Secure Operating Systems

Letting application developers shot themselves in the foot...

Andreas Happe

PUBLISHED BY ANDREAS HAPPE

HTTPS://SPECIAL-CIRCUMSTANCES.AT/WEBSEC/

First printing, October 2019

Inhaltsverzeichnis

Isolation auf verschiedenen Ebenen

In diesem Bereich werden verschiedene Aspekte eines Computersystems auf die Isolation bzw. Abgrenzung verschiedener Elemente hin diskutiert.

0.1 Warum Isolation?

Warum wurde dieser Ansatz gewählt? Die Grundlage aller Sicherheitsfragen ist die Frage nach dem schützenswerten Gut — also welche Operationen bzw. Daten vor Angreifern zu schützen sind. Die klassiche Antwort auf diese Frage ist die CIA-Triade:

C–Confidenciality : nur berechtigte Personen dürfen auf Daten zugreifen.

I–Integrity : nur berechtige Personen dürfen Daten modifizieren.

A–Availability : das System muss "zugreifbar" sein, ansonsten kann es nicht verwendet werden.

Sofern nur ein einziger Anwender ein Programm zu einem Zeitpunkt alleine auf einem Computer ausführen kann, und dabei keine Daten mit anderen Programmen oder Anwendern geteilt werden können (keine Dateisysteme, Netzwerkfreigaben, Side-Channels, etc.), ist die Integrität und Vertraulichkeit der Daten gewährleistet. Problematisch ist es, wenn mehrere Benutzer Programme auf einem Computer ausführen können bzw. Daten/Operationen geteilt werden. Hierbei muss sicher gestellt werden, dass die Zugriffe der Benutzer untereinander auf Integrität und Confidenciality hin überprüft werden. Confidenciality und Integrity sind daher stark von der Separierung und Isolation der Benutzer abhängig (Analog, Isoluation von VMs, Containern, Prozessen, etc.).

Availability ist stark von der Verfügbarkeit geteilter Ressourcen (wie z. B. CPU-Zeit, Arbeitsspeicher, Storage, Netzwerkbandbreite) abhängig. Hier muss der Zugriff einer Partei so eingegrenzt werden, dass diese nicht negativ andere Parteien beeinträchtigen kann. Die involvierten Parteien müssen also voneinander Isoliert werden.

0.2 Betrachtete Ebenen

Im Zuge dieser Vorlesung betrachten wir folgende Ebenen:

- Hardware und Firmware: die Server/Desktop/Mobile-Hardware
- Betriebssystem und Treiber: dies entspricht dem klassichen Linux-Kernel
- Prozesse: Anwendungen die gerade als ein Benutzer ausgeführt werden.

Zusätzlich wurden verschiedenste Virtualisierungsebenen inkludiert:

- Virtuelle Maschinen: mehrere Betriebssysteme können parallel auf der identen Hardware betrieben werden. Wir betrachten hier primär L1-Hypervisors wie Xen oder VMWare ESX die zwischen der Hardware und dem Betriebssystem liegen.
- Container sind eine leichtgewichte Virtualisierungslösung welche eine Gruppe von Prozessen isoliert. Sie wurden initial zum Verteilen von Anwendungen verwendet, bieten aber mittlerweile auch Sicherheitsfeatures.

1. Hardware- und Firmware

Hardware ist die “niedrigste” von uns betrachtete Systemschicht. Wenn ein Angriff gegen diese erfolgreich ist, wird der Sicherheit der darüber liegenden Schichten das Fundament entzogen. Hardwareangriffe besitzen zumeist eine physikalische Komponente, ein Angreifer kann also nicht rein virtuell bzw. remote agieren.

1.1 Angriff: Tailored Access Operations

Während eines Tailored Access Angriffs wird eine Hardware-Backdoor direkt in/auf der Hardware plaziert. Dies kann während der Produktion oder z. B. auch später während des Transports der Hardware zum Einsatzort/Kunden geschehen. Klassicherweise sind Motherboards oder Netzwerkequipment das Ziel dieser Art des Angriffs.

Im Zuge der Snowden-Enthüllungen wurde bekannt, dass die NSA Tailored Access Angriffe mit Kosten von ca. $500 pro Gerät bezifferte. 2019 konnten private Hacker für ca. $200 CISCO-Firewalls mit einer Hardware-Backdoor versehen[1].

Da eine Untersuchung jeder gelieferten Hardwarekomponente unrealistisch ist, kann man als Administrator hier nur reaktiv agieren. Basierend auf einer Inventarisierung der verwendeten Hardware kann man bei Bekanntwerden von Tailored Access Operations möglichst zeitnahe die Hardware ersetzen.

1.2 Angriff: Management-Tools

Server befinden sich zumeist in physikalisch schwer erreichbaren Rechenzentren. Dies erschwert die Administrationstätigkeiten da einfache Aufgaben (wie z. B. das Betätigen des Reset-Schalters) mit einem hohen zeitlichen Aufwand verbunden sind. Aus diesem Grund werden hier meistens Remote-Management Möglichkeiten angeboten.

Diese entsprechen einem separaten Computer innerhalb des eigentlichen Computers und werden BCM (Baseband Configuration Management), LOM (Lights Out Management) oder OOB

[1] `https://www.wired.com/story/plant-spy-chips-hardware-supermicro-cheap-proof-of-concept/`

management (Out-of-Bands Management) genannt. Typischerweise werden zumindest folgende Möglichkeiten angeboten:

- Bildschirmweiterleitung
- virtuelle Eingabe Devices (z. B. Tastatur und Maus)
- virtuelle Medien (z. B. zum Booten und Setup eines neuen Betriebssystems)
- Power-Management

Ein Angreifer mit Zugriff auf diese Management-Tools besitzt also die gleichen Möglichkeiten wie ien Angreifer mit Zugriff auf die Hardware. Ergo müssen diese Tools auch abgesichert werden:

- Betrieb über ein eigenes, gesichertes, internes Management Netzwerk
- Nur autorizierter Zugriff möglich (User-Accounts, keine Default-Accounts, gut gewählte Passwörter)
- Software muss auf dem aktuellen Sicherheitsstand gehalten werden. Falls der Hardware-Hersteller keinen Support mehr bietet, gibt es teilweise Open-Source Lösungen wie z. B. OpenBMC.

Ähnliche Lösungen gibt es bei Hardware, die schwer “greifbar” ist wie z. B. Notebooks (Computrace, zum Glück nicht mehr) oder Mobilegeräte (Mobile Device Management).

1.3 Cold Boot Attacks

Daten im Arbeitsspeicher eines Computers gehen im Normalfall während eines Reboots veloren — dies ist implizit dadurch begründet, dass RAM Zellen nicht mehr refreshed werden (periodisch mit Strom versorgt) und dadurch ihre Information verlieren.

Bei einem Cold Boot Angriff friert ein Angreifer physikalisch Speicherstellen ein und rebootet das System z. B. über ein externen Medium. Durch das Kühlen der RAM-Chips behalten diese ihre Information und diese können durch das neu gestartete Betriebssystem ausgelesen werden.

Als Gegenmassnahme kann der Arbeitsspeicher mit einem temporären Schlüssel verschlüsselt werden. Dieser Schlüssel darf einen Reboot nicht überleben, auf diese Weise kann ein Angreifer den Arbeitsspeicherinhalt nach einem Reboot nicht mehr entschlüsseln. Eine Speicherung des Schlüssels im RAM verbietet sich daher selbst, es werden meistens TPMs oder Bereiche der CPU verwendet. Zusätzlich wird häufig versucht, sensible Daten in-memory nach deren Notwendigkeit zu überschreiben — dadurch wird versucht, das verwundbare Zeitfenster (während dem ein Cold-Boot Attack funktionieren würde) zu minimieren.

1.4 Firmware

Firmware ist Software, welche benötigt wird damit Hardware ihre Funktion erfüllen kann. Im Gegensatz zu Betriebssystemtreibern wird Firmware direkt in der Hardware eingespielt und ausgeführt. Firmware ist daher autark gegenüber dem Betriebssystem. Beispiele für Komponenten, die häufig Firmware benötigen sind Netzwerkadapter, Modems, SSDs, etc.

Firmware besteht meistens aus einem embedded Betriebssytem wie z. B. Linux oder Minix. Sie wird von Herstellern meist als Binärpaket bereitgestellt und kann daher nicht einfach analysiert werden.

Da Firmware Software ist, beinhaltet sie Fehler. Dies impliziert, dass Administratoren aktiv Firmware-Updates installieren müssen. Firmware würde das Setup einer mächtigen und schwer detektierbaren Backdoor ermöglichen — neue Firmware sollte daher nur von vertrauenswürdigen Quellen über sichere Transportwege bezogen werden.

1.5 DMA/Inception-Style Angriffe

Unter Direct-Memory-Access (DMA) versteht man eine Technik, bei der Geräte direkt auf den Hauptspeicher zugreifen können (ohne die Computer-CPU oder das Betriebssystem zu involvieren). Dies wird aus Latenz- und Performance-Gründen präferiert. Externe Schnittstellen die DMA erlauben sind z. B. IEEE1394/FireWire, Thunderbolt oder PCMCIA.

Ein Angreifer kann dies verwenden um als vorgetäuschtes DMA-Device den Arbeitsspeicher des Computer auszulesen bzw. zu modifizieren. Letzteres kann z. B. verwendet werden um die Passwort-überprüfen-Funktion eines Computers kurzzuschließen (jedes Passwort zu akzeptieren).

Neuere Betriebssysteme verhindern diese Angriffe indem sie diese Verwendung erst nach einer initialien Autentifizierung/Autorisierung des Hardware-Devices erlauben. Falls dies nicht möglich ist, kann die Hardwareschnittstelle digital oder physikalisch “deaktiviert” werden.

1.6 Polymorphe USB devices

Ein Angriffsvektor sind USB Geräte, die sich im Laufe der Zeit als utnerschiedliche Geräte ausgeben. Bekannst sind hier z. B. die sogenannten Rubber-Duckys. Diese erscheinen initial als USB-Sticks, wechseln aber nach mehreren Stunden ihren Betriebsmodus auf Keyboard. Über Shortcuts wird ein Textfile mit dem Windows-Editor angelegt, mit Kommands gefüllt und schlußendlich ausgeführt. Alternativ kann der USB Stick eine USB Netzwerkkarte emulieren und auf diese Weise als Netzwerksniffer misbraucht werden.

Als Gegenmassnahme können wiederum USB ports per Software oder physikalisch deaktiviert werden. Unter Linux gibt es mit USB Guard die Möglichkeit, Policies für USB Geräte zu hinterlegen.

2. Betriebssystem und Treiber

Das Betriebsystem verwaltet die Systemressourcen eines Computers und stellt diese Anwendungsprogrammen kontrolliert zur Verfügung.

Historisch gewachsen wurden initial Programme direkt auf der Hardware einprogrammiert und ausgeführt. Zu einem Zeitpunkt war genau ein Programm eines Anwenders auf einer Maschine laufend. Aus Effizienzgründen kam im Laufe der Zeit die Multi-Programm und Multi-User Funktionalität hinzu. Zum effizienten Teilen von Ressourcen und Informationen entstanden z. B. Dateisysteme und Bibliotheken.

Während des Sytemstarts übernimmt initial das BIOS oder UEFI die Kontrolle über den Computer. Wenn diese ein Betriebssystem vorfinden (bzw. einen Bootloader), übergeben sie die Kontrolle an jenes. Das OS konfiguriert nun Kernel- und User-Space, initialisiert und konfiguriert die vorgefundene Hardware und bietet schlußendlich ein Interface zum Starten für Applikationsprogramme an. Diese können wiederum, falls benötigt, Funktionen des Betriebssystems aufrufen.

Der Kernel befindet sich im Kernel-Space. Innerhalb diesens sind alle CPU-Befehle erlaubt, es kann auf beliebige Speicherbereiche zugegriffen werden. Für Anwendungsprogramme wird der Userspace verwendet. Hier ist nur ein eingeschränkter Befehlssatz erlaubt, ein Anwendungsprogramm besitzt nur Zugriff auf seinen eigenen Speicher. Der Userspace kann exportierte Kernel-Funktionen aufrufen, der Aufruf wird als Syscall bezeichnet. Bei jedem Wechsel zwischen User- und Kernel-Space müssen “teure” Wartungsarbeiten druchgeführt werden, daher sollten diese Wechsel aus Performancegründen minimiert werden.

Im Kernel-Space befinden sich Treiber, Dateisysteme, Netzwerkstacks, etc. Da jeder Teil des Kernelspaces auf alle anderen Teile zugreifen kann, würde ein Sicherheitsfehler in einer unwichtigen Komponente trotzdem ein kompromittiertes Gesamtssytem zur Folge haben. Ein logischer Ansatz wäre es, Funtionen, die sich innerhalb des Kernel-Spaces befinden zu minimieren und statt dessen Treiber, Dateisysteme und ähnliches im User-Space zu betreiben. Die Bezeichnung für solche Systeme ist Microkernels, Vertreter sind z. b. Minix oder L4. L4 ist minimal und bietet nur sieben Operationen für Anwendungsprogramme an. Aufgrund der Größe von Microkernel kann deren Fehlerfreiheit teilweise formal bewiesen werden. Der große Kritikpunkt an Microkernel ist, dass sie durch die vileen User-/Kernel-Space Wechsel ineffizient sind. Real verwendete Systeme verwenden

daher zumeist monolithische Kernel (also Treiber, Dateisysteme, etc. im Kernel-Space) oder Hybridsysteme (Microkernel bei denen allerdings viele Funktionen in den Kernelspace verschoben wurden).

2.1 UEFI Secure Boot

Secure Boot wird verwendet um den Bootvorgang des Betriebssystems abzusichern. Die UEFI firmware besitzt einen öffentlichen Schlüssel, wird eine Komponente gestartet (z. B. der Bootloader oder Kernel) muss diese Komponente mit dem dazugehörenden privaten Schlüssel signiert worden sein. Dadurch weiss der Anwender, dass der Betriebssystemkernel von einer authentischen Quelle bezogen und nicht zwischenzeitlich verändert wurde.

Wird Linux mittels Secure Boot verwendet gelten Einschränkungen:

- nur signierte kernel module (Treiber) können geladen werden.
- kexec kann nur mit signierten Kernel verwendet werden.
- Hibernation und Resume sind deaktiviert.
- User-Space Zugriff auf Speicher (user-space DMA) ist deaktiviert.

Ein Kritikpunkt an UEFI Secure Boot ist, dass die Person mit der Kontrolle über den private Schlüssel schlußendlich Kontrolle über die Software, die auf einem Computer ausgeführt wird, besitzt. Kommerzielle Hardware/Software-Herstellen wollten teilweise die Kontrolle über diese Schlüssel nicht an die Käufer der Hardware übertragen bzw. weigerten sich, Möglichkeiten für Kunden zu schaffen selbst Schlüssel zu hinterlegen.

2.2 Verifizieren von Dateisystemen (IMA/EVM und dm_verity

Nehmen wir an, ein Kernel wurde mittels UEFI Secure Boot gestartet. Woher kann dieser nun erkennen, dass das Filesystem nicht offline verändert wurde? Hierfür kann IMA/EVM verwendet werden. Vereinfacht ausgedrückt, werden Hashsummen für Dateien generiert und die signierten Hashsummen als erweiterte Attribute mit den Dateien gespeichert. Der Kernel erlaubt den Zugriff auf Dateien nur wenn die Signaturen korrekt überprüfbar sind. Falls ein TPM Chip in das System integriert wurde, kann dieser verwendet werden um auch zur Laufzeit Dateien zu verändern (der TPM chip signiert die neuen Hashsummen ohne dass der private Schlüssel im Speicher vorrätig gehalten werden muss).

dm_verity erlaubt die transparente Überprüfung der Integrität eines Datenträgers. Für jeden Block des Datenträgers wrid ein kryptographischer Hash generiert und gespeichert. Ein Block wird vom Kernel nur gelesen, wenn der Hash übereinstimmt. dm_verity unterstützt nur read-only Operationen und wird z. B. für Systempartitionen (die immer als vollständiges Image eingespielt werden) verwendet.

