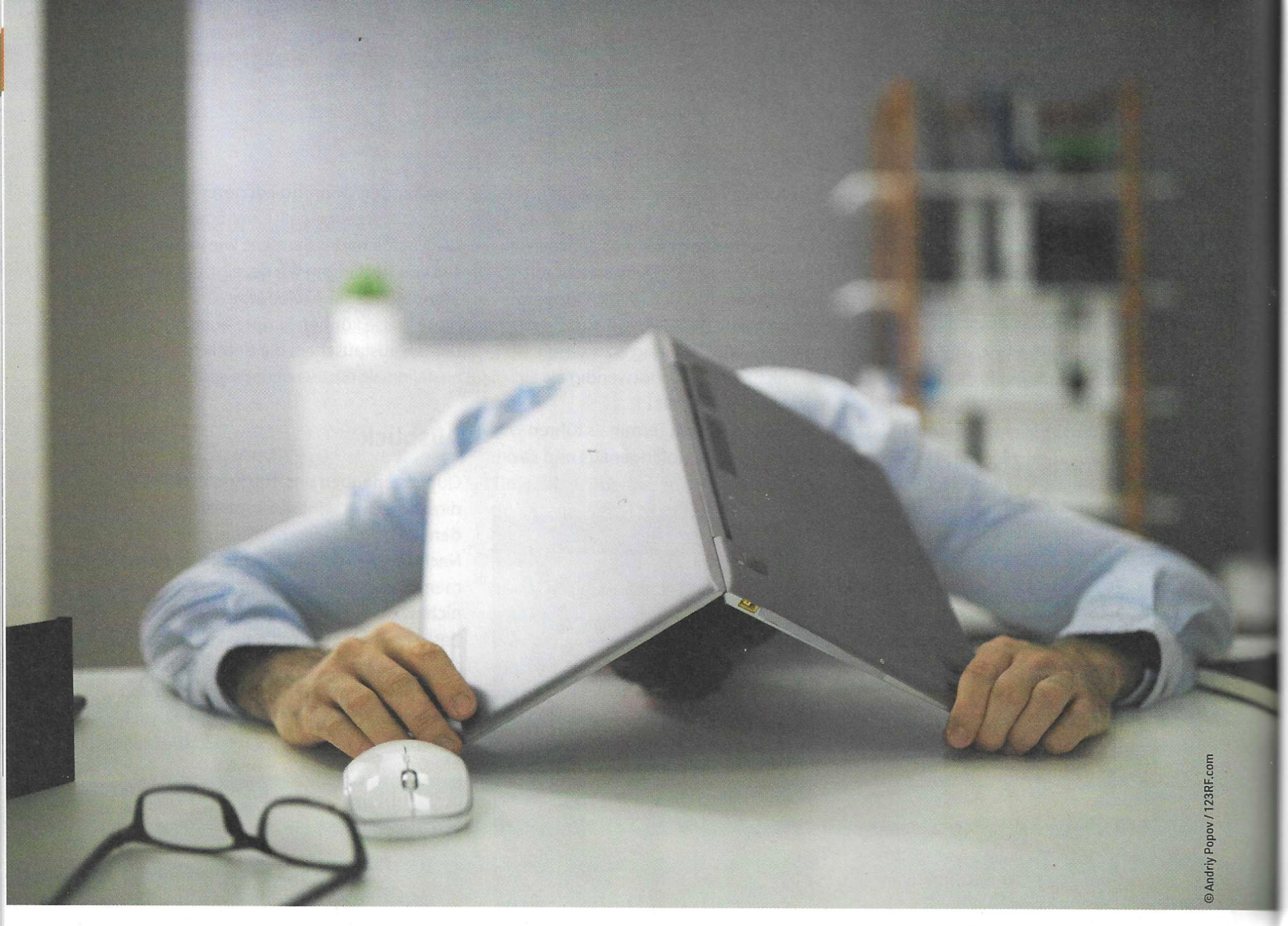


Passwortmanager in Go

Versteckspiel

Eine Go-Applikation für das Terminal hilft Mike Schilli, sich seine Passwörter zu merken. Mike Schilli

Ob auf Notizzetteln unter dem Bildschirm oder in einer kommerziellen Applikation wie OnePass, irgendwo müssen sich Benutzer ihre Passwörter aufschreiben. Die in dieser Ausgabe vorgestellte Go-Applikation für das Terminal legt die sensiblen Daten verschlüsselt auf der Festplatte ab und zeigt nach Eingabe des Master-Passworts ausgewählte Einträge an. Die geheimen Daten hinterlassen nur im Speicher des Rechners Spuren, die sich jedoch nach Schließen des Programms automatisch verflüchtigen.

