ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΑΥΤΟΜΑΤΗ ΕΝΙΣΧΥΣΗ ΑΣΦΑΛΕΙΑΣ ΣΕ ΠΕΡΙΒΑΛΛΟΝ DOCKER ΜΕΣΩ ΣΥΣΤΗΜΑΤΟΣ ΥΠΟΧΡΕΩΤΙΚΟΥ ΕΛΕΓΧΟΥ (MAC)

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Φώτιος Λουκίδης-Ανδρέου

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Απρίλιος 2017

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΑΥΤΟΜΑΤΗ ΕΝΙΣΧΥΣΗ ΑΣΦΑΛΕΙΑΣ ΣΕ ΠΕΡΙΒΑΛΛΟΝ DOCKER ΜΕΣΩ ΣΥΣΤΗΜΑΤΟΣ ΥΠΟΧΡΕΩΤΙΚΟΥ ΕΛΕΓΧΟΥ (MAC)

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Φώτιος Λουκίδης-Ανδρέου

Επιβλέπων : Νεκτάριος Κοζύρης

Καθηγητής Ε.Μ.Π

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 3η Απριλίου 2017.

.....	.....	.....
Ν. Κοζύρης	Γ. Γκούμας	Ν. Παπασπύρου
Καθηγητής Ε.Μ.Π	Επ. Καθηγητής Ε.Μ.Π	Αν. Καθηγητής Ι.Π

Αθήνα, Απρίλιος 2017

……

Φώτιος Λουκίδης-Ανδρέου

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π

ΠΕΡΙΛΗΨΗ

Η εικονοποίηση σε επίπεδο λειτουργικού συστήματος (operating-system-level virtualization), γνωστή και ως containers, είναι μία συνεχώς ανερχόμενη τεχνολογία, η οποία δίνει τη δυνατότητα εκτέλεσης πολλών εφαρμογών ταυτόχρονα, η κάθε μία από τις οποίες βρίσκεται σε απομονωμένο περιβάλλον, ενώ παράλληλα, η επιπλέον κατανάλωση υπολογιστικών πόρων κρατιέται στο ελάχιστο μέσω της χρήσης του πυρήνα του host λειτουργικού συστήματος. Αυτή τη στιγμή, η πιο διαδομένη πλατφόρμα για την κατασκευή, την εκτέλεση και τον έλεγχο των containers είναι το Docker, λόγω των ευκολιών που προσφέρει στην κατασκευή και στο διαμοιρασμό των containers. Ωστόσο παρά τις ευκολίες και τα οφέλη που παρέχουν αυτές οι λύσεις, παρατηρείται αργή υιοθέτηση αυτής της τεχνολογίας, η οποία οφείλεται σε μεγάλο βαθμό σε ανησυχίες που υπάρχουν σχετικά με την ασφάλεια [1]. Στην εργασία αυτή μελετάμε τους κινδύνους που εμφανίζονται σε περιβάλλοντα εικονοποίησης και πώς αυτοί μπορούν να αντιμετωπιστούν. Εξετάζουμε τα εργαλεία που προσφέρουν τα Linux και πώς μπορούμε μέσα από αυτά να δημιουργήσουμε ένα ασφαλές απομονωμένο περιβάλλον εκτέλεσης διεργασιών. Με γνώμονα τις δυνατότητες που μας δίνει ο πυρήνας των Linux και τις απειλές που αντιμετωπίζει ένας Docker container δημιουργήσαμε ένα εργαλείο που παράγει αυτόματα Apparmor προφίλ, προστατεύοντας αποτελεσματικά και δυναμικά το host σύστημα. Τέλος, αξιολογούμε την επιβάρυνση στις επιδόσεις που επιφέρει αυτό το σύστημα ασφάλειας στους containers.

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ

Linux containers, εικονοποίηση σε επίπεδο λειτουργικού συστήματος, Docker, namespaces, cgroups, chroot, Mandatory Access Control (MAC), Apparmor, αυτοματοποίηση

ABSTRACT

Operating-system-level virtualization, commonly referred to as containers, is a continuously developing technology, which allows the concurrent execution of many processes, each of which is run in an isolated virtualized environment, while the overhead of the virtualization is kept minimal, by allowing each process to use the host kernel. At the time of writing the most used platform for container life-cycle management is Docker due to the features and easiness of use that it provides. Despite the benefits provided by operating-system-level virtualization, the adoption of this technology is quite slow mainly due to security related considerations [1]. In this diploma thesis we try to evaluate the dangers that are present in such an environment and means with which they can be minimized or even confronted. We examine carefully the tools provided by Linux operating system and how they can be used to create a secure containerized environment. By taking into account the dangers involved in a Docker environment and the tools mentioned above, we design a tool that automates the process of creating Apparmor profiles for Docker containers, so that host system can be efficiently protected during the creation and the runtime of containers. Finally, we evaluate the designed solution both from security and performance perspective.

KEY WORDS

Linux containers, operating system level virtualization, Docker, namespaces, cgroups, chroot, Mandatory Access Control (MAC), Apparmor, automation

Ευχαριστίες

Αρχικά ευχαριστώ ιδιαιτέρως τον επιβλέποντα καθηγητή Νεκτάριο Κοζύρη που μου έδωσε την ευκαιρία να ασχοληθώ με ένα τόσο ενδιαφέρον αντικείμενο. Επίσης θα ήθελα να ευχαριστήσω ιδιαίτερα την Κατερίνα Δόκα και τον Γιάννη Γιαννακόπουλο για την άριστη συνεργασία που είχαμε, την κατανόηση που έδειξαν, την βοήθεια τους σε τεχνικά ζητήματα, αλλά και τις ιδέες τους σχετικά με την εργασία. Τέλος θα ήθελα να ευχαριστήσω την οικογένεια μου για την αμέριστη συμπαράστασή της.

Φώτιος Λουκίδης – Ανδρέου,

Αθήνα, 3 Απριλίου 2017

Περιεχόμενα

1 ΕΙΣΑΓΩΓΗ

1.1 Κίνητρο

Η εικονοποίηση επιπέδου λειτουργικού συστήματος, γνωστή και ως containers ή software containers [2], είναι μία τεχνολογία που τα τελευταία χρόνια αναπτύσσεται ραγδαία. Οι containers έχουν μετατραπεί σε μία production-ready τεχνολογία, και μπορούμε να τους δούμε να χρησιμοποιούνται ήδη σε πολλές εφαρμογές και να υποστηρίζονται από πολλούς οργανισμούς και εταιρίες. Χαρακτηριστικό παράδειγμα αποτελεί η Google η οποία κλείνει ήδη τα 10 χρόνια χρήσης containers [3]. Από το 2014 δήλωσε ότι χρησιμοποιούσε containers για όλες τις υπηρεσίες που προσφέρει (από αναζήτηση ιστοσελίδων μέχρι και υπηρεσίες ηλεκτρονικού ταχυδρομείου) ξεκινώντας περισσότερο από δύο δισεκατομμύρια containers την εβδομάδα [4]. Ένα άλλο χαρακτηριστικό παράδειγμα αποτελεί η eBay, με εξίσου μεγάλες ανάγκες για scalability και για κατανεμημένη εκτέλεση των απαιτούμενων εργασιών, η οποία χρησιμοποιεί Kubernetes (το Kubernetes είναι σύστημα οργάνωσης και δρομολόγησης containers) για να εξυπηρετήσει τις ανάγκες της [5], καθώς και το box, ο οργανισμός Wikemedia Foundation (που συντηρεί την Wikipedia) και το Spotify. Τέλος, παραδείγματα εταιριών που επενδύουν σε τεχνολογίες εικονοποίησης σε επίπεδο λειτουργικού συστήματος είναι η CloudFoundry, η Amazon Web Services, η Canonical, η Rackspace και η Openshift.

Το ενδιαφέρον για τους Linux containers συνεχώς αυξάνεται, λόγω της δυνατότητας που παρέχουν στους χρήστες να τρέχουν τις εφαρμογές τους σε ένα απομονωμένο περιβάλλον, κρατώντας το overhead της εικονοποίησης στο ελάχιστο. Αυτό επιτυγχάνεται καθώς οι διεργασίες του εικονοποιημένου περιβάλλοντος χρησιμοποιούν τον πυρήνα του host συστήματος για να εκτελέσουν τις απαραίτητες κλήσεις συστήματος, χωρίς να απαιτείται κάποιο ενδιάμεσο στρώμα προσομοίωσης συσκευών, όπως συμβαίνει με τις συμβατικές μεθόδους εικονοποίησης. Το παραπάνω πλεονέκτημα σε συνδυασμό με το ότι επιτρέπουν εύκολο “πακετάρισμα” και διανομή των εφαρμογών κάνει τους containers κατάλληλους για σύγχρονες αρχιτεκτονικές τύπου PaaS ή microservices [6] και για μοντέλα ανάπτυξης εφαρμογών τύπου DevOps και Continuous Integration / Continuous Delivery (CI / CD). Συνεπώς, οι containers μπορούν να χρησιμοποιηθούν για απαιτητικές, κατανεμημένες, αξιόπιστες και highly available εφαρμογές [7].

Ωστόσο, παρά τις δυνατότητες που προσφέρει η χρήση εικονοποίησης σε επίπεδο λειτουργικού συστήματος η υιοθέτηση τους σε κάποιες περιπτώσεις είναι αργή. Σε αυτό οφείλεται κυρίως η ανησυχία που εκφράζουν αρκετοί για την ασφάλεια των containers [8], καθώς αφενός οι containers είναι μία νέα τεχνολογία και συνεπώς δεν έχουν δημιουργηθεί ακόμα αρκετά ισχυρές πρακτικές για την ασφάλιση των συστημάτων αυτών και αφετέρου στο ότι οι χρήστες δεν έχουν εξοικειωθεί ακόμη με τη χρήση τους. Σε αυτό το σημείο αξίζει να σημειωθεί ότι παρόλο που οι containers προσφέρουν δυνατούς μηχανισμούς για απομόνωση και προστασία, είναι δυνατόν να μεγαλώσει το attack surface με τη χρήση μίας πλατφόρμας για containers όπως το Docker, εάν αυτή δεν χρησιμοποιηθεί σωστά και δεν γίνουν οι απαραίτητες ρυθμίσεις για τον θωρακισμό της. Αντίθετα, όμως, οι εικονικές μηχανές (όπως το KVM, το

VirtualBox και το VMWare) εμφανίστηκαν πολύ πριν την εικονοποίηση επιπέδου λειτουργικού συστήματος και έχουν χρησιμοποιηθεί εκτενώς. Αυτό συνεπάγεται ότι έχουν δημιουργηθεί ώριμα και αξιόπιστα εργαλεία με αντοχή σε διάφορες επιθέσεις, καθώς και το ότι υπάρχει πολύ καλή υποστήριξη και γνώση για αυτά τα εργαλεία. Για αυτό το λόγο πολλοί προτιμούν ακόμη να χρησιμοποιούν τις παραδοσιακές εικονικές μηχανές, έως ότου βελτιωθεί η τεχνογνωσία και η τεχνολογία των Linux containers.

Στην εργασία αυτή θα προσπαθήσουμε να αναλύσουμε τους μηχανισμούς ασφάλειας των containers και να αξιολογήσουμε το επίπεδο απομόνωσης που προσφέρουν. Στη συνέχεια θα εξετάσουμε τις αδυναμίες που μπορούν να εμφανιστούν σε ένα περιβάλλον εικονοποίησης επιπέδου λειτουργικού συστήματος και κατά πόσο ένας container μπορεί να περιορίσει μία compromised εφαρμογή ώστε να μην προχωρήσει η επίθεση στο host μηχάνημα ή στους άλλους containers. Επίσης θα μελετήσουμε τις αδυναμίες που μπορεί να επέλθουν χρησιμοποιώντας μία πλατφόρμα για Linux containers όπως το Docker, εάν αυτή δεν ρυθμιστεί σωστά. Έπειτα θα δούμε πώς μπορούμε χρησιμοποιώντας τα εργαλεία του πυρήνα των Linux και το Apparmor, ένα εύχρηστο σύστημα υποχρεωτικού ελέγχου (Mandatory Access Control ή MAC), να δημιουργήσουμε ένα αυτοματοποιημένο σύστημα ασφάλειας για την πλατφόρμα του Docker. Θα δούμε πως μπορούμε να εξελίξουμε το εργαλείο αυτό ώστε να μπορεί να τροποποιεί τα profile του Apparmor ανάλογα με τις ανάγκες της εφαρμογής, ώστε να δίνει όσο το δυνατόν λιγότερα δικαιώματα. Τέλος θα αξιολογήσουμε αυτό το σύστημα, ώστε να διαπιστώσουμε το κατά πόσο επιβαρύνει την εκτέλεση των εφαρμογών που βρίσκονται μέσα στους containers.

1.2 Τύποι εικονοποίησης

Πριν προχωρήσουμε αξίζει να συγκρίνουμε την εικονοποίηση επιπέδου λειτουργικού συστήματος με την εικονοποίηση υλικού (hardware virtualization), γνωστή και ως εικονοποίηση πλατφόρμας (platform virtualization), η οποία επιτρέπει την κατασκευή εικονικών μηχανών ικανών να συμπεριφέρονται ως ένας κανονικός υπολογιστής, ανεξάρτητος από το host λειτουργικό σύστημα προσομοιώνοντας το φυσικό μηχάνημα με τη χρήση software. Σε αυτή την κατηγορία εικονοποίησης μπορούμε να διακρίνουμε δύο βασικές υποκατηγορίες ανάλογα με τον τύπο της εικονικής μηχανής που παράγεται, την πλήρη εικονοποίηση και την παραεικονοποίηση. Επίσης ανάλογα με λογισμικό ελέγχου (hypervisor) της εικονικής μηχανής μπορούμε να διακρίνουμε τους επόπτες τύπου-1 (bare-metal ή native hypervisors) και τους επόπτες τύπου-2 (hosted hypervisors). Τέλος η εικονοποίηση μπορεί να είναι υποβοηθούμενη από το hardware ή όχι.

1.2.1 Τύποι εικονοποίησης υλικού (Hardware virtualization)

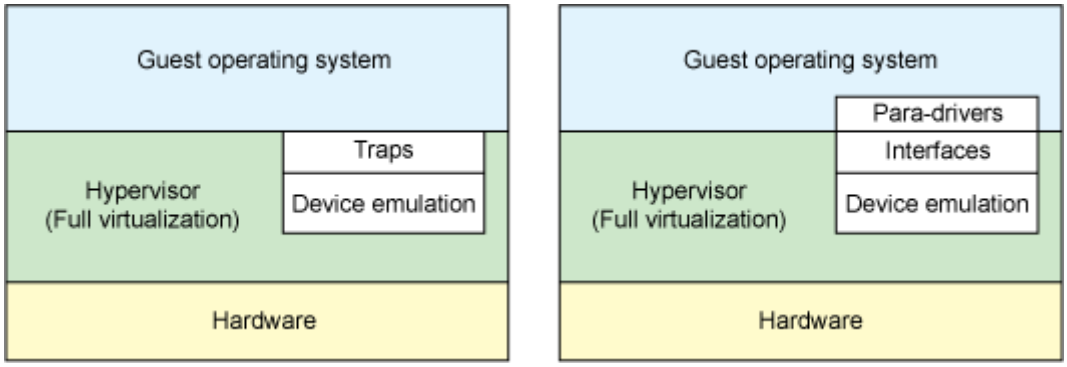
1.2.1.1 Πλήρης εικονοποίηση (Full virtualization)

Οι εικονικές μηχανές που βασίζονται στην πλήρη εικονοποίηση, μπορούν να προσομοιώσουν ένα περιβάλλον (σχεδόν) πανομοιότυπο με το φυσικό μηχάνημα και ανεξάρτητο του host λειτουργικού συστήματος. Με αυτό τον τρόπο η εικονική μηχανή μπορεί να περιέχει διαφορετικό φιλοξενούμενο (guest) λειτουργικό σύστημα από το host, αρκεί να υποστηρίζεται από το φυσικό μηχάνημα. Παρ' όλα αυτά, ειδικά στην περίπτωση τη hardware υποβοηθούμενης εικονοποίησης (που αυτή τη στιγμή υποστηρίζεται από τις περισσότερες οικογένειες επεξεργαστών) η επίδοση μπορεί να προσεγγίζει κατά πολύ την bare-metal επίδοση, ωστόσο παραμένει πιο αργή από την παραεικονοποιήση και την OS-level εικονοποίηση και, συνεπώς, λιγότερο αποτελεσματική ως προς την χρησιμοποίηση των διαθέσιμων υπολογιστικών πόρων και πιο ενεργοβόρα.

Η πλήρης εικονοποίηση, ακριβώς επειδή προσομοιώνει όλο ή μεγάλο τμήμα του υλικού παρέχει την ισχυρότερη μορφή απομόνωσης και ασφάλειας σε σχέση με τους υπόλοιπους τύπους εικονοποίησης. Η διαφυγή από ένα τέτοιο εικονοποιημένο περιβάλλον θεωρείται πολύ δύσκολη, καθώς η επικοινωνία του φιλοξενούμενου λειτουργικού συστήματος με το host μηχάνημα είναι πολύ μικρή (αν και διαφέρει ανάλογα με την υλοποίηση και τον τύπο του επόπτη της εικονικής μηχανής [9]). Ωστόσο κατά καιρούς έχουν εμφανιστεί αρκετές αδυναμίες σε διάφορες πλατφόρμες εικονοποίησης, κυρίως όμως στον τομέα των εικονοποιημένων driver [10]. Για παράδειγμα στο VMWare Workstation, μπορούσε κανείς να ξεφύγει από το guest λειτουργικό και να εκτελέσει αυθαίρετο κώδικα (arbitrary code execution) στο host μηχάνημα (CVE-2009-1244) ή απομακρυσμένοι (αλλά authenticated) χρήστες, εκμεταλλευόμενοι αδυναμία του NAT συστήματος του VMware, να εκτελέσουν αυθαίρετο κώδικα μέσω EPRT και PORT FTP εντολών (CVE-2005-4453). Παρόμοιες αδυναμίες έχουν βρεθεί και στα συστήματα Xen που προκαλούν Denial Of Service (συνήθως freezes ή restarts), ενώ στο Hyper-V υπάρχει τουλάχιστον ένα γνωστό escape από το guest σύστημα [11]. Παρ' όλα αυτά, σε γενικές γραμμές, η πλήρης εικονοποίηση παραμένει η πιο ασφαλής λύση, καθώς είναι μια δοκιμασμένη τεχνολογία, με πολύ καλή υποστήριξη για πληθώρα εικονικών συσκευών.

1.2.1.2 Παραεικονοποίηση (Paravirtualization)

Η παραεικονοποίηση προσφέρει ένα περιβάλλον εκτέλεσης παρόμοιο, αλλά όχι ίδιο με το φυσικό μηχάνημα με στόχο την επίτευξη καλύτερων επιδόσεων. Προκειμένου, να μην προσομοιώνονται απαιτητικές διαδικασίες σε ένα εικονικό περιβάλλον, δημιουργούνται οι κατάλληλοι οδηγοί συσκευών (drivers) και οι απαραίτητες διεπαφές (interfaces), ώστε να εκτελούνται απευθείας στο host μηχάνημα (βλ. Σχήμα 1). Ορίζονται hooks, όπου το guest και το host λειτουργικό μπορούν να αναγνωρίζουν και να περιμένουν αυτές τις διαδικασίες. Κατά αυτόν τον τρόπο οι εικονικές μηχανές απλοποιούνται, αφού δεν χρειάζεται να υλοποιηθεί εικονοποίηση για διαδικασίες που αναλαμβάνει ο host μέσω του paravirtualization, όπως θα απαιτούνταν στην περίπτωση της πλήρους εικονοποίησης.



Εικόνα 1: Εικονοποίηση συσκευής σε περιβάλλον πλήρους εικονοποίησης (αριστερά) και σε περιβάλλον παραεικονοποίησης (δεξιά)

Όμως, ως συνέπεια των παραπάνω, το φιλοξενούμενο λειτουργικό σύστημα δεν μπορεί να εκτελείται όπως σε ένα φυσικό μηχάνημα, αλλά θα πρέπει να είναι γίνουν οι απαραίτητες τροποποιήσεις στον πυρήνα του ώστε να υποστηρίξει τα APIs (διασύνδεση υπερκλήσεων ή hypercalls) του παραεικονοποιημένου περιβάλλοντος. Αυτό συνεπάγεται ότι ένας τέτοιος custom πυρήνας, μπορεί να μένει πίσω σε σχέση με τους mainline πυρήνες, καθυστερώντας έτσι σημαντικές αναβαθμίσεις ασφάλειας. Επίσης, εφόσον ο πυρήνας γίνεται custom και επειδή αυξάνεται η επικοινωνία του guest με το host σύστημα, αυξάνονται οι κίνδυνοι διαφυγής από την εικονική μηχανή.

Αυτή η μορφή εικονοποίησης υποστηρίζεται από πολλές πλατφόρμες με κύριο εκπρόσωπο το αρκετά δοκιμασμένο στην παραεικοποίηση Xen, το οποίο βασίζει μεγάλο μέρος του κώδικά του στο project QEMU. Επίσης το KVM παρέχει υποστήριξη για αρκετές συσκευές για διάφορα guest λειτουργικά συστήματα (όπως Linux, Windows, FreeBSD και OpenBSD) χρησιμοποιώντας το VirtIO API, το οποίο παρέχει υψηλές επιδόσεις IO (input output operations) σε δίκτυο και σε δίσκο (και γενικά σε block devices), που προσεγγίζουν σε ορισμένες περιπτώσεις την bare-metal επίδοση και ξεπερνούν κατά πολύ τις επιδόσεις της πλήρους εικονοποίησης στο IO [12] [13].

1.2.2 Τύποι λογισμικών ελέγχου εικονικών μηχανών (Hypervisors)

Συνήθως τα λογισμικά εποπτείας εικονικών μηχανών διαχωρίζονται σε δύο κατηγορίες [14], αν και δεν είναι πάντα εύκολος ο σαφής διαχωρισμός:

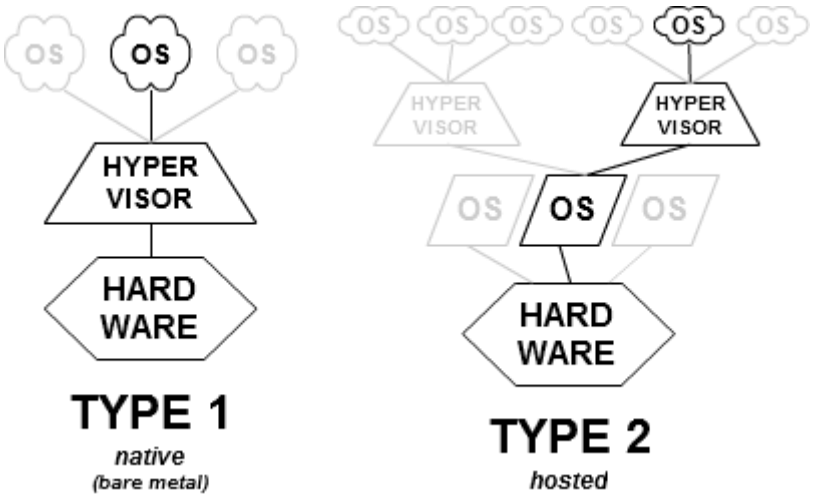
Επόπτες Τύπου-2 (Type-2 ή Hosted Hypervisors)

Οι επόπτες αυτού του επιπέδου τρέχουν πάνω από κάποιο λειτουργικό σύστημα γενικού σκοπού, σαν ένα συνηθισμένο πρόγραμμα σε χώρο χρήστη (userspace), παρέχοντας τη δυνατότητα εκτέλεσης πολλών εικονικών μηχανών σε ένα host φυσικό μηχάνημα. Δεν απαιτείται η ύπαρξη κάποιου ειδικού λειτουργικού συστήματος ώστε να μπορούν να παρέχουν τις απαραίτητες υπηρεσίες για εικονοποίηση. Παρόλο που οι επόπτες δεύτερου τύπου είναι αρκετά ευέλικτοι και εύχρηστοι, συνήθως έχουν

μειωμένες επιδόσεις, καθώς ο κώδικας που απαιτείται για την προσομοίωση κάποιων λειτουργιών του φιλοξενούμενου λειτουργικού συστήματος τρέχει σε χώρο χρήστη, το οποίο σημαίνει ότι πρέπει να γίνουν αρκετές κλήσεις συστήματος μέχρις ότου ολοκληρωθούν. Ένα άλλο μειονέκτημα που εμφανίζουν οι hypervisors αυτού του τύπου, είναι η μειωμένη ασφάλεια. Αυτό συμβαίνει καθώς η ασφάλεια και η αξιοπιστία του συστήματος εξαρτάται όχι μόνο από την υλοποίηση του λογισμικού ελέγχου της εικονικής μηχανής, αλλά και από την υλοποίηση και τη σταθερότητα του host λειτουργικού συστήματος.

Επόπτες Τύπου-1 (Type-1, Native ή Bare-Metal Hypervisors)

Οι επόπτες αυτής της κατηγορίας τρέχουν απευθείας πάνω στο φυσικό μηχάνημα χωρίς να παρεμβάλλεται κάποιο λειτουργικό σύστημα, σε αντίθεση με τους επόπτες τύπου 2 (βλ. εικόνα 2). Συνήθως μαζί με τον hypervisor υπάρχει ένα ενσωματωμένο λειτουργικό σύστημα, βελτιστοποιημένο, ώστε να υποβοηθά την εικονοποίηση, το οποίο παραμένει διαφανές για τον τελικό χρήστη. Με αυτόν τον τρόπο επιτυγχάνεται σαφώς βελτιωμένη επίδοση, ενώ ταυτόχρονα μειώνεται το attack surface και άρα ενισχύεται η ασφάλεια του συστήματος. Αξίζει να σημειωθεί ότι η πρώτη εικονοποίηση χρησιμοποιούσε hypervisors αυτού του τύπου και είχε σχεδιαστεί από την IBM στη δεκαετία του 1960 [15].

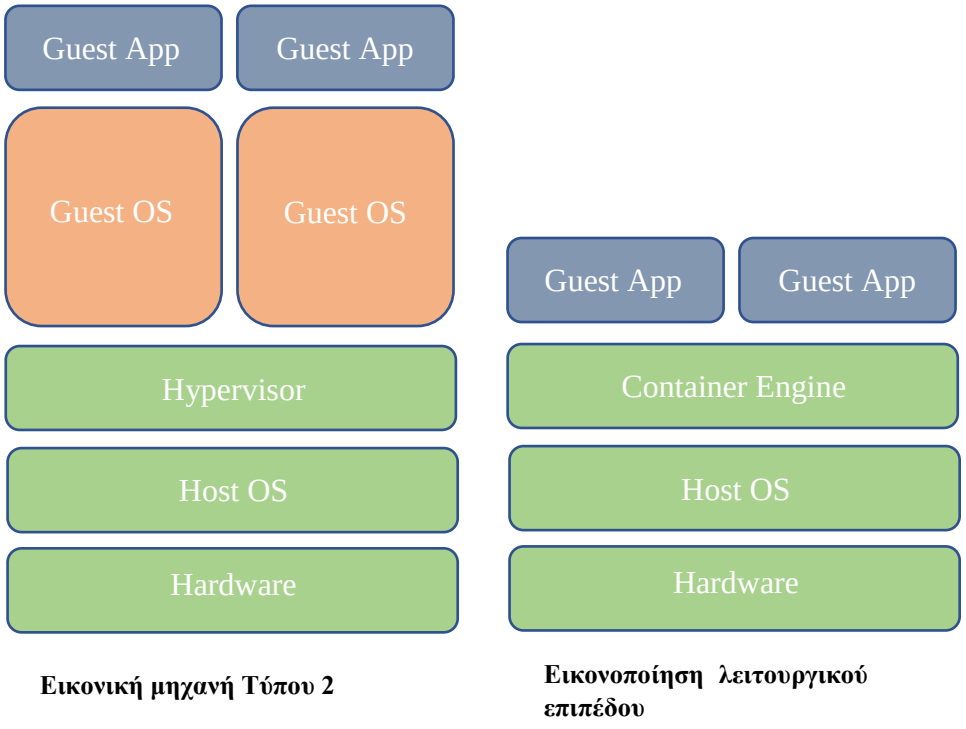


Εικόνα 2: Επόπτες Τύπου-1 και Τύπου-2

Τα τελευταία χρόνια έχει εμφανιστεί η έννοια του επόπτη 0, ο οποίος από πολλούς θεωρείται ως υποπερίπτωση του bare-metal επόπτη. Σε αυτήν την κατηγορία ο επόπτης είναι ακόμη πιο κοντά στο υλικό. Εδώ απουσιάζει το ενσωματωμένο λειτουργικό σύστημα που συνοδεύει έναν επόπτη τύπου 1, ενώ ο επόπτης αποτελείται από τα απολύτως απαραίτητα για την εικονοποίηση τμήματα λογισμικού. Κατ' αυτόν τον τρόπο οι επιδόσεις, η ασφάλεια και η αξιοπιστία του συστήματος είναι ακόμη βελτιωμένες σε σχέση με τους hypervisors των άλλων δύο τύπων.

1.2.3 Εικονοποίηση Επιπέδου λειτουργικού συστήματος

Η εικονοποίηση σε επίπεδο λειτουργικού συστήματος (operating system-level virtualization), είναι μία μέθοδος εικονοποίησης, όπου δεν χρειάζεται προσομοίωση των λειτουργιών κάποιων συσκευών του υπολογιστή, επειδή το host λειτουργικό μοιράζεται τον πυρήνα του με το φιλοξενούμενο σύστημα, γεγονός που συνεπάγεται αυξημένες επιδόσεις, αλλά μεγαλύτερους κινδύνους στην ασφάλεια του συστήματος. Προκειμένου να υποστηριχτεί αυτή η μορφή εικονοποίησης, το host λειτουργικό σύστημα παρέχει ορισμένες λειτουργίες οι οποίες επιτρέπουν σε διεργασίες που τρέχουν σε χώρο χρήστη (user-space) να βλέπουν το δικό τους απομονωμένο περιβάλλον, ενώ παράλληλα χρησιμοποιούν τον ίδιο πυρήνα με τον host. Αυτή η μέθοδος είναι γνωστή ως containers στα Linux, Zones στα συστήματα Solaris, Jails στα FreeBSD, αλλά εμφανίζεται και με άλλες μορφές όπως τα software containers, τα sandboxes και τα chroot jails. Στην εργασία αυτή θα εστιάσουμε στους containers των Linux.



Εικόνα 3: Εικονική μηχανή και Container

Η τεχνολογία αυτή προσφέρει πολλά πλεονεκτήματα, ανάλογα με την χρήση και την υλοποίηση της. Αρχικά προσφέρει πολύ καλές επιδόσεις και έχει μικρή έως καθόλου επιβάρυνση στον χρόνο εκτέλεσης της εφαρμογής, καθώς δεν απαιτείται προσομοίωση ή μετάφραση κώδικα από guest σε host σύστημα, μιας και οι εφαρμογές του φιλοξενούμενου περιβάλλοντος χρησιμοποιούν τον πυρήνα του host (βλ. εικόνα 3). Επίσης, αυτού του τύπου η εικονοποίηση είναι ανεξάρτητη από το φυσικό μηχάνημα και δεν χρειάζεται κάποια υποστήριξη από το υλικό, καθώς υλοποιείται απευθείας σε επίπεδο λογισμικού από το λειτουργικό σύστημα. Σημαντικό είναι το γεγονός ότι οι containers καταναλώνουν σαφώς λιγότερο χώρο στο δίσκο, αφού δεν χρειάζεται να υπάρχει κάποιο επιπλέον λειτουργικό σύστημα για να είναι λειτουργικοί.

Μεγαλύτερη, ακόμη, εξοικονόμηση χώρου μπορεί να επιτευχθεί στην περίπτωση χρήσης τεχνολογίας Copy-on-Write (CoW) σε συνδυασμό με κάποιο Union filesystem, όπως το AuFS. Αυτή η δυνατότητα προσφέρεται πλέον από πολλούς containers, καθώς με αυτόν τον τρόπο εάν απαιτείται η εκτέλεση του ίδιου container ή της ίδιας εφαρμογές πολλές φορές ταυτόχρονα, ο χρήστης χρειάζεται να έχει αποθηκεμένο στον σκληρό του μόνο ένα αντίγραφο. Τέλος, οι containers επιτρέπουν το πακετάρισμα των εφαρμογών και των κατάλληλων ρυθμίσεων έτσι ώστε να είναι μεταφέρσιμα και έτοιμα να τρέξουν σε ένα νέο περιβάλλον με ελάχιστες έως καθόλου επιπλέον παραμετροποιήσεις, γεγονός που τους καθιστά εύκολα μεταφέρσιμους και κατ' επέκταση κατάλληλους για χρήση σε scalable συστήματα και σε cloud.

Ωστόσο, αυτά τα χαρακτηριστικά έρχονται με κάποια κόστη. Το σημαντικότερο, ίσως, είναι ότι αδυνατούν να παρέχουν τόσο ισχυρή απομόνωση όσο μία εικονική μηχανή και συνεπώς το host μηχάνημα είναι πιο ευάλωτο σε επιθέσεις από υπονομευμένες guest εφαρμογές. Και αυτό, διότι, αφενός οι διεργασίες που τρέχουν μέσα στους containers έχουν πρόσβαση σε όλες τις κλήσεις συστήματος του host, ενώ σε ένα VM όχι, και αφετέρου, διότι σε πολλές περιπτώσεις μοιράζονται τους ίδιους πόρους με τον host. Ένας άλλος περιορισμός που αντιμετωπίζουν οι containers, είναι το γεγονός ότι πολλά features που κάνουν τους containers ασφαλείς, έχουν προστεθεί πρόσφατα σχετικά στον πυρήνα των Linux, το οποίο σημαίνει ότι οι containers δεν είναι συμβατοί με παλιότερους πυρήνες. Επίσης πολλοί πόροι του λειτουργικού συστήματος δεν μπορούν να προστατευτούν ακόμα, καθώς τα λειτουργικά συστήματα, κατά κύριο λόγο, δεν έχουν υλοποιηθεί με τους απαραίτητους μηχανισμούς για να μπορούν να προσφέρουν την απαραίτητη απομόνωση. Χαρακτηριστικό παράδειγμα αυτού του προβλήματος αποτελεί η υλοποίηση στον πυρήνα των Linux των user namespaces, η υλοποίηση των οποίων ξεκίνησε στον πυρήνα 2.6.23 (γύρω στο 2006 [16]), ολοκληρώθηκε το μεγαλύτερο μέρος στην έκδοση 3.8 του πυρήνα [17] (Φεβρουάριος 2013). Ένας άλλος περιορισμός που επιβάλει η χρήση των container είναι το ότι το φιλοξενούμενο σύστημα δεν μπορεί να έχει διαφορετικό λειτουργικό σύστημα από το host, από τη στιγμή που μοιράζονται τον ίδιο πυρήνα, καθιστώντας το έτσι σαφώς λιγότερο ελαστική επιλογή σε σχέση με τις εικονικές μηχανές (ωστόσο υπάρχουν containers που επιτρέπουν την εκτέλεση λειτουργικού άλλης έκδοσης, αλλά της ίδιας ή συγγενούς οικογένειας λειτουργικών, όπως στα Solaris και τα Illumos). Τέλος, δεν υπάρχει αυτή τη στιγμή η δυνατότητα της ζωντανής μεταφοράς (live migration) ενός container που εκτελείται, όπως προσφέρουν αρκετά συστήματα εικονικών μηχανών (για παράδειγμα το OpenVZ και το VMware), αν και γίνονται αρκετές προσπάθειες προς την κατεύθυνση αυτή.