2.3 Linux Security Modules

Um 2002 herum gab es Bestrebungen, das Linux Berechtigungskonzept um einen MAC-Modus (Mandatory Access Control) zu erweitern. Da es mehrere konkurrierende Implementierungen gab, wurde der Kernel um das LSM interface erweitert. Dieses erlaubt es, dynamisch ein security module (welches eine MAC implementiert) in den Kernel zu integrieren. Bekannte LSM-Implementierungen sind z. B. SELinux (RedHat-Umfeld) oder AppArmor (Debian-, Ubuntu- und SuSE-Umfeld).

2.4 Exkurs: Datenverschlüsselung

Eine Alternative zur klassichen Integritätssicherung ist das Verschlüsseln von Daten mit integritätssichernden Verfahren. Dadurch wird nicht nur deren Integrität, sondern auch deren Ver-

traulichkeit gewährleistet. Ein wichtiger Punkt hierbei ist, dass die Verschlüsselung immer nur für data-at-rest gilt. Innerhalb eines aktiven Systems hat ein Systembenutzer Zugriff auf die entschlüsselten Daten. Dieser Schutzmechanismus schützt also davor, über gestohlene Festplatten DAten zu verlieren, aber nicht vor einem Hacker, der in ein aktives System eingedrungen ist[1].

Problematisch ist die Verwaltung des Verschlüsselungskeys: da dieser nicht auf einem unverschlüsselten Laufwerk innerhalb des Systems vorliegen darf, muss er entweder vor der Entschlüsselung eingegeben werden oder z. B. in einem TPM vorliegen.

Die Verschlüsselung erfoglt zumeist im Kernel-Space: dies hat Performance-Gründe, ist aber auch in der Sicherheit bedingt da Daten im User-Space angreifbarer sind. Prinzipiell kann man zwischen block- und datei-basierter Verschlüsselungsmechanismen unterscheiden.

2.4.1 Block-based Encryption

Bei einer block-basierten Verschlüsselung werden alle Lese- und Schreibzugriffe auf eine Festplatte verschlüsselt. Im Normalfall wird im System eine virtuelle Festplatte dargestellt; alle Zugriffe werden verschlüsselt und schlußendlich nur in verschlüsselter Form auf einer physikalischen Festplatte persistiert. Da alle Zugriffe verschlüsselt werden, ist diese Methode für das verwendete Dateisystem vollkommen transparent, sie müssen also auch nicht zur Verwendung mit der Verschlüsselung angepasst werden. Dies bedeutet allerdings auch, dass das Filesystem nicht Zugriffe für die Verschlüsselung optimieren kann. Ein Vorteil ist, dass keine komplexe Policy für die verschlüsselten Dateien notwendig ist (da alle Dateien gleich behandelt werden). Ein Nachteil ist der potentiell auftretende Performance-Overhead da alle Dateien verschlüsselt werden, auch wenn nur eine kleine Teilmenge der Daten wirklich sensible und verschlüsselungs-würdig ist. Ein weiteres Problem kann die Reencryption bzw. der Wechsel des Schlüssels bei einer naiven Implementierung sein. Da das Blockdevice nicht weiss, welche Blöcke bereits Daten beinhalten, muss jeder Block gelesen, entschlüsselt, neu verschlüsselt und danach wieder geschrieben werden. Dies muss für jeden Block der Festplatte geschehen und nicht nur für die verwendeten Blöcke (da diese nicht dem Blockdevice bekannt sind).

2.4.2 File-based Encryption

Bei der datei-basierten Verschlüsselung werden einzelne Dateien verschlüsselt. Dies führt zwar potentiell zu dem Problem, das komplexe und fehleranfällige Policies verwendet werden müssen, dafür kann dieser Ansatz zu einer höheren Performance führen da nur sensible Dateien verschlüsselt werden. Während diese Möglichkeit schon von frühen UNIX Systemen vorgesehen wurde (mittels der chattr-Operation) wurde diese nie implementiert. Aktuell (2019) gibt es Bestrebungen per-file Verschlüsselung in einzelne Dateisysteme zu implementieren: aktuell in ext4 und F2FS. Beides sind Dateisysteme die im Mobil-Umfeld verwendet weren — die fein-granulare Verschlüsselung dürfte hier die Akku-Laufzeit positiv beeinträchtigen.

2.4.3 Hardware-assisted Encryption

Die Verschlüsselung kann auch in geeignete Hardware ausgelagert werden. Die Grundidee ist, dass der Anwender der Festplatte/SSD vor Verwendung ein Passwort mitteilt und alle Daten von dem Controller (innerhalb der SSD/Festplatte) mit diesem Passwort verschlüsselt werden. Dies erschwert natürlich das Booten von einer verschlüsselten SSD. Ein häufiger Standard in diesem Umfeld ist OPAL. Unter Linux hat sich OPAL nicht durchgesetzt; Bedenken gab es über die Datensicherheit, die Frage wie Softwareupdates in SSDs/Festplatte eingespielt werden können und der Umstand, dass CPU-based encryption immer effizienter wurde (und daher die CPU-Auslastung durch die Verschlüsselung immer überschaubarer).

[1] Hint: die meisten Drucker verschlüsseln daher integrierte Festplatten, daher ist ein explizites Shreddern der Festplatten nicht notwendig.

3. Virtualisierung

Bei der Virtualisierung wird einem Gast-Betriebssystem (das in einer virtuellen Maschine, VM läuft) von einem Hypervisor/Hostsystem Hardware vorgetäuscht. Das Betriebssystem innerhalb der virtuellen Maschine besitzt Treiber für die bereitgestellte Hardware. Bei er sogenannten Paravirtualisierung ist dem Gastsystem bekannt, dass es innerhalb einer VM betrieben wird, dadurch kann dieses Hardwarezugriffe optimieren. Alternativ kann das Gastsystem keine Kenntnisse über die Virtualisierung besitzen, in diesem Fall werden Treiber für konventielle Hardware innerhalb der VM verwendet.

Virtualisierung erlaubt das Betreiben mehrerer Betriebssysteme auf der identen Hardware. Dadurch, dass parallel mehrere virtualisierte Maschinen auf der identen Hardware betrieben werden wird eine höhere Kosten- und Energieeffizienz erlangt (da dadurch die Hardware höher ausgelastet wird bzw. Leerlaufzeiten vermieden werden können). Virtuelle Maschinen können einfach zwischen Host-Systemen verschoben werden (da sie die realle Hardware des Hosts von der virtualisierten Hardware welche von der VM erwartete werden kapseln) und vereinfachen so die Administration.

3.1 Security Impact

Aus Sicherheitssicht sollten virtuelle Maschinen voneinander getrennt agieren. Auf diese Weise können Applikationen “sauber” isoliert werden: ein Angreifer, der Zugriff auf eine Applikation in einer VM erlangt, besitzt keinen Zugriff auf eine Applikation in einer zweiten VM auf der identen Hardware. VMs besitzen relativ statisch zugeordnete Ressourcen (CPU, RAM, Festplattenplatz) und bieten auf diese Weise eine Schutzmöglichkeit gegenüber Denial-of-Service-Angriffen.

Ein wichtiges Merkmal von virtuellen Maschinen ist, dass jede VM einen eigenständigen Kernel besitzt. Ruft eine Applikation in einer virtuellen Maschine eine Operation auf, wird ein syscall an den Kernel in der virtuellen Maschine gesendet. Dieser setzt einen hypercall an den Hypervisor/das Hostsytem ab. Dieses führt nun den Hardwarezugriff durch. Diese weitere Hierarchieebene führt zu Effizienzverlusten, erhöht aber auch die Sicherheit: ein Angreifer muss initial in eine VM einbrechen (Z. B. über einen Webserver der in der VM betrieben wird), dann eine Möglichkeit finden in der VM “root”-Rechte zu erlangen und anschließend aus der VM in das Hostsystem auszubrechen. ERst dann kann er versuchen in eine weitere VM einzubrechen.

Während dies die Komplexität eines Angriffs erhöht, gab es erfolgreiche VM-Escapes. Aus diesem Grund müssen Administratoren sich sowohl um Sicherheitsupdates der verwendeten VMs als auch um Sicherheitsupdates für das Hostsystem/den Hypervisor kümmern.

Virtualisierungslösungen besitzen zumeist ein (web-basiertes) Managementsystem über welches mit den virtuellen Maschinen kommuniziert, administrative Tätigkeiten ausgeführt, und die Konfiguration der virtuellen Maschinen angepasst werden kann. Dieses Interface entspricht einem BMC-Management System und muss auch dementsprechend abgesichert werden.

Ein weiteres Problem sind side-channel attacks: während VMs sich nicht direkt untereinander beeinflussen können, kann das Verhalten gemeinsam genutzter Ressourcen sensible Informationen liefern. Beispiele wären z. B. eine VM welche über ein geteiltes Netzwerkdateisystem auf eine andere VM indirekt Einfluss nehmen kann. Da zwei VMs auf der identen realen Hardware betrieben werden, sind hardware-basierte Timing Angriffe wie z. B. RowHammer möglich.

4. Container/Sandboxes

Container sind eine leichtgewichtige Virtualisierungslösung. Container separieren Betriebssystem-Ressourcen, so sind z. B. innerhalb eines Containers nur jene Prozesse sichtbar die ebenso innerhalb des Containers gestartet wurden bzw. besitzt z. B. ein Prozess in einem Container nur eine eingeschränkte Sicht auf jene Dateien, welche explizit einem Container zugeordnet wurden. Ein Container ist also ein Prozessverbund mit eingeschränkter Sicht/Zugriff auf Ressourcen innerhalb des Hostsystems. Im Gegensatz zu VMs besitzt ein Container keinen eigenen Kernel. Auf diese Weise entfällt die damit verbundene Ineffizienz, allerdings ist dies aus Sicherheitssicht suboptimal: wenn ein Angreifer innerhalb eines Containers einen Fehler im Kernel findet und ausnutzen kann, besitzt er bereits Vollzugriff auf das Hostsystem.

Container werden gerne zum Verteilen von Software verwendet. Sie erlauben das Erstellen von leichtgewichtigen virtualisierten Maschinen welche z. B. mehrere konfigurierte Komponenten einer Applikation (z. B. App-Server, Datenbanken, Cache-Server, Web-Server) “fertig” konfiguriert beinhalten. Auf diese Weise werden komplexe Installationsprozeduren vermieden. Problematisch hierbei war, dass Container initial keine Sicherheitsmaßnahmen zur gründlichen Abschottung von Containern untereinander bzw. auch keine Absicherung des Hostsystems gegenüber dem Container besassen. Diese wurden nach-und-nach implementiert, Administratoren müssen allerdings sicherstellen, dass diese Möglichkeiten auch konfiguriert und aktiviert sind.

4.1 Sandboxes

In der IT gibt es ein ähnliches Konzept: Sandboxen. Dies sind separate Teile eines Betriebssystems welche nur stark eingeschränkt auf den Rest des Systems zugreifen können. Diese werden verwendet um nicht-vertrauenswürdigen bzw. potentiell gefährlichen Code auszuführen ohne die Sicherheit des Restsystems zu gefährden. Es wird angenommen, dass Container und Sandboxen langfristig verschmelzen werden.

4.2 VM vs Container

Verglichen zu VMs sind Container ressourcen-effizienter: da kein weiterer Kernel zwischen dem Container und dem Host zwischengeschalten wurde (wie bei einer VM), entstehen weniger Context-Switches. Ein Dateizugriff innerhalb des Containers geschieht direkt im Filesystem des Hosts — bei einer VM würde hier innerhalb der VM sowohl ein virtuelles Dateisystem als auch eine virtuelle Festplatte zusätzlich zu der Festplatte des Hosts verwendet werden und dadurch den Zugriff verlangsamen. Ressourcen werden bei Containern dynamisch zugeteilt — bei VMs wird Festplatten- und Arbeitsspeicher zumeist statisch einer VM zugeordnet und schützt daher gegenüber Overcommitment. Da beim Start eines Containers kein eigener Kernel gestartet werden muss, ergeben sich geringere Bootzeiten. Und nicht zuletzt besitzen Container-Lösungen wie z. B. Docker eine bessere Usability (für Administratoren) verglichen zu klassichen Virtualisierungslösungen.

Die Schattenseiten von Containern ist die höhere Verwundbarkeit gegenüber Kernel-Sicherheitslücken als auch die schlechtere Absicherung gegenüber Denial-of-Service-Angriffen (aufgrund der fehlenden statischen Ressourcenverteilung).

Container-Lösungen besitzen zumeist ein Management-System auf dem Container-Hostsystem. Dieses ist analog zu einem VM-Management-System oder zu einem BMC-System und muss dementsprechend abgesichert werden.

4.3 Virtualization-based Security and Desktops

Während Container initial verstärkt auf Servern vorgefunden wurden, werden sie mittlerweile auch stark von Desktop-Betriebssystemen (z. B. als flatpak) oder von mobilen Betriebssystemen wie Android oder iOS zur Isolation einzelner Desktop-Applikationen untereinander verwendet.

Eine weitere neuere Entwicklung ist Virtualisation-based Security (diese ist allerdings zumeist VM- und nicht Container-basiert). Hier werden kritische Funktionen bzw. schützenswerte Daten in virtuelle Maschinen verschoben. Wenn der Host auf diese Daten bzw. Funktionen zugreifen will, muss er diese Operationen als Netzwerkoperation der virtuellen Maschine aufrufen. Auf diese Weise werden diese stärker vom Hostsystem gekapselt. Ein Angreifer der Zugriff auf das Hostsystem erlangt, erlangt nicht automatisch auch Zugriff auf die — in der VM gekapselten — Daten. Windows 10 verwendet dieses Konzept zur Speicherung von Netzwerk-Credentials, das Linux-basierte QubesOS verwendet eine virtuelle Maschine zur sicheren Kapselung von privaten GPG/PGP-Schlüsseln.

5. Dateien und Prozesse

Anwender arbeiten schlußendlich nicht mit Computern, VMs oder Containern, sondern verwenden Dateien und Prozesse innerhalb jener.

5.1 Datei-Basics

Dateien besitzen unter UNIX traditionell drei Berechtiungsebenen: Besitzer (user), Gruppe (group) und Other (Rest of World). Analog dazu besitzt jede Datei einen eindeutigen Besitzer und eine eindeutige Gruppe. Für jede dieser drei Berechtigungsebenen können Zugriffsrechte vergeben werden: r für Leserechte, w für Schreibrechte, x um das Recht eine Datei auszuführen (execute) anzuzeigen. Wird eine Datei ausgeführt wird mit dem in der Datei enthaltenen Code ein Prozess gestartet. Bei Verzeichnissen kann ebenso das execute (x)-Recht gesetzt werden, bei ihnen bedeutet dies, dass ein Benutzer den Inhalt des Verzeichnisses auflisten kann (also die Dateien innerhalb des Verzeichnisses listen kann).

Hier ein Beispielslisting welches die Zugriffsrechte der Datei /etc/passwd ausgibt

```
# ls -ahl /etc/passwd
-rw-r--r--. 1 root root 3.7K Sep 17 19:49 /etc/passwd
```

Bei dem Listing sieht man, wie die Rechte der Datei */etc/passwd* mittels dem Kommando *ls* ausgegeben werden. Der Datei ist der User *root* und die Gruppe *root* als Besitzer zugeordnet. Die Zugriffsrechte sind *-rw-r–r–*.. Die erste Stelle kann aktuell noch ignoriert werden, diese wird von den Zugriffsrechten für den Besitzer (*-rw*, Lese- und Schreib-Rechte) gefolgt. Sowohl die Gruppe als auch alle anderen Benutzer können nur lesend (*r–*) auf die Datei zugreifen.

Mit dem Kommando *chown* kann der Besitzer einer Datei geändert werden, mit *chgrp* die besitende Gruppe. Mittels *chmod* können die Zugriffsrechte modifiziert werden.