Findige Anwender könnten nun einfach alle Account-Namen und Passwörter in einer Textdatei ablegen und diese verschlüsseln. Um aber neue Einträge hinzuzufügen, müsste die Datei entschlüsselt und nach dem Editieren wieder verschlüsselt werden. Damit auf der Platte keine Klardaten verbleiben, müsste ein Schrubbbefehl hinterher die gelöschte Datei noch überschreiben. Außerdem kämen nach dem Entschlüsseln sämtliche Passwörter gleichzeitig hoch, prangten prominent auf dem Bildschirm, und ein vorbeieilender Kollege mit Adleraugen könnte dann vielleicht eines oder mehrere davon erhaschen.

Eine Reihe von Passwort-Apps verwaltet Passwörter vorbildlich, aber wer vertraut schon sensible Daten wildfremden Firmen an und verlässt sich darauf, dass diese keine Fehler beim Verschlüsseln oder bei der Verwaltung machen? Abgesehen davon schlagen Apps wie OnePass mit nicht unerheblichen monatlichen Gebühren zu Buche.

Das im Folgenden vorgestellte Programm Password View (pv) verwaltet eine verschlüsselte Kollektion von Passwörtern und zeigt nach Eingabe des Master-Passwords jeweils einen ausgewählten Eintrag in einer Terminal-UI an 1. Sie können dann durch die Einträge scrollen und dort den gewünschten heraussuchen, bevor dessen sensible Daten dann tatsächlich auf Knopfdruck erscheinen.

Auf das Drücken der Eingabetaste hin lässt Pv für den ausgewählten Eintrag die Sternchen verschwinden und enthüllt die geheimen Account- und Passwortdaten. Fahren Sie mit den Cursor-Tasten [K] und [J] (wie bei Vi(m)) in der Liste herauf oder herunter, maskiert Pv den freigegebenen Eintrag dann wieder mit Sternchen. Durch einen Druck auf [Q] falten Sie das Programm zusammen, das dabei das Terminalfenster blankputzt. Keinerlei sensitive Daten verbleiben, und selbst auf der Festplatte steht lediglich noch die verschlüsselte Passwortdatei.

Portabel

Als Go-Binary enthält das Programm bereits alles, was es zur Laufzeit auf ähnlichen Architekturen benötigt. Sie brauchen also lediglich das Binary sowie die verschlüsselte Passwortdatei auf die Systeme zu kopieren, auf denen Sie Zugriff auf die Passwörter wünschen. Mit der Option `--add` aufgerufen, fragt Pv erst nach dem Master-Passwort. Geben Sie es korrekt ein, dürfen Sie am Prompt `New Entry:` eine neue Zeile an die verschlüsselte Datei anfügen (Listing 1).

Dabei ist das erste Wort der zu dem Passwort gehörige Dienst (in unserem Beispiel `gmail`), dessen Name auch im maskierten Zustand einer Zeile in der Benutzerschnittstelle erscheint (wie in der ersten Zeile von Abbildung 1 zu sehen). Der Name dient als Navigationshilfe, um den gesuchten Eintrag zu finden, auszuwählen und anzuzeigen. Der Rest der neu eingefügten Zeile beinhaltet den Benutzernamen und das Passwort. Deren Format können Sie beliebig wählen und zum Beispiel anstelle der vollständigen Daten nur Merkhilfen speichern.