1.3 Συνεισφορά

Η εργασία αυτή φιλοδοξεί να παράσχει ένα εύχρηστο και ασφαλές σύστημα για Docker containers χρησιμοποιώντας το AppArmor για να ενισχύσει την άμυνα του host περιβάλλοντος, ώστε να μπορεί να κανείς να εξετάσει τους containers ως μία βιώσιμη εναλλακτική των εικονικών μηχανών για ένα ευρύ φάσμα εφαρμογών. Προκειμένου να επιτευχθεί αυτός ο στόχος, κατ' αρχάς, θα εξετάσουμε αναλυτικά τους μηχανισμούς που προσφέρει ο πυρήνας των Linux (συνήθως ίδιοι οι παρόμοιοι μηχανισμοί μπορούν να βρεθούν και σε άλλα σύγχρονα λειτουργικά συστήματα) στους

οποίους στηρίζεται η τεχνολογία της εικονοποίησης επίπεδου λειτουργικού συστήματος, θα διερευνήσουμε το επίπεδο απομόνωσης που προσφέρουν και πώς θα μπορούσαμε να χρησιμοποιήσουμε κάποιους από αυτούς ως επιπρόσθετο επίπεδο ασφαλείας, καθώς και πιθανές αδυναμίες ή περιορισμούς έχουν. Στη συνέχεια, αφού εξετάσουμε τις αδυναμίες των containers ως σύστημα, αλλά και συγκεκριμένα της πλατφόρμας του Docker, θα αναπτύξουμε και θα αξιολογήσουμε ένα αυτοματοποιημένο σύστημα ασφάλειας για Docker containers, το οποίο θα στηρίζεται σε αυτοματοποιημένη κατασκευή και εφαρμογή Apparmor profiles, κατά τη δημιουργία των containers, αλλά και κατά τη φάσης της εκτέλεσης της εφαρμογής σε κάποια περίοδο εκπαίδευσης του συστήματος, εφόσον το επιθυμεί ο χρήστης. Στόχος της εργασίας είναι ο χρήστης να αποκομίζει τα οφέλη και τις ευκολίες που προσφέρει η πλατφόρμα του Docker, όπως η χρήση συγκεκριμένων υπολογιστικών πόρων, η γρήγορη και εύκολη δημιουργία και διαχείριση containers και το repository από containerized εφαρμογές, χωρίς, όμως, να θυσιάζει την ασφάλεια του συστήματος. Προς την κατεύθυνση αυτή, έχει δημιουργηθεί ένα command line interface όμοιο με αυτό του Docker, ώστε ο χρήστης να μπορεί να χρησιμοποιήσει απευθείας το αυτοματοποιημένο σύστημα ασφάλειας, χωρίς να απαιτείται επιπλέον κόπος για την εκμάθηση του εργαλείου, κάτι το οποίο συμβαδίζει με την φιλοσοφία του Docker για απλοποίηση των containers.

2 CONTAINERS

2.1 Ιστορία

Η έννοια του container εμφανίστηκε για πρώτη φόρα το 1979, με την υλοποίηση στα UNIX (στην έκδοσης 7, V7) της κλήσης συστήματος chroot. Αυτή η κλήση συστήματος είχε τη δυνατότητα να αλλάζει τον κατάλογο-ρίζα του συστήματος αρχείων μίας διεργασίας και των παιδιών της, δημιουργώντας έτσι μία διαφορετική όψη του συστήματος αρχείων στο εσωτερικό του chroot (γνωστό πλέον ως chroot jail), προσφέροντας κατ' αυτόν τον τρόπο μία πρώτη μορφή απομόνωση διεργασιών. Το chroot στη συνέχεια υλοποιήθηκε και στα BSD (έκδοση 4.2) το 1982. Ο όρος jail χρησιμοποιήθηκε το 1991 για πρώτη φορά για να περιγράψει ένα chroot περιβάλλον από τον Bill Cheswik κατά την υλοποίηση ενός honeypot [18].

Λίγο αργότερα δημιουργήθηκε η έννοια του namespace από το κατανεμημένο λειτουργικό σύστημα Plan 9 της Bell Labs, του οποίο η σχεδίαση ξεκίνησε τη δεκαετία του 80, ενώ η πρώτη σταθερή έκδοση βγήκε το 1992. Το λειτουργικό λόγω της κατανεμημένης φύσης του, χρησιμοποιούσε εκτενώς τα namespaces προκειμένου να μπορέσει να παρέχει σε κάθε διεργασία ένα ξεχωριστό χώρο εκτέλεσης, ο οποίος αναγνωρίζεται (για κάθε διεργασία) από ένα μοναδικό namespace και επηρεάζει κομμάτια το λειτουργικού, όπως το σύστημα αρχείων, τα mounts και το IPC (inter-process communication). Για παράδειγμα δύο διεργασίες μπορεί να βλέπουν χρησιμοποιώντας το ίδιο path δύο διαφορετικά αρχεία, επειδή βρίσκονται σε διαφορετικά namespaces. Μεταγενέστερα, το 2002, επηρεαζόμενοι από τα namespaces του Plan 9, αλλά με διαφορετική φιλοσοφία, υλοποιήθηκε στον πυρήνα των Linux το πρώτο namespace (mount namespace) και ενσωματώθηκε στην έκδοση 2.4.19 του πυρήνα. Στη συνέχεια, από το 2006 και συνεχίζοντας μέχρι σήμερα, πολλά τμήματα του λειτουργικού μπορούν να απομονωθούν με namespaces, όπως τα pids (process ids), ωστόσο παραμένουν πολλά για τα οποία δεν υπάρχουν namespaces, όπως η υπηρεσία χρονισμού ή το syslog.

Η επόμενη σημαντική εξέλιξη ήρθε σχεδόν μία δεκαετία μετά από το Plan 9, το 2000, με τα FreeBSD jails (στην έκδοση 4.0 των FreeBSD), από μία μικρή εταιρία hosting (από τον Derrick Woolworth ιδιοκτήτη της R&D Associates) προκειμένου να προσφέρει σαφή διαχωρισμό και ασφάλεια μεταξύ των υπηρεσιών των πελατών της και ευκολότερη διαχείριση αυτών. Η κλήση συστήματος jail παρείχε υπηρεσίες sandboxing, όπως απομόνωση του συστήματος αρχείων, των χρηστών, των διεργασιών και του δικτύου. Έτσι, με την κλήση jails δημιουργούνταν μικρά, απομονωμένα συστήματα, γνωστά ως jails, τα οποία μπορούσαν να έχουν τη δικιά τους IP, διαφορετικές εγκαταστάσεις ανά jail και διαφορετικά configurations. Αυτό έως το 2004 είχε οδηγήσει στην καθιέρωση του όρου jailbreak.

Το 2001 εμφανίστηκε ο μηχανισμός Linux VServer, ένας μηχανισμός που επέτρεπε τον διαχωρισμό διαφόρων τμημάτων του λειτουργικού, όπως το σύστημα αρχείων, οι διευθύνσεις δικτύου, η μνήμη και ο χρόνος CPU, αν και έλειπαν προχωρημένα features όπως αυτά που παρέχουν σήμερα τα namespaces. Ήταν η πρώτη φορά, όπου μία τεχνική εικονοποίησης επιπέδου λειτουργικού προσέφερε ένα σαφώς ορισμένο security context για κάθε χρήστη, δίνοντας του την αίσθηση ότι τρέχει σε ένα

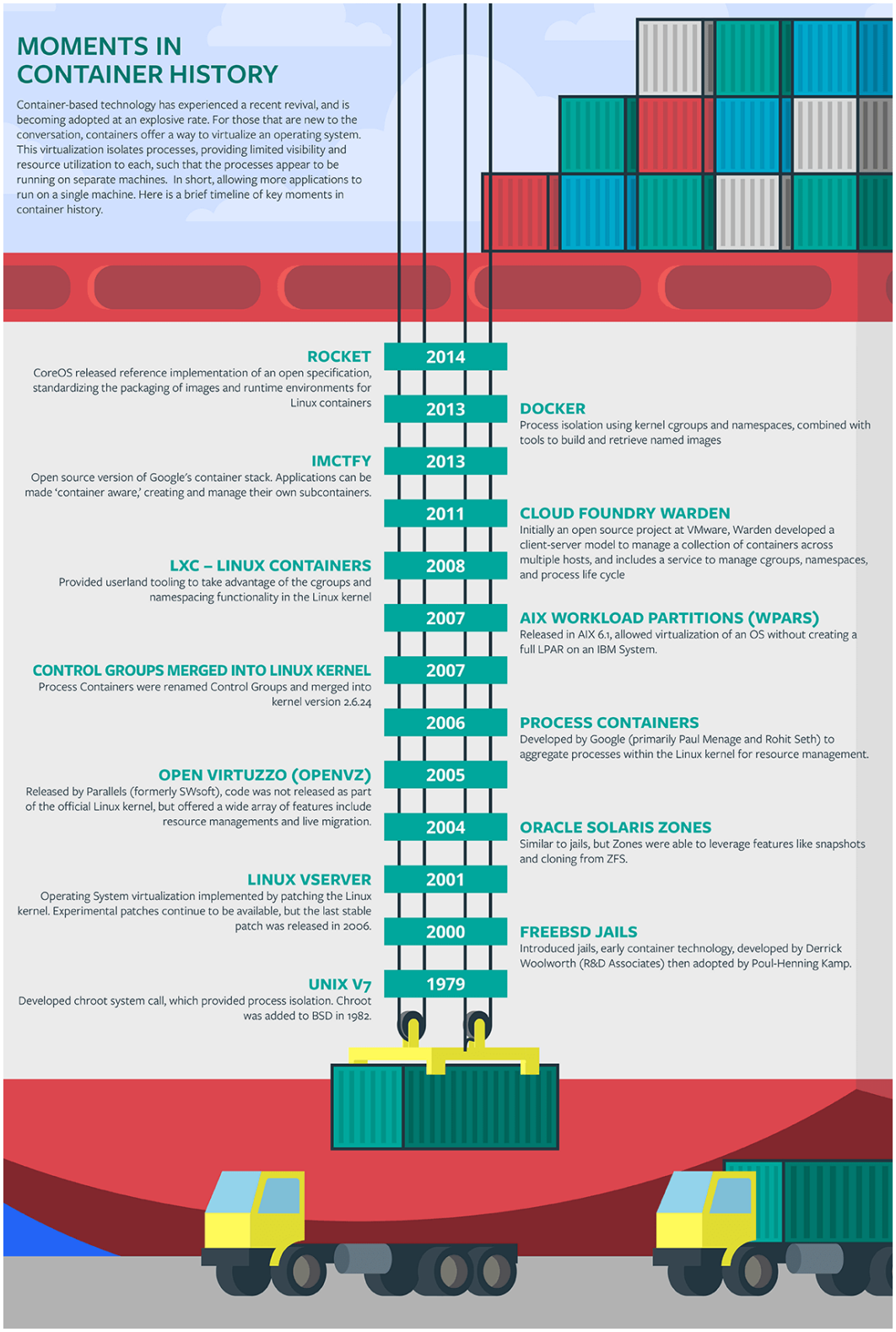
απομονωμένο μηχάνημα, ενώ στην πραγματικότητα έτρεχε σε έναn ιδιωτικό εικονικό εξυπηρετητή (virtual private server). Και όλα αυτά, σε μία περίοδο που η παραεικονοποίηση είχε αρχίσει να κερδίζει έδαφος, ενώ η εικονοποίηση με υποβοήθηση από Hardware δεν ήταν διαδεδομένη, καθώς ήταν πολύ δαπανηρή. Ωστόσο το project τελικά δεν έγινε τόσο δημοφιλές, καθώς η χρήση του VServer, απαιτούσε έναν τροποποιημένο πυρήνα που ήταν δύσκολο να υποστηριχθεί από πολλές Linux διανομές και δυσχεραίνει τις εργασίες των διαχειριστών αυτών των συστημάτων.

Στη συνέχεια, το 2004, ακολούθησαν οι Solaris containers (ή αλλιώς Solaris Zones), προσφέροντας παρόμοιες δυνατότητες με τα jails, όπως οριοθέτηση μεταξύ των διεργασιών διαφορετικών χρηστών και διαχείριση πόρων, και επιπλέον τη δυνατότητα για snapshots και για κλωνοποίηση των containers.

Το 2005, εμφανίστηκε μία παρόμοια τεχνολογία, το Open VZ (Open Virtuozzo), το οποίο παρείχε δυνατότητες εικονοποίησης, απομόνωσης, διαχείρισης πόρων και checkpointing. Ωστόσο και το Open VZ, απαιτούσε έναν custom πυρήνα, αφού ποτέ δεν εντάχθηκε στον mainline πυρήνα.

Τον επόμενο χρόνο, το 2006, άρχισαν να αναπτύσσονται από μηχανικούς της Google (κυρίως από τον Paul Menage και τον Rohit Seth) οι Process Containers, μία τεχνολογία που επέτρεπε σε ομάδες από διεργασίες να έχουν σαφή οριοθέτηση της χρήσης ορισμένων πόρων του συστήματος, όπως CPU, μνήμη, I/O δίσκου ή δικτύου κλπ. Τελικά οι Process Containers μετονομάστηκαν (στα τέλη του 2007) σε control groups ή cgroups για συντομία, για να αποφευχθεί οποιαδήποτε σύγχυση με τον όρο container, καθώς ήδη από εκείνη την εποχή ήταν όρος που χρησιμοποιούνταν αρκετά και με διαφορετικές ονομασίες. Τελικά, τα control groups εντάχθηκαν το 2008 στον στάνταρ πυρήνα στην έκδοση 2.6.24 (μετά από αρκετές τροποποιήσεις που επέτρεπαν καλύτερη διαχείριση και έλεγχο περισσότερων πόρων του συστήματος) [19].

Από το 2008 και μετά αρχίζουν αν εμφανίζονται οι πρώτες πλατφόρμες για Linux Containers, με πρώτο το LXC, το οποίο παρείχε εργαλεία σε χώρο χρήστη για τη δημιουργία και την διαχείριση containers αξιοποιώντας τα namespaces και τα control groups. Άλλα projects που ακολούθησαν και ασχολήθηκαν με τεχνολογίες σχετικές με containers είναι το Cloud Foundry Warden (αρχικά project ανοιχτού κώδικα από τη VMware, 2011), LMCTFY (Let Me Contain That For You, project ανοιχτού κώδικα της Google, 2013), Docker (2013), Kubernates (project ανοιχτού κώδικα της Google, 2014) και Rocket (υλοποίηση του CoreOS, 2014).



Εικόνα 4: Ιστορική ανασκόπηση των Containers

2.2 Οφέλη και χρήσεις

Από τότε που εμφανίστηκαν τα πρώτα chroot περιβάλλοντα και τα FreeBSD Jails, οι τεχνολογίες που βοηθούν την κατασκευή των containers έχουν βελτιωθεί αισθητά και πλέον οι containers θεωρούνται μία ώριμη τεχνολογία που μπορεί να υποστηρίξει παραγωγικά συστήματα (production ready), υποκαθιστώντας τις εικονικές μηχανές. Τα πλεονεκτήματα που προσφέρουν είναι ποικίλα και μπορούν να αξιοποιηθούν σε ένα ευρύ φάσμα εφαρμογών. Κατ' αρχάς, προσφέρουν ένα απομονωμένο περιβάλλον εκτέλεσης με ελάχιστο έως μηδενικό overhead. Αυτός είναι και ο κυριότερος λόγος για τον οποίον το ενδιαφέρον για τους containers έχει αυξηθεί σημαντικά τα τελευταία χρόνια, καθώς μπορούν να μειώσουν σημαντικά τα κόστη σε σχέση με τη χρήση εικονικών μηχανών, αφού το διαθέσιμο hardware αξιοποιείται πολύ πιο αποδοτικά (για παράδειγμα σε ένα συμβατικό laptop υπολογιστή μπορεί κανείς εύκολα να εκκινήσει 100 containers ταυτόχρονα, ενώ μπορεί να εκτελεί λιγότερο από 10 εικονικές μηχανές ταυτόχρονα). Επίσης, προσφέρουν αποτελεσματική απομόνωση των χρησιμοποιούμενων πόρων, όπως μνήμη, CPU και ταχύτητα I/O, με δυνατότητες ρύθμισης των πόρων που δίνονται στον κάθε container κατά την εκτέλεση της εφαρμογής, ανάλογα με τις ανάγκες που προκύπτουν. Ακόμη, επιτρέπουν το εύκολο πακετάρισμα και διανομή των εφαρμογών μαζί με τις απαραίτητες ρυθμίσεις τους, γεγονός που μειώνει αισθητά τον κόπο και τον χρόνο που απαιτείται, ώστε να εγκατασταθεί και να είναι έτοιμη για χρήση μία εφαρμογή. Για τους παραπάνω λόγους, έχουν σχεδιαστεί πολλοί containers (ή πιο σωστά container engines), κάθε ένας από τους οποίες δίνει έμφαση σε διαφορετικά πλεονεκτήματα της εικονοποίησης επιπέδου λειτουργικού συστήματος και, συνεπώς, είναι κατάλληλος για διαφορετικού τύπου εφαρμογές από τους άλλους.

Τα παραπάνω χαρακτηριστικά, καθιστούν τους containers κατάλληλους για σύγχρονες αρχιτεκτονικές και σύγχρονα μοντέλα προγραμματισμού. Πιο συγκεκριμένα, εξαιτίας των παραπάνω χαρακτηριστικών, διευκολύνουν κατά πολύ την χρήση του cloud και την αξιοποίηση των δυνατοτήτων του, γεγονός που αποδεικνύεται και από το ότι πολλοί πάροχοι υπηρεσιών cloud, όπως η Amazon και η Google, παρέχουν υποστήριξη για containers και επενδύουν σε αυτή την τεχνολογία. Χάρη στο γεγονός ότι καταναλώνουν λίγους πόρους, ενώ ταυτόχρονα επιτρέπουν εύκολο πακετάρισμα των εφαρμογών, οι containers σε πολλές περιπτώσεις χρησιμοποιούνται σε cloud based εφαρμογές, προσφέροντας εύκολο scale up και scale down σε πραγματικό χρόνο ανάλογα με τον φόρτο του συστήματος και γρήγορο deployment (π.χ. eBay, Box, Spotify). Επίσης, ευνοούν την αρχιτεκτονική των microservices, βάση της οποίας, ευνοείται η modular υλοποίηση μικρών λειτουργιών του συστήματος, αντί της μονολιθικής υλοποίησης ολόκληρης της απαιτούμενης λειτουργικότητας, πρακτική η οποία βοηθά στην επαναχρησιμοποίηση κώδικα και σε ευκολότερη συντήρηση. Προκειμένου να υποστηριχθούν τα παραπάνω, έχουν δημιουργηθεί πολλές πλατφόρμες για αυτόματο orchestration των containers, οι οποίες διευκολύνουν κατά πολύ τη διαχείριση του συστήματος, όπως το Kubernates και το Apache Mesos. Εκτός από τα παραπάνω, οι containers ευνοούν σύγχρονα μοντέλα προγραμματισμού όπως το DevOps, το οποίο φέρνει πιο κοντά τους υλοποιητές ενός συστήματος με το τμήμα της συντήρησης της υποδομής, και τα CI/CD (continuous integration/continuous

delivery), σύμφωνα με τα οποία σχεδιάζονται, υλοποιούνται, δοκιμάζονται και παραδίδονται μικρά τμήματα της απαιτούμενη λειτουργικότητας ενός συστήματος. Τέλος, σε πολλές περιπτώσεις οι containers είναι κατάλληλοι για test και QA (quality assurance) περιβάλλοντα, καθώς η εφαρμογή που βρίσκεται στην δοκιμαστική φάση, καθώς και τα κατάλληλα εργαλεία, βρίσκονται πακεταρισμένα μέσα σε ένα container, τον οποίο μπορεί κανείς εύκολα να κατεβάσει και να εκτελέσει μέσα σε ελάχιστο χρόνο.

Τα οφέλη των containers, όμως δεν περιορίζονται μόνο στην εικονοποίηση των servers. Διάφορες εφαρμογές που χρησιμοποιούμε καθημερινά στους υπολογιστές μας, μπορούν να αξιοποιήσουν τα πλεονεκτήματα που προσφέρουν οι containers. Μέσω της εικονοποίησης επιπέδου λειτουργικού συστήματος παρέχονται, πλέον, έτοιμα εργαλεία με τα οποία μπορεί κανείς να θωρακίσει ευαίσθητα τμήματα της εφαρμογής του, ή και ολόκληρη την εφαρμογή, εύκολα και αποτελεσματικά. Για παράδειγμα, μέχρι στιγμής, ορισμένοι φυλλομετρητές (όπως το Chromium και το Google Chrome) χρησιμοποιούν custom sandboxes για να εκτελέσουν κώδικα που προέρχεται από untrusted πηγές του διαδικτύου προστατεύοντας έτσι αποτελεσματικά τον χρήστη. Ωστόσο, επειδή αυτές οι υλοποίησες είναι δύσκολες και πολύπλοκες, τα application sandboxes χρησιμοποιούνται από ελάχιστες εφαρμογές. Οι containers, όμως, παρέχουν εύκολα ένα παρόμοιο περιβάλλον με τα sandboxes, που μπορεί να χρησιμοποιηθεί για την προστασία από διάφορους κινδύνους. Τέλος, αρχίζουν να δημιουργούνται τεχνολογίες οι οποίες βοηθούν στο να εκτελούνται όλες οι εφαρμογές ενός υπολογιστή σε απομονωμένα περιβάλλοντα. Χαρακτηριστικό παράδειγμα αποτελεί το Oz σύστημα, το οποίο ξεκινά κάθε desktop εφαρμογή (με την βοήθεια του xpra) ενός Linux συστήματος σε ένα απομονωμένο περιβάλλον, προστατεύοντας αποτελεσματικά εφαρμογές που χρησιμοποιούμε καθημερινά και είναι εκτεθειμένες σε διάφορες επιθέσεις μέσω του διαδικτύου, όπως οι φυλλομετρητές και τα προγράμματα ηλεκτρονικής αλληλογραφίας. Αντίστοιχες υπηρεσίες προσφέρει το υψηλής ασφάλειας λειτουργικό σύστημα SubgraphOS, αξιοποιώντας τεχνολογίες που έχουν αναπτυχθεί λόγω των containers, και χρησιμοποιώντας έναν Linux πυρήνα με gresecurity και το σύστημα Oz.

2.3 Εργαλεία

Σε αυτό το σημείο αξίζει να εξετάσουμε τα εργαλεία και τις τεχνολογίες που χρησιμοποιούνται για την κατασκευή των containers, ώστε να κατανοήσουμε την λειτουργία τους και πώς μπορούμε να τα συνθέσουμε ώστε να σχηματίσουμε ένα απομονωμένο περιβάλλον. Θα προσπαθήσουμε να περιγράψουμε τα δυνατά σημεία του κάθε μηχανισμού, αλλά και να καταλάβουμε τους περιορισμούς τους. Αυτό θα μας επιτρέψει να δούμε με ποιον τρόπο μπορούμε να ασφαλίσουμε έναν container, ώστε να προστατέψουμε το host σύστημα, αλλά και τους υπόλοιπους containers που μπορεί να τρέχουν σε αυτό το μηχάνημα, σε περίπτωση επίθεσης στον container.

2.3.1 Chroot

Ένα από τα πρώτα εργαλεία που χρησιμοποιήθηκε για απομόνωση διεργασιών του λειτουργικού, όπως είδαμε και στην ιστορία των containers, είναι το *chroot*. Η κλήση συστήματος chroot αλλάζει τον κατάλογο-ρίζα (root directory) της διεργασίας που την καλεί, καθώς και των παιδιών της, δηλαδή των διεργασιών που αυτή εκκινεί (αφού εκτελέσει την κλήση συστήματος). Αυτό σημαίνει ότι με αυτή την κλήση συστήματος, απλώς αλλάζει ένα συστατικό στοιχείο του path resolution, όταν μονοπάτια προς αναζήτηση ξεκινούν από τον κατάλογο-ρίζα (,δηλαδή από το /). Δηλαδή, αλλάζει το reference της τρέχουσας ρίζας , '/', ακριβώς όπως η κλήση συστήματος *chdir* αλλάζει το reference του current working directory, '**.**' . Συνεπώς, η κλήση συστήματος chroot, παρόλο που αλλάζει τον κατάλογο που βλέπει ως ρίζα μία διεργασία, το οποίο σημαίνει ότι αυτή η διεργασία είναι περιορισμένη στο να βλέπει και να προσπελαύνει μόνο τους φακέλους και τα αρχεία που βρίσκονται κάτω από το νέο root φάκελο, δεν αρκεί από μόνη της για να παρέχει ασφάλεια, αλλά αποτελεί μόνο ένα μικρό κομμάτι της απομόνωσης ενός προγράμματος [20]. Αυτό συμβαίνει διότι υπάρχουν αρκετοί τρόποι με τους οποίους μία διεργασία που βρίσκεται σε chroot μπορεί να ξεφύγει από αυτό, αρκεί να έχει τα κατάλληλα δικαιώματα, όπως υποδεικνύεται από το manpage της κλήσης συστήματος [21].

Παρακάτω φαίνεται ένα παράδειγμα χρήσης ενός chroot jail, καθώς και ένα απλό escape. Για ευκολία ξεκινάμε τη διαδικασία από το shell χρησιμοποιώντας το εργαλείο (χώρου χρήστη) chroot (το οποίο εκτελεί την κλήση συστήματος chroot), σε ένα φάκελο όπου υπάρχει το εκτελέσιμο ενός shell, καθώς και το εκτελέσιμο ενός chroot escape:

Εικόνα 5: Παράδειγμα chroot περιβάλλοντος και chroot escape

Παρατηρούμε αρχικά ότι το user-space εργαλείο *chroot*, σε αντίθεση με την κλήση συστήματος, εκτελεί και *chdir* μέσα στο chrooted environment, γι' αυτό και ξεκινάμε το νέο shell απευθείας μέσα στον φάκελο chroot και πώς η εκτέλεση της chroot απαιτεί root δικαιώματα ή πιο συγκεκριμένα CAP_SYS_CHROOT capability. Επίσης βλέπουμε ότι αρχικά τρέχοντας την εντολή *whoami*, δεν μπορούμε να βρούμε την αντιστοίχιση του χρήστη με ID 0, καθώς λείπουν τα κατάλληλα αρχεία, ούτε μπορούμε να τρέξουμε την εντολή *ls*, αφού δεν έχουμε το αντίστοιχο εκτελέσιμο μέσα στον φάκελο chroot/. Όμως, αφού τρέξουμε το εκτελέσιμο unchroot.exe, οι δύο εντολές εκτελούνται σωστά, το οποίο σημαίνει ότι έχουμε αλλάξει φάκελο, μιας και δεν υπήρχε το εκτελέσιμο του *ls* στον φάκελο που είμασταν πριν και μάλιστα με το *ls* βλέπουμε ότι όντως έχουμε φτάσει στο root φάκελο του συστήματος, αφού βλέπουμε φακέλους όπως το /boot και το /proc.

Όσον αφορά το εκτελέσιμο unchroot.exe, ο κώδικας σε C που απαιτείται είναι ο ακόλουθος:

```
#include <unistd.h>
#include <sys/stat.h>

int main(){
      int i;
      mkdir("foo",0755);
      chroot("foo");
      for (i=0; i<100; i++) chdir("..");
      chroot(".");
      return execl("/bin/sh","-i",NULL);
}
```

Δηλαδή, το μόνο που απαιτείται είναι να έχουμε το δικαίωμα να δημιουργήσουμε φάκελο μέσα στο chrooted περιβάλλον και να εκτελέσουμε την κλήση συστήματος chroot (CAP_SYS_CHROOT). Στη συνέχεια βρισκόμαστε έξω από το jail, αφού ο νέος κατάλογος ρίζα (στο παράδειγμα foo, δηλαδή '/'= 'chroot/foo/') δείχνει ήδη στο παιδί τους τρέχοντος καταλόγου (το οποίο είναι ο φάκελος που έγινε το πρώτο chroot, δηλαδή '.' = 'chroot/'). Τέλος εκτελώντας chdir προς τα πίσω, δηλαδή προς την αρχική ρίζα, μπορούμε να ξεφύγουμε και να προσπελάσουμε οποιοδήποτε αρχείο του

συστήματος (αρκεί να έχει πρόσβαση ο χρήστης που χρησιμοποιούμε, μιας και μπορεί να μην είναι root, αφού το escape μπορεί να γίνει από οποιονδήποτε με αρκετά priviledges).

2.3.2 Namespaces

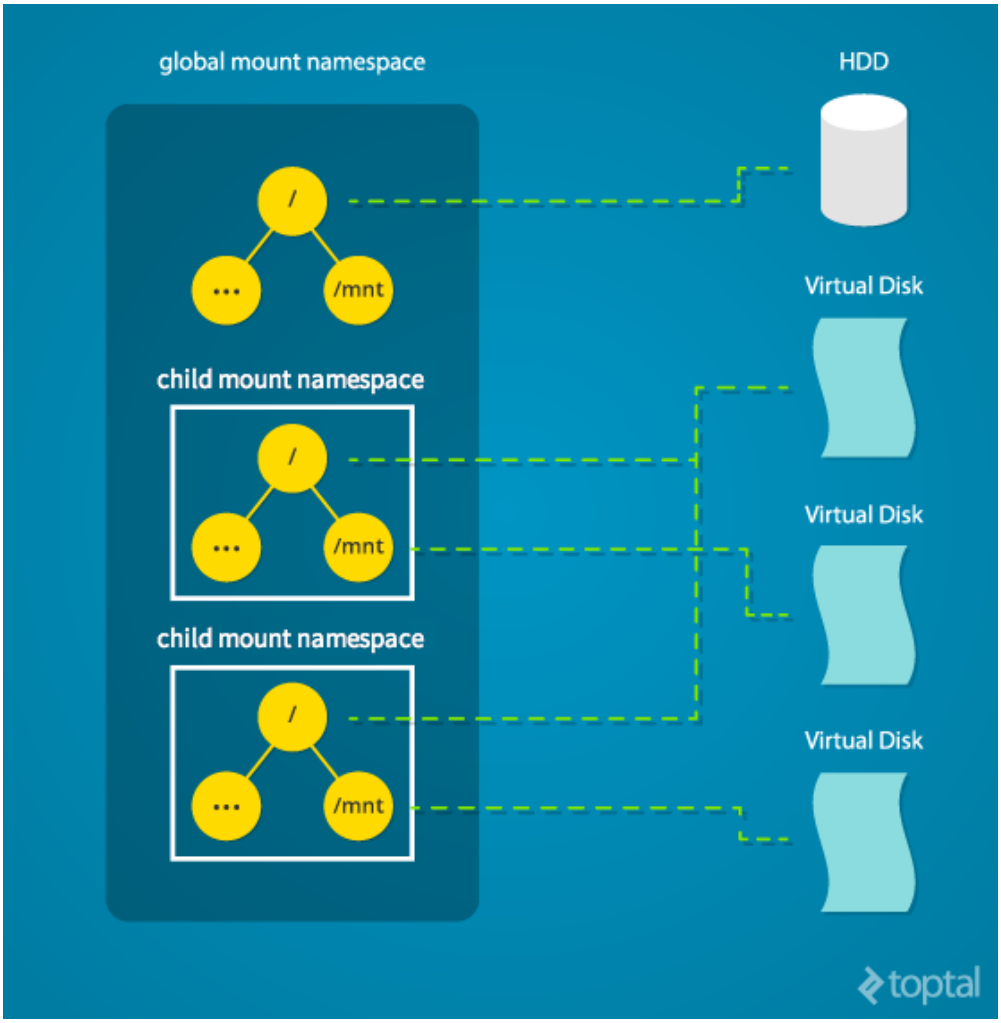
Τα Namespaces είναι ένα πρόσφατο σχετικά feature που προσφέρει ο πυρήνας των Linux, το οποίο επιτρέπει στις διεργασίες να βλέπουν μία δικιά τους όψη για κάποια τμήματα του συστήματος. Αυτό σημαίνει πως οποιαδήποτε αλλαγή κάνουν μέσα σε αυτό το namespace θα το βλέπουν μόνο οι διεργασίες που μοιράζονται αυτό το namespace (ή και τα child namespaces εφόσον το εν λόγω namespace είναι ιεραρχικό) και δεν θα επηρεάσει τις υπόλοιπες διεργασίες του λειτουργικού συστήματος. Τα νέα namespaces δημιουργούνται εκτελώντας την κλήση συστήματος *clone*(2) ή την *unshare*(2) με τα κατάλληλα flags, ενώ μία διεργασία μπορεί να μπει σε υπάρχοντα namespaces καλώντας την *setns*(2). Η κλήση συστήματος clone χρησιμοποιείται για να δημιουργήσει νέα διεργασία, παιδί της καλούσας, μέσα στα νέα namespaces, ενώ η unshare τοποθετεί την τρέχουσα διεργασία στα νέα namespaces.