Zusätzlich gibt es mehrere vordefinierte Attribute welche unter UNIX mittels *lsattr* ausgelesen und mittels *chattr* gesetzt werden können. Diese Attribute wurden initial definiert, wurden allerdings teilweise niemals implementiert. Beispiel für nicht implementierte Attribute wären E (Datei wird verschlüsselt), u (wenn die Datei gelöscht wird kann sie wieder hergestellt werden), s (Datei wird

beim Löschen mit 0en überschrieben) oder c (Datei wird transparent komprimiert). Umgesetzt wurde z. B. die Möglichkeit eine Datei als immutable zu markieren (kann nicht beschrieben, umbenannt oder gelöscht werden) oder eine Datei ein den append-only Modus zu versetzen (Daten können nur am Ende angehängt werden, z. B. für Logdateien).

Ein Beispiel der Verwendung:

```
# touch testfile
# lsattr testfile
--------------e----- testfile
# sudo chattr +i testfile
# rm testfile
rm: cannot remove 'testfile': Operation not permitted
# mv testfile testfile.renamed
mv: cannot move 'testfile' to 'testfile.renamed': Operation not permitted
```

Zuerst wurde mit *touch* eine neue leere Datei erstellt und anschließend mit *lsattr* deren Attribute ausgegeben. Das gesetzte e-Attribut sagt aus, dass die Dateiinhalte als Extends gespeichert werden — eine ext4-Eigenheit. Mit dem Kommando *chattr* wird nun das immutable Attribut gesetzt. Anschließend kann die Datei weder gelöscht noch umbenannt werden.

5.2 Extend Attributes und ACLs

Applikationen benötigen teilweise die Möglichkeit Metadaten an Dateien anzuhängen, dies wurde unter Linux über den extended attributes (xattr oder attr genannt) Mechanismus implementiert. Extended Attribute können an Dateien angehängt werden und werden über einen eindeutigen Schlüssel identifiziert. Dieser ist hierarchisch und wird über . unterteilt, Attribute die von Benutzern zugeordnet werden sollten immer mit *user.* beginnen. Ein extended Attribute kann ein Wert zugewiesen werden. Attribute werden mit *setfattr* gesetzt und mittels *getfattr* ausgelesen (das Attribut *user.fubar* wird auf Wert *somevalue* gesetzt):

```
# touch testfile2
# setfattr -n user.fubar -v somevalue testfile2
# getfattr testfile2
# file: testfile2
user.fubar

# getfattr -d testfile2
# file: testfile2
user.fubar="somevalue"
```

Extended Attributes werden unter Linux zur Implementierung von ACLs verwendet. Diese erweitern das traditionelle UNIX Dateirechte-Konzept (*rwx*). Die Limitierung, dass eine Datei nur einen Besitzer und eine Gruppe besitzt machte das Teilen von Daten auf traditionellen UNIX-Systemen komplexer. Dies führte häufig dazu, dass die Zugriffsrechte von Dateien sehr lax vergeben wurden um das Teilen zu vereinfachen. Dies hatte einen negativen Einfluss auf die Sicherheit. Ein weiteres Problem waren Linux-Netzwerkdateiserver welche Windows-Clients bedienten. Unter Windows können einer Datei Zugriffsrechte für beliebige Benutzer und Gruppen zugeordnet werden, dies war auf das Benutzer/Gruppen-Konzept von Linux nicht einfach konvertierbar.

ACLs sind Access Control Lists und erlauben die Angabe von Zugrifsberechtiungen zusätzlich zu den traditionellen UNIX-Rechten. Die konfigurierten Zugriffsrechte werden über extended attributes den Dateien und Verzeichnissen zugeordnet. Die Zugriffsrechte können mit *setfacl* gesetzt und mit *getfacl* ausgelesen werden. Das Format der Recht ist *u:<benutzername>:<rechte>* für Benutzerrechte bzw. *g:<gruppenname>:<recht>* für Gruppen. Ein Beispiel:

```
# touch testfile3
# setfacl -m g:wheel:w testfile3
# ls -ahl testfile3
-rw-rw-r--+ 1 andy andy 0 Oct 23 20:54 testfile3
# getfacl testfile3
# file: testfile3
# owner: andy
# group: andy
user::rw-
group::rw-
group:wheel:-w-
mask::rw-
other::r--
```

Hier wurde bei der Datei *textfile3* Schreibrechte der Gruppe *wheel* gegeben. Diese werden zusätzlich zur eigentlichen Gruppe (*andy*) gespeichert. Beim Listen der Datei fällt das + am Ende der Zugriffsrechte auf: dies ist das Zeichen, dass zusätzliche Berechtiungsinformationen bei der Datei hinterlegt wurden. Diese werden schlußendlich mit *getfacl* angezeigt, dabei fällt auf das sowohl Zugriffsrechte für die eigentliche Gruppe *andy* als auch für die Gruppe *wheel* angezeigt werden.

5.3 Prozesse

Ein Prozess ist die Ausführung eines Programms zur Laufzeit. Das Programm wird zumeist aus einer Datei geladen (bei welcher das executable-Flag gesetzt war). Eine Datei wird unter UNIX immer einem User zugeordnet, ebenso wird ein Prozess auch immer einem User zugeordnet. Der User, als der ein Prozess ausgeführt ist, ist by default allerdings nicht der Besitzer der Datei sondern der aktuell eingeloggte Benutzer der die Datei ausführt. Dies wird auch *effective user* genannt. Sytemressourcen wie Speicherverbrauch, geöffnete Dateien, etc. werden unter UNIX immer einem offenen Prozess zugeordnet.

Prozesse, die dem gleichen User zugeordnet sind, können sich untereinander beeinflussen (z. B. über Signale). Dies impliziert, dass aus Sicherheitsgründen Prozesse unterschiedlicher Funktionen (z. B. webserver und datenbankserver) mit unterschiedlichen Benutzern betrieben werden sollten. Auf diese Weise wird verhindert, dass ein kompromittierter Datenbankprozess ebenso den Webserverprozess beeinflussen kann.

5.4 The Problem with Root

Unter UNIX besitzt die Benutzer-ID (UID) 0 eine besondere Bedeutung. Benutzer, welche über diese UID besitzen, sind als Administratoren (traditionell *root* genannt) markiert und besitzen weitgehende Sytemrechte. Sie können Hardware modifzieren, Benutzerrechte verändern und im Normalfall auf alle Dateien innerhalb des Systems zugreifen. Root-User sind quasi “Gott” auf einem UNIX-System. Dies ist teilweis historisch gewachsen: UNIX kommt aus dem Grossrechner-Umfeld wo viele Benutzer auf einem System erlaubt waren, aber nur wenige Benutzer die Systemverwalutng über hatten. Ein normaler Benutzer sollte niemals mit der Systemverwaltung betraut werden.

Da ein Prozess, der als root ausgeführt wird, massive Zugriffsmöglichkeiten hat, sind diese ein lukratives Angriffsziel. Wird ein Fehler in einem root-Prozess gefunden und kann dieser ausgenutzt werden, besitzt ein Angreifer quasi uneingeschränkten Zugriff auf das System.

5.5 SUID and Capabilities

Unter UNIX können viele Funktionen nur mit root-Rechten ausgeführt werden. Ein Beispiel hierfür ist die Möglichkeit beliebige Netzwerkpakete zu generieren. Dies ist problematisch, da z. B. das *ping*-Kommando diese Möglichkeit benötigt um die Erreichbarkeit externer Hosts zu überprüfen. Jedem User, der das *ping*-Kommando benötigt das Administratoren-Passwort zu geben würde die Sicherheit vollkommen unterlaufen.

Als Alternative kann das *Sticky-Bit* bzw. *SUID-Binaries* verwendet werden. Ist dieses bei einem ausführbaren File gesetzt, wird beim Starten des Files nicht der aktuell eingeloggte Benutzer als effektiver Benutzer verwendet, sondern der Besitzer des, dem Programm zugrundeliegenden, Files verwendet.

Dies erlaubt es nun jedem Benutzer das ping-Kommando mit root-Rechten aufzurufen. Allerdings ist dies ein Sicherheitsproblem, da eine Datei mit einem gesetzten SUID-Bit und einem Programmfehler ausreicht, damit ein Angreifer mit root-Rechten auf das System zugreifen kann. Falls ein Programm mit SUID-Bit auf eine Datei zugreift, erfolgt dieser Zugriff ebenso mit den Rechten des Besitzers des ausführbaren Programms. Ein SUID-Programm mit root-Rechten kann also auf alle Dateien des Systems zugreifen.

Ein guter Administrator wird regelmäßig eine Liste der Dateien mit SUID-Rechten am System erstellen und versuchen, diese Liste minimal zu halten.

Die binäre Unterteilung zwischen root/non-root wurde von Linux-Entwicklern früh als Sicherheitsproblem erkannt. Um diesen harten Split zu verbessern wurden *Capabilities* eingeführt. Capabilities beschreiben eine Teilmenge der Administratoren-Operationen, so kann z. B. ein Prozess mit *cap_net_raw* beliebige Netzwerkpakete generieren, auch wenn er keine root-Rechte besitzt. Capabilities werden intern ausführbaren Dateien über extended attributes zugeordnet:

```
# setcap cap_net_raw=ep /bin/ping
```

Nun kann das Kommando *ping* ohne Root-Rechte als normaler Benutzer gestartet werden.

5.6 SecComp

Um 2002 gab es vermehrt Bestrebungen, private Rechenkapazitäten als Grid wohltätigen Operationen wie z. B. folding@home oder seti@home zur Verfügung zu stellen. Dabei kam die Frage auf, woher Benutzer sich sicher sein konnten, dass der auf ihren Geräten ausgeführte Code nicht ihr System negativ beeinflussen würde. Als Lösung wurde seccomp geboren.

Ein Prozess kann über einen syscall aus dem normalen Modus in den seccomp Modus wechseln. Ein Wechsel vom seccomp Modus in den normalen Modus ist anschließend nicht mehr möglich, dies ist a one-way street. Im seccomp Modus sind nur noch vier Syscalls für den Prozess erlaubt: *read* und *write* um auf bereits geöffnete Dateien und bereits offene Netzwerkverbindungen zuzugreifen, *exit* um den Prozess zu beenden und *sigreturn* um auf Signale zu reagieren. Dies stellte eine sehr restriktive Sandbox dar.

Die Sandbox war leider zu restriktiv und wurde später durch *seccomp-bpf* ersetzt. BPF steht für Berkely-Paket-Filter und beschreibt die Sprache einer minimalen VM welche die Kategorisierung und Filterung von Netzwerkpaketen bzw. von Aufrufen erlaubt. Bei *seccomp-bpf* werden die Regeln für das kontrollierte Programm in BPF geschrieben und dem überwachten Programm zugeordnet. Dieser Ansatz war flexibler und wird z. B. von Android 8.0 verwendet um die Isolution von Apps zu erzwingen. Ähnlich wird dies von Docker verwendet um Container voneinander zu trennen oder z. B. von Chrome und Firefox um die Prozesse der jeweiligen Browser-Tabs vor einander abzusichern.

5.7 Side-Effects zwischen Prozessen

Es gibt unter UNIX noch weitere Möglichkeiten wie sich unterschiedliche Prozesse beeinflussen können.

Alle Prozesse können z. B. auf das /tmp Dateisystem lesend und schreibend zugreifen. Dies wird teilweise von Programmen verwendet um temporäre Dateien zu speichern. Falls hierbei nicht-ausreichende Zugriffsrechte gewählt werden, können weitere Programme diese Informationen abgreifen.

Die Verwendung der traditionellen graphischen Oberfläche X11 ist ebenso problematisch, da hier jedes Programm auf die Eingaben der anderen graphischen Programme zugreifen kann. Hier verschafft der Einsatz der neuen Wayland-Technologie Besserung.

Generell wird auch im Desktop-Bereich stärker auf Container-Lösungen zur Isolution der jweiligen Applikationen gesetzt. Ein Beispiel hierfür ist z. B. flatpak.

5.8 Mandatory Access Control

Linux implementiert als Basis ein discretionary access control Model (DAC). Dies bedeutet, dass ein Benutzer voller Diskretion über die Vergabe von Rechten für seine Daten an andere Benutzer besitzt. Insbesondere der root-User hat starke Kontrolle über sowohl seine, als auch für die Daten anderer Benutzer.

Im Behördenumfeld bzw. bei Unternehmen mit starker Geheimhaltung ist dies nicht ausreichend. In diesem Fall will man auf Systemen Policies hinterlegen, welche Benutzerzugriffe auf Dateien "hart" limitieren. Auch der Root-Benutzer muss sich an diese Policies halten. Diese Policies sind also bindend und werden vom Betriebssystem exekutiert. Der Name für solche Systeme ist daher *Mandatory Access Control (MAC)*. Unter Linux werden solche als LSM (Linux Security Module) abgebildet, die bekanntesten Beispiele sind SELinux und AppAmor.

Ein Beispiel für ein AppArmor-Profile (welches Zugriffsrechte für *tcpdump* einschränkt):

```
#include <tunables/global>

/usr/sbin/tcpdump {
  #include <abstractions/base>
  #include <abstractions/nameservice>
  #include <abstractions/user-tmp>

  capability net_raw,
  capability setuid,
  capability setgid,
  capability dac_override,
  network raw,
  network packet,

  # for -D
  capability sys_module,
  @{PROC}/bus/usb/ r,
  @{PROC}/bus/usb/** r,

  # for -F and -w
  audit deny @{HOME}/.* mrwkl,
  audit deny @{HOME}/.*/ rw,
  audit deny @{HOME}/.*/** mrwkl,
  audit deny @{HOME}/bin/ rw,
  audit deny @{HOME}/bin/** mrwkl,
  @{HOME}/ r,
  @{HOME}/** rw,
```

```
  /usr/sbin/tcpdump r,
}
```

6. Interals

Wie werden die jeweiligen Konzepte unter Linux abgebildet?

6.1 Namespace

Namespaces erlauben es Partitionen über Systemressourcen einzuführen. Dadruch wird gemeint, dass ein Prozess in einem Namespace nicht mehr alle Systemressourcen sieht, sondern nur jene, die seinem Namespace zugeordnet wurden.

Nach dem Systemstart besitzt Linux einen namespace: jeder Prozess sieht also alle Ressourcen. Es können nun weitere Namespaces angelegt werden; Prozesse können Namespaces joinen bzw. verlassen.

Folgende Systemresoucen können über namespaces kontrolliert werden:

- mount: Verwaltung von Mount-Points
- pid: Prozesse in einem Namespace sehen nur andere Prozesse in dem Namespace. Der Vater-Namespace sieht auch alle Prozesse des Kind-Namespaces.
- net: dies entspricht einem virtualisierten Netzwerk-Stack. Eine Netzwerkkarte kann genau einem Namespace zugeordnet werden.
- ipc: inter-prozess communication
- user: erlaubt für virtuelle Benutzergruppen in namespaces. Ein Prozess der als root-User (uid: 0) in einem Namespace exekutiert wird, ist z. B. ein normaler User im user-namespace des Vater-Namespaces.
- cgroup
- UTS: namespace für hostnamen

6.2 Control Groups (cgroups)

Unter Linux werden Ressourcen-Limits traditionell immer an Prozessen angehängt. In der Realtität will man allerdings zumeist eine Gruppe von Prozessen mit Limits versehen: z. B. will ein Admin den Speicherverbrauch einer Applikation limitieren, die Applikation besteht dabei aus mehrerne Webservern und einem Datenbankserver.

Control Groups erlauben es, Prozesse in eine Control Group zu plazieren und Ressourcen-Limits auf diese Control Gruppe anzuwenden.

6.3 Wie verwendet der Kernel diese?

Der Linux Kernel besitzt kein *Container*-Konzept. Für ihn sind Container einfach nur Gruppen von Prozessen, die vom Rest des Systems mittels namespaces abgeschottet werden und deren Ressourcen über eine gemeinsame Control Group kontrolliert werden. Der Zugriff untereinander und vom Container auf den Kernel wird zumeist über seccomp-bpf abgesichert.

Container wurden auf diese Weise implementiert, damit der Kernel nicht seine Meinung gegenüber Containern erzwingt. Gleichzeitig sollte dadurch die dynamische Evolution von Containern beschleunigt werden.

II Linux-Stuff

Dieses Kapitel betrachtet einige Sicherheitsaspekte der Linux-Konfiguration. Da es sich um eine Security- und nicht um eine Administrationsvorlesung handelt, wird auf klassiche Systemkonfiguration (z. B. Netzwerkkonfiguration) nicht direkt eingegangen.

