzing helps you write correct code

Good testing candidates

- parsers
- encoders/decoders
- marshallers/unmarshallers
- complex code in general that can be unit tested

Running fuzz tests in CI

is problematic

`go test -fuzz` invocation can only run one Fuzz target at a time

`ClusterFuzzLite` inadequately supports native Go fuzz tests

- problems with understanding/extracting failing inputs
- inconvenient to run locally

We built `go-ci-fuzz`

> CLI and set of GitHub Actions to help you run Native Go Fuzz Tests in CI.
>
> It's a light wrapper around `go test -fuzz=` supporting multiple test targets.

https://github.com/form3tech-oss/go-ci-fuzz

Drag & Drop GitHub action

```
name: Go CI Fuzz - Scheduled
on:
  workflow_dispatch: {}
  schedule:
    - cron: '0 2 * * *'

permissions:
  contents: read

jobs:
  Fuzz:
    runs-on: ubuntu-20.04
    steps:
    - uses: actions/checkout@b4ffde65f46336ab88eb53be808477a3936bae11 # v4.1.1
    - uses: actions/setup-go@93397bea11091df50f3d7e59dc26a7711a8bcfbe # v4.1.0
      with:
        go-version: stable
    - name: Run fuzzers
      id: build
      uses: form3tech-oss/go-ci-fuzz/ci/github-actions/fuzz@v0.1.1
      with:
        fuzz-time: 30m
        fail-fast: false
```

Let's connect

- https://mionskowski.pl
- hello@mionskowski.pl
- @maciekmm:attendees.fosdem.org

References

- https://go.dev/doc/tutorial/fuzz
- https://github.com/google/oss-fuzz-gen
- https://en.wikipedia.org/wiki/Fuzzing
- https://google.github.io/clusterfuzzlite/
- https://github.com/form3tech-oss/go-ci-fuzz
- https://mionskowski.pl/posts/unmasking-go-html-parser-bug/