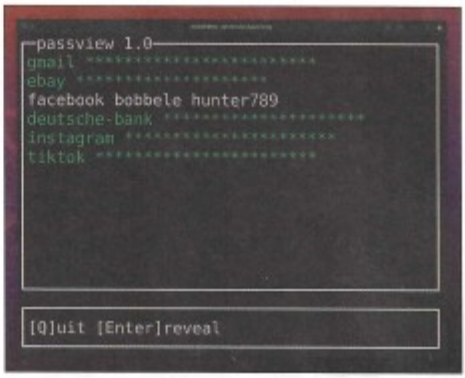
Nach Eingabe der neuen Daten (und auch dann, falls Sie Pv ohne Optionen aufrufen) erscheint dann die Terminal-UI mit der scrollbaren Listbox auf dem Bildschirm, die auf Wunsch die ausgewählten Einträge enthüllt.

Listing 1: Neues Passwort

```
$ pv --add
Password: ***
New entry: gmail bodo@gmail.com
hunter123
```

Tausendsassa

Der Passwort-Safe nutzt eine symmetrische Verschlüsselung. Er verwendet also dasselbe Passwort, um die Datei mit den geheimen Passwörtern sowohl zu ver- als auch zu entschlüsseln. Das Projekt Age auf Github bietet eine von einem Google-Ingenieur geschriebene fertige Go-Library zum Ver- und Entschlüsseln von Daten 2. Sie arbeitet hauptsächlich nach Public-Key-Verfahren, aber auch symmetrische Verschlüsselung steht auf dem Programm. Laut Projektseite spricht man Age wie das italienische Wort „aghe" (Nadeln) aus.



1 **Der Passwort-Speicher Pv in Aktion.**

Symmetrisch verschlüsselt

Für eine Passwortdatei, auf die stets nur ein Benutzer zugreift, stellt eine symmetrische Verschlüsselung die praktischste Lösung dar. Falls sich jedoch mehrere Personen den Zugriff teilen sollen, lässt sich mithilfe von Public-Private-Schlüssel-Paaren (ebenfalls mit Methoden aus der Age-Library) eine Lösung implementieren, die unterschiedlichen Anwendern den Zugriff auf eine geteilte Datei mit dem jeweils eigenen Passwort erlaubt.

Listing 2 zeigt die beiden später vom Hauptprogramm genutzten Funktionen `writeEnc()` und `readEnc()` für das Ver- und Entschlüsseln der Klartextdaten. Zeile 9 definiert mit `test.age` den Namen der verschlüsselten Passwortdatei auf der Festplatte.

Die Age-Library nutzt zum Schreiben (also zum Verschlüsseln) ein Objekt vom Typ `Recipient`, also einen Empfänger, der verschlüsselte Daten zugeschickt bekommt. Der Aufruf der Funktion `NewScryptRecipient()` in Zeile 11 nimmt als einzigen Parameter das Passwort ent-

2 **Das Projekt Age stellt eine Go-Library zum Ver- und Entschlüsseln von Daten bereit.**

```
-----BEGIN AGE ENCRYPTED FILE-----
YWdlLWVuY3J5cHRpb24ub3JnL3YxCi0+IHNjcnlwdCA2aXRUandQdEJhYmJRR3dT
NjYrczd3IDE4CkhqcXV2TDgyNExFQitnSG8yRmNiZFlTOHVmT1JmVkpETzhGOVhT
V2daajAKLS0tIDB1QXBNaFN5V0VlYnNDNWd6eUJnOEVpZ2JDMzNrbkpwcmpnWC9z
L1VxS3MKcIBeYKU0B9inHLvqp1W6C65gs1+lUYnDuUDA2n27GDsJo18T/b7r7xGg
qPkpYOqdU2R8NupZQtUoezB16ezL5ST0Gjk1J0bsdKblVcha38FH7o3LXKzJKUG7
oWjZjdKVBSI9UFtEPwig/r4VOA6uNe2qvTqgoXVLbCA74N8b6vgPxI4ftIQQ50Ch
uYJ8fMr8F4dsW5/uS9WA6wAWrLGXCK47RXFSIr5xYQUk8wwhLbmGUyVYMU6dgPaW
agdIEllt7jX0zZnZ54j2YtKM5qfBfq+VU3NEdaVa
-----END AGE ENCRYPTED FILE-----
~
~
"test.age" 10 lines --20%--                    2,1           All
```

3 **So sieht die verschlüsselte Passwortdatei auf der Festplatte aus.**

gegen, und Scrypt deutet auf die symmetrische Crypt-Funktion hin, die Age implementiert. Zeile 15 öffnet die Passwortdatei zum Schreiben und legt sie per O_CREATE neu an, falls sie noch nicht existiert. Derselben Option fehlt in der Programmiersprache C auf Unix übrigens das letzte E, dort heißt sie O_CREAT. Ken Thompson, einer der Unix-Gründerväter, wurde einmal gefragt, was er denn besser machen würde, falls er Unix noch einmal zu entwerfen hätte, und er antwortete darauf prompt: „I'd spell 'creat' with an 'e'". Go hat ihm diesen Wunsch jetzt offensichtlich erfüllt.