7. User Management

Unter UNIX gibt es traditionell eine strikte Trennung zwischen normalen Benutzern und Systemusern (root, UID: 0). Benutzer mit Systemrechten werden minimiert um die Angriffsfläche bzw. Fehlbenutzungen zu vermeiden.

Die Shell dient zur Interaktion des Benutzers mit dem System. Hier wird zwischen Login- und Non-Login shells unterschieden. Ersteres ist der erste Prozess der für einen Benutzer nach einem erfolgreichem Login gestartet wird und konfiguriert zumeist das User-Environment (z. B. setzt konfigurierte Umgebungsvariablen).

Um remote (über ein Netzwerk) auf ein System zuzugreifen sollte seit 1995 SSH verwendet werden. Ältere Alternativen wie z. B. rsh oder telnet bieten keinen Integritäts- oder Vertraulichkeitsschutz, ihr Einsatz verbietet sich daher selbst. Eine SSH-Benutzerauthentication kann über Benutezrname/Passwort-Kombination oder über ein public-key-Verfahren erfolgen. Aus Sicherheitsgründen (z. B. um Brute-Force-Angriffe zu vermeiden) soltle letzters gewählt werden. Aus diesem Grund deaktivieren viele Distributionen den remote root-Zugriff sofern Benutzername/Passwort als Authentifizierungsmethode gewählt wurde.

7.1 Speicherung der User Credentials

Traditionell werden in UNIX die Anmeldedaten auf mehrere Textdateien aufgeteilt. In */etc/passwd* sind Informationen wie login, Heimatverzeichnis, Login-Shell, UID gespeichert. In */etc/group* befindet sich die Zuordnung von Benutzern zu Gruppen. Initial beinhaltete die */etc/passwd* auch den Passwort-Hash jedes Benutzers. Da auf diese Datei von allen Benutzern lesend zugegriffen werden kann, ist dies ein Sicherheitsproblem (jeder eingeloggte Benutzer könnte die Passwort-Hashes aller anderen Benutzer auslesen). Um dieses Problem zu lösen werden Passwort-Hashes mittlerweile getrennt in der */etc/shadow* gespeichert. Ein Zugriff auf diese Datei ist dem root-Benutzer vorbehalten.

Ein rein-statisches System zur Benutzerauthentication würde schnell an die Grenzen stoßen. Im Unternehmensumfeld will man gerne Benutzer gegen ein zentrales Verzeichnis authentifizieren (z. B. ActiveDirectory oder LDAP), bei kritischen Accounts würde sich die Verwendung einer Mehrfaktorenauthentifizierung anbieten. Linux ermöglicht diese Dynamik durch PAM — dem Plugable

Authentication Mechanism. In dem Verzeichnis */etc/pam.d* befindet sich eine Konfigurationsdatei pro Programm/Dienst, welches eine Benutzerauthentication benötigt. Die jeweilige Konfigurationsdatei konfiguriert, aus welchen Quellen Authentifikationsdaten bezogen, und wie diese angewendet werden sollten. Auf diese Weise können Netzwerkanmeldungen, Multi-Faktoren-Authentication aber auch optionale Features wie Passwortrichtlinien modular konfiguriert werden.

7.2 su, runuser und sudo

Die saubere Trennung zwischen normalen Benutzern und Systemusern hat seinen Ursprung auf Grossrechnern. Hier findet sich eine relativ stabile Systemkonfiguration vor, ein normaler User sollte nie Administrationstätigkeiten ausführen müssen. Dies ist auf einem privaten Desktop tendentiell nicht mehr der Fall. Falls ein Benutzer sich für Administrationstätigkeiten imemr aus- und als Administrator einloggen müßte, würde dies wahrscheinlich zu sehr nerven, und dazu führen, dass a la longue alle Benutzer mit Administratorenrechten versehen wären. Da dies suboptimal aus Sicherheitssicht ist, wurden einfache Möglichkeiten zum Wechsel der Benutzeridentität während der Laufzeit geschaffen.

Mittels des Kommandos *su* (superuser) kann ein Benutzer seine Identität auf "root" hochstufen. Hierfür ist die Eingabe des root-Passworts notwendig. Analog dazu kann ein root-Benutzer mitdem Kommando *runuser* die Identität eines anderen Benutzers annehmen — hierfür ist kein Kennwort notwendig (da dieses Kommando von einem Systemuser ausgeführt wird). Mit "su -l" kann eine login-Shell gestartet werden.

Problematisch hierbei ist, dass beim Übergang zum Superuser Umgebungsvariablen nicht sanitized, sondern großteils aus der aufrufenden Shell übernommen werden. *su* setzt die Variablen *HOME* und *SHELL* immer neu, *USER* und *LOGNAME* werden nur neu gesetzt, falls der neue Benutzername ungleich *root* ist. Dies ist problematisch da viele Programme diese Umgebungsvariablen auslesen und aufgrund dieser Aktionen ausführen. Wenn ein Angreifer eine Umgebungsvariable tainten kann, und danach *su* ausgeführt wird, hat er effektiv eine Umgebungsvariable des root-Users vergiftet.

Eine Alternative ist die Verwendung des Kommandos *sudo*. Wird dieses Tool verwendet wird ein minimales Set an Umgebungsvariablen generiert, es werden by default aber keine Umgebungsvariablen aus der aufrufenden Shell übernommen. Zur Verwendung von *sudo* muss das Passwort des aktuellen Benutzers (nicht des aufzurufenden Benutzers) eingegeben werden. Ein weiterer Vorteil von *sudo* ist, dass über die Konfigurationsdatei */etc/sudoers* pro Benutzer die ausführbaren Kommandos auf bestimmte Kommands beschränkt werden kann. Mit "sudo -i" kann eine Login-Shell gestartet werden.

7.3 Limits

Mittels *ulimit* können Ressourcen-Limits an eine Login-Shell (und alle ihre Kinder) gebunden werden. Über diese können z. B. die maximal verbrauchbare CPU-Zeit, Speicherverbrauch oder die Handhabung von Core-Dumps konfiguriert werden. Mit "ulimit -a" können die aktuellen Limits ausgegeben werden:

```
# ulimit -a
-t: cpu time (seconds)              unlimited
-f: file size (blocks)              unlimited
-d: data seg size (kbytes)          unlimited
-s: stack size (kbytes)             8192
-c: core file size (blocks)         unlimited
-m: resident set size (kbytes)      unlimited
-u: processes                       62968
```

```
-n: file descriptors                1024
-l: locked-in-memory size (kbytes)  64
-v: address space (kbytes)          unlimited
-x: file locks                      unlimited
-i: pending signals                 62968
-q: bytes in POSIX msg queues       819200
-e: max nice                        0
-r: max rt priority                 0
-N 15:                              unlimited
```

Mit *umask* können die Dateirechte neu angelegter Dateien und Verzeichnisse vorkonfiguriert werden.

Quotas können pro Dateisystem verwendet werden um Benutzern oder Gruppen Limits bei dem maximal verbrauchten Speicherplatz vorzuschreiben.

8. Service Management

Systeme benötigen zumeist mehrere Hintergrunddienste. Auf Servern umfassen diese die bereitgestellten Services, auf Desktops müssen Hintergrundprozesse gestartet werden. Das Starten und Überwachen dieser Services ist Teil dieses Kapitels.

Unter UNIX wird ein Dienst, ohne interaktiver Shell-Interaktionsmöglichkeit mit einem Benutzer, traditionell als *daemon* bezeichnet.

8.1 Init

Service-Management ist stark mit dem Init-System verbunden. Das init-System wird als erster Prozess vom Kernel gestartet (per default ist dies */sbin/init*, dies kann über die Kernel-Bootoption *boot=* konfiguriert werden). Das init-Sytem konfiguriert das System, startet daemons und bietet schlußendlich dem Benutzer die Möglichkeit sich einzuloggen.

8.1.1 SysV-Init

Im UNIX-SysV-Init werden verschiedene Shell-Scripts unter */etc* zur Konfiguration des Systems verwendet. Diese werden in einem wohl-definierten Verzeichnis (abhängig vom konfigurierten Runlevel) vorgefunden, Skripts werden in alphabetischer Reihenfolge nacheinander als Benutzer *root* aufgerufen. Per Konvention können die meisten Skripts mittels eines Parameters gesteuert werden, zumeist werden *start*, *stop*, *status* und *restart* als Parameter akzeptiert. Jeder daemon benötigt ein eigenes Skript für dessen Konfiguration.

Das Schreiben der Skripts ist nicht trivial, es sollten folgende Aspekte beachtet werden:

- Startreihenfolge und Abhängigkeiten zwischen Skripts: wurden notwendige Prozesse bereits gestartet bzw. ist z. B. das Netzwerk bereits verfügbar?
- Wechsel des effektiven Benutzers von *root* auf einen nicht-root Benutzer
- Setzen von Ressourcen-Limits
- Behandlung von gestarteten Kind-Prozessen
- Setup eines Monitoring-Tools welches abgestürzte Prozesse ggf. neu startet

Während das init-Grundsystem einfach und verständlich ist, artet das Schreiben von init-Skripts schnell in komplexen Skripts aus. Da jeder daemon ein Skript bereitstellen muss, ist dies mit

Redundanzen verbunden. Im worst-case werden defekte init-Skripts als Basis für neue init-Skripts gewählt.

8.1.2 SystemD

2010 wurde von RedHat-Mitarbeitern SystemD als neues init-System vorgestellt. Im Gegensatz zu SysV — einfaches Grundsystem, komplexe Skript-basierte Konfiguration — besteht es aus einem komplexen Grundsystem, verwendet aber einfache Konfigurationsdateien.

Grundeinheit von Systemd sind sog. Units welche jeweils einen Systemdienst kapseln. Eine Unit wird mittels eines Konfigurationsfiles beschrieben, Optionen regeln das genaue Verhalten des daemons. Beispiele für Optionen:

- Kommands für das Starten/Stoppen eines Dienstes
- Abhängigkeiten zwischen Units (dynamische Startreihenfolge)
- zu verwendende User und Gruppen
- Ressourcen-Limits, Control Group Configuration
- Verwendung von AppArmor, SELinux und Capabilitites
- Umgebungsvariablen
- Verhalten im Fehlerfall, Neustarten von Diensten

Da dieses System einen Bruch mit 50 Jahren UNIX-Tradition darstellte wurde es kontrovers empfangen. DAbei half es nicht, dass immer mehr Features in SystemD integriert wurden (cron, logging, network management, etc.) und sowohl feature-creep als auch eine neue Monokultur befürchtet wurde. Mittlerweile wird SystemD von allen größeren Linux-Distributionen verwendet.

SystemD bietet auch mehrere Sicherheitsoptionen:

- mit der *ProtectHome* Option kann Diensten der Zugriff auf das Home-Directory (*/home*, */root*, */run/user*) utnersagt, oder ein leeres Home-Directory vorgetäuscht werden.
- *ProtectSystem* erlaubt es, Systemverzeichnisse wie */user* oder */boot* für den jeweiligen Dienst schreibgeschützt erscheinen zu lassen. Dies kann auf */etc* bzw. auf das Gesamtsystem ausgedehnt werden.
- *PrivateTmp* täuscht jedem Dienst ein privates und leeres */tmp*-Verzeichnis vor.
- Der Zugriff auf Netzwerke kann für den jeweiligen Dienst eingeschränkt werden.
- Der Zugriff auf Hardware-Devices kann für den jeweiligen Dienst eingeschränkt werden.
- Der Zugriff auf Kernel-Einstellungen (z. B. über die virtuellen */proc* und */sys* Dateisysteme) kann eingeschränkt werden.

9. Network Security

Da es sich bei der Vorlesung um eine Security-, und nicht um eine Administrations-, Vorlesung handelt, werden einzelne Security-relevante Aspekte beleuchtet während grundlegende Konfiguration nicht erwähnt wird.

9.1 Minimierung der Angriffsfläche

Server müssen an einem Netzwerkport lauschen (engl. “listen”) um Kontaktanfragen entgegen nehmen zu können. Wenn ein Service nicht an ein Netzwerkinterface gebunden (engl. “bind”) ist, kann es über dieses Netzwerkinterface keine Anfragen entgegen nehmen und daher auf diese Weise auch nicht angegriffen werden.

Um listen Sockets der Protokolle TCP und UDP samt Informationen für den Service, der diesen Socket geöffnet hat, anzuzeigen, kann das Kommando “netstat -tulpn” verwendet werden:

```
# sudo netstat -tulpn
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
↪  PID/Program name
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
↪  949/cupsd
tcp        0      0 127.0.0.1:44321         0.0.0.0:*               LISTEN
↪  1165/pmcd
tcp        0      0 127.0.0.1:4330          0.0.0.0:*               LISTEN
↪  7154/pmlogger
tcp        0      0 127.0.0.1:43599         0.0.0.0:*               LISTEN
↪  11537/vim
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
↪  1/systemd
tcp6       0      0 ::1:631                 :::*                    LISTEN
↪  949/cupsd
tcp6       0      0 ::1:44321               :::*                    LISTEN
↪  1165/pmcd
tcp6       0      0 ::1:4330                :::*                    LISTEN
↪  7154/pmlogger
```

```
tcp6       0      0 :::111                  :::*                    LISTEN
↪  1/systemd
udp        0      0 0.0.0.0:5353            0.0.0.0:*
↪  846/avahi-daemon: r
udp        0      0 0.0.0.0:46351           0.0.0.0:*
↪  846/avahi-daemon: r
udp        0      0 192.168.8.101:68        0.0.0.0:*
↪  936/NetworkManager
udp        0      0 0.0.0.0:111             0.0.0.0:*
↪  1/systemd
udp        0      0 127.0.0.1:323           0.0.0.0:*
↪  869/chronyd
udp6       0      0 :::42800                :::*
↪  846/avahi-daemon: r
udp6       0      0 :::5353                 :::*
↪  846/avahi-daemon: r
udp6       0      0 :::111                  :::*
↪  1/systemd
udp6       0      0 ::1:323                 :::*
↪  869/chronyd
```

Eine neuere Möglichkeit diese Information anzuzeigen wäre “ss -pult”:

```
# sudo ss -pult
Netid     State       Recv-Q      Send-Q                  Local Address:Port
↪     Peer Address:Port
udp       UNCONN      0           0                             0.0.0.0:mdns
↪         0.0.0.0:*          users:(("avahi-daemon",pid=846,fd=12))
udp       UNCONN      0           0                             0.0.0.0:46351
↪         0.0.0.0:*          users:(("avahi-daemon",pid=846,fd=14))
udp       UNCONN      0           0                 192.168.8.101%enp3s0:bootpc
↪         0.0.0.0:*          users:(("NetworkManager",pid=936,fd=19))
udp       UNCONN      0           0                             0.0.0.0:sunrpc
↪         0.0.0.0:*          users:(("systemd",pid=1,fd=42))
udp       UNCONN      0           0                           127.0.0.1:323
↪         0.0.0.0:*          users:(("chronyd",pid=869,fd=5))
udp       UNCONN      0           0                                [::]:42800
↪            [::]:*          users:(("avahi-daemon",pid=846,fd=15))
udp       UNCONN      0           0                                [::]:mdns
↪            [::]:*          users:(("avahi-daemon",pid=846,fd=13))
udp       UNCONN      0           0                                [::]:sunrpc
↪            [::]:*          users:(("systemd",pid=1,fd=44))
udp       UNCONN      0           0                              [::1]:323
↪            [::]:*          users:(("chronyd",pid=869,fd=6))
tcp       LISTEN      0           5                           127.0.0.1:ipp
↪         0.0.0.0:*          users:(("cupsd",pid=949,fd=7))
tcp       LISTEN      0           5                           127.0.0.1:44321
↪         0.0.0.0:*          users:(("pmcd",pid=1165,fd=0))
tcp       LISTEN      0           5                           127.0.0.1:dey-sapi
↪         0.0.0.0:*          users:(("pmlogger",pid=7154,fd=7))
tcp       LISTEN      0           5                           127.0.0.1:43599
↪         0.0.0.0:*          users:(("vim",pid=11537,fd=3))
tcp       LISTEN      0           128                           0.0.0.0:sunrpc
↪         0.0.0.0:*          users:(("systemd",pid=1,fd=41))
tcp       LISTEN      0           5                               [::1]:ipp
↪            [::]:*          users:(("cupsd",pid=949,fd=6))
tcp       LISTEN      0           5                               [::1]:44321
↪            [::]:*          users:(("pmcd",pid=1165,fd=3))
tcp       LISTEN      0           5                               [::1]:dey-sapi
↪            [::]:*          users:(("pmlogger",pid=7154,fd=8))
```

```
tcp       LISTEN      0           128                                  [::]:sunrpc
↪              [::]:*         users:(("systemd",pid=1,fd=43))
```

Das konkrete Beispiel zeigt meinen Desktop. Programme die auf localhost (127.0.0.1 bzw. :::1) hören, können nur ausgehend von der lokalen Maschine angesprochen werden und stellen daher ein kleineres Sicherheitsrisiko dar. Prozesse die auf 0.0.0.0 hören, sind über alle konfigurierten Netzwerke der Maschine erreichbar.

