Hannes Mehnert

Wem müssen wir beim Benutzen von Software vertrauen?

Möglichkeiten zur radikalen Verkleinerung der *Trusted Computing Base*

Das Thema der FIfF-Konferenz 2017 war „TRUST – Wem kann ich trauen im Netz und warum". Ich beschäftige mich mit Vertrauenswürdigkeit von Software und war deshalb als Vortragender[1] eingeladen. Das Vertrauen in Software spielt eine wichtige Rolle, um von Vertrauen im Netz reden zu können. Seit mehreren Jahren forsche ich an einer Alternative zu existierenden Betriebssystemen, die ich in diesem Artikel vorstellen werde.

Heutzutage sind Computer aus der westlichen Welt nicht mehr wegzudenken. Viele schützenswerte persönliche Daten werden mit Hilfe von Computern elektronisch verarbeitet, seien es die Kreditkartendaten, die bei einer Online-Flugbuchung übermittelt werden, Ausweisdaten bei einer Grenzkontrolle, Zugangsdaten beim Online-Banking, oder persönliche Daten in sozialen Netzwerken wie Facebook, Instagram, Twitter oder GitHub. All den bei diesen Diensten verwendeten EDV-Systemen muss vertraut werden, die Datensicherheit der persönlichen Daten sicherzustellen. Dieses Vertrauen muss von der Eingabe an der Tastatur, die auch wieder eine Firmware hat, über die installierte Software (Webbrowser, Betriebssystem etc.), den Übertragungsweg (beispielsweise TLS als sicheres Protokoll, sowohl hinsichtlich der Spezifikation als auch der Implementierung), bis hin zum Server des Diensteanbieters (Software, Administrierende, physikalische Sicherheit im Rechenzentrum) reichen. Auf die Hardware (CPU, deren Mikrocode, und andere Peripherie wie Tastatur, Bildschirm, Netzwerkkarte) und die Umgebung (berechtigte Administrierende, Zugangskontrolle zum Rechenzentrum) gehe ich in diesem Artikel nicht ein. Betrachten wir lediglich die Software, sind dies heutzutage Millionen Zeilen Code, die beispielsweise bei einer Kreditkartenbuchung ausgeführt werden.

Wie können wir Software vertrauen?

Um Software Vertrauen entgegenzubringen, gibt es mehrere Möglichkeiten: Wenn den Entwickelnden vollständig vertraut wird, keine Fehler bei der Programmierung zu machen, heißt dies aber nicht, dass die Software fehlerfrei ist. Umso mehr Quellcode involviert ist, desto mehr Fehler schleichen sich ein. Die Menge an Quellcode ist also entscheidend: je weniger, desto einfacher ist es, diesen nachzuvollziehen und dessen Korrektheit zu überprüfen, und ihm somit zu vertrauen. Auch die verwendete Programmiersprache spielt eine große Rolle, da Programmiersprachen unterschiedliche Fehlerquellen zulassen: Wenn beispielsweise der Speicher von einer Laufzeitumgebung automatisch (durch einen Algorithmus) verwaltet wird, können Entwickelnde keine Fehler in der Speicherverwaltung machen. Natürlich muss dem implementierten Algorithmus vertraut werden, aber wenn dieser mathematisch korrekt bewiesen wurde, oder schon allein, wenn die Laufzeitumgebung auf vielen Computern verwendet wird, ist die Wahrscheinlichkeit gering, dass noch Fehler gefunden werden. Die Möglichkeit besteht immer, auch bei einem Beweis kann es sein, dass die Annahmen nicht korrekt sind (zum Beispiel Annahmen über das Zahlensystem; Computer verwenden meist Datentypen mit endlichen Wertebereichen, nicht unendliche Zahlenmengen wie in der Mathematik). Wenn der Quellcode von Software nicht vorliegt, kann nur durch Beobachtung des Laufzeitverhaltens oder durch *Reverse Engineering* Vertrauen hergestellt werden – hierauf möchte ich nicht näher eingehen.