Panzern zum Transfer

Die Option O_TRUNC staucht eine bereits existierende Passwortdatei auf null zusammen, sodass nachfolgende Print-Befehle sie überschreiben. Der Datensalat in der verschlüsselten Datei könnte einen Editor verwirren, und eventuell wäre ein Transferprogramm bei der Übertragung des Inhalts übers Netz versucht, binäre Sequenzen umzustrukturieren. Deshalb setzt Zeile 20 einen Writer vom Typ armor auf, der quasi eine Panzerung um die Binärdaten legt. Sie erscheinen dann zwar immer noch verschlüsselt im Editor, aber wenigstens als gleich lange Zeilen ohne Escape-Sequenzen 3.

Listing 2: crypto.go

```
01 package main
02 import (
03   "bytes"
04   "filippo.io/age"
05   "filippo.io/age/armor"
06   "io"
07   "os"
08 )
09 const secFile string = "test.age"
10 func writeEnc(txt string, pass string) error {
11   recipient, err := age.NewScryptRecipient(pass)
12   if err != nil {
13     return err
14   }
15   out, err := os.OpenFile(secFile, os.O_
   RDWR|os.O_CREATE|os.O_TRUNC, 0600)
16   if err != nil {
17     return err
18   }
19   defer out.Close()
20   armorWriter := armor.NewWriter(out)
21   defer armorWriter.Close()
22   w, err := age.Encrypt(armorWriter, recipient)
23   if err != nil {
24     return err
25   }
26   defer w.Close()
27   if _, err := io.WriteString(w, txt); err != nil
   {
28     return err
29   }
30   return nil
31 }
32 func readEnc(pass string) (string, error) {
33   identity, err := age.NewScryptIdentity(pass)
34   if err != nil {
35     return "", err
36   }
37   out := &bytes.Buffer{}
38   in, err := os.Open(secFile)
39   if err != nil {
40     return "", err
41   }
42   defer in.Close()
43   armorReader := armor.NewReader(in)
44   r, err := age.Decrypt(armorReader, identity)
45   if err != nil {
46     return "", err
47   }
48   if _, err := io.Copy(out, r); err != nil {
49     return "", err
50   }
51   return out.String(), nil
52 }
```

An den verwendeten Funktionen lässt sich schön der in Go oft verwendete Writer-Mechanismus illustrieren. Ein Writer nimmt immer Daten entgegen und schreibt sie irgendwo hin. So öffnet beispielsweise `OpenFile()` in Zeile 15 eine Datei und gibt einen Writer mit dem Namen `out` zurück. Der Panzerungsmechanismus aus dem Paket *armor* nimmt das Writer-Objekt entgegen und liefert ein eigenes Writer-Objekt `armorWriter` zurück. Dieses wiederum nimmt die Funktion `Encrypt()` in Zeile 22 entgegen und gibt einen weiteren Writer `w` zurück, in den dann Zeile 27 mit `io.WriteString()` hineinzuschreiben beginnt.

Der Code implementiert also eine Verknüpfung von geschachtelten Funktionen, die vage an eine Unix-Pipe erinnert: Am Anfang schreibt man hinein, am Ende purzeln die mehrfach bearbeiteten Ergebnisdaten heraus. Dank des von Go unterstützten Writer-Interfaces müssen sich die Funktionen in der Kette auch keine Gedanken über den Typ der Daten machen, die sie da transportieren: Solange jedes einzelne Glied in der Kette das Writer-Interface unterstützt, läuft alles wie am Schnürchen.