Aus Sicherheitssicht ist es empfehlenswert die Anzahl der (öffentlichen) Services zu minimieren. Dies vermindert die Angriffsfläche — ein Service, mit dem ein Angreifer nicht kommunizieren kann, ist ein Service das nicht direkt angegriffen werden kann.

9.2 Linux-Firewall

Linux besass im Laufe der Zeit mehrere Kernel-Level Firewalls (ipfw, ipfwadm, ipchains, iptables, nftables). Aktuell (Stand 2019) wird iptables von nftables ersetzt. Diese implementiert eine minimal BPF-VM im Kernel welche zur Beschreibung von Firewall-Regeln verwendet wird. Diese VM wird nun verwendet um Pakete zu klassifizieren und potentiell schlußendlich Regeln auf Netzwerkpakete anzuwenden.

In dieser Vorlesung betrachten wir nur Firewall-Aspekte welche eine Kommunikation zum Host betreffen und nicht Netzwerk-Monitoring Funktionen.

Falls Services mit listening Ports minimiert wurden, sollten eigentlich keine Firewall-Regeln für eingehende Verbindungen benötigt werden. Als zusätzliche Absicherung kann die Firewall unterrichtet werden, nur Traffic auf diesen listening Ports entgegenzunehmen und allen weiteren Traffic abzulehnen:

```
# Set default chain policies
iptables -P INPUT REJECT

# Accept on localhost
iptables -A INPUT -i lo -j ACCEPT

# accept traffic for SSH
iptables -I INPUT -p tcp --dport 22 -j ACCEPT

# accept traffic for web server (https)
iptables -I INPUT -p tcp --dport 443 -j ACCEPT
```

Mit diesen Regeln werden per default alle eingehenden Verbindungen unterbunden (REJECT). Danach wird explizit Kommunikation zu sich selbst (localhost) erlaubt bzw. die Ports 22 und 443 freigeschalten.

Bei einem Serve stellt sich die Frage, ob dieser überhaupt Verbindungen nach extern initiieren muss. Normalerweise antwortet ein Server nur auf eingehende Anfragen und kontaktiert wenige externe Server wie Software-Update-Server und NTP-Server um die Systemzeit zu synchronisieren. Hierfür könnte folgendes Regelwerk zusätzlich verwendet werden:

```
# Set default chain policies
iptables -P OUTPUT REJECT

# Accept on localhost
iptables -A OUTPUT -o lo -j ACCEPT

# allow softwareupdates
iptables -A OUTPUT -p tcp --dport 443 -d update.server.fake -j ACCEPT
```

```
# allow NTP for system time synchronisation
iptables -A OUTPUT -p udp --dport 123 -d ntp.server.fake -j ACCEPT

# Allow established sessions to receive traffic
iptables -A INPUT -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
```

Hier wird zuerst der Default auf das Sperren ausgehenden Traffics gestellt. Danach wrden initiale Verbindungen für localhost, für den update-server (nur HTTPS) und für den Time-Server (nur NTP) freigeschalten. Schlußendlich werden Pakete, welche als Antwortpakete für etablierte eingehende Verbindungen (hier hatten wir SSH und HTTPS initial freigeschalten) zugelassen.

An diesem Beispiel kann man auch ein Problem von IP-Tables erkennen: es wird für den Update-Server Port 443 (HTTPS) freigeschalten, es gibt allerdings keine Möglichkeit zu kontrollieren, welche Daten über diese verschlüsselte Verbindung ausgetauscht werden. Bei einer funktionierenden end-to-end encryption sollte dies auch technisch nicht möglich sein. Um dies dennoch zu ermöglichen, muss man auf ISO/OSI-Level 7 (application layer) filtern. Dies kann mittels eines lokalen HTTP/S-Proxy geschehen. Dieser wird von Applikationen, welche auf externe HTTP/S-Ressourcen zugreifen wollen, verwendet und besitzt Regeln, welche Zugriffe erlaubt und welche unterbunden werden müssen (der Proxy besitzt Vollzugriff auf die ausgehenden HTTP Requests bzw. auf die eingehenden Antwort-Dokumente). In diesem Fall würden alle ausgehenden HTTPS Verbindungen zusätzlich per Firewall-Regel unterbunden und nur die ausgehende Kommunikation vom HTTP Proxy erlaubt werden.

10. Software Management

Ein aufgesetztes System ist zumeist nicht ausreichend zum Betrieb einer Applikation. Zumeist müssen Supportprogramme, Bibliotheken und auch Applikationen auf dem System hinzugefügt werden. Im Betrieb muss das System periodisch (bzw. in Ausnahmefällen auch außerplanmäßig) auf den aktuellen Sicherheitsstand gehoben werden.

Fragen, die während des Software-Managements häufig auftauchen:

- wie kann die Integrität einer Software überprüft werden?
- welche Dateien wurden druch eine Installation hinzugefügt bzw. modifiziert?
- wie kann ein UPdate bzw. eine Installation wieder rückgängig gemacht werden?

In diesem Kapitel wird die historische Entwicklung betrachtet. Generell wird als Empfehlung ausgesprochen, Software und Updates immer nur von vertrauenswürdigen Quellen automatisiert zu beziehen.

10.1 Selbst-Compilieren von Software

Open-Source Software wird unter Linux häufig in Quelltextform zugestellt. Hierbei werden häufig während der Konfiguration und Installation folgende Schritte durchlaufen:

1. Konfigurieren der Software (z. B. mittels *GNU autotools*). Hier wird überprüft, ob benötigte Komponenten (Compiler und Bibliotheken) auf dem Build-Host vorhanden sind.
2. Kompilieren, z. B. mittels *gcc*
3. Installieren, z. B. mittels *make install*
4. Deinstallieren, potentiell mittels *make uninstall*

Bei diesem Vorgang können mehrere Probleme auftreten: der Anwender muss selbst die Integrität der heruntergeladenen Sourcen überprüfen. In einer perfekten Welt würde er auch die Korrektheit des Source Codes analysieren (z. B. überprüfen, ob keine backdoor im Source Code versteckt wurde).

Während des Konfigurierens werden wahrscheinlich Bibliotheken als fehlend erkannt. hier muss der Anwender manuell diese dependencies auflösen und installieren. Falls während des Kompilierens Fehler aufgrund der Benutzerkonfiguration auftreten, muss der Anwender diese selbst berreinigen. Nach fertiger Kompilation sollte der Anwender die erzeugten Pakete auf Korrektheit

hin überprüfen. Es gibt nach der Installation keine Möglichkeit zu erkennen, welche Dateien installiert wurden. Eine automatisierte Deinstallation ist nicht möglich. Ein Update besteht aus Herunterladen, Konfigurierne, Kompilieren und Installation einer neuen Version — das Löschen alter Versionen ist nicht vorgesehen. Alle diese Schritte sind zusätzlich zeitaufwendig, dies kann bei einem kritischen Systemupdate problematisch sein.

Bei kommerzieller Software, welche als Archiv ausgeliefert wird, entfallen das Konfigurieren und Kompilieren der Software, ansonsten sind die Probleme ident.

10.2 Linux Package Management

Die Situation mit selbst-kompilierten Programmen ist für größere Deployments problematisch. Hier kommen nun Distributionen ins Spiel, diese bieten automatisiert Paket-Updates über zentralisierte Server an.

Die zwei großen Paketfamilien unter Linux sind *DEB* und *RPM*. Ersteres wird von Debian-basierten Distributionen wie z. B. Ubuntu verwendet, letzteres von RedHat-basierten Distributionen wie z. b. Fedora. Beide Formate bieten Features wie Versionskontrolle, Dependency- und Konfliktmanagement, Integritätsüberprüfung als auch Uninstallation.

Die Bedienung auf der Kommandozeile ist realtiv ähnlich: *apt update && apt install postfix* (Ubuntu) vs *dnf install postfix* (Fedora). Die Paketnamen sind zumeist auch sehr ähnlich.

Beide Systeme arbeiten mit zentralisierten Repositories samt Trust-Management. Die Integrität der Softwarepakete wird sowohl auf Paketebene (Signaturen) als auch auf Transportebene sichergestellt. Wird TLS verwendet, ist auch die Vertraulichkeit auf der Transportebene gewährleistet. Achtung: werden neue Repositories manuell hinzugefügt, muss der Administrator die Integrität jener überprüfen und erst danach hinzufügen.

Nach mehreren Zwischenfällen mit Packetservern besitzen nun einige Distributionen eine Zero-Trust Architektur: durch einen kompromittierten Softwarepaketserver können keine Pakete mit Schadcode erstellt und zugestellt werden. Ähnliche Setups werden aktuell von der Autoindustrie für Software-Updates analysiert[1].

Eine aktuelle Entwicklung sind reproductive builds[2]: durch sie wird der Link zwischen Source Code und den generierten Binaries gewährleistet. Aktuell sind Distributionen dabei, dies umzusetzen. Distributionen führen zusätzlich eine Qualitätskontrolle der generierten Pakete durch und backporten Sicherheitspatches für die Softwareversionen, welche durch die Distribution bereitgestellt wird (da diese teilweise älter als aktuelle Softwareversionen sind, bekommen sie nicht automatisch Sicherheitsupdates durch den Softwarehersteller).

Durch den Einsatz von Distributionspackages wird die Sicherheit durch vertrauenswürdige, geprüfte und automatisierte Updates erhöht. Ein Inventar der installierten Dateien und Pakete wird durch die Distribution geführt.

Der Grossteil dieser Vorteile geht verloren, wenn Administratoren Software am Package-Management “vorbei” installieren.

10.3 Addon-Software

Wird Software am Package-Management vorbei installiert, gehen der Grossteil der Security-Benefits verloren. Die installierte Software muss wieder vom Administrator selbst gewartet werden. Warum kann diese Situation auftreten?

- die zu installierende Software ist nicht in einem Package-Repository vorhanden (z. B. aus Lizenzgründen)

[1] Quelle: lwn, `https://lwn.net/Articles/794391/`

[2] `https://lwn.net/Articles/757118/`

- die zu installierende Software benötigt spezielle Laufzeitumgebungsversionen)Ruby, JavaScript, Python, etc.) oder Bibliotheken, die in den Paketquellen nicht vorhanden sind.

Zusätzlich gibt es das Problem, das verschiedene Applikationen potentiell zueinander-inkompatible Laufzeitversionen benötigen. Da mittels des Paket-Management ursprünglich nur genau eine Version einer Software installiert werden konnte, verhinderte dies das Deployment von zwei Applikationen am gleichen Host, wenn diese zueinander inkompatible Laufzeitversionen benötigten.

Die "Lösung" für diese Probleme war eigentlich ein Rückschritt: am Applikationsserver wird die benötigte Software samt der Laufzeitumgebung und benötigten Libraries direkt im Benutzerverzeichnis eines Benutzers installiert. Zur Laufzeit wird die Applikation mit diesen lokalen Environments gestartet. Diese Lösung inkludiert eine vollkommen separate Laufzeitverwaltung (z. b. *rbenv-build* zum Kompilieren von Ruby-Versionen) als auch eine separate Bibliotheksverwaltung (z. B. *bundler* für Ruby oder *venv/virtualenv* für Python). In Extremfällen wurde von diesen Tools das gesamte Applikationsdeployment auf Produktivsystemen durchgeführt. Natürlich werden hier keine automatisierten System- oder Sicherheitsupdates durchgeführt, die Vertrauenswürdigkeit der Package-Server wurde auch druch mehrere Supply-Chain Attacks negativ beeinflusst.

Benötigt Software weitere Server-Daemons müssen auch diese installiert, konfiguriert und gewartet werden. Dies führte zu langen Installationsanweisungen. eine positive Charaktereigenschaft von ITlern ist unsere Faulheit, a.k.a. nicht gleiche Tätigkeiten wiederholen zu müssen falls diese automatisierbar sind. Dies führte zu Shell-Scripts, welche automatisiert ein System konfigurieren. Problematisch dabei ist, dass die meisten Administratoren diese Skripts nicht kontrollieren bevor sie diese als root-Benutzer aufrufen. Dies ist ein recipe for destruction bzw. für Sicherheitsprobleme. Der Höhepunkte dieser Tendenz waren Installationsanweisungen a la:

```
# curl http://somehost/somescript | bash
```

Also das Herunterladen eines Setup-Skripts von einem unbekannten Host über ungesicherte HTTP-Verbindungen und anschließende Ausführen des heruntergeladenen Skripts als root-Benutzer. What could go wrong?

10.4 Container-based Applikations-Setups

Container-basierte Setups boten einen einfachen Ausweg aus dieser Situation. Der Anwendungsentwickler packaged seine Anwendung und deren Abhängigkeiten (Bibliotheken, Daemons) in einem oder mehreren Containern und stellt diese den Systemverwaltern zu Verfügung.

Dies ist zumindest besser als das blinde Ausführen von ungeprüften Installationsskripts mit Administrator-Rechten.

Allerdings gibt es auch hier Probleme bzw. Reibungspunkte:

- Die Container werden zumeist über zentrale Stellen wie DockerHub bereitgestellt. Die identen Sicherheitsprobleme wie bei zentrallen Paketservern treten hierbei auf. Aufgrund des jungen Alters von Container-Repositories sind hier noch nicht soviele Sicherheitsmassnahmen in-place (Signaturen, reproductive builds, etc.).
- Der Inhalt eines Containers kann auch Schadcode beinhalten. Der Autor eines Containers muss ebenso vertrauenswürdig wie der Autor der Software sein. Initiale Containersysteme (wie z. B. Docker) besassen nicht wirklich Sicherheitsabsicherungen. Eine Applikation in einem Container konnte auch auf das Hostsystem zugreifen. Diese Situation wird mit neueren Containermanagement-Versionen besser.
- Container müssen regelmässig mit Sicherheitsupdates versorgt werden und neue Containerversionen müssen vom Administrator auch eingespielt werden. Auch wenn es sich hier meist um ein zentralisiertes System handelt, wird wieder eine Parallelhierarchie zur klassichen Paketverwaltung aufgebaut.

10.5 Container-based Distributionen

Mobile Betriebssysteme vermieden diese falsche Dichtonomie von Applikationen (Containern) und System-Bestandteilen (klassisches Unix-/Linux-Paketmanagement) indem sie Endbenutzern nur die Installation von Applikationen (in Containern) erlaubten und den Zugriff auf das Betriebssystem selbst verhinderten (ausgenommen gerootete bzw. gejailbreakte Telefone). Einige Linux-Distributionen gehen einen ähnlichen Weg, z. B. ChromeOS oder Fedora Silverblue.

Am Beispiel Fedora Silverblue:

- Das Basissystem wird nicht über Pakete erstellt, sondern über "versionierte Dateisystemimages". Fedora Silverblue verwendet hierfür *ostree* — dieses bezeichnet sich selbst als "git for filesystems". Wird ein Systemupdate durchgeführt wird, analog zu einem Docker Image Update, wird in einer atomaren Operation das gesamte Dateisytem geupdatet (also auch mehrere Applikationen auf einmal). Diese vesionierten Dateisysteme erlauben es, diese atomaren Systemupdates auch wieder rückgängig zu machen. Da sie im Normalfall read-only gemounted werden, erhöhen sie auch die Resistenz gegenüber Angreifern und erlauben den Einsatz von Integritätstechnologien wie *dm_verity*.
- Applikationen werden als Container von einem zentralen Repository bezogen. Updates werden ebenso über diesen zur Verfügung gestellt. Eine klassiche Paketverwaltung mittels apt/dnf/rpm/dpkg wird nicht bereitgestellt. Als Containertechnologie verwendet Fedora Silverblue flatpaks, diese bieten auch eine Sicherheits-relevante Abschottung der Container untereinander.