Die *Trusted Computing Base (TCB)* eines Systems ist die Hardware und Software, die sicherheitsrelevant ist: ein Fehler in einem Bestandteil der TCB korrumpiert die Sicherheit des Gesamtsystems. Die TCB eines Webbrowsers umfasst

- neben eigenem Quellcode, der die grafische Benutzeroberfläche realisiert,
- mindestens ein Rendering-Modul mit HTML- und CSS-Parser sowie Regeln, um die angeforderten Inhalte als Webseite auf dem Bildschirm darzustellen,
- meist auch einen JavaScript-Interpreter, zum Parsen und Ausführen von JavaScript-Code,
- diverse Programmteile, die Fonts, Bilder, Videos etc. parsen und darstellen können – häufig werden hierzu externe Bibliotheken (libpng, libjpeg etc.) verwendet,
- weitere Bibliotheken, die benutzt werden, um TLS-Verbindungen aufzubauen oder Basisfunktionalität bereitzuhalten (die Standard Library),
- und das Betriebssystem selbst, welches Funktionalität für Netzwerkverbindungen, Dateizugriffe, Grafikkartentreiber etc. bereitstellt.

Der gesamte Programmcode wird von einem Compiler zu Maschinencode übersetzt, wobei der Compiler wiederum auch Teil der TCB ist.

Um Vertrauen in ein Programm herzustellen, muss jede/r sich vorstellen können, wie sich das Programm zur Laufzeit verhält. Für einfache Programme, wie mathematische Funktionen, ist dies nicht schwer. Bei komplexeren Programmen, die Netzwerkverbindungen aufbauen und abbauen sowie Daten auf der Festplatte lesen und schreiben, ist eine Intuition über das Laufzeitverhalten der Programmiersprache – der formalen Semantik – hilfreich. Jede Funktion eines Programms muss begutachtet werden, und diese ist einfacher zu verstehen, wenn sie wenig Code beinhaltet, und dieser nur sehr begrenzt auf Daten zugreifen kann (beispielsweise durch Isolation und Kapselung in einer Programmiersprache).

Programmiersprachen unterscheiden sich durch die bereitgestellten Abstraktionen, und wie diese verwendet werden. Einfache Abstraktionen, wie Variablen und Funktionen, sind nahezu allgegenwärtig. Objektorientierte Sprachen bringen ein Objektsystem mit, das Abstraktionen erlaubt (Interfaces, Vererbung, …). Typen enthalten zur Compile-Zeit Information über die Form der Werte zur Laufzeit. Mit Hilfe von statischen Typsystemen können schon zur Compile-Zeit Fehler verhindert werden (das Programm lässt sich dann nicht kom-

pilieren), die in anderen Sprachen erst zur Laufzeit als Fehler auftreten. Bei imperativen Programmen werden Berechnungsabläufe beschrieben, wohingegen bei deklarativen Programmen die Beschreibung des Problems im Vordergrund steht. Funktionale Sprachen wie *Haskell* oder *OCaml* unterstützen die Entwicklung von deklarativen Programmen, und dank ihres ausdrucksstarken Typsystems können viele komplexe Invarianten durch das Typsystem sichergestellt werden. Proof-Assistenten, wie Coq, Isabelle und HOL4, oder abhängig typisierte Sprachen wie *Agda* und *Idris*, erlauben mathematische Spezifikationen als Typ zu formulieren, der sichergestellt wird. Ein einfaches Beispiel: die Länge einer Liste erhöht sich nach dem Hinzufügen eines Listenelements um eins,

add : n a list -> a -> (n + 1) a list

Bestehende Betriebssysteme sind sehr umfangreich (Millionen von Zeilen von Code) und zum Großteil in der imperativen und maschinennahen Sprache C vor Ewigkeiten (faktisch: Jahrzehnten, was aber bei der Entwicklungsgeschwindigkeit von Computern Ewigkeiten gleichkommt) entwickelt worden. Und sie werden weiterentwickelt, dabei wird häufig Abwärtskompatibilität angestrebt, somit wächst die Codemenge stetig.

MirageOS

Unser Ansatz, den ich im Folgenden vorstelle, basiert darauf, ein Betriebssystem von Grund auf neu zu entwickeln, statt mit bestehendem Code zu arbeiten. Vor rund 10 Jahren wurde *MirageOS* an der University of Cambridge gestartet. Beim Design von MirageOS fließen Erfahrungen aus bestehenden Betriebssystemen ein. Das Design umfasst neben Performance, Skalierbarkeit, Sicherheit (*Defense in Depth*) auch Lesbarkeit und Modularisierung. Als Programmiersprache wird OCaml eingesetzt, eine Multiparadigmen-Sprache mit funktionalen und objektorientierten Merkmalen sowie automatischer Speicherverwaltung und programmierbarem Modulsystem, eine formale Semantik des Sprachkerns. Einige Bestandteile, wie die Laufzeitumgebung von OCaml, sind nach wie vor in C entwickelt, aber nur in der Größenordnung von Tausenden Zeilen, statt Millionen wie in anderen Betriebssystemen.