Σε αυτό το σημείο αξίζει να τονίσουμε ότι τα namespaces προστέθηκαν στον πυρήνα σε αρκετά μεταγενέστερες εκδόσεις (ξεκινώντας από την 2.4.19, όπως προαναφέρθηκε) και υλοποιήθηκαν σταδιακά, καθώς δεν υπήρχε αρχική πρόβλεψη στον πυρήνα για τέτοιου είδους features απομόνωσης. Αυτή η διαδικασία έχει αποδειχτεί ότι είναι σαφώς πιο δύσκολη και πιο ευάλωτη σε λάθη, σε σχέση με την ανάπτυξη ενός συστήματος σχεδιασμένο ώστε να παρέχει συγκεκριμένες υπηρεσίες ασφάλειας. Το παραπάνω γεγονός έρχεται να επιβεβαιώσει το κενό ασφαλείας (CVE 2013-1858) που βρέθηκε τον Μάρτιο του 2013 στο user namespace στον πυρήνα 3.8 [22]. Γι' αυτό το λόγο είναι δύσκολο και χρονοβόρο να καλυφτούν όλα τα υποσυστήματα των Linux και κατά καιρούς έχουν ζητηθεί επιπρόσθετα namespaces για διάφορα τμήματα του λειτουργικού, όπως για το timing service,τον syslog δαίμονα και το sys pseudo file system.

Παρακάτω παρουσιάζονται τα namespaces που έχουν υλοποιηθεί μέχρι στιγμής με χρονολογική σειρά, καθώς και μερικά σύντομα παραδείγματα χρήσης τους.

2.3.2.1 Mount

Το mount namespace είναι το πρώτο namespace που υλοποιήθηκε στον πυρήνα των Linux (το 2002) και ένα από τα σημαντικότερα. Μπορεί να προσφέρει ένα περιβάλλον παρόμοιο με το chroot, ωστόσο αρκετά πιο ασφαλές. Πιο συγκεκριμένα, επιτρέπει στις διεργασίες που βρίσκονται εντός του namespace να έχουν διαφορετική όψη της ιεραρχίας των συστημάτων αρχείων από το υπόλοιπο σύστημα. Δηλαδή, μία διεργασία (ή μία ομάδα διεργασιών), μπαίνοντας σε ένα νέο mount namespace, μπορεί να αφαιρέσει κάποια mounted file συστήματα (τα οποίο κληρονόμησε από το αρχικό namespace) και να προσθέσει άλλα mounts, χωρίς να επηρεάσει τις διεργασίες του υπόλοιπου συστήματος που βρίσκονται εκτός του συστήματος στο αρχικό namespace (initial-namespace). Στην εικόνα φαίνεται μία σχηματική αναπαράσταση της έννοιας του mount namespace:



Εικόνα 6: Mount namespaces

Ο πυρήνας των Linux κρατά μία δομή η οποία περιέχει όλα τα mount points του συστήματος, μαζί με τους τύπους των συστημάτων αρχείων και τις ρυθμίσεις (flags) που χρησιμοποιήθηκαν. Κάθε φορά που δημιουργείται ένα νέο mount namespace, αυτή η δομή αντιγράφεται, ώστε να κρατά τις απαραίτητες πληροφορίες για το νέο namespace, γι' αυτό και αρχικά, by default, το νέο namespace διατηρεί την ίδια ιεραρχία του file system με το parent namespace. Για τον λόγο αυτό, στην περίπτωση που θέλουμε να απομονώσουμε το περιβάλλον όπου θα εκτελεστεί μία επικίνδυνη διεργασία, αφού δημιουργήσουμε το νέο mount namespace, πρέπει να αλλάξουμε επικίνδυνα mount points, όπως η ρίζα (/), το sys και το proc με νέα. Συνεπώς βλέπουμε ότι με αυτόν τον τρόπο πετυχαίνουμε σαφώς μεγαλύτερη απομόνωση σε σχέση με ένα chroot περιβάλλον. Μάλιστα, επειδή σε ορισμένες περιπτώσεις η απομόνωση που δίνεται είναι μεγαλύτερη από την επιθυμητή, η αρχική σχεδίαση των mount namespaces έχει επεκταθεί (το 2006 στον πυρήνα 2.6.15), ώστε να υποστηρίζεται η έννοια των shared, των private και των master-slave mount points, γεγονός που επιτρέπει αλλαγές που γίνονται σε ένα namespace να μεταδίδονται και στα υπόλοιπα, εφόσον αυτό είναι επιθυμητό [23]. Για παράδειγμα αν υποθέσουμε ότι εισάγουμε στον υπολογιστή ένα νέο οπτικό δίσκο ή ένα USB flash drive, μπορούμε κάνοντας τα κατάλληλα mount points shared, να έχουμε αυτόματο mounting των νέων μέσων σε όλα τα namespaces, εφόσον τα νέα μέσα γίνουν mount σε κάποιο namespace.

Σε αντίθετη περίπτωση, δηλαδή, αν δεν μαρκαριστεί το mount point ως shared θα πρέπει να γίνει χειροκίνητο mounting των νέων μέσων σε όλα τα namespaces.

2.3.2.2 IPC (Inter-process communication)

Το IPC namespace, απομονώνει τους inter-process communication μηχανισμούς ξεκινώντας από την έκδοση 2.6.19 έκδοση του πυρήνα των Linux. Συγκεκριμένα προσφέρει απομόνωση για τα System V IPC αντικείμενα και, αργότερα, (από την έκδοση 2.6.30 και μετά) τις POSIX ουρές μηνυμάτων. Μέσα σε αυτά τα αντικείμενα περιλαμβάνονται οι ουρές μηνυμάτων (είτε System V είτε POSIX), τα τμήματα μοιραζόμενης μνήμης, που είναι το πιο σύνηθες σενάριο, και ομάδες σημαφόρων [24], τα οποία κατά καιρούς έχουν υπάρξει στόχος επιθέσεων DoS (Denial of Service) ή ακόμη και privilege escalation (π.χ. CVE-2015-7613 [25]) και συνεπώς αυτό το namespace θα μπορούσε να περιορίσει τις παραπάνω επιθέσεις. Αξίζει να σημειωθεί ότι αυτά τα recourses δεν μπορούν να προσπελασθούν μέσω κάποιου path και γι αυτό τα mount namespaces δεν επαρκούν. Αντίστοιχα, όμως, με τα mount namespaces, προκειμένου να μπορούν να επικοινωνήσουν δύο διεργασίες με τους παραπάνω μηχανισμούς IPC, είναι υποχρεωτικό να βρίσκονται στο ίδιο namespace. Αυτό σημαίνει ότι όταν μία διεργασία εκτελέσει την clone (ή την unshare) κλήση συστήματος χρησιμοποιώντας την σημαία *CLONE_NEWIPC* μπαίνει πλέον σε ένα περιβάλλον με απομονωμένους IPC μηχανισμούς από το υπόλοιπο σύστημα. Σε περίπτωση που χρειάζεται IPC επικοινωνία της διεργασίας που βρίσκεται στο νέο namespace με την διεργασία πατέρα (που βρίσκεται στο παλιό namespace) θα πρέπει να γίνουν οι κατάλληλες ρυθμίσεις προτού η διεργασία παιδί αρχίσει να εκτελείται [26].

2.3.2.3 UTS

Το UTS namespace (πυρήνας 2.6.19) απομονώνει δύο προσδιοριστικά του συστήματος. Το hostname ή και nodename και το NIS (Network Information Service) domain name. Το όνομα UTS προέρχεται από το όνομα της δομής που επιστρέφει η κλήση συστήματος *uname* για να εμφανίσει τα παραπάνω αναγνωριστικά, η οποία ονομάζεται utsname, από το "UNIX Time-sharing System". Ουσιαστικά αυτό το namespace επιτρέπει στις διεργασίες να ορίσουν ένα δικό τους hostname ή domain name το οποίο μπορεί να χρησιμοποιηθεί για script που αρχικοποιούν το σύστημα αναλόγως με το host name, χωρίς να χρειάζονται επιπλέον ρυθμίσεις. Αξίζει να σημειωθεί ότι παρά το γεγονός ότι απομονώνει μόνο αυτές τις δύο πληροφορίες του συστήματος, η δημιουργία ενός UTS namespace απαιτεί διαχειριστικά δικαιώματα (συγκεκριμένα CAP_SYS_ADMIN), καθώς εάν επιτραπεί σε μία διεργασία να τροποποιεί αυθαίρετα το hostname του λειτουργικού, το σύστημα μένει εκτεθειμένο σε διάφορες επιθέσεις [27]. Για παράδειγμα επειδή το hostname επιτρέπεται να έχει οποιουσδήποτε χαρακτήρες, μπορεί κανείς κατασκευάζοντας κατάλληλες ακολουθίες χαρακτήρων να οδηγήσει σε διαγραφή σημαντικών αρχείων του συστήματος (αφού ο χαρακτήρας / επιτρέπεται σαν μέρος του hostname) ξεκινώντας μία set-user-ID εφαρμογή σε ένα UTS namespace. Επίσης εάν κάποια setuid εφαρμογές χρησιμοποιεί το hostname σαν μέρος του ονόματος ενός αρχείο κλειδώματος (lock file), τότε

εκτελώντας την εφαρμογή σε ένα κατάλληλο UTS namespace, θα μπορούσε να εξουδετερώσει το αρχείο κλειδώματος αφήνοντας πιθανά instances της εφαρμογής που τρέχουν σε άλλα namespaces εκτεθειμένα σε race conditions.

Στη συνέχεια ακολουθεί ένα παράδειγμα χρήσης του UTS namespace.

```
ubuntu@namespcdemo:~/demo/namespaces$ hostname
namespcdemo
ubuntu@namespcdemo:~/demo/namespaces$ sudo ./utsdemo.exe
sudo: unable to resolve host namespcdemo
Simple UTS namespace demonstration
UTS namespace inside parent:
Nodename inside parent process: namespcdemo
Parent Pid: 29297       PPID: 29296
Child PID: 29298.
Meanwhile into the UTS namespace:
Nodename inside child process: saturn
root@saturn:~/demo/namespaces# ls -l /proc/1/ns/uts
lrwxrwxrwx 1 root root 0 Feb 27 02:30 /proc/1/ns/uts -> uts:[4026531838]
root@saturn:~/demo/namespaces# ls -l /proc/29297/ns/uts
lrwxrwxrwx 1 root root 0 Feb 27 02:42 /proc/29297/ns/uts -> uts:[4026531838]
root@saturn:~/demo/namespaces# ls -l /proc/29298/ns/uts
lrwxrwxrwx 1 root root 0 Feb 27 02:42 /proc/29298/ns/uts -> uts:[4026532149]
root@saturn:~/demo/namespaces# exit
Child 29298 terminated normally
End of UTS demo!
ubuntu@namespcdemo:~/demo/namespaces$ hostname
namespcdemo
ubuntu@namespcdemo:~/demo/namespaces$
```

Εικόνα 7: Παράδειγμα χρήσης του UTS namespace

Στο παραπάνω παράδειγμα αρχικά βρισκόμαστε σε ένα shell στο αρχικό UTS namespace, στο οποίο το hostname είναι **namespcdemo**. Στη συνέχεια εκτελούμε σαν διαχειριστές (ένας απλός χρήστης δεν μπορεί να δημιουργήσει νέο namespace) μία εφαρμογή, η οποία αρχικά εκτελεί την κλήση συστήματος clone ως εξής:

```
pid_t child_pid = clone( test_func, stack + STACK_SIZE
                  ,CLONE_NEWUTS|SIGCHLD , NULL);
```

Βλέπουμε δηλαδή, ότι δημιουργεί μία ένα διεργασία, η οποία θα ξεκινήσει τρέχοντας την **test_func** σε ένα νέο UTS namespace, λόγω του CLONE_NEWUTS flag.

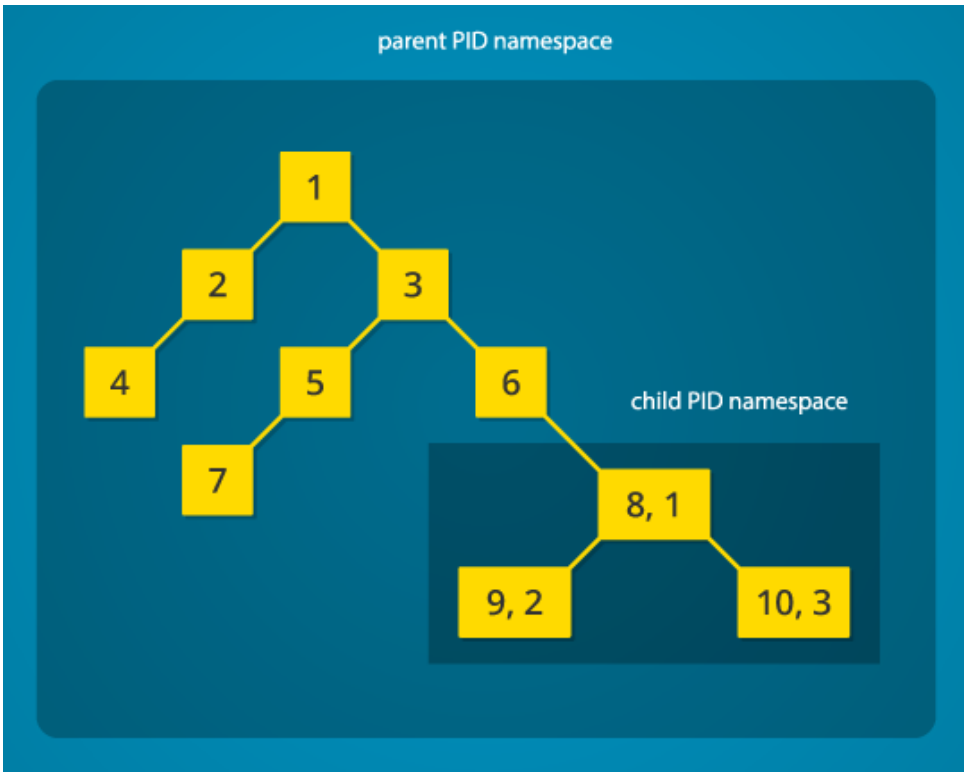
Στη συνέχεια η νέα διεργασία καλεί την *sethostname* κλήση συστήματος για να αλλάξει το hostname σε saturn και έπειτα *execve* για να αρχίσει ένα shell μέσα στο νέο namespace. Παρατηρούμε, όμως ότι τερματίζοντας την νέα διεργασία η αλλαγή αυτή δεν έχει μεταφερθεί στο αρχικό namespace, το οποίο επιβεβαιώνουμε καλώντας την εντολή hostname. Τέλος, το γεγονός ότι βρισκόμασταν σε διαφορετικό namespace φαίνεται εξετάζοντας τα symbolic links που βρίσκονται μέσα στο proc file system. Συγκεκριμένα μπορούμε να δούμε το namespace της κάθε διεργασίας εξετάζοντας το /proc/PID/ns/xxx , όπου xxx το εν λόγω namespace. Δύο διεργασίες με ίδιο namespace έχουν ίδιο inode number για το /proc/PID/ns/xxx. Συνεπώς στο παράδειγμα μας μπορούμε να δούμε ότι η init (με pid 1) και η αρχική διεργασία με pid 29297 βρίσκονται

στο ίδιο uts namespace, ενώ το παιδί της (με pid 29298) βρίσκεται σε διαφορετικό, καθώς εκτελώντας ls -l επιστρέφεται διαφορετικό file handle.

2.3.2.4 PID

Τα PID namespaces είναι σημαντικό feature που προστέθηκε στον mainline πυρήνα στην έκδοση 2.6.24. Μέσω του PID namespace, απομονώνονται τα process ids των διεργασιών εντός του namespace, από το υπόλοιπο σύστημα. Πιο συγκεκριμένα, όταν δημιουργείται ένα νέο namespace η διεργασία που μπαίνει πρώτη σε αυτό παίρνει το PID 1 και συνεπώς γίνεται η init μέσα στο namespace, δηλαδή εάν κάποια διεργασία μείνει ορφανή εντός του namespace θα παραπεμφθεί στην init του namespace by default [28]. Επίσης, εάν αυτή η διεργασία πεθάνει τότε στέλνεται SIGKILL και σε όλες τις υπόλοιπες διεργασίες του namespace και δεν θα επιτρέπονται πλέον forks, εφόσον η init του namespace έχει τερματίσει.

Κάθε διεργασία που ξεκινά μέσα σε ένα PID namespace καταλαμβάνει ένα PID, όχι μόνο μέσα στο namespace, αλλά και στο αρχικό namespace (βλ. εικόνα 8), και συνεπώς καταλαμβάνει περισσότερο από ένα PID. Σε αυτό το σημείο αξίζει να τονίσουμε πως το PID namespace μπορεί να είναι εμφωλευμένο. Αυτό σημαίνει ότι μία διεργασία εντός ενός PID namespace μπορεί να δημιουργήσει ένα νέο namespace. Σε αυτή την περίπτωση δημιουργείται μία ιεραρχία από PID namespaces, όπου κάθε διεργασία μπορεί να δει τα PIDs των διεργασιών του ίδιου namespace, καθώς και των απογόνων αυτού, αλλά όχι των προγόνων. Συνεπώς, από το αρχικό namespace μπορεί κανείς να προσπελάσει όλες τις διεργασίες και όλα τα δέντρα διεργασιών που έχουν δημιουργηθεί.



Εικόνα 8: PID namespace

Εκτός από την απομόνωση που προσφέρουν τα PID namespaces, είναι πολύ σημαντικά για την επίτευξη του migration των containers. Χωρίς αυτά η μετακίνηση ενός container είναι σχεδόν αδύνατη, αφού στην περίπτωση που το PID των διεργασιών που εκτελούνται εντός του container μπορεί να είναι κατειλημμένο στο τελικό σύστημα, καθιστώντας αδύνατη την εκτέλεση του container.

Στη συνέχεια μπορούμε να δούμε πως χρησιμοποιείται ένα PID namespace στην πράξη και πως φαίνεται το περιβάλλον που δημιουργείται παρατηρώντας το εντός και εκτός του namespace.

```
ubuntu@namespacesdemo:~/demo/namespaces$ sudo ./pidns1.exe
Simple PID namespace demonstration
Parent Pid: 13321       PPID: 13320
Child PID: 13322.
Waiting for child to terminate.
Child PID:1      PPID: 0
ubuntu@namespacesdemo:~/demo/namespaces$
```

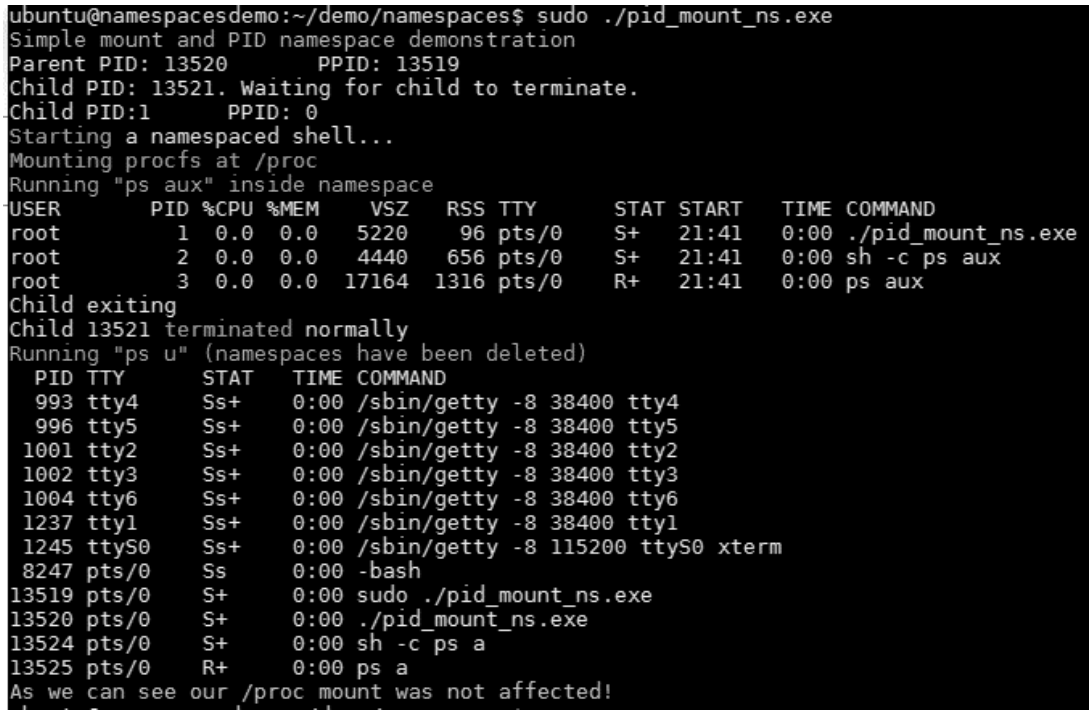
Εικόνα 9: Απλή εφαρμογή του PID namespace

Στο παραπάνω παράδειγμα βλέπουμε ότι η διεργασία με PID 13321 δημιουργεί μία διεργασία παιδί καλώντας την *clone* με το flag **CLONE_NEWPID**. Η διεργασία αυτή μπαίνει συνεπώς σε ένα νέο PID namespace και καλώντας την *getpid* βλέπει το PID του εαυτού της να είναι 1 (και του πατέρα της 0), συνεπώς είναι η διεργασία init το νέου namespace. Ωστόσο ο πατέρας της νέας διεργασίας, ο οποίος βρίσκεται στο αρχικό namespace, βλέπει την ίδια διεργασία παιδί να έχει PID 13322. Συνεπώς βλέπουμε ότι η ίδια διεργασία έχει διαφορετικό PID στα δύο namespaces.

Ωστόσο αν εκτελέσουμε ps από την διεργασία παιδί, θα μας επέστρεφε PID 13322, επειδή το εργαλείο ps προκειμένου να δει τις διεργασίες που τρέχουν στο σύστημα, καθώς και κάποιες πληροφορίες για αυτές, ανατρέχει στο proc σύστημα αρχείων, το οποίο είναι mounted από την εκκίνηση του συστήματος και, επομένως, περιέχει πληροφορίες για το initial namespace. Προκειμένου, λοιπόν, οι διεργασίες ενός νέου PID namespace να έχουν πρόσβαση σε ένα σωστό proc fs, θα πρέπει να δημιουργηθεί ταυτόχρονα και ένα mount namespace (CLONE_NEWNS) και στη συνέχεια να γίνει mount στον κατάλογο /proc το νέο proc σύστημα αρχείων ώς εξής:

```
mount("proc","/proc","proc",0,NULL)
```

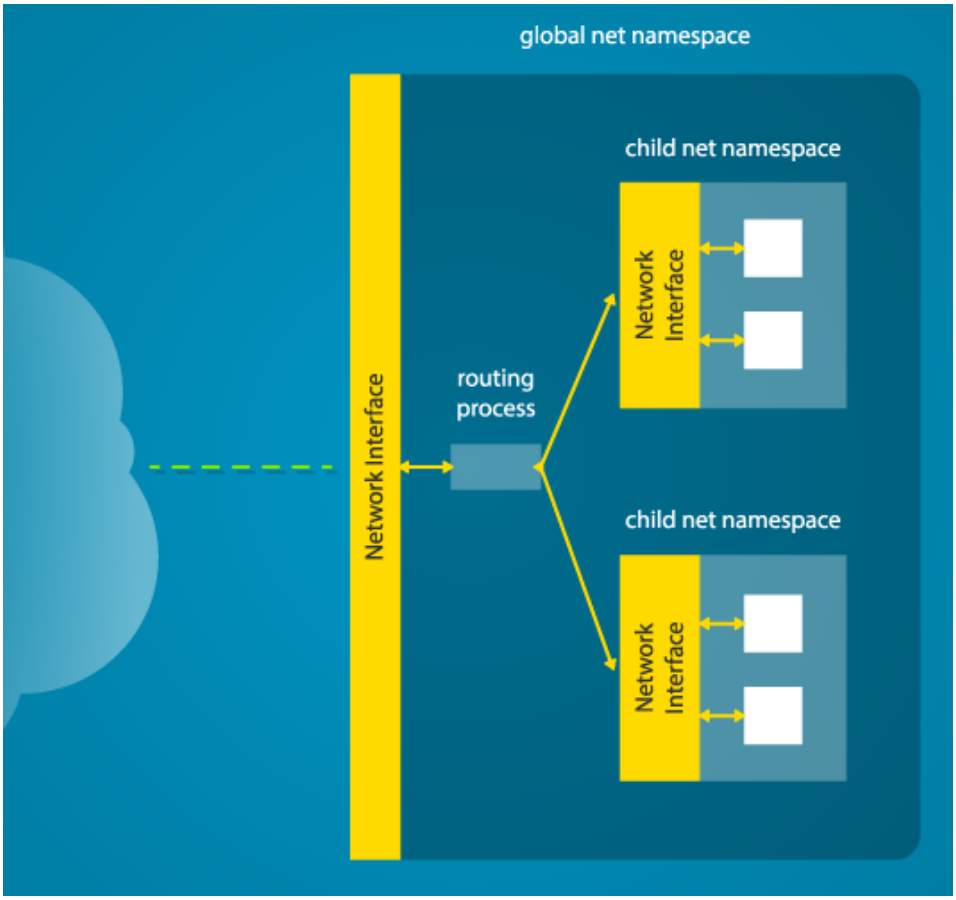
Έπειτα εκτελώντας ps η διεργασία παιδί θα μπορεί να δει τα σωστά PIDs χωρίς να έχει πρόσβαση στις διεργασίες του parent namespace. Ακόμη, παρόλο που κάνουμε mount το νέο proc σύστημα αρχείων, το υπόλοιπο σύστημα δεν επηρεάζεται καθώς το τρέχουμε μέσα σε ένα mount namespace. Το σενάριο αυτό μπορούμε φαίνεται αναλυτικά στην εικόνα 10:



Εικόνα 10: Εφαρμογή των PID και mount namespaces σε συνδυασμό

2.3.2.5 Network

Τα network namespaces εισήχθησαν για πρώτη φορά στην έκδοση 2.6.24 του πυρήνα και το μεγαλύτερο μέρος τους ολοκληρώθηκε στην έκδοση 2.6.29, καθώς έπρεπε να τροποποιηθούν αρκετές λειτουργίες του networking των Linux, ώστε να είναι namespace aware [29]. Τα namespaces αυτά προσφέρουν εικονοποίηση των πόρων του συστήματος που σχετίζονται με τη δικτύωση. Πιο συγκεκριμένα κάθε network namespace μπορεί να έχει δικές του συσκευές δικτύου, δικό του firewall, στοίβα IPv4 και IPv6, κανόνες δρομολόγησης, sockets, ιδιωτικό κατάλογο /proc/net και /sys/ckass/net κοκ. Όταν δημιουργείται ένα νέο namespace, αποκτά απευθείας ένα ξεχωριστό loopback interface. Στη συνέχεια μπορεί κανείς να μεταφέρει μία φυσική συσκευή δικτύου μέσα στο νέο namespace (δεν μπορεί μία φυσική συσκευή δικτύου να υπάρχει σε περισσότερα από ένα namespace) ή να δημιουργήσει κάποιο custom interface. Προκειμένου να υπάρξει επικοινωνία μεταξύ των διαφορετικών network namespaces, έχει υλοποιηθεί ένα εικονικό ζεύγος συσκευών (veth), το οποίο συμπεριφέρεται ως ένα δικτυακό pipe, και τα άκρα του οποίου μπορούν να βρίσκονται σε διαφορετικά namespaces. Συνεπώς, με τη χρήση veth και με τις κατάλληλες ρυθμίσεις (π.χ. routing, gateways, IPs κλπ.) μπορεί κανείς να έχει επιλεκτική επικοινωνία μεταξύ απομονωμένων με network namespaces διεργασιών ή containers ή ακόμα και πρόσβαση σε κάποιο τοπικό δίκτυο και το internet. Αξίζει να σημειωθεί ότι για τα network namespaces υπάρχει πολύ καλή υποστήριξη από userspace εργαλεία τα οποία μπορεί κανείς εύκολα να χρησιμοποιήσει για να κατασκευάσει interfaces σε νέο namespace, να μεταφέρει συσκευές, να στήσει τα κατάλληλα ip routes και γενικά να εκτελέσει στο επιθυμητό namespace οποιαδήποτε εντολή μέσα από shell.



Εικόνα 11: Network namespaces

2.3.2.6 User

Τα user namespaces είναι ένα από τα πιο σύνθετα namespaces γι' αυτό και η είσοδος στον πυρήνα δεν έγινε παρά μόνο στην έκδοση 3.8 του πυρήνα. Σκοπός τους είναι να παρέχουν απομόνωση σε ιδιότητες και χαρακτηριστικά του χρήστη, όπως το user και το group id και τα capabilities. Συγκεκριμένα δημιουργούν ένα απομονωμένο χώρο, όπου ο χρήστης μπορεί να έχει όλα τα δικαιώματα, να συμπεριφέρεται δηλαδή ως root, ενώ έξω από αυτό να είναι κάποιος unprivileged χρήστης. Συνεπώς, με τα user namespaces δίνεται η δυνατότητα σε ένα χρήστη να έχει όλα τα δικαιώματα (capabilities) εντός του namespace, χωρίς ωστόσο να μπορεί να επηρεάσει privileged διεργασίες που βρίσκονται εκτός του namespace ή ευαίσθητα τμήματα του υπόλοιπου συστήματος. Μάλιστα, σε αντίθεση με τα υπόλοιπα namespaces, τα user namespaces μπορούν να δημιουργηθούν από unprivileged χρήστες, γεγονός που επιτρέπει την δημιουργία unprivileged containers. Δηλαδή με την αξιοποίηση των user namespaces, μπορούν να κατασκευαστούν containers από διεργασίες χωρίς διαχειριστικά δικαιώματα, κάτι που μειώνει αισθητά το attack surface των containers, μιας και συνήθως οι πλατφόρμες των containers έχουν root δικαιώματα για να ξεκινήσουν και να διαχειριστούν τους containers. Ωστόσο, ακόμα και σήμερα, οι τεχνικές δυσκολίες είναι πολλές, γι' αυτό και παρόλο που πολλές πλατφόρμες δίνουν αυτή την δυνατότητα, δεν είναι το default, καθώς πολλές ρυθμίσεις δεν μπορούν να γίνουν χωρίς την επέμβαση του διαχειριστή (π.χ. η ρύθμιση του δικτύου).

Δημιουργώντας ένα νέο user namespace μπορούμε να δημιουργήσουμε μία αντιστοίχιση των user και group ids μέσα στο namespace και έξω από αυτό θέτοντας τα κατάλληλα mappings στα αρχεία /proc/PID/uid_map και /proc/PID/gid_map. Κατ' αυτόν τον τρόπο ένας χρήστης με id 1000 στο αρχικό namespace μπορεί να έχει id 0 στο νέο namespace και συνεπώς να είναι διαχειριστής σε αυτό. Μέσα σε αυτόν τον χώρο ο χρήστης μπορεί να διαχειριστεί όλα τα resources τα οποία ανήκουν στο namespace αυτό. Η λίστα με τους πόρους αυτούς συνεχώς μεγαλώνει, ωστόσο μέχρι σήμερα το σημαντικότερο είναι ότι ένας χρήστης με CAP_SYS_ADMIN μπορεί να δημιουργήσει mounts ή bind mounts των τύπων /proc, /sys, tmpfs, cgroups (από την έκδοση 4.6 και μετά) και άλλα. Αντίθετα, δεν μπορεί να κάνει mount block συστημάτων αρχείων (εκτός και αν έχει CAP_SYS_ADMIN στο αρχικό namespace, άρα έχει διαχειριστικά δικαιώματα), να δημιουργήσει νέες συσκευές (mknod) ή να αλλάξει την ώρα του συστήματος.

Σημαντικό είναι το γεγονός ότι τα user namespaces είναι ιεραρχικά επιτρέποντας την ύπαρξη έως 32 εμφωλευμένων επιπέδων. Όταν δημιουργείται ένα νέο namespace, καταγράφεται από τον πυρήνα το euid (effective user ID) της διεργασίας που δημιούργησε το namespace και αυτός ο χρήστης θεωρείται ιδιοκτήτης του namespace. Ο χρήστης αυτός διατηρεί όλα τα capabilities μέσα σε αυτό το namespace, αλλά και σε όλους τους απογόνους αυτού. Γενικότερα οι διεργασίες που είχαν ένα δικαίωμα στο πατρικό namespace, το διατηρούν και σε όλους τους απογόνους αυτού. Έτσι παρόλο που το σύστημα προστατεύεται από τυχόν κακόβουλες διεργασίες που εκτελούνται σε child namespaces, αφήνει στο root χρήστη του συστήματος τη δυνατότητα να διαχειρίζεται όλα τα user namespaces.

Τέλος, σημαντικό είναι το γεγονός ότι όταν το user namespace συνδυάζεται με τα υπόλοιπα namespaces, τότε πρώτα δημιουργείται το user namespace και στη συνέχεια τα υπόλοιπα. Κατ' αυτόν τον τρόπο μία unpriviledged διεργασία μπορεί καλώντας την clone να δημιουργήσει πρώτα ένα user namespace και στη συνέχεια, αφού στο νέο namespace θα έχει διαχειριστικά δικαιώματα (αν και το CAP_SYS_ADMIN αρκεί) μπορεί να δημιουργήσει και όποιο από τα υπόλοιπα namesapces επιθυμεί. Έτσι υποστηρίζεται και η δημιουργία unpriviledged containers.

Στη συνέχεια ακολουθεί ένα παράδειγμα χρήσης του user namespace.

```
ubuntu@namespacesdemo:~/demo/namespaces$ ./userns.exe
Simple user namespace (CLONE_NEWUSER) demonstration
Parent user namespace
eUID: 1000      rUID: 1000
Capabilities=
Parent Pid: 16984       PPID: 15430
Child PID: 16985.
Meanwhile into the user namespace:
Nodename inside child process: eUID: 0  rUID: 0
Capabilities= cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap
raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot
lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,cap_ma
root@ubuntutest-2:~/demo/namespaces# exit
Child 16985 terminated normally
End of user namespace demo!
ubuntu@namespacesdemo:~/demo/namespaces$
```

Εικόνα 12: Απλή εφαρμογή των User Namespaces

Στο παραπάνω παράδειγμα βλέπουμε ότι η αρχική διεργασία είχε euid 1000, δηλαδή έτρεχε χωρίς priviledges, το οποίο επιβεβαιώνεται και από το ότι δεν έχει κανένα capability. Παρόλα αυτά εκτελώντας την κλήση συστήματος clone με το flag CLONE_NEWUSER δημιουργεί μία διεργασία παιδί η οποία όπως φαίνεται παραπάνω έχει όλα τα δικαιώματα στο namespace στο οποίο βρίσκεται. Αξίζει να σημειωθεί ότι προκειμένου να είναι το uid και το guid 0 πρέπει η διεργασία πατέρας, η οποία πλέον είναι ο owner του namespace, να θέσει τις κατάλληλες τιμές στα αρχεία των uid και guid mappings, τα οποία στην προκειμένη περίπτωση είναι τα /proc/16985/uid_map και /proc/16985/gid_map και περιέχουν την γραμμή:

```
0    1000    20
```

, το οποίο σημαίνει ότι η αντιστοίχιση ξεκινά από τον χρήστη με uid 0 εντός του namespace, ο οποίος αντιστοιχίζεται στον χρήστη με uid 1000 εκτός του namespace και η αντιστοίχιση αυτή συνεχίζεται για τα επόμενα 20 uids. Με τον ίδιο ακριβώς τρόπο γίνεται και η αντιστοίχιση σε gids.