Es ist noch nicht absehbar (2019), ob diese Experimente den Linux-Desktop nachträglich beeinflussen werden, oder ob es sich nur um kurzlebige Experimente handelt.

10.6 Going Beyond Single Systems

Bis jetzt haben wir nur die Administration einzelner Systeme betrachtet — innerhalb eines Unternehmens müssen allerdings zeitweise hunderte Systeme verwaltete werden. Im Linux-Umfeld betrifft dies zumeist Server (mobile Systeme werden später getrennt betrachtet).

Administratoren diese Tätigkeit manuell ausführen zu lassen ist fehleranfällig. Ein weiterer Kritikpunkt an manuellen Tätigkeiten ist die fehlende Dokumentation, insbesondere wenn kurzfristig ad-hoc Änderungen an der Konfiguration durchgeführt werden (z. B. im Zuge von Debugging). In Summe führt dies zu einer fehlenden Reproduzierbarkeit und zu einem Sicherheitsrisiko: was passiert, wenn ein Systemadministrator kompromittiert wird bzw. ausfällt?

Ein weiteres Problem ist die fehlende Parallelisierbarkeit manueller Tätigkeiten. Im Falle eines kritischen Sicherheitsupdates müssen Dutzende bis Tausende Server in kürzester Zeit aktualisiert werden.

Eine Lösung für dieses Problem versprechen Tools wie ansible, chef oder puppet. Bei diesen wird meistens eine zentralle Datenbank mit Konfiguraons- und Installationsanweisungen mit einem Inventarssystem für Systeme kombiniert. Ein Administrator kann nun die hinterlegten Konfigurationsanweisungen an Systeme zugeordnet werden. Diese Anweisungen werden dann automatisiert parallel ausgeführt, die Ergebnisse dieser Operationen werden durch die Tools auch dokumentiert. Die hinterlegten Konfigurationsanweisungen sollten in einem Versionierungssystem abgelegt werden. Dadurch werden Daten generiert, die im Zuge des Change-Managements verwendet werden können. Aus Admin-Tätigkeiten werden quasi Konfirationsdaten.

Während der durchgängige Einsatz eines solchen Systems viele Vorteile bringt, ist er mit einem erhöhten initialen Aufwand verbunden und benötigt Konsequenz der Administratoren. Es ist möglich, solche Tools auch nur für Teilaspekte der Konfiguration/Wartung zu verwenden, z. B. zum Unternehmens-weiten Ausrollen von kritischen Sicherheitsupdates.

10.6.1 Infrastructure as Code

Ein Grundsatz Agiler Methoden heißt *Working Software over comprehensive documentation*.

Der Fokus auf *Working Software* anstatt auf Dokumentation schlägt sich auch bei dem Deployment (dem Installieren der Software) nieder: dies wird zumeist automatisiert als Skript durchgeführt und nicht als Dokumentation ausgeliefert (und entspricht dadurch bereits dem DevOps-Gedanken).

Da im Zuge von Agilen Methoden versucht wird möglichst früh und möglichst häufig lauffähigen Code beim Kunden bereitzustellen (bzw. als Testservice dem Kunden zur Verfügung zu stellen) passieren Installationsvorgänge regelmäßig. Um hier nun Redundanzen zu vermeiden (bzw. um Konfigurationsfehler zu verhindern) wurden hier (historisch betrachtet) Installationsanweisungen immer stärker durch automatisierte Skripts ersetzt. Danach wurden dezidierte Deploymentstools (wie z. B. *capistrano*) für das Setup der Applikation konfiguriert und verwendet. Im Laufe der Zeit wurden diese Tools nicht nur für die Applikation selbst, sondern auch für Datenbanken, Systemservices, etc. angewandt; die historische Evolution sind mittlerweile dezidierte Frameworks die zum Setup der Systeme dienen (wie z. B. Puppet, Chef oder Ansible).

Ein weiterer Vorteil dieses Ansatz ist, dass die verwendete Konfiguration innerhalb der (hoffentlich) verwendeten Source Code Versionierung automatisch versioniert und Veränderungen dokumentiert werden. Dies erlaubt das einfachere Debuggen von Regressionen.

10.7 Exkurs: Kernel Live-Updates

Linux-Systemadministratoren sind stolz auf Systeme mit jahrelangen uptimes (durchgängige Laufzeiten). Allerdings ist z. B. ein Restart nach einem Sicherheitsupdate des Kernels zwingend notwendig. Wird dieser nicht durchgeführt, wurde zwar ein neuer Linux-Kernel installiert, das System verwendet allerdings bis zum nächsten Reboot noch die verwundbare alte Version des Kernels. Der Reboot führt zu Downtimes, diese sollten offensichtlich minimiert werden.

Ein erster Schritt war der Einsatz von *kexec*: damit wird noch immer ein restart des Systems durchgeführt, allerdings kann dadurch die benötigte Reboot-Zeit verkürzt werden.

Eine neuere Lösung ist kernel live-patching. Hierbei wird zur Laufzeit der gerade exekutierende Kernel in-memory so modifiziert, dass Sicherheitsupdates ohne Reboot eingespielt werden. Unter Linux gab es mehrere, konkurrierende, Distributions-spezifische Ansätze (*ksplice*, *kgraft*, *kpatch*), mittlerweile ist das Kernel-seitige Interface grossteils vereinheitlicht. Dieses Feature bzw. das Bereitstellen von live-updates wird von Distributionen zumeist als Premium-Feature verkauft.

11. Backup

Backups sind ein wesentlicher Bestandteil des Business Continuity Management bzw. dessen Teilbereich Desaster Recovery. Auch wenn dieser Vorgang nicht direkt sicherheitsrelevant ist, sollte folgendes erwähnt werden:

- "There are two kind of people: those that do Backups and those that wish they had backups."
- Nach dem Erstellen eines Backups sollte dieses auch verifiziert werden.
- Das Wiederherstellen eines Backups sollte vor dem Ernstfall auch getestet werden.

Leider gibt es mittlerweile kein Linux Standard-Backup-Tool. Die Namen einiger Kommandozeilentools leiten sich von "alten" Bandbackups her, so steht *tar* z. B. für *tape archive*.

Aus Sicherheitssicht muss die Integrität und Vertraulichkeit von Backups wie jene der Originaldaten behandelt werden. Es macht keinen Sinn, einen Server zu verschlüsseln, wenn Backups unverschlüsselt gespeichert werden. Eine gute Verschlüsselung von Backups samt Integritätsschutz erklärt sich von selbst. Dies betrifft insbesondere Backuplösungen welche Cloud-Storage zur Ablage verwenden.

12. Intrusion Detection, Auditing und Virenscanner

Intrusion Detection Systems (IDS) dienen zur Identifikation von Angriffsversuchen bzw. zur Identifikation von Angreifern die bereits am System sind. Es wird zumeist zwischen Netzwerk- und Host-basierten IDS unterschieden, passend zum Thema der Vorlesung betrachten wir nur Host-basierte IDS.

Ein häufiges Unterscheidungsmerkmal der Methoden ist der Zeitpunkt der Anwendung. Es gibt Methoden, die permanent während der Laufzeit des Systems mitlaufen ("online") und es gibt Tools die periodisch gestartet werden, diese können auch auf Kopien eines Datenträgers angewendet werden ("offline").

Analyse-Tools benötigen einen tiefen Einblick in das aktuelle System und werden daher zumeist mit erhöhten Privilegion (z. B. als root-User) betrieben. Dies bedeutet auch, dass Angreifer versuchen, diese Tools anzugreifen, da diese quasi jede Datei am System berühren und gleichzeitig mit erhöhten Privilegien berieben werden — quasi das perfekte Angriffsziel.

12.1 Datei-basierte IDS

Bekannte Vertreter dieser Kategorie sind *tripwire* und *samhain*. Diese Tools bauen aufgrund eines bekannten "guten" Systemzustands eine Datenbank mit File-Hashes auf und können diese zu einem späteren Zeitpunkt mit den Hashes der aktuellen Dateien vergleichen. Falls ein Angreifer eine Datei zwischenzeitliche verändert hat, hat isch dadurch dessen Dateihash ebenso verändert und dies wird durch das Tool detektiert. Dies ist eine periodische Operation, die offline passiert. Ein Problem sind Dateien, die durch einen Benutzer im Normalbetrieb generiert oder verändert werden, da für diese kein Hash bzw. ein inkorrekter Hash in der Datenbank hinterlegt sind.

12.2 Prozess-basierte IDS

Bekannte Vertreter sind *wazuh* und *samhain*. Hier wird zumeist ein Agent im System installiert welcher Telemetriedaten (wie z. B. Prozesse, deren Zugriffe oder aktive Netzwerkverbindungen) erfasst und auswertet. Dies ist daher ein online-System. Falls ein verdächtiges Verhalten identifiziert

wird, wird zumeist ein zentrales System notifiziert. Neumodisch verwenden diese Verfahren auch behavioral analysis oder machine-learning Ansätze zur Pattern Recognition.

12.3 Rootkit Detection

Bekannte Vertreter: *rkhunter* und *chkrootkit*. Diese Tools analysieren ein Filesystem bzw. Memory auf das Vorhandensein von Backdoor-/Rootkit-Spuren anhand von Signaturen oder Heuristiken. Tools die Dateisysteme analysieren gehören in die offline-Kategorie, Tools die Arbeitsspeicher analysieren in die Online-Kategorie.

12.4 Virenscanner

Virenscanner haben unter Linux traditionell nicht den identen Stellenwert wie in der Windows-Welt. Wahrscheinlich liegt dies darin begründet, das Linux usprüngliche kein lohnendes Angriffsziel war. Eine weitere Eigenart ist, dass Linux initial stark als Server verwendet wurde: Virenscanner überprüfen zumeist übertragene Dateien nicht auf Schadmuster gegenüber dem Linux-Host seblst, sondern auf Windows-Schadmuster um potentielle Windows-Clients zu schützen.

Im OSS-Bereich wird gerne *clamav* als Virenscanner verwendet. Dieser wird gerne über *amavisd* innerhalb von Mailsystem zur Überprüfung transferierter Emails (bzw. deren Anhänge) verwendet.

Lange waren unter Linux Virenscanner klassiche Offline-Scanner welche periodisch liefen. Mit dem Linux-Kernel 3.8 wurde das *fanotify*-Interface erweitert, welches deine Online-Virenüberprüfung ermöglicht (jeder Dateizugriff wird automatisch auf Virenmuster hin überprüft). *clamav* utnerstützt dies seit Verion 0.99 (Dezember 2015).

Virenscanner verwenden zumest root-Rechte und müssen jede Datei analysieren. Daher sind sie, wie IDS im Allgemeinen, lohnende Angriffsziele. Sie verwenden daher Konzepte wie Sandboxing um sich selbst gegenüber Angriffen innerhalb der zu überprüfenden Dateien zu schützen.

12.5 Auditing

Unter Auditing-Software versteht man Software, welche am System Daten sammelt um Fehlerverhalten zu erkennen. Dies entspricht einem Superset der Beschreibung eines Prozess-orientierten IDS. Auditing-Software kommt eher stark aus dem System-Debug-Bereich, man kann allerdings erkennen, dass Auditing- und IDS- mittlerweile verschmelzen bzw. die identen Datenquellen zur Datensammlung verwenden.

Unter Linux ist *auditd* bekannt. Mittels dieses daemons können gezielt Systemcalls überwacht und dessen Aufrufe analysiert werden. *auditd* wird zumeist als low-level Datenquelle für weitere Systeme wie *SELinux* oder IDSes verwendet.

13. Logging

Logging ist ein wichtiger Teil der Administration: Log-Daten erlauben das nachträgliche Debuggen von Problemen, inklusive Security-Incidents. Damit dies sinnvoll möglich ist, darf ein Angreifer Log-Daten nicht modifzieren können. Dies kann durch kryptographische Methoden erreicht werden, oder durch Verwendung eines getrennten Logging-Hosts auf dem über das Netzwerk nicht direkt zugegriffen werden kann. Netzwerkfähigkeit ist auch aus einem anderen Grund wichtig: es sollten Daten aus verschiedensten Quellen gesammelt und korreliert werden. Werden Container verwendet, werden zumeist keine Daten lokal innerhalb des Containers gespeichert (da dies die *stateless-assumption* von Microservices brechen würde). Log-Daten müssen in diesem Fall extern gespeichert werden, dies führt ebenso zu einer Netzwerk-Lösung.

Schlussendlich sollten die gesammelten Logdaten auch analysiert werden. In diesem Fall ist es hilfreich, wenn die gesammelten Daten eine Struktur besitzen und dadurch zielstrebiger druchforstet werden können.

13.1 syslog

Unter UNIX ist syslog als logging-Standard etabliert. Dieses System besteht aus zwei Komponenten: einer client-seitigen Bibliothek zur Bekanntgabe von Log-Informationen und einem Serverprozess, der diese Logininformationen entgegen nimmt und persistiert. Mehrere Softwareprodukte implementieren diesen Standard, am bekanntesten dürfte *syslog-ng* sein.

Über das Client-Interface können beim Logging mehrere Informationen übergeben werden:

- facility: gibt den Systembereich/die Logquelle an, z. B. mail, news, etc.
- severity: gibt den Schweregrad des Events an, z. B. INFO, WARNING, CRITICAL
- Ein Zeitstempel gibt den Zeitpunkt des Auftretens an (ohne Zeitzone)
- Die Logmeldung als String ohne vorgegebene interne Struktur

Diese Daten werden vom Logging-Daemon gesammelt und zumeist in Log-Dateien in Textform persistiert (ein Log-File pro facility). Es findet kein Integritätsschutz statt, Daten werden zumeist periodisch gelöscht. Auf Netzwerkebene wird hierfür der Standardport 514 per UDP verwendet, neuere Varianten verwenden TLS um auf Netzwerkebene Integrität und Vertraulichkeit zu gewährleisten.

13.2 journald

Journald ist ein Teil von *Systemd* und wurde initial verwendet, wenn kein weiterer Logging-Daemon am System konfiguriert wurde. Mittlerweile verwenden einige Distributionen nur noch *journald*. Verglichen zu klassischen Logging-Daemons traf *journald* eine eher kontroverse Designentscheidung: es verwendet ein Binärformat zur Persitierung. Dieses Format erlaubt es speziell nach mehreren Datenfeldern zu filtern (dies ist bei klassichen syslog in Textform nicht möglich, da eine Syslog-Zeile einfach nur eine Zeile in einem Textfile ist).

Mit der sogenannten *sealing*-Operation können periodisch Integritätsinformationen gesichert werden, welche das Detektieren von unauthentifizierten Journaländerungen erlabut. Durch das Binärformat können allerdings Standard-UNIX-Tools wie *grep* nicht mehr direkt mit *journald*-Logs verwendet werden. Dies führte zur erwähnten Kontroverse.

13.3 ELK-Stack

Im Web 2.0-Umfeld wird für Logging-Zwecke gerne der ELK-Stack verwendet, jeder Buchstabe steht hierbei für eine Komponente:

- E: Elastic Search als Datenbank
- L: LogStash zum Sammeln von Loginformationen auf mehreren Netzwerkhosts
- K: Kibana als Webplattform zum Erstellen von Auswertungen und Dashboards

Im Kubernetes-Umfeld wird LogStash auch gerne mit *fluentd* ersetzt.