Heutzutage benutzen viele Dienste Virtualisierungstechnologien wie Hypervisoren, um die einzelnen Dienste stark voneinander zu isolieren, damit der Schaden bei einer Kompromittierung gering ist. Die Aufgabe eines Hypervisors auf einem physikalischen Computer ist, die Ressourcen wie Arbeitsspeicher, Festplatte, Netzwerkkarten zu verwalten und den einzelnen virtuellen Maschinen zuzuweisen. Auch der Scheduler, der entscheidet, welche virtuellen Maschinen auf welchen Prozessoren ausgeführt werden, ist Teil des Hypervisors. Ein herkömmliches Unix-System enthält mehrere Prozesse, die voneinander isoliert auf den verfügbaren Prozessoren ausgeführt werden. Zur Isolation gibt es virtuellen Speicher, ein Prozess kann nicht auf den Speicher eines anderen zugreifen.

MirageOS ist ein Unikernel[2], der als virtuelle Maschine auf herkömmlichen Hypervisoren (*Xen*, *KVM*, *Bhyve*, *VMM*) ausgeführt wird. MirageOS beinhaltet allerdings nur einen einzigen Prozess, braucht somit keinen Scheduler und auch keinen virtuellen Speicher. Jeder MirageOS-Unikernel ist ein maßgeschneidertes System mit einer Aufgabe. Zur Compile-Zeit wird die Funktionalität durch die Auswahl der Bibliotheken zusammengestellt. Danach werden die Zielplattform gewählt und die plattformabhängigen Treiber (Netzwerkkarte, persistenter Speicher) benutzt. Grundlegende Netzwerkdienste wie DHCP, DNS, Firewall, Webserver sind für MirageOS bereits in OCaml als Bibliotheken implementiert und können nahezu beliebig zusammengestellt werden. Das virtuelle Maschinen-Image eines DNS-Servers ist beispielsweise etwa 3 MB groß, enthält neben dem Boot-Code und der OCaml-Laufzeitumgebung einen Netzwerkkartentreiber sowie einen TCP/IP-Stack und benutzt die DNS-Bibliothek. Ein Dateisystem, Prozessverwaltung, Nutzerverwaltung, interaktive Shell sind hier nicht notwendig und daher gar nicht erst im Unikernel enthalten. Die Angriffsfläche wird dadurch enorm reduziert (locker um zwei Größenordnungen), und durch die Benutzung von OCaml sind diverse Angriffsvektoren (Pufferüberläufe etc.) bereits eliminiert.

MirageOS (Abbildung 1 a) führt auf dem Hypervisor direkt die OCaml-Laufzeitumgebung aus und spart somit im Vergleich zu einem herkömmlichen System (Abbildung 1 b) viele Layer ein, die zur Komplexität beitragen.

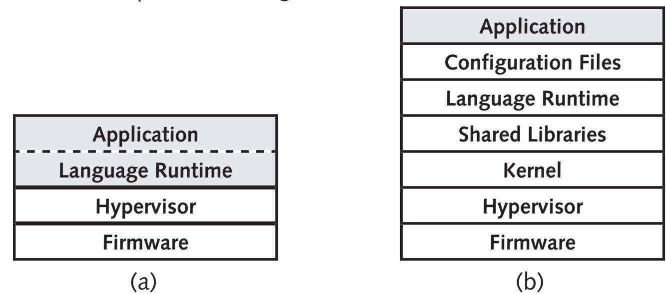

Abbildung 1: Software-Layer, (a) MirageOS, (b) herkömmlich. Thomas Gazagnaire, CC-BY-SA-4.0

Die Basis von MirageOS bilden hunderte Bibliotheken, die möglichst modular und deklarativ entwickelt wurden und werden. Durch das Modulsystem in OCaml abstrahieren wir zum Beispiel die Zielplattform: ein Unikernel kann sowohl zum Testen und Debuggen zu einem normalen Unix-Binary kompiliert werden, als auch zu einer virtuellen Maschine. Die einzelnen Betriebssystemkomponenten sind als Interfaces spezifiziert, und enthalten erweiterbare Fehlerdefinitionen. Eine konkrete Implementierung muss diesem Interface genügen (die entsprechende Funktionalität zur Verfügung stellen), kann aber neue Fehlerfälle hinzufügen. Konsumenten der Interfaces können auf Fehler programmatisch reagieren. Das erlaubt es beispielsweise, ein Dateisystem zu implementieren, das Daten via Netzwerk liest und schreibt – wo somit Kommunikationsfehler auftreten können. Jedes dieser Interfaces ist versioniert, eine neue Version muss nicht die identische Funktionalität wie die vorherige bereitstellen.

Die von uns entwickelten Bibliotheken, die Protokolle implementieren, wie TLS, TCP/IP, DNS, DHCP oder Git, haben meist einen Encoder und einen Decoder, um Binärdaten in typisierte Datenstrukturen zu wandeln oder einen Fehler zurückzugeben. Das zentrale Protokoll-Handling ist rein funktional und führt selbst keine Kommunikation aus. Die Funktion bekommt einen Zustand und ein Paket, und liefert einen neuen Zustand und möglicherweise Pakete zum Ausgeben zurück. Ein dediziertes Modul kümmert sich darum, Pakete zu empfangen, diese dem rein funktionalen Protokollhandling weiterzugeben und Pakete zu versenden. Der Vorteil ist, dass das Protokollhandling, meist eine State Machine, ohne reale Eingabe und Ausgabe getestet werden und mit nur lokalem Verständnis nachvollzogen werden kann.

Da ein Unikernel nur die unbedingt notwendige Funktionalität umfasst, ist dessen Konfiguration und Administration weniger komplex als ein General-Purpose-System. Daher sind Unikernels einfacher zu betreiben, und können auch von Nicht-ExpertInnen betrieben werden. Daher muss nicht mehr der IT-SpezialistIn im Freundeskreis vertraut werden, sondern der Betrieb kann von jeder Person selbst gemacht werden. Aktuell werden MirageOS-Unikernels nur als Quellcode verteilt, in Zukunft sind aber auch plattformspezifische Binärpakete denkbar. Sollte in einer benutzten OCaml-Bibliothek eine Sicherheitsschwachstelle gefunden werden, müssen alle Unikernels, die diese Bibliothek benutzen, neu kompiliert und deployed werden.

Als Beispiel soll hier *Canopy* dienen, ein MirageOS-Unikernel, der zum Großteil im Frühling 2014 in kurzer Zeit (eine Woche) entstanden ist. Canopy ist ein Content-Management-System, es stellt Content – in Form von Markdown-Dateien in einem Git-Repository – als Webserver zur Verfügung. Der Unikernel beinhaltet neben einem Webserver, sprich einem kompletten TCP/IP-Stack, einer HTTP-Implementierung und einer TLS-Bibliothek auch eine Git-Implementierung, die dazu benutzt wird, das über Boot-Parameter angegebene Repository in den Speicher zu klonen. Das Git-Repository enthält auch Konfigurationsinformationen (Name des Blogs, UUID, Startseite, Stylesheet). Um einen neuen Artikel einzupflegen, wird dieser in das Git-Repository *committed* und Canopy erhält eine spezielle HTTP-Anfrage, bei der Canopy das in den Speicher geklonte Git-Repository vom Server updatet. Berechtigungen, wer neue Artikel hinzufügen oder modifizieren darf, sind durch den Git-Server geregelt und nicht Teil von Canopy. Web-Feeds (Atom und RSS) werden von Canopy automatisch erzeugt und benutzen die Zeitstempel (*erstellt* und *letzte Änderung*) und Informationen zum Autor aus dem Git-Repository. Das Image der virtuellen Maschine ist 8 MB groß, beinhaltet die oben genannten Protokolle und kann auf verschiedenen Hypervisoren ausgeführt werden. Canopy benutzt keinen persistenten Speicher, sondern nur Arbeitsspeicher (etwa 20 MB plus die Größe des Git-Repositories). Canopy braucht sonst an Systemressourcen nur eine virtuelle Netzwerkkarte. Kein USB, keine Tastatur, keinen Bildschirm. Logs können bei Bedarf via syslog an einen anderen Rechner gesendet werden. Mein Blog (*https://hannes.nqsb.io*) benutzt Canopy, aber auch andere Seiten benutzen Canopy (*http://canopy.mirage.io*, *http://robur.io*).

Andere Beispiele sind ein autoritativer DNS-Server, ein DNS-Resolver, DHCP-Server, verschiedenste Webseiten, eine Firewall für QubesOS, ein Pong-Spiel im Qubes oder SDL-Framebuffer, und weitere, die gerade aktiv entwickelt werden.

Schlussfolgerung

Unikernels – neben MirageOS gibt es andere in anderen Programmiersprachen – sind schlanker als traditionelle Unix-Betriebssysteme. Durch die Reduktion der Trusted Code Base und durch die Verwendung einer Hochsprache sind sowohl die Angriffsfläche als auch die Menge der Angriffsvektoren minimiert. Da schon zur Compile-Zeit die benötigten Bibliotheken ausgewählt werden, ist die Komplexität eines Unikernels geringer als die traditioneller Multifunktionsbetriebssysteme. Daher ist der Betrieb eines Unikernels einfacher und robuster. Die Performance ist gleichauf mit anderen Implementierungen: unsere TLS-Implementierung erreicht bis zu 85 % der Geschwindigkeit von OpenSSL.[3]

Die Community um MirageOS wächst stetig, ist offen und hilfsbereit gegenüber Neueinsteigenden. Die Anwendungsfälle für Unikernels sind vielfältig, von Desktop-Anwendungen über digitale Infrastruktur und Webservices, bis hin zu robusten Services als Internet der Dinge. Mein Ziel mit MirageOS ist es, mehr Menschen zu ermöglichen, ihre eigene digitale Infrastruktur zu betreiben und damit Kontrolle über die eigenen persönlichen Daten zu bekommen. Es ist noch ein langer Weg. Viele Grundlagen sind bereits vorhanden, aber andere Bibliotheken fehlen noch – wie automatisierte verschlüsselte Backups, verteilt auf mehrere Computer.

Falls MirageOS Interesse geweckt hat, gibt es mehr Information auf *https://mirage.io*, auch dazu, welche Kommunikationskanäle die Community verwendet.

Anmerkungen und Referenzen

1 *Vortragsaufzeichnung unter fiff.de/r/181024, Folien fiff.de/r/181030*

2 *Siehe Madhavapeddy A, Scott DJ (2013) Unikernels; Rise of the virtual library operating system. CACM 57(1):61–69, doi:10.1145/2541883.2541895, sowie ACM Queue 11(11). http://queue.acm.org/detail.cfm?id=2566628*

3 *Siehe Kaloper-Meršinjak D, Mehnert H, Madhavapeddy A, Sewell P (2015) Not-quite-so-broken TLS; Lessons in re-engineering a security protocol specification and implementation. 24th USENIX Security Symposium (USENIX Security '15). USENIX Association, Washington, D.C., S 223–238. https://usenix15.nqsb.io/*

4 *Mehnert H, Ohlig J, Schirmer S (2013) Das Curry-Buch; Funktional programmieren lernen mit JavaScript. O'Reilly, Beijing*

Hannes Mehnert

Hannes Mehnert, PhD, forscht in mehreren Richtungen, von Programmiersprachen (Typsysteme, Visualisierungen von Compiler-Optimierungen) über funktionale Korrektheitsbeweise von objektorientierten Programmen, IDEs für abhängig typisierte Sprachen bis hin zu Netzwerk- und Sicherheitsprotokollen (TCP/IP, TLS, OTR). Er ist Co-Autor eines Buches über indische Küche und funktionale Programmierung in JavaScript.[4] Seit 2014 arbeitet er an MirageOS mit, von 2014 bis 2017 als PostDoc an der University of Cambridge, UK, seit 2018 bei einer gemeinnützigen GmbH in Berlin (*http://robur.io*). In seiner Freizeit ist Hannes nicht nur ein Hacker, sondern auch Barista. Er reist gern mit seinem Liegerad und repariert es auch selbst.

#FIfFKon17