2.3.2.7 Cgroup

Το τελευταίο namespace που υλοποιήθηκε είναι το cgroup namespace, το οποίο προστέθηκε πρόσφατα στην έκδοση 4.6 του πυρήνα των Linux [30]. Με την εισαγωγή αυτού του namespace μπορεί πλέον να εικονοποιηθεί η όψη των cgroups ανά γκρουπ διεργασιών. Πλέον κάθε cgroup namespace έχει το δικό του cgroup κατάλογο-ρίζα. Συγκεκριμένα όταν μία διεργασία καλεί την clone ή την unshare κλήση συστήματος με την CLONE_NEWCGROUP σημαία ενεργή, εισέρχεται σε ένα control group namespace, όπου οι φάκελοι που όριζαν τα cgroups της διεργασίας πριν την εκτέλεση της κλήσης συστήματος, μεταφέρονται στους αντίστοιχους καταλόγους-ρίζα του νέο namespace. Αν αυτή η διαδικασία προσπαθήσει να δει τα cgroups μίας διεργασίας που βρίσκεται σε διαφορετικό cgroup namespace (εξαιρουμένου των child cgroup namespaces) τότε, αντί να δει το πραγματικό cgroup path, θα δει ένα μονοπάτι που θα περιέχει το **../** για κάθε πρόγονο namespace, το οποίο όμως δεν μπορεί πλέον να προσπελάσει.

Αυτός ο μηχανισμός έρχεται να λύσει σημαντικά προβλήματα που αντιμετώπιζαν οι containers. Χωρίς το cgroup namespace, ήταν δυνατόν διεργασίες ενός container να προσπελάσουν cgroup μονοπάτια που βρίσκονται εκτός του container και συνεπώς να συλλέξουν σημαντικές πληροφορίες για την πλατφόρμα στην οποία εκτελείται ο container [31]. Επίσης, μία διεργασία μπορούσε να τροποποιήσει το parent cgroup αν αυτό είχε δημιουργηθεί από τον χρήστη της διεργασίας, γεγονός που θα της επέτρεπε να αυξήσει, τελικά, και τα δικά της resources (ή να τα στερήσει από όλες τις διεργασίες που ανήκουν στο parent cgroup). Τέλος, με την υποστήριξη αυτού του feature είναι σαφώς ευκολότερο να υποστηριχθεί το migration των containers, καθώς αποφεύγονται πιθανά path conflicts που μπορεί να δημιουργηθούν μεταξύ του αρχικού και του τελικού συστήματος. Για παράδειγμα αν στο blkio υποσύστημα υπάρχει ένα foo cgroup που χρειάζεται για το confinement του container, αλλά στο target σύστημα ο αντίστοιχος φάκελος υπάρχει ήδη, χωρίς την υποστήριξη των cgroup namespaces θα έπρεπε να μετονομαστεί αυτό το cgroup, ενώ τώρα, αρκεί απλός να δημιουργηθεί ένα namespace.

2.3.3 cgroups (control groups)

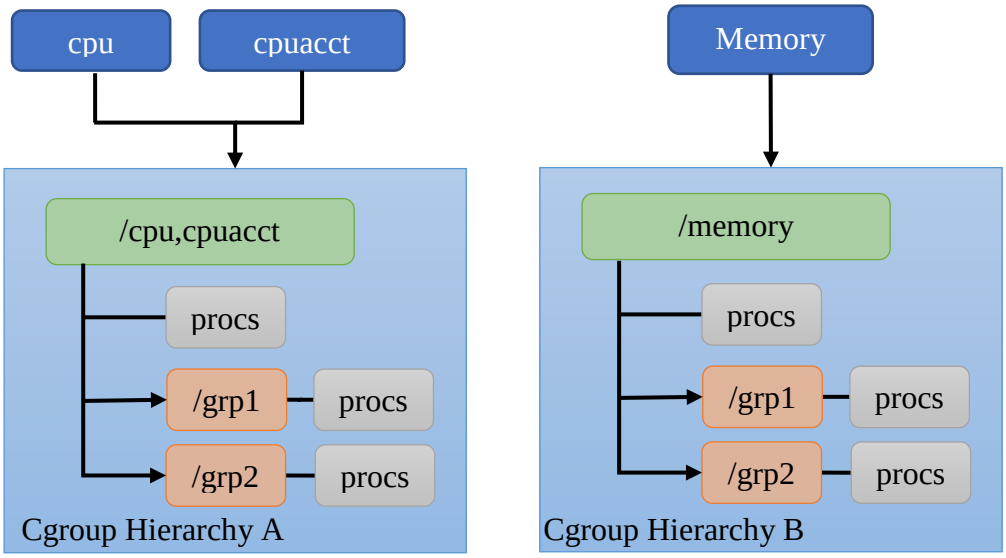
Τα cgroups (συντομογραφία των control groups) είναι ένα πρόσφατο σχετικά feature του πυρήνα των Linux που επιτρέπει την απομόνωση κάποιο πόρων του υπολογιστή, όπως CPU και μνήμη, για μία ομάδα από διεργασίες και γι' αυτό το λόγο χρησιμοποιείται συχνά, αν όχι πάντα, στη δημιουργία containers. Τα cgroups είχαν αρχικά ονομαστεί process containers, ωστόσο ενσωματώθηκαν στον στάνταρ πυρήνα στην έκδοση 2.6.24 (Ιανουάριος 2008) με το όνομα cgroups για να αποφευχθεί πιθανή σύγχυση του ονόματος, καθώς ο όρος containers χρησιμοποιούνταν σε πολλές περιπτώσεις. Πρόσφατα τα cgroups σχεδιάστηκαν (για δεύτερη φορά) εκ νέου και πλέον διακρίνονται στην έκδοση 1, η οποία είναι η έκδοση με την οποία εμφανίστηκαν στον πυρήνα 2.6.24, και στην έκδοση 2, η οποία εμφανίστηκε στην έκδοση 4.5 του πυρήνα, τον Μάρτιο του 2016.

Τα control group σχεδιάστηκαν με σκοπό να προσφέρουν ενιαίο τρόπο διαχείρισης των resources σε πολλές περιπτώσεις χρήσης, γι' αυτό και έχουν ευρύ φάσμα εφαρμογής. Μπορεί κανείς να δει τα cgroups να χρησιμοποιούνται για να ενισχύσουν την απομόνωση σε περιβάλλοντα εικονοποίησης επιπέδου λειτουργικού συστήματος, αλλά και σε περιπτώσεις περιορισμού του CPU priority μίας και μόνο διεργασίες (αντί την κλήση συστήματος nice). Μέσω των control groups μπορεί κανείς ανά ομάδα διεργασιών να ορίσει όρια (hard limits) στη χρήση ορισμένων πόρων του συστήματος, να δώσει προτεραιότητα σε αυτούς (π.χ. για I/O throughput του δίσκου), να παρακολουθήσει το ποσοστό χρήσης των πόρων του κάθε υποσυστήματος του λειτουργικού και τέλος να ελέγξει την ομάδα διεργασιών (πάγωμα και συνέχιση εκτέλεσης, επανεκκίνηση και τέλος checkpointing). Μερικά από τα υποσυστήματα που υποστηρίζουν τα cgroups είναι τα ακόλουθα:

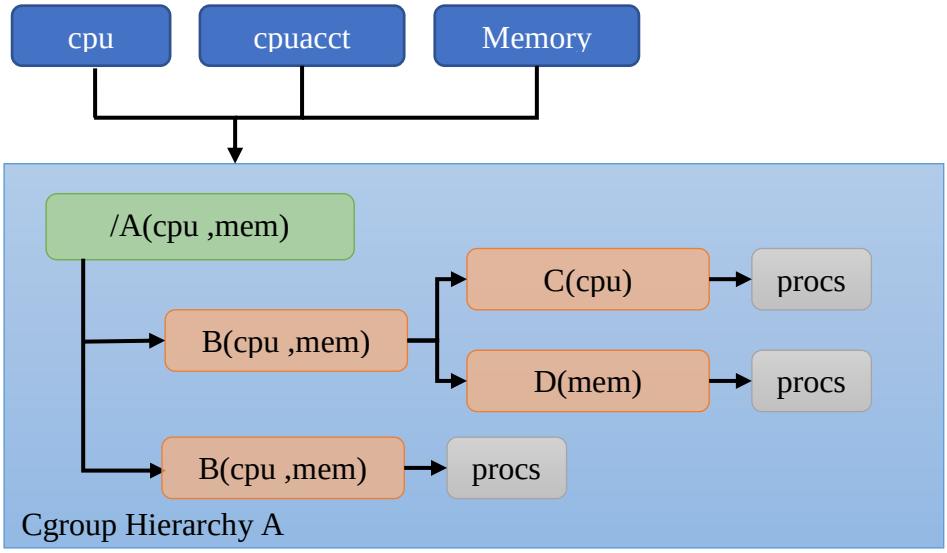
- cpu: Χρησιμοποιεί τον χρονοδρομολογητή των Linux για να περιορίζει την πρόσβαση των διεργασιών στη CPU
- cpuset: Ορίζει φυσικά CPU cores και κόμβους μνήμης ανά γκρουπ διεργασιών
- cpuacct: Παρακολουθεί και καταγράφει την χρήση CPU
- memory: Θέτει όρια χρήσης της μνήμης και παρακολουθεί τη χρήση της
- freezer: Σταματά και συνεχίζει την εκτέλεση των διεργασιών του cgroup
- blkio: Θέτει όρια στο I/O των block devices
- net_prio: Ορίζει την προτεραιότητα της κίνησης του δικτύου για κάθε δικτυακό interface.
- devices: Ελέγχει (επιτρέπει ή όχι) την πρόσβαση των διεργασιών στις συσκευές του δικτύου.

Τα control groups μοιάζουν με τις διεργασίες των Linux συστημάτων, στο ότι είναι ιεραρχικά και στο ότι κληρονομούν ορισμένες ιδιότητες από το πατρικό cgroup. Ωστόσο, τα cgroups έκδοσης 1 διαφέρουν από τις διεργασίες, καθώς οι διεργασίες όλες (εκτός από την init) έχουν κοινό πρόγονο τη διεργασία init με pid 1, άρα η ιεραρχία τους σχηματίζει ένα δέντρο, αλλά, αντίθετα, τα cgroups μπορεί να σχηματίζουν πολλές ιεραρχίες ταυτόχρονα σε ένα σύστημα και συνεπώς πολλά δέντρα. Αυτό συμβαίνει καθώς για κάθε υποσύστημα των Linux που διαχειρίζονται τα cgroups, πρέπει να υπάρχει μία νέα ιεραρχία από control groups. Δηλαδή ένα δέντρο από cgroups έκδοσης 1 διαχειρίζεται κάθε φορά μόνο έναν πόρο του συστήματος από αυτά που περιγράψαμε

παραπάνω, όπως CPU, μνήμη, freezer κλπ. Ο παραπάνω περιορισμός δεν ισχύει στα cgroups έκδοσης 2, τα οποία μπορούν να διαχειριστούν πολλά υποσυστήματα ταυτόχρονα και συνεπώς χρειάζεται η ύπαρξη ενός και μόνο δέντρου (βλ εικόνα 8 και 9 για σύγκριση των δύο εκδόσεων) [32]. Σε περίπτωση που ο ελεγκτής κάποιου υποσυστήματος υποστηρίζει μόνο έκδοση 1 (ή ο χρήστης του συστήματος επιθυμεί να το χρησιμοποιήσει στην legacy έκδοση), τότε αυτόματα γίνεται mounted στον κατάλογο ρίζα των cgroups ως legacy δέντρο ώστε να υπάρχει συμβατότητα προς τα πίσω. Η ιεραρχία των cgroups δίνει ευελιξία, αλλά και απομόνωση, καθώς ένα πατρικό cgroup μπορεί να επηρεάσει παραμέτρους των παιδιών του, αλλά το ανάποδο είναι αδύνατο. Επίσης, η δημιουργία ενός child cgroup μπορεί μόνο να έχει τα ίδια ή λιγότερα resources από αυτά του πατέρα. Τέλος αξίζει να τονιστεί ότι μία διεργασία μπορεί να βρίσκεται μόνο σε ένα cgroup στην κάθε ιεραρχία, ή πιο σωστά στο κάθε υποσύστημα.



Εικόνα 13: Legacy Cgroup (έκδοση 1)



Εικόνα 14: Cgroups έκδοση 2

2.3.4 Capabilities

Τα capabilities είναι ένας σημαντικός μηχανισμός του πυρήνα, ο οποίος επιτρέπει τον διαχωρισμό των δικαιωμάτων του root χρήστη σε πολλές μικρότερες δυνατότητες, οι οποίες μπορούν να δοθούν σε διεργασίες ή σε εκτελέσιμα προκειμένου να επιτελέσουν διαχειριστικές ενέργειες. Τα capabilities εμφανίστηκαν για πρώτη φορά στον πυρήνα 2.2, ωστόσο στη συνέχεια εμπλουτίστηκαν αρκετά.

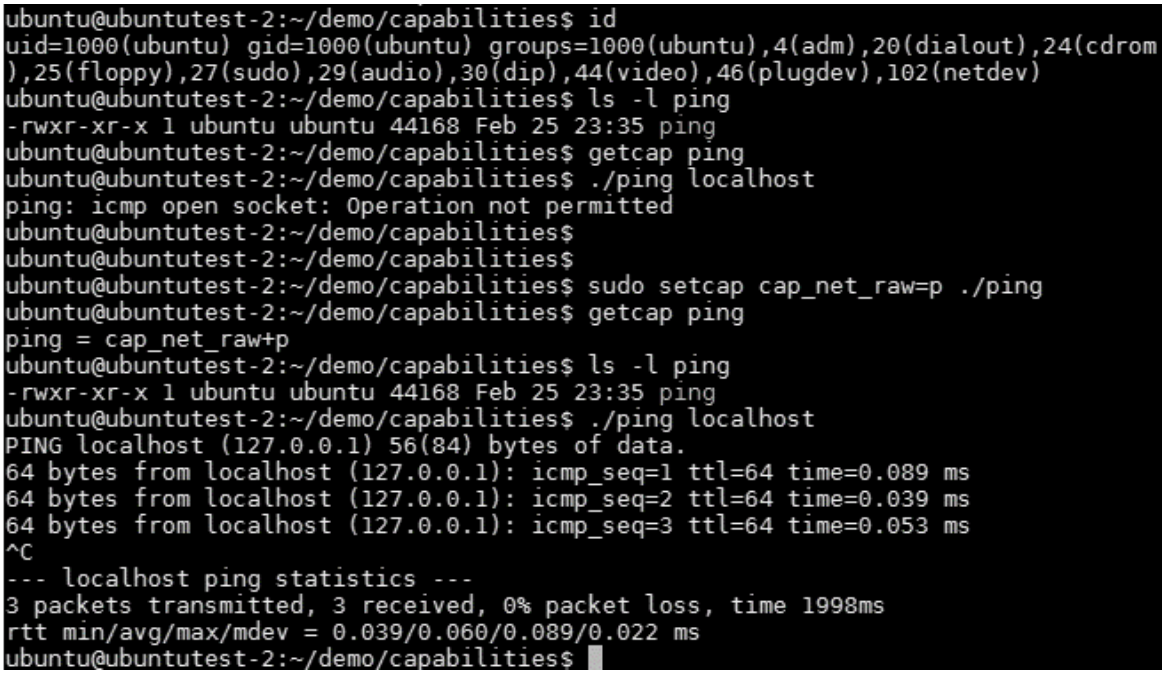
Πριν την υποστήριξη των capabilities από τα Linux, οι διεργασίες χωρίζονταν σε δύο κατηγορίες. Σε αυτές που έτρεχαν ως root (με uid 0) οι οποίες είχαν απεριόριστες δυνατότητες στο λειτουργικό και στις υπόλοιπες (με uid μεγαλύτερο από 0), οι οποίες μπορούσαν να εκτελούν μόνο unpriviledged εργασίες. Ο συνηθισμένος τρόπος που ακολουθούνταν, ώστε οι χρήστες χωρίς διαχειριστικά δικαιώματα να μπορούν να εκτελέσουν priviledged εργασίες ήταν μέσω των setuid και setgid bits. Μέσω αυτού του μηχανισμού ένα εκτελέσιμο μπορούσε να τρέξει από οποιονδήποτε χρήστη σαν να ήταν ο ιδιοκτήτης του αρχείου. Το σύνηθες σενάριο, λοιπόν, σε περίπτωση που ένα πρόγραμμα χρειαζόταν να εκτελεστεί με προσωρινά elevated priviledges, ήταν να ανήκει στον διαχειριστή και τρέχει με ενεργοποιημένο το setuid flag. Κατ' αυτόν τον τρόπο αφού εκτελούνταν τα απαραίτητα priviledged operations, μπορούσε να αλλάξει το euid (effective uid) μέσω της κλήσης συστήματος setuid ρίχνοντας όλα τα root δικαιώματα που είχε. Για παράδειγμα, προκειμένου να εκτελεστεί ένας web server, όπως ο Apache, από έναν συνηθισμένο χρήστη, θα ενεργοποιούνταν το setuid flag ώστε να μπορέσει να ανοίξει ένα socket στην πόρτα 80 (το οποίο είναι priviledged operation) και στη συνέχεια θα έριχνε όλα τα δικαιώματα αλλάζοντας το effective uid του σε αυτό του χρήστη.

Στα παραπάνω το πρόβλημα είναι ότι ο server θα έχει όλα τα διαχειριστικά δικαιώματα, ενώ στην πραγματικότητα χρειάζεται ένα μικρό υποσύνολο αυτών. Έτσι αφήνει ανοιχτό σε επιθέσεις ένα μεγαλύτερο παράθυρο από αυτό που θα έπρεπε. Αυτό ακριβώς έρχονται να λύσουν τα capabilities, χωρίζοντας τα δικαιώματα του root χρήστη σε μικρότερα σαφώς καθορισμένα τμήματα, τα capabilities. Στην ουσία θα μπορούσε κανείς να τα κατανοήσει ως tokens, τα οποία δίνονται στις διεργασίες προκειμένου να έχουν πρόσβαση σε αυτές τις priviledged ενέργειες. Όταν, δηλαδή, μία διεργασία (ή πιο σωστά ένα thread) προσπαθεί να εκτελέσει μία τέτοια ενέργεια, ο πυρήνας επιτρέπει στην διεργασία να συνεχίσει την εκτέλεσή της μόνο εφόσον έχει το κατάλληλο token, εκτός και αν έχει euid 0, οπότε και παρακάμπτει όλους τους ελέγχους του πυρήνα. Μερικά σημαντικά capabilities είναι τα ακόλουθα:

- CAP_CHOWN: Επιτρέπει την αυθαίρετη αλλαγή των UIDs και GIDs ενός αρχείου
- CAP_DAC_OVERRIDE: Παρακάμπτει τους ελέγχους read, write και execute ενός αρχείου.
- CAP_KILL: Παρακάμπτει τον έλεγχο για την αποστολή σημάτων σε διεργασίες
- CAP_MKNOD: Επιτρέπει την δημιουργία συσκευών μέσω της κλήσης mknod
- CAP_NET_ADMIN: Δίνει πρόσβαση σε διάφορες διαχειριστικές ενέργειες που σχετίζονται με το δίκτυο
- CAP_NET_RAW: Δίνει τη δυνατότητα χρήσης RAW και PACKET socket
- CAP_SETFCAP: Επιτρέπει την αλλαγή των file capabilities
- CAP_SYS_ADMIN: Δίνει διάφορα διαχειριστικά δικαιώματα, όπως τη δυνατότητα δημιουργίας νέων namespace.
- CAP_SYS_CHROOT: Επιτρέπει την εκτέλεση της chroot
- CAP_SYS_PTRACE: Δίνει τη δυνατότητα tracing οποιασδήποτε διεργασίας
- CAP_SYS_TIME: Επιτρέπει την αλλαγή της ώρας του συστήματος

Συνεπώς βλέπουμε ότι τα capabilities προσφέρουν (στις περισσότερες περιπτώσεις) fine-grained έλεγχο των διαχειριστικών δικαιωμάτων, το οποίο είναι αδύνατο μέσω του μηχανισμού setuid. Προκειμένου να υπάρξει ευελιξία στον τρόπο με τον οποίο κληρονομούνται και κληροδοτούνται τα capabilities από διεργασία σε διεργασία, ο πυρήνας των Linux υποστηρίζει διάφορα flag sets, τα οποία τελικά καθορίζουν εάν μία διεργασία έχει ή όχι ένα capability. Τα flag sets αυτά είναι τα ακόλουθα: permitted, inheritable, effective, bounding και (πρόσφατα) ambient set [33]. Επίσης από τον πυρήνα 2.6.24 και μετά υποστηρίζονται τα file capabilities, τα οποία ελέγχουν τα capabilities που μπορεί να έχει ένα εκτελέσιμο αρχείο (εδώ υποστηρίζονται τα flag sets permitted και inheritable, καθώς και το effective bit το οποίο καθορίζει αν τα permitted capabilities του αρχείου θα μεταφερθούν στο effective κατά την εκτέλεση μίας κλήσης execve).

Στη συνέχεια ακολουθεί ένα απλό παράδειγμα, ώστε να κατανοήσουμε την χρήση και τις δυνατότητες που μας δίνουν τα capabilities:



Εικόνα 15: Παράδειγμα χρήσης των capabilities

Στο παραπάνω παράδειγμα προσπαθούμε να εκτελέσουμε το εργαλείο ping με έναν χρήστη που δεν έχει διαχειριστικά δικαιώματα (με uid 1000). Παρατηρούμε ότι το εργαλείο ping αδυνατεί να εκτελεστεί, παρόλο που ανήκει στον χρήστη που το εκτελεί, κι αυτό, διότι, δεν έχει το δικαίωμα να ανοίξει icmp socket. Το πρόβλημα αυτό, μπορεί εύκολα να λυθεί δίνοντας στο εκτελέσιμο αρχείο το CAP_NET_RAW capability (το capability μπορεί να το δώσει μόνο κάποιος priviledged χρήστης, γι' αυτό και στο παράδειγμα η εκτέλεση του εργαλείου setcap γίνεται με sudo). Στη συνέχεια έχοντας δώσει το capability, ο χρήστης μπορεί να εκτελέσει το εργαλείο ping, όπως φαίνεται παραπάνω.

2.3.5 Seccomp filters

Τα seccomp (SECure COMPuting) φίλτρα έρχονται να συμπληρώσουν τα capabilities, επιτρέποντας σε μία διεργασία να εκτελεί μόνο συγκεκριμένες κλήσεις συστήματος. Παρόλο που τα capabilities μπορούν να περιορίσουν ή να δώσουν πρόσβαση σε συγκεκριμένες κλήσεις συστήματος, δεν είναι δυνατή η fine-grained προσέγγιση των κλήσεων συστήματος, κενό το οποίο έρχονται να καλύψουν τα seccomp φίλτρα. Συνεπώς είναι ιδιαίτερα χρήσιμο feature σε περίπτωση που απαιτείται η εκτέλεση κάποιου άγνωστου εκτελέσιμου, γι' αυτό και χρησιμοποιείται από τους φυλλομετρητές Chromium και Chrome. Η seccomp κλήση συστήματος μπήκε για πρώτη φορά στον mainline πυρήνα στην έκδοση 2.6.12 (Μάρτιος 2005).

Η πρώτη έκδοση που υλοποιήθηκε (η οποία τώρα υποστηρίζεται μέσω του SECCOMP_SET_MODE_STRICT operation), γνωστή και ως mode 1 seccomp, επέτρεπε στην καλούσα διεργασία (ή thread) να εκτελέσει μόνο τις ακόλουθες τέσσερις κλήσεις συστήματος: read, write, _exit και sigreturn. Στην περίπτωση που η διεργασία

προσπαθήσει να καλέσει κάποια κλήση συστήματος εκτός από τις παραπάνω, ο πυρήνας της στέλνει SIGKILL.

Η δεύτερη έκδοση (SECCOMP_SET_MODE_FILTER operation), η οποία και χρησιμοποιείται περισσότερο, επιτρέπει τον ορισμό συνόλων από κλήσεις συστήματος και των αντίστοιχων arguments (προαιρετικά) με την βοήθεια των BPF (Berkley Packet Filter), τα οποία, αν και σχεδιασμένα για το δίκτυο, θεωρούνται αποδοτική και δοκιμασμένη υλοποίηση. Ο χρήστης μπορεί να ορίσει ποια από αυτά τα σύνολα από system calls θέλει να επιτρέπονται και ποια όχι, προσφέροντας έτσι τόσο τη δυνατότητα whitelisting, όσο και τη δυνατότητα blacklisting των κλήσεων συστήματος (κάτι που αδυνατούν να προσφέρουν τα capabilities). Πιο συγκεκριμένα υποστηρίζονται πέντε φίλτρα, βάση των οποίων αποφασίζεται η ενέργεια που θα εκτελέσει ο πυρήνας μόλις ζητηθεί η εκτέλεση μίας κλήσης συστήματος: SCMP_ACT_ALLOW (οι κλήσεις που τοποθετούνται σε αυτό το φίλτρο επιτρέπονται), SCMP_ACT _KILL (εάν γίνει κληθεί κάποια κλήση συστήματος από αυτό το σύνολο, τότε η διεργασία σκοτώνεται), SCMP_ACT_TRAP (δίνεται η δυνατότητα χειρισμού της κλήσης), SCMP_ACT_ERRNO (επιστρέφει συγκεκριμένο error number) και SCMP_ACT_TRACE (ειδοποιεί τις διεργασίες που κάνουν trace με PTRACE_O_SECCOMP το thread όταν γίνει κλήση συστήματος που περιέχεται σε αυτό το φίλτρο). Δίνεται έτσι σημαντική ευελιξία στον περιορισμό των κλήσεων συστήματος μίας διεργασίας, αλλά και στη διαχείρισή και την παρακολούθησή τους σε περίπτωση που ζητηθεί η εκτέλεση μίας μη αναμενόμενης κλήσης συστήματος.

2.3.6 Συστήματα Υποχρεωτικού Ελέγχου - MAC

Τα παραπάνω εργαλεία μπορούν να χρησιμοποιηθούν για την κατασκευή και το hardening των containers, προσφέροντας ένα ασφαλές και απομονωμένο περιβάλλον εκτέλεσης. Ωστόσο, ακόμα και ένα τέτοιο περιβάλλον δεν έχει ολοκληρωμένη ασφάλεια, καθώς είναι εκτεθειμένο σε αδυναμίες που ανακαλύπτονται συχνά στον πυρήνα και οι οποίες είναι πολύ δύσκολο να εξαλειφθούν, αφού πάντα ελλοχεύει ο κίνδυνος του προγραμματιστικού λάθους. Ο μόνος τρόπος να καλυφθεί αυτό το κενό είναι μέσω των συστημάτων υποχρεωτικού ελέγχου, MAC (Mandatory Access Control).

Τα συστήματα MAC, ελέγχουν και περιορίζουν την πρόσβαση και τη δυνατότητα που έχει ένας χρήστης ή μια διεργασία να προκαλέσουν αλλαγές ή να εκτελέσουν ενέργειες σε ορισμένα τμήματα του συστήματος. Για παράδειγμα, μπορούν να καθορίσουν τους χρήστες που έχει πρόσβαση σε συγκεκριμένα αρχεία του συστήματος (ανεξάρτητα από το DAC του λειτουργικού), τους χρήστες που έχουν τη δυνατότητα να διαχειρίζονται τα interfaces του δικτύου, να ανοίγουν TCP sockets ή να χρησιμοποιούν τους IPC μηχανισμούς του λειτουργικού. Ο έλεγχος πρόσβασης στα συστήματα υποχρεωτικού ελέγχου είναι κεντρικός και επιβάλλεται με τον ορισμό πολιτικών (policies) από τον διαχειριστή του συστήματος. Οι χρήστες του συστήματος δεν έχουν την δυνατότητα μεταβολής των πολιτικών αυτών. Συνεπώς, ένα σύστημα MAC παρόλο που μπορεί να ελέγχει την πρόσβαση στα αρχεία του συστήματος διαφέρει σημαντικά από το DAC (discretionary access control) που χρησιμοποιούν πολλά UNIX συστήματα για τον έλεγχο των δικαιωμάτων των χρηστών και των group,

καθώς στο DAC ένας χρήστης μπορεί να αλλάξει τις προσβάσεις που έχει ο ίδιος η ακόμα και οι άλλοι χρήστες (για παράδειγμα μπορεί να δώσει η να αφαιρέσει δικαιώματα εκτέλεσης ή τροποποίησης ενός αρχείου). Για το λόγο αυτό σε ένα MAC σύστημα ο διαχειριστής του συστήματος επιβάλει μία πολιτική και το MAC εγγυάται την επιβολή της πολιτικής αυτής, χωρίς να υπάρχει κίνδυνος κάποιος χρήστης να την τροποποιήσει (είτε κατά λάθος είτε επίτηδες).

Τα MAC συστήματα σχετίζονται με την φιλοσοφίας του multi-level security (MLS) βάση τις οποίας ένα σύστημα πρέπει να είναι σε θέση να διαχειρίζεται πληροφορίες και δεδομένα που έχουν διαφορετικές απαιτήσεις ασφάλειας, ανάγκη η οποία είχε προέλθει από στρατιωτικές κυρίως εφαρμογές. Για το λόγο αυτό τα πρώτα MAC (και τα πρώτα πρωτόκολλα ασφαλείας) είχαν σχεδιαστεί για να καλύψουν στρατιωτικές ανάγκες. Ωστόσο σήμερα τα MAC έχουν φύγει από αυτή τη φιλοσοφία και χρησιμοποιούνται κυρίως για την προστασία από δικτυακές επιθέσεις, malwares και vulnerabilities των συστημάτων. Συγκεκριμένα, μπορούν να περιορίσουν (ή και να εξαλείψουν) την ζημιά που προκαλείται από επιτυχημένες επιθέσεις σε εφαρμογές η το λειτουργικό, εμποδίζοντας τον επιτιθέμενο από το να αποκτήσει ανεβασμένα δικαιώματα στο μηχάνημα (εμποδίζουν το privilege escalation).

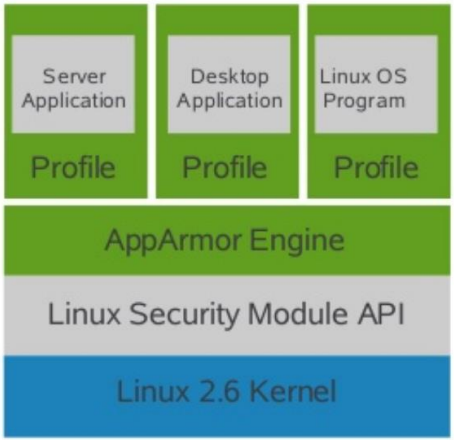
Στα Linux οι μηχανισμοί MAC μπορούν να υλοποιηθούν χρησιμοποιώντας το framework Linux Security Modules (LSM), το οποίο ενσωματώθηκε στον mainline πυρήνα από την έκδοση 2.6. Το LSM προσφέρει hooks σε διάφορα τμήματα του λειτουργικού (κυρίως σε κλήσεις συστήματος που προσπελαύνουν επικίνδυνες δομές δεδομένων του πυρήνα) που μπορούν να χρησιμοποιηθούν για διάφορα μοντέλα ασφάλειας, παρέχοντας έτσι υποστήριξη για πολυεπεξεργαστικά συστήματα[1]. Σήμερα έχουν υλοποιηθεί διάφορα συστήματα επιβολής υποχρεωτικού ελέγχου τα οποία χρησιμοποιούν το LSM για τους απαραίτητους ελέγχους. Μερικά από αυτά είναι τα AppArmor, SELinux, TOMOYO και SMACK. Δημοφιλής επίσης είναι το grsecurity patch του πυρήνα, το οποίο προσφέρει μία custom υλοποίηση MAC (και πιο συγκεκριμένα υλοποιεί ένα role-based access control –RBAC- σύστημα).

2.3.6.1 AppArmor

Το AppArmor είναι ένα MAC security module του πυρήνα Linux (ενσωματώθηκε στον mainline πυρήνα από την έκδοση 2.6.36), το οποίο σε πολλές διανομές είναι ενεργοποιημένο by default, και περιέχει ένα σύνολο από userspace εργαλεία για την διαχείριση και την παρακολούθησή του. Επιτρέπει την επιβολή μίας MAC πολιτικής μέσω profiles που ορίζονται ανά εκτελέσιμο, για αυτά που επιθυμεί να περιορίσει ο διαχειριστής του συστήματος. Τα προφίλ ορίζονται σε αρχεία κειμένου, το καθένα από τα οποία αντιστοιχίζεται συνήθως σε πολιτική ενός εκτελέσιμου αρχείου. Στη συνέχεια αυτά τα αρχεία μετατρέπονται σε binaries με τη βοήθεια ενός parser και φορτώνονται στον πυρήνα είτε κατά την εκκίνηση του υπολογιστή, είτε κατ' εντολή του διαχειριστή.

[1] Αντίθετα το Systrace utility, που χρησιμοποιούνταν τότε για να επιβάλλει security policies, χρησιμοποιούσε system call interposition, το οποίο ήταν ανοιχτό σε επιθέσεις TOCTTOU (time of check to time to use) εξαιτίας των race conditions που εμφανίστηκαν στα multiprocessor συστήματα.