III Container

Container sind eine Technologie, welche die letzten Jahre geprägt hat. In diesem Kapitel werden einige Sicherheitsaspekte von Container beleuchtet. Das Produkt, dass Container am Markt den Durchbruch verschaffte, war Docker. Viele Docker-Features wurden mittlerweile standardisiert und dienen nun als Grundlage weiterer Container-Lösungen. Wir betrachten aus diesem Grund initial Docker und seine Container/Images. Mittlerweile bewegt sich die Entwicklung von einzelnen Containern auf das Steuern von mehreren Containern (Orchestration) hin. Hier sind docker-compose, docker-swarm und Kubernetes bekannte Lösungen von denen sich aktuell Kubernetes durchsetzt. Zusätzlich werden schlußendlich noch Services-Meshes und Cloud-Native Software erwähnt.

14. Docker

Docker ist der Name einer Containertechnologie welche stark zum Durchbruch von Containern beigetragen hat. Ein Container ist eine Form der Applikationsvirtualisierung bei der Zumeist eine Applikation (oder ein Verbund von Applikationen) vom Rest des Systems isoliert wird. Der Applikationsverbund wird Container oder Guest genannt, das System auf dem ein oder mehrere Gäste betrieben werden, wird Host genannt.

Container isolieren zumeist die Dateisystem- und Prozessansicht. Dadurch wird innerhalb des Containers ein getrenntes Dateisystem angezeigt und es werden nur Prozesse angezeigt, die innerhalb des Containers sichtbar sind. Im Gegensatz zu virtuellen Maschinen verwenden der Host und die Container den identen Kernel — dies bedingt auch die höhere Ressourcen-Effizienz die zumeist mittels Containern (verglichen zu virtuellen Maschinen) erreicht wird.

Container haben die Herangehensweise der Softwareentwicklung und des Softwaredeployments verändert. Dadurch, dass sie schnell und effizient erzeugt werden können, kann die Softwareentwicklung und das -testen innerhalb eines Containers durchgeführt werden. Dies erlaubt es, ein virtuelles Betriebssystem samt benötigten Bibliotheken innerhalb des Containers bereitzustellen und erlaubt dadurch eine genaue Kontrolle der Laufzeitumgebung. Dadurch wird auch erlaubt, dass z. B. ein Host mehrere Container mit unterschiedlichen Laufzeitversionen und Bibliotheken problemlos verwenden kann. Durch das Aufsplitten der Funktionalität auf mehrere Container (z. B. Applikationsserver-Container und Datenbank-Container) wird die Feingranularität des Softwareverbunds erhöht. Wird ein erstellter Container auch zur Auslieferung der Software an den Kunden verwendet, kann der Entwickler davon ausgehen, dass die identen Bibliotheksversionen vom Kunden verwendet werden.

Initial konnte Docker nicht als Security-Sandbox verwendet werden da Docker wenige Sicherheitskonzepte umsetzte. Initial war es einfach ausgehend von einem Container andere Container zu beeinflussen bzw. das Hostsystem zu modifizieren. Mittlerweile (2019) wurde die Sicherheitssituation verbessert, bzw. gibt es Möglichkeiten die Container voneinander (und vom Host) abzuschotten.

Mittlerweile ist das Beschreibungsformat eines Container-Images (des gespeicherten virtuellen Dateisystems) durch die Open Container Initiative (OCI) standardisiert worden. Dadurch hat sich

der Marktkampf von der verwendeten Technologie eines einzelnen Containers hin zu mehreren konkurrierenden Systemen zur Steuerung mehrerer Container verschoben.

14.1 Dockerfiles

Docker-Images werden meistens mit Hilfe von Dockerfiles gebaut. Diese Files beinhalten eine Anleitung die beschreibt, wie das resultierende Image erzeugt bzw. welche Kommandos beim Instanzieren des Containers gestartet werden sollen.

Dies entspricht dem bevorzugten deklarativen (beschreibenden) Ansatz von DevOps bzw. auch dem *Working Code over Comprehensive Documentation* Ansatz Agiler Methoden.

Ein Beispiel:

```
FROM ubuntu
ADD . /code
WORKDIR /code
RUN pip install -r requirements.txt
EXPOSE 4000
CMD ["python", "app.py"]
```

Anhand dieses Konfigurationsfiles wird folgendes ausgeführt:

- FROM wird verwendet um ein Base-Image zu definieren. In diesem Fall wird eine aktuelle Ubuntu Distribution als Basis für alle weiteren Schritte verwendet. Das Image wird zumeist von einer Docker Registry bezogen und kann auch versioniert sein (es kann also eine spezifische Version einer Linux-Distribution verwendet werden).
- mittels ADD wird ein lokales Verzeichnis in den Container kopiert, mittels WORKDIR wird innerhalb des Containers das Arbeitsverzeichnis auf das kopierte Verzeichnis gesetzt.
- RUN führt ein Kommando innerhalb des Containers aus. In diesem Fall werden Python-Dependencies installiert.
- Mit EXPOSE wird dem Container mitgeteilt, dass Port 4000 extern erreichbar sein sollte. Dies wird für alle Services, die verfügbar sein müssen, konfiguriert werden. In dem Fall wird davon ausgegangen, dass die installierte Applikation (*app.py*) auf den Netzwerkport 4000 hört.
- CMD wird nicht während des Setups des Images ausgeführt, sondern wenn ein Container instanziert (gestartet) wird. In dem Fall wird mitgeteilt dass *python app.py* gestartet werden soll.

14.2 Docker Registries

Analog zu Source Code Repositories werden auch gebaute Docker Images gerne zentral gespeichert und verwaltet. Für diesen Zweck werden Docker Registries verwendet. Diese bieten zumeist Möglichkeiten zur Überprüfung der Integrität der hochgeladenen Docker-Images als auch die Möglichkeit der Versionierung der Images an.

Docker Registries können grob in private und öffentliche Registries eingeteilt werden. Private Registries befinden sich innerhalb des Firmennetzwerkes und nur authentifiziert Benutzer können Images hochladen bzw. beziehen. Ein Beispiel hierfür wäre eine private Gitlab-Instanz samt Docker-Registry.

Es gibt auch öffentliche Docker Registries bei denen mehrere Benutzer Images hochladen- und andere Benutzer diese Container beziehen können. Häufig werden diese Registries verwendet, um Standard-Images mit verschiedenen Linux-Distributionen oder Standard-Software wie z. B. Webserver oder Datenbankserver zu beziehen.

Werden öffentliche Images verwendet, sollte man diese immer auf das Vorhandensein von Sicherheitsschwachstellen prüfen. Untersuchungen haben gezeigt, dass der Großteil der Images veraltete Systemkomponenten mit bekannten Sicherheitslücken beinhalten. Ein weiteres Problem ist es, wenn Images mit Schadcode absichtlich in einer öffentlichen Docker-Registry platziert und auf diese Weise ein Angriff durchgeführt wird.

14.3 Best Practises for Containers

Die Best-Practises in diesem Bereich sind primär für einzelne Container gedacht.

Mittels der *FROM*-Direktive wird ein bestehendes Image als Basis für ein neues Image gewählt. Dies ist zumeist einer der ersten Dockerfile-Befehle, anschließend wird das Image für die jeweilige Aufgabe mittels dem Dockerfile angepasst. Die Wahl des Base-Images sollte sorgfältig durchgeführt werden, dabei sollten nur vertrauenswürdige Quellen und aktuelle Images (mit allen aktuell bekannten Sicherheitspatches) verwendet werden. Wird ein Base-Image mit vielen Tools und Features gewählt, besitzt der Angreifer automatisch eine große Angriffsfläche und führt ebenso zu einem erhöhten Resourcenverbrauch (ein Debian Base-Image dürfte um die 250 Megabyte benötigen). Aus diesen Gründen wird bei Containern meist eine minimale Linux Distribution verwendet, als Beispiel dient hier das beliebte Alpine-Linux. Ein Alpine Base Image benötigt um die 4-5 Megabyte, ist also um ein Vielfaches kleiner als ein Standard-Linux-Image. Dies verringert die potentielle Angriffsfläche. Ebenso sind in Alpine alle Tools mit stack-smashing protections (helfen gegen Buffer-Overflows) compiliert bzw. wurde weiteres Hardening auf Systemebene durchgeführt.

Eine Grundidee von Containern ist es, dass innerhalb des Containers kein State persistiert wird — es sollen also keine Daten innerhalb eines Containers gespeichert werden. Daten werden zumeist in externen Datenbanken oder über gemountete Verzeichnisse (freigegeben über das Netzwerk oder direkt vom Host in den Gast freigegeben) gespeichert. Dies macht aus Gründen der Betriebsführung Sinn (bei einem Update kann ein neues Image mit den bestehenden Daten gestartet werden), besitzt aber auch einen positiven Effekt auf die Sicherheit. Wenn innerhalb des Images keine Daten gespeichert werden müssen, kann das gesamte Image schreibgeschützt verwendet werden. Dies erschwert es einem Angreifer Schwachstellen auszunutzen (aufgrund des fehlenden Schreibzugriffs). Ein *read-only*-Image impliziert, dass Log-Dateien auch nicht innerhalb des Images, sondern zentral in einem Log-Server gespeichert werden müssen: auch dies ist aus Sicherheitssicht eine positive Entscheidung.

Container sind Prozessverbunde; Administratoren können daher Linux-Subsysteme zur Resourcensteuerung für Prozesse (*cgroups*) auch für Container verwenden. Durch diese kann z. B. der CPU- oder RAM-Verbrauch von Containern limitiert, und dadurch DoS-Angriffe vermieden werden.

Eine interessante Einstellung ist *pids-limit*. Dadurch wird die Anzahl der Prozesse innerhalb des Containers limitiert. Wenn z. B. nur ein einziger Applikationsserver innerhalb des Containers laufen soll, kann mit mit *pids-limit:1* dies auch dem Linux-Kernel mitgeteilt werden, dieser verhindert dann das Starten weiterer Prozesse. Falls ein Angreifer nun eine Lücke findet, über die er Systembefehle exekutieren kann, kann er diese nicht ausnutzen da er hierfür einen weiteren Prozess starten hätte müssen. dies auch dem Linux-Kernel mitgeteilt werden, dieser verhindert dann das Starten weiterer Prozesse. Falls ein Angreifer nun eine Lücke findet, über die er Systembefehle exekutieren kann, kann er diese nicht ausnutzen da er hierfür einen weiteren Prozess starten hätte müssen.

Container können als *privilegierte* Container gestartet werden. In dem Fall besitzen diese weitreichenden Zugriff auf Features des Hostsystems. Aus diesem Grund sollten niemals privilegierte Container verwendet werden.

Container verwenden auch das Linux-Capability Model. Dieses wurde eingeführt um den allmächtigen *root* User einzuschränken bzw. um normalen Benutzern Zugriff auf spezielle sicherheitsrelevante Systemoperationen zu geben. Typische Capabilities sind z. B. die Fähigkeit Net-

zwerktraffic mitzusniffen, die Fähigkeit beliebige Netzwerkpakete zu erstellen (*raw sockets*) oder auch die Fähigkeit Dateisystemrechte umgehen zu können. Bei Docker-Images sind per Default viele Capabilities aktiviert, diese sollten im Zuge des Hardening-Prozesses deaktiviert werden. Das neuere OCI-Imageformat setze hier einen richtigen Schritt und aktiviert per Default nur drei Capabilities, weitere Capabilities müssen manuell aktiv hinzugefügt werden.

14.4 Docker Security Problems

Die potentiellen Sicherheitsprobleme von Docker können grob in vier Bereiche eingeteilt werden:

container breakouts : es sollte niemals möglich sein, aus einem Container aus unauthorized auf einen anderen Container oder auf das Host-System zuzugreifen.

kernel exploits : bei einer Container-Lösung verwenden alle beteiligten Container als auch der Host den identen Betriebssystem-Kernel. Dies ist der Grund für die hohe Ressourcen-Effizient von Containerlösung, bedingt leider auch, dass wenn ein Container einen Kernel-Exploit ausnutzen kann, dieser Container Vollzugriff auf alle Systeme mit diesem Kernel (inkl. dem Host-System) erhält.

DoS attacks : DoS Angriffe ausgehend von einem Container auf shared resourcen (wie z. B. verfügbare CPU, RAM, Festplattenspeicher oder Netzwerkbandbreite) treffen, sofern keine Absicherungen getroffen wurden, alle Container die auf dem gleichen Host laufen.

poisoned images : wenn nicht-überprüfte Images verwendet werden, hat dies einen negativen Security-Impact. Auf der einen Seite können diese Images *nur* das Problem haben, dass hier fehlerhafte Software eingesetzt wurde (veraltete Versionen), auf der anderen Seite kann ein Angreifer versuchen dezidierte Schadsoftware in ein Image zu verpflanzen und ein Opfer dazu zu bringen, dieses Image als Container in Betrieb zu nehmen.

14.4.1 Security Best Practises when running Docker

Docker-Engine ist ein Service, welches auf jedem Docker-Node läuft. Zur Administration muss mit diesem Service kommuniziert werden, hierfür gibt es zwei Möglichkeiten: Kommunikation über TCP (Netzwerkprotokoll) oder die Kommunikation über einen Unix Domain Socket[1] (*docker.socket*, dies kann auch über das Netzwerk über SSH geschehen. Aus Sicherheitssicht würde ich die Kommunikation mittels Unix Domain Socket bevorzugen: Unix-Systemadmins kennen SSH seit mehreren Jahrzehnten, die notwendigen Sicherheitsmaßnahmen sind daher gut bekannt. Falls TCP verwendet wird, sollte zumindest die Kommunikation mittels TLS abgesichert werden.

Bei Verwendung von *docker.socket* sollte dieser Socket niemals in einen Container inkludiert werden. Würde dies geschehen, könnte ein Prozess innerhalb des Containers diesen Socket verwenden um Docker zu steuern — in dem Fall könnte z. B. ein beliebiger weiterer Container mit privilegierten Rechten bzw. erweiterten Capabilities gestartet werden. Leider ist dieses Setup durchaus häufig vorzufinden, gerade bei Continuous Integration Systemen. Bei diesen werden häufig die Builds und Tests selbst innerhalb eines Containers ausgeführt (um Probleme mit Dependencies zu vermeiden), man will aber gleichzeitig das fertige Softwareprodukt als Container bauen (um diesen dann dem Kunden zu übermitteln). In diesem Fall will man also innerhalb eines Containers einen neuen Container bauen und benötigt daher genau diese Funktionalität, die man eigentlich vermeiden will.

Container sollten regelmäßige und automatisiert auf Schwachstellen hin geprüft werden, dies betrifft insbesondere Container, die von externen Quelle bezogen wurden. Tools die dabei verwendet werden können:

- anchore: überprüft den Container auf bekannte Schwachstellen (und listet diese als CVEs)
- clair

[1] Dies ist eine virtuelle Datei innerhalb des Dateisystems in welche Befehle geschrieben werden können.

- dagda: scannt den Inhalt eines Containers mittels eines Virenscanners.

Container sind zur Laufzeit normale Linux-Prozesse und können daher mit gewohnten Tools wie *SELinux*, *Apparmor* oder *SecComp* limitiert werden.

Alphabet geht mit *gVisor* einen Schritt weiter. *gVisor* ist ein virtueller Kernel, der als Linux-Prozess gestartet werden kann. Eingehende Anfragen an diesen Kernel werden an das zugrundeliegende Linux-System weitergeleitet. *gVisor* wird nun verwendet, um einen Container zu starten. Falls dieser Container einen Kernel-Exploit ausnutzt, wird der virtuelle Kernel gehackt (und nicht der Host-Kernel). Verglichen zu einem traditionellen Container-System erhält der Angreifer durch den Kernel-Exploit nicht automatisch Vollzugriff auf das zugrundeliegende Hostsystem.

Mittels des Tools *docker-bench-security* kann eine Docker-Installation auf häufige Sicherheitsfehler bzw. fehlende Hardeningmassnahmen hin überprüft werden. Die Ausgabe des Tools ist eine ausgefüllt Checkliste, welche die weiteren Verbesserungsmaßnahmen auflistet.

15. Services and Orchestration

Aus Gründen der Sicherheit, Testbarkeit und Wartbarkeit werden Applikationen zumeist auf mehrere Container aufgeteilt. Dies führt allerdings dazu, dass zum Testen bzw. im Produktivbetrieb mehrere Container konfiguriert, gestartet und miteinander verbunden werden müssen. Da dies mit substantiven Administrationsaufwand verbunden ist, wurde dies natürlich im Sinne des DevOps-Spirits automatisiert.

Diese Container-übergreifende Konfiguration wird Orchestration genannt, analog zu einem Dirigenten der ein Orchester aus vielen einzelnen Musikern zu einem Gesamtkunstwerk orchestriert. In diesem Umfeld gibt es zwei Lösung direkt von Docker: *docker-compose* und *docker-swarm*. Als Alternative wird von Alphabet *Kubernetes* positioniert. Im Zuge dieser Vorlesung wird docker-compose und docker-swarm betrachtet, da bei diesen der Setup-Aufwand geringer ist. Im aktuellen Marktumfeld sieht es aktuell danach aus, als ob Kubernetes die Marktführung übernommen hat.

docker-compose erlaubt das Setup eines Container-Verbundes, die Container werden allerdings nur auf einem einzelnen Computer gestartet — eine gute Lösung für lokales Testen. *Kubernetes* und *docker-swarm* verteilten die konfigurierten Container zur Steigerung der Verfügbarkeit und Performance normalerweise auf mehrere Nodes.

15.1 Docker-Swarm Konfiguration

Docker-Swarm (wie auch Docker-Compose) verwendet eine deklarative Konfigurationssprache um das Deployment der jeweiligen Container zu konfigurieren. Ein initiales Beispiel eines Webservers:

```
version: "3"
services:
  web:
    # replace username/repo:tag with your name and image details
    image: username/repo:tag
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: "0.1"
```

```
        memory: 50M
      restart_policy:
        condition: on-failure
  ports:
   - "4000:80"
  networks:
   - webnet
networks:
  webnet:
```

Im Subbaum *Services* werden die unterschiedlichen Services konfiguriert. Ein Service besteht aus einem Image und mehreren Deployment-Optionen. In dem konkreten Fall wird das Image fünf mal gestartet, jeder dieser Container darf maximal 10% CPU-Zeit eines Cores bzw. 50 Megabyte RAM verbrauchen (1.1). Im Falle eines Fehlers wird der Container neu gestartet. Zusätzlich wird der Container-Port 4000 als Port 80 veröffentlicht.

Ein weiteres Beispiel zeigt, wie zwei Container konfiguriert werden können:

```
version: '3'

services:
  web:
    image: 127.0.0.1:5000/stackdemo
    ports:
     - "8000:8000"
  redis:
    image: redis:alpine
```

Wird dieser Swarm gestartet wird sowohl ein Container (*web*) von einer lokalen Registry geladen als auch ein Container (*redis*) von einer (vermutlich öffentlichen) Registry geladen und gestartet.

Innerhalb des Docker-Swarms kann ein Container einen anderen Container über den Service-Namen ansprechen. So kann z. B. *web* auf den Redis-Cache über den hostnamen *redis* zugreifen.

15.2 Docker-Secrets

Ähnlich wie bei Source-Code Repositories sind in Container/Images gespeicherte Credentials oder private/secret keys problematisch. Gerade wenn ein Image in einer externen (oder sogar öffentlichen) Registry gespeichert wird, kann ein Angreifer potentiell Zugriff auf das Image erhalten und dadurch diese Secrets extrahieren.

Zusätzlich kommt zumeist noch ein weiteres Problem hinzu. Da mittels Docker-Compose/Swarm meistens mehrere Container verwaltet werden, muss eine Secret-Änderung auch parallel an mehreren Containern durchgeführt werden.

Ein Beispiel zur manuellen Konfiguration von Secrets mittels *Docker-Secrets*. Initial muss ein Secret mit einem Schlüssel (*my_secret_data*) und einem Wert (*This is a secret*) konfiguriert werden:

```
$ printf "This is a secret" | docker secret create my_secret_data -
```

Dieses Secret kann nun in einen Container weitergeleitet werden (im Normalfall würde man dies über ein Docker-Compose File konfigurieren):

```
$ docker service  create --name redis --secret my_secret_data redis:alpine
```

Innerhalb des Containers kann nun auf eine virtuelle Datei unter `/run/secrets/my_secret_data` zugreifen.

Secrets können auch im Betrieb gelöscht und entfernt werden:

```
$ docker service update --secret-rm my_secret_data redis
```

15.3 Kubernetes

Kubernetes ist eine Open-Source Container-Orchestration Software die ursprünglich von Google bereit gestellt wurde, schlußendlich allerdings an die Cloud Native Computing Foundation gespendet wurde. Kubernetes selbst ist stark von Borg geprägt, dies ist Googles internes Cluster-Management System (der interne Codename für Kubernetes war initial “Seven of Nine”, quasi die “freundliche” Version von Borg).

15.3.1 Architektur

Kubernetes verwendet eine Client-/Server-Architektur. Computer innerhalb des Clusters können in Nodes (diese betreiben/hosten die Container) und master (Konfigurations- und Management-Server) aufgeteilt werden.

Auf master-Systemen laufen mehrere essentielle Services:

- *etcd* ist eine verteilte Konfigurationsdatenbank und ist intern als Key-/Value-Store ausgeführt. Einzelne Kubernetes-Services lauschen auf Änderungen der Datenbank und starten in Abhängigkeit jener Aktionen. Ein Angreifer mit Zugriff auf die Datenbank hat effektiv den Kubernetes-Cluster übernommen.
- *kube-apiserver* ist der admin-seitige Management-Daemon. Administratoren können mit diesen über ein HTTP REST interface kommunizieren. Im Normalfall wird dieser durch Konfigurationstools wie z. B. *kubectl* durchgeführt.
- Der *kube-scheduler* ist der Service, welcher aktiv Container innerhalb des Clusters auf Nodes verteilt.

Auf jedem Node werden ebenso Management-Dienste ausgeführt:

- *kubelet*: ist die interne Gegenstelle zur Clusterverwaltung und startet/managed die auf dem node laufenden Container.
- *kube-proxy* ist ein Kommunikationsproxy und wird vor Container geschalten. Die gesamte “externe” Kommunikation mit Containern wird über diesen Proxy geschleust.

15.3.2 Pods

Container werden in Kubernetes nicht direkt auf nodes verteilt, es wird eine Organistaionseinheit names “pods” verwendet. Ein Pod beinhaltet mehrere Container. Die Container eines Pods besitzen den gleichen Lifecycle (werden als ein Gesamtes gestartet und gestoppt), besitzen Zugriff auf gemeinsame Ressourcen wie Environemnt, Volumes (Dateien), sin din einem gemeinsamen geteilten Netzwerk und ein Pod wird immer als ein Gesamtes auf einem node deployed (alle Container eines Pods laufen immer auf dem gleichen Node).

Die Idee hinter Pods ist der Fokus auf eine laufende logische Applikation anstatt einzelne Container zu fokusieren. Eine Applikation besteht zumeist aus mehreren Komponenten, z. B., Web-Server und Datenbank-Server. Da die Applikation nur funktioniert, wenn alle Einzelkomponenten gestartet wurden, macht hier der gemeinsame lifecycle Sinn. Aus Performance-Gründen ist es sinnvoll, dass Container eines Pods (die vermutlich stark miteinander kommunizieren) relativ lokal zueinander betrieben werden um Netzwerklatenzzeiten zu reduzieren.

15.3.3 Kubernetes Security

Die Sicherheit eines Kubernetes-Deployments ist sowohl von der Sicherheit der Kubernetes-Infrastruktur, als auch von der Sicherheit der verwendeten Container abhängig.

Die Infrastruktur ist stark von der Sicherheit der verwendenten Service-Daemons abhängig. Der Zugriff auf diese sollte nur einem eingeschränkten authentifizierten (und authorisierten) Benutzerkreis möglich sein. Besonders wichtig ist hier der Zugriff auf *etcd*, da dieser die gesamte Cluster-Configuration beinhaltet. Real-World Security Incidents beinhalteten z. B. öffentlich verfügbare *etcd* ohne Authentication von denen die Credentials der zugrunde liegenden Cloud-Plattform extrahiert werden konnten.

Jegliche Form von Master-Interfaces müssen geschützt werden, dies betriff innsbesondere das HTTP REST *kube-apiserver* interface. Werden credentials für dieses auf Containern verloren, kann auf diese Weise der gesamte Cluster übernommen werden.

Auf nodes ist die *kubelet*-Gegenstelle problematisch: per default ist der unauthentifizierte Zugriff auf diese Schnittstelle möglich. Ein Angreifer mit Zugriff auf das lokale Netzwerk (z. B., ausgehend von einem Container) kann so neue pods definieren und starten.

Prinzipiell wird Kubernetes häufig selbst in der Cloud deployed. Die Zugangsdaten für die Cloud-Infrastruktur sind in mehreren Kubernetes-Services enthalten und sind ein beliebtes Angriffsziel.

Container innerhalb eines Pods besitzen ein geteiltes Vertrauensverhältnis, dies sollte bei der Verwendung von Containern aus externen Quellen beachtet werden. Ein Angreifer, der Zugriff auf einen Container erhält, besitzt höchstwahrscheinlich früher oder später Zugriff auf die weiteren Container eines Pods. Eine häufige Angriffsart ist der Einbruch in einen Container und danach das Starten eines neuen priviligierten Containers über ein ungeschütztes internes Management-Interface. Der neue Container kann nun auf das Host-System mit erweiterten Rechten zugreifen.

Es gibt die Möglichkeit, sowohl für die Netzwerkkommunikation als auch für Pods Policies zu definieren (*PodSecurityPolicy* und *NetworkPolicy*), dies ist empfehlenswert. Kubernetes besitzt ein Konzept namens “daemon sets”: dies sind Dienste, die automatisch auf allen Ndoes installiert werden (z. B. Monitoring oder IDS). Diese Funktion ist natürlich auch für Angreifer interessant.

16. Cloud-Native Applications

Cloud-Native Applications sind Applikationen, welche von Grund auf für den Betrieb in der Cloud ausgelegt wurden. Sie bestehen aus Servcies, die per Container verpackt, auf elastischer Infrastruktur, mittels DevOps-Prozessen ausgerollt werden. Sie verwenden Orchestration-Software um eine Isolierung von der zugrunde liegenden Hardware bzw. den verwendeten Cloud-Providern sicherzustellen. Architektonisch werden Funktionen häufig als Micro-Services ausgeführt, es gbit eine klare Separierung zwischen state-full und state-less services.

16.1 Microservices

Bei Microservices wird eine Applikation auf viele einzelne Services aufgeteilt. Jeder Service sollte nur genau eine Aufgabe lösen und dies über ein wohldefiniertes technologie-neutrales Netzwerkinterface, wie z. B. einem HTTP REST Interface, anbieten. Dadurch wird (theoretisch) die Produktentwicklung entkoppelt, einzelne Services können getrennt voneinander entwickelt und gewartet werden. Da ein neutrales Kommunikationsformat verwendet wird, können einzelne Microservices in getrennten Programmiersprachen und Frameworks implementiert werden.

Aus Sicherheitssicht ist die verbesserte Isolation von Geschäftsfunktionen (eine Funktion pro Service) und die bessere Testbarkeit vorteilhaft. Durch den Fokus auf einen minimalen Funktionsumfang pro Service werden hier auch verstärkt Lösungen wie zentralisiertes Logging oder SSO verwendet.

16.2 Service-Meshes

Laut Heise sollte 2019 das Jahr der Service Meshes werden…

Service Meshes sind eine Technologie die häufig mit Microservices kombiniert werden. Ein Microservice sollte genau eine Geschäftsfunktion implementieren und sich sonst mit wenig anderem beschäftigen. Bei einem klassichen Service würde dies durch Verwendung eines Frameworks geschehen — dies widerspricht allerdings dem technologie-agnostischen Ansatz von Microservices. Die Alternative ist es, diese Funktionen quasi aus dem Service zu ziehen und durch die Cloud-Umgebung bereitzustellen. Dies kann z. B. durch einen vorgelagerten Proxy geschehen, der die

gesammte Kommunikation zum Microservice durchschleust, dabei aber Funktionen wie Tracing, Verschlüsselung, Logging und Authentifikation durchführt.

Historisch kann *kube-proxy* (Bestandteil von Kubernetes) als erste verbreitete Service Mesh Lösung gesehen werden, mittlerweile gibt es mit *istio* und *linkerd* weitere Implementierungen. Um Service-Mesh Proxies besser von klassichen Proxies abzugrenzen, werden diese zumeist *Sidecar Proxies* genannt. Durch das Service Mesh kommnizieren Microservices nicht mehr direkt miteinander, sondern die Proxies kommunizieren in Vertretung der Microservices.

Welche Aufgaben können durch die Service Meshes umgesetzt:

- Verbindungssicherheit
- Authentication/Authorization
- Logging und Tracing
- Metriken
- Load-Balancing, Circuit-Breaking, ...

IV Mobile Betriebssysteme

Während die Geschwindigkeit der Entwicklung sich bei Server- und Desktopsystemen leicht verlangsamt hatte, gab es eine Vielzahl an Innovationen im Umfeld mobiler Betriebssyteme seit der Einführung des Apple iPhone und des initialen Google Android Systems. Eventuell ist das Wort Innovation hier falsch verwendet, aber viele Konzepte die bereits in Nischen auf konventionellen Geräten umgesetzt wurden, wurden durch SmartPhones massentauglich bzw. wurden diese durch SmartPhones zum ersten Mal von der breiten Masse der Benutzer erkannt und für "praktisch" befunden.

Im Zuge der Vorlesung betrachten wir zumeist moderne iOS (12 aufwärts) und Android (9.0 aufwärts) Geräte.

Mobilegeräte bzw. mobile Betriebssysteme besitzen eine andere Ausgangslage als klassische Systeme. Der Hersteller des Geräts kann stärker auf die vorhandene Hardware Einfluss nehmen. Im Apple iPhones Fall kontrolliert der Hersteller die Hardware direkt, im Android-Fall sind Software-Features bzw. Android-Compliance mit Hardware-Requirements verbunden. Microsoft versuchte zwar ähnliches mit Windows, war damit allerdings vergleichsweise weniger erfolgreich.

Ebenso werden Mobilgeräte von Anwendern nicht als die generischen Computer, die sie sind, gesehen. Die Hersteller können Benutzer weitaus stärker einschränken als diese es bei einem Desktop akzeptieren würden. Dies erlaubt es, einige Sicherheitsoptionen zu implementieren, die sie auf einem Desktop nicht erlauben würden. So sind z. B. die meisten Benutzer nicht Administratoren ihres Telefones und können auch nicht beliebige Software installieren — Möglichkeiten, auf die sie bei einem Desktop tendentiell bestehen würden.

Folgende Elemente sind bei Mobiltelefonen umgesetzt:

- Abgesicherter Bootvorgang
- Festplattenverschlüsselung
- Bezug von Software aus einer zentralen Quelle samt teilweiser automatisierter Verifikation auf Schadcode
- Verwendung von Security-Containern pro Applikation
- Feingranulare Capabilities pro Applikation bzw. Applikations-Container
- Deklarative Erzwingung der Verwendung sicherer Netzwerkverbindungen für Applikationen
- Besondere Vorkehrungen zum Speichern von sensiblen Daten wie Credentials (inkl. in-memory protection)
- Strikte Trennung von Admin (root) und normalen Benutzern. Auf Mobilgeräten ist es nicht per-default möglich, root-Benutzer zu werden.
- Verfügbarkeit von Spezialhardware wie Fingerprintsensoren und TPM-Chips
- Zentrales Management der Device-Konfiguration bzw. der installierten Software

Im Zuge dieses Kapitels betrachten wir die Umsetzung dieser Konzepte unter Android und iOS.

17. Apple iOS

- no third-party app store
- besserer Verbund von Hard- und Software (siehe in-memory secret-encryption)

18. Google Android

- multiple third-party app stores
- verwendet seccomp-bpf
- Problem mit Systemupdates einiger Hersteller
- dm_verity

19. MDM and MEP

V Abschließende Worte

History never repeats..

- ISOLATION/CONTAINMENT zwischen Ebenen bzw. Elementen einer Ebene (z. B. Server, Container, Prozesse, etc.)
- TRUST — woher kommt meine Software, Container, Pakete
- AWARENESS — was sind die Möglichkeiten eines Angreifers