In dem vorliegenden Fall nutzen die Age-Funktionen sogar das `WriteCloser`-Interface, das sowohl `Write()`- als auch `Close()`-Aufrufe kennt. Letztere sind gerade bei gepufferten Ausgaben enorm wichtig: Unterbleibt der `Close()`-Aufruf, wird der Cache am Ende unter Umständen nicht geleert, und es stellt sich am Ziel ruckzuck abgehackter und damit völlig unlesbarer Datensalat ein.

Listing 3: `util.go`

```
01 package main
02 func mask(s string) string {
03   masked := []byte(s)
04   tomask := false
05   for i := 0; i < len(s); i++ {
06     if tomask {
07       masked[i] = '*'
08     } else {
09       masked[i] = s[i]
10     }
11     if s[i] == ' ' {
12       tomask = true
13     }
14   }
15   return string(masked)
16 }
```

Verschlüsselt lesen

Umgekehrt liest `readEnc()` ab Zeile 32 Daten aus der verschlüsselten Passwortdatei aus und gibt sie als Zeichenkette zurück. Dazu nimmt die Funktion das eingegebene Master-Passwort für die symmetrische Verschlüsselung als String entgegen und pimpt den zu einem Identity-Objekt auf, für den späteren Aufruf der Funktion `Decrypt()` in Zeile 44.

Doch auch hier muss der Reader erst einmal die Panzerung der verschlüsselten Daten durchbrechen. Das erledigt der Reader vom Typ `armor` in Zeile 43 mit einem weiteren Reader auf die geöffnete Passwortdatei als Parameter. Um die Daten aus dem Reader des Panzerbrechers auszulesen und zur Rückgabe in einer Zeichenkette abzulegen, saugt Zeile 48 mit `io.Copy()` alle Daten ab und legt sie in den bereitgestellten `bytes`-Puffer `out`. Dessen Methode `String()` macht aus dem Daten-Array einen String, Zeile 51 gibt die somit vorliegenden Klardaten an den Aufrufer zurück.

Die Einträge in der Listbox des Passwort-Viewers sollen später nicht alle sofort bei Erscheinen der UI hochkommen. Vielmehr soll man nur das erste Wort jeder Zeile lesen können, während den

Listing 4: `pv.go`

```
01 package main
02 import (
03   "bufio"
04   "errors"
05   "flag"
06   "fmt"
07   "golang.org/x/crypto/ssh/terminal"
08   "os"
09   "strings"
10 )
11 func main() {
12   add := flag.Bool("add", false, "Add new
   password entry")
13   flag.Parse()
14   fmt.Printf("Password: ")
15   password, err := terminal.ReadPassword(int(os.
   Stdin.Fd()))
16   if err != nil {
17     panic(err)
18   }
19   txt, err := readEnc(string(password))
20   if err != nil {
21     if !errors.Is(err, os.ErrNotExist) {
22       panic(err)
23     }
24   }
25   if *add {
26     fmt.Printf("\rNew entry: ")
27     reader := bufio.NewReader(os.Stdin)
28     entry, _ := reader.ReadString('\n')
29     txt = txt + entry
30     writeEnc(txt, string(password))
31     return
32   }
33   lines := strings.Split(strings.TrimSuffix(txt,
   "\n"), "\n")
34   runUI(lines)
35 }
```

Rest Sternchen zieren. Dazu nimmt die Utility-Funktion `mask()` in Listing 3 einen String entgegen, iteriert über dessen Zeichen und ersetzt sie durch einen Asterisk, sofern das Flag `tomask` gesetzt ist. Anfangs ist das nicht der Fall, bis Zeile 11 ein Leerzeichen im String erkennt und der Algorithmus sich daraufhin an der Stelle nach dem ersten Wort wähnt. Dort setzt er `tomask` auf `true` und begräbt den Rest der Zeichenkette unter Sternen.

Die Hauptfunktion `main()` in Listing 4 fragt mit dem Paket *flag* das optionale Kommandozeilenargument `--add` ab. Hat der Benutzer es gesetzt, springt der If-Block in Zeile 25 in den Code ab Zeile 26, der vom User einen neuen Passworteintrag aus der Standardeingabe entgegennimmt und an den Text der entschlüsselten Passwortdatei hängt.

Keine Datei, kein Problem

Dazu hat vorher Zeile 14 mit dem Prompt `Password:` zur Eingabe des Master-Passworts aufgefordert, das Zeile 15 mithilfe des Standardpakets *terminal* und dessen Funktion `ReadPassword()` einliest. Letztere stellt die Standardausgabe auf stumm, also kann der Benutzer das Passwort wie gewohnt blind eintippen.

Stimmt das Passwort nicht mit für die Passwortdatei gesetzten überein, schlägt `readEnc()` in Zeile 19 fehl und `panic()` in Zeile 22 bricht das Programm ab. Schlägt `readEnc()` allerdings fehl, weil die Passwortdatei noch nicht existiert, bekommt Zeile 21 das mit und lässt das Programm laufen, bis weiter unten entweder ein neuer Eintrag angehängt oder die leere Datei in der UI angezeigt wird.

Aus den Zeilen der entschlüsselten Datei entfernt Zeile 33 dann mit `TrimSuffix()` das letzte Newline-Zeichen und spaltet die Zeilen mit `Split()` (beide aus dem Standard-Paket *strings*) in ein Array von Strings auf. Das übergibt es in Zeile 34 an die Funktion `runUI()`, damit diese danach die Benutzerschnittstelle startet. Sie läuft so lange, bis der Anwender sie abbricht und damit das Hauptprogramm endet.

Zwei Widgets

Die Terminal-UI nutzt das Paket *termui* von Github. Listing 5 initialisiert dessen Funktionen in Zeile 12 mit `ui.Init()` und quittiert in der `defer`-Anweisung in Zeile 15 mittels `ui.Close()` einen Abbruch seitens des Users. Das faltet die UI sauber zusammen, damit ein brauchbares Terminal für die Shell zurückbleibt.

Die Benutzerschnittstelle aus Abbildung 1 besteht aus zwei gestapelten Widgets: Oben liegt eine Listbox mit den Passwort-Einträgen, durch die der Benutzer scrollen kann. Sie beherrscht auch ein Paging über mehrere Seiten, falls die Anzahl der Einträge über eine Seite hinauswächst. Am unteren Rand des Terminalfensters klebt ein Paragraph-Widget, das angibt, welche Tasten der User als Nächstes drücken kann: [Eingabe] enthüllt das ausgewählte Passwort, [Q] beendet das Programm.

Damit die UI die gesamte Geometrie des Terminalfensters ausnutzen kann, fragt Zeile 25 dessen Dimensionen mit

Listing 5: `ui.go`

```
01 package main
02 import (
03   "fmt"
04   ui "github.com/gizak/termui/v3"
05   "github.com/gizak/termui/v3/widgets"
06 )
07 func runUI(lines []string) {
08   rows := []string{}
09   for _, line := range lines {
10     rows = append(rows, mask(line))
11   }
12   if err := ui.Init(); err != nil {
13     panic(err)
14   }
15   defer ui.Close()
16   lb := widgets.NewList()
17   lb.Rows = rows
18   lb.SelectedRow = 0
19   lb.SelectedRowStyle = ui.NewStyle(ui.ColorBlack)
20   lb.TextStyle.Fg = ui.ColorGreen
21   lb.Title = fmt.Sprintf("passview 1.0")
22   pa := widgets.NewParagraph()
23   pa.Text = "[Q]uit [Enter] reveal"
24   pa.TextStyle.Fg = ui.ColorBlack
25   w, h := ui.TerminalDimensions()
26   lb.SetRect(0, 0, w, h-3)
27   pa.SetRect(0, h-3, w, h)
28   ui.Render(lb, pa)
29   uiEvents := ui.PollEvents()
30   for {
31     select {
32     case e := <-uiEvents:
33       switch e.ID {
34         case "k":
35           hideCur(lb)
36           lb.ScrollUp()
37           ui.Render(lb)
38         case "j":
39           hideCur(lb)
40           lb.ScrollDown()
41           ui.Render(lb)
42         case "q", "<C-c>":
43           return
44         case "<Enter>":
45           showCur(lb, lines)
46           ui.Render(lb)
47       }
48     }
49   }
50 }
51 func hideCur(lb *widgets.List) {
52   idx := lb.SelectedRow
53   lb.Rows[idx] = mask(lb.Rows[idx])
54 }
55 func showCur(lb *widgets.List, lines []string) {
56   idx := lb.SelectedRow
57   lb.Rows[idx] = lines[idx]
58 }
```

der Hilfsfunktion `TerminalsDimensions()` des *termui*-Pakets ab. Aus der Breite und Höhe des Fensters bestimmen die Zeilen 26 und 27 dann die Lage und Dimensionen der beiden übereinander liegenden Widgets. Dabei erhält das Paragraph-Widget die untersten drei Zeilen, die oben liegende Listbox den Rest. Horizontal breiten sich beide Widgets bis an die Ränder des Terminalfensters aus.

Die Einträge der Listbox sitzen als Array-Slice von Strings im Attribut `Rows` der Listbox. Zeile 17 besetzt es mit dem Array-Slice `rows`. Darin hat die For-Schleife vorher ab Zeile 9 die unmaskierten Originaleinträge aus `lines` (der Inhalt der Passwortdatei im Klartext) abgelegt, und zwar mit Sternchen maskiert. Die zwei Array-Slices für maskierte und unmaskierte Einträge machen es später einfach, maskierte Einträge zu enthüllen: Der Code muss nur auf derselben Indexnummer in den Original-Slice schauen, um die Sternchen wieder zu entfernen.

Nachdem Zeile 28 die Widgets auf den Schirm gebracht hat, feuert Zeile 29 mit `PollEvents()` eine Goroutine ab, die künftig nebenläufig alle Tastendrücke des Users abfängt und in den Channel `uiEvents` schickt. Von dort holt sie die `select`-Anweisung in der endlosen For-Schleife ab Zeile 30 ab und reagiert jeweils sofort auf alle ankommenden Ereignisse. Drückt der Benutzer [K], um nach oben zu scrollen, verhüllt Zeile 35 mit `hideCur()` (ab Zeile 51) und der Funktion `mask()` ein vorher im aktuellen Listbox-Eintrag eventuell schon enthülltes Passwort. Anschließend erteilt `ScrollUp()` der Listbox den Befehl, einen Eintrag nach oben zu scrollen, und das anschließende Render-Kommando zeigt die Veränderung in der UI an. Analoges gilt für den Druck auf [J], mit dem der Nutzer in der Liste nach unten scrollt.

Einen Druck auf die Eingabetaste fängt Zeile 44 ab und ruft die ab Zeile 55 definierte Funktion `showCur()` auf. Sie holt den ursprünglichen, unmaskierten Passworteintrag aus der Liste `lines` und ersetzt die aktuelle Zeile der Listbox damit. Schwuppdiwupp, schon steht das Passwort enthüllt da. `hideCur()` ab Zeile 51 erledigt das Umgekehrte und maskiert den Eintrag mithilfe der Funktion `mask()`.

Installation

Wie immer lässt sich das Binary mit dem typischen Dreisatz (Listing 6) aus dem Go-Code erzeugen. Er holt sämtliche abhängigen Libraries von Github, übersetzt sie und bindet alles zusammen, sodass das fertige Binary pv entsteht. Dieses können Sie anschließend auf jeden Zielrechner mit ähnlicher Architektur kopieren. Es läuft dort klaglos und zaubert die UI praktischerweise auch auf Remote-Maschinen ins Terminal.

Die Passwortdatei `test.age` sollten Sie für den Produktionsbetrieb noch auf eine Datei im Home-Verzeichnis kopieren, dann ist der Passwort-Merker auch schon betriebsbereit. (*uba/jlu*) ■

Listing 6: Programm kompilieren

```
$ go mod init pv
$ go mod tidy
$ go build pv.go crypto.go util.go ui.go
```

Dateien zum Artikel herunterladen unter
www.lm-online.de/dl/47385

Weitere Infos und interessante Links
www.lm-online.de/qr/47385

Der Autor

Michael Schilli arbeitet als Software Engineer in der San Francisco Bay Area in Kalifornien. In seiner seit 1997 laufenden Kolumne forscht er jeden Monat nach praktischen Anwendungen verschiedener Programmiersprachen. Unter mschilli@perlmeister.com beantwortet er gern Ihre Fragen.