Ένα profile ορίζει (κυρίως whitelist) πολιτικές που σχετίζονται με τα δικαιώματα του εκτελέσιμου πάνω σε αρχεία του συστήματος (τα δικαιώματα μπορεί να είναι ανάγνωση, εγγραφή, κλείδωμα, εκτέλεση και άλλα), πάνω σε τμήματα του δικτύου (π.χ. TCP/UDP κίνηση, συγκεκριμένο interface, socket), δυνατότητα αποστολής signals, tracing διεργασιών, χρήσης του D-Bus, mounting συστημάτων αρχείων και τέλος χρήσης συγκεκριμένων capabilities. Ακόμη, σε ένα προφίλ μπορούν να οριστούν οι συνθήκες κάτω από τις οποίες ένα εκτελέσιμο μπορεί να αλλάξει profiles (π.χ. λόγω execve, pivot_root κ.α.) επιτρέποντας πιο fine-grained έλεγχο. Τέλος, σημαντικό είναι πως το AppArmor μπορεί να διαχειριστεί αλλαγές στα path των αρχείων που συμβαίνουν λόγω pivot_root, chroot ή λόγω των mount namespaces, γεγονός που το διευκολύνει την χρήση του για hardening των containers.



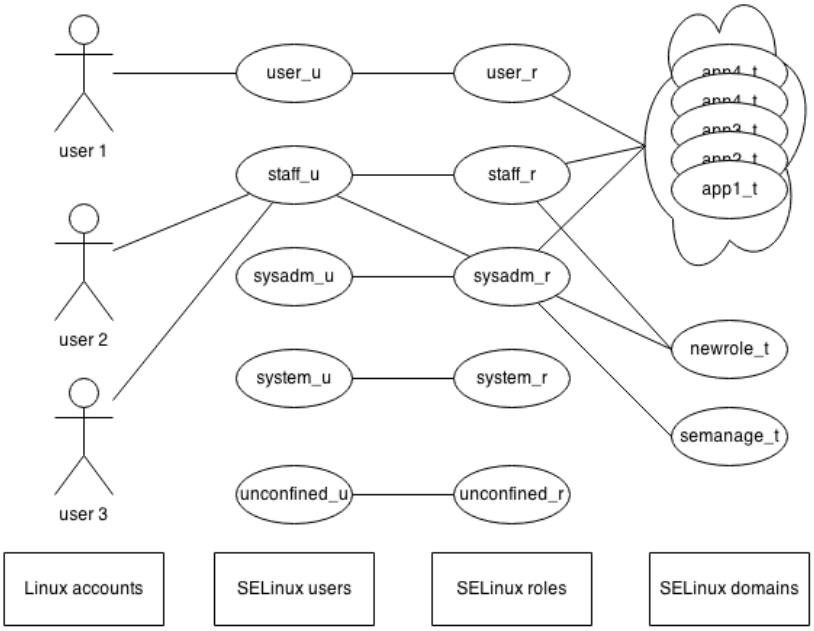
Εικόνα 16: Αρχιτεκτονική AppArmor

Το AppArmor επιτρέπει την επιβολή των πολιτικών είτε σε enforce, είτε σε complain mode. Στην πρώτη περίπτωση αν κάποιο πρόγραμμα προσπαθήσει να προσπελάσει πόρους που δεν επιτρέπονται από το profile, η αντίστοιχη κλήση συστήματος θα επιστρέψει σφάλμα access denied, ενώ στη δεύτερη η κλήση συστήματος θα επιστρέψει κανονικά καταγράφοντας την παράβαση της πολιτικής. Ωστόσο μπορεί κανείς μέσα στο ίδιο profile να χρησιμοποιήσει και τα δύο modes, γεγονός που δίνει αρκετή ευελιξία. Η καταγραφή των παραβάσεων μπορεί να χρησιμοποιηθεί είτε κατά τον σχεδιασμό της πολιτικής, είτε για παρακολούθηση των προσβάσεων ευαίσθητης πληροφορίας [34].

2.3.6.2 SELinux

Όπως και το AppArmor έτσι και το SELinux (Security-Enhanced Linux) είναι ένα LSM module του πυρήνα και περιλαμβάνει ένα σύνολο από userspace εργαλεία για τον ορισμό των πολιτικών και την παρακολούθηση αυτών. Το SELinux ενσωματώθηκε στον mainline πυρήνα στην έκδοση 2.6, ωστόσο ο σχεδιασμός του και οι πρώτες εκδόσεις είχαν ξεκινήσει αρκετά νωρίτερα από την NSA των ΗΠΑ με την μορφή ενός patch του πυρήνα.

Το SELinux προσφέρει μεγάλη ευελιξία εφαρμόζοντας μία λογική που συνδυάζει συστήματα RBAC, MIC (Mandatory integrity controls) και TE (type enforcement). Σε αντίθεση με το AppArmor, ορίζει για κάθε διεργασία, χρήστη και κάθε resource του συστήματος ένα context, το οποίο αποτελείται από τρία αναγνωριστικά: χρήστης (οι SELinux χρήστες διαφέρουν από τους χρήστες του λειτουργικού), ρόλος και τύπος (ή domain). Μία διεργασία μπορεί να χρησιμοποιήσει έναν πόρο του συστήματος ή ένα αρχείο μόνο αν έχει την ίδια τριπλέτα αναγνωριστικών (δηλαδή το ίδιο context) με αυτό. Συνεπώς προκειμένου να χρησιμοποιήσει πόρους που βρίσκονται σε διαφορετικά contexts θα πρέπει να αλλάξει το context της διεργασίας. Το αν επιτρέπεται μία αλλαγή context και οι συνθήκες που πρέπει να πληρούνται προκειμένου να γίνει αυτή η αλλαγή καθορίζονται από την εφαρμοζόμενη πολιτική. Επίσης η πολιτική περιέχει booleans που επιτρέπουν την ενεργοποίηση ή όχι ενός κανόνα, κατά το runtime. Παρακάτω φαίνεται μία λογική αναπαράσταση των SELinux contexts, όπου μπορούμε να δούμε ότι ένας χρήστης πιθανόν να συνδέεται με πολλά contexts, καθένα από τα οποία του δίνει επιπλέον δικαιώματα (για παράδειγμα ο χρήστης 2 μπορεί να αποκτήσει βάση του παρακάτω σχήματος πολλαπλά contexts π.χ. τα εξής: staff_u:staff_r:newrole_t, staff_u:staff_r:app1_t, staff_u:sysadm_r:newrole_t κ.ο.κ., καθένα από τα οποία πιθανόν του επιτρέπει πρόσβαση σε διαφορετικό σύνολο από resources).



Εικόνα 17: SELinux contexts

Συνεπώς βλέπουμε ότι παρόλο που και το Apparmor και το SELinux είναι συστήματα υποχρεωτικού ελέγχου, ακολουθούν σαφώς διαφορετική προσέγγιση στο πώς περιγράφουν τις πολιτικές και τις εφαρμόζουν. Αυτό έχει σαν αποτέλεσμα να είναι κατάλληλα για διαφορετικού τύπου εφαρμογής, αφού το AppArmor είναι πιο εύχρηστο, ενώ το SELinux επιτρέπει μεγαλύτερη ευελιξία, με το μειονέκτημα ότι σε πολλές περιπτώσεις για να υποστηριχθεί μία νέα εγκατάσταση (ή για να μοιραστούν εφαρμογές ένα νέο φάκελο) θα πρέπει να γίνουν αλλαγές στο labeling που θα επηρεάσουν πολλές πολιτικές του συστήματος καθιστώντας τη διαδικασία αυτή αργή και ευάλωτη σε λάθη.

2.4 Απειλές

Παρά τα εργαλεία που χρησιμοποιούνται για να εικονοποιήσουν διάφορα τμήματα του λειτουργικού και να προσφέρουν ένα απομονωμένο περιβάλλον, η ασφάλεια των containers απέχει αρκετά από την ασφάλεια που μπορούν να προσφέρουν οι εικονικές μηχανές. Είναι σημαντικό λοιπόν, πριν προχωρήσουμε, να εξετάσουμε τις επιθέσεις στις οποίες είναι εκτεθειμένος ένας container. Οι επιθέσεις αυτές συνήθως μπορούν να χωριστούν σε αυτές που στοχεύουν το host σύστημα και σε αυτές που στοχεύουν τη μηχανή των containers, και κατ' επέκταση τους containers που φιλοξενούνται από αυτή.

Η πρώτη κατηγορία επιθέσεων είναι η πιο επικίνδυνη. Στον πυρήνα των Linux, οποίος στην εικονοποίηση σε επίπεδο λειτουργικού είναι κοινός για όλους τους χρήστες, έχουν βρεθεί κατά καιρούς διάφορες αδυναμίες, μερικές από τις οποίες μπορούν να οδηγήσουν ακόμα και σε πλήρη έλεγχο του συστήματος. Μάλιστα πολλές από τις αδυναμίες που έχουν βρεθεί τα τελευταία χρόνια οφείλονται στα namespaces και στον νέο κώδικα που γράφτηκε[2] για την υποστήριξη της εικονοποίησης σε επίπεδο λειτουργικό και συνεπώς αυτές οι αδυναμίες μπορούν να εκμεταλλευτούν από έναν κακόβουλο χρήστη ο οποίος έχει τον έλεγχο ενός container. Επίσης, αρκετές επιθέσεις στοχεύουν στο να εμποδίσουν ή και να διακόψουν τις παρεχόμενες υπηρεσίες (DoS) στερώντας το υπόλοιπο σύστημα από πόρους (όπως μνήμη, CPU, σκληρός δίσκος, I/O, file descriptors, urandom device κλπ.). Τέλος μέσω του container μπορεί κανείς να μάθει αρκετές πληροφορίες για το host μηχάνημα, όπως να προσδιορίσει την πλατφόρμα container (π.χ. μέσω cgroups) που χρησιμοποιείται και την έκδοση του πυρήνα, τις οποίες μπορεί να αξιοποιήσει για να προχωρήσει την επίθεσή του.

Η δεύτερη κατηγορία επιθέσεων περιλαμβάνει, επίσης, αρκετές απειλές. Λόγω της μη επαρκούς απομόνωσης σε ορισμένες περιπτώσεις είναι δυνατόν κανείς να συλλέξει ευαίσθητες πληροφορίες (π.χ. λογαριασμούς χρηστών) για άλλους containers που τρέχουν στο ίδιο μηχάνημα (για παράδειγμα μέσω arp poisoning). Ένας ακόμη στόχος ενός επιτιθέμενου θα μπορούσε να είναι οι μοιραζόμενες βιβλιοθήκες και τα εκτελέσιμα αρχεία, τα οποία χρησιμοποιούνται συχνά για εξοικονόμηση αποθηκευτικού χώρου και μνήμης. Τέλος, καθώς η μηχανή που αρχικοποιεί και διαχειρίζεται τους containers είναι κοινή, αν ένας κακόβουλος container βρει αδυναμίες σε αυτή, θα μπορέσει να αποκτήσει πλήρη πρόσβαση σε όλους τους containers που αυτή διαχειρίζεται (π.χ. [35]).

Αξίζει να σημειωθεί ότι στις παραπάνω απειλές δεν συμπεριλάβαμε αυτές που σχετίζονται με συγκεκριμένες υλοποιήσεις που ακολουθούν οι διάφορες πλατφόρμες, ούτε τον τρόπο με τον οποίο αυτές λειτουργούν, καθώς αυτές οι απειλές δεν σχετίζονται με τους μηχανισμούς υποστήριξης των containers που προσφέρει ο πυρήνας (οι οποίοι παρουσιάστηκαν στο κεφάλαιο 2.4), αλλά με τους συγκεκριμένους τρόπους που χρησιμοποιεί η κάθε πλατφόρμα για να δημιουργεί, να πακετάρει και να εκκινεί τους containers, καθώς με το πώς εξουσιοδοτεί τους χρήστες της να κάνουν αυτές τις ενέργειες.

[2] Ωστόσο ακόμα και σε παλιό κώδικα βρίσκονται συχνά αδυναμίες. Χαρακτηριστικό παράδειγμα αποτελεί η αδυναμία CVE-2017-6074 [48].

3 DOCKER

3.1 Γενικά

Το Docker είναι μία πλατφόρμα διαχείρισης και εκτέλεσης containers ανοιχτού κώδικα, η οποία αναπτύσσεται ενεργά και υποστηρίζεται από πολλούς cloud providers και πλατφόρμες cloud ή πλατφόρμες υποδομής ως υπηρεσία (IaaS), όπως το Openstack Compute (Nova) . Παρόλο που είναι μία νέα πλατφόρμα (με πρώτη έκδοση στις 13 Μαρτίου το 2013 υποστηριζόμενη από την startup εταιρία Docker) σε σχέση με άλλες υλοποίησες των containers, όπως το LXC, είναι η πλέον χρησιμοποιούμενη πλατφόρμα, τόσο για προσωπική χρήση όσο και για παραγωγικά συστήματα, βάσει ερευνών του 2015 [36] και 2016 [37]. Μάλιστα από πολλούς θεωρείται ότι το Docker έπαιξε σημαντικό ρόλο στην αύξηση του ενδιαφέροντος για την τεχνολογία των Linux containers, αλλά και γενικότερα της εικονοποίησης σε λειτουργικό επίπεδο.

Σημαντικό ρόλο στην υιοθέτηση του Docker έπαιξαν οι ευκολίες που προσφέρει. Αρχικά το Docker διαθέτει ένα repository από έτοιμους containers, ή, κατά την ορολογία που χρησιμοποιείται στο Docker, εικόνες (images). Εκεί μπορεί κανείς να βρει μεγάλη ποικιλία από εφαρμογές έτοιμες προς εκτέλεση, όπως βάσεις δεδομένων, web servers ή κάποιο έτοιμο λειτουργικό (χρησιμοποιώντας πάντα τον πυρήνα του host), τα οποία μπορούν να εκκινήσουν με τη χρήση μίας εντολής. Επιπλέον, διευκολύνει κατά πολύ το πακετάρισμα μίας εφαρμογής, αφού παρέχονται τα κατάλληλα εργαλεία, ώστε να αποθηκευτούν οι κατάλληλες ρυθμίσεις, τα αρχεία και οι βιβλιοθήκες που χρειάζεται σε ένα και μόνο αρχείο, το οποίο ονομάζεται dockerfile. Μέσα σε ένα dockerfile περιέχονται οδηγίες προς το Docker για την κατασκευή του επιθυμητού image (όπως πακέτα λογισμικού, ρυθμίσεις δικτύου, ορισμός χρηστών), το οποίο στη συνέχεια μπορεί να εκτελεστεί δημιουργώντας ένα instance, δηλαδή ένα container. Αξίζει να σημειωθεί ότι το dockerfile είναι εύκολα μεταφέρσιμο, λόγω του μικρού μεγέθους, αφού περιέχει μόνο τις οδηγίες κατασκευής της εικόνας και όχι την ίδια την εικόνα. Ωστόσο αν κάποιος επιθυμεί μπορεί εύκολα να αποθηκεύσει σε κάποιο repository (είτε ιδιωτικό είτε δημόσιο) την εικόνα, ώστε να την μοιράζεται απευθείας με τους συνεργάτες του, γεγονός που είναι χρήσιμο σε περιπτώσεις που οι containers περιέχουν δεδομένα. Τέλος το Docker προσφέρει εργαλεία που βοηθούν στην διαχείριση πολλών μηχανημάτων ταυτόχρονα, ώστε να διευκολύνεται η εγκατάσταση του Docker σε πολλούς servers και η διαχείριση (δημιουργία, εκκίνηση, κλπ) των containers σε αυτούς, καθώς και το πακετάρισμα και η διαχείριση εφαρμογών που απαρτίζονται από πολλούς containers, ορίζοντας τις κατάλληλες παραμέτρους σε ένα απλό αρχείο αντίστοιχο του dockerfile.

3.2 Βασικές έννοιες και λειτουργία

Το Docker κατασκευάζει τον κάθε container από μία εικόνα (image). Οι εικόνες περιέχουν όλα τα αρχεία που χρειάζεται ένας container για να παρέχει τις βασικές του λειτουργίες. Συνήθως, λοιπόν περιέχει μία δομή καταλόγων παρόμοια με αυτή που συναντάμε σε ένα λειτουργικό σύστημα Linux (όπως φάκελοι bin, usr, etc, κλπ) , ώστε

να είναι εύκολο κανείς να την επεκτείνει τοποθετώντας μέσα τις εφαρμογές που θέλει. Οι εικόνες αυτές μπορεί να προέρχονται είτε από κάποιο Docker repository είτε από ένα dockerfile, όπως είδαμε παραπάνω.

Συνήθως οι containers κατασκευάζονται, εκτελούνται μία φορά και μετά διαγράφονται, καθώς ο χρόνος που απαιτεί η δημιουργία ενός νέου container είναι πολύ μικρός. Ωστόσο, επειδή για ορισμένες εφαρμογές είναι απαραίτητο να αποθηκεύονται κάποια δεδομένα, το Docker παρέχει τα volumes, τα οποία ουσιαστικά είναι ένας φάκελος mounted στον container ειδικά σχεδιασμένος για αποθήκευση δεδομένων. Τα volumes, επομένως, επιτρέπουν την ύπαρξη των containers μίας χρήσης, καθώς και την ύπαρξη μοιραζόμενου αποθηκευτικού χώρου μεταξύ των containers.

Εκτός από τα παραπάνω σημαντικό κομμάτι των cotnainers είναι η δικτύωση, αφού λίγες εφαρμογές πλέον τρέχουν σε απομόνωση. Το Docker, λοιπόν παρέχει αρκετές ευκολίες σε αυτόν τον τομέα. Ο κάθε container κατασκευάζεται by default σε ένα ξεχωριστό network namespace, το οποίο όμως συνδέεται με μία γέφυρα με όλους τους containers και το internet. Παρέχεται η δυνατότητα κατασκευής διαφορετικών δικτύων πέραν της γέφυρας που κατασκευάζεται με μία τυπική εγκατάσταση Docker. Κατ' αυτόν τον τρόπο ο χρήστης μπορεί να ορίσει ποιοι containers θα βρίσκονται στο ίδιο δίκτυο, ποιοι θα έχουν πρόσβαση στο εξωτερικό δίκτυο, αλλά και να κατασκευάσει διαφορετικού τύπου δίκτυα ανάλογα με τις ανάγκες του.

Στη συνέχεια θα παρουσιάσουμε τις βασικές εντολές του docker για την διαχείριση των containers και των images, καθώς πάνω σε αυτές θα βασιστεί το σύστημα αυτοματοποιημένης ασφάλειας που θα παρουσιάσουμε στη συνέχεια. Οι εντολές που ακολουθούν παρουσιάζονται με συντομία, χωρίς την πλήρη σύνταξη, καθώς σκοπός είναι βοηθήσουν τον αναγνώστη να αποκτήσει μία πρώτη επαφή με το Docker CLI Client:

- **docker pull IMAGE:** Κατεβάζει τοπικά την εικόνα από το repository
- **docker create IMAGE [COMMAND]**: Δημιουργεί έναν container βάση της εικόνας IMAGE και τροποποιεί τον container ώστε να εκκινεί με την εντολή COMMAND (μπορεί να είναι κάποιο εκτελέσιμο αρχείο, κάποιο script κλπ.)
- **docker start CONTAINER**: Εκκινεί ένα σταματημένο container.
- **docker run IMAGE [COMMAND]**: Δημιουργεί έναν container βάση της εικόνας IMAGE και τον εκκινεί με την εντολή COMMAND
- **docker stop CONTAINER**: Σταματά την εκτέλεση του container.
- **docker rm CONTAINER**: Διαγράφει τον container
- **docker ps**: Δείχνει τους containers που έχουν δημιουργηθεί και την κατάσταση τους
- **docker attach**: Συνδέει το τρέχον terminal με έναν container που εκτελείται
- **docker network create NETWORK**: Κατασκευάζει ένα δίκτυο, το οποίο μπορούν να χρησιμοποιήσουν οι containers για επικοινωνία
- **docker network connect NETWORK CONTAINER**: Συνδέει τον container στο συγκεκριμένο δίκτυο
- **docker volume create:** Δημιουργεί ένα volume

- **docker volume rm VOLUME:** Διαγράφει το συγκεκριμένο volume

Ακολουθεί ένα σύντομο παράδειγμα χρήσης του Docker. Ο container του παραδείγματος είναι βασισμένος σε ένα Ubuntu image και εκτελεί (σε interactive mode) το bash, όπως φαίνεται στην παρακάτω εικόνα:

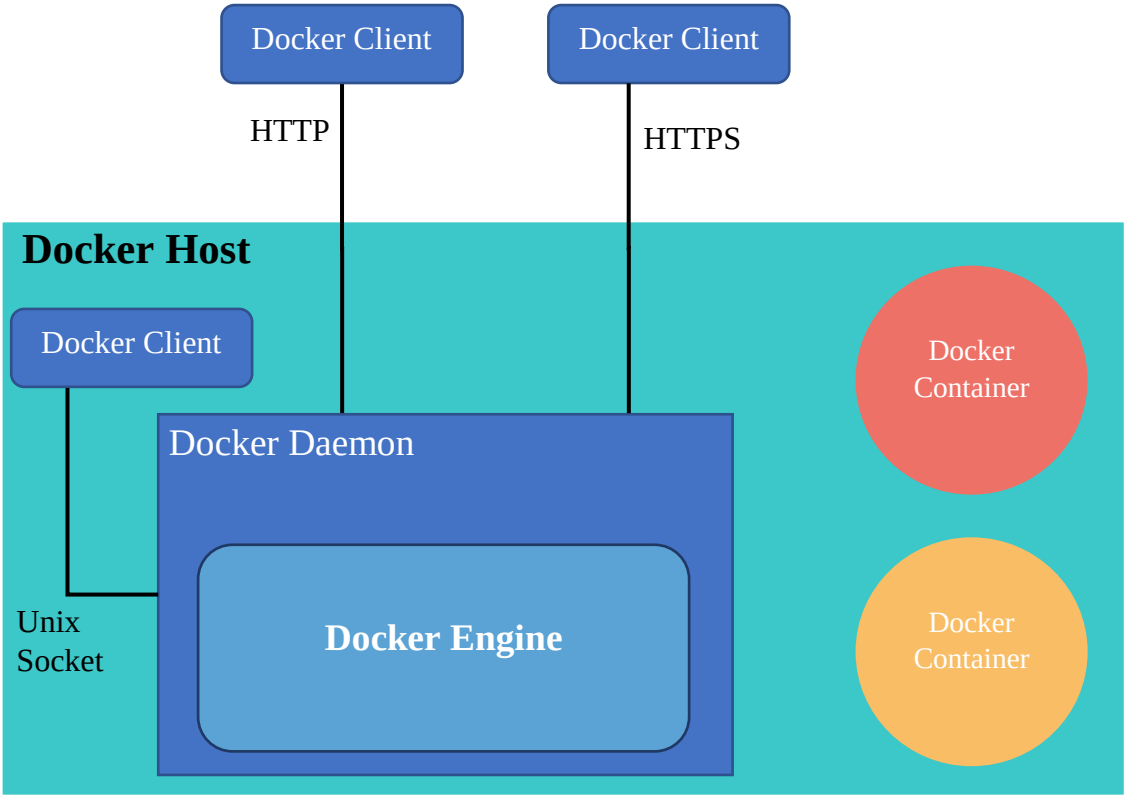
```
[22:11:19] root@dockerdemo:~$ docker create -it --name=Ubuntu-Demo ubuntu:16.04 /bin/bash
de13123ff057f20f17486c88d56be2ebf03f62ce2d05c758e3660a0d221916c9
[22:11:37] root@dockerdemo:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
 PORTS               NAMES
de13123ff057        ubuntu:16.04        "/bin/bash"         6 seconds ago
                     Ubuntu-Demo
[22:11:41] root@dockerdemo:~$ docker start Ubuntu-Demo
Ubuntu-Demo
[22:11:51] root@dockerdemo:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
 PORTS               NAMES
de13123ff057        ubuntu:16.04        "/bin/bash"         19 seconds ago      Up 3 seconds
                     Ubuntu-Demo
[22:11:54] root@dockerdemo:~$ docker attach Ubuntu-Demo
root@de13123ff057:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
root@de13123ff057:/# whoami
root
root@de13123ff057:/# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.1  0.0  18232  1996 ?        Ss   20:11   0:00 /bin/bash
root        12  0.0  0.0  34416  1456 ?        R+   20:12   0:00 ps aux
root@de13123ff057:/# [22:12:14] root@dockerdemo:~$
[22:12:14] root@dockerdemo:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
 PORTS               NAMES
de13123ff057        ubuntu:16.04        "/bin/bash"         42 seconds ago      Up 25 seconds
                     Ubuntu-Demo
[22:12:17] root@dockerdemo:~$ docker stop Ubuntu-Demo
Ubuntu-Demo
[22:12:24] root@dockerdemo:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
        PORTS               NAMES
de13123ff057        ubuntu:16.04        "/bin/bash"         51 seconds ago      Exited (0) 3 second
s ago                       Ubuntu-Demo
[22:12:26] root@dockerdemo:~$ docker rm Ubuntu-Demo
Ubuntu-Demo
[22:12:35] root@dockerdemo:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
 PORTS               NAMES
[22:12:38] root@dockerdemo:~$
```

Εικόνα 18: Παράδειγμα χρήσης Docker

3.3 Αρχιτεκτονική

Σε αυτό το σημείο αξίζει να δούμε την δομή ενός Docker συστήματος και πώς διαχειρίζεται βασικά τμήματα των containers. Παρόλο που έχουν αναπτυχθεί πολλά Docker εργαλεία γύρω από τους containers, εμείς θα ασχοληθούμε με τα βασικά συστατικά μίας τυπικής εγκατάστασης του Docker, όπως αυτή περιέχεται στο repository πολλών Linux διανομών. Σε μία τέτοια περίπτωση μπορούμε να διακρίνουμε τα εξής components του Docker:

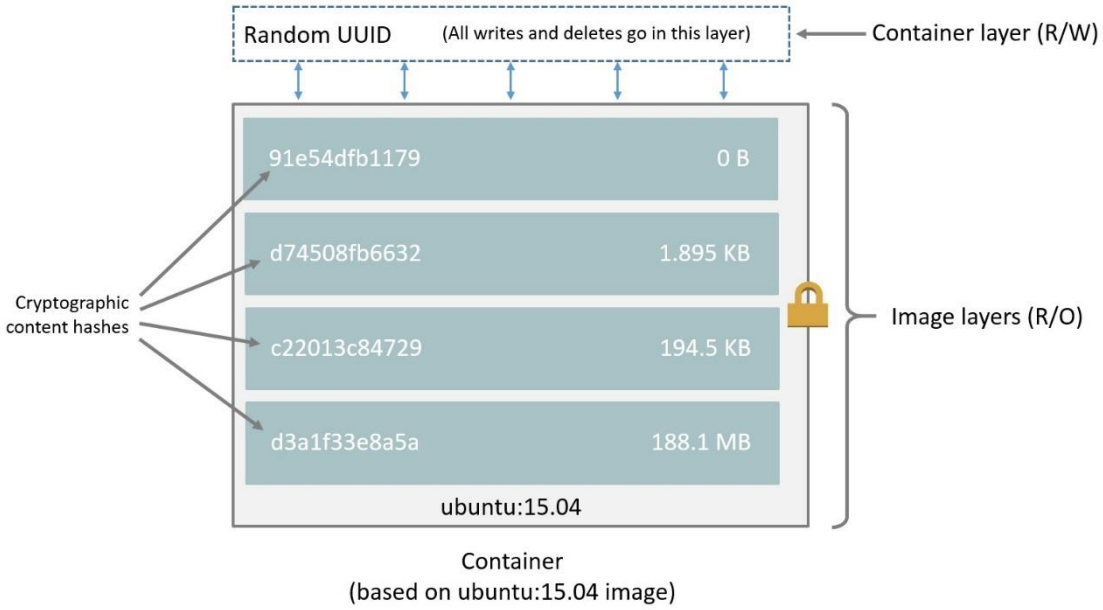
- **Docker Client**: Είναι η εφαρμογή που χρησιμοποιεί ο χρήστης προκειμένου να διαχειριστεί τους containers, τις εικόνες, τα volumes και τα απαραίτητα δικτυακά κομμάτια των containers. Συνήθως, ως Docker Client χρησιμοποιείται το command line εργαλείο *docker* το οποίο επικοινωνεί με τον Docker Daemon, μέσω UNIX socket, ωστόσο ο client μπορεί να βρίσκεται σε κάποιον απομακρυσμένο υπολογιστή και να χρησιμοποιεί HTTP ή HTTPS πρωτόκολλο για την επικοινωνία αυτή.
- **Docker Daemon**: Ο δαίμονας αυτός παρέχει ένα REST interface για την διαχείριση των containers, και επιτρέπει στου πελάτες να συνδέονται είτε μέσω UNIX sockets είτε δικτυακά. Μέσω του Docker Daemon παρέχεται όλη την απαιτούμενη λειτουργικότητα, όπως η διαχείριση και η παρακολούθηση των containers.
- **Docker Engine**: Είναι το κομμάτι που αναλαμβάνει την εκτέλεση όλων των λειτουργιών που σχετίζονται με την διαχείριση των images, των containers, του δικτύου μεταξύ τους και των volumes. Αξίζει να σημειωθεί ότι προσφέρει και αυτή ένα REST API, το οποίο μπορεί να χρησιμοποιηθεί για διαχείριση του Docker μέσα από custom εφαρμογές. Η υλοποίηση του Docker Engine έχει γίνει σε Go και έχει υποστεί σημαντικές αλλαγές στον τρόπο διαχείρισης των containers. Αρχικά, χρησιμοποιούσε LXC (ως execution driver) για να αλληλοεπιδρά με τα χαμηλού επιπέδου τμήματα του πυρήνα, όπως τα cgroups και τα namespaces, ενώ στην έκδοση 0.9 (Μάρτιος του 2014) χρησιμοποιήθηκε μία custom βιβλιοθήκη, η libcontainer [38], γραμμένη επίσης σε golang. Από την έκδοση 1.11 μέχρι σήμερα χρησιμοποιείται το εργαλείο runC, το οποίο αναπτύσσεται ανεξάρτητα από το Docker (αν και η αρχή του runC έγινε με δωρεά από το Docker της βιβλιοθήκης libcontainer). Στόχος των παραπάνω αλλαγών, ήταν να γίνει περισσότερο modular και platform-agnostic το Docker. Πιο συγκεκριμένα το Docker Engine προσφέρει τη λειτουργικότητα σχετικά με τα images, αλλά οτιδήποτε σχετίζεται με τους containers παρέχεται μέσω του runC, δηλαδή δεν αναλαμβάνει την εικονοποίηση λειτουργικού επιπέδου, καθώς υπάρχουν διαφορετικοί μηχανισμοί σε κάθε λειτουργικού και αντ' αυτού χρησιμοποιεί κατάλληλους execution driver.



Εικόνα 19: Τυπική αρχιτεκτονική ενός Docker Host

Στη συνέχεια θα εξετάσουμε τον τρόπο με τον οποίο το Docker αποθηκεύει τα images και τους containers, καθώς η οργάνωση του συστήματος αρχείων που χρησιμοποιεί έχει σημαντικές επιπτώσεις και στον τομέα της ασφάλειας. Προκειμένου να εξοικονομηθεί αποθηκευτικός χώρος οι εικόνες που χρησιμοποιούνται για την κατασκευή των containers αποθηκεύονται σε στρώματα (layers). Τα layers των images είναι διαθέσιμα στον χρήστη μόνο προς ανάγνωση (read-only) και συνεπώς όταν δημιουργείται ένας container πάνω από τα στρώματα του image, τοποθετείται ένα λεπτό στρώμα με δυνατότητες ανάγνωσης και εγγραφής, στο οποίο αποθηκεύονται κάποιες απαραίτητες ρυθμίσεις και αρχεία για την εκκίνηση του container, αλλά και οι αλλαγές που γίνονται εντός του container (όπως η δημιουργία νέων αρχείων, αλλαγές σε υπάρχοντα κλπ) μέσω της τεχνικής COW (Copy-on-write). Κατ' αυτόν τον τρόπο τα layers μπορούν να χρησιμοποιηθούν από πολλά images και πολλούς containers ταυτόχρονα, αλλά ταυτόχρονα κάθε container είναι ανεξάρτητος από τους άλλους, αφού οι αλλαγές αποθηκεύονται σε διαφορετικά R/W layers για κάθε container. Επίσης το R/W στρώμα επιτρέπει την αποθήκευση της κατάστασης των containers και, συνεπώς, παρέχονται δυνατότητες παύσης και εκκίνησης της λειτουργίας του container. Κάθε layer (από την έκδοση 1.10 και μετά), εκτός από τα R/W layers, συνοδεύεται από ένα κρυπτογραφικό hash, το οποίο χρησιμεύει αφενός σαν αναγνωριστικό, αλλά και σαν τρόπος ανίχνευσης πιθανών τροποποιήσεων αυτών των layers. Χρησιμοποιώντας αυτό το αναγνωριστικό το Docker Engine κατεβάζει τις ζητούμενες εικόνες ανά layer ελέγχοντας αν υπάρχει ήδη αποθηκεμένο αυτό το layer, κερδίζοντας κατ' αυτόν τον τρόπο σημαντικό χρόνο σε περιπτώσεις που οι εφαρμογές

χρησιμοποιούν τις ίδιες εικόνες ως βάση. Τέλος, αξίζει να σημειωθεί ότι το Docker παρέχει την παραπάνω λειτουργικότητα μέσα από διάφορους storage drivers καθένας από τους οποίους υποστηρίζει διαφορετικό σύστημα αρχείων, όπως τα AUFS, το οποίο είναι το default, OverlayFS, BTRFS και ZFS. Τα παραπάνω συστήματα αρχείων παρέχουν διαφορετικά features και συνεπώς είναι κατάλληλα για διαφορετικού τύπου εργασίες. Για παράδειγμα τα πρώτα δύο ανήκουν στην κατηγορία των Union file συστημάτων, ενώ τα δύο τελευταία παρέχουν τη δυνατότητα snapshotting, provisioning χώρου και copy-on-write.



Εικόνα 20: Layers μίας Docker εικόνας και ενός αντίστοιχου container

3.4 Σύγκριση με άλλες πλατφόρμες container

Είναι σημαντικό να κατανοήσουμε τα σημεία στα οποία διαφέρει το Docker σε σχέση με διαφορετικές υλοποιήσεις των containers. Το Docker σε αντίθεση με άλλες τεχνολογίες εικονοποίησης επιπέδου λειτουργικού, ευνοεί την κατασκευή container μέσα στις οποίες εκτελείται μόνο μία διεργασία. Η αρχική φιλοσοφία πίσω με την οποία σχεδιάστηκε το Docker ήταν η δυνατότητα δημιουργίας lightweight containers, καθένας από τους οποίους θα εκτελέσου όσο το δυνατόν λιγότερη εργασία. Σε περιπτώσεις όπου για να παραχθεί μία υπηρεσία χρειάζονταν περισσότερες από μία εφαρμογές (όπως π.χ. μία βάση δεδομένων και ένας web server), τότε θα έπρεπε κανείς να δημιουργήσει πολλού containers οι οποίοι θα επικοινωνούν μεταξύ τους. Συνεπώς, συνήθως σε ένα Docker container, δεν υπάρχουν υπηρεσίες που συναντά κανείς σε ένα συνηθισμένο λειτουργικό σύστημα, όπως logging, ssh και cron jobs, και ο χρήστης παροτρύνεται να χρησιμοποιεί τα εργαλεία που προσφέρει η πλατφόρμα για αυτές τις λειτουργίες. Αντίθετα οι περισσότερες πλατφόρμες για containers, όπως το LXC και το rkt, προσφέρουν ένα πλουσιότερο εικονοποιημένο περιβάλλον που προσεγγίζει κατά πολύ αυτό που προσφέρει μία εικονική μηχανή. Με αυτόν τον τρόπο μπορεί

κανείς να εγκαταστήσει και να ρυθμίσει εφαρμογές μέσα σε LXC containers όπως ακριβώς θα έκανε σε ένα φυσικό μηχάνημα ή σε μία εικονική μηχανή.

Μία άλλη διαφορά είναι ότι το Docker αποθηκεύει τις εικόνες και τους containers σε layers, ενώ άλλα συστήματα αποθηκεύουν τα απαραίτητα αρχεία σε φακέλους του host λειτουργικού, παρέχοντας ωστόσο υποστήριξη για layers εάν το επιθυμεί ο χρήστης (π.χ. για αντιγραφή ενός υπάρχοντος container).

Σημαντικές είναι και οι διαφορές στη δικτύωση. Το Docker παρέχει απλό interface για την διασύνδεση των containers μεταξύ τους, με το host μηχάνημα και με το internet, παρέχοντας την δυνατότητα σύνδεσης του σε συγκεκριμένα bridges και αναλαμβάνοντας τις ρυθμίσεις που απαιτούνται για αυτή διασύνδεση (βλ. Παράγραφο 2.3.2.5). Αντίθετα στο LXC, παρόλο που οι ρυθμίσεις για τη διασύνδεση του δικτύου γίνονται επίσης αυτόματα κατά τη δημιουργία του container με τη βοήθεια ενός NAT δικτύου, ο χρήστης έχει πρόσβαση σε όλες τις δικτυακές ρυθμίσεις και δυνατότητες που του προσφέρουν τα Linux (π.χ. static IPs και iptables).

Τέλος, σημαντικό είναι ότι το Docker προσφέρει, ίσως, τις ισχυρότερες default ρυθμίσεις σχετικά με την ασφάλεια. Ξεκινώντας ένα container με τις στάνταρ ρυθμίσεις, το Docker ρίχνει αρκετά capabilities και εφαρμόζει seccomp φίλτρα, κάτι που περιορίζει αρκετά το attack surface, ενώ άλλες τεχνολογίες container δεν προσφέρουν τέτοιο είδους defaults (με εξαίρεση το LXC που προσφέρει seccomp filtering, αλλά όχι με τόσο αποδοτικά defaults).

3.5 Διερεύνηση ασφάλειας Docker

Σε αυτή την ενότητα θα προσπαθήσουμε να ανακαλύψουμε κινδύνους για την ασφάλεια ενός συστήματος που χρησιμοποιεί Docker, ώστε να δούμε τον τύπο των αδυναμιών που μπορεί να προκύψουν από την χρήση μίας πλατφόρμας. Όπως θα δούμε και στη συνέχεια, παρόλο που το Docker παρέχει ισχυρά security defaults για τους containers και παρόλο που η ασφάλεια του συνεχώς ενισχύεται (πιο πρόσφατα στην έκδοση 1.10), συνεχίζει να έχει αρκετά τρωτά σημεία.

Η πλατφόρμα του Docker προσφέροντας αρκετές ευκολίες και αυτοματοποιήσεις για τον χρήστη, εισάγει κάποια κενά ασφαλείας αν δεν χρησιμοποιηθεί σωστά. Αρχικά ο δαίμονας του Docker τρέχει σαν root στο σύστημα, ώστε να μπορεί να αρχικοποιεί τα κατάλληλα namespaces και τα δικτυακά κομμάτια, οπότε όποιος χρήστης έχει πρόσβαση στον δαίμονα είναι πρακτικά και αυτός root καθώς μπορεί με πολλούς τρόπος να αποκτήσει πλήρη έλεγχο του συστήματος (π.χ. δημιουργώντας έναν container με volume τον κατάλογο ρίζα του host συστήματος). Συνεπώς, πρόσβαση στον δαίμονα θα πρέπει να έχουμε μόνο χρήστες που μπορούμε να εμπιστευτούμε. Επίσης σημαντικό είναι το γεγονός ότι η χρήση του δαίμονα δεν απαιτεί πιστοποίηση του χρήση, παρά μόνο αν γίνουν οι κατάλληλες ρυθμίσεις. Αυτό μπορεί να γίνει σημαντικό πρόβλημα είτε με χρησιμοποιώντας το δαίμονα τοπικά είτε επιτρέποντας τις HTTP και HTTPS συνδέσεις.

Ένα άλλο επίφοβο σημείο του Docker είναι η χρήση απομακρυσμένων repository, καθώς το Docker by default δεν εξακριβώνει την πηγή της εικόνας. Παρόλο που πρόσφατα το Notary έγινε μέρος της επίσημης έκδοσης του Docker, επιτρέποντας την επιβεβαίωση της πηγής της εικόνας μέσω από Certificates, το feature αυτό παραμένει απενεργοποιημένο στις στάνταρ ρυθμίσεις. Αυτό είναι μία αδυναμία που μπορεί εύκολα κανείς να εκμεταλλευτεί στέλνοντας στο Docker Engine κακόβουλες εικόνες, αντί για αυτές που ζητάει ο χρήστης.

Εκτός από τα παραπάνω, αξιοσημείωτη απειλή αποτελεί το γεγονός ότι οι containers, εκτός και αν ρυθμιστούν διαφορετικά, βρίσκονται στο ίδιο υποδίκτυο. Αυτό επιτρέπει σε έναν επιτιθέμενο ο οποίος έχει αποκτήσει τον έλεγχο ενός container να εξαπολύσει πληθώρα δικτυακών επιθέσεων στους υπόλοιπους hosted containers, με σκοπό τον πλήρη έλεγχο τους, την υποκλοπή πληροφοριών ή την διακοπή της παρεχόμενης υπηρεσίας.

Τέλος, αξίζει να σημειωθεί ότι το Docker χρησιμοποιείται συχνά στα πλαίσια της φιλοσοφίας της αμετάβλητης υποδομής (Immutable Infrastructure), βάση της οποίας οτιδήποτε χρησιμοποιείται για την παροχή μίας υπηρεσίας, π.χ. μία εφαρμογή ή ένας server, δεν αναβαθμίζεται, αλλά μόνο αντικαθίσταται από κάτι νέο. Και αυτό, διότι στο Docker λόγω του σκεπτικού των μικρών containers, όπου υπάρχει μίας διεργασία ανά container συχνά λείπουν οι απαιτούμενοι package managers. Συνεπώς, επειδή τα updates είναι δύσκολο να γίνουν, προτιμάται η αντικατάσταση ενός παλιού container με νέο. Ωστόσο, σε αυτές τις περιπτώσεις ελλοχεύει ο κίνδυνος καθυστερημένης υιοθέτησης των νέων εκδόσεων, με ότι αυτό συνεπάγεται για τον τομέα της ασφάλειας. Για παράδειγμα, εάν προκύψει ένα zero-day vulnerability αντί να ενημερωθούν οι containers με το νέο patch μόλις είναι διαθέσιμο, θα πρέπει να αντικατασταθούν με νέους, κάτι το οποίο μπορεί να μην είναι πάντα εφικτό (π.χ. λόγω απαιτήσεων availability του service).

4 DOCKER-SEC

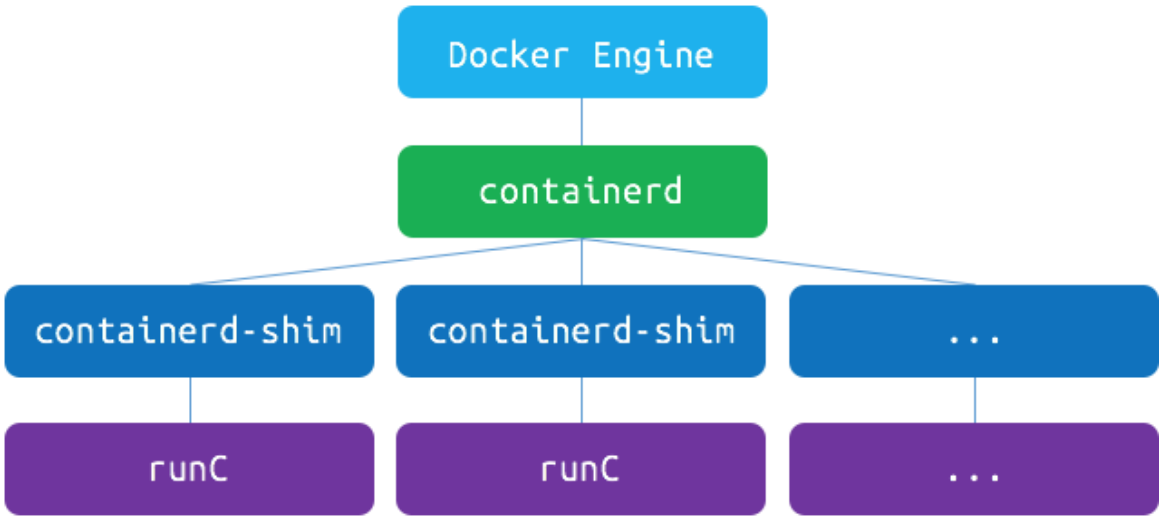
Σε αυτή την ενότητα θα παρουσιάσουμε την υλοποίηση του συστήματος αυτοματοποιημένης ενίσχυσης της ασφάλειας για περιβάλλοντα εικονοποίησης λειτουργικού επιπέδου Docker. Όπως έχει προαναφερθεί, σκοπός της εργασίας είναι η προστασία του host συστήματος από επιθέσεις που ξεκινούν από έναν υπονομευμένο ή κακόβουλο container, καθώς και η προστασία των υπόλοιπων containers που φιλοξενούνται στο ίδιο host μηχάνημα. Η εργασία αυτή φιλοδοξεί να προστατέψει με αυτοματοποιημένο τρόπο containers που υποστηρίζονται από την πλατφόρμα του Docker, επιτρέποντας την κατασκευή κατάλληλων προφίλ του Apparmor, για την εκκίνηση και το runtime του container. Επίσης παρέχεται στον χρήστη η δυνατότητα να εφαρμόσει ακόμα πιο αυστηρά profiles με την εφαρμογή μίας περιόδου training, ορισμένη από το χρήστη, κατά την οποία ο χρήστης έχει την δυνατότητα να εξασκήσει το τμήμα της λειτουργικότητας της εφαρμογής που τον ενδιαφέρει, βάση της οποίας θα κατασκευαστούν τα κατάλληλα προφίλ Apparmor. Το βασικό τμήμα του αυτοματοποιημένου συστήματος ασφάλειας είναι το εργαλείο docker-sec, το οποίο είναι γραμμένο σε bash και λειτουργεί ως ένας wrapper για τις εντολές του Docker. Μέχρι στιγμής υποστηρίζονται περίπου οι μισές εντολές του Docker, κατά κύριο λόγο χωρίς τροποποιήσεις και με όλα τα arguments που αυτές επιτρέπουν. Δηλαδή, ο χρήστης μπορεί να συνεχίσει να χρησιμοποιεί τις εντολές του Docker, τις οποίες είχε συνηθίσει, με τις ίδιες επιλογές, ωστόσο αλλάζοντας το πρώτο λεκτικό της εντολής από docker σε docker-sec, ώστε να κληθεί το εργαλείο, να αναλύσει τις εντολές του χρήστη και να επιτελέσει τις κατάλληλες λειτουργίες. Εκτός από τις εντολές του Docker, υλοποιούνται και κάποιες ακόμα που επιτρέπουν τον ορισμό της περιόδου training. Σε αυτήν την περίπτωση αρκεί ο χρήστης να δώσει το σήμα της εκκίνησης και του τέλους αυτής της περιόδου. Με τη σήμανση του τέλους, το εργαλείου αναλύει τα δεδομένα που συνέλεξε και προχωρά σε κατασκευή νέου προφίλ ή σε τροποποίηση του υπάρχοντος.

4.1 Προστασία μοιραζόμενων πόρων

Αρχικά πρέπει να συνειδητοποιήσουμε τον βαθμό προστασίας που προσφέρουν οι μηχανισμοί που παρουσιάσαμε στην ενότητα 2.3 και ιδίως τα namespaces. Τα namespaces εικονοποιούν ορισμένα τμήματα του συστήματος, απομονώνοντάς τα. Ωστόσο προκειμένου να παρέχουν στις εφαρμογές που τα χρησιμοποιούν την απαιτούμενη λειτουργικότητα, σε πολλές περιπτώσεις συνδέονται πάνω στα νέα namespaces πόροι από το host σύστημα, οι οποίοι δεν μπορούν να απομονωθούν μέσω namespaces. Για παράδειγμα παρόλο που το mount namespace δίνει στον container μία διαφορετική όψη της ιεραρχίας των συστημάτων αρχείων, την οποία μπορεί να την μεταβάλει ανάλογα με τις ανάγκες που προκύπτουν χωρίς να επηρεάζει το host σύστημα, συνήθως πάνω σε αυτό το namespace χρειάζονται συστήματα αρχείων τα οποία είναι κοινά με τον host, όπως τα cgroups και το sysfs, από τα οποίο ο container μπορεί να αποκτήσει πρόσβαση σε ευαίσθητες πληροφορίες και ρυθμίσεις του συστήματος. Αντίστοιχα, συνδυάζοντας το pid με το mount namespace, παρόλο που στο νέο procfs οι κατάλογοι που συνδέονται με τα PIDs των διεργασιών δείχνουν μόνο

τις διεργασίες που έχουν εκκινηθεί μέσα στο νέο pid namespace, οι παράμετροι του πυρήνα παραμένουν εκτεθειμένες στο εσωτερικό του container. Τα παραπάνω προκύπτουν από το γεγονός ότι ο πυρήνας είναι κοινός για το host μηχάνημα και τους containers και συνεπώς αν κάποιος επιθυμεί να δώσει πρόσβαση στο procfs ή στο sysfs, θα δώσει πρόσβαση και στις αντίστοιχες πληροφορίες και παραμέτρους του πυρήνα, καθώς δεν έχουν υλοποιηθεί για αυτά κατάλληλα namespaces. Για τον λόγο αυτό θα πρέπει να βρούμε ποιους πόρους δίνει το Docker στους container, ποιοι από αυτούς είναι ευαίσθητοι και ποιους θα πρέπει να προστατέψουμε με τη βοήθεια του Apparmor. Επίσης είναι σημαντικό να δούμε μέσω ποιον διαδικασιών δίνονται αυτοί οι πόροι από το Docker στους containers, ώστε να επιτρέπουμε μόνο τις «νόμιμες» προσβάσεις σε αυτούς.

Πριν προχωρήσουμε θα εξετάσουμε την αρχιτεκτονική του Docker Engine, ώστε να δούμε ποια components προστατεύει το εργαλείο docker-sec και γιατί. Από την έκδοση 1.11 και μετά το Docker Engine εγκατέλειψε την μονολιθική προσέγγιση που είχε για τον execution driver, ο οποίος χρησιμοποιούσε τη βιβλιοθήκη libcontainer, και χώρισε την αρχιτεκτονική σε διάφορα layers, δημιουργώντας έτσι τον σαφή διαχωρισμό μεταξύ των χαμηλού επιπέδου λειτουργιών για τους containers και των υπόλοιπων. Συγκεκριμένα, το Docker από την έκδοση 1.11 χρησιμοποιεί το **runC** για την διαχείριση του κύκλου ζωής των containers, συμπεριλαμβανομένου των διαδικασιών που απαιτούνται για την σωστή αρχικοποίηση και εκκίνησή τους, επομένως θα μας απασχολήσει αρκετά η σωστή προστασία του. Το runC, προέκυψε από δωρεά του κώδικα της βιβλιοθήκης libcontainer και πλέον αναπτύσσεται ανεξάρτητα από το Docker, βάση του OCI (Open Container Initiative) στάνταρ [39]. Συνεπώς οι βασικές αρχές που θα παρουσιάσουμε στην συνέχεια και οι προτάσεις σχετικά με την προστασία του runC μέσω Apparmor ισχύουν και για άλλες πλατφόρμες container. Παρακάτω φαίνεται η βασική αρχιτεκτονική που χρησιμοποιεί αυτή τη στιγμή το Docker για την διαχείριση των containers:



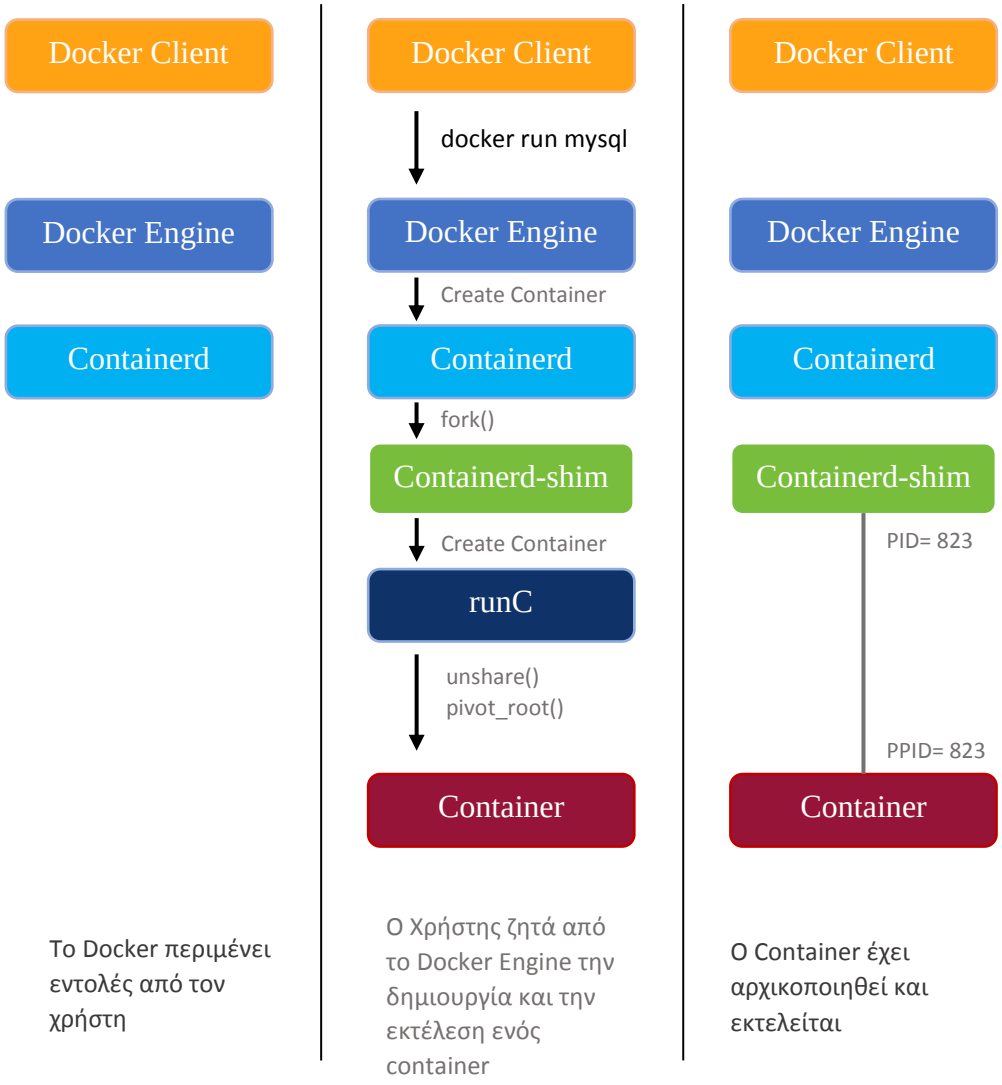
Εικόνα 21: Αρχιτεκτονική Docker Engine (έκδοση 1.11)

Στο παραπάνω σχήμα βλέπουμε ότι το Docker Engine χρησιμοποιεί (από την έκδοση 1.11 και μετά) τον δαίμονα **containerd** για την επικοινωνία με το runC. Ο δαίμονας αυτός, είναι ένας container supervisor ο οποίος μπορεί να χρησιμοποιηθεί με για την διαχείριση οποιουδήποτε OCI compatible container. Παρέχει στον χρήστη ένα gRPC interface και προσφέρει ένα βρόχο συμβάντων (event loop) χωρίς κλειδώματα, μέσα στον οποίο ο δαίμονας μπορεί να διαχειριστεί ταυτόχρονα αιτήματα για τους containers που διαχειρίζεται περνώντας τις εντολές προς τις containerd-shim διεργασίες που ελέγχουν τον κάθε container[3]. Το containerd-shim χρησιμεύει ώστε, μετά την αρχικοποίηση του container από το runC, να μένει ως ανοιχτό ως διεργασία πατέρας του container και συνεπώς να παρακολουθεί τη λειτουργία του και να εκτελεί την επικοινωνία που απαιτείται (μέσω ptrace και signals) με αυτόν, στέλνοντας τις απαραίτητες εντολές προς εκτέλεση στο runC (οι διεργασίες του runC τερματίζουν μετά την εκτέλεση των απαραίτητων ενεργειών). Αυτή η αρχιτεκτονική επιτρέπει στο Docker Engine να μπορεί να κάνει επανεκκινήσεις, άρα και upgrades, χωρίς να επηρεάζει την λειτουργία των containers. Επίσης, το Docker δεν χρειάζεται πλέον να διαχειρίζεται τις χαμηλού επιπέδου λειτουργίες που σχετίζονται με τον κύκλο ζωής των containers, όπως τα cgroups και τα namespaces, αλλά μπορεί να εστιάσει στην παροχή της υπόλοιπης λειτουργικότητας συμπεριλαμβανομένου της διαχείρισης των images, των volumes, της δικτύωση μεταξύ των containers[4] και του logging.

Στο παρακάτω σχήμα (εικόνα 22) μπορούμε να δούμε αναλυτικά πως το Docker χρησιμοποιεί τα διάφορα components για να ελέγξει τον κύκλο ζωής του container. Αρχικά, σε μία τυπική εγκατάσταση του Docker οι βασικές διεργασίες που εκτελούνται είναι το Docker Engine (συγκεκριμένα ο Docker Daemon), το οποίο περιμένει εντολές από κάποιον Docker Client και ο δαίμονας Containerd, ο οποίος περιμένει εντολές από το Docker Engine. Όταν ο χρήστης δώσει εντολή για δημιουργία νέου container (π.χ. docker run) το Docker Engine παραλαμβάνει και επεξεργάζεται το αίτημα. Αφού βρει ή δημιουργήσει (από dockerfile) το image από το οποίο θα κατασκευάσει τον container προωθεί το αίτημα με τις κατάλληλες ρυθμίσεις στο containerd. Ο δαίμονας αυτός στη συνέχεια ξεκινά μία νέα διεργασία containerd-shim με τις ρυθμίσεις που παρέλαβε από το Engine. Η διεργασία αυτή με τη σειρά της χρησιμοποιεί το runC για την αρχικοποίηση και την εκκίνηση του container. Μόλις ολοκληρωθούν αυτές οι ενέργειες το runC τερματίζει. Ωστόσο το containerd-shim συνεχίζει να τρέχει ώστε να επιβλέπει τον container, ο οποίος πλέον είναι παιδί του. Επομένως, σε επόμενες εντολές που θα στείλει ο χρήστης, ο containerd δαίμονας θα μπορεί να προωθήσει την εντολή στην containerd-shim διεργασία που επιβλέπει τον container, η οποία θα την εκτελέσει με την βοήθεια του runC. Αξίζει να σημειωθεί ότι εκτός από την run οι εντολές create, start, stop, pause, resume, exec, signal και delete του Docker προσφέρονται μέσω του containerd και του runC και δεν τις διαχειρίζεται πλέον το Docker Engine.

[3] Με αυτήν την αρχιτεκτονική είναι εύκολο σε συμβατικούς υπολογιστές να εκκινήσει κανείς 100 containers σε χρόνο λίγο μεγαλύτερο από ένα δευτερόλεπτο.

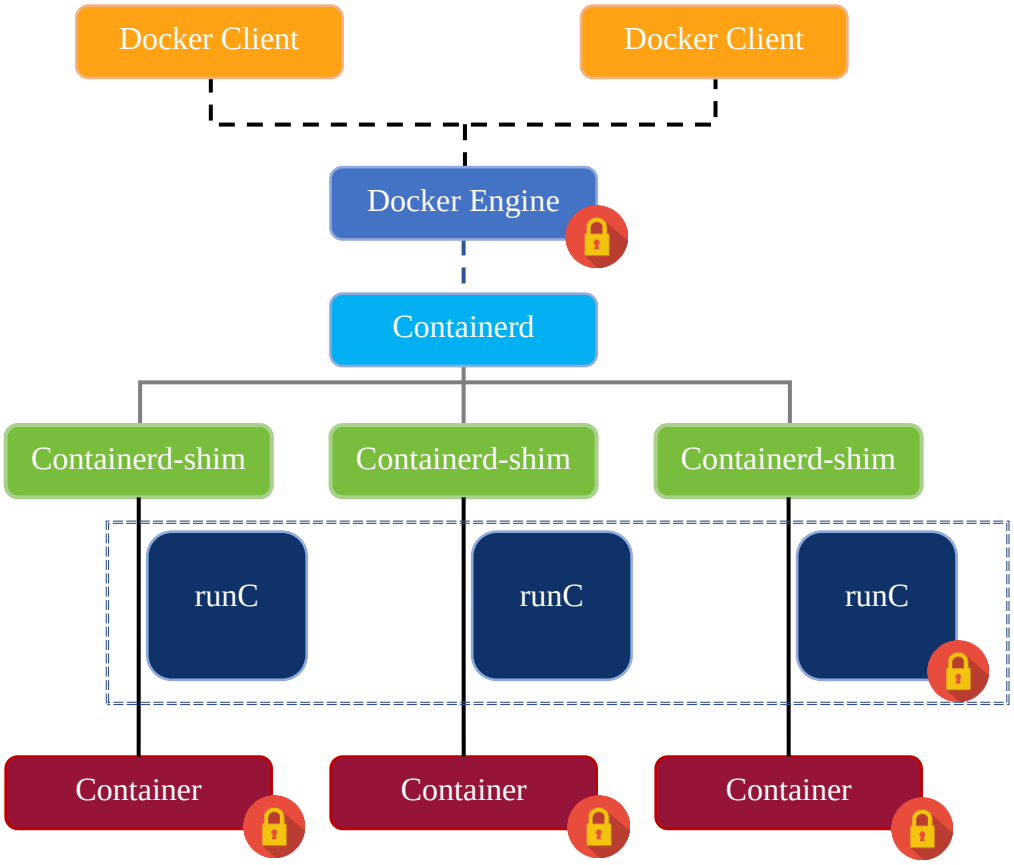
[4] Ωστόσο σε επόμενες εκδόσεις το containerd θα μπορεί να διαχειρίζεται όχι μόνο primitives δικτύου, αλλά και την δικτύωση μεταξύ των containers, όπως τα bridges.



Εικόνα 22: Δημιουργία container σε Docker περιβάλλον

Παρατηρώντας, λοιπόν, την αρχιτεκτονική του Docker, μπορούμε να δούμε ποιες διεργασίες απαιτούν την μεγαλύτερη προσοχή, ώστε να γίνει πιο ασφαλές το περιβάλλον. Αρχικά, το docker-sec επιβάλλει Apparmor profiles στους containers, αφού είναι και το μόνο κομμάτι του συστήματος όπου μπορεί να τρέξει αυθαίρετος κώδικας και όπου έχουν πρόσβαση χρήστες των εικονοποιημένων εφαρμογών, συνεπώς είναι το πρώτο σημείο στο οποίο μπορεί να αποκτήσει πρόσβαση ένας επιτιθέμενος. Σκοπός μας είναι η εφαρμογή σε κάθε container ενός ξεχωριστού προφίλ κατασκευασμένο ώστε να δίνει τα λιγότερα δικαιώματα που απαιτούνται για την εκτέλεση της εφαρμογής εντός του container. Ακόμη, το docker-sec δίνει ιδιαίτερη προσοχή στην προστασία με Apparmor του runC, καθώς είναι το εργαλείο που διαχειρίζεται όλες τις χαμηλού επιπέδου λειτουργίες των containers, όπως είδαμε παραπάνω, και επομένως αν κάποιος container καταφέρει να ελέγξει το runC θα δημιουργήσει σημαντικά προβλήματα όχι μόνο στους υπόλοιπους containers, αλλά και στο host μηχάνημα. Μάλιστα το docker-sec έχει δημιουργηθεί με σκοπό να προστατεύει ολόκληρη την διαδικασία της εκκίνησης του container, δηλαδή όλη την διαδικασία από την στιγμή που το runC ξεκινά την αρχικοποίηση του container μέχρι

να δώσει τον έλεγχο στη διεργασία που βρίσκεται στο εσωτερικό του container. Τέλος, προστατεύουμε το Docker Engine, μίας και από εκεί οι χρήστες έχουν πλήρη έλεγχο στους containers, τα images, τα volumes και το δίκτυο. Παρακάτω (εικόνα 23) φαίνονται σχηματικά τα τμήματα του Docker που θα προστατέψουμε με τη βοήθεια του Apparmor.



Εικόνα 23: Σχηματική αναπαράσταση των τμημάτων του Docker που προστατεύει το εργαλείο docker-sec

4.2 Εφαρμογή των profiles

Σε αυτή την ενότητα θα παρουσιάσουμε τον τρόπο με τον οποίο το docker-sec εφαρμόζει τα προφίλ ώστε να προστατέψει όχι μόνο τους containers, αλλά και όλη τη διαδικασία εκκίνησής τους. Ωστόσο, πριν ξεκινήσουμε, είναι χρήσιμο να δούμε συνοπτικά τον τρόπο με τον οποίο το Docker οργανώνει τις εικόνες, τους containers και τις ρυθμίσεις τους στο σύστημα αρχείων του host, καθώς το σύστημα αυτοματοποιημένης ασφάλειας κατασκευάζει τα προφίλ βασισμένο σε αυτήν την οργάνωση. Θα εξετάσουμε μία τυπική εγκατάσταση του Docker, όπου ο storage driver είναι ο default, δηλαδή ο aufs driver. Το Docker αποθηκεύει τα περισσότερα αρχεία

και τις περισσότερες στατικές ρυθμίσεις κάτω από τον κατάλογο /var/lib/docker. Κάτω από αυτόν βρίσκεται ο φάκελος aufs, ο οποίος θα μας απασχολήσει περισσότερο. Μέσα σε αυτόν υπάρχουν οι εξής κατάλογοι: diff, όπου αποθηκεύονται σε layers όλα τα αρχεία των containers και των images, layers, όπου υπάρχει ένα αρχείο για κάθε layer που δείχνει ποια layers υπάρχουν κάτω από αυτό, και τέλος, ο mnt φάκελος, όπου το Docker ενώνει τα διάφορα layers σε ένα mount point (μέσω του AUFS)[5], το οποίο είναι το file σύστημα που βλέπει ο κάθε container. Εκτός από τον aufs, κάτω από το docker, υπάρχουν οι φάκελοι container, με πληροφορίες και στατικές ρυθμίσεις για τους containers, images, με πληροφορίες που επιτρέπουν στο docker να αναγνωρίζει τα αποθηκευμένα layers ώστε να τα επαναχρησιμοποιεί όπου χρειάζεται χωρίς να τα κατεβάζει ξανά, network, με τα απαραίτητα αρχεία για τη λειτουργία του δικτύου μεταξύ των containers, και volumes, όπου περιέχονται τα αρχεία που αποθηκεύουν οι containers στο κάθε volume. Οι προσωρινές ρυθμίσεις, τα αρχεία που απαιτούνται για το runtime των containers (π.χ. pipes για το stdin και το stdout, sockets για το δίκτυο) και τα logs των containers βρίσκονται αποθηκευμένα κάτω από τον φάκελο /var/run/docker. Συνεπώς, τους δύο αυτούς καταλόγους (/var/lib/docker και /var/run/docker) το εργαλείο docker-sec τους αξιοποιεί στην αυτόματη κατασκευή των προφίλ, καθώς από εκεί αντλεί σημαντικές πληροφορίες για τους containers που τρέχουν στο Docker. Επιπλέον, φροντίζουμε ώστε σε αυτούς τους φακέλους να έχουν πρόσβαση μόνο όσες διεργασίες χρειάζεται πραγματικά να έχουν, καθώς περιέχουν ευαίσθητα αρχεία των containers[6].

```
[01:20:27] root@dockmonitor:/var/lib/docker$ tree -L 2
.
├── aufs
│   ├── diff
│   ├── layers
│   └── mnt
├── containers
│   ├── 1851794543a89b3ebe558b3444b6e02b6d61d4073b263802f64a0291d77e5628
│   ├── 1e2f9c7ef13d3f1fef56257ca65b11070751d1c58126656e54ec7637a6afc26c
│   ├── 292cedbaaf6641472bb3ac639746e71a91bbe1d8be317b5191f7382eba5c5c6d
│   ├── a26ce34cea236a2a1d437c9814eb1f66807e805a143e1d09e4c9241d2d139a56
│   ├── a3d81beff57b965848a6647b6d05847129e798d50fc8990c9c9837170915306d
│   ├── a42512f549bc4d30afc6d24d6455356bd67b95717f7d754c72ca854033ca0160
│   ├── d5f4105b1a21604581c6283ef1dbd872e72da6980c33534e376d0503f7f3a205
│   ├── de2d1f3e34cc2e67aaec0013608a141259f35f989fe18b7770a2810b6369a32d
│   └── e579f2d7a383471f81e1331d64f69f33b3caf694eaf33b06a951ff3b2c804427
├── image
│   └── aufs
├── network
│   └── files
├── swarm
├── tmp
├── trust
└── volumes
    ├── 1bc20fd8cc39d72b6f51ce6eab5dd44db26ff2cda714f0724ce4ca0ab0895769
    ├── 30eea922dabecb1e141841bf262ac0024259bc46fa8e8668826716a8176b5d40
    └── metadata.db
```

Εικόνα 24: Τυπική δομή του καταλόγου /var/lib/docker

[5] Το Docker χρησιμοποιεί τα AUFS για να ενώσει όλα τα στρώματα των εικόνων (και το init layer του container) σε read-only mode με το λεπτό layer σε read/write mode κάτω από ένα σημείο στον φάκελο mnt, όπως είχαμε δει στην ενότητα 3.3.

[6] Αν ένας κακόβουλος χρήστης αποκτήσει πρόσβαση σε αυτά μπορεί να προκαλέσει σοβαρά προβλήματα σε αυτά, όπως να σταματήσει τους containers (στέλνοντας exit εντολή μέσα από το stdin pipe) ή να τοποθετήσει κακόβουλα αρχεία στα layers των images (τα οποία θα είναι ορατά από όλους τους containers που χρησιμοποιούν αυτά τα layers).

Εικόνα 25: Τυπική δομή του καταλόγου /var/run/docker

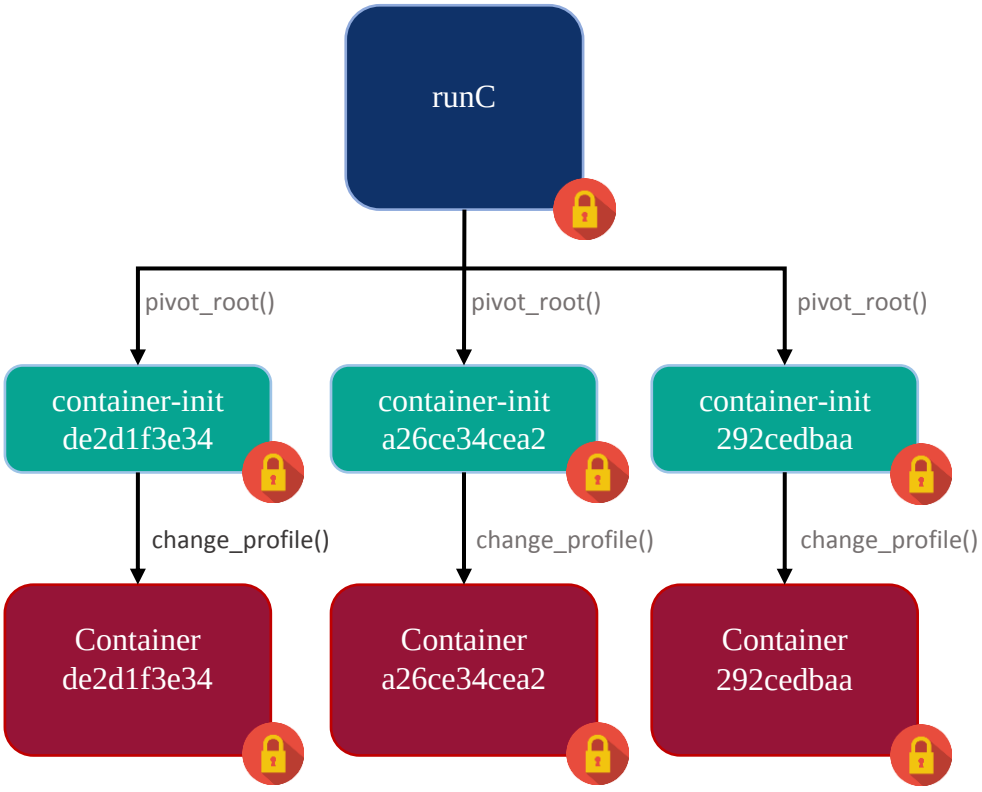
Έχοντας δει το πώς το Docker αποθηκεύει τα αρχεία των containers και τις απαιτούμενες ρυθμίσεις, μπορούμε να μελετήσουμε τον τρόπο με τον οποίο εκκινεί τους containers, ώστε να εφαρμόσουμε τα κατάλληλα AppArmor profiles. Προκειμένου να αναλύσουμε τις ενέργειες που εκτελεί το runC για να ξεκινήσει σε ένα απομονωμένο περιβάλλον τους containers, χρησιμοποιήσαμε τον auditd δαίμονα και τις δυνατότητες που δίνει το AppArmor για auditing των confined διεργασιών. Ο auditd δαίμονας είναι το τμήμα του Linux Audting System το οποίο καταγράφει σε logs τα συμβάντα που έχουμε ορίσει στο αρχείο audit.rules. Μέσω αυτού του συστήματος μπορούμε να καταγράφουμε τις κλήσεις συστήματος που επιθυμούμε και τις προσβάσεις σε συγκεκριμένα αρχεία του συστήματος (file watches). Συνεπώς καταγράφοντας τις κλήσεις συστήματος όπως οι unshare, clone, execve, socket, mount, chroot και pivot_root μπορεί κανείς να δει τις διαδικασίες που επιτελούνται από το runC πριν δώσει τον έλεγχο στην διεργασία που ορίζει ο χρήστης να ξεκινά με την εκκίνηση του container. Ένα configuration του auditd (σε αρχιτεκτονική x86-64) που μπορεί να μας δώσει κάποιες από αυτές τις πληροφορίες είναι το εξής:

```
-a always,exit -F arch=b64 -S pivot_root -F key=piv_root
-a always,exit -F arch=b64 -S chroot -F key=change_root
-a always,exit -F arch=b64 -S execve -F key=execve_call
-a always,exit -F arch=b64 -S execveat -F key=execve_call
-a always,exit -F arch=b64 -S clone -F key=clone_ call
-a always,exit -F arch=b64 -S fork -F key=clone_ call
-a always,exit -F arch=b64 -S vfork -F key=clone_call
-a always,exit -F arch=b64 -S socket -F key= socket_call
```

Αναλύοντας τις πληροφορίες που συλλέγουμε από το auditing, βλέπουμε ότι το runC, αρχικά δημιουργεί τα νέα namespaces (mount, IPC, UTS, PID και network με τις στάνταρ ρυθμίσεις) που θα χρησιμοποιήσει ο container. Στη συνέχεια δημιουργεί το κατάλληλο περιβάλλον βάσει των ρυθμίσεων του container, συνδέοντας τα νέα namespaces με πόρους του συστήματος. Για παράδειγμα το docker-runC, αναλαμβάνει να προετοιμάσει το νέο mount namespace με τα συστήματα αρχείων που θα χρειαστεί

ο νέος container. Αρχικά, με τη δημιουργία του νέου mount namespace, αντιγράφεται η παλιά ιεραρχία συστημάτων αρχείων σε αυτό και συνεπώς είναι απαραίτητο να τροποποιηθεί ώστε να περιοριστούν οι προσβάσεις που έχει το νέο mount namespace (ως κλώνος του αρχικού mount namespace). Συνεπώς, το runC κάνει remount τον κατάλογο ρίζα με την επιλογή rprivate (recursive private), δηλαδή ορίζει το mount point αυτό και όλα όσα βρίσκονται από κάτω του ως private mount, το οποίο σημαίνει ότι οποιεσδήποτε αλλαγές συμβούν στο νέο namespace δεν θα μεταφερθούν στο παλιό και το αντίστροφο. Στη συνέχεια, εκτελεί mount για διάφορα συστήματα αρχείων, όπως procfs, devices (π.χ. shm, consoles, mqueue) και cgroups (εκτελεί mount για όλα τα control groups που αντιστοιχούν στον συγκεκριμένο container, όπως blkio, cpu κλπ.) και μερικά από αυτά τα μαρκάρει ως read-only. Αφού εκτελέσει τις περισσότερες αρχικοποιήσεις, προχωρά σε pivot_root. Η κλήση συστήματος pivot_root μετακινεί το παλιό root σύστημα αρχείων της καλούσας διεργασίες σε ένα φάκελο old_root[7] και τοποθετεί στη θέση του ένα νέο root filesystem. Με αυτόν τον τρόπο η διεργασία βλέπει πλέον, το mount point που έχει δημιουργηθεί από τα aufs για τον container, ως κατάλογο ρίζα. Έπειτα, εκτελεί κάποιες αρχικοποιήσεις που απομένουν, όπως η διαγραφή του καταλόγου που περιέχει το παλιό root σύστημα αρχείων και ορισμένα remounts και δίνει τον έλεγχο στο εκτελέσιμο που έχει ορίσει ο χρήστης ως σημείο εκκίνησης του container.

Βάσει τις διαδικασίας που ακολουθεί το runC για την αρχικοποίηση των containers, καταλήγουμε στην εφαρμογή των προφίλ όπως φαίνονται στην παρακάτω εικόνα:



Εικόνα 26: Προφίλ κατά την εκκίνηση των container

[7] Στην περίπτωση του Docker ονομάζεται .pivot_root συνοδευόμενο από εννέα τυχαία ψηφία

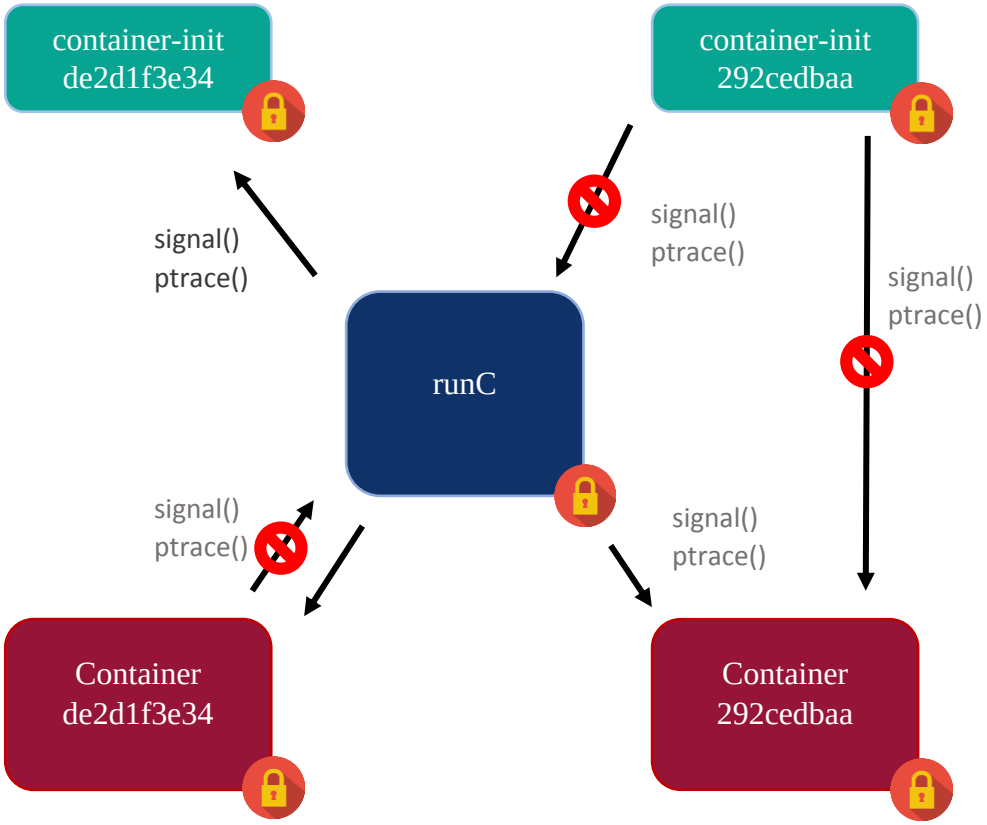
Κατ' αρχάς, όταν κάποιος Docker Client ζητήσει την εκκίνηση κάποιου container, το Docker μέσω του containerd εκκινεί το runC, το οποίο στα πλαίσια του συστήματος ασφαλείας που παρουσιάζουμε σε αυτή την εργασία, προστατεύεται από ένα AppArmor profile. Μέσα στο προφίλ του runC περιέχονται οι κατάλληλοι κανόνες pivot_root που επιτρέπουν την εκκίνηση όλων των container που διαχειρίζεται το Docker. Μέσα από αυτούς του κανόνες το AppArmor δίνει την δυνατότητα στη διεργασία που εκτελεί την κλήση συστήματος pivot_root να αλλάξει profile ανάλογα με το σημείο στο οποίο εκτελεί pivot_root. Συνεπώς, για κάθε container υπάρχει ένας κανόνας pivot_root στο mount point που χρησιμοποιεί ο container (/var/lib/docker/aufs/mnt/<container_id>) το οποίο χρησιμοποιείται για την είσοδο σε ένα άλλο AppArmor προφίλ, το οποίο είναι ειδικά σχεδιασμένο για την εκκίνηση του container. Αυτά τα προφίλ, χρησιμεύουν όχι μόνο για την προστατευμένη εκκίνηση του container, αλλά και στην επιβολή του τελικού προφίλ που θα χρησιμοποιηθεί κατά την εκτέλεση του (συγκεκριμένου) container. Αυτή η αλλαγή γίνεται μέσω του AppArmor κανόνα change_profile, ο οποίος ενεργοποιείται όταν κληθεί είτε η συνάρτηση aa_change_profile είτε η aa_change_onexec. Κατ' αυτόν τον τρόπο δημιουργούμε μία πλήρως προστατευμένη ροή εκτέλεσης των containers, η οποία έχει αφετηρία το profile του runC, συνεχίζει με την επιβολή προφίλ αρχικοποίησης, το οποίο είναι διαφορετικό για κάθε container, και τέλος καταλήγει στην εφαρμογή του runtime προφίλ του container, το οποίο είναι επίσης διαφορετικό για κάθε container.

Σε αυτό το σημείο αξίζει να εξετάσουμε ορισμένες λεπτομέρειες των προφίλ που παρουσιάσαμε παραπάνω. Το runC, όπως είναι αναμενόμενο, προστατεύεται από ένα AppArmor profile το οποίο έχει τις κατάλληλες προσβάσεις για την αρχικοποίηση και τον έλεγχο των containers. Αρχικά, δίνονται ορισμένα capabilities, όπως sys_admin, net_admin, mknod (για την δημιουργία sockets), sys_ptrace (για τον έλεγχο των containers) και άλλα, τα οποία είναι απαραίτητα για τις διαχειριστικές ενέργειες που εκτελεί. Επιπρόσθετα, δίνει στο runC ορισμένους δικτυακούς πόρους, αλλά και την δυνατότητα να κάνει mount όλα τα απαραίτητα συστήματα αρχείων για έναν container, όπως περιγράψαμε παραπάνω, συμπεριλαμβανομένου των volumes που μπορεί να απαιτούνται για κάποιον container. Τέλος, το AppArmor προφίλ αυτό έχει δικαίωμα να εκτελεί ptrace στον εαυτό του, σε όλα τα προφίλ εκκίνησης των containers και σε όλα τα προφίλ που χρησιμοποιούνται για την προστασία της εκτέλεσης των containers, ενώ εμποδίζει το ptrace από προφίλ των containers. Αντίστοιχα επιτρέπει την αποστολή σημάτων σε προφίλ των container, αλλά όχι τη λήψη σημάτων από αυτά. Κατ' αυτόν τον τρόπο, το runC μπορεί να ελέγχει την εκτέλεση των containers, ώστε να μπορεί να προσφέρει την απαιτούμενη λειτουργικότητα (π.χ. docker exec), αλλά το προφίλ προστατεύει το runC από σήματα και tracing που έρχονται από containers[8].

Τα προφίλ που χρησιμοποιούνται για την προστατευμένη εκκίνηση και αρχικοποίηση των containers, έχουν κανόνες αντίστοιχης φύσης με το profile του runC, ωστόσο με σαφώς μειωμένα δικαιώματα. Συγκεκριμένα, επιτρέπονται διάφορα capabilities με σημαντικότερα τα sys_admin και setpcap, ώστε να μπορεί να ρίχνει τα περιττά capabilities πριν δώσει τον έλεγχο της εκτέλεσης στον container, καθώς και ορισμένα mounts και unmounts (ώστε να αφαιρεθεί το mount point του παλιού root

[8] Αυτό θα μπορούσε να συμβεί μόνο από έναν κακόβουλο container και δεν αποτελεί τμήμα της σχεδίασης του Docker.

filesystem, το οποίο αντιστοιχεί στο root του αρχικού mount namespace). Επίσης, επιτρέπει την λήψη σημάτων και το tracing από το runC, καθώς, και την αλλαγή προφίλ σε αυτό που θα χρησιμοποιήσει ο container για την υπόλοιπη λειτουργία του.



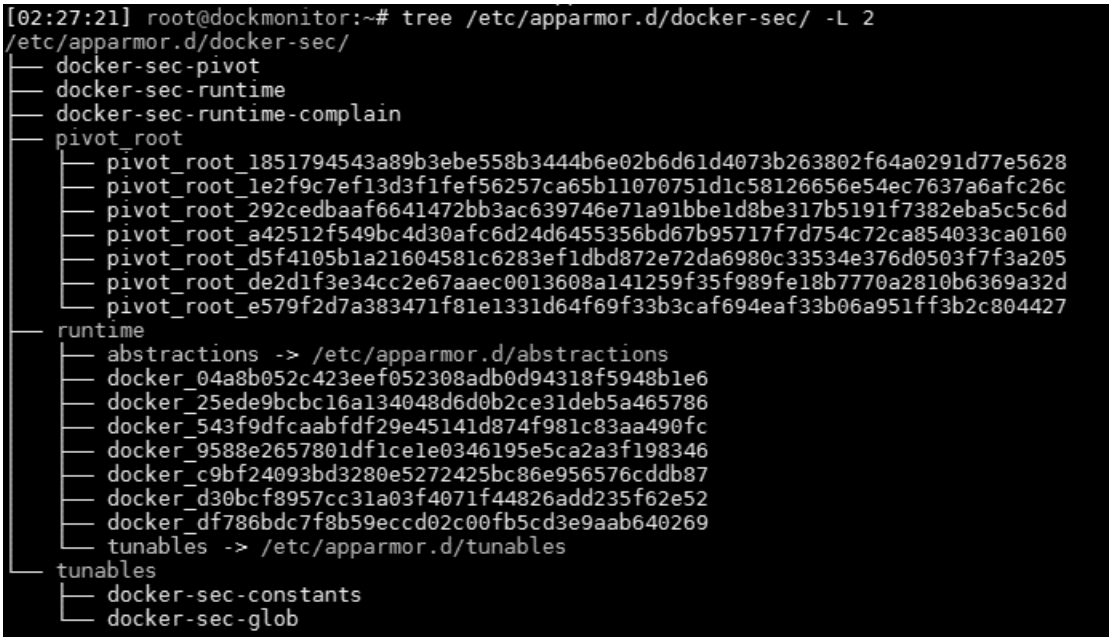
Εικόνα 27: Μπλοκάρισμα απαγορευμένων σημάτων και ptrace από το AppArmor

Τέλος το προφίλ του runtime των containers, μπορεί να διαφέρει σημαντικά ανάλογα με τις ρυθμίσεις του container και την εφαρμογή, ωστόσο κατά κανόνα περιέχει μερικούς βασικούς κανόνες προστασίας που απαγορεύουν την πρόσβαση του container σε ορισμένα ευαίσθητα αρχεία του συστήματος, όπως το /proc/PID/mem και την εκτέλεση των κλήσεων συστήματος mount και umount.

4.3 Δομή

Έχοντας δει το πώς χρησιμοποιούνται τα AppArmor προφίλ για την προστασία των Docker container, μπορούμε να εξετάσουμε την δομή του συστήματος αυτοματοποιημένης ασφάλειας. Στο σύστημα αυτό διακρίνουμε δύο βασικά τμήματα. Το πρώτο αποτελείται από τα AppArmor προφίλ και το δεύτερο από το docker-sec εργαλείο, μέσα από το οποίο δημιουργούνται και ενημερώνονται τα κατάλληλα προφίλ.

Τα profiles του AppArmor αποθηκεύονται κάτω από τον κατάλογο /etc/apparmor.d. Μέσα σε αυτόν βρίσκονται τα προφίλ του Docker Engine και του runC, καθώς και ο κατάλογος docker-sec. Ο φάκελος docker-sec περιέχει τα προφίλ που χρησιμοποιούνται για την αρχικοποίηση των containers (κάτω από τον κατάλογο



Εικόνα 28: Δομή των AppArmor προφίλ του αυτοματοποιημένου συστήματος ασφάλειας

docker-sec/pivot_root), τα προφίλ του runtime των containers (στον φάκελο docker-sec/runtime) και τα αρχεία των glob pattern που χρησιμοποιούν τα υπόλοιπα προφίλ. Αυτή δομή φαίνεται καθαρά παρακάτω:

Το δεύτερο τμήμα αποτελείται από το εργαλείο docker-sec, το οποίο όπως προαναφέρθηκε είναι γραμμένο σε bash και λειτουργεί ως ένας wrapper του Docker CLI Client. Προκειμένου να παρέχει την απαιτούμενη λειτουργικότητα χρησιμοποιεί δύο άλλα script αρχεία, το *docker-sec_parser.sh*, το οποίο χρησιμοποιείται για το parsing των profiles και των logs που απαιτούνται, και το *docker-sec_utils.sh*, το οποίο παρέχει διάφορες συναρτήσεις που βοηθούν στην εφαρμογή των προφίλ, την αντιστοίχισή τους με τους containers, στην διεπαφή με το χρήστη κ.α.

4.4 Στατική ανάλυση

Σε αυτή την ενότητα θα περιγράψουμε με ποιον τρόπο εφαρμόζονται και τροποποιούνται τα κατάλληλα Apparmor profiles όταν κατασκευάζεται ένας νέος container. Η διαδικασία αυτή επιτελείται μέσω του εργαλείου docker-sec και, συγκεκριμένα, όταν ο χρήστης εκτελέσει docker-sec create ή docker-sec run, εντολές κατά τις οποίες το Docker Engine κατασκευάζει τους ζητούμενους containers. Το εργαλείο συλλέγει ορισμένες πληροφορίες για τον container από τις επιλογές που δίνει ο χρήστης στο Docker μέσω της εντολής docker-sec create (η οποία προσφέρει την ίδια σύνταξη και τις ίδιες επιλογές με την εντολή docker create) πριν την δημιουργία του container και ορισμένες πληροφορίες μετά τη δημιουργία του (δηλαδή αφού εκτελεστεί η εντολή docker create). Επομένως, με την κατασκευή του container στο περιβάλλον

Docker, το εργαλείο docker-sec έχει μαζέψει τις απαραίτητες πληροφορίες για τον νέο container και μπορεί να εφαρμόσει τα κατάλληλα προφίλ, ώστε το περιβάλλον να είναι έτοιμο για την εκκίνηση του νέου container, όποτε το επιθυμεί ο χρήστης. Συνεπώς, γίνεται μία στατική ανάλυση των παραμέτρων που χρησιμοποίησε ο χρήστης για την κατασκευή του νέου container, καθώς και των αρχείων που δημιούργησε το Docker. Βάσει αυτών κατασκευάζονται τα κατάλληλα προφίλ πριν την εκκίνηση του container και, κατ' επέκταση, χωρίς να γίνεται χρήση κάποιων runtime πληροφοριών του container.

Αρχικά, όταν ένας χρήστης εκτελέσει την εντολή docker-sec create, το εργαλείο ελέγχει αν ο χρήστης έχει ζητήσει από το Docker να εκτελέσει τον container κάτω από κάποιο AppArmor προφίλ. Σε αντίθετη περίπτωση (που αναμένεται να είναι και η συνηθέστερη), το εργαλείο κατασκευάζει ένα όνομα για το runtime profile που θα χρησιμοποιηθεί για τον container και δίνει αυτό στην εντολή docker create, μέσω της παραμέτρου: *--security-opt apparmor=RUNTIME_PROF*. Σε αυτό το σημείο το προφίλ αυτό δεν είναι απαραίτητο να υπάρχει και να έχει γίνει enforce, καθώς η εντολή docker create απλώς προετοιμάζει το σύστημα αρχείων του container και τις κατάλληλες ρυθμίσεις, χωρίς να τον εκκινεί. Στην συνέχεια το docker-sec εκτελεί την τροποποιημένη docker create. Τέλος, καλείται η συνάρτηση του docker-sec parser για την κατασκευή των απαραίτητων profile, δηλαδή ενός προφίλ εκκίνησης του container και ενός προφίλ εκτέλεσης, και την ενημέρωση του προφίλ του runC, ώστε να επιτρέπει την μετάβαση σε αυτά.

Προκειμένου να δημιουργηθούν τα κατάλληλα προφίλ, το docker-sec parser συλλέγει πληροφορίες για τον container, όπως το id του (το οποίο είναι ένα SHA256 checksum, όπως είχαμε δει σε προηγούμενη ενότητα) και το σημείο mount που θα χρησιμοποιηθεί για το pivot_root. Βάσει αυτών, κατασκευάζει το runtime προφίλ στον κατάλογο /etc/apparmor.d/docker-sec/runtime/ και το προφίλ εκκίνησης του container στο path /etc/apparmor.d/docker-sec/pivot_root/ δίνοντας σε αυτά τα προφίλ προκαθορισμένες προσβάσεις που είναι οι ελάχιστες που απαιτούνται για Docker containers. Στη συνέχεια φροντίζει ώστε το runC να ενημερωθεί για τα νέα προφίλ και να επιτρέπει την μετάβαση στο προφίλ εκκίνησης μέσω pivot_root και την μετάβαση απευθείας στο προφίλ εκτέλεσης μέσω change_profile. Ομοίως, το προφίλ εκκίνησης του container τροποποιείται ώστε να επιτρέπει την μετάβαση στο αντίστοιχο runtime προφίλ. Τέλος, προκειμένου να διευκολύνεται η διαχείριση των προφίλ και να παραμένει μικρός ο αριθμός των συνολικών κανόνων, οι πληροφορίες που συλλέγονται κατά την παραπάνω διαδικασία, δηλαδή το id του container, το mount point και τα ονόματα των δύο containers, αποθηκεύονται σε κατάλληλες μεταβλητές στο αρχείο docker-sec-glob, από όπου μπορούν εύκολα να προσπελαστούν από οποιοδήποτε προφίλ τα χρειάζεται, δηλαδή, κατά κύριο λόγο, από το προφίλ του runC.

Μόλις κατασκευαστούν τα προφίλ, το docker-sec, αφού συλλέξει ορισμένες ρυθμίσεις του container, αναλαμβάνει να ενημερώσει, κατά κανόνα, το προφίλ του runC και το pivot_root profile του container, αφού αυτά αναλαμβάνουν την προετοιμασία για την εκκίνησή του. Προκειμένου να διαπιστώσει τις παραμέτρους που έδωσε ο χρήστης, αλλά και ορισμένες ρυθμίσεις που αφορούν το περιβάλλον του Docker, το docker-sec χρησιμοποιεί την εντολή docker inspect, η οποία παρουσιάζει τις παραμέτρους του container σε json μορφή και επιτρέπει την εμφάνιση μόνο

συγκεκριμένων τμημάτων της πληροφορίας αυτής. Για παράδειγμα, προκειμένου να υποστηριχθούν τα Docker volumes, το docker-sec parser διαβάζει από το json αρχείο των παραμέτρων το σχετικό τμήμα μέσω της εντολής docker-sec inspect που επιστρέφει τη ζητούμενη πληροφορία μορφοποιημένη. Από εκεί μπορεί να εξάγει τις απαιτούμενες παραμέτρους για το mounting των volumes, όπως ο κατάλογος πηγή, ο προορισμός του mount καθώς και ρυθμίσεις, όπως το αν είναι read-only ή όχι. Βάσει των παραπάνω μπορούν να κατασκευαστούν οι κατάλληλοι AppArmor κανόνες, οι οποίοι προστίθενται στο προφίλ του runC, το οποίο είναι και αυτό που αναλαμβάνει τα mounts (όπως είχαμε δει παραπάνω). Ωστόσο σε άλλες περιπτώσεις, όπως στην περίπτωση που ο container ξεκινάει με άλλον χρήστη από τον root, το προφίλ που πρέπει να τροποποιηθεί είναι το προφίλ που χρησιμοποιείται κατά την αρχικοποίησης του container, καθώς εκεί εκτελούνται τα τελευταία βήματα της αρχικοποίησης, όπως η αλλαγή χρήστη, η οποία απαιτεί το chown capability, το οποίο δεν δίνεται by default στα pivot_root προφίλ.

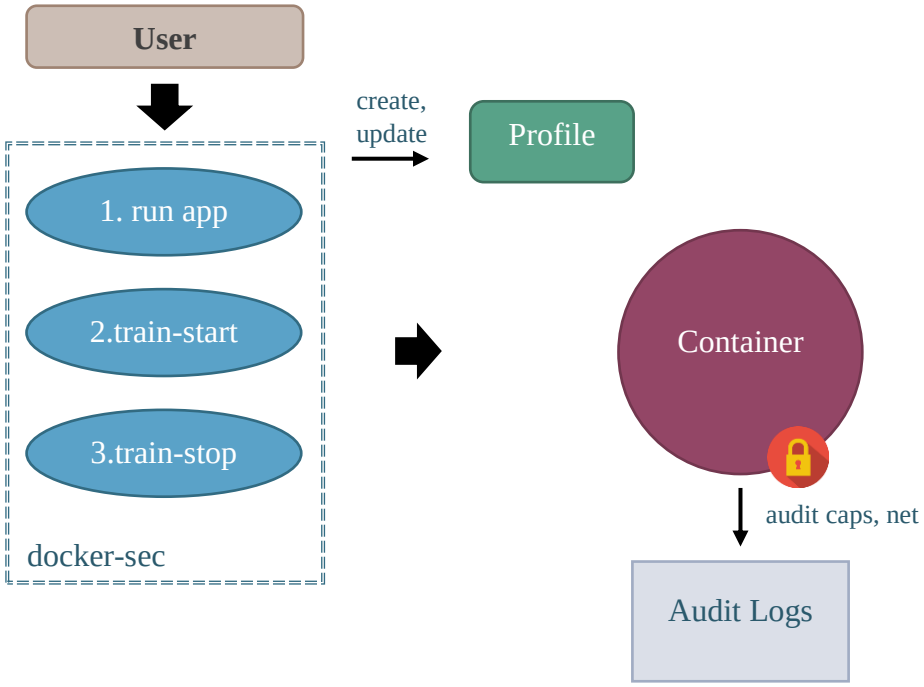
Αντίστοιχη διαδικασία ακολουθείται και κατά την διαγραφή ενός Docker container. Όταν ένας χρήστης δώσει την εντολή docker-sec rm –p CONT_NAME, τότε το docker-sec πριν διαγράψει τον container, φροντίζει να καθαρίσει τα προφίλ που σχετίζονται με αυτόν. Αρχικά αφαιρεί από το προφίλ του runC τους κανόνες που σχετίζονται με τον container που είναι προς διαγραφή, δηλαδή τον αντίστοιχο κανόνα pivot_root, τον κανόνα change_profile και τέλος, ρυθμίσεις που πιθανόν έγιναν για την εξυπηρέτηση του container, όπως η άδεια να κάνει mount συγκεκριμένα volumes. Στη συνέχεια απενεργοποιεί τα προφίλ της εκκίνησης και της εκτέλεσης του container με την εντολή aa-disable[9] και τέλος, τα διαγράφει από το σύστημα αρχείων.

Προτού προχωρήσουμε είναι χρήσιμο να δούμε το πώς το runC χρησιμοποιείται σε περιπτώσεις εκτός της αρχικοποίησης του container και πώς προστατεύεται με το AppArmor προφίλ. Εδώ θα εξετάσουμε την περίπτωση χρήσης της εντολής docker exec, με την οποία ο χρήστης ζητά την εκτέλεση μίας εντολής από τον container. Αυτή η λειτουργικότητα παρέχεται μέσω του runC. Το runC, αρχικά, στέλνει ένα σήμα προς τον container για να εξακριβώσει την κατάστασή του και στη συνέχεια εκτελεί ptrace σε αυτόν, ώστε να μπορεί να ελέγξει τη λειτουργία του. Αυτές οι λειτουργίες επιτρέπονται μέσω των κανόνων του AppArmor που έχουν οριστεί στο προφίλ του runC με την χρήση του docker-sec-glob αρχείου (π.χ. κανόνες της μορφής "ptrace (trace) peer=@{CONTAINER_RUNTIME_PROFILES}", όπου η μεταβλητή CONTAINER_RUNTIME_PROFILES περιέχει τα ονόματα των προφίλ που χρησιμοποιούνται για την εκτέλεση των containers). Στην συνέχεια, το runC εκτελεί την κλήση συστήματος chroot μέσα στο mount point του container (στον φάκελο /var/lib/docker/aufs/mnt/<ID>/) και στην συνέχεια αλλάζει το AppArmor profile σε αυτό που χρησιμοποιεί ο container. Για τον λόγο αυτό είναι απαραίτητο να επιτρέπουμε στο AppArmor να μεταβαίνει σε όλα τα runtime προφίλ των containers, όπως είχε αναφερθεί νωρίτερα. Τέλος, εφόσον το runC βρίσκεται ήδη μέσα στον container, εκτελεί την εντολή που είχε ζητήσει ο χρήστης.

[9] Είναι απαραίτητο πρώτα να απενεργοποιηθεί οποιοδήποτε προφίλ θέλουμε να αφαιρέσουμε και μετά να διαγραφεί το αντίστοιχο αρχείο από τον σκληρό, διότι διαφορετικά το AppArmor μπορεί να εμφανίσει πρόβλημα κατά την είσοδο νέου προφίλ ή την τροποποίηση παλιού.

4.5 Περίοδος training

Σε αυτή την ενότητα θα περιγράψουμε αναλυτικότερα τον μηχανισμό που επιτρέπει στον χρήστη να προσαρμόζει το AppArmor προφίλ εκτέλεσης του container που επιθυμεί και στη συνέχεια θα εξετάσουμε την υλοποίηση του. Το docker-sec παρέχει στον χρήστη την δυνατότητα να εκκινεί την περίοδο training για ένα συγκεκριμένο container μέσω της εντολής train-start <CONTAINER_NAME>. Αφού ο χρήστης ξεκινήσει την περίοδο training για τον container που επιθυμεί, χρησιμοποιεί τα τμήματα της εφαρμογής που τον ενδιαφέρουν, ώστε το docker-sec να συλλέξει τις προσβάσεις και τα δικαιώματα που απαιτούνται για την εκτέλεση αυτών των λειτουργιών. Τέλος, αφού καλύψει το τμήμα της εφαρμογής που τον ενδιαφέρει[10], σηματοδοτεί το τέλος της περιόδου training μέσω της εντολής train-start <CONTAINER_NAME>. Το docker-sec, με τη λήξη της περιόδου του training αναλύει το audit log, όπου συλλέγονται οι προσβάσεις που αιτήθηκε ο container και τις προσθέτει στο προφίλ του, σταματώντας τα επιπλέον δικαιώματα που πιθανόν είχε προηγουμένως. Στην περίπτωση που ο χρήστης διαπιστώσει ότι παρέλειψε να χρησιμοποιήσει κάποιο τμήμα της εφαρμογής που τον ενδιαφέρει ή ότι χρειάζεται επιπλέον προσβάσεις, μπορεί να επαναλάβει την παραπάνω διαδικασία (εκκίνηση του training, εκτέλεση εφαρμογής, λήξη training), όσες φορές απαιτείται μέχρι να πετύχει το επιθυμητό αποτέλεσμα. Ωστόσο, κατά την περίοδο του training του προφίλ εκτέλεσης του container είναι σημαντικό να έχουν πρόσβαση μόνο εξουσιοδοτημένοι και έμπιστοι χρήστης στον container και στην εφαρμογή του container. Διαφορετικά, είναι πιθανό να καταγραφούν προσβάσεις σε πόρους του συστήματος που δεν χρειάζονται από τον container, υπονομεύοντας την ασφάλεια του συστήματος.



Εικόνα 29: Διαδικασία εκπαίδευσης προφίλ εκτέλεσης container

[10] Στην περίοδο training, το docker-sec υποστηρίζει την επανεκκίνηση του container σε περίπτωση που ο χρήστης επιθυμεί να συμπεριλάβει στο παραγόμενο προφίλ τις διαδικασίες εκκίνησης και τερματισμού της εφαρμογής.

Προκειμένου το docker-sec να παρέχει την παραπάνω λειτουργικότητα, βασίζεται στις δυνατότητες που έχει το AppArmor για auditing ορισμένων προσβάσεων που ζητά μία διεργασία. Όπως έχει προαναφερθεί, το AppArmor, μπορεί να θέσει ένα προφίλ είτε σε enforce mode, όπου οι κανόνες του προφίλ επιβάλλονται και δεν επιτρέπονται παραβάσεις, και σε complain mode, όπου οι παραβάσεις των κανόνων καταγράφονται, αλλά επιτρέπουν την εκτέλεση των αντίστοιχων κλήσεων συστήματος. Εκτός από τα παραπάνω, δίνεται η δυνατότητα μέσω κατάλληλων κανόνων να γίνει μίξη αυτών των δύο modes, παρέχοντας μεγαλύτερη ευελιξία. Συγκεκριμένα, διατηρώντας ένα προφίλ σε enforce mode, μπορούμε να επιλέξουμε να καταγράφουμε την εφαρμογή ενός κανόνα, γεγονός που μας επιτρέπει να παρακολουθούμε τις προσβάσεις που καθορίζει ο κανόνας, ενώ παράλληλα οι υπόλοιποι κανόνες συνεχίζουν να επιβάλλονται και να προστατεύουν το σύστημα. Επομένως, αξιοποιώντας αυτή τη δυνατότητα μπορούμε να παρακολουθούμε την πρόσβαση του container σε συγκεκριμένους πόρους που ορίζουμε εμείς.

Συνεπώς, όταν ο χρήστης καλέσει το docker-sec για να εκκινήσει την περίοδο training ενός container, το docker-sec μεταβάλλει το runtime προφίλ του container, ώστε να περιέχει κανόνες για παρακολούθηση των capabilities και του δικτύου, όπως οι παρακάτω:

```
audit capability,
audit network,
```

και το ξαναφορτώνει στον πυρήνα σε enforce mode. Στη συνέχεια, καθώς γίνεται χρήση της εφαρμογής που περιέχει ο container, όποτε εκτελεί κάποια κλήση συστήματος, η οποία απαιτεί δικαιώματα που περιέχονται στους audit κανόνες, το AppArmor την καταγράφει στο log. Για παράδειγμα, αν ο container προσπαθήσει να αλλάξει τον ιδιοκτήτη ενός αρχείου που δεν ανήκει στο χρήστη που εκτέλεσε το chown, θα καταγραφεί μια εγγραφή στα logs που θα περιγράφει ότι ο container και συγκεκριμένα το προφίλ εκτέλεσής του, χρησιμοποίησε το capability CAP_CHOWN. Αντίστοιχα, αν ο container ανοίξει ένα socket, θα καταγραφεί στα logs αυτή η ενέργεια μαζί με πληροφορίες, όπως ο τύπος του socket, το πρωτόκολλο που χρησιμοποιήθηκε και η εντολή που ζήτησε την δημιουργία του socket. Παρακάτω φαίνονται αποσπάσματα από τα logs που δημιουργήθηκαν από το AppArmor και χρησιμοποιήθηκαν από το docker-sec για εκπαίδευση ενός runtime προφίλ:

```
type=AVC msg=audit(1489793152.856:36326): apparmor="AUDIT" operation="capable"
profile="docker_1a336d5e02f8d64cf1d3ca5ab1fe3e08c6b61e77" pid=28732 comm="apt-get"
capability=0  capname="chown"
type=AVC msg=audit(1489793163.168:36493): apparmor="AUDIT" operation="create"
profile="docker_1a336d5e02f8d64cf1d3ca5ab1fe3e08c6b61e77" pid=28827 comm="tar"
family="unix" sock_type="stream" protocol=0 requested_mask="create" addr=none
type=AVC msg=audit(1489793156.016:36377): apparmor="AUDIT" operation="recvmsg"
profile="docker_1a336d5e02f8d64cf1d3ca5ab1fe3e08c6b61e77" pid=28737 comm="http"
laddr=172.17.0.2 lport=42418 faddr=91.189.88.162 fport=80 family="inet" sock_type="dgram "
protocol=6 requested_mask="receive"
```

Βασισμένο στις παραπάνω πληροφορίες, με την λήξη της περιόδου εκπαίδευσης, το docker-sec αναλύει τα logs και προσθέτει στο κατάλληλο προφίλ τις επιπλέον προσβάσεις που απαιτούνται, όπως προκύπτουν από την εκτέλεση της εφαρμογής κατά

την περίοδο training. Για παράδειγμα βάσει των παραπάνω logs το docker-sec θα δώσει δικαίωμα για chown (κανόνας: capability chown,), δικαίωμα χρήσης των unix sockets που είναι τύπου stream και σε datagram sockets IPv4 (κανόνας: network inet dgram). Αφού προστεθούν οι κατάλληλοι κανόνες το docker-sec αφαιρεί τους audit κανόνες και φορτώνει το προφίλ ξανά στον πυρήνα σε enforce mode. Το προφίλ που προκύπτει από αυτή την διαδικασία είναι πιο περιοριστικό από το αρχικό προφίλ που προκύπτει από τη στατική ανάλυση που εφαρμόζει το docker-sec κατά τη δημιουργία ενός container, καθώς εκείνο το προφίλ επιτρέπει όλα τα capabilities και όλες τις δικτυακές προσβάσεις[11].

Σε αυτό το σημείο αξίζει να εξετάσουμε τον λόγο για τον οποίο κατά την περίοδο εκπαίδευσης, περιορίζονται τα capabilities και τα δικαιώματα του προφίλ στο δίκτυο και όχι σε άλλους πόρους. Αρχικά, παρόλο που το Docker αρχικοποιεί τους containers ρίχνοντας τα περισσότερα capabilities, οι containers έχουν πρόσβαση σε 14 capabilities κάποια από τα οποία μπορούν να χρησιμοποιηθούν για να διαταράξουν τη λειτουργία των υπόλοιπων containers και γενικά να αποδειχτούν επικίνδυνα (π.χ. net_raw, mknod), ενώ το OCI πρότυπο ορίζει την παροχή τριών μόνο capabilities κατά την αρχικοποίηση των containers [40]. Αντίστοιχα προβλήματα μπορεί να δημιουργηθούν από την ανεξέλεγκτη χρήση του δικτύου από τους containers, με τεχνικές όπως το arp poisoning, γι' αυτό και είναι σημαντικό να περιορίσουμε την πρόσβαση του container στο δίκτυο όσο αυτό είναι εφικτό. Σε αντίθεση, όμως, με τους παραπάνω πόρους, δεν μπορούμε να περιορίσουμε περαιτέρω τα δικαιώματα για ptrace ή unmount, καθώς αυτό θα προκαλούσε διάφορα προβλήματα στη λειτουργικότητα του container, όπως για παράδειγμα στην εκτέλεση της εντολής ps. Άλλα δικαιώματα, όπως τα signals ή το D-Bus είναι απενεργοποιημένα εξ' αρχής, καθώς η χρήση τους σε περιβάλλον Docker είναι πολύ περιορισμένη, λόγω της αρχιτεκτονικής που ενθαρρύνει το Docker, δηλαδή της χρήσης μίας διεργασίας ανά container.

4.6 Αποτελέσματα

Μέσω του docker-sec προσφέρεται επιπλέον ασφάλεια στην πλατφόρμα του Docker χωρίς να απαιτείται παρέμβαση του χρήστη. Συγκεκριμένα, όπως είδαμε και νωρίτερα, μέσω της κατασκευής προφίλ για το runC, την εκκίνηση και την εκτέλεση των containers, επιτυγχάνεται μεγαλύτερη απομόνωση μεταξύ των containers. Μέσα από την χρήση διαφορετικών προφίλ για κάθε container στη φάση εκκίνησης και στη φάση εκτέλεσης, μειώνονται σημαντικά οι μοιραζόμενοι πόροι μεταξύ των containers, αφού πλέον ο καθένας εκτελείται σε διαφορετικό security context με σαφώς καθορισμένες μεταβάσεις σε νέο context. Το γεγονός αυτό προστατεύει το σύστημα από επιθέσεις κακόβουλων ή υπονομευμένων containers όπου ο επιτιθέμενος προσπαθεί να επηρεάσει την λειτουργία των υπόλοιπων containers στοχεύοντας σε αδυναμίες του πυρήνα ή του container engine, καθώς οι διεργασίες που ξεκινούν μέσα από το περιβάλλον του container περιορίζονται από κατάλληλα AppArmor προφίλ. Τα προφίλ αυτά επιτρέπουν πρόσβαση σε πόρους, όπως οι δικτυακές συσκευές, που

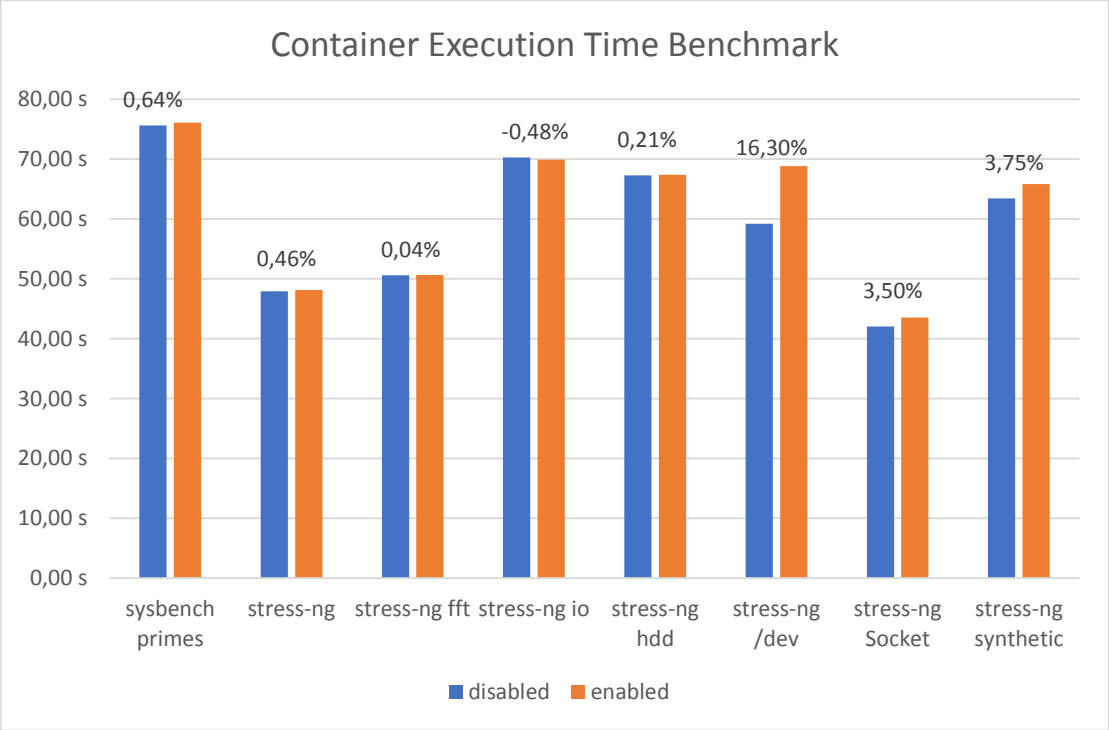
[11] Για την ακρίβεια επιτρέπει όλες τις προσβάσεις που επιτρέπει το Docker Engine στους containers, καθώς by default, το Docker περιορίζει αρκετά τα δικαιώματα των containers όπως έχει αναφερθεί στην παράγραφο 3.5.

βρίσκονται μέσα στα namespaces του container, τα οποία διαφέρουν από τα αντίστοιχα namespaces των υπολοίπων container, καθώς και από τα αρχικά namespaces του host μηχανήματος. Επίσης, τα προφίλ εκτέλεσης κατά κανόνα, δεν περιέχουν κανόνες για αλλαγή προφίλ ή δικαιώματα αποστολής σημάτων, ενώ το tracing επιτρέπεται μόνο μεταξύ διεργασιών που έχουν το ίδιο προφίλ και συνεπώς, σε περίπτωση που κάποια διεργασία από αυτές προσπαθήσει να ξεφύγει από τον περιορισμό των namespaces και των cgroups και να αποκτήσει πρόσβαση σε διεργασίες που έχουν περισσότερα δικαιώματα, θα αποτύχει χάρη στο AppArmor. Ακόμα μεγαλύτερη ασφάλεια παρέχεται αν ο χρήστης αξιοποιήσει την δυνατότητα που δίνεται από το docker-sec για training των προφίλ εκτέλεσης, ώστε οι κανόνες των capabilities και του δικτύου να γίνουν όσο το δυνατόν αυστηρότεροι, μειώνοντας αποτελεσματικά τα δικαιώματα και την επικοινωνία του container, χωρίς, ωστόσο, να περιορίζεται η λειτουργικότητα της εφαρμογής που εκτελείται μέσα σε αυτόν. Επομένως, οι κίνδυνοι από τους containers μειώνονται αρκετά και περιορίζονται, κυρίως, σε λάθος ρυθμίσεις της πλατφόρμας του Docker και των containers που μπορεί να σχετίζονται με το δίκτυο, με την αμέλεια ορισμού συγκεκριμένου ορίου χρήσης των υπολογιστικών πόρων μέσω cgroups κλπ. Για τον λόγο αυτό, εκτός από την αυτοματοποίηση της κατασκευής των προφίλ, είτε μέσω στατικής ανάλυσης είτε μέσω εκπαίδευσης του συστήματος ασφάλειας, είναι απαραίτητο ο χρήστης να ακολουθεί τις βασικές αρχές για ασφάλεια που προτείνονται από το Docker, ανάλογα με τον τύπο εφαρμογής του και με τις απαιτήσεις.

Χαρακτηριστικό παράδειγμα όπου το σύστημα που παρουσιάσαμε θα ήταν χρήσιμο σε πραγματικές συνθήκες αποτελεί η πρόσφατη αδυναμία του runC και του Docker Engine (αφορά προηγούμενες εκδόσεις της 1.12.6) CVE-2016-9962 [41]. Εξ' αιτίας της αδυναμίας αυτής, όταν ο χρήστης εκτελέσει την εντολή docker exec, η διεργασία εντός του container μπορεί να επιτεθεί μέσω ptrace στην διεργασία του runC καθώς η τελευταία εισέρχεται στον χώρο του container. Στη συνέχεια ο επιτιθέμενος αποκτά πρόσβαση για ανάγνωση και εγγραφή στους ανοιχτούς file descriptors της διεργασίας του runC, γεγονός που μπορεί να του δώσει πρόσβαση στο δίκτυο και ίσως τη δυνατότητα εκτέλεσης αυθαίρετου κώδικα. Ωστόσο μία τέτοια επίθεση δε θα είχε αποτέλεσμα εναντίον ενός συστήματος που έκανε χρήση των δυνατοτήτων του docker-sec, καθώς οι δύο αυτές διεργασίες θα βρίσκονταν σε διαφορετικά προφίλ AppArmor και μόνο η διεργασία του runC θα είχε δικαίωμα αλλαγής προφίλ (μόνο προς μία διεύθυνση) ή δικαίωμα ελέγχου της διεργασία εντός του container μέσω ptrace, αλλά όχι το αντίστροφο.

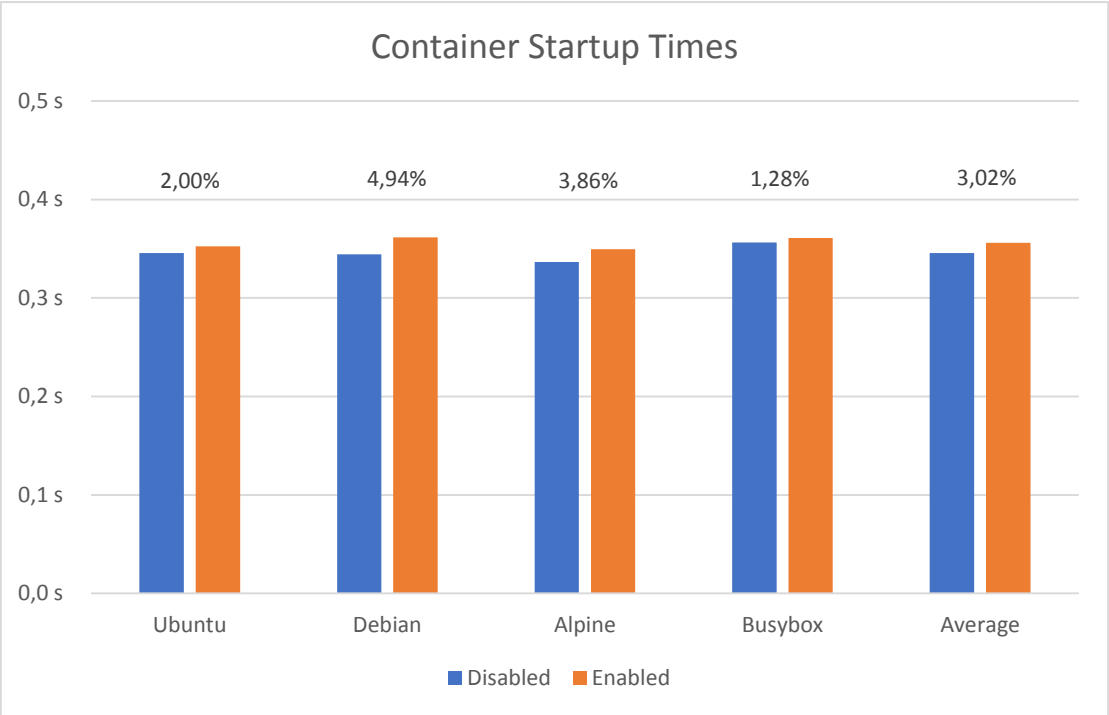
Προτού κλείσουμε, είναι σημαντικό να εξετάσουμε τις επιπτώσεις του αυτοματοποιημένου συστήματος ασφάλειας στην επίδοση του συστήματος, καθώς οι υψηλές επιδόσεις των containers και η μικρή κατανάλωση πόρων είναι, ίσως, ο σημαντικότερος λόγος για την υιοθέτηση τους. Αρχικά εκτελώντας διάφορες απαιτητικές διαδικασίες σε containers, όπως εύρεση πρώτων αριθμών και υπολογισμός fft, και μετρώντας τον χρόνο εκτέλεσής τους έχοντας ενεργό το προφίλ εκτέλεσης του container και απενεργοποιώντας το, παρατηρούμε ότι η εφαρμογή AppArmor προφίλ επιφέρει καθυστερήσεις μικρότερες του 0.5% στην κανονική λειτουργία του container. Στη συνέχεια χρονομετρώντας διάφορες διεργασίες βλέπουμε ότι μόνες περιπτώσεις που παρατηρείται αισθητή καθυστέρηση (της τάξης του 10% με 15%) είναι σε περιπτώσεις που γίνεται συνεχής έλεγχος των προσβάσεων από το AppArmor, το οποίο

μπορεί να συμβεί αν μία διεργασία ανοίγει και κλείνει συνεχώς sockets ή αρχεία, χωρίς να τα χρησιμοποιεί, κάτι που δεν είναι συχνό για πραγματικές εφαρμογές. Παρακάτω φαίνονται τα αποτελέσματα διάφορα benchmarks που εξετάζουν την επίδοση διαφορετικών τμημάτων μία εφαρμογής, όπως χρήση μόνο CPU (υπολογισμός πρώτων αριθμών, υπολογισμός fft και άλλα), χρήση μόνο IO, άνοιγμα και κλείσιμο αρχείων, συσκευών (/dev/zero) και sockets, καθώς και ένα benchmark που συνδυάζει τα παραπάνω.



Εικόνα 30: Χρόνοι εκτέλεσης διάφορων benchmark εντός των container με το AppArmor απενεργοποιημένο (μπλε) και ενεργοποιημένο (πορτοκαλί)

Στη συνέχεια, προχωρήσαμε σε μέτρηση του χρόνου εκκίνησης ενός container. Προκειμένου να μετρήσουμε με ακρίβεια τις επιπτώσεις της χρήσης τριών προφίλ κατά την εκκίνηση των container (μετάβαση από το runC σε προφίλ εκκίνησης του container και από εκεί σε προφίλ εκτέλεσης), κατασκευάσαμε containers οι οποίοι εκτελούσαν μία απλή εντολή echo κατά την εκκίνησή τους. Δημιουργήθηκαν containers βασισμένοι σε εικόνες Ubuntu, Debian, Alpine και Busybox και εκτελέστηκαν χίλιες φορές στο ίδιο περιβάλλον έχοντας ενεργοποιημένα τα AppArmor προφίλ και χίλιες χωρίς αυτά. Το αποτέλεσμα που μετρήθηκε, όπως φαίνεται και στο παρακάτω σχήμα, είναι ότι η χρήση AppArmor επέφερε καθυστερήσεις της τάξης του 3% ή, ισοδύναμα, προκάλεσε καθυστέρηση κατά 20ms.



Εικόνα 31: Χρόνοι εκκίνησης διάφορων container με το AppArmor απενεργοποιημένο (μπλε) και ενεργοποιημένο (πορτοκαλί)

Τέλος, ο χρόνος δημιουργίας των containers είναι σαφώς αυξημένος, καθώς, κατά την δημιουργία κάθε νέου container, κατασκευάζονται δύο νέα προφίλ AppArmor και ενημερώνεται ένα υπάρχον, το οποίο σημαίνει ότι πρέπει να φορτωθούν στον πυρήνα τρία διαφορετικά προφίλ, διαδικασία η οποία είναι χρονοβόρα. Αντίστοιχα, κατά την διαγραφή του container, πρέπει να αφαιρεθούν δύο προφίλ από τον πυρήνα και ένα να ενημερωθεί. Συνεπώς δημιουργείται αρκετή καθυστέρηση σε αυτές τις διεργασίες. Στα πλαίσια της εργασίας μετρήσαμε τον χρόνο εκκίνησης, εκτέλεσης μίας σύντομης εντολής echo και τον χρόνο διαγραφής ενός container βασισμένου σε εικόνα Ubuntu. Έπειτα από επανάληψη της παραπάνω διαδικασίας εκατό φορές έχοντας τα κατάλληλα προφίλ και το docker-sec απενεργοποιημένα και εκατό ενεργοποιώντας τα, παρατηρήσαμε ότι η επιβολή των AppArmor προφίλ και ο χρόνος που απαιτεί το docker-sec, έχουν ως αποτέλεσμα τον διπλασιασμό και ορισμένες φορές τον τριπλασιασμό του χρόνου κατασκευής, εκκίνησης και διαγραφής του container σαν σύνολο. Ωστόσο, στις περισσότερες εφαρμογές αυτό δεν αποτελεί πρόβλημα, καθώς οι χρόνοι αυτοί παραμένουν κάτω των δύο δευτερολέπτων, ενώ ο container μπορεί να τρέχει από μερικά λεπτά έως μερικές μέρες. Συνεπώς, όσο η διαδικασίες που εκτελούνται εντός των containers είναι αισθητά μεγαλύτερες από αυτό τον χρόνο δημιουργίας και διαγραφής του, η καθυστέρηση του ενός δευτερολέπτου εξαιτίας των AppArmor προφίλ μπορεί να θεωρηθεί αμελητέα.

5 ΣΥΝΟΨΗ

5.1 Συμπεράσματα

Στην εργασία αυτή σχεδιάσαμε ένα αυτοματοποιημένο σύστημα ασφάλειας για το περιβάλλον του Docker, στο οποίο δεν χρειάζεται παρέμβαση του χρήστη για την κατασκευή των AppArmor προφίλ. Είδαμε, ότι ακόμα και χωρίς training, το σύστημα ασφάλειας μπορεί να αμυνθεί αποτελεσματικά ενάντια σε απειλές που οφείλονται σε αδυναμίες του πυρήνα των Linux ή της υλοποίησης του container engine. Αυτές οι αδυναμίες είναι δύσκολο να προληφθούν, όπως αποδεικνύεται και από σχετικά στατιστικά των CVE καταλόγων [49], οπότε τα συστήματα MAC είναι από τις πιο αποτελεσματικές τεχνικές άμυνας στις άγνωστες απειλές που προκύπτουν εξαιτίας αυτών των κενών ασφαλείας, καθώς σε πολλές περιπτώσεις τα συστήματα HIDS είναι δύσκολο να ρυθμιστούν σωστά ώστε να είναι αποτελεσματικά, ενώ παράλληλα απαιτούν αρκετό χρόνο εκμάθησης. Αντίθετα, σε αυτήν την εργασία βλέπουμε μία λύση η οποία προσφέρει επιπλέον προστασία, χωρίς να επιβαρύνει τον χρόνο εκτέλεσης των περισσότερων εφαρμογών και χωρίς να απαιτεί επιπλέον γνώσεις από τον χρήστη του Docker. Ακόμη προσφέρεται η δυνατότητα εκπαίδευσης του συστήματος, ώστε να κατασκευαστεί ένα αυστηρότερο προφίλ εκτέλεσης του container, το οποίο μπορεί επίσης να περιορίσει σε μεγάλο βαθμό το attack surface της πλατφόρμας. Τέλος, σημαντικό είναι το γεγονός ότι το υπό εξέταση σύστημα είναι εφαρμόσιμο και σε άλλες πλατφόρμες container, αφού οι βασικές αρχές που παρουσιάσαμε στις ενότητες 2 και 4.1 ισχύουν για όλες τις υλοποιήσεις των container, άρα η αρχιτεκτονική των AppArmor προφίλ που σχεδιάσαμε μπορεί να επαναχρησιμοποιηθεί με μικρές τροποποιήσεις. Μάλιστα, στην περίπτωση των OCI containers οι αλλαγές που απαιτούνται είναι ελάχιστες (κυρίως αλλαγή των path), αφού τα AppArmor προφίλ και οι κανόνες που χρησιμοποιούνται μπορούν να παραμείνουν ίδια.

5.2 Σχετικές Εργασίες

Το θέμα της ασφάλειας των containers έχει απασχολήσει αρκετά την ερευνητική κοινότητα και έχουν δημοσιευτεί έρευνες και επιστημονικά άρθρα που εξετάζουν το θέμα από διάφορες πλευρές και από διαφορετικές οπτικές γωνίες. Ωστόσο, σε θέματα που αφορούν στην αυτοματοποίηση της ασφάλειας και ιδιαίτερα μέσω συστημάτων υποχρεωτικού ελέγχου οι δημοσιεύσεις είναι λίγες. Η πιο σχετική εργασία στον χώρο της αυτοματοποιημένης ασφάλειας για Docker περιβάλλοντα και ίσως η μόνη που κάνει χρήση του AppArmor είναι η εργασία με τίτλο «**Automatic security hardening and protection of linux containers**», η οποία προέκυψε από μία συνεργατική προσπάθεια έξι ερευνητών ασφαλείας προερχόμενους από το πανεπιστήμιο της Μπολόνιας, την IBM και το πανεπιστήμιο Ben-Gurion του Ισραήλ [34]. Σε αυτή την εργασία παρουσιάζεται το σύστημα **LiCShield** το οποίο κατασκευάζει AppArmor προφίλ βασιζόμενα σε στοιχεία που συλλέγονται με τη βοήθεια του SystemTap κατά τη διάρκεια της κατασκευής των containers και κατά τη διάρκεια της εκτέλεσης τους, ορίζοντας μία περίοδο training κατά την εκκίνηση του container. Τα στοιχεία που μαζεύονται αναλύονται και παράγονται τα κατάλληλα προφίλ. Η χρήση του LiCShield συνιστάται να γίνεται παράλληλα με την εφαρμογή κάποιου συστήματος ανίχνευσης εισβολής (Host-based Intrusion Detection System, HIDS) και μάλιστα στην εργασία

αυτή έχει δοθεί ιδιαίτερη σημασία στον τρόπο με τον οποίο τα στοιχεία που συλλέγονται από την περίοδο εκπαίδευσης των προφίλ μπορούν να αξιοποιηθούν και στην εκπαίδευση του HIDS.

Η προσέγγιση του LiCShield, έχει σημαντικές διαφορές με την δική μας, όπως θα δούμε στη συνέχεια. Σημαντικό ρόλο σε αυτό παίζει το γεγονός ότι το LiCShield έχει σχεδιαστεί με γνώμονα τα cloud deployments του Docker. Συνεπώς, σε αντίθεση με το docker-sec, δεν προχωρά σε στατική ανάλυση των πληροφοριών που μπορεί να συλλέξει από το Docker και τις παραμέτρους με τις οποίες ο χρήστης δημιουργεί τον container. Μία άλλη σημαντική διαφορά είναι ότι το LiCShield χρησιμοποιεί ένα προφίλ για όλες τις διεργασίες που εκκινούν τους containers, κάτι το οποίο θα μπορούσε να αποδειχτεί επικίνδυνο για την ασφάλεια του συστήματος. Ακόμα, το LiCShield προσπαθεί να καλύψει την προστασία όχι μόνο του host, αλλά και του ίδιου του container, γεγονός που καθιστά απαραίτητη την ύπαρξη περιόδου training. Για τον λόγο αυτό δίνει μεγάλη έμφαση στην εύρεση όλων των αρχείων που χρησιμοποιεί η εφαρμογή και προσπαθεί μέσω προκαθορισμένων glob pattern να δώσει πρόσβαση σε αυτά κρατώντας τον αριθμό των κανόνων σχετικά χαμηλό, ωστόσο, δεν περιορίζει capabilities και τις δικτυακές προσβάσεις των containers, σε αντίθεση με το εργαλείο docker-sec.

Εκτός από την παραπάνω εργασία, στον ευρύτερο τομέα της ασφάλειας του Docker οικοσυστήματος, έχουν γίνει σημαντικές προσπάθειες και έχουν αναπτυχθεί αρκετά εργαλεία που μπορούν να βοηθήσουν στην προστασία των Containers. Μεγάλο μέρος από αυτά λειτουργεί ως σύστημα ανίχνευσης εισβολών είτε σε επίπεδο λειτουργικού συστήματος παρακολουθώντας προσβάσεις σε ευαίσθητα αρχεία ή συγκεκριμένες κλήσεις συστήματος, όπως το Starlight της IBM [42], είτε σε επίπεδο δικτύου παρακολουθώντας συμβάντα από όλους τους υπολογιστές ενός υποδικτύου και εξάγοντας συμπεράσματα για το σύστημα, όπως το Bluemix. Για αυτά τα συστήματα έχουν μελετηθεί και αναπτυχθεί είτε αυτόματες τεχνικές ανίχνευσης είτε τεχνικές που ευνοούν τον χειροκίνητο ορισμό πολιτικών, άλλοτε εστιάζοντας στην ταχύτητα του συστήματος, άλλοτε στην παροχή μεγαλύτερης ασφάλειας και άλλοτε στην εύκολη χρήση του συστήματος. Αξιόλογο έργο έχει επιτελεστεί στον τομέα της ανάπτυξης καλών πρακτικών για την χρήση των container (π.χ. η έκθεση για την ασφάλεια των containers από το NCC group: "Understanding and Hardening Linux Containers" [43]). Σημαντικότερη, ίσως, συνεισφορά σε αυτό τον τομέα είναι η λίστα από το CIS με σημαντικά σημεία ελέγχου για κάθε εγκατάσταση Docker [44], η οποία υλοποιείται από το αυτόματο εργαλείο docker-bench-security [45], το οποίο ελέγχει περισσότερα από εκατό σημεία της Docker πλατφόρμας που μπορεί να υπονομεύσουν την ασφάλεια του συστήματος. Τέλος, προτάσεις γίνονται για την ενσωμάτωση των συστημάτων υποχρεωτικού ελέγχου στην διαδικασία κατασκευής των container από dockerfiles, όπου ο χρήστης θα μπορούσε με ενιαίο τρόπο να ορίζει την κατασκευή του container και τις πολιτικές ασφαλείας που πρέπει να ακολουθεί, ώστε οι πολιτικές ασφαλείας να είναι μέρος του container και συνεπώς, να μεταφέρονται εύκολα μαζί του [46].

5.3 Επεκτάσεις

Σε αυτή την εργασία αναπτύξαμε ένα σύστημα το οποίο μπορεί με αυτοματοποιημένο τρόπο να ενισχύσει την ασφάλεια της Docker πλατφόρμας, προστατεύοντας την αποτελεσματικά από επιθέσεις που έχουν ως αφετηρία έναν υπονομευμένο container και στοχεύουν στην κατάληψη του host ή των υπόλοιπων containers. Ωστόσο, μένουν ακόμα αρκετές πτυχές που θα πρέπει να καλυφθούν. Αρχικά, η υλοποίηση θα πρέπει να εμπλουτιστεί ώστε να καλύπτει μεγαλύτερο μέρος της λειτουργικότητας του Docker, όπως διαφορετικοί storage drivers πέραν του AUFS, απομακρυσμένους clients (υλοποιήση του REST API του Docker), καθώς και παλαιότερες εκδόσεις του Docker. Ακόμη θα ήταν χρήσιμη η προσπάθεια μείωσης του χρόνου που απαιτείται για τη δημιουργία των προφίλ, ώστε να ελαττωθεί ο χρόνος εκκίνησης των container. