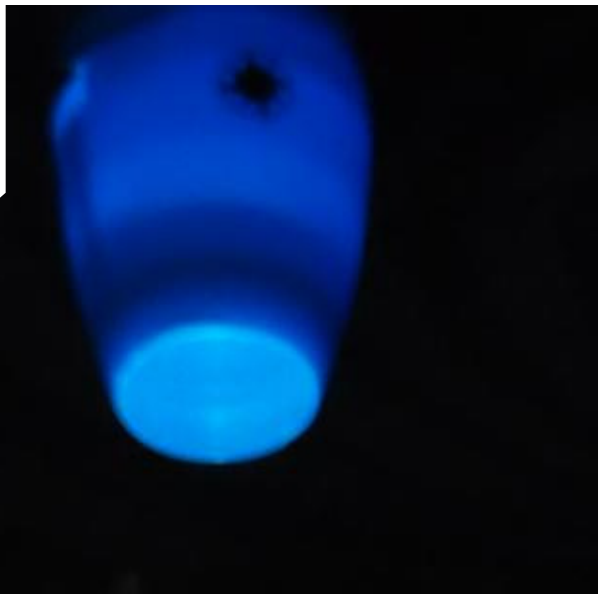
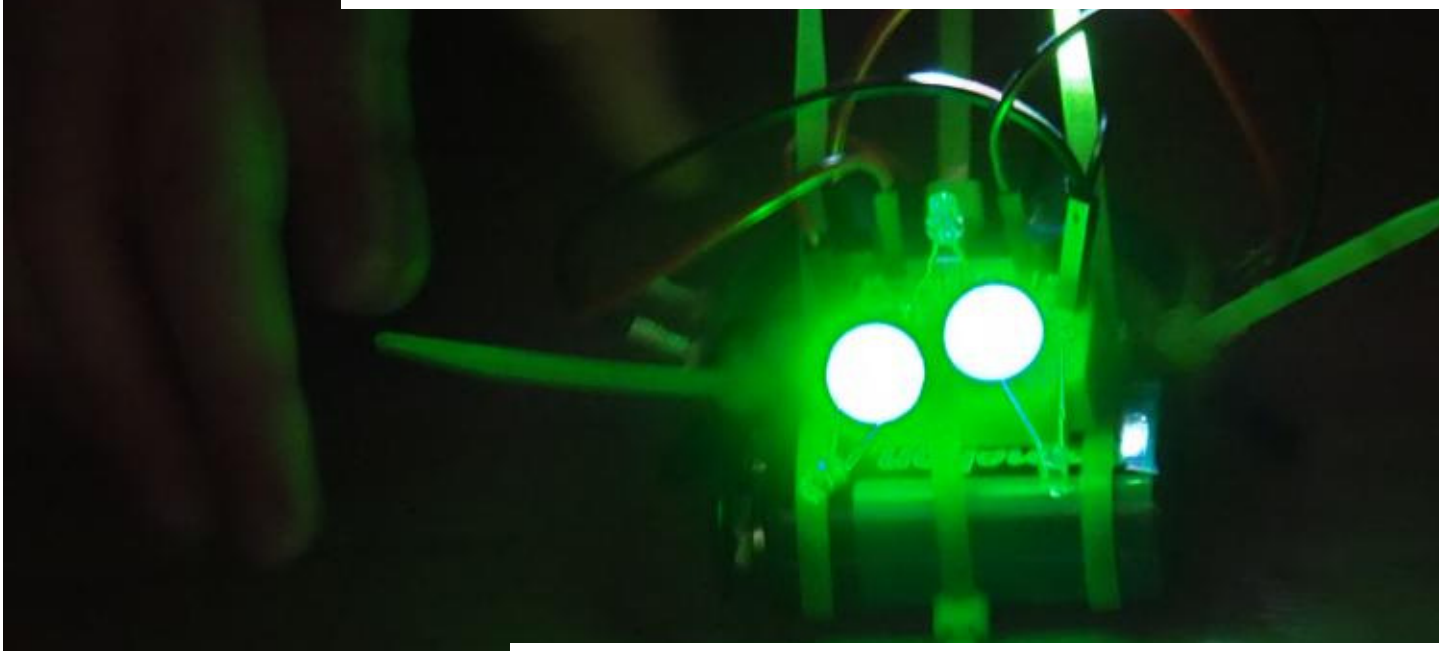
UpTimes

Mitgliederzeitschrift der
German Unix User Group



Kernel-Regressions bekämpfen

Infrastrukturtests automatisieren



Druckersicherheit analysieren

guug

2017-1

Inhaltsverzeichnis

Auf ein Neues: Liebe Mitglieder!
von Ingo Wichmann und Nils Magnus **3**

Real-Life-Forum: Das Frühjahrsfachgespräch 2017
von Anika Kehrer **6**

Feind in meinem Netz: Firmwareanalyse eines Multifunktionsdruckers
von Yves-Noel Weweler **10**

Freiwillige Feuerwehr: Kernel-Regressions bekämpfen
übersetzt von Anika Kehrer **17**

Ein Gral für DevOps: Infrastrukturtests automatisieren
von Christopher J. Ruwe **21**

Shellskripte mit Aha-Effekt VIII: Bash-Arrays
von Jürgen Plate **29**

Support vermarkten: Das Thomas-Krenn-Wiki als Beispiel für Content Marketing
von Anika Kehrer **37**

Hilfreiches für alle Beteiligten: Autorenrichtlinien **43**

Über die GUUG: German Unix User Group e.V. **46**

Impressum **47**

Auf ein Neues
Liebe Mitglieder!

Neues Jahr, neuer Vorstand, neue Pläne: Willkommen bei der Sommer-UpTimes 2017.

von **Ingo Wichmann und Nils Magnus**

Der eine wartet,
dass die Zeit sich wandelt.
Der andere packt sie kräftig an –
und handelt.

Johann Wolfgang von Goethe

Haben nur wir den Eindruck, dass sich unsere Technikwelt in letzter Zeit einmal wieder etwas schneller dreht? Dass die Informationstechnik ein schnelllebiges Geschäft ist, wissen wir alle. Aber in letzter Zeit nimmt sie wieder mächtig Fahrt auf: Konnte man sich vor ein paar Jahren darauf ausruhen, dass es dann und wann einen etwas schnelleren Prozessor und mehr Speicher gab, haben sich mit Virtualisierung, Cloud Computing, Containern und Orchestrierungswerkzeugen zwischenzeitlich völlig andere Perspektiven auf unsere Technik ergeben.

Das Tröstliche dabei: Eine Konstante ist seit Jahrzehnten dabei und nimmt erfreulicherweise sogar wieder an Bedeutung zu – der Geist von Unix. Es macht schon stolz zu sehen, wie ein Satz von Techniktugenden (modularer Aufbau, Kombinierbarkeit von Einzelkomponenten, offene Architektur, herstellerübergreifende Produkte, Fokus auf das Netzwerk) eine so lange Zeit überdauert, egal ob es sich nun um ein BSD, Solaris, Linux oder Android handelt.

Auch Anderes ändert sich und bleibt sich dennoch treu: Der *GUUG*-Vorstand beispielsweise. Nach über 20 Jahren mit kurzen Unterbrechungen im Dienst der Mitgliedervertretung — viele davon als erster Vorsitzender — hat Martin Schulte bei den letzten Vorstandswahlen nicht mehr kandidiert. Wir danken Martin für die umfassende Arbeit und sein immerwährendes Engagement. Das gilt natürlich auch für die anderen ausgeschiedenen Kollegen Julius Bloch und Erwin Hoffmann (der aktuell noch kommissarisch die Kasse führt).

Das neue Team steht im Zeichen der Verbindung von Bewährtem und Neuem: Neben langjährigen GUUG-Mitgliedern hat die Mitgliederversammlung auch Aktive des *LinuxTag* in das Vertretungsorgan berufen. Mit diesem Verein plant die GUUG schließlich auch eine Verschmelzung in diesem Sommer. Sobald die Steuerabschlüsse und rechtlichen Voraussetzungen erfüllt sind, informieren wir Euch über die folgenden Schritte. Mit etwas Glück kann das schon in wenigen Wochen anstehen.

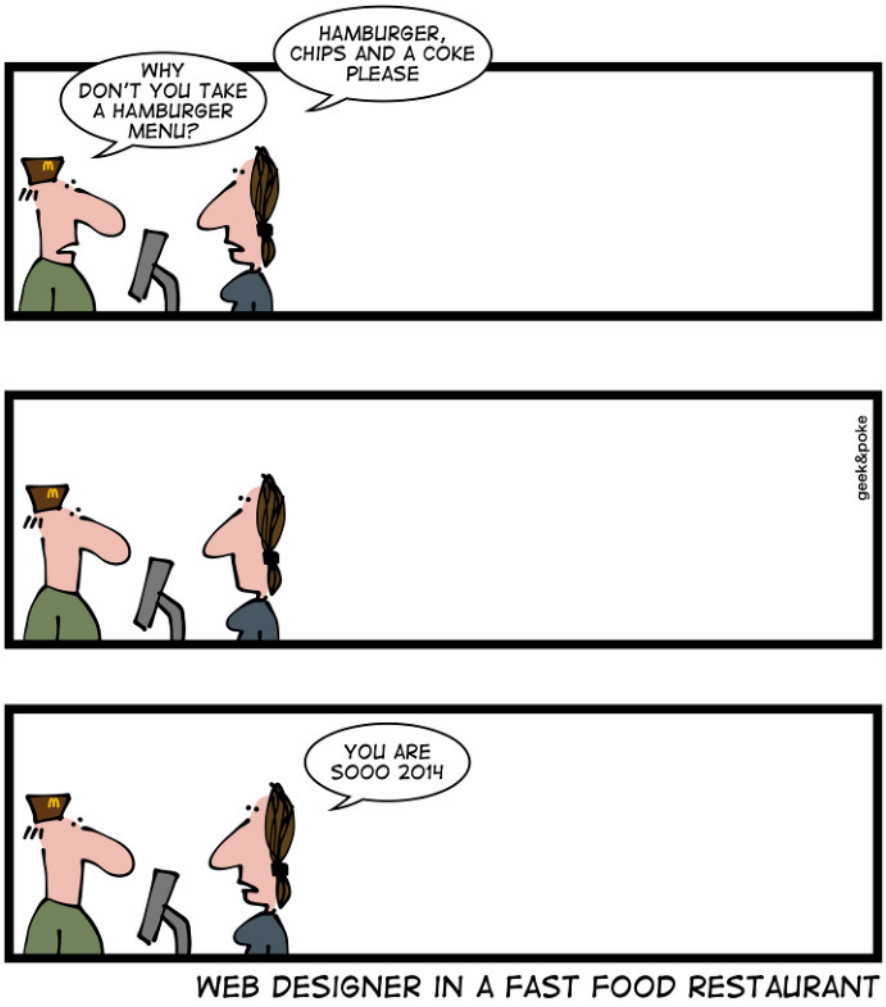
Jetzt jedoch steht die Ferienzeit vor der Tür. Hoffen wir, dass die Zeit sommerlicher Temperaturen mehrheitlich zu kühlenden Besuchen am Badesee oder im Biergarten führt (vormerken: am 29. Juli ist *Admin Appreciation Day*) und weniger zu USV-Failovers im überlasteten Serverraum. Sei die *UpTimes* dabei eine erquickende Lektüre!

Übrigens: Um die UpTimes nach dieser Sommerausgabe nahtlos fortzuführen, sucht die GUUG einen Chefredakteur oder eine Chefredakteurin, der oder die die Fäden in der Hand hält und das Erscheinen der folgenden Ausgaben sicherstellt. Einzelheiten entnehmt bitte dem Kasten *Stellenausschreibung: Chefredaktion für die UpTimes*.

Und es gibt sogar neue Pläne: Wir haben eine Umfrage unter den GUUG-Mitgliedern durchgeführt, ob sie gern die UpTimes auch in der hochverfügbaren Variante, sprich auf Papier in der Hand halten würden. Da wir darauf eine vielversprechende Resonanz erhalten haben, starten wir einen experimentellen Testballon. Wer unter https://guug.de/uptimes-versand bestätigt, dass seine Anschrift noch stimmt, kann die UpTimes in gedruckter Form zugeschickt bekommen. Diesen Beta-Service probieren wir einige Ausgaben lang unverbindlich aus. Die PDF- und ePub-Versionen gibt es weiterhin unverändert zum Download unter http://www.uptimes.de/.

Über Ingo und Nils

Ingo Wichmann ist seit vielen Jahren Mitglied im *German Unix Users Group* e.V. (GUUG) und fungiert seit 2017 als erster Vorsitzender des Vorstands. Das idyllische Konferenzhaus *Linuxhotel* im Essener Ortsteil Horst, in dem Ingo als Geschäftsführer und Referent arbeitet und lebt, wäre ohne ihn nicht dasselbe. Auch als langjähriger Vorsitzender und Veranstalter des *LinuxTag* hat er der Open-Source-Community seinen Dienst erwiesen. Nils Magnus ist Mitgründer des LinuxTag, ebenfalls seit etlichen Jahren GUUG-Mitglied und seit 2014 Mitglied im GUUG-Vorstand. Beruflich verdingt er sich als leitender Security-Consultant, Referent und Fachautor. Als Mitveranstalter beim LinuxTag und bei der GUUG organisiert er seit über 15 Jahren Konferenzen und Workshops. Nils zieht gerade nach Berlin.



Stellenausschreibung: Chefredaktion für die UpTimes

Die German Unix Users group (GUUG e.V.) sucht zum 1. Juli 2017 eine **Chefredaktion** für die *UpTimes*, die Mitgliederzeitschrift der GUUG.

Gesucht ist ein Mitglied oder eine externe Person, die sich der UpTimes, dem Mitgliedermagazin des GUUG e.V., gesamtverantwortlich annimmt. Sie koordiniert und produziert federführend zwei bis vier Ausgaben pro Jahr. Journalistische oder redaktionelle Erfahrung ist sehr nützlich, aber nicht zwingend notwendig. Eine grundsätzliche Affinität zu technischen Themen ist erwünscht, insbesondere rund um unixoide Betriebssysteme und damit verbundener Anwendungen. Zu den Aufgaben zählen:

(1) Artikelakquise und Autorenkorrespondenz: Suchen und Finden von Themen und Autoren.

(2) Textredaktion und Bilderrecherche: inhaltliche und sprachliche Qualitätssicherung sowie Sicherstellen, dass das Magazin ausreichend viele Abbildungen, Autorenbilder und Bildunterschriften enthält.

(3) Zusammenarbeit mit und Förderung der ehrenamtlichen Redaktionsmitglieder: Mailinglisten-Updates, Korrespondenz mit der Redaktion und Außenstehenden, Pflege der Organisationsseite für jede Ausgabe im GUUG-Wiki.

(4) Vorbereiten der UpTimes-Inhalte für ihre Verarbeitung durch die Produktions-Toolchain, die `git` und LaTeX enthält. Für den Satz und die Betreuung der IT-Systeme wird gesorgt.

(5) Weiterentwicklung des Layouts und der grafischen Gestaltung in Absprache mit den dafür zuständigen Mitgliedern.

(6) Zusammenarbeit mit den Organen und Gruppen innerhalb der GUUG: Unterstützung von Anliegen des Vereines unter Wahrung der redaktionellen Unabhängigkeit.

(7) Handling der Autorenabrechnungen in Form von Honorarberechnung und Informationsverteilung an Autoren und Kassenwart. Die Rechnungen selbst schicken die Autoren an den Verein, der auch die Zahlungen abwickelt.

Es sind verschiedene Vergütungsmodelle möglich, etwa auf Stundenbasis oder pauschal pro Ausgabe. Büroeinrichtung und Kommunikationsinfrastruktur organisiert die Chefredaktion selbständig. Die Wahl des Arbeitsortes ist frei wählbar. Der Vorstand vergütet Materialaufwände und Reisekosten nach Absprache, zum Beispiel die Teilnahme am FFG, der hauseigenen Fachkonferenz der GUUG, zu Berichts- und Kontaktzwecken.

Interessierte melden sich bitte zusammen mit einer Beschreibung ihrer Motivation und ihrer redaktionellen Erfahrung bei der Redaktion unter <redaktion@uptimes.de>, die für Fragen zur Verfügung steht.

Dieses Stellengesuch kann gern an geeignete Stellen innerhalb und außerhalb des Vereins weitergereicht werden.

Real-Life-Forum
Das Frühjahrsfachgespräch 2017

Das *FFG* (Frühjahrsfachgespräch) der GUUG fand in seiner 2017-er Ausprägung am 21. bis 24. März in Darmstadt statt.

von **Anika Kehrer**

Es fällt immer auf,
wenn jemand über Dinge redet,
die er versteht.

Helmut Käutner

Ungewöhnlich, aber gut war die Idee, eine Keynote mal an das Ende der Veranstaltung zu legen. Der Informatiker und Journalist Hanno Böck (Abbildung 1) verschaffte somit am frühen Freitagabend des 24. März den bisweilen schon ermatteten Geistern Frischluft, als er die mangelnde Wissenschaftlichkeit der IT-Sicherheit (und in Teilen auch der Wissenschaft an sich) anprangerte. Sein Vortrag war allein schon deswegen anregend, weil man sich für den Ironieanteil seiner so lakonischen wie informativen Ausführungen nochmal neu kallibrieren musste – ein gelungener Event-Abschluss. Böcks Slides sind mit einigen anderen der FFG-Vorträge auf den GUUG-Webseite verfügbar [1].

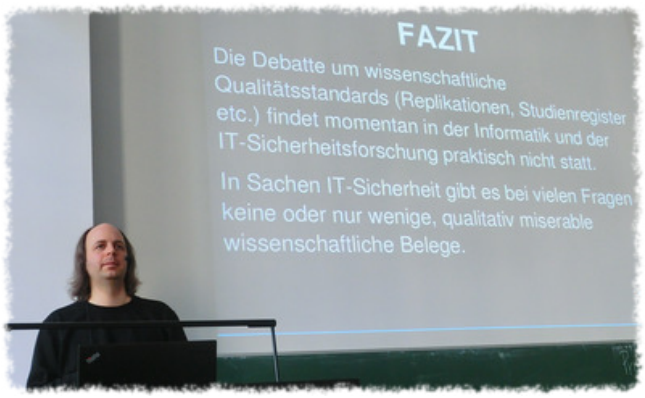


Abbildung 1: IT-Journalist Hanno Böck zog vom Leder. (Fotos: Anika Kehrer)

Mehrfach hatte ja die Community beim diesjährigen Frühjahrsfachgespräch bemängelt, dass die Veranstaltungsankündigung relativ spät fertig geworden war. Es haben in der Folge nicht ganz so viele Konferenzbesucher (rund 60) die steinernen Treppen der Technischen Universität Darmstadt abgenutzt, als es in anderen Jahren auf anderen Universitätstreppen der Republik der Fall gewesen sein mochte. Es wurde aber auch mehrfach angemerkt, dass das Programm sich sehen lassen konnte [2]. Die zwei Vormittags- und Nachmittagsblöcke, parallel in zwei Räumen laufend, waren großteils so geclustert, dass thematisch benachbarte Vorträge aufeinander folgten. Auf den Wortteil ‚Fachgespräch' in ‚Frühjahrsfachgespräch' wirkte sich das förderlich aus: Nicht nur Zuhörer, sondern auch die Referenten kamen in den abschließenden Q&A-Teilen ihrer Vorträge miteinander ins Gespräch.

Software Storage

Man stutzte etwas, als Felix Hupfeld (Abbildung 2) seinen Vortrag *Software Storage – Storage zum Herunterladen* mit einer Übersicht zur Entwicklung der Netzwerkgeschwindigkeit begann. Über Storage-Software und -Hardware in verteilten Systemen ging er zu Ansätzen der Fehlertoleranz in Software über. Es gelte die „magische Zahl 3": Zwei Systeme mit automatischem Failover funktionieren nicht – man brauche ein drittes System, außerdem menschliches Entscheiden im Zweifelsfall. Er landete beim „Storage der nächsten Generation": ein verteiltes System, Software-basiert auf Standard-Hardware, mit Fehlertoleranz und Redundanz in der Software.

Google nannte der Referent als lebenden Beweis, dass Software-Storage funktioniere. Umso wichtiger sei bei so einem verteilten Storage-System natürlich das Netzwerk: Man brauche schnelle Netze. Daher das Referat darüber am Anfang. Da die Netze schneller sind als die Hardware, sei es auch egal, ob das Storage lokal oder remote ist – nur ein paar hundert Nanosekunden Latenz wiesen Ethernet-Switche heute auf.

Abbildung 2: Felix Hupfeld wusste, wovon er redet.

Dass der Referent quasi seine eigene Geschäftsgrundlage verargumentierte (und mithin bewarb) störte bei seinem sachlichen Vortrag kaum. Am Berliner Bahnhof Zoo sitzt nämlich die von ihm und einem Kollegen 2013 gegründete Firma *Quobyte*. Produkt der Software-Firma ist das *Data Center File System* [3], das eine Weiterentwicklung des EU-geförderten verteilten Dateisystems *XtreemFS* [4] für Hardware-entkoppelte Storageinfrastrukturen ist. Dieses wiederum haben Hupfeld und sein Quobyte-Gründerkollege 2006 selbst im Rahmen ihrer Doktorarbeit angestoßen. Der Vortrag war also ein gutes Beispiel dafür, wie sachliche Tiefe – kombiniert mit konzentriertem Sprechen – Genuss für den Hörer schafft, auch wenn der Vortragende finanzieller Stakeholder des Themas ist.

Ceph

Lars Marowsky-Bree (Abbildung 3) arbeitet seit sage und schreibe 17 Jahren bei dem Linux-Distributor *SUSE* und ist dort derzeit Architekt für *Ceph*-basiertes Enterprise-Storage. Naheliegenderweise referierte er über *Ceph Status und Roadmap*. Auch Ceph begann 2004 sein Ideenleben als Doktorarbeit, rief der Referent ins Gedächtnis [5]: Der Doktorand – Sage Weil ist heute bei Red Hat anstellig – wollte ein skalierbares Dateisystem schreiben, und es stellte sich heraus, dass das ganz schön kompliziert ist. Inzwischen skaliert es in der Praxis aber auf mehrere tausend Server (etwa beim *CERN*, [6]). Seine Basis ist *RADOS* (reliable autonomic distributed object store), worauf Daten-Handling für drei Fälle sitzt: Object Storage, Block-Devices, und *CephFS*. CephFS wurde im April 2016 mit der LTS-Version 10.2 (Jewel) von Ceph als stabil erklärt.

Abbildung 3: Lars Marowsky-Bree war begeistert von *BlueStore*.

Begeistert zeigte sich der Referent von *BlueStore* [7], das ab Version 10.2 experimentell an Bord ist, und das insbesondere für Kunden wertvoll sei, weil alles damit zwei mal schneller gehe. In der aktuellen Stable-Release 11.2 (Kraken) von Januar ist BlueStore aber noch als experimentell gekennzeichnet [8]. Die Entwickler arbeiten derzeit an der Fertigstellung der nächsten LTS-Release 12 (Luminous), die für dieses Frühjahr angekündigt war und für die der Referent (wohlgemerkt zum FFG-Zeitpunkt Ende März) angab, dass BlueStore hier nicht nur stabil, sondern auch bereits der Default sein sollte.

Sicher authentisieren

Der *NetKnights*-Sicherheitsberater Cornelius Kölbel (Abbildung 4) schlug eine Bresche für Zweifaktor-Authentisierung. Der normale Anwender, begann er, tendiere ja dazu, ein Passwort einfach immer wiederzuverwenden, das er sich merken kann. Daraus resultiert dann der Unkenruf: Das Passwort ist tot – sprich, nicht sicher genug. Kölbel ist der Meinung, auf ein Passwort zu verzichten sei Quatsch, denn immerhin erschwere es den Angriffsweg. Aber man könnte es zu besseren Sicherheit um eine weiteres Attribut anreichern. Zum Beispiel könnte über die bloße Zeichenkette hinaus eine Rolle spielen, wer das Passwort besitzt, oder welche Merkmale es hat. So landet man bei Zweifaktor-Authentisierung.

Der eine Faktor darf hierbei das gewöhnliche Passwort bleiben. Der zweite Faktor muss sich durch drei Eigenschaften auszeichnen. Erstens: Eindeutigkeit. Er darf also nicht kopierbar sein. Zweitens: Der Verlust soll bemerkbar sein. Bei einem Fingerabdruck wäre das zum Beispiel nicht der Fall. Drittens soll er rückrufbar und neu ausstellbar sein – für biometrische Merkmale gilt das

zum Beispiel nicht. Biometrische Merkmale eignen sich darum zwar für die Identifizierung, also den Benutzernamen, aber nicht als Authentisierungsfaktor, erklärte der Referent. Als Beispiele zweiter Faktoren führte er Einmalpassworte, eine Smartcard und *U2F* ins Feld. Hinter U2F verbirgt sich der so genannte *Universal 2nd Factor*, den die *FiDO Alliance* (Fast iDentity Online, [9]) ausbrütet: Die Idee ist, dass sich die Nutzer von Webdiensten zusätzlich zum Passwort mit einem Stück Hardware authentisieren, um Identitätsmissbrauch zu vermeiden.

Abbildung 4: Cornelius Kölbel sah die Schuld mal nicht nur beim User.

Was genau als zweiter Faktor das Beste ist, wollte Kölbel nicht final entscheiden. Er haderte damit, dass seine eigentlich favorisierte Lösung U2F bei Anwendern schlicht nicht zum Einsatz komme. Das liege aber zur Abwechslung mal nicht am User, sondern an den Herstellern. Das U2F-Device ist nämlich typischerweise ein USB-Stick, und die FiDO Alliance hat eigentlich zum Ziel, die Anmeldung bei Google, Facebook und Consorten sicherer zu machen. Das würde aber bedeuten, dem Browser Zugriff auf die USB-Schnittstelle zu geben – schwierig, schwierig, wog er den Kopf hin und her.

Insbesondere litt der Referent darunter, dass pro Dienst ein separates Schlüsselpaar nötig wird. Das lasse nämlich die Zahl der Schlüssel auf dem Device explodieren. Darum arbeitet der Stick mit einem Masterkey, der auf dem Device liegt und auf dessen Basis die Schlüsselpaare entstehen. Das Problem: Der Masterkey wird vom Hersteller generiert. „Uns wird nicht zugetraut, so etwas selbst zu erzeugen“, zuckt der Referent traurig die Schultern.

Abbildung 5: Gutgelaunte Restmenge der FFG-Besucher 2017 am Freitag gegen 19 Uhr.

Gehet hin und redet

Das Berichtete sind natürlich nur Schlaglichter, relativ willkürlich herausgegriffen aus rund 30 Vorträgen des FFG 2017. Nie ist ein Vortrag so gut wie der andere, und immer stellt sich bei einem Event jedweder Art die Frage, ob man wirklich dagewesen sein muss. Neben seinem deutlichen fachlichen und eventplanerischen Anspruch ist das FFG aber stets auch ein Forum, in dem sich GUUG-Mitglieder und Artverwandte treffen. Einige kennen sich seit Jahrzehnten – doch mittlerweile treten auch Jüngere wieder aufs Parkett (Abbildung 5).

Abbildung 6: Geselliger Abend des FFG 2017.

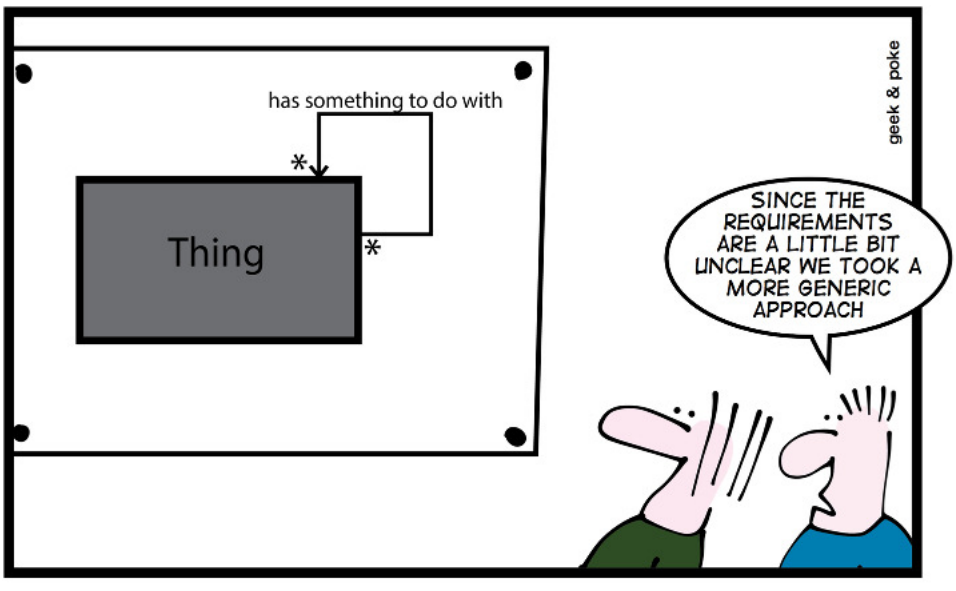
Der Schlüssel des Ganzen ist Reden. Gehet also hin und redet: Auf dem Social Event (Abbildung 6), in den Q&A-Sessions, auf dem Flur . Zur Not redet mitten in einen Vortrag hinein! Letzteres aber am liebsten dann in solcher Güte, dass es Beifall statt Brummen verdient.

Links

[1] Verfügbare Vortragsfolien des *FFG* 2017: https://guug.de/veranstaltungen/ffg2017/slides.html
[2] Programm des *FFG* 2017: https://guug.de/veranstaltungen/ffg2017/programm.html
[3] *Data Center File System*: https://www.quobyte.com/de/product
[4] *XtreemFS*: http://www.xtreemfs.org
[5] Sage A. Weil: Ceph; Reliable, scalable, and high-prformance distributed storage. Dissertation, Santa Cruz 2007: https://ceph.com/wp-content/uploads/2016/08/weil-thesis.pdf
[6] Aufzeichnung des Vortrags *Ceph beim CERN* von Dan van der Ster beim Openstack-Summit Vancouver 2015: https://www.openstack.org/videos/vancouver-2015/ceph-at-cern-a-year-in-the-life-of-a-petabyte-scale-block-storage-service
[7] Vortragsfolien *BlueStore, a new storage backend for Ceph* von Sage Weil im Jahr 2017: https://de.slideshare.net/sageweil1/bluestore-a-new-storage-backend-for-ceph-one-year-in
[8] Release Notes von *Ceph*: http://docs.ceph.com/docs/master/release-notes/
[9] *FiDO Alliance*: https://fidoalliance.org

Über Anika

Anika Kehrer arbeitet seit rund zehn Jahren als Technikjournalistin für verschiedene Medien und ist seit der Wiederauflage der *UpTimes* im Sommer 2012 als Chefredakteurin an Bord. Ihre Arbeit ist auszugsweise auf der Webseite https://www.torial.com/anika.kehrer zu sehen. Auf Google Plus (https://plus.google.com/114887154907909509357) reichert sie die digitale Öffentlichkeit um subjektiv Relevantes an. Manchmal auch ironisch.



Feind in meinem Netz
Firmwareanalyse eines Multifunktionsdruckers

Sicherheits- und Erkenntnisinteresse führten zu diesem Artikel, der Schritt für Schritt durch die Firmwareanalyse des Epson-Multifunktionsdruckers WF-2540 geht.

von **Yves-Noel Weweler**

Du willst nicht zornsüchtig sein?
Gut, dann sei nicht neugierig.

Seneca

Die Glücklichen sind neugierig.

Nietzsche

Ein Epson WF-2540-Multifunktionsdrucker [1] war gerade zur Hand, sodass die Untersuchungen auf diesem Gerät erfolgten. Um an die Firmware eines Gerätes zu kommen, gibt es in der Regel folgende Optionen:

- Firmware beim Update abfangen,
- Firmware aus Hardware auslesen,
- Firmware beim Hersteller herunterladen.

Viele Geräte haben eine Aktualisierungsfunktion, bei der das Gerät selbst die Firmware herunterlädt und installiert. Der Netzwerkverkehr lässt sich zum Beispiel mit *Wireshark* [2] aufzeichnen und dann nach Firmwaredateien durchsuchen. Alternativ liest man die Firmware aufwendig aus der Hardware eines Gerätes aus. Dazu später mehr. Denn glücklicherweise bieten die meisten Hersteller Firmware und Updatetools zum Download an, sodass diese nur noch heruntergeladen werden müssen.

Für den WF-2540 stellt Epson einen Firmwareinstaller für Microsoft-Windows-Betriebssysteme bereit. Diese .exe-Datei ist dazu da, die Firmware eines Gerätes über USB oder über das Netzwerk zu aktualisierten. Um an die Firmware zu gelangen, die sich irgendwo im Installer verbirgt, entpacken wir ihn:

```
$ gunzip installer.exe
e_dge321.dll
ENBoost.dll
ENWA.ini
EPFWUPD.exe
FWG658TL.efu
iconv.dll
Resources/
unicows.dll
Unzip32.dll
```

Ins Auge sticht die Datei `FWG658TL.efu`. Mit etwas Fantasie könnte die Dateiendung für ‚Epson Firmware Update' stehen. Da wir nicht wissen, womit wir es zu tun haben, versuchen wir mit *file* herauszufinden, um was für eine Art von Datei es sich handelt:

```
$ file FWG658TL.efu
FWG658TL.efu: Zip archive data, at least v2.0 to extract
```

Entpacken wir das ZIP-Archiv, kommt eine ominöse Datei mit der Endung .rcx zum Vorschein:

```
$ unzip FWG658TL.efu
Archive:  FWG658TL.efu
inflating: FWG658TL.rcx
```

In Erwartung einer Binärdatei öffnen wir sie mit einem Hexeditor, wobei sich herausstellt, dass sie mit einem Textteil beginnt. Sieht man sich die .rcx-Datei dann in einem Texteditor an, ähnelt ihr Anfang einer *INI*-Datei. Darauf folgen jedoch mehrere Megabyte Binärdaten. In den Einträgen der INI-Datei finden sich außerdem das Wort ‚Firmware' und sogar die vom Hersteller für den Installer angegebene Firmwareversion. Die Firmware haben wir also schon einmal gefunden.

Updatemechanismus nutzen

Wie aber wird die Firmware an das Gerät übermittelt, und wie wird mit ihr verfahren? Um das herauszufinden, probieren wir das netzwerkbasierte Firmwareupdate des Installers aus. Das mit Wireshark aufgezeichnete Update sieht aus wie in Abbildung 1.

Im Netzwerkverkehr finden sich keine Verbindungen zum Hersteller. Die Firmware ist also komplett im Installer enthalten. Das Update selbst erfolgt unverschlüsselt über HTTP in zwei Stufen. Es bereitet den Drucker zunächst mit einer

GET-Nachricht auf das Update vor. Mit Erhalt der Nachricht zeigt der Drucker auf dem Display an, dass ein Update stattfindet. Durch eine *POST*-Nachricht schickt der Updatemechanismus in einem zweiten Schritt Binärdaten an den Drucker. Dabei überträgt er die vorher gefundene .rcx Datei, aber ohne den INI-Abschnitt am Dateianfang.

Source	Destination	Protocol	Info
laptop	printer	HTTP	GET /FIRMWAREUPDATE HTTP/1.1
printer	laptop	HTTP	HTTP/1.1 200 OK
laptop	printer	HTTP	POST /DOWN/FIRMWAREUPDATE/ROM1 HTTP/1.1
printer	laptop	HTTP	HTTP/1.1 204 No Content

Abbildung 1: Aufgezeichnete Netzwerkkommunikation bei der Firmwareaktualisierung.

Nun wissen wir grob etwas über die Firmware-Datei und ihren Übertragungsweg. Es gilt, Näheres darüber herauszufinden. Der WF-2540 bietet laut Handbuch zusätzlich zu normalen Firmwareupdates einen speziellen Wiederherstellungsmodus an. In diesem Modus soll man nach einem fehlgeschlagenen Firmwareupdate neue Firmware installieren können. Für uns klingt das gut, falls wir das Gerät nämlich bei unseren Versuchen unbrauchbar machen sollten. Außerdem zeigt der Wiederherstellungsmodus Status und Fehlernachrichten an. Diese könnten helfen, den Prozess besser zu verstehen.

Das Handbuch enthält allerdings keine Informationen darüber, wie dieser Wiederherstellungsmodus zu nutzen ist. Eine Suche im Netz fördert zu Tage: Dieser versteckte Betriebsmodus wird aktiv, wenn man das Gerät mit der Tastenkombination *STOP + LEFT + COPY + POWER* einschaltet. Siehe da: Das Gerät startet und teilt uns auf dem Display mit, dass es auf Firmware wartet.

Die jetzt beim Update ausgegebenen Statusmeldungen ermöglichen, ein Ablaufdiagramm zu erstellen. Dafür führen wir mehrere Updates mit originaler Firmware durch. Damit aber möglichst verschiedene Pfade durchlaufen werden, verändern wir nach Zufallsprinzip in einem Hexeditor jeweils einige Bytes in der Firmware, um Fehlerzustände zu provozieren. Über die angezeigten Statusmeldungen ergibt sich so das Ablaufdiagramm in Abbildung 2.

Beim Update werden also zwei Dateien gelesen und irgendetwas mit der Bezeichnung ‚IPL' auf Korrektheit geprüft. Anschließend werden die Dateien in zwei separate ROM-Bereiche geschrieben.

Annäherung: Entropie-Analyse der Firmwarestruktur

Firmware ist in der Regel speziell auf einen einzelnen Gerätetyp zugeschnitten. Verschiedene Hersteller nutzen unterschiedliche Formen von Firmware für ihre Geräte. Herstellerübergreifend definierte Datenformate und Darstellungsformen sind unüblich.

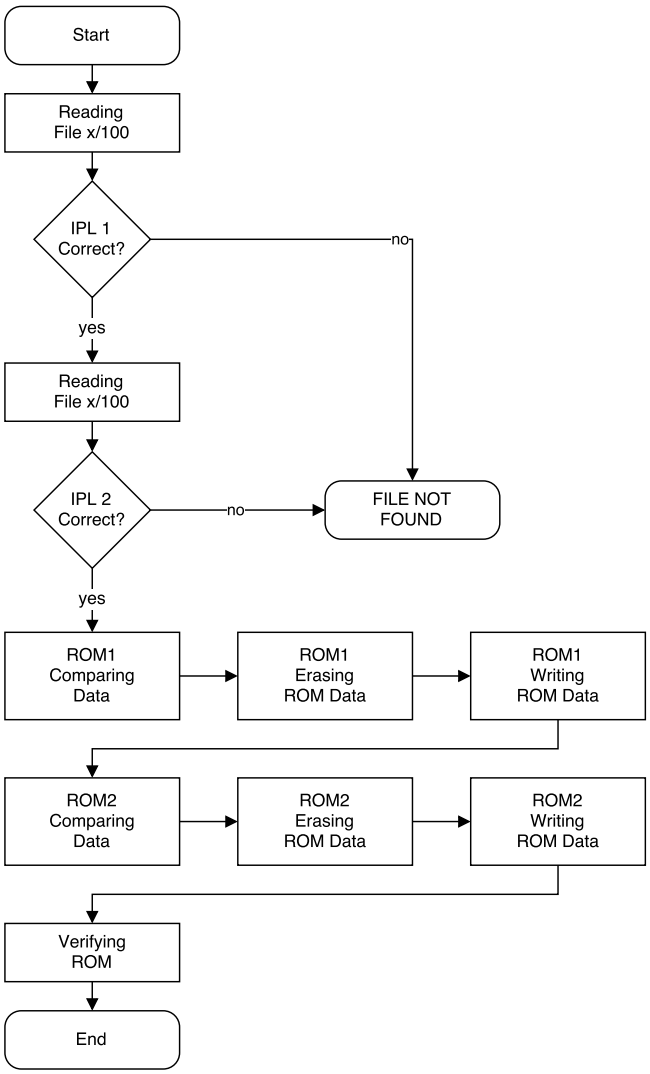


Abbildung 2: Ablaufdiagramm des Firmwareupdate-Prozesses.

Bei unbekannter Firmware ist es wichtig, zunächst einen Überblick zu bekommen und herauszufinden, womit man es zu tun hat. Dazu eignet sich hervorragend das Programm *binwalk* [3], welches in der Firmware nach bekannten Signaturen sucht. Dadurch lassen sich schnell bekannte Dateisysteme, Datentypen und komprimierte Daten innerhalb der Firmware identifizieren. Binwalk erlaubt außerdem, grafisch die Entropie einer Datei anzuzeigen, um deren Struktur besser zu verstehen. Wir durchsuchen also die Firmwaredatei nach Signaturen bekannter Dateiformate.

Dabei sucht das Tool nach Bytefolgen, mit denen bekannte Dateiformate starten. Das bringt jedoch nicht selten auch eine ganze Reihe von falsch erkannten Inhalten mit sich. Letztlich ist Handarbeit gefragt, um die Brauchbarkeit der Funde zu bewerten.

Deswegen gehen wir die Liste der Funde manuell durch und überprüfen offensichtlich falsche Funde mit einem Hexeditor. Falsche Funde sind dann wahrscheinlich, wenn die Daten schon auf den ersten Blick nicht zu den Beschreibungen oder Datenstrukturen passen, die für viele Datenformate (etwa JPG) dokumentiert sind. Mit etwas Suchen finden sich oft auch Datenstrukturen aus Implementierungen, die solche Dateien verarbeiten. Die Überprüfung erfolgt anhand der gefundenen Bytefolge, ob nämlich eine valide Datei dieses Typs bestehen könnte. Falls nein, brauchen wir diesen Fund nicht weiter zu beachten. Falls ja, betrchten wir den Fund näher, um mehr über die Firmware zu lernen – etwa ihre Dateisysteme.

Im Entropiegraphen (Abbildung 3) zeigt sich klar erkennbar, dass die Firmware aus mehreren Blöcken besteht. Bereiche mit hoher Entropie deuten auf Inhalte mit hohem Informationsgehalt hin. Stark komprimierte oder verschlüsselte Daten haben beispielsweise eine hohe Entropie. Im Falle unserer Firmware trifft dies etwa auf den komprimierten Kernel und die komprimierten Dateisysteme vom Typ *CRAMFS* (Compressed ROM file system) zu. Die Bereiche des Graphen mit einer Entropie nahe Null enthalten hingegen fast gar keine Informationen. Es handelt sich dabei um Blöcke mit Nullbytes.

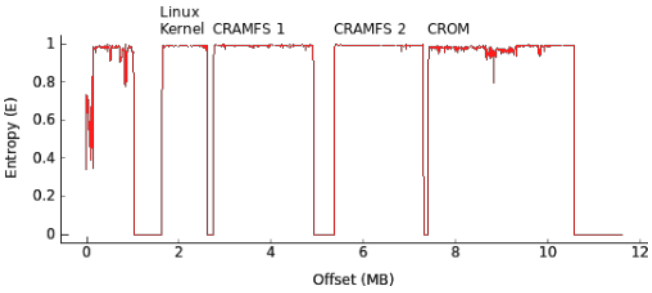


Abbildung 3: Entropiegraph der Firmware.

Nun wird klarer, womit wir es zu tun haben: Auf dem WF-2540 läuft ein Linuxkernel mit zwei komprimierten CRAMFS-Dateisystemen.

Sprungbrett Dateisystem

Da wir jetzt wissen, wo welche Arten von Dateisystemen in der Firmware enthalten sind, versuchen wir, diese genauer zu untersuchen. Mit `dd` schneiden wir die CRAMFS-Dateisysteme aus der Firmware aus. Anschließend entpacken wir sie mit `cramfsck`. Das erste CRAMFS enthält das Root-Dateisystem, das zweite proprietäre Programme und Konfigurationsdateien. Zum booten startet der Kernel lediglich ein Shellskript, welches Dateisysteme einhängt und das System einrichtet. Abgesehen von einem rudimentären *Busybox* existieren zwei proprietäre Anwendungen.

Welche Software auf dem WF-2540 läuft und was sie genau tut, ließe sich nun genauer untersuchen. Das ist bereits viel wert, aber noch besser wäre es natürlich, wenn wir die Software in Aktion beobachten und selbst verändern könnten. Um Zugriff auf den Drucker zu bekommen, versuchen wir also, auf dem Gerät eine Shell zu aktivieren, die auf Netzwerkverbindungen wartet. Dazu fügen wir auf dem entpackten Dateisystem in `/etc/inted.conf` folgende Zeile hinzu, die auf Port 4711 eine Shell öffnet:

```
4711 stream tcp nowait root /bin/sh sh -i
```

Mittels `mkcramfs` erstellen wir ein neues Dateisystem, das wir mit `dd` zurück in die Firmware kopieren. Anschließend starten wir den Installer mit der neuen Firmware und versuchen unser Glück. Wir stellen fest, dass das Gerät nach wenigen Sekunden ohne ersichtlichen Grund das Firmwareupdate stoppt. Wäre ja auch zu schön gewesen, wenn alles beim ersten Versuch funktioniert hätte.

Schutz aushebeln: Datenstrukturen genauer ansehen

Unbekannte Schutzmechanismen scheinen zu verhindern, dass der Drucker unsere manipulierte Firmware installiert. Jetzt wird es knifflig. Die grundlegenden Inhalte der Firmware sind gefunden. Wie aber werden diese Inhalte zur eigentlichen Firmware zusammengesetzt und vom Drucker gelesen? Beim Erstellen der Firmware kommen sehr wahrscheinlich Schutzmechanismen gegen Übertragungsfehler oder Veränderungen zur Anwendung. Ohne zu verstehen, wann diese angewendet werden und wie sie funktionieren, können wir das System nicht verändern. Also zerlegen wir sie in Details.

Damit der Drucker die Firmware verarbeiten kann, muss sie eine Struktur haben, in der Informationen abgelegt sind, die etwas über sie selbst sagen und darüber, wie mit ihr zu verfahren ist. Wie bei einer klassischen Datei ist davon auszugehen, dass die Firmware mit einer Art Dateiheader

beginnt. Ein Blick auf den zuvor erstellten Entropiegraphen bestätigt einen noch unbekannten Datenblock am Anfang der Datei. Mit `hexdump` lassen wir uns den Dateianfang ausgeben:

```
$ hexdump -C firmware | head
45 50 53 4f 4e 20 49 50  4c 66 00 4c 4a 00 30 00  |EPSON IPLf.LJ.0.|
01 00 59 35 00 00 00 00  00 00 00 00 00 10 00 00  |..Y5............|
02 00 8f 25 00 00 00 00  00 00 01 00 00 00 71 00  |...%..........q.|
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

Die ersten Bytes bilden die Zeichenkette *EPSON IPL*. Die Bezeichnung ‚IPL' ist bereits im Ablaufdiagramm für den Update-Prozess aufgetaucht. Allerdings ist sie dort zweimal zu finden. Wir suchen also nach der Zeichenkette, um zu prüfen, ob sie auch in der Firmware mehrfach vorkommt. Ergebnis: Die Zeichenkette ist auch genau zweimal in der Firmware enthalten. Wenn wir deren Positionen zusammen mit den bisherigen Funden in einer Grafik einzeichnen, ergibt sich die Strukturübersicht in Abbildung 4.

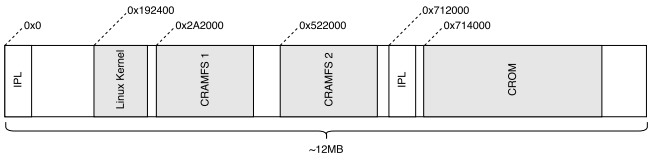


Abbildung 4: Strukturübersicht der Firmware.

Die aufgedeckten IPL-Datenstrukturen beschreiben, wie die Firmware aufgebaut ist. Wie aber kann diese Anhäufung von Bytes entziffert werden? Dazu gehört viel Geduld und langwieriges Probieren. Doch es gibt Schritte, die dabei helfen:

(1) Verschiedene Datenstrukturen sammeln.

(2) Unterschiede feststellen.

(3) Aufgaben einzelner Felder in der Datenstruktur rekonstruieren.

Es lohnt sich zunächst einmal, mehrere Datenstrukturen aus verschiedene Firmwareversionen für das zu untersuchende Gerät zu ziehen, um etwas Varianz zu haben. Je mehr, desto besser. Auch Firmware für andere Geräte aus derselben Modellreihe kann hilfreich sein.

Die gesammelten Datenstrukturen lassen sich dann zum Beispiel mit Binwalk vergleichen (Abbildung 5). So lassen sich zum einen einzelne Bereiche in der Datenstruktur identifizieren. Zum anderen hat man verschiedene Beispiele für Daten, die Felder in der Datenstruktur enthalten können. Das hilft ungemein dabei, deren Funktion zu studieren. Beim Interpretieren der Datenstrukturen ist immer darauf zu achten, in was für einer Ordnung die Bytes möglicherweise abgelegt sind. Wird das aus den Daten allein nicht ersichtlich, kann es nötig sein, die Werte zu interpretieren und zu schauen, welches Ergebnis am sinnvollsten erscheint.

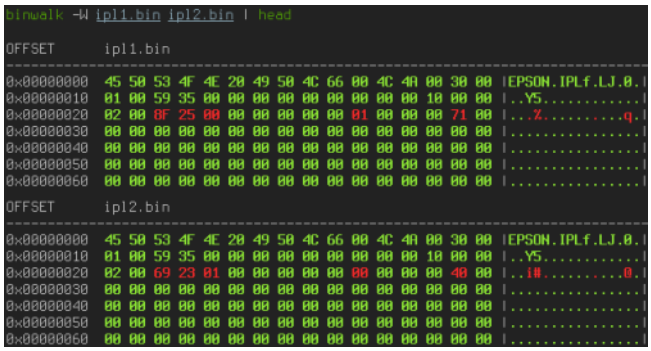


Abbildung 5: Unterschiede verschiedener IPL-Datenstrukturen.

Die IPL-Datenstrukturen des WF-2540 starten zum Beispiel allgemein mit dem Magic-Byte *EPSON IPL* und scheinen nach einem Vielfachen von 16 Bytes zu enden. Auf die Zeichenkette am Anfang folgen ein paar Bytes, deren Funktion nicht ganz klar ist. Sie enthalten allerdings an einer Stelle immer die zwei Zeichen, mit denen der alphabetische Teil der Versionsnummer beginnt. Beispielsweise stehen die Zeichen *LJ* für die Version *48.48.LJ05DC*. Die ersten 16 Bytes enden mit einem zwei-Byte-Feld, dass immer genau die Länge der Datenstruktur in Bytes enthält. Da die Werte im Little-Endian-Format ablegt sind, muss man die Bytes beim Lesen drehen.

Folgende Beschreibung ist das Ergebnis der Betrachtung vieler unterschiedlicher Firmwareversionen. Wie in Abbildung 5 zu sehen, verbleiben noch 32 weitere unbekannte Bytes. Bei diesen Bytes ist auffällig, dass sich ihre Struktur alle 16 Bytes ähnelt. Ein 16-Byte-Abschnitt scheint mit einer zwei-Byte-Zahl zu starten. Blöcke, die mit der gleichen Zahl beginnen, besitzen die gleiche Struktur. Wir nennen einen solchen Abschnitt im Folgenden einen *Record*. Ein weiteres, auf das erste folgende zwei-Byte-Feld scheint in jeder Firmwareversion anders zu sein. Dieses Feld ist aller Wahrscheinlichkeit nach eine Prüfsumme.

Jeder Record verweist auf einen Abschnitt mit Daten. Die letzten vier Bytes eines Record beschreiben die Länge eines solchen Abschnittes in Bytes. In jeder Firmwaredatei starten die Daten des ersten Record implizit mit *4kB (0x1000)* nach dem ersten Byte der IPL-Datenstruktur. ‚Implizit' bedeutet hier, dass dieser Abstand nicht explizit

etwa durch ein Feld in der Firmware angegeben ist. Die Datenblöcke der Records folgen aufeinander. Das Ende eines Blockes markiert den Anfang des nächsten Blocks: Auf den letzten Datenblock einer IPL-Struktur folgt also nahtlos eine weitere IPL-Datenstruktur mit neuen Records. Eine grafische Übersicht der analysierten Datenstruktur ist in Abbildung 6 zu sehen.

Abbildung 6: Übersicht der IPL-Datenstruktur.

Zum Kern vordringen: Prüfsummen reverse-engineeren

In der IPL-Datenstruktur verbleiben nun noch die unbekannten Prüfsummen. Noch ist unklar, nach welchem Algorithmus und über welche Daten sie gebildet werden. Um das herauszufinden, haben wir folgende Möglichkeiten:

(1) Verschiedene Algorithmen und Datenbereiche ausprobieren.

(2) Auslesen des Flash-Speichers auf der Hardware.

(3) Reverse-Engineering der Updateroutinen.

Im ersten Anlauf probieren wir also einfach Kombinationen von Algorithmen und Datenbereichen aus. Da wir die möglichen Wertebereiche nur schwer eingrenzen können, geben wir wegen der großen Anzahl an Kombinationen aber klein bei und schreiten zur aufwendigeren zweiten Möglichkeit fort. Ein Update schreibt Teile der Firmware auf den Speicher des Druckers. Diese geschriebenen Daten werden höchstwahrscheinlich in der Firmware von den Prüfsummen abgedeckt. Wenn wir den Speicher auslesen, können wir diese Daten vielleicht identifizieren und somit den Bereich eingrenzen, über den die Prüfsummen gebildet werden. Führt auch das nicht zu neuen Erkenntnissen, bleibt noch die dritte und aufwendigste Möglichkeit – mittels Reverse-Engineering die Updateroutinen identifizieren, um sie zu untersuchen.

Abbildung 7: Hauptplatine des Epson WF-2540.

Zunächst versuchen wir aber, den persistenten Speicher der Hardware auszulesen. Dazu zerlegen wir das Gerät und entfernen die Hauptplatine (Abbildung 7). Auf der Platine befinden sich zwei Flash-Speicherbausteine. Diese ICs werden über den *SPI*-Bus (Serial Peripheral Interface) angesteuert. SPI wird von einer ganzen Reihe von Hardware unterstützt. Um den Speicher auszulesen, reicht daher ein einfacher Einplatinenrechner wie der Raspberry Pi oder ein günstiger separater Speicherprogrammierer. Wir nutzen Letzteres. Zum Auslesen löten wir Kabel an die Kontakte der ICs und verbinden sie mit dem Programmierer (Abbildung 8).

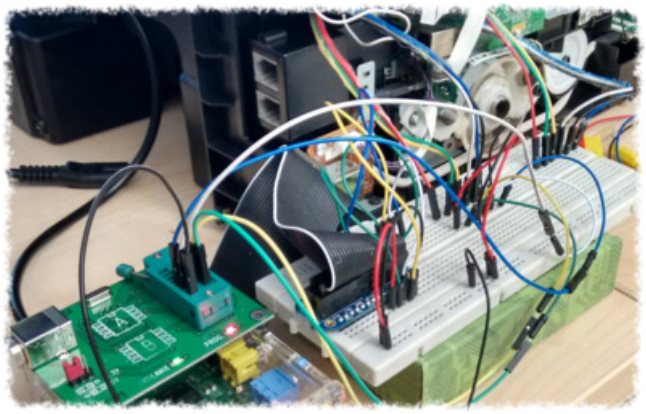


Abbildung 8: Auslesen der Speicherbausteine auf der Hauptplatine.

Die ausgelesenen Daten aus dem persistenten Speicher vergleichen wir mit der Firmware (Abbildung 9). Der Vergleich zeigt, dass die Updateroutinen Teilbereiche der Firmware eins zu eins auf den Drucker kopieren. Diese kopierten Bereiche sind auch genau jene, auf welche die Records der IPL-Datenstrukturen verweisen. Mit den identifizierten Datenbereichen erfolgt ein Rückschritt zu Listenpunkt 1, indem wir verschiedene Algorithmen ausprobieren. Dadurch erhalten wir die genaue Bildungsvorschrift der Prüfsummen: Sie entstehen durch Aufsummieren von Bytes als zwei-Byte-Integer ohne Behandlung des Überlaufs. Abbildung 10 zeigt die in die Prüfsummen einbezogenen Daten.

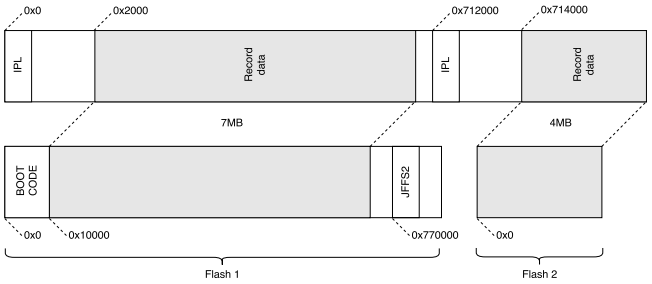


Abbildung 9: Vergleich zwischen Firmware und persistentem Speicher.

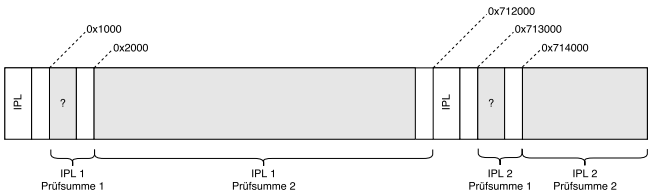


Abbildung 10: Daten, die in die Prüfsummen einfließen.

Der Erfolg ist das Problem: Eigene Firmware aufspielen

Nun ist der Moment der Wahrheit gekommen. Wir nehmen uns die zuvor erstellte Firmware, die mit der Rootshell auf Netzwerkverbindungen wartet. Wir berechnen neue Prüfsummen und tragen sie in die Datenstrukturen ein. Zu unserer Begeisterung – oder unserem Erschrecken – führt der Drucker das Update mit unserer selbst erstellten Firmware ohne Murren durch.

Ein Verbindungsversuch zur Rootshell mittels `netcat` ist ebenfalls erfolgreich. Wir haben nun während des Betriebs vollen Rootzugriff auf den Drucker. Weitere Analysen würden nun zur Laufzeit auf dem Livesystem erfolgen. Damit wir eigene Software auf dem Drucker nutzen können, kann zum Beispiel einen ARM-Cross-Compiler eingerichtet werden. Mit diesem lassen sich dann eigene Kernelmodule und Programme für den Drucker erstellen.

Die Frage nach der Sicherheit des Epson WF-2540 war der eigentliche Grund, sich mit dessen Firmware zu beschäftigen. Aus der Untersuchung lassen sich folgende sicherheitskritische Probleme zusammentragen:

- Die Firmware ist nicht signiert.
- Die Prüfsummen sind schwach.
- Keine Authentifizierung bei Updates.

Aufgrund fehlender Signaturen kann das Gerät nicht überprüfen, woher die ihm gelieferte Firmware stammt. Damit ist die Unterscheidung zwischen Schadsoftware und Herstellersoftware nicht möglich. Ein weiteres Problem geht von den verwendeten Prüfsummen aus. Sie decken wichtige Bereiche gar nicht ab, wie etwa die IPL-Datenstrukturen. Defekte oder auch willentlich eingefügte Veränderungen gerade der IPL-Bereiche können aber zu defekter Firmware und damit auch zu defekten Geräten führen. Deswegen sollten die Prüfsummen eigentlich verhindern, dass beispielsweise Übertragungsfehler zu defekter Firmware führen. Hinzu kommt, dass die Prüfsummen größere Bereiche mit ungenutzten Nullbytes einschließen. Da die Bytes lediglich aufsummiert werden, kann ein Angreifer durch Manipulation der Nullbytes beliebige Prüfsummen erzeugen. Zu allem Überfluss werden Updates auch noch ohne Authentifizierung ausgeführt. Der Drucker kann also nicht bestimmen, wer das Update durchführt, und ob derjenige überhaupt dazu berechtigt ist.

Jeder, der physikalischen Zugriff auf den Drucker hat oder sich im gleichen lokalen Netzwerk mit ihm befindet, kann diesen also übernehmen. Manipulierte Firmware lässt sich problemlos über USB, LAN oder auch WLAN aufspielen.

Dieser Zustand lässt sich noch weiter eskalieren. Schaut man sich die HTTP-Nachrichten genauer an, so fällt auf, dass das Gerät keine Schutzvorkehrungen gegen *CSRF* besitzt (Cross-Site Re-

quest Forgery, siehe Abbildung 11). Ein Angreifer muss daher nicht einmal direkt mit dem Gerät kommunizieren, um die Firmware zu ersetzen. Nahezu jeder Netzwerkteilnehmer, der sich im gleichen Netzwerk mit dem Drucker befindet, lässt sich dazu missbrauchen.

Dafür muss der Angreifer lediglich Javascript auf einer Webseite platzieren, die einer der Netzwerkteilnehmer aufruft. Das Javascript führt dann vom Browser des Opfers aus das Update durch. Ein Drucker in meinem lokalen Netzwerk kann also durch das bloße Abrufen einer Website infiziert werden. Sensible Informationen und Dokumente, die der Drucker gedruckt, gefaxt oder kopiert hat, lassen sich nach Belieben auslesen und verändern. Sogar das eingebaute Modem des Druckers kann zur Kommunikation genutzt werden, falls dieses keine Netzwerkverbindung mehr haben sollte. Von einem infizierten Gerät aus ist es möglich, weitere Geräte über das Netzwerk oder die USB-Verbindung anzugreifen.

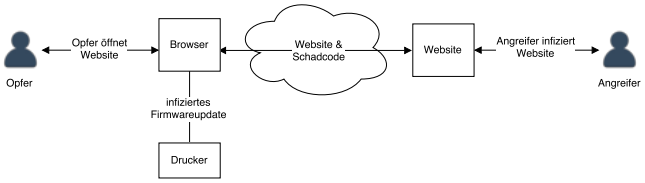


Abbildung 11: Angriffsszenario für eine CSRF-Attacke.

Epson wurde im September 2015 über die gewonnen Erkenntnisse informiert. Mangels eines direkten Ansprechpartners für Sicherheitsprobleme gingen die Information einfach per E-Mail an die Pressestelle des Unternehmens. Im Dezember 2015 kam eine Rückmeldung, die ankündigte, das Problem Ende Januar 2016 durch eine neue Firmware zu beheben. Mitte Februar 2016 erschienen dann neue Firmwareversionen. Auch wenn nicht jede Untersuchung erfolgreich sein mag, so hofft der Autor dieser Zeilen, den Ansporn zum genaueren Hinschauen ab und zu geliefert zu haben.

Links

[1] Epson WF-2540:
https://www.epson.de/products/printers/inkjet-printers/microbusiness/epson-workforce-wf-2540wf
[2] *Wireshark*: https://www.wireshark.org
[3] *binwalk*: https://github.com/devttys0/binwalk

Über Yves-Noel

Yves-Noel Weweler besitzt den Bachelor in der Informatik und studiert aktuell Informatik im Masterstudiengang an der Fachhochschule Münster. Vor seinem Studium hat er eine Ausbildung zum Informationstechnischen Assistenten abgeschlossen. Er ist Gründungsmitglied und Vorsitzender der Studentengruppe der *Gesellschaft für Informatik Münsterland*.

Freiwillige Feuerwehr
Kernel-Regressions bekämpfen

In einem englischsprachigen Vortrag erläuterte Heise-Journalist Thorsten Leemhuis auf der *LinuxCon Europe* 2016, was Kernel-Regressions sind, warum es so wichtig ist, sie zu finden, und wie man das macht – ein feuriges Plädoyer, mit dem Löschzug auszurücken. Der folgende deutsche Text entstand mit freundlicher Genehmigung des Vortragenden.

übersetzt von **Anika Kehrer**

Man kämpft nicht nur
mit dem Schwert,
sondern auch mit dem Herzen.

Gustav Stresemann

Angenommen, nach einem Kernelupdate funktioniert etwas an einem Linux-System schlechter als vorher. Zum Beispiel ist die Performance schlechter, der Stromverbrauch höher, oder etwas geht schlicht nicht mehr. Eine solche Verschlechterung nach einem Update ist eine Regression.

Jede Regression ist ein Bug. Aber nicht jeder Bug ist eine Regression. Das ist ein entscheidender Unterschied. Wie zum Beispiel in einem eigenen Haus gibt es auch beim Kernel immer irgendetwas, das nicht funktioniert. Wenn der Klempner kommt, und seine Arbeit zu neuen Problemen führt, wirst du seine Arbeit reklamieren: Du bezahlst ihn schließlich nicht dafür, etwas schlechter zu machen – das wäre eine Regression. Aber du kannst ihm nicht anlasten, wenn etwas in anderen Bereichen schon lange vorher nicht funktioniert hat – das wäre ein Bug. Ähnlich wie sich Hausbesitzer mit Macken ihrer vier Wände arrangieren, leben auch die Kernelentwickler mit einigen Bugs. Regressions hingegen sind etwas, das sie fürchten.

Und zwar aus folgendem Grund: Die Entwickler verändern und verbessern den Linux-Kernel ständig. Der Code wächst dabei alle zehn Wochen um rund 300.000 Zeilen. Der Kernelcode besitzt Stellen, wo Neues hinzukommt. Er besitzt aber auch ungeschliffene Bereiche und Stellen, die nie wirklich beendet wurden. Dinge mit einer neuen Release zu verschlechtern (sprich: eine Regression zu haben), ist gefährlich. Denn jede Regression verprellt ein paar mehr Tester. Das wiederum führt zu noch mehr unerkannten Regressions. Das verscheucht noch mehr Tester. Das führt zu noch mehr unerkannten Regressions. Das Ganze ist also offensichtlich die falsche Richtung.

Wie Regressions zu behandeln sind

Linus Torvalds hat daher seit Anbeginn der Linux-Zeit klar gestellt: Keine Regressions! Wenn ein Code-Change eine Regression verursacht, dann wird

(1) entweder die Regression behoben,

(2) oder der Commit mit der Änderung rückgängig gemacht.

Das klingt soweit einfach und prima. Es funktioniert auch ziemlich gut beim Linux-Kernel. Es gibt aber ein Problem. Der Auslöser der Regression muss bekannt sein, um ihn rückgängig machen zu können. Nur das Problem zu beschreiben, reicht in den meisten Fällen daher nicht aus.

Die Losung lautet also: Leg den Finger in die Wunde und spüre den Commit auf, der die Regression verursacht. Finde also das Problem in 12.000 Commits. Denn das ist die typische Zahl der Änderungen zwischen einem Kernel und der ersten Vorabversion des Nachfolgers. Noch einmal rund 1.500 Commits mehr sind es, bis der Nachfolger dann fertig ist.

Die Ursache zu finden klingt aufwendig, ist es aber gar nicht: Du checkst mit dem Quellcodeverwaltungssystem einfach eine Änderung aus, die sich etwa auf der Hälfte von diesen rund 12.000 Commits befindet. Aus diesem Zwischenstand baust du dann einen Kernel und prüfst, ob sich die Regression mit ihm zeigt. Tritt sie auf, verfährst du genauso mit der ersten Hälfte der 12.000 Commits; tritt sie nicht auf, tust du dasselbe mit der zweiten Hälfte der Commits:

$$12000 \div 2 \div 2 \div 2 \div 2 \div 2 \div 2 \div 2 \div 2 \div 2 \div 2 \div 2 \div 2 \div 2 \div 2 \div 2 =< 1$$

Mit anderen Worten baust du etwa 15 mal einen Kernel und schaust jeweils, ob das Problem noch auftritt. So findest du den schuldigen Commit. Man nennt den Prozess *bisection*, und `git bisect` vereinfacht ihn [1].

Wichtig dabei ist das Timing. Es ist genau dasselbe, wenn du die Arbeit des Klempners reklamierst: Der lacht dich aus, wenn du 12 oder 24 Monate nach seinem Auftrag einen Fehler an einer Stelle reklamierst, an der andere Klempner seitdem gearbeitet haben. Darum ist es wichtig, frühzeitig und regelmäßig auf Regressions zu testen. Ohnehin wird es mit der Zeit immer schwerer, das Problem einzugrenzen und zu beheben: Schon der Klempner weiß oft nicht mehr, was er vor ein oder zwei Jahren gemacht hat; ein Programmierer weiß es mit Sicherheit nicht mehr. Mithin wird der Heuhaufen größer, in dem du selbst die Nadel suchst. Zwischen Kernelversion 4.4 und 4.8 lagen um die 55.000 Commits, und auch Bugs häufen sich an. Vielleicht hast du es ja gar nicht mit einer Regression, sondern mit mehren zu tun – oder ein oder mehrere Bugs schießen beim Testen quer. Es wird wirklich schwer, das auseinanderzuhalten.

Schließlich wird es mit verstreichender Zeit immer komplizierter, eine Änderung rückgängig zu machen. Manchmal kann das auch sehr schnell unmöglich werden. Ein Commit, der für die ersten Vorabversion von 4.8 vorgenommen wurde, ist normalerweise leicht rückgängig zu machen, bevor Version 4.8 erscheint. Das gilt auch dann, wenn es die Änderung in einer großen Gruppe von Changes ist, denn notfalls wird diese dann komplett zurück genommen. Aber sobald Version 4.8 draußen ist, geht das nicht mehr so einfach. Vielleicht wurde dann schon etwas für Version 4.9 gemergt, das von dem schuldigen Change abhängig ist. Oder der schuldige Change besitzt ein Feature, auf das Leute bereits zählen – dann würde das Rückgängigmachen dieses Changes sozusagen eine Regression für diese Leute bedeuten.

Darum ist es wichtig, neue Kernelversionen frühzeitig zu testen – am besten noch, bevor sie veröffentlicht sind. Darum solltest du Pre-Releases (aka *rc*) der Kernel testen (zum Beispiel Version *4.9-rc2*). Und: Nein, es ist nicht so gefährlich, wie es sich vielleicht anhört. Das sehen wir uns gleich mal an.

Beim Testen gilt es zu bedenken: Es gibt Probleme, die keine Regression sind. Du solltest dich zum Beispiel nicht darauf verlassen, dass Device-Namen (`sda`, `sdb`, …) gleich bleiben. Du solltest keine externen Treiber verwenden, die sich nicht im Kerneltree befinden. Aber lass dich davon nicht entmutigen. Und vor allem mach dir bewusst, warum die Mühe für dich ganz persönlich wichtig ist.

Warum man Kernel-Regressions bekämpfen sollte

Aus bestimmten Gründen ist es für jeden einzelnen Linux-Nutzer extrem wichtig, Kernel-Regressions zu bekämpfen. Moderne Computer sind extrem komplex. Außerdem gibt es hunderte Distributionen mit unterschiedlichen Komponenten und Treibern. Dazu kommen etliche Möglichkeiten, den Kernel zu konfigurieren und zu nutzen. So gibt es alleine diverse Dateisysteme (*btrfs*, *ext4*, *xfs*, …) und verschiedene Storage-Layer (*mdadm*, *crypto*, *lvm*, …), die sich auf vielfältige Weisen kombinieren lassen. Ähnlich sieht es beim Netzwerksubsystem und anderen Bereichen aus. Alle diese Kombinationen müssen getestet werden, denn manche Regressions tauchen lediglich in bestimmten Kombinationen auf. Andere Regressions zeigen sich nur mit einer der Zillionen von Hardware-Komponenten da draußen. Manches Problem tritt nur in einer ganz bestimmten Kombination von Bauteilen auf, oder nur mit einer speziellen Firmware.

Das macht dein Setup höchstwahrscheinlich einzigartig. Du kannst dich also nicht darauf verlassen, dass andere die Regressions finden, die deine Systeme betreffen. Dein Setup ist vielleicht auch dann einzigartig, wenn du bekannte, weitverbreitete Systeme benutzt. Linus Torvalds hat zum Beispiel jüngst einen Dell XPS 13 gekauft [2]. Man könnte sich dann denken: Das Notebook müsste ja gut funktionieren. Aber die Bezeichnung *XPS 13* steht für eine ganze Reihe verschiedener Systeme, denn ungefähr einmal pro Jahr erscheint ein neue Gerätegeneration, die neuere und manchmal sogar ganz anderen Bauteile enthält. Und selbst zwischen den Notebook-Modellen einer Generation gibt es oft größere und signifikante Unterschiede.

Torvalds hat beispielsweise nicht erwähnt, ob er das FHD- oder das QHD-Touch-Display genommen hat – und vielleicht tritt eine Regression nur mit der Konfiguration auf, die der Linux-Vater gerade nicht hat. Manchmal ist auch die Storage-Konfiguration entscheidend. Und übrigens unterscheiden sich auch die Modelle *XPS 13* und die *XPS 13 Developer Edition* teilweise. Die Developer-Edition (das ist die, die mit Ubuntu kommt) des 2015er-Modells nutzt beispielsweise ein anderes Wlan-Modul. Außerdem kam zeitweise auch ei-

ne andere Firmware zum Einsatz, die den Audio-Codec ganz anders konfigurierte.

Alle diese kleinen Unterschiede können eine Regression triggern, die nur auf einer der vielen Modell-Varianten auftritt. Das Dell-Notebook ist dabei noch ein einfaches Beispiel, denn bei den verbreiteten Lenovo-T- oder -X-Laptops gibt es noch mehr Modellvarianten. Einige T450-Modelle kommen beispielsweise mit integrierter GPU, andere mit dedizierter, ganz zu schweigen von Addons wie Repeatern oder Dockingstations. Immerhin ändert sich die Modellnummer mit jeder neuen der Generation (T450 wird zu T460), statt wie beim XPS 13 immer gleich zu bleiben.

Wenn du dein Setup nicht testest, wirst du früher oder später in eine Regression laufen. Umzukehren wird dann manchmal schwer oder unmöglich sein: Selbst nach einem kleinen `apt-get`-Update fehlt vielleicht gerade einfach die Zeit, um den vorher eingesetzten Kernel wieder heran zu holen. Nach einem größeren Distributionsupdate ist die Rückkehr zu einem alten Kernel vielleicht immens aufwendig oder auch deswegen keine wirkliche Option mehr, weil deine alte Distro keinen Sicherheitsupdates mehr bekommt.

Also ist es wichtig, dein spezifisches Setup immer wieder auf Regressions zu testen, damit du Probleme früh erkennst, solange du sie noch leicht umgehen und korrigieren lassen kannst. Je weniger verbreitet oder älter dein Setup ist, und je exotischer deine Distribution und deren Konfiguration ist, desto wichtiger ist es. Darum ist es in deinem eigenen Interesse, beim Bekämpfen von Regressions zu helfen.

Wie man auf Regressions testet

Das ist ganz einfach: Führe den Vorabversionen des Mainline-Kernel aus – also „rc“ oder „Prerelease“ genannten Kernel-Versionen, wie in *Linux 4.9-rc1*. Nein, das ist nicht gefährlich. Das Risiko für Datenverlust ist etwas höher als bei Distro-Kernels, aber es ist trotzdem klein. Und Backups machst du ja sowieso, nicht wahr?

Du benutzt entweder ein schon kompiliertes Kernel-Image, oder du baust den Kernel selbst. Beide Varianten sind nicht wirklich schwierig. Im täglichen Gebrauch ist ein vorkompilierter Kernel völlig in Ordnung. Solche findest du in Repositories, die entsprechende Debian- und RPM-Pakete zur Verfügung stellen. Bei Fedora gibt es das Vanilla-Repo [3], bei Suse heißt das Repo *KOTD* (Kernel of the Day), bei Ubuntu gibt es das *Linux-mainline PPA*. Einige Distributionen liefern sogar RC-Kernel, zum Beispiel *Fedora Rawhide*.

Der Nachteil des kompilierten Kernels ist, dass sie manchmal distributionsspezifische Patches enthalten. Manchmal sind das sogar ganz schön viele. Es ist also ratsam, eine im Distro-Kernel gefundene Regression mit einem Vanilla-Kernel erneut zu testen. Den Vanilla-Kernel brauchst du allerdings sowieso, wenn du den Kernel bisektieren willst. Einen eigenen Kernel zu kompilieren bereitest du wie folgt vor (Beispiel Fedora):

```
$ dnf install make gcc git openssl-devel
```

Dann führst du folgende Kommandos aus:

```
01 $ git clone \
    git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
02 $ make olddefconfg localmodconfg
03 $ make -j 12 bzImage modules
04 $ sudo make modules_install install
```

Zeile 01 lädt das Git-Verzeichnis mit den Kernelquellen herunter. Das kann eine Weile dauern. Zeile 02 übernimmt mit der Make-Anweisung `olddefconfg` die Konfiguration deines Distributions-Kernels und setzt alle noch nicht gesetzten Konfigurationsparameter auf den Standardwert. Das `localmodconfg` deaktiviert alle Module, die auf deinem System in dem Moment nicht genutzt werden, um die Zeit zum Kompilieren zu reduzieren. Dieses Make-Target verursacht eine Handvoll Rückfragen. In den allmeisten Fällen kannst du hier *Enter* drücken, um den Standardwert zu übernehmen. Die resultierende Konfiguration ist nicht perfekt, aber schnell erzeugt und für die meisten Fälle gut genug. Zeile 03 baut schließlich das Kernel-Image und seine Module. Auf einem fünf Jahre alten Lenovo T420 dauert das übrigens 12 Minuten. Zeile 04 installiert das Ganze, was auf besagtem T420 mit Fedora 75 Sekunden dauert. Wenn du eine Regression jagst, die in der jüngsten Kernelversion enthalten ist, prüfe die vorangegangene Major-Release, ob die Regression nicht schon dort auftritt.

Das Ganze ist also nicht wirklich schwer. Nun schaust du, ob alles funktioniert. Wenn nicht, kannst du einfach zu deinem vorigen Kernel zurückkehren. Wenn ja, und wenn du nach einigen Tagen einen neuen Snapshot des Kernels ziehen willst, gibst du Folgendes ein:

```
01 $ git pull
02 $ make olddefconfg
03 $ make -j 12 bzImage modules
04 $ sudo make modules_install install
```

Wenn du alte Kernel und ihre Module loswerden willst, listest du sie mit `ls -tr /lib/modules/` auf. Dann suchst du die Version, die du löschen willst. Pass dabei aber

auf, dass ein funktionierender Kernel installiert bleibt – der Distro-Kernel zum Beispiel. Dann löschst du das Verzeichnis und alle Dateien in `/boot/`, die die Versionsnummer des zu löschenden Kernels enthalten. Optional aktualisierst du die Grub-Konfiguration mit einem Befehl wie `grub-mkconfg` oder (bei Ubuntu) `update-grub`.

Wie man Regressions meldet

Drei Schritte solltest du gehen, um allen das Leben leicht zu machen (auch dir selbst):

(1) Schicke eine Mail an die passende Mailingliste und setze die relevanten Entwickler in CC;

(2) Gib ausführliche, aber relevante Information;

(3) Sei hartnäckig! Aber sei auch immer freundlich und offen für Hinweise.

Wie findet man Kernelmailinglisten und -entwickler? Führe `scripts/get_maintainer.pl` aus den Kernelquellen aus. Sieh dir die Kernelmailinglisten an [4]. Oder suche im Web, indem du Worte wie ‚mailing list linux development' mit den passenden Themenworten kombinierst, etwa ‚usb', ‚networking', ‚wirless' oder ‚wireless intel'. Im Zweifel nutze die *LKML* (Linux Kernel Mailing List, [5]).

Du kannst deine Regressions auch beim Kernel-Bugzilla [6] melden. Die Betreuer einige Subsystemen haben den am nicht im Blick und sehen den Fehlerbericht dann nicht. In anderer Subsystemen funktioniert das aber gut – zum Beispiel für ACPI, PCI und Powermanagement. Manche Entwickler benutzen auch andere Bugzilla-Instanzen, zum Beispiel die von *freedesktop.org*. Wenn du unsicher bist, frag um Hilfe. Den Vortragenden dieser Worte zum Beispiel. Aber Vorsicht – er skaliert nicht besonders gut.

Links

[1] Christian Couder: Fighting regressions with `git bisect` (2009). URL: https://www.kernel.org/pub/software/scm/git/docs/git-bisect-lk2009.html
[2] Torvalds' Laptopkauf: https://plus.google.com/+LinusTorvalds/posts/VZj8vxXdtfe
[3] Vanilla-Kernel bei Fedora: https://fedoraproject.org/wiki/Kernel_Vanilla_Repositories
[4] Kernel-Mailinglisten: https://kernelnewbies.org/ML
[5] *Linux Kernel Mailing List*: https://lkml.org
[6] Kernel-Bugzilla: https://bugzilla.kernel.org

Über Thorsten

Thorsten Leemhuis ist seit vielen Jahren Redakteur bei *c't* und *Heise Open*, wo er unter anderem das *Kernel-Log* schreibt (https://www.heise.de/thema/Kernel_Log). Der Fedora-Fan beobachtet die Kernelentwicklung seit über zehn Jahren intensiv. Neuerdings erzeugt er manchmal Listen mit Regressions des Mainline-Kernels, um den Kernelentwicklern die Arbeit zu erleichtern. Der schreibende und vortragende Linux-Experte ist auf vielen Linux- und Open-Source-Veranstaltungen anzutreffen und unter <linux@leemhuis.info> zu erreichen.

Ein Gral für DevOps
Infrastrukturtests automatisieren

Konfigurationsmanagement macht IT-Infrastruktur zu Code. Für Code ist Continuous Integration das Ziel. Mit *Serverspec* und *Kitchen* gelingen automatische Tests. Für die Pipeline schließen das Plugin *Job DSL* von *Jenkins* 2.0 und das Jenkinsfile die Lücke, welche Repo-Anbieter mit inkompatiblen Eigenentwicklungen bisher nur so naja schließen konnten.

von **Christopher J. Ruwe**

Nur ein schlechter Plan
erlaubt keine Änderung.

Publilius Syrus

Wenn Konfigurationsmanagement den Aufbau von IT-Infrastruktur automatisiert, spricht man von Infrastructure as Code. Solchen Code entwickelt und testet jemand, wie jede andere Software auch. Automatisierte Tests – die Erfahrung lehrt nämlich, ist es nicht automatisiert, wird es ignoriert – müssen nah am entwickelten Code liegen, und zwar in demselben Code-Repository wie das Testobjekt. Die Ausführungs-Sequenzen solcher Tests sind in so genannten Pipelines spezifiziert und werden von bestimmten Operationen auf das Repository (`commit`, `pull`, `merge`, etc.) getriggert.

Das Jenkinsfile ist eine solche Pipeline für den CI-Server *Jenkins* [1]. Baut man damit die Ausführung von Infrastrukturtests geschickt, ist nicht nur vollständige Automatisierung des Testablaufes möglich, sondern auch die Hochskalierung auf VM- oder Cloud-Infrastrukturen. Technisch spricht nichts dagegen, die Testausführung abschließend um das Deployment zu erweitern – es winkt Continuous Deployment.

Das Problem: Fehlanzeige bei Infrastrukturtests

Nicht erst seit Meyer [2] fordern progressive Software-Architekten, dass Software modular konstruiert werden sollte. Dabei wird die Funktion einer Applikation aus logisch abhängigen, aber technisch isolierbaren Komposita von Einzelkomponenten erbracht, die in der Gesamtheit die Funktion der Software bereitstellen. Für IT-Infrastruktur drängt sich solch ein Aufbau geradezu auf: Moderne Server liefern tausende Prozesse, deren Zusammenwirken mehrere Dienste definiert. Diese wiederum sind üblicherweise über mehrere Hosts verteilt und stellen ein klassisches verteiltes System dar [3].

In der Praxis entspricht aber weder die Methodik derjenigen eines modular aufgebauten verteilten Systems, sondern eher einer Wucherung. Noch haben sich Techniken zur Abnahme eines solchermaßen gebauten Systems etabliert, abgesehen von budgetärem Mittelabfluß und Zeitablauf als Maß für den Projektfortschritt. Egal, ob klassisch manuell oder automatisiert gebaut, ähneln die entstandenen Artefakte nach wenigen Monaten Gebäuden, die über Jahrhunderte verwittert, erweitert und an ständig wechselnde Bedürfnisse angepasst wurden, ohne es allerdings mit dem Charme pittoresker Ruinen aufnehmen zu können. Auf einen Nachweis der Funktion verzichtet man üblicherweise – warum auch, der Kunde zahlt ja trotzdem – und überlässt die Korrektheit dem Zufall. Würde man mit demselben Vorgehen zum Beispiel eine Prozess-Strecke in einem chemischen Werk konstruieren, 1) könnte die Fabrik absolut gar nichts produzieren, 2) wären die Verantwortlichen entlassen worden und in Haft, 3) wären die betroffenen Unternehmen in den nächsten Jahren extremem regulatorischen und aufsichtsrechtlichen Druck ausgesetzt.

Dem Autoren ist aus seiner Consulting-Praxis kein technisches Produkt bekannt, dem zwingend eine sauber konstruierte Systemarchitektur zu Grunde liegt (es sei denn, man geht zu Google oder Netflix). Nimmt man mal an, dass eine IT-Infrastruktur genauso systematisch abzutesten und abzunehmen wäre wie die Software, die sie beherbergt, dann erscheint das als de-facto-Abnahmeverfahren akzeptierte Abwarten und mehr oder weniger formalisierte Herumklicken eher untauglich. Ich habe erlebt, wie einmal so genannte fachliche Experten den parallelen Neuaufbau eines Web-Frontends innerhalb von 10 Minuten abnickten. Der Load-Balancer steuerte allerdings durch Session-Cookies jeweils

auf die letztbesuchte Strecke. Die testenden ‚Experten' leerten grundsätzlich nie Browser-Cache und Cookie-Store („Es geht dann schneller"). Und natürlich verhielt sich der alte Aufbau nicht allein deswegen anders, nur weil es daneben eine neue Strecke gab.

Manuelles Testen ist grundsätzlich nur mit höchster Disziplin reproduzierbar. Selbst dann ist es für die im Cloud-Umfeld innerhalb weniger Minuten horizontal skalierenden Systeme viel zu langsam. Seine Testergebnisse – „Die Webseite geht nicht"– sind zu unpräzise, zu schwer zu kommunizieren und decken oft nicht die relevanten Details ab, die später richtig schmerzhaft werden.

Verloren geben sollte man Infrastrukturtests deswegen aber nicht. Automatisierungscode ist Software. Da liegt es nahe, sich bei den Kollegen aus der Softwareentwicklung zu bedienen. Dort hat man nämlich aus langer und sehr unangenehmer Erfahrung mit Fehlern im Produktionsprozess Toolsets entwickelt, Erzeugnisse bereits vor der Lieferung zu testen. Die Idee, solche Toolsets auf die Entwicklung von Infrastruktur abzubilden, erscheint fast schon trivial. Zwar erzwingt das keine systematische Konstruktion. Aber zusätzlich zum Vorteil formaler Nachweisführung resultiert automatisches Testen methodisch früher oder später fast zwingend in einen strukturierten Aufbau.

Methodik zum Testen von IT-Infrastruktur

Tests sollten methodisch strukturiert und systematisch entwickelt sein. Man unterscheidet dafür zwischen dem Aufbau des Tests, der Testausführung und dem Testablauf (die sogenannte Pipeline). Bei Anwendungssoftware ist es üblich, mehrstufig zu testen: Man beginnt mit der Einzelkomponente, fährt mit logisch zusammenhängenden Komponenten fort und endet beim Gesamtsystem aller Komponenten. Mit steigendem Abstraktions- und damit Komplexitätsgrad spricht man von *Unit-Test* (Einzelkomponente), *Integration-Test* (Komposit) und *Acceptance-Test* (vollständiges System, siehe Abbildung 1).

Das lässt sich nun analog auf die IT-Infrastruktur abbilden. Beim Unit-Test könnte man die Einzelkomponente als die Konfiguration einer bestimmten Software auffassen, etwa eines Webservers. Bei *Puppet* entspräche das einem *module*, bei *Ansible* heißt es *role* (Vorsicht: Der Begriff *role* ist bei Puppet und Ansible wechselseitig inkompatibel belegt). Puppet – das mehrstufig vorgeht – kompiliert nun aus dem Code einen so genannten Katalog und appliziert diesen anschließend automatisch auf den Ziel-Host. Der Puppet-Katalog drängt sich als Testobjekt förmlich auf. Größeren Puppet-Modulen liegen meist (Katalog-)Tests bei, die aus einer Menge typischer Host-Invarianten wie *hostname* oder *architecture* einen Katalog kompilieren und diesen dann auf Korrektheit prüfen. Ansible appliziert direkt. Hier müsste man die tatsächliche Applikation auf einen Server testen.

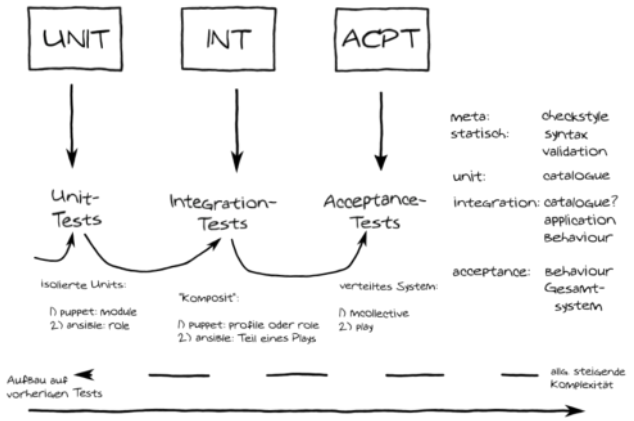


Abbildung 1: Testsystematik mit Unit-, Integration- und Acceptance-Test.

Für den Integration-Test auf der nächsten Stufe könnte man bei Puppet die Komposition eines aus mehreren Modulen bestehenden Konfigurationskatalogs prüfen. Die Funktion eines Webservers wird ja nicht allein durch die Webserver-Software bestimmt. Es gibt parallel dazu zum Beispiel noch (Funktions-)User und Gruppen, Rechte, vielleicht *logrotate*-Regeln. Interessanter als der Katalog ist es aber, die Konfiguration tatsächlich auf einen Host zu applizieren. Dies ist dann auch für Systeme möglich, die nicht zweistufig kompilieren und applizieren, also etwa Ansible. Denn hat man erfolgreich auf einen Host appliziert, weiß man nicht nur, dass der Katalog enthält, was er enthalten soll, sondern auch, dass er auch lauffähig ist. Daraufhin lässt sich das Verhalten des Servers prüfen: Ist der Webserver erfolgreich gestartet? Sind alle Ports offen, die offen sein sollen? Kann der Webserver auf die *static files* zugreifen? Und so weiter.

Für den Acceptance-Test ergibt sich hierauf aufbauend methodisch keine wesentliche Änderung mehr. Der Nachweis über die Funktion der Einzelkomponenten ist bereits erbracht, und es bleibt zu zeigen, dass die Verbindungen zwischen ihnen korrekt konfiguriert sind. Etwa: Kann der Webserver vom Applikations-Server dynamischen Content abgreifen? Kann der Applikationsserver auf

die Datenbank zugreifen? Sperrt die Firewall Zugriffe aus dem öffentlichen Internet auf die Datenbank?

Programmierte Testausführung mit *Serverspec* und *Kitchen*

Das vorhin verrissene Verfahren, semiformalisiert auf Webinterfaces herumzuklicken, wird durch einen methodischen Rahmen ein wenig verbessert. Es wird aber nicht erheblich verbessert, wenn in diesem methodischen Rahmen immer noch Reproduzierbarkeit fehlt, die Geschwindigkeit zu gering ist oder die Kommunikation von Testergebnissen nicht gelingt. Stattdessen ist es – wenigstens bei den Kollegen Softwareentwicklern – üblich, mit speziellen Testframeworks das erwartete Verhalten eines Softwaresystems zu spezifizieren und dieses erwartete Verhalten programmatisch abzutesten.

Puppet-Katalogtests prüfen zum Beispiel programmatisch, ob der Code kompiliert und ob die Konfiguration einzelner Softwarekomponenten oder komplexerer Softwaremodule so aufgebaut ist, wie es der Absicht des Infrastrukturentwicklers entspricht. Das folgende Beispiel ist ein Auszug aus einem Puppet-Katalogtest für eine Einzelkomponente. Es prüft, ob der Katalog eine Gruppe mit der Spezifikation einer *logrotate*-Regel enthält. Der Test nutzt die Puppet-Klasse `logrotate::rule` (Zeilen 04 und 05), die in den Zeilen 07 bis 13 durch eine Datei mit den aufgeführten Eigenschaften (Zeilen 11 bis 13) realisiert wird:

```
01 describe 'logrotate::rule' do
02   let(:title) { 'nginx' }
03
04   it { is_expected.to
05        contain_class('logrotate::rule') }
06
07   it do
08        is_expected.to
09            contain_file('/etc/logrotate.d/nginx')
10                .with({
11                    'ensure' => 'present',
12                    'owner'  => 'root',
13                    <...>}
14      end
15  end
```

Analog zeigt folgendes Beispiel einen Puppet-Katalogtest auf ein Komposit. Es zeigt eine Reihe von Einzelprüfungen (Zeilen 03 bis 06) mit gegebenenfalls weiterführenden Spezifikationen (Zeile 07):

```
01 describe 'role::webserver::devstage' do
02
03 it { is_expected.to contain_class('nginx') }
04 it { is_expected.to contain_nginx__vhost(...) }
05 it { is_expected.to
                contain_class(logrotate::rule') }
06 it { is_expected.to contain_service('nginx')
07                .with_ensure('running') }
```

Ein solches Vorgehen stellt einen ersten Fortschritt dar. Bei der Entwicklung von Puppet-Modulen, die eine bestimmte Software auf k verschieden Systemen konfigurieren, ist ein solcher Test unersetzlich.

Höherwertige Integrationstests erreicht man aber erst, wenn man nicht nur die entstehenden Arbeitsanweisungen in Form des Katalogs prüft, sondern auch das Ergebnis. Im Gegensatz zum ordnungsgemäßen Verwaltungshandeln interessiert uns sozusagen nicht nur der Nachweis fehlerfreier Anweisung zum Bau von Flughäfen, etwa ob alle Baupläne da sind – sondern ein Flughafen existiert real, und es gibt Flugzeuge, die starten und landen. Also prüft man analog zum Beispiel, ob der Nutzer mit ID 2036 wirklich angelegt werden kann, ob die User-IDs nicht bereits kollidierend belegt ist, oder ob der Dienst *sshd* nicht nur theoretisch laufen können sollen müsste, sondern ob man sich tatsächlich einloggen kann.

Ein bewährtes Verfahren hierzu ist, sich irgendeine Spielmaschine in der *DEV*-Umgebung zu suchen (es setzt sich allmählich durch, nicht sofort am *PROD*-System herumzuprobieren) und diese kaputt zu fummeln, damit die Kollegen Softwareentwickler an der weiteren Arbeit und damit am weiteren Produzieren von Bugs gehindert werden. Geschickter erscheint indes, eine virtuelle Maschine zu erzeugen, deren Ausfall (wirklich!) niemanden stört, auf diese Maschine die Automatisierung zu applizieren und danach mit einem Testframework die Funktionen abzutesten. Nach den Tests kann man die Maschine wieder zerstören und somit in jeder Iteration unbelastet neu testen.

Aus dem Umfeld von *Chef* stammt für Infrastrukturtests das Werkzeug *Kitchen* [4]. Es hat aber viele aktiv entwickelte Plugins, sodass man dieses Werkzeug auch für andere Frameworks wie Puppet und Ansible verwenden kann, sowie für lokale VMs etwa in *VirtualBox* oder *lxd*, für VMs auf dedizierten Maschinen via *VMWare* oder sogar in der Cloud (*EC2*, *GCE*) [5]. Kitchen erzeugt aus einem deklarativen Datensatz eine lokale virtualisierte Umgebung oder eine Container-isolierte Umgebung. Automatisierungscode wird darauf schon während der Entwicklung iterativ und häufig appliziert und erlaubt so eine Laborsituation.

Folgendes Beispiel zeigt den Auszug aus einer Testhost-Deklaration. Ein `provisioner` konfiguriert in den Zeilen 01 bis 08 das Automati-

sierungsframework. Zeilen 10 bis 18 geben über `platforms` die Parametrisierung der Testmaschine an. Schließlich binden die `suites` den Provisionierer und die Plattformen für eine spezielle Rolle inklusive des dazugehörigen Tests zusammen:

```
01 provisioner:
02   name: puppet_apply
03   manifests_path: examples
04   modules_path: 'modules:..'
05   hiera_data_path: hieradata
06   hiera_deep_merge: true
07   puppet_verbose: true
08   puppet_debug: false
09
10 platforms:
11   - name: debian/stretch/lxd-priv
12     driver_plugin: lxd_cli
13     driver_config:
14       image_name: puppet-base-stretch
15       profile:
16         - default
17         - privileged
18         - docker
19
20 suites:
21   - name: webserver
22     provisioner:
23       manifest: nginx/install.pp
```

Kitchen erzeugt also sozusagen eine Art Testlabor. Genauso, wie Kataloge programmatisch Tests spezifizieren und ausführen, lässt sich auch die letztliche Applikation und das Serververhalten in diesem Labor automatisiert testen. Eine Sprachbibliothek, die die Tests selbst beschreibt, stellt beispielsweise das auf Ruby basierende *Serverspec* [6] zur Verfügung. Serverspec ist auch für Menschen und fast schon für nicht-Techniker klar verständlich und prüft das Verhalten einer Software. Innerhalb der Ruby-Libraries kann Serverspec bekannte Unix-Kommandos ausführen (`command`) und matcht deren Output (`stout/stderr`) in bekannter Perl- oder Ruby-artiger Syntax gegen einen regulären Ausdruck. Dies erhöht die Akzeptanz deutlich. Dazu ist es möglich, Tests mit *SSH* zu verteilen [7].

Folgendes Beispiel zeigt einen Serverspec-Test auf dem konfigurierten Host. Zeilen 02 bis 05 beschreiben den Service. Zeilen 07 bis 10 sagen, was der angegebene Port tun soll. Und Zeilen 12 bis 15 spezifizieren, wie die Prozesse des Service aussehen sollen:

```
01 describe 'Check the webserver.' do
02   describe service('nginx') do
03     it { should be_enabled }
04     it { should be_running }
05   end
06
07   describe port(80) do
08     it { should be_listening.with('tcp') }
09     it { should be_listening.with('tcp6') }
10   end
11
12   describe process('nginx') do
13     its(:user) { should eq 'www-data' }
14     its(:count) { should eq 4 }
15   end
16 end
```

Nach demselben Schema testet Serverspec das Verhalten auf einem Client-Host durch Aufruf von Standardkommandos. Im folgenden Beispiel besorgt das das Kommando `curl` und dessen Ausgaben:

```
01 describe 'Check connections from localhost.' do
02  describe command("curl -I http://www.cruwe.de") do
03   its(:stdout) {should match /HTTP\/1.1 301/}
04   its(:stdout) {should match /Location: https:\/\/<...>/}
05   its(:stdout) {should match /Server: nginx/}
06   its(:stdout) {should match /Content-Type: text\/html/}
07   its(:stdout) {should match /X-Clacks-Overhead:
08                                   GNU Terry Pratchett/}
09  end
10 end
```

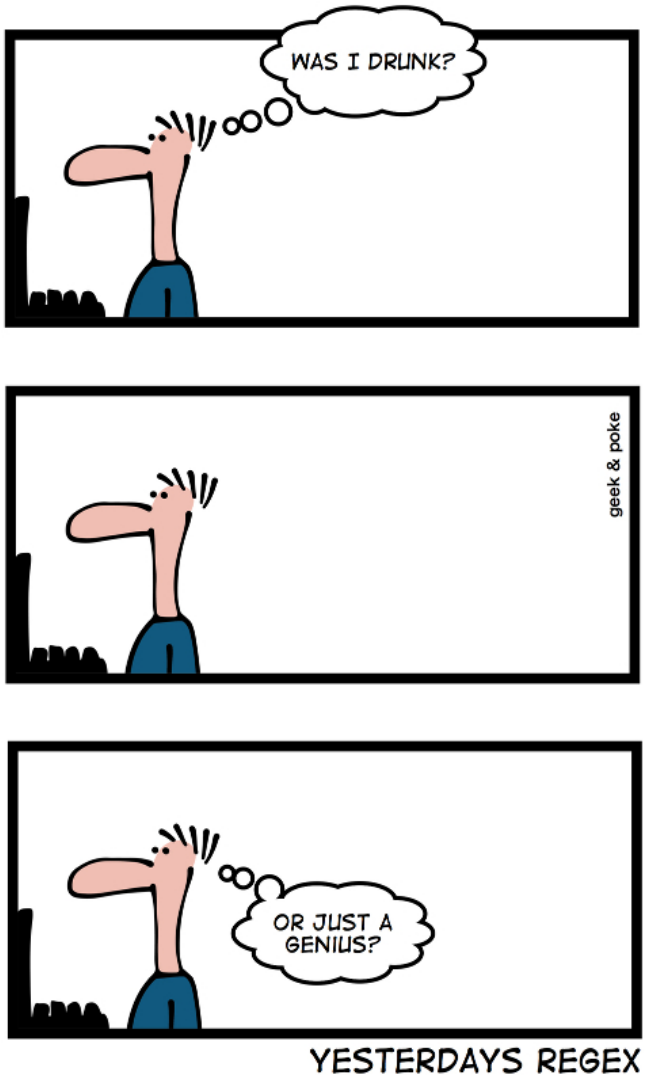
Einfach aneinandergereiht testet Serverspec das Verhalten eines Gesamtsystems aus mehreren Hosts:

```
01 describe 'Check www from localhost.' do
02   <...>
03 end
04
05 describe 'Check app-server from frontend.' do
06   <...>
07 end
```

Damit lassen sich Integrations- und Akzeptanztests auf mehrere Systeme mit demselben Toolset durchführen. Wenn man sich in der Entwicklung von Tests einigermaßen geschickt anstellt, hat man durch den Test sogar das Monitoring seiner Systeme entwickelt, dessen Notwendigkeit im Gegensatz zu Infrastrukturtests nur sehr selten in Frage gestellt wird. Und weil Kitchen angenehm pluggable aufgebaut ist, ist es an dieser Stelle einfach, lokale VM-Provider durch Hosts bei einem Cloud-Provider (*AWS*, *GCE*, *Azure*) ohne Anpassung des sonstigen Testaufbaus zu substituieren, was Integrations- und Akzeptanztest auch hoch skalierend automatisierbar und damit reproduzierbar macht.

Abhängig vom Grad der Abbildungsgenauigkeit des Systems und der Vollständigkeit der Tests verschiebt ein solches Vorgehen den Überraschungsmoment – „Geht es, geht es nicht?“– auf einen Zeitpunkt vor dem tatsächlichen Deployment. Das ist zwar langweilig, erscheint unter Einbeziehung kardiologischer Gesichtspunkte allerdings charmant. Alle Maschinen eines größeren verteilten Systems lokal auf dem Entwicklerlaptop zu simulieren, ist jedenfalls nur in einfachen Fällen möglich. Ein hochverfügbarer *Kubernetes*-Cluster

mit *Weblogic*-Applikationsservern übersteigt beispielsweise erwiesenermaßen die Leistungsfähigkeit vom Laptop des Autoren, dessen darauffolgende Anfrage nach einem Gerät mit 2-Sockel-8-Kerner, 128 GByte RAM und Sherpa für die Akkus in der Budgetverhandlung unverständlicherweise verworfen wurde.



Programmierter Testablauf (Pipeline)

Tests werden in der Praxis in erster Linie auf der lokalen Workstation des Entwicklers ausgeführt. Der manuelle Aufwand lässt sich mit Konstrukten analog des altbewährten Makefile auch hinreichend gut begrenzen. Und manche Betriebe lösen das so entstehende Compliance-Problem mit Gruppendynamik (vulgo: *Code Red*). Die Praxis zeigt aber auch, dass noch nicht einmal solche Tests aus Zeitdruck entweder gar nicht oder nur schluderig durchgeführt werden, oder dass Kollegen die Testergebnisse nicht verstehen. Natürlich ist es wegen der Lokalität der Workstation auch recht schwierig, eine übergreifende Teamsicht auf die Testergebnisse herzustellen. Diese Teamsicht ist aber gerade an Prüf- und Messpunkten im Entwicklungsprozess, den sogenannten *quality gates*, zwingend erforderlich.

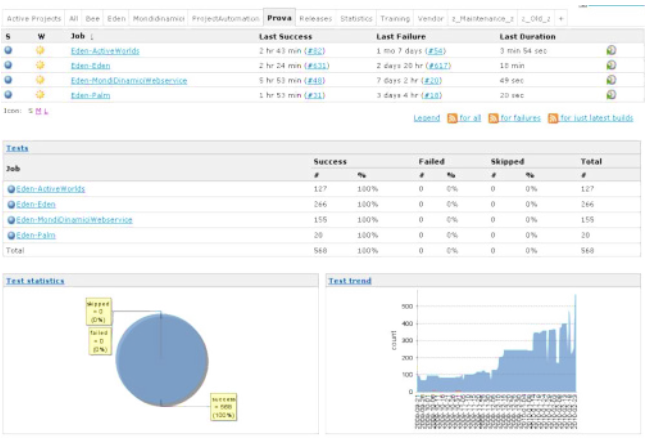


Abbildung 2: Beispiel eines Test-Summary von Jenkins.

Also sollte man den Testablauf genauso programmatisch abbilden, wie die einzelnen Tests selbst. Die Softwareentwickler bedienen sich hierzu sogenannter Continuous-Integration-Server wie dem weit verbreiteten Jenkins. Ein Jenkins-Server überwacht definierte Code-Repositories und führt bei Änderungen am Repo bestimmte Programme oder Skripte aus, die verschiedene Aufgaben vom Testen bis hin zum Deployment wahrnehmen. Nach Abschluss aller Schritte publiziert Jenkins die Ergebnisse auf einem Webinterface tabellarisch und mit URLs (Abbildung 2). Stakeholder können die Ergebnisse abrufen, auswerten und hoffentlich berichtigen.

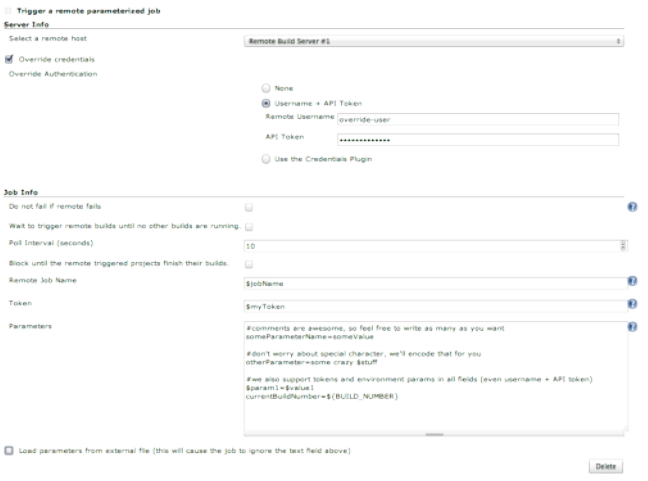


Abbildung 3: Konfiguration eines Jenkins-Jobs.

Die Konfiguration der notwendigen Testjobs über ein Webfrontend (Abbildung 3) passt allerdings nicht besonders gut zu den Methoden der Softwareentwicklung. Auch die Bearbeitung von XML-Files, dem internen Format des Jenkins, eig-

net sich eher als Maßnahme im Outplacement-Prozess (vulgo: Mobbing) denn als zielführende Entwicklungsmethode (Abbildung 4, zur Abschreckung geeignet).

Sinnvoller ist, die Konfiguration von Testjobs ebenfalls im Code-Repository nahe bei den tatsächlichen Tests zu spezifizieren. Eine Methode hierzu ist das Workflow-Plugin und die *Job-DSL* von Jenkins, die beide seit Jenkins 2.0 allgemein zur Verfügung stehen [8]. Hierbei sind die Arbeitsschritte (Stages) in Groovy in einem sogenannten Jenkinsfile programmiert und werden bei bestimmtem Ereignissen ausgelöst, meist bei einer Operation auf ein Repository. Der Zusammenhang mit der programmierten Testausführung ist also folgender: Kitchen beschreibt und erzeugt Laborumgebungen für Tests. Mit Serverspec lassen sich die Tests darin beschreiben. Die Jenkins-Job-DSL und das Jenkinfile beschreiben nun den sachlich und zeitlich geordneten Ablauf der Tests in den dazugehörigen Laborumgebungen.

```
<?xml version='1.0' encoding='UTF-8'?>
<org.jenkinsci.plugins.workflow.multibranch.WorkflowMultiBranchProject plugin="workflow-multibranch@2.9.2">
<actions/>
<description></description>
<properties/>
<folderViews
 class="com.cloudbees.hudson.plugins.folder.views.DefaultFolderViewHolder"
 plugin="cloudbees-folder@5.14">
 <...>
</folderViews>
<healthMetrics>
 <...>
</healthMetrics>
<orphanedItemStrategy
 class="com.cloudbees.hudson.plugins.folder.computed.DefaultOrphanedItemStrategy"
 plugin="cloudbees-folder@5.14">
 <pruneDeadBranches>true</pruneDeadBranches>
 <daysToKeep>0</daysToKeep>
 <numToKeep>0</numToKeep>
</orphanedItemStrategy>
<...>
```

Abbildung 4: Interne XML-Darstellung eines Jenkins-Jobs.

In folgendem Ausschnitt aus einem Jenkinsfile ist zu sehen, dass die Arbeitsschritte in Form von *try-catch-finally*-Ausdrücken spezifiziert sind (Zeilen 02 und 17). Zeile 03 deklariert die spätere grafische Darstellung. Zeilen 06 und 08 bauen die Umgebung auf. Die Zeilen 10 bis 14 erzeugen (`create`), provisionieren (`converge`) und testen (`verify`) die Applizierung eines Codes auf einen simulierten Host, hier mit Docker. Schließlich stellt Zeile 18 den durch Kitchen erzeugten XML-Output dem Jenkins-JUnit-Publisher auf einer Webseite zur Verfügung. Die grafische Darstellung der einzelnen Arbeitsschritte (Stages) und die Gesamtzusammenfassung des Tests sind in den Abbildungen 5 und 6 zu sehen.

```
01 stage('Integrate') {
02  try{
03   wrap([$class: 'AnsiColorBuildWrapper',
                   'colorMapName': 'XTerm']) {
04    sh '''#!/bin/bash -l
05
06    rvm use 2.2
07
08    eval "$(ssh-agent)"
09
10    bundle exec kitchen create docker || true
11    ssh-add ./.kitchen/docker_id_rsa
12    bundle exec kitchen create docker
13    bundle exec kitchen converge docker
14    bundle exec kitchen verify docker
15    '''
16   }
17  } catch (err){ } finally {
18   junit '**/reports/xml/serverspec-result.xml'
19  }
20 }
```

Damit die Testanweisungen nahe an den Tests liegen, die wiederum nahe am Automatiserungscode liegen sollten, liegt das Jenkinsfile im Code-Repository des Automatisierungscodes. Die Testanweisungen werden dann bei Registrierung des Repos im Jenkins ausgelesen, iim Wesentlichen durch Angabe eines http(s)-Endpoints für das Repos. So wird aus dem Repository die zugehörige Testkonformation für jeden Branch des Repositories automatisch erzeugt, in dem eine Jenkinsfile liegt. Dies erleichtert die Automatisierung der Test- und gegebenenfalls Deployment-Pipeline erheblich. Und dies bewirkt Reproduzierbarkeit, Geschwindigkeit der Ausführung, Verständlichkeit der Tests und ihrer Ergebnisse sowie einen Testhorizont jenseits der kognitiven Froschperspektive des einzelnen Mitarbeiters.

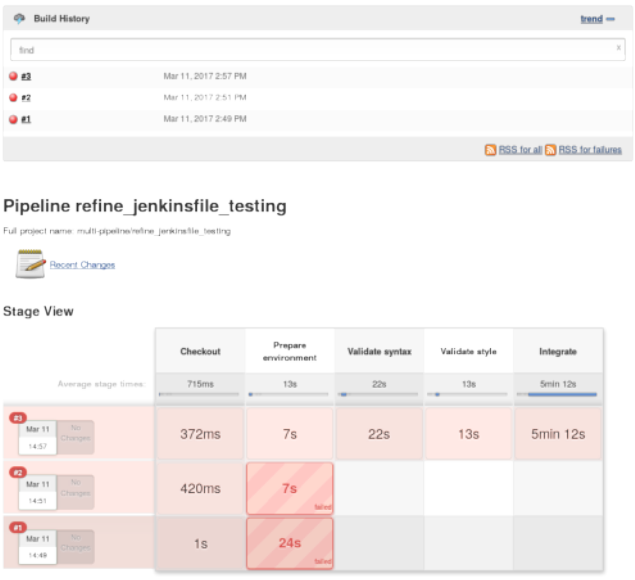


Abbildung 5: Darstellung der Stage-Ergebnisse.

(Keine) Alternative zum Jenkinsfile

Selbstverständlich ließen sich auch andere Techniken zur automatischen Durchführung von Tests nutzen. Die Test-Pipeline mit dem Jenkinsfile hat nicht nur zufällig große Ähnlichkeit mit *git-hooks*

im Repository eines Entwicklungsprojektes. Git-Hooks steuern vor bestimmten Schritten (commit, push, merge) Tests an, die eine bestimmte Code-Qualität im Repo erzwingen und umgekehrt die Verschmutzung des Repos durch schlechten Code verhindern. Außerdem gibt es ganze Produkte, die Continuous-Integration-Pipelines zur Verfügung stellen, zum Beispiel die *gitlab-ci* von Gitlab.com oder die *travis-ci*, die durch enge Integration mit Github.com bekannt ist. Beide sind allerdings an einen speziellen Git-Repository-Provider gekoppelt. Man könnte hierbei daher von Git-Hooks auf Speed sprechen.

Im Wesentlichen ist das Angebot des Jenkins-Workflow-Plugins zu (ausgeprägten) Hooks äquivalent, somit zu den CI-Pipelines der Repository-Provider. Alle können in der jeweiligen Sprache der Pipeline beliebige Programme ansteuern, Programm-Output aufbereiten und grafisch darstellen. Alle Varianten gewährleisten, dass Testcode und Testausführung sehr eng am eigentlichen Code entwickelt und fortgeschrieben werden können.

Git-Hooks haben sich für diese Zwecke allerdings nicht durchgesetzt. Am stärksten fehlt ihnen wohl ein (buntes?) Webinterface zur Publikation von Testergebnissen. Auch die CI-Lösungen der Repository-Provider sieht der Autor dieser Zeilen skeptisch: Die Fragmentierung der Pipeline-Lösungen der Repositories ist wirklich bemerkenswert. Mit *travis.ci* für Github.com, *gitlab-ci* von Gitlab.com und der Bitbucket-Implementierung von *Atlassian* existieren bereits drei miteinander inkompatible CI-Lösungen, jeweils mit vergleichbarem Verbreitungsgrad. Im Gegensatz dazu ist Jenkins in fast allen Entwicklerprojekten anzutreffen. Damit ist Jenkins der de-facto-Standard der CI-Server. Mit der Job-DSL und dem Jenkinsfile löst er sich von grafischer Konfiguration und schließt somit die Lücke, die vorher die Repositories mit inkompatiblen Eigenentwicklungen zu schließen versuchten.

Ohne formale Tests und formale Korrektheitsnachweise ist es schwer sich vorzustellen, wie man die fortschreitende Integration von IT in Geschäftsprozesse und in ganze Unternehmen (Stichwort Industrie 4.0) beherrschbar gestalten soll. Von Qualität und Störungsfreiheit ist dabei noch gar nicht die Rede. Welches Werkzeug zum Einsatz kommt, ist letzten Endes egal, solange nur endlich automatische Tests auch in der Infrastrukturentwicklung selbstverständlich werden. Zwar versagen auch in der Softwareentwicklung die Test- und Abnahmeprozeduren noch zu oft. Sie sind aber zumindest selbstverständlich und etabliert. Die methodische Konvergenz von Entwicklung und Operations stimmt hoffnungsvoll, dass auch wir das schaffen werden.

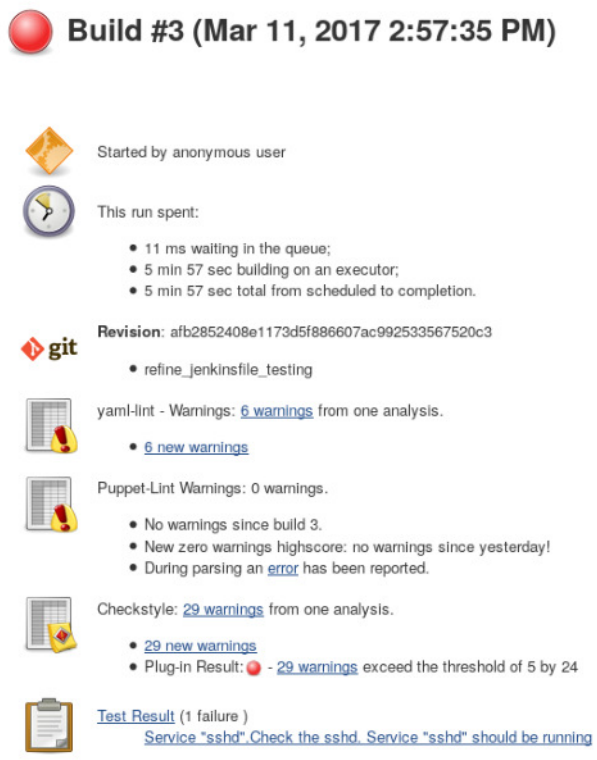


Abbildung 6: Darstellung der gesamten Testergebnisse.

Links

[1] CI-Server *Jenkins*: https://jenkins.io
[2] Bertrand Meyer: Object-Oriented Software Construction. Upper Saddle River, Prentice Hall 1998 (2nd ed.).
[3] Andrew S. Tanenbaum and Maarten van Steen: Distributed Systems; Principles and Paradigms. Upper Saddle River, Prentice Hall 2006 (2nd rev. ed.).
[4] *Kitchen*: https://kitchen.ci
[5] Ökosystem von *Kitchen*: https://github.com/test-kitchen/test-kitchen/blob/master/ECOSYSTEM.md
[6] *Serverspec*: http://serverspec.org
[7] *Serverspec*-Tests mit *SSH* verteilen: http://serverspec.org/advanced_tips.html
[8] *Jenkins* 2.0: https://jenkins.io/2.0/

Über Christopher

Christopher J. Ruwe (Jahrgang 1928) ist freiberuflicher IT-Consultant mit den Schwerpunkten Unix und DevOps. In den aktuellen Bestrebungen vieler größerer Unternehmungen, IT-Systeme „in die Cloud zu bringen“, berät er technisch und organisatorisch zu Fragen des Deployments, der Konfiguration und des Betriebes von Software in hoch und schnell skalierenden Umgebungen.

Shellskripte mit Aha-Effekt VIII
Bash-Arrays

Auch an denjenigen, die schon über 20 Jahre mit der Bash arbeiten, sind Bash-Arrays bisher vielleicht vorbeigegangen. Dieser Artikel zeigt, wie man Arrays auch in der Bash verwenden kann, so wie von Perl und anderen Sprachen bekannt.

von **Jürgen Plate**

Any sufficiently advanced technology is indistinguishable from magic.

Arthur C. Clarke

Any sufficiently advanced bug is indistinguishable from a feature.

Rich Kulowiec

Arrays ermöglichen, eine geordnete Folge von Werten eines bestimmten Typs zu speichern und zu bearbeiten. Allerdings erlaubt die Bash nur eindimensionale Arrays. Dabei werden zwei grundlegende Typen unterschieden: indizierte und assoziative Arrays.

Indizierte Arrays verwenden positive Integer-Zahlen als Index. Die Indizes sind meist *sparse*, also nicht unbedingt zusammenhängend. Wie in C und vielen anderen Sprachen beginnen die ganzzahligen Indizes bei 0. Indizierte Arrays gab es zuerst bei der Bourne-ähnlichen Shell, der *ksh*. Sie tragen immer das Attribut `-a`.

Assoziative Arrays (manchmal als *Hash* bezeichnet) verwenden beliebige nichtleere Zeichenketten als Index. Diese Arrays sind immer ungeordnet, sie assoziieren nur Index-Wert-Paare. Wenn man mehrere Werte daraus abruft, kann man sich nicht darauf verlassen, dass sie in derselben Reihenfolge wiedergegeben werden, in der sie eingefügt wurden. Assoziative Arrays tragen immer das Attribut `-A`. Im Gegensatz zu indizierten Arrays müssen sie immer ausdrücklich deklariert werden.

Indizierte Arrays

Ein indiziertes Array (Feld, Vektor, Reihung) ermöglicht die Verarbeitung mehrerer gleichartiger Elemente, wobei auf jedes Element über seinen Index eindeutig zugegriffen werden kann. Oft verwendet man Arrays in Schleifen, um nicht auf eine Reihe von einzelnen Variablen zurückgreifen zu müssen.

Die folgenden Möglichkeiten versehen eine Variable mit Array-Attribut, wobei beim indizierten Array keine Deklaration notwendig ist. Diese erfolgt meist implizit durch die Wertzuweisung:

- `ARRAY=()` deklariert ein indiziertes Array namens *ARRAY* und initialisiert es als leer. Dies kann auch zum Leeren eines bestehenden Arrays verwendet werden.
- `ARRAY[0]=xxx` besetzt das erste Element eines indizierten Arrays (Wertzuweisung). Wenn noch kein Array namens *ARRAY* existiert, wird es erzeugt.
- `declare -a ARRAY` deklariert ein indiziertes Array namens *ARRAY*. Ein bereits vorhandenes Array wird dadurch nicht initialisiert.

Wenn Sie einer Array-Komponenten einen Wert zuweisen wollen, erfolgt dies genauso wie bei den normalen Shell-Variablen. Sie geben lediglich noch den Feldindex eingeschlossen in eckige Klammern an. Insofern unterscheidet sich die Bash nicht von C, Perl, Python oder anderen Programmiersprachen. Sie müssen lediglich daran denken, dass rechts und links vom Gleichheitszeichen kein Leerzeichen stehen darf. Das folgende Beispiel belegt die dritte (!) Array-Komponente mit dem String *zwo*:

```
array[2]=zwo
```

Was geschieht in diesem Fall mit den Komponenten `array[0]` und `array[1]`? Bei der Bash-Programmierung dürfen Arrays Lücken enthalten (wie auch in Perl oder anderen Interpretersprachen). Im Unterschied zu anderen Programmiersprachen muss man in der Bash das Array auch nicht vor der Verwendung deklarieren. Man sollte es jedoch trotzdem zu tun, denn einerseits kann man damit angeben, dass die Werte innerhalb eines Arrays zum Beispiel als Integer gültig sein sollen (`typeset -i array`), und andererseits lässt sich so ein Array mit Schreibschutz versehen (`typeset -r array`).

Häufig will man beim Anlegen eines Arrays dieses mit Werten initialisieren. In der Bash erfolgt dies durch Klammern der Werte:

```
array=(Merkur Venus Erde Mars Jupiter Saturn)
```

Bei der Zuweisung beginnt der Feldindex automatisch bei 0, also enthält `array[0]` den Wert *Merkur*. Aber ist es auch möglich, eine Zuweisungsliste an einer anderen Position zu beginnen, indem der Startindex angegeben wird:

```
array=([2]=Pluto Eris ...)
```

Diese Methode ist jedoch nicht geeignet, neue Elemente an ein Array anzuhängen. Ein existierendes Array wird immer komplett überschrieben. Die Anzahl der Elemente, die Sie einem Array übergeben können, ist bei der Bash beliebig. Als Trennzeichen zwischen den einzelnen Elementen dient das Leerzeichen.

Auch aus einer Datei lässt sich ein Array erzeugen. Das Kommando `cat` und `$(...)` heben den Dateiinhalt auf die Kommandozeile und damit in die Initialisierungsliste:

```
array=($(cat "datendatei"))
```

Der Zugriff auf die einzelnen Komponenten eines Arrays erinnert an C oder andere Sprachen. Er erfolgt mittels

```
${array[index]}
```

Einziger Unterschied zu anderen Sprachen: Für die Referenz auf den Array-Inhalt sind geschweifte Klammern zu verwenden, da die eckigen Klammern ja schon Metazeichen der Bash sind (Bedingung analog dem `test`-Kommando) und sonst eine Expansion auf Shell-Ebene versucht würde.

Alle Elemente eines Arrays lassen sich folgendermaßen ausgeben, ähnlich wie bei Kommandozeilenparametern:

```
echo ${array[*]}
echo ${array[@]}
```

Der Unterschied ist der gleiche wie bei den Kommandozeilenparametern (`$*` und `"$"`). Ohne Anführungszeichen expandieren beide Ausdrücke gleich, mit Anführungszeichen expandiert `${array[*]}` zu allen Elementen in einem, während `${array[]}` zu den einzelnen Elementen in Anführungszeichen expandiert. Das ist insbesondere bei *for*-Schleifen wichtig, die über die einzelnen Elemente iterieren.

`${!array[*]}` listet alle Indizes des Arrays auf. Diese Liste ist interessant, wenn das Array Lücken bei den Indizes aufweist. Das folgende Beispiel listet Index und Inhalt auf:

```
01 for IND in ${!array[*]}
02 do
03   echo "$IND ${array[$IND]}"
04 done
```

Die Anzahl der belegten Elemente im Array erhält man wie folgt (auch diese Syntax ähnelt der Kommandozeile bzw. der Shell-Variablen `$#`):

```
echo ${#array[*]}
```

Wer feststellen will, wie lang eine Array-Komponente ist, geht genauso vor wie beim Ermitteln der Anzahl belegter Elemente im Array, nur dass anstelle des Sternchens der entsprechende Feldindex zum Einsatz kommt:

```
echo ${#array[1]}
```

Array-Beispiele

Jetzt wird es aber Zeit für einige Beispiele. Zunächst definieren wir ein Array und füllen es mit Werten. Danach probieren wir einige der oben beschriebenen Ausgaben:

```
$ array=(Merkur Venus Erde Mars Jupiter Saturn\
       Uranus Neptun)
```

Fünfte Komponente ausgeben:

```
$ echo ${array[4]}
Jupiter
```

Wieviele Planeten sind es (ohne Pluto)?

```
$ echo ${#array[*]}
8
```

Wieviele Buchstaben hat das Wort *Erde*?

```
$ echo ${#array[2]}
4
```

Auch Teile eines Array lassen sich selektieren. So liefern die folgenden Ausdrücke verschiedene *Slices* des Arrays:

- die Array-Komponenten von 'index' bis 'index+length-1' inklusive: `${array[@]:index:length}`
- die Array-Komponenten von '0' bis 'length-1' inklusive: `${array[@]::length}`
- die Array-Komponenten von 'index' bis zum Ende: `${array[@]:index}`

Auch dazu einige Beispiele mit den zugehörigen Ausgaben:

```
$ array=(Merkur Venus Erde Mars \
       Jupiter Saturn Uranus Neptun)

$ echo "${array[@]:2}"
Erde Mars Jupiter Saturn Uranus Neptun

$ echo "${array[@]:1:3}"
Venus Erde Mars

$ echo "${array[@]::1}"
Merkur
```

Löschen, kopieren und konkatenieren

Das Löschen eines Arrays oder einer Komponente sieht genauso aus wie bei den Shell-Variablen. Das komplette Array löscht `unset array`. Einen Index anzugeben, löscht einzelne Elemente: `unset array[index]`. Das funktioniert aber nicht immer, wie folgendes Beispiel zeigt:

```
$ array=(Merkur Venus Erde Mars \
       Jupiter Saturn Uranus Neptun)

$ INDEX=4

$ echo ${#array[@]}
8

$ echo ${array[$INDEX]}
Jupiter

$ unset ${array[$INDEX]}

$ echo ${#array[@]}
8
```

Oha! Hier sollte nun eine *7* stehen, weil `unset` ja den Jupiter gelöscht hat. Mal sehen:

```
$ echo ${array[$INDEX]}
Jupiter
```

Der Fehler liegt darin, dass die Bash ein Array als sogenannten Hash speichert (*sparse*). Man kann sich also nicht darauf verlassen, dass eine Komponente nach dem `unset` tatsächlich weg ist.

Zum Kopieren eines kompletten Arrays können wir entweder eine Schleife programmieren, in der jede Komponente des einen Arrays dem anderen Array zugewiesen wird. Es geht aber auch mit weniger Aufwand. Dafür kombinieren wir das Auflisten aller Elemente des einen Arrays mit dem Zuweisen einer Liste mit Werten an ein zweites Array:

```
array_neu=( ${array[@]} )
```

Auch hierzu ein Beispiel als interaktive Shell-Sitzung:

```
$ array=(Merkur Venus Erde Mars \
       Jupiter Saturn)

$ echo ${array[@]}
Merkur Venus Erde Mars Jupiter Saturn
```

array auf *yarra* kopieren und *yarra* ausgeben:

```
$ yarra=( ${array[@]} )

$ echo ${yarra[@]}
Merkur Venus Erde Mars Jupiter Saturn
```

Das Aneinanderhängen zweier Arrays geht genauso wie das Konkatenieren von Strings. Wir bilden einfach eine Initialisierungsliste aus beiden Arrays, die wir dann einem neuen Array zuweisen:

```
Gesamt=("${Array1[@]}" "${Array2[@]}")
```

Zwei Praxisbeispiele

Eine Liste der Dateinamen des aktuellen Verzeichnisses bekommen wir übrigens mit der folgenden Zuweisung:

```
Dateien=(*)
```

Zum Abschluss eine praktische Nutzanwendung für Arrays. Beim Raspberry Pi müssen die GPIO-Ports für die parallele Ein- und Ausgabe zu Beginn initialisiert werden. Sind es zahlreiche Ports, macht man das am besten in einer Schleife. Nur steckt die Schleife meist in den Tiefen des Skripts, und man muss für das Ändern oder Hinzufügen von Ports erst einmal suchen. Viel wartungsfreundlicher ist es, die Portzuordnung am Anfang des Skripts (oder sogar in einer Konfigurationsdatei) vorzunehmen. Das sieht dann beispielsweise folgendermaßen aus – hier ordnen wir Ports LEDs zu:

```
GPIO=(27 22 19 17 13 6 5)
```

Die Initialisierung der Ports erfolgt nun in einer Schleife über die Array-Komponenten (Zeile 02). Nach Expansion durch die Shell lautet diese Zeile `for P in 27 22 19 17 13 6 5`, also eine ganz normale Schleifenanweisung. Die Zeilen 04 bis 11 sorgen für die Initialisierung der GPIO-Ports. Der genaue Inhalt ist an dieser Stelle nicht so wichtig – hier ist nur die Anwendung der Laufvariablen `P` für *Port* von Belang:

```
01 SCG=sys/class/gpio
02 for P in ${GPIO[@]}
03 do
04   if [ ! -f /$SCG/gpio${P}/direction ] ; then
05     echo "Initialisiere GPIO $P"
06     echo "$P" > /$SCG/unexport 2> /dev/null
07     echo "$P" > /$SCG/export
08     echo "out" >/$SCG/gpio${P}/direction
09     echo "0" >/$SCG/gpio${P}/value
10     chmod 666 /$SCG/gpio${P}/direction
11     chmod 666 /$SCG/gpio${P}/value
12   fi
13 done
```

Übrigens ließen sich mit dem Befehl `PARAM=($@)` auch alle Kommandozeilen-Parameter in einem Array speichern und dann wahlfrei darauf zugreifen. Das ist manchmal einfacher als andere Methoden des Zugriffs auf die Parameter. Um das Beispiel oben weiterzuspinnen, soll das folgende Skript für jede LED `0` oder `1` angeben und diese Parameter der Reihe nach auf den oben definierten GPIO-Ports ausgeben (also der erste Parameter auf Port 27, der zweite auf Port 22, und so weiter). Als Laufvariable dient der Index des Parameter-Arrays (Zeile 02). In der Schleife wird dann der I-te Parameter auf den I-ten GPIO-Port ausgegeben (Zeile 04 und 05):

```
01 PARAM=( $@ )
02 for I in ${!PARAM[@]}
03 do
04   echo ${PARAM[$I]} > \
05     /sys/class/gpio/gpio${GPIO[$I]}/value
06 done
```

Assoziative Arrays

Seit Version 4 gibt es in der Bash assoziative Arrays (manchmal als *Hash* bezeichnet). Man kann sie sich als Verknüpfung von zwei Arrays vorstellen, wobei eines die Daten enthält und das andere die Schlüssel, welche die einzelnen Elemente des Datenarrays indizieren. Im Gegensatz zu indizierten Arrays müssen assoziative Arrays immer ausdrücklich deklariert werden. Die Anweisung `declare -A ARRAY` deklariert ein assoziatives Array namens *ARRAY*.

Als Index kennen sie beliebige nichtleere Zeichenketten. Assoziative Arrays sind immer ungeordnet, sie assoziieren nur Index-Wert-Paare. Wer mehrere Werte aus dem Array abruft, kann sich nicht darauf verlassen, dass diese in derselben Reihenfolge wiedergegeben werden, in der sie eingefügt wurden.

Einen Wert weisen wir genauso zu wie bei den indizierten Array, nur dass Sie hier keine Ganzzahl als Index verwenden, sondern eine Zeichenkette (*Key*). Zum Beispiel:

```
01 declare -A namen
02 namen[Donald]=Duck
03 namen[Daniel]=Düsentrieb
04 namen[Gustav]=Gans
```

Im Key dürfen auch Leerzeichen oder andere exotische Zeichen auftauchen. Dann ist der Key aber in Anführungszeichen zu setzen, beispielsweise:

```
05 namen["Onkel Dagobert"]=Duck
```

Elemente des Index können Leerzeichen sogar am Anfang oder Ende des Strings enthalten. Index-Elemente, die nur aus Leerzeichen bestehen, sind jedoch nicht zulässig.

Ähnlich wie bei Perl oder PHP lassen sich auch mehrere Elemente in einem Rutsch definieren:

```
01 declare -A namen
02 namen=(
03  [Donald]=Duck
04  [Daniel]=Düsentrieb
05  [Gustav]=Gans
06 )
```

Auch beim Zugriff auf den Inhalt assoziativer Arrays verfährt man wie bei indizierten Arrays. Dabei sei nochmals darauf hingewiesen, dass die Reihenfolge, in der die Elemente im Array gespeichert werden, nicht durch die Reihenfolge der Eintragung festgelegt ist. Um an den Inhalt eines Hashes zu gelangen, gibt man den Key an:

```
echo ${namen[Donald]}
```

Anstelle eines konstanten Keys lässt sich auch eine Shell-Variable verwenden, zum Beispiel:

```
$ Key=Gustav
$ echo ${namen[$Key]}
Gustav
```

Ist die Variable `Key` nicht definiert, liefert die Bash die Fehlermeldung `bad subscript`. Es ist also wichtig, dass alle Variablen, die als Index verwendet werden, auch definiert sind. Ist dagegen der Index im Array nicht enthalten, liefert `${namen[$Key]}` einen leeren String. Die Existenz eines Eintrags testet folgendes Skript:

```
01 if [ ${namen[$Key]+_} ]; then
02   echo "$Key gefunden"
03 else
04   echo "$Key nicht gefunden"
05 fi
```

Zeile 01 verwendet die Parametersubstitution der Bash. Wenn das Hash-Element existiert, wertet die Bash den Ausdruck als _ aus, andernfalls als Nullzeichenfolge. Wer will, kann auch ‚Jawoll' oder irgend etwas anderes anstatt des Unterstrichs innerhalb der geschweiften Klammern verwenden.

Es gibt noch weitere Möglichkeiten in der Bash, auf Existenz oder Fehlen von Hashes zu testen, wobei sich alle auf die Parametersubstitution stützen. Das folgende Protokoll einer Sitzung zeigt zwei Alternativen:

```
$ declare -A ax
$ ax[foo]="bar"
$ echo ${ax[foo]:-FEHLT}
bar
$ echo ${ax[nix]:-FEHLT}
FEHLT
$ echo ${ax[foo]:+EXISTIERT}
EXISTIERT
```

Alle **Indexschlüssel** (Keys) erhält man mittels `${!namen[@]}`, und auf die Werte kann mit `${namen[@]}` zugegriffen werden. Der Befehl `echo ${!namen[@]}` ergibt dann für das obige Beispiel:

```
Daniel Gustav Donald
```

Die **Werte** dazu liefert der Befehl `${namen[@]}`. Man erhält die Ausgabe:

```
Düsentrieb Gans Duck
```

Die **Länge** eines assoziativen Arrays (im Prinzip ist dies die Anzahl der Schlüssel) erhält man mittels `${#ARRAY[@]}`. Im Beispiel von oben erhält man also den Wert 3.

Am häufigsten wird man aber Schleifenkonstruktionen antreffen, die über Keys oder Werte iterieren. Die *for*-Schleife in Zeile 01 läuft über die Keys:

```
01 for i in "${!namen[@]}"
02 do
03   echo $i
04 done
```

Ohne das Ausrufezeichen vor dem Array-Namen iteriert die Schleife in Zeile 01 über die Werte:

```
01 for w in "${namen[@]}"
02 do
03   echo $w
04 done
```

Die folgende Schleife gibt in Zeile 03 alle Key-Werte-Paare aus:

```
01 for i in "${!namen[@]}"
02 do
03   echo "$i --> ${namen[$i]}"
04 done
```

Startet man das letzte Skript, erhält man als Ausgabe:

```
Daniel --> Düsentrieb
Gustav --> Gans
Donald --> Duck
```

Die Reihenfolge entspricht nicht der Reihenfolge der Eintragung. Das kann man auch mit einem weiteren Versuch demonstrieren. Fügt man nun noch den Onkel Dagobert hinzu, liefert das Programm als Ausgabe:

```
Onkel Dagobert --> Duck
Daniel --> Düsentrieb
Gustav --> Gans
Donald --> Duck
```

Demo-Adressverwaltung

Jetzt wird es wieder Zeit für ein etwas längeres Beispiel. Das folgende Programm stellt eine einfache Adressverwaltung dar, wobei die Namen den Schlüssel bilden und als Wert dann die Adresse im assoziativen Array eingetragen wird.

Zur Verwaltung des Arrays stehen zwei einfache Funktionen zur Verfügung. Nach der Deklaration des Arrays `Adresse` als Hash in Zeile 01 definieren die Zeilen 03 bis 07 die Funktion `Speichere_Adresse`, die Namen und Adresse als Parameter übernimmt und im Hash ablegt. Die zweite Funktion `Suche_Adresse` stellt zunächst fest, ob der Name als Schlüssel gespeichert ist, und gibt daraufhin die Adresse aus (Zeilen 09 bis 14). Findet das Skript den Namen nicht, gibt es eine Fehlermeldung aus und setzt den Rückgabewert auf 99 (Zeilen 15 bis 19).

Die Zeilen 21 bis 25 lesen die Adressdaten aus der Datei `adressen.csv` ein. Dort stehen pro Zeile Name und Adresse getrennt durch ein Semikolon. Dieses dient auch nach dem Einlesen der Zeile als Trennzeichen für die beiden `cut`-Befehle (Zeilen 22 und 23), welche die beiden Felder an `Speichere_Adresse` übergeben. Die Zeilen 27 bis 33 fragen dann die Adressdaten mittels `Suche_Adresse` ab.

```
01 declare -A Adresse
02
03 Speichere_Adresse ()
04   {
05   Adresse["$1"]="$2"
06   return $?
07   }
08
09 Suche_Adresse ()
10   {
11   if [ ${Adresse[$1]+_} ]
12   then
13     echo "$1's Adresse lautet ${Adresse[$1]}."
14     return $?
15   else
16     echo "$1's Adresse ist nicht vorhanden."
17     return 99
18   fi
19   }
20
21 while read LINE; do
22   NAME=$(echo $LINE | cut -d';' -f1)
23   ADDR=$(echo $LINE | cut -d';' -f2)
24   Speichere_Adresse "$NAME" "$ADDR"
25 done < adressen.csv
26
27 Suche_Adresse "Donald Duck"
28 Suche_Adresse "Harry Potter"
29 Suche_Adresse "Jules Maigret"
30 Suche_Adresse "Sherlock Holmes"
31 Suche_Adresse "Herman Munster"
32 Suche_Adresse "Thomas A. Edison"
33 Suche_Adresse "John Doe"
```

Das folgende Listing zeigt die Ausgabe des Programms:

```
Donald Duck's Adresse lautet Flower Road 13, \
  Duckburg, Callisota.
Harry Potter's Adresse lautet Privet Drive 4, \
  Little Whinging, UK.
Jules Maigret's Adresse lautet Boulevard \
  Richard-Lenoir 132, Paris, FR.
Sherlock Holmes's Adresse lautet Baker Street \
  221b, London, UK.
Herman Munster's Adresse lautet Mockingbird \
  Lane 1313, Mockingbird Heights, USA.
Thomas A. Edison's Adresse lautet 37 Christie \
  Street, Menlo Parc, NJ, USA.
John Doe's Adresse ist nicht vorhanden.
```

Lösch- und Umwandlungsoperationen

Um ein assoziatives Array zu löschen, genügt es nicht, das Array neu zu deklarieren. Diese Re-Deklaration bewirkt eigentlich gar nichts. Das folgende Programm gibt zweimal `bar` aus:

```
01 declare -A ARRAY
02 ARRAY[foo]=bar
03 echo ${ARRAY[foo]}
04
05 declare -A ARRAY
06 echo ${ARRAY[foo]}
```

Stattdessen ist das assoziative Array mittels `unset` gänzlich aus dem Gedächtnis der Bash zu entfernen und dann natürlich neu zu deklarieren. Beim Programm kommt so die Zeile 05 hinzu:

```
01 declare -A ARRAY
02 ARRAY[foo]=bar
03 echo ${ARRAY[foo]}
04
05 unset ARRAY
06 declare -A ARRAY
07 echo ${ARRAY[foo]}
```

Meist will man jedoch nicht das ganze Array löschen, sondern nur einzelne Einträge. Dies erfolgt fast auf die gleiche Art und Weise wie oben, jedoch mit dem Unterschied, dass man bei `unset` den Schlüssel des zu löschenden Eintrags angibt. Beim folgenden Programm löscht Zeile 11 den Eintrag *zwei*:

```
01 declare -A ARRAY
02 ARRAY=([eins]=one
03        [zwei]=two
04        [drei]=three)
05
06 echo ${ARRAY[eins]}
07 echo ${ARRAY[zwei]}
08 echo ${ARRAY[drei]}
09 echo ""
10
11 unset ARRAY[zwei]
12 echo ${ARRAY[eins]}
13 echo ${ARRAY[zwei]}
14 echo ${ARRAY[drei]}
```

Enthält der Schlüssel Leerzeichen oder andere Sonderzeichen, muss er in Gänsefüßchen oder Apostrophe eingeschlossen werden, was auch gilt, wenn man den Schlüssel als Variable übergibt, zum Beispiel:

```
unset ["Donald Duck"]
unset ['Donald Duck']
unset ["$Key"]
```

Man kann sogar ein assoziatives Array in ein indiziertes Array konvertieren – wenn auch mit unvorhersagbarem Ergebnis, was die Reihenfolge betrifft. Folgendes Beispiel deklariert ein assoziatives Array,füllt es und gibt es aus (Zeilen 01 bis 09). Zeile 11 erzeugt die Zuweisung aus `ARRAY` ein indiziertes Array `KEYS`. Zu beachten sind die runden Klammern, die den Effekt bewirken:

```
01 declare -A ARRAY
02 ARRAY=(["eins"]="one"
03        ["zwei"]="two"
04        ["drei"]="three")
05
06 echo "Ausgabe ARRAY"
07 for K in ${!ARRAY[@]}; do
08   echo "$K --> ${ARRAY[$K]}"
09 done
10
11 KEYS=( ${!ARRAY[@]} )
12
13 echo ""
14 echo "Ausgabe KEYS"
15 for K in 0 1 2; do
16   echo "$K --> ${KEYS[$K]}"
17 done
```

Die Ausgabe des indizierten Arrays in den Zeilen 14 bis 17 zeigt, dass die Reihenfolge der Speicherung im Hash wieder mal nicht der Eingabereihenfolge entsprach, und daher das indizierte Array vielleicht nicht ganz den Vorstellungen des Programmierers entspricht:

```
Ausgabe ARRAY
zwei --> two
eins --> one
drei --> three

Ausgabe KEYS
0 --> zwei
1 --> eins
2 --> drei
```

Anwendungen für Hashes

Wofür sind die assoziativen Arrays (Hashes) eigentlich wirklich gut? Hashes sind nützlicher als Arrays, wenn man auf ein Element lieber mit einer expliziten Bezeichnung (einem Schlüssel) als mit einem bloßen numerischen Index zugreifen möchte. Eine Anwendung ist die Prüfung, ob beispielsweise Schlüssel(worte) in einer Datei vorhanden sind. In solche Fällen kommt es nur auf den Schlüssel und nicht auf den Wert an. Ein typisches Beispiel wäre das Erstellen einer Wortliste aus einer Datei. Hierbei kommt es nur auf die Worte an und nicht auf deren Häufigkeit.

Das folgende Programm hat eine ähnliche Funktion wie das Unix-Kommando `uniq`. Es speichert jedes Wort nur ein mal, egal wie häufig es vorkommt. Das Beispielprogramm geht davon aus, dass in einer Datei jeweils nur ein Wort in jeder Zeile steht.

Die ganze Magie verbirgt sich in Zeile 05. Sie speichert das gelesene Wort als Schüssel, wobei der Wert beliebig sein darf. Ist der Schlüssel neu, bekommt der Hash ein weiteres Element, ist er dagegen schon vorhanden, wird der Hash nicht erweitert. Die Ausgabe (Zeilen 08 bis 11) kennen wir nun schon aus anderen Programmen. Diesmal wurde nur eine Sortierung nachgeschaltet.

```
01 declare -A STAT
02
03 # Worte einlesen
04 while read WORT; do
05   STAT[$WORT]=1
06 done < text
07
08 # Kontrollausgabe
09 for I in ${!STAT[@]}; do
10   echo "$I"
11 done | sort
```

Es gehört zwar nicht direkt zu Thema, aber vielleicht fragt sich manch einer, wie man eine normale Textdatei so in Worte aufsplittet, dass sich in jeder Zeile genau ein Wort befindet. Hier hilft das Tool `tr`. Das folgende Skript (ein Einzeiler, der zur Erläuterung mehrzeilig geschrieben wird) erstellt aus einer beliebigen Textdatei eine Wortliste. In Zeile 01 ersetzt `tr` mehrere aufeinander folgende Leerzeichen durch ein einziges. In Zeile 02 entfernt es alle Satzzeichen und Zahlen (alles bis auf die Buchstaben). In Zeile 03 wandelt es die Leerzeichen zu Newline-Zeichen. Schließlich wird noch sortiert, und dann entfernt `uniq` die Doubletten:

```
01: cat $DATEI | tr -s ' ' ' ' | \
02: tr -d ' '-'@' | \
03: tr ' ' '\n' | \
04: sort | uniq
```

Eine weitere typische Anwendung ist das Erstellen einer Häufigkeitsstatistik. Bei jedem Auftreten eines Schlüssels wird diesmal der Wert inkrementiert, und schon hat man die Häufigkeiten der Schlüssel ermittelt. Das soll eine fiktive Notenstatistik demonstrieren. Fiktiv ist die Statistik deshalb, weil die Datendatei per Skript und der `RANDOM`-Variablen erzeugt entstand. Deshalb gibt es auch keine Normalverteilung.

Zuerst werden die Noten wieder per Schleife in den Zeilen 08 bis 11 eingelesen. Im Gegensatz zum Beispiel oben wird jedoch der Wert inkrementiert (Zeile 09). Deshalb listet die Kontrollausgabe (Zeilen 13 bis 19) diesmal auch Schlüssel und Wert auf. Womit der gewünschte Zweck eigentlich schon erfüllt wäre.

Weil aber bei der Ausgabe von Häufigkeiten gerne ein Histogramm erzeugt wird, ist das Skript entsprechend gepimpt. Zuerst ermittelt es in den Zeilen 21 bis 27 das Maximum der Werte. Dann kann nämlich die Länge der Histogramm-Balken normiert werden. Die Variable `HMAX` legt die maximale Länge fest, sie ist in Zeile 04 definiert. Zusammen mit dem ermittelten Maximum `MAX` kann dann die Länge des Balkens für einen bestimmten Wert nach der Formel

$$LEN = \frac{Wert * HMAX}{MAX}$$

ermittelt werden. Das wird in der Schleife über alle Schlüssel erledigt (Zeilen 29 bis 38). Um das Histogramm zu zeichnen, gibt die innere Schleife `LEN` mal ein Doppelkreuz aus (Zeilen 34 bis 36). Ganz zum Schluss wird das Histogramm nach Noten sortiert:

```
01 declare -A STAT
02
03  # Max. Länge eines Histogrammbalkens
04  HMAX=40
05
06 # Noten einlesen
07 COUNT=0
08 while read NOTE; do
09   STAT[$NOTE]=$(( ${STAT[$NOTE]} + 1 ))
10   COUNT=$(( $COUNT + 1 ))
11 done < noten
12
13 # Ausgabe: Note, Anzahl (unsortiert)
14 for I in ${!STAT[@]}; do
15   echo "$I ${STAT[$I]}"
16 done
17
18 # Gesamtanzahl ausgeben
19 echo $COUNT
20
21 # Maximalwert der Anzahlen bestimmen
22 MAX=0
23 for I in ${!STAT[@]}; do
24   if [ ${STAT[$I]} -gt $MAX ]; then
25     MAX=${STAT[$I]}
26   fi
27 done
28
29 # Histogramm ausgeben (sortiert)
30 echo ""
31 for I in ${!STAT[@]}; do
32   echo -n "$I (${STAT[$I]}): "
33   LEN=$(( ${STAT[$I]} * $HMAX / $MAX ))
34   for K in $(seq 1 $LEN); do
35     echo -n "#"
36   done
37   echo ""
38 done | sort
```

Zum Testen generieren wir 1000 Notenwerte, die das Skript klaglos schluckt. Die Ausgabe zeigt, wie schon erwähnt, nicht die zu erwartende Normalverteilung, sondern eine relativ ordentliche Gleichverteilung:

```
3,7 81
5,0 82
3,3 76
3,0 86
2,7 87
2,3 76
2,0 81
4,0 70
4,3 65
4,7 61
1,7 71
1,0 81
1,3 83
1000

1,0 (81): ###################################
1,3 (83): ####################################
1,7 (71): ##############################
2,0 (81): ###################################
2,3 (76): ################################
2,7 (87): #########################################
3,0 (86): ########################################
3,3 (76): ################################
3,7 (81): ####################################
4,0 (70): ##############################
4,3 (65): ############################
4,7 (61): ############################
5,0 (82): ####################################
```

Eine Bemerkung zum Schluss: Wer anfängt, mit der Bash zu spielen, und verzweifelt, weil das Skript nicht läuft, denkt an den Aufruf `bash -vx skriptname`. Hier listet die Bash auf, wie sie bei jeder Zeile die Parameter ersetzt und was bei der Ausführung dieser Zeile geschieht. Wenn das immer noch nicht hilft, macht man es wie früher die BASIC-Programmierer: Zeile für Zeile eingeben und ständig das Programm laufen lassen (*programming by try and error*).

Über Jürgen

Jürgen Plate ist Professor für Elektro- und Informationstechnik an der *Hochschule für angewandte Wissenschaften München*. Er beschäftigt sich seit 1980 mit Datenfernübertragung und war, bevor der Internetanschluss für Privatpersonen möglich wurde, in der Mailboxszene aktiv. Unter anderem hat er eine der ersten öffentlichen Mailboxen – TEDAS der *mc*-Redaktion – programmiert und 1984 in Betrieb genommen.

Support vermarkten
Das Thomas-Krenn-Wiki als Beispiel für Content Marketing

Das in einem Unternehmen vorhandene Wissen gehört zu seinem Kapital: Jeder nickt. Dieses Wissen auch zu Marketingzwecken zu nutzen, schlägt zwei Fliegen mit einer Klappe: Einige stutzen. Dieses Vorgehen kostet Zeit und Geld: Alle gehen. Außer die Thomas Krenn AG, die weiß, wie es geht.

von **Anika Kehrer**

Es ist Zeitverschwendung,
etwas Mittelmäßiges zu machen.

Madonna

Der Begriff *Content-Marketing* ist erst ein paar Jahre in Verwendung, die Methode jedoch viel älter. Kundenmagazine sind ein klassisches Beispiel, Unternehmensblogs ein eher neueres. Unter Content Marketing sind nämlich Inhalte (Content) zu verstehen, die bei potenziellen Kunden die Bekanntheit und wahrgenommene Attraktivität eines Produktes fördern (Marketing).

Das kann auf unterschiedlichste Weise geschehen. Prinzipiell kann der in einem gedruckten Pressepublikation veröffentlichte Fachartikel eines Technikmitarbeiters dieses Ziel genauso erreichen, wie das Glossar eines Marketingmitarbeiters auf der unternehmenseigenen Webseite: Derjenige, der interessanten Content erstellt, kann einen Marketingeffekt für sich oder sein Produkt erzielen.

Rufen wir uns nun IT-lastige Firmen vor Augen. Solche, bei denen wir überwiegend Consultants, Ingenieure, Software-Entwickler und vielleicht noch Produktmanager haben. Wir haben Admins und Architekten. Wir haben jede erdenkliche Tonart von Techsprech und jeden erdenklichen Grad von Systematik, und wir haben eine Welt von Codezeilen, Konfigurationsinterfaces, Kommunikationsprotokollen und Hardware-Specs. Man sollte meinen – und die Autorin ist davon überzeugt – dass in solche einer Welt zwangsläufig ein üppiger Garten an Ereignissen und Sachverhalten blüht, die für thematisch Interessierte von Belang sind, sowohl in der Sache, als auch als Genuss. Potentielle Inhalte im Sinne des Content-Marketing also.

Solche Inhalte nicht nur zu haben, sondern auch aufzubereiten, stellen sich viele allerdings eher leicht vor. Vielleicht steht dahinter der Mechanismus, Gewerke zu unterschätzen, die man nicht oder nicht gut kennt. Während Kundenmagazine von Marketingabteilungen mit einigem Aufwand erstellt werden, sieht die Vorstellung von Content-Marketing-interessierten Verantwortlichen in IT-Firmen eher so aus: Ein paar der Techies schreiben halt was nebenbei. Höchstens muss man das dann noch sprachlich ein wenig geradeziehen.

Diese Auffassung unterschätzt jedoch, dass jeder Autor eine Menge Zeit benötigt, ganz zu schweigen von bestimmten schreiberischen Fähigkeiten. Wie man Qualitätssicherung eines Textes vornimmt, welche Wirkung man mit welcher Schreibmethode erzielt, oder was ein Leser möglicherweise überhaupt interessant findet, gehört nicht zum IT-Handwerk, und auch nicht zwingend zur jedermanns Talentrepertoire.

IT-Dokumentation trifft Schreibinteresse

Man kann es also als erstaunlich bezeichnen, dass Firmen manchmal trotzdem der Meinung sind, jenes andere Handwerk nebenbei mitausführen zu können. Umso erbaulicher ist zu sehen, dass manche es schaffen. Ein Beispiel ist der Serverhersteller und -dienstleister Thomas Krenn AG (Abbildung 1) mit seinem Thomas-Krenn-Wiki [1].

Es besteht aus konkreten Anleitungen, wie man sie herkömmlich in einer Support-Datenbank verpacken würde, und ergänzt sie mit Wissensstücken über Technik und Technologien, die man sie ansonsten in Fachartikeln findet. Mit dem Nutzwert für den Leser erzielt das Thomas-Krenn-Wiki Erfolg. Bei genauerem Hinsehen verläuft das aber weder einfach mal so, noch nebenbei. Und geplant war es schon gar nicht. Aber es wurde konsequent begleitet.

Abbildung 1: Die Thomas Krenn AG im bayerischen Freyung, 2002 gegründet, 160 Mitarbeiter. (Foto: Thomas Krenn AG)

Bei Thomas Krenn hat der diplomierte Computersicherheitsfachmann Werner Fischer im Jahr 2005 nach seinem Studium als IT-Technologe angefangen. Gemeinsam mit einem Kollegen hat er zum Beispiel Abläufe automatisiert und monitort. Die Tools, die dabei zum Einsatz kamen, wurden von diversen Linux-Distributionen unterstützt. So entstand beispielsweise der Bedarf zu notieren, welcher Port wo zum Einsatz kommt – ein Fall klassischer IT-Dokumentation. Das war der Ursprung des Thomas-Krenn-Wikis, wie ihn Fischer bei einem Vortrag auf dem FFG 2017 berichtet hat [2]. Die Wikistartseite datiert laut Seiteninformationen auf den 22. Februar 2008 [3]. Ein gutes Jahr später begann die Firma, neue Wikiartikel unter einem eigenen Account zu twittern [4]. Wieder ein Jahr später, im Jahr 2010, gab es sogar eine Pressemitteilung zum Thomas-Krenn-Wiki [5].

Inzwischen arbeitet Fischer im Bereich *Communications/Knowledge Transfer* des Unternehmens. Einen Hang zum Schreiben bringt er offensichtlich mit, da er seit 2006 Fachartikel in verschiedenen Medien publiziert. Interessant ist dennoch, dass es zum Beispiel zum Thomas-Krenn-Kundenmagazin, das zwischen 2012 und 2014 erschienen ist [6], offenbar keinerlei separate Ankündigung oder Hintergrundinformation gibt. Außerdem existiert auf dem Thomas-Krenn-Webauftritt die Rubrik *tkmag* [7]. Dahinter verbirgt sich ein Online-Fachmagazin, ebenfalls ohne weitere Selbstbeschreibung oder eigene Pressemitteilung. Es wartet mit eigenen Artikeln auf, wozu auch Darreichungen wie eine kleine SSH-Referenz gehören. Die Wikiartikel blendet es ein und verlinkt auf sie.

Ohne Zugangshürden

Allerdings muss sich der Leser bei einigen Online-Artikeln des *tkmag* registrieren. Beim Wiki ist das nicht der Fall. Wäre auch komisch für ein Wiki. Aber wie Fischer im Vortrag erklärte, ist das gewollt und mit Absicht so: Die Wikiinformationen sollen leicht zugänglich sein, Service-Orientierung habe Vorrang. Es stellt sich natürlich die Frage, warum das bei den *tkmag*-Artikeln anders sein sollte. Aber viele Unternehmen stellen manche Inhalte hinter eine Registration Wall, zumeist Whitepaper und Webinare. Ist schließlich ihr gutes Recht. Sie werden ihre Gründe haben, auch wenn nicht jeder stets den Sinn erkennen kann.

Zudem hat man mit dem *tkmag* augenscheinlich etwas anderes im Sinn als mit dem Wiki. Das erwähnte SSH-Sheet (Abbildung 2) ist zum Beispiel nicht im Wiki zu finden. Das Wiki besitzt im Gegensatz zum *tkmag* einen eigenen Twitterfeed, der aber im Wiki selbst gar nicht beworben wird. Auch ist das Wiki mehrsprachig, das *tkmag* nicht. Denkbar und nachvollziehbar ist, dass das Unternehmen an der Abgrenzung tüftelt, welches Medium wie welchen Zweck am besten erfüllt. Content-Marketing betreibt es jedenfalls bereits auf recht differenziertem Niveau.

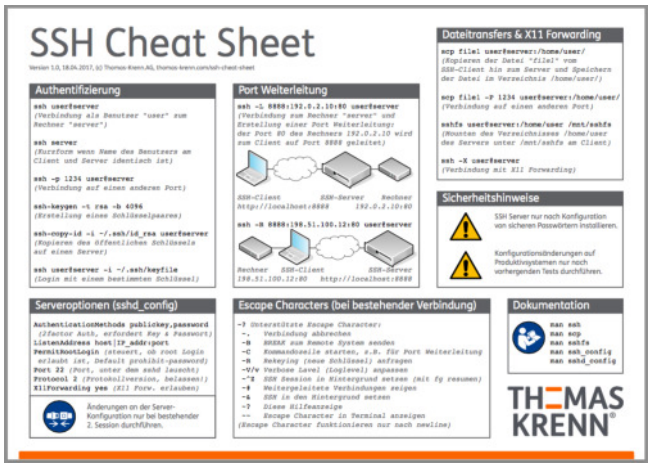


Abbildung 2: *SSH Cheat Sheet* von Thomas Krenn – nicht im Wiki, aber im *tkmag* [8]. (© Thomas Krenn AG, CC-BY-SA)

Das Thomas-Krenn-Wiki erfuhr jedenfalls im Unterschied zum *tkmag* viel öffentliche Aufmerksamkeit – nicht nur vom Unternehmen selbst, sondern auch von Lesern. Die Zugriffszahlen hätten zum Beispiel schnell die des Shops überstiegen, berichtete Fischer in seinem Vortrag. Die Zahl der Inhaltsseiten – der Content also – liegt derzeit bei etwa 1.700. Im Monat Februar 2017 verzeichnete dieser Content laut Fischers Angaben rund 500.000 Seitenaufrufe von etwa 250.000 Usern. Sämtlichen dieser User hat Thomas Krenn einen Dienst erwiesen und sein Angebot damit positiv in Erscheinung gebracht – sprich: Marketing gemacht.

Allerdings: Ob der Konsument eines Inhalts wirklich das Gefühl hat, dass ihm ein Dienst erwiesen wird, hängt davon ab, ob ihm das Ganze auch etwas bringt. Beim Thomas-Krenn-Wiki hat man sich Nutzwert auf die Fahnen geschrieben (*Red Bull* oder *Coca Cola* zum Beispiel setzen stattdessen auf Unterhaltung). Absicht und Ergebnis sind nun nicht immer deckungsgleich. Doch Thomas Krenn schafft es. Was bei Fischers Vortrag interessant war, ist das Ausmaß der Qualitätssicherung, die das Unternehmen dafür in die Wikiartikel zu investieren bereit ist.

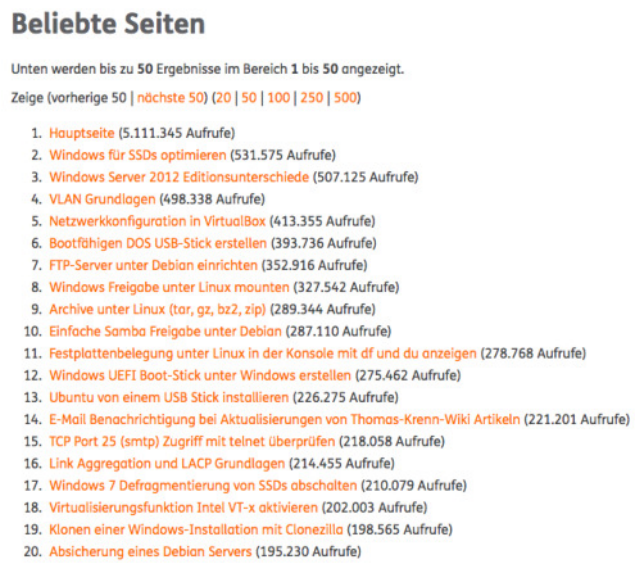
Mit Qualitätskontrolle

Die Inhalte des Wikis (Abbildung 3), wie sie ursprünglich angedacht waren, sind denkbar geradeheraus: „Information zu Installationen und zu bekannten Problemen und Lösungsansätzen“. So liest es sich auf der Wikistartseite in den ersten Versionen von 2008 (wenn man in die Versionsgeschichte der Seite lugt). Bis 2017 hat sich der Inhalt zu folgendem Umfang gemausert, wie es auf der heutigen Version der Startseite zu lesen ist: „Hier finden Sie das gesammelte Know-how der Mitarbeiter der Thomas-Krenn.AG. Die Artikel reichen von Installations- und Konfigurationsanleitungen über technischen Erklärungen bis hin zu konkreten Problemlösungen.“

Eine reife Leistung. Und ein Satz, wie er in einem Content-Marketing-Konzept stehen könnte. Tut er aber nicht – und tat er nie. Das Ergebnis fußt stattdessen auf freier Initiative und überlegter Begleitung.

Bei seinem Vortrag friemelte Werner Fischer zum Beispiel ausführlich auseinander, wie er mögliche Themen und Quellen bewertet – etwa nach Aktualität, Zeitersparnis und Technik. Er berichtete, dass irgendwo gefundene Lösungen für Probleme stets erst nachgestellt werden, bevor sie – geprüft und verifiziert – ins Wiki wandern. Zuletzt hat das Wikiteam eine Art Redaktionsplan für Aktualisierungsbedarf und Revision eingeführt: Anhand einer Wiki-Kategorie, etwa *Review QI 2017*, sammeln sie Softwareartikel, von denen sie zum Beispiel wissen, dass eine neue Softwareversion ansteht.

Der Referent gab zu, dass sie zwar nicht unbedingt hinterher kommen. Aber solche zeitlichen Ressourcen kann man erst planen, wenn Transparenz besteht, was eigentlich zu tun ist. Gerade bei Wikis droht bekanntlich ziemlich schnell eine ziemlich tötliche Wucherung der Inhalte, teils durch Veralterung bedingt. Alte Softwareversionen erhalten vom Wikiteam bei Thomas Krenn zur Eindämmung einen Hinweis, dass die Info nicht mehr gepflegt wird.



Beliebte Seiten

Unten werden bis zu **50** Ergebnisse im Bereich **1** bis **50** angezeigt.

Zeige (vorherige 50 | nächste 50) (20 | 50 | 100 | 250 | 500)

1. Hauptseite (5.111.345 Aufrufe)
2. Windows für SSDs optimieren (531.575 Aufrufe)
3. Windows Server 2012 Editionsunterschiede (507.125 Aufrufe)
4. VLAN Grundlagen (498.338 Aufrufe)
5. Netzwerkkonfiguration in VirtualBox (413.355 Aufrufe)
6. Bootfähigen DOS USB-Stick erstellen (393.736 Aufrufe)
7. FTP-Server unter Debian einrichten (352.916 Aufrufe)
8. Windows Freigabe unter Linux mounten (327.542 Aufrufe)
9. Archive unter Linux (tar, gz, bz2, zip) (289.344 Aufrufe)
10. Einfache Samba Freigabe unter Debian (287.110 Aufrufe)
11. Festplattenbelegung unter Linux in der Konsole mit df und du anzeigen (278.768 Aufrufe)
12. Windows UEFI Boot-Stick unter Windows erstellen (275.462 Aufrufe)
13. Ubuntu von einem USB Stick installieren (226.275 Aufrufe)
14. E-Mail Benachrichtigung bei Aktualisierungen von Thomas-Krenn-Wiki Artikeln (221.201 Aufrufe)
15. TCP Port 25 (smtp) Zugriff mit telnet überprüfen (218.058 Aufrufe)
16. Link Aggregation und LACP Grundlagen (214.455 Aufrufe)
17. Windows 7 Defragmentierung von SSDs abschalten (210.079 Aufrufe)
18. Virtualisierungsfunktion Intel VT-x aktivieren (202.003 Aufrufe)
19. Klonen einer Windows-Installation mit Clonezilla (198.565 Aufrufe)
20. Absicherung eines Debian Servers (195.230 Aufrufe)

Abbildung 3: Die 20 beliebtesten Artikel im Thomas-Krenn-Wiki.

A propos Wucherung: An welchem Ort in der Themenstruktur ein Artikel landet, entpuppt sich im Thomas-Krenn-Wiki als Bestandteil qualitätsorientierter Content-Produktion. Fischer zählte nämlich bewusstes Strukturieren und Einordnen des Inhalts zu den Punkten, die zu nachhaltigem Dokumentieren führen – mithin zu einer sauberen Ablage. Praktisch hat das zum einen den Effekt, einen klaren und vor allem beständigen URL zu gewährleisten („Cool URIs don’t change“, zitierte der Referent Tim Berners-Lee). Nicht nur vom Leser, auch von Google würden gleichbleibende Webadressen honoriert, erklärte Fischer – und die Auffindbarkeit bei Google ist bei einer frei zugänglichen Wissensressource nicht nur marketingrelevant, sondern vor allem in der Sache das A und O. Zum anderen zwingt das aber den Schreibenden, sich Gedanken zu machen, worüber er eigentlich schreibt und wie es einzuordnen ist.

Schreiben für den Leser

Fischers Tipps für nachhaltiges Dokumentieren lauten in vielen Teilen genauso, wie sie auch generell für gutes Schreiben lauten würden. Das ist nämlich eine Art zu schreiben, die (mindestens) dem Leser erleichtert, das Geschriebene zu lesen und zu verstehen. Es ist also sozusagen wurscht, ob du für das Thomas-Krenn-Wiki schreibst, oder etwa eine Produkterklärung im Social-Media-Kanal deines Unternehmens. Laut Werner Fischer

führen zum Beispiel folgende Tipps zu nützlicher Wissensdarbietung:

- Struktur: Klarer Gegenstand, klares Ziel, klarer Kontext.
- Inhalt: Problemlösung, aber auch Grundlagenexkurse, die vielleicht anderswo wieder wichtig werden.
- Aufbau: Einleitung, knappe und informative Überschriften, vom Allgemeinen zum Speziellen.
- Stil: Kurze Sätze, Beispiele, Belege.
- Rechtschreibkorrektur.
- Mehraugenprinzip.

Als Extratipp legte der Referent den Zuhörern Bilder mit Pfeilen und Erklärungen ans Herz. Er wusste zu berichten, dass eine Grafik schon mal zu tagelanger Arbeit ausartet (Abbildung 4) – und schmunzelte dabei. Tatsächlich ist das aber, verpackt in eine Randbemerkung, ein wesentlicher und oft unterschätzter Mechanismus in der Produktion guter Inhalte: Es dauert nicht nur, das Ganze zu verstehen. Sondern es dauert auch ganz erstaunlich lange, das Ganze sinnvoll aufzubereiten.

Einen interessanten Effekt beschrieb der Referent hinsichtlich der Sorgsamkeit bei der Kategorisierung der Wikiartikel. Es hat, berichtete er, einen großen Unterschied bedeutet, ob die Wikiautoren für ein internes oder ein öffentliches Wiki schrieben. Eine Zeit lang habe Thomas Krenn nämlich auch ein internes Wiki betrieben. Hierbei sind die Kategorien explodiert – es ist also das passiert, was man als Wucherung kennt. Die Autoren sind offenbar weniger sorgsam mit der Einordnung ihrer Beiträge umgegangen. Aber bei dem externen Wiki reißen sich die Leute offenbar mehr zusammen. Als entscheidenden Unterschied vermutete der Referent außerdem, dass beim externen Wiki stets Peer Review stattfand, beim internen nicht. Es lag hingegen nicht daran, wie der Referent auf Nachfrage erklärte, dass bei dem internen Wiki mehr Autoren zu Gange waren – nach dem Motto, viele Köche verderben den Brei. Die Zahl der Autoren war vielmehr in etwa gleich.

Ja, und wer schreibt den Kram?

Die Autoren. Das brisanteste Stichwort beim Thema Content-Produktion fällt zum Schluss. Pläne, Regeln und ausgeklügelte QA-Prozesse nützen nichts, wenn niemand da ist, der Texte schreibt (oder Grafiken erstellt). Die entscheidende Frage ist hier, wen im Unternehmen man zum Autoren abstellt, der bitteschön regelmäßig liefern soll. Ist es der Consultant? Er sitzt schließlich an der Quelle immer neuer und interessanter Problemstellungen. Ist es ein Opsler? Er kennt die Konfigurationen und hilfreiche Tunes. Ist es die Produktmanagerin? Die weiß, was der Kunde wirklich will. Oder das Marketing? Hier würde man das traditionell am ehesten ansiedeln.

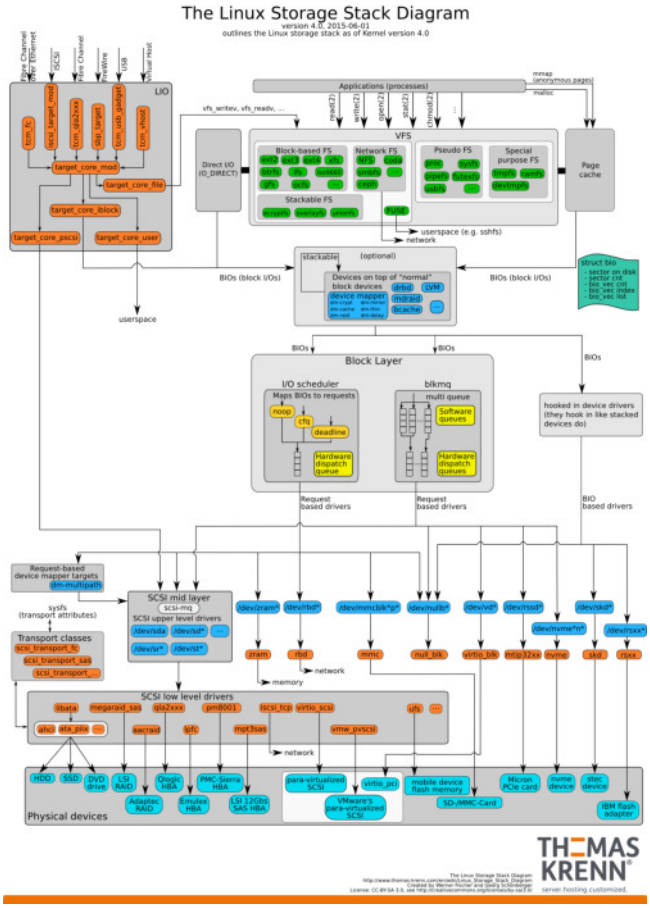


Abbildung 4: In dieser Grafik stecken mehrere Personentage. (© Thomas Krenn AG, CC-BY-SA)

Im Windschatten der Aufmerksamkeit gibt es da aber noch jemanden: den Support. Und bei Thomas Krenn sind die Autoren – neben schreibinteressierten Technikern – tatsächlich der Support. Interessanterweise sind die Supportkollegen auch diejenigen, die von der reinen IT-Dokumentation als Quelle enorm unterschätzt werden. Das lernt man zumindest, wenn man Konferenzen besucht, die sich mit Technischer Dokumentation befassen. Ein Vortrag auf einer solchen Konferenz gab mal Folgendes zu bedenken, sinngemäß wiedergegeben, zugespitzt auf einen verzeifelten Ausruf: ‚Der Support hört die Probleme und Fragen der Kunden jeden einzelnen Tag. Nutzt das doch! Im Büro sitzt der Support oft am anderen Ende des Flurs von Entwicklung und Dokumentation. Warum nur, ach, warum? ‘

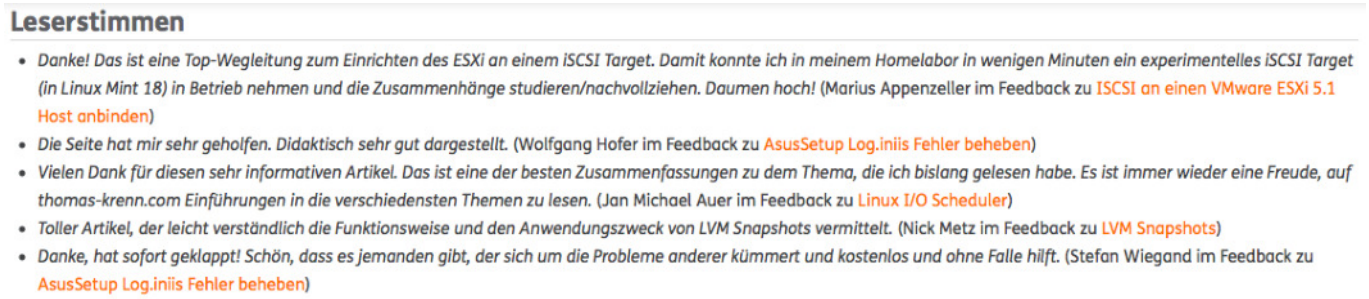


Abbildung 5: Glückliche Leser sind potenzielle Kunden.

Bei Thomas Krenn funktioniert es offenbar erstaunlich gut, dass die Supportleute Artikel schreiben. Es ist aber auch so, dass sie einen Bonus bekommen, wenn ihr Artikel viel angeklickt wird, wie der Referent dankenswerterweise verriet.

Als Verantwortlicher rümpft man nun vielleicht die Nase und wirft die Hände hoch, warum denn immer alles vom Geld abhängen muss. Aber wovon denn sonst? Wir gehen schließlich arbeiten, um Geld zu verdienen – und zwar in der Arbeitszeit, nicht in unbezahlten Überstunden. Die Zeit, die das Schreiben kostet, halst sich niemand auf, der nicht muss (und dann wird es verständlicherweise nicht besonders gut) oder keinen besonderen Anreiz dafür hat. Allerdings liegt auch ein eigentlich ziemlich naheliegender Trick darin, diejenigen zum Schreiben zu ermuntern, die sowieso schon den ganzen Tag Dinge erklären. Wenn dann noch Einzelne aus anderen Abteilungen hinzukommen, die einfach gern schreiben, sollte die Autorenzahl brauchbar werden.

Vielleicht spielt auch Folgendes eine Rolle: Das Ganze läuft nicht unter Marketing. Wenn Werner Fischer Vorträge über das Thomas-Krenn-Wiki hält, hört man nichts von „Vorteilen“, dass alles „einfach“ sei oder das irgendjemand von irgendetwas „profitiert“ – blumige Begriffe, die mit Duft locken wollen, aber dann eben leider auch oft luftig bleiben, weil sie den behaupteten Mehrwert zwar behaupten, aber nicht belegen. Stattdessen ist bei Fischer von „Dokumentation“, „Relevanz“ und „Support“ die Rede.

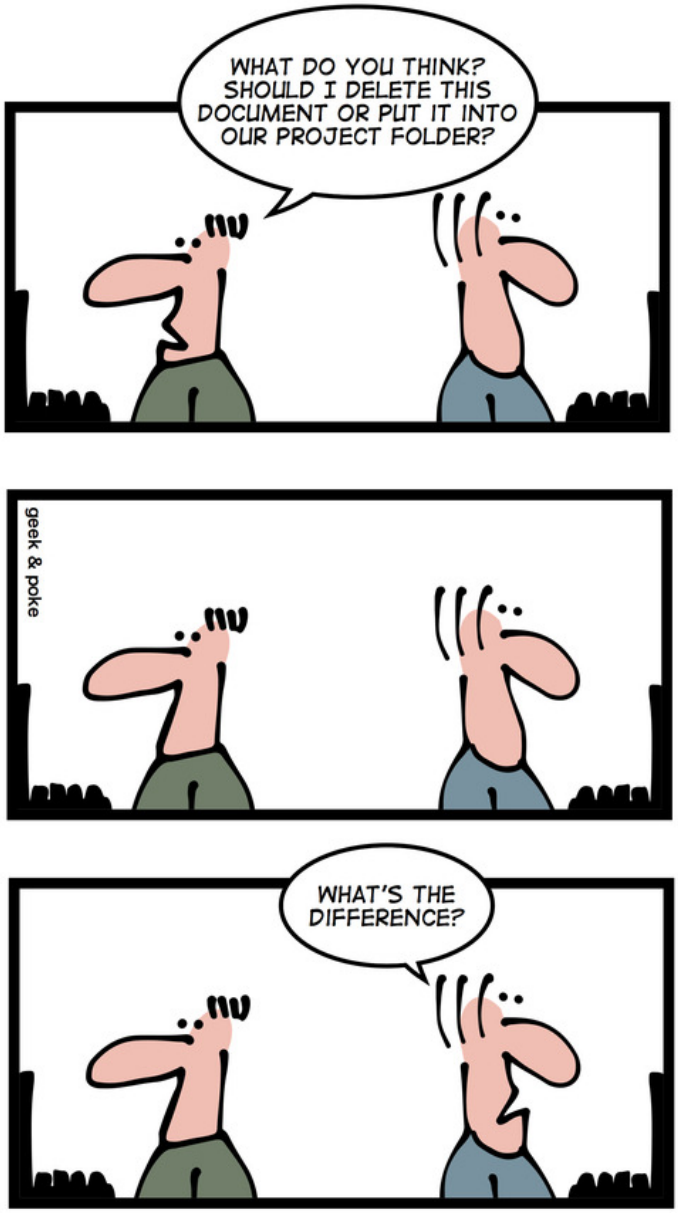
Wenn man Marketing also tatsächlich am Inhalt aufhängt, ist es auch kein Wunder, dass man ihn stolz öffentlich bewerben kann, ohne dass es komisch wirkt. Wissen und Problemlösung als Service gehört in die Kategorie *Tue Gutes und rede darüber*. Dass jeder Leser ein potenzieller Kunde ist, ist die Triebfeder des Ganzen, wirkt aber nicht so bemüht wie so manche Produktbroschüre, und erzeugt auch kein solches Hintenrumgefühl wie *Advertorials* oder *Native Ads*.

Die Früchte erntet das Unternehmen Thomas Krenn sichtbar in Form von klassischen Leserstimmen (Abbildung 5). Seit 2015 sammelt die *Über*-Seite des Wikis Feedback [9]. Jeder Autor würde sich baden in Wertungen wie dieser: „Das ist eine der besten Zusammenfassungen zu dem Thema, die ich bislang gelesen habe.“ Und war da nicht auch was mit Marketing, also zum Beispiel Markenbildung und Image-Pflege? Dafür ist es wichtig, dass der Name fällt. Und, ah, sieh mal an: „Es ist immer wieder eine Freude, auf thomas-krenn.com Einführungen in die verschiedensten Themen zu lesen“, setzt der Kommentator noch einen drauf. Da wird der Unternehmensname in ein denkbar positives Licht gerückt. Mission accomplished.

Offenlegung von Interessenverflechtungen: Die Autorin kennt das Unternehmen, das Gegenstand dieses Artikels ist, sowie den im Artikel erwähnten Vortragsreferenten in ihrer Eigenschaft als LinuxTag-Mitglied. Sie hat ihm im Jahr 2013 persönlich die Ehrung ‚LinuxTag Hero‘ überreicht, die sie ihm für Thomas Krenns wiederholte Open-Source-Förderung auf dem LinuxTag verliehen hat. Vom Thomas-Krenn-Wiki waren diese Begegnungen unabhängig: Auf das Thema des Wikis stieß die Autorin durch Werner Fischers Vortrag auf dem FFG 2017.

Links

[1] Startseite des Thomas-Krenn-Wikis: https://www.thomas-krenn.com/de/wiki/Hauptseite
[2] Werner Fischers Vortrag auf dem FFG 2017: https://guug.de/veranstaltungen/ffg2017/abstracts.html#4_4_1
[3] 2008-er Version der Wikistartseite:
https://www.thomas-krenn.com/de/wikiDE/index.php?title=Hauptseite&dir=prev&action=history
[4] Twitterfeed des Thomas-Krenn-Wikis: https://twitter.com/tkwiki
[5] 2010-er Pressemeldung zum Thomas-Krenn-Wiki:
https://www.thomas-krenn.com/de/unternehmen/presse/pressemeldung.werner-fischer-technology-specialist-praesentiert-das-thomas-krenn-wiki.html
[6] Thomas-Krenn-Kundenmagazin: https://www.thomas-krenn.com/de/tkmag/tk-insights/kundemagazine/
[7] *tkmag*: https://www.thomas-krenn.com/de/tkmag/
[8] *SSH Cheat Sheet* von Thomas Krenn:
https://www.thomas-krenn.com/de/tkmag/allgemein/ssh-tricks-und-tipps-im-ssh-cheat-sheet/
[9] About-Seite des Thomas-Krenn-Wikis mit Leserstimmen:
https://www.thomas-krenn.com/de/wiki/Thomas-Krenn-Wiki:Über



Hilfreiches für alle Beteiligten
Autorenrichtlinien

Selbst etwas für die UpTimes schreiben? Aber ja! Als Thema ist willkommen, was ein GUUG-Mitglied interessiert und im Themenbereich der GUUG liegt. Was sonst noch zu beachten ist, steht in diesen Autorenrichtlinien.

Der Schriftsteller ragt zu den Sternen empor,
Mit ausgefranstem T-Shirt.
Er raunt seiner Zeit ihre Wonnen ins Ohr,
Mit ausgefranstem T-Shirt.

Frei nach Frank Wedekind, Die Schriftstellerhymne

Wir sind an Beiträgen interessiert. Wir, das ist diejenige Gruppe innerhalb der GUUG, die dafür sorgt, dass die UpTimes entsteht. Dieser Prozess steht jedem GUUG-Mitglied offen. Der Ort dafür ist die Mailingliste <redaktion@uptimes.de>.

Welche Themen und Beitragsarten kann ich einsenden?

Die UpTimes richtet sich als Vereinszeitschrift der GUUG an Leser, die sich meistens beruflich mit Computernetzwerken, IT-Sicherheit, Unix-Systemadministration und artverwandten Themen auseinandersetzen. Technologische Diskussionen, Methodenbeschreibungen und Einführungen in neue Themen sind für dieses Zielpublikum interessant, Basiswissen im Stil von *Einführung in die Bourne Shell* hingegen eher nicht. Wer sich nicht sicher ist, ob sein Thema für die UpTimes von Interesse ist, kann uns gern eine E-Mail an <redaktion@uptimes.de> schicken.

Neben Fachbeiträgen sind Berichte aus dem Vereinsleben, Buchrezensionen, Konferenzberichte, humoristische Formen und natürlich Leserbriefe interessant. Auch Notizen aus einem Vortrag oder Seminar, die Du besucht hast, eignen sich als Grundlage für einen eigenen kleinen Text darüber. Wer nicht gleich mehrseitige Artikel schreiben möchte, beginnt also einfach mit einem kleineren Text.

Auch Übersetzungen sind von Interesse. Oft stößt man zum Beispiel auf englischsprachige Beiträge anderswo, die man interessant findet. Solche Beiträge kannst Du wie einen eigenen Artikel als Übersetzung der UpTimes vorschlagen. Du stehst dann als Übersetzer über dem Artikel, der Originalautor erhält – wenn möglich – einen Autorenkasten. Bevor Du Übersetzungen einreichst, kontaktierst Du den Originalautoren und bittest ihn um Erlaubnis zur Übersetzung und Veröffentlichung in der UpTimes. Dieser Schritt ist wichtig, denn die UpTimes benötigt das Nutzungs- und Bearbeitungsrecht für die Veröffentlichung, und diese Rechte muss man sich qua Erlaubnis separat und explizit einholen. Für den Autorenkasten bittest Du den Originalautoren entweder um eine Kurzbeschreibung, oder recherchierst sie selbst. Beispiele, wo wir Beiträge Dritter für die UpTimes verarbeitet haben, sind die Artikel *OpsReport-Card – Fragen für Sysadmins* in Ausgabe 2014-2, *IT-Sicherheit als Realpolitik* in Ausgabe 2015-2 oder *Kernel-Regressions bekämpfen* in Ausgabe 2017-1.

Fachbeiträge sind sachbezogen, verwenden fachsprachliches Vokabular und klärende Erläuterungen, besitzen technische Tiefe und ggf. auch Exkurse. Berichte aus dem Vereinsleben greifen aktuelle Themen auf oder legen Gedankengänge rund um die GUUG und ihre Community dar. Konferenzberichte zeigen, welche Veranstaltungen jemand besucht hat, was er/sie dort erfahren hat und ob die Veranstaltung nach Meinung des Autoren beachtenswert oder verzichtbar war. Unterhaltsame Formen können ein Essay oder eine Glosse sein, aber auch Mischformen mit Fachartikeln (Beispiel: der *Winter-Krimi* in Ausgabe 2013-3). Auch unterhaltsame Formen besitzen jedoch inhaltlichen Anspruch. Denn die UpTimes ist und bleibt die Mitgliederzeitschrift eines Fachvereines.

In der UpTimes legen wir daher auch Wert auf formale publizistische Gepflogenheiten und einheitliche Schreibweisen. Dafür sorgt zum Beispiel ein einheitliches Layout der Artikel, oder etwa die grundsätzliche Vermeidung von Worten in Großbuchstaben (entspricht typografisches Schreien) oder von Worten in Anführungsstrichen zum Zeichen der Uneigentlichkeit (entspricht Distanzierung von den eigenen Worten). Wichtig sind außerdem beispielsweise Quellenangaben bei Zita-

ten, Kenntlichmachung fremder Gedanken, Nachvollziehbarkeit der Argumentation sowie Informationen zum Autor nach dem Artikel.

In welchem Format soll ich meinen Artikel einsenden?

ASCII: Am liebsten blanke UTF8-Texte. Gern mit beschreibenden Anmerkungen oder Hinweisen, die nicht zum eigentlichen Text gehören (kenntlich gemacht zum Beispiel mit Prozentzeichen).

LaTeX: Wir setzen die UpTimes mit LaTeX. Eine Vorlage mit den von uns verwendeten Auszeichnungen für Tabellen, Kästen und Abbildungen gibt es unter http://www.guug.de/uptimes/artikel-vorlage.tex. Mit dieser Vorlage kann der Autor selbst seinen Artikel mit den UpTimes-spezifischen Bestandteilen und Formatierungen konzipieren. Die Einsendung erfolgt dann mit sämtlichem Text in dieser .tex-Quelldatei (kein PDF), die dem weiteren Prozess zu Grunde liegt. Andernfalls überführen wir die Einsendung selbst in das UpTimes-Format. Weil wir – wie es sich beim Publizieren gehört – mehrspaltig setzen und ein homogenes Erscheinungsbild anstreben, verwenden wir für die UpTimes bestimmte, wiederkehrende Formatierungen. Einige sind obligatorisch (etwa der Artikelkopf), andere müssen nicht verwendet werden (zum Beispiel der Exkurskasten). Es gibt daher innerhalb des homogenen Erscheinungsbildes einen gewissen Freiheitsgrad für den Autor. Es ist nicht erwünscht, vom Formatkatalog abzuweichen oder eigene Layoutanweisungen einzusenden. Wir behalten uns vor, Texte für die Veröffentlichung in der UpTimes umzuformatieren.

Listings: Der mehrspaltige Druck erlaubt maximal 45 Zeichen Breite für Code-Beispiele, inklusive 1 Leerzeichen und einem Zeichen für den Zeilenumbruch innerhalb einer Code-Zeile (Backslash). Breitere Listings formatieren wir um, verkleinern die Schriftgröße oder setzen sie als separate Abbildung. Listingzeilen werden in der UpTimes numeriert und im Artikeltext mit Bezug auf diese Zeilennummern erläutert. Wir halten das für höflich und nützlich gegenüber dem Leser. Es hilft, die Listings so auszuwählen und zu präsentieren, dass sie dem Leser maximale Erkenntnis bringen.

Bilder: Wir verarbeiten gängige Bildformate, soweit ImageMagick sie verdaut und sie hochauflösend sind. Am besten eignen sich PNG- oder PDF-Bilddateien. Bei längeren Artikeln ist zur Vermeidung von Textwüsten ungefähr mit 1 Abbildung pro 3000 Zeichen zu planen. Das müssen nicht Bilder sein, sondern auch Tabellen, Listings oder ein Exkurskasten lockern den Text auf. Versieht Eure Bilder nicht mit Rahmen oder Verzierungen, weil die Redaktion diese im UpTimes-Stil selbst vornimmt.

Wie lang kann mein Artikel sein?

Ein einseitiger Artikel hat mit zwei Zwischentiteln um die 2.700 Anschläge. Mit etwa 15.000 Anschlägen – inklusive 3 Abbildungen – landet man auf rund vier Seiten. Wir nehmen gern auch achtseitige Artikel, achten dabei aber darauf, dass der Zusammenhang erhalten bleibt und dass es genug Bilder gibt, damit keine Textwüsten enstehen.

Wer Interesse hat, für die UpTimes zu schreiben, macht sich am besten um die Zeichenzahl nicht so viele Gedanken – auch für kurze oder lange Formate finden wir einen Platz. Die Redaktion ist bei der konkreten Ideenentwicklung gern behilflich. Für eine Artikelidee an <redaktion@uptimes.de> reicht es, wenn Ihr ein bestimmtes Thema behandeln wollt.

Wohin mit meinem Manuskript?

Am einfachsten per E-Mail an <redaktion@uptimes.de>. Das ist jederzeit möglich, spätestens jedoch vier Wochen vor dem Erscheinen der nächsten UpTimes. Zum Manuskript ist ein kleiner Infotext zum Autor wichtig, ein Bild wünschenswert.

Nützlich ist, wenn der Text vor Einsendung durch eine Rechtschreibkorrektur gelaufen ist. `aspell`, `ispell` oder `flyspell` für Textdateien sowie die von LibreOffice bieten sich an. Wenn Ihr Euren Text an die Redaktion schickt, solltet Ihr also weitestmöglich bereits auf die Rechtschreibung geachtet haben: Nach der Einschickung ist Rechtschreibung und Typo-Korrektur Aufgabe der Redaktion. Die Texte in der UpTimes folgen der neuen deutschen Rechtschreibung.

Wie verlaufen Redaktion und Satz?

Wir behalten uns vor, Texte für die Veröffentlichung in der UpTimes zu kürzen und zu redigieren. Das bedeutet, dafür zu sorgen, dass der Artikel nicht ausufert, versehentliche Leeraussagen wegfallen, Syntax und Satzanschlüsse geglättet werden, dass Passiva und Substantivierungen verringert und Unklarheiten beseitigt werden (die

zum Beispiel Fragen offen lassen oder aus Passivkonstruktionen resultieren, ohne dass der Schreibende das merkt). Manchmal ist dieser Prozess mit Nachfragen an den Autoren verbunden.

Die endgültige Textversion geht jedem Autoren am Ende zur Kontrolle zu. Dabei geht es um die inhaltliche Kontrolle, ob durch den Redaktionsprozess Missverständnisse oder Falschaussagen entstanden sind. Danach setzt die Redaktion die Artikel. Wenn der Satz weitgehend gediehen ist – also ein *Release Candidate* der UpTimes als PDF vorliegt – erhalten die Autoren als erste diesen RC. Danach wird die UpTimes dann veröffentlicht.

Gibt es Rechtliches zu beachten?

Die Inhalte der UpTimes stehen ab Veröffentlichung unter der CC-BY-SA-Lizenz, damit jeder Leser die Artikel und Bilder bei Nennung der Quelle weiterverbreiten und auch weiterverarbeiten darf. Bei allen eingereichten Manuskripten gehen wir davon aus, dass der Autor sie selbst geschrieben hat und der UpTimes ein nichtexklusives, aber zeitlich und räumlich unbegrenztes Nutzungs- und Bearbeitungsrecht unter der CC-BY-SA einräumt.

Bei Fotos oder Abbildungen Dritter ist es rechtlich unabdingbar, dass der Autor sich bei dem Urheber die Erlaubnis zu dieser Nutzung einholt, und fragt, wie die Quelle genannt zu werden wünscht. Die Frage nach der CC-BY-SA ist hierbei besonders wichtig.

An Exklusivrechten, wie sie bei kommerziellen Fachzeitschriften üblich sind, hat die UpTimes kein Interesse. Es ist den Autoren freigestellt, ihre Artikel noch anderweitig nach Belieben zu veröffentlichen.

Bekomme ich ein Autorenhonorar?

Für Fach- und literarische Beiträge zahlt die GUUG dem Autor nach Aufforderung durch die Redaktion und Rechnungstellung durch den Autor pro Seite 50 € zuzüglich eventuell anfallender USt. Bei Übersetzungen erhalten Übersetzer und Originalautor je die Hälfte. Beiträge für die Rubrik „Vereinsleben“, Buchrezensionen und Artikel bezahlter Redakteure sind davon ausgenommen. Gleiches gilt für Paper, wenn die UpTimes die Proceedings der Konferenz enthält.

Für die nächste redaktionelle Ausgabe wird ein Chefredakteur gesucht: UpTimes 2017-2, Winterausgabe

- Redaktionsschluss: N.N.
- Erscheinung: N.N.
- Gesuchte Inhalte: Fachbeiträge über Unix und verwandte Themen, Veranstaltungsberichte, Rezensionen, Beiträge aus dem Vereinsleben.
- Manuskript-Template für UpTimes-spezifische Formate: http://www.guug.de/uptimes/artikelvorlage.tex
- Fragen, Artikelideen und Manuskripte an: Redaktionsmailingliste <redaktion@uptimes.de>

Über die GUUG
German Unix User Group e.V.

Vereinigung deutscher Unix-Benutzer

Die Vereinigung Deutscher Unix-Benutzer hat gegenwärtig rund 700 Mitglieder, davon etwa 90 Firmen und Institutionen.

Im Mittelpunkt der Aktivitäten der GUUG stehen Konferenzen. Ein großes viertägiges Event der GUUG hat eine besondere Tradition und fachliche Bedeutung: In der ersten Jahreshälfte treffen sich diejenigen, die ihren beruflichen Schwerpunkt im Bereich der IT-Sicherheit, der System- oder Netzwerkadministration haben, beim *GUUG-Frühjahrsfachgespräch* (FFG).

Seit Oktober 2002 erscheint mit der *UpTimes* – die Sie gerade lesen – eine Vereinszeitung. Seit 2012 erscheint die UpTimes einerseits zu jedem FFG in Form einer gedruckten Proceedings-Ausgabe (ISBN), und andererseits im Rest des Jahres als digitale Redaktionsausgabe (ISSN). Daneben erhalten GUUG-Mitglieder zur Zeit die Zeitschrift *LANline* aus dem Konradin-Verlag kostenlos im Rahmen ihrer Mitgliedschaft.

Schließlich gibt es noch eine Reihe regionaler Treffen (http://www.guug.de/lokal): im Rhein-Ruhr- und im Rhein-Main-Gebiet sowie in Berlin, Hamburg, Karlsruhe und München.

Warum GUUG-Mitglied werden?

Die GUUG setzt sich für eine lebendige und professionelle Weiterentwicklung im Open Source-Bereich und für alle Belange der System-, Netzwerkadministration und IT-Sicherheit ein. Wir freuen uns besonders über diejenigen, die bereit sind, sich aktiv in der GUUG zu engagieren. Da die Mitgliedschaft mit jährlichen Kosten

Fördermitglied	350 €
persönliches Mitglied	70 €
in der Ausbildung	30 €

verbunden ist, stellt sich die Frage, welche Vorteile damit verbunden sind?

Neben der Unterstützung der erwähnten Ziele der GUUG profitieren Mitglieder auch finanziell davon, insbesondere durch die ermäßigten Gebühren bei den Konferenzen der GUUG und denen anderer europäischer UUGs. Mitglieder bekommen außerdem *c't* und *iX* zum reduzierten Abopreis.

Wie GUUG-Mitglied werden?

Füllen Sie einfach das Anmeldeformular unter https://www.guug.de/verein/mitglied/ aus und schicken Sie es per Fax oder Post an die unten auf dem Formular angegebene Adresse.

Impressum

Uptimes – Mitgliederzeitschrift der
German Unix User Group (GUUG) e.V.
Herausgeber: GUUG e.V.
Antonienallee 1
45279 Essen
E-Mail: <redaktion@uptimes.de>
Internet: http://www.guug.de/uptimes/

Autoren dieser Ausgabe: Ingo Wichmann und Nils Magnus, Anika Kehrer, Yves-Noel Weweler, Christopher J. Ruwe, Jürgen Plate
V.i.S.d.P: Ingo Wichmann, Vorstandsvorsitzender
Anschrift siehe Herausgeber.
Chefredaktion: Anika Kehrer
Redaktion: Mathias Weidner
LaTeX-Layout (PDF): Robin Därmann
XHTML-Layout (ePub): Mathias Weidner
Titelgestaltung: Hella Breitkopf
Bildnachweis: Titelbild *kabibo-Roboter folgt dem Licht* von Hella Breitkopf. Comicreihe *geek & poke* von Oliver Widder, CC-BY-SA, mit zusätzlicher freundlicher Genehmigung von Oliver Widder. Andere Quellennachweise am Bild, sofern nicht unter CC BY SA.
Verlag: Lehmanns Media GmbH, Hardenbergstraße 5, 10623 Berlin
ISSN: 2195-0016

Für Anzeigen in der UpTimes wenden Sie sich bitte an <werbung@guug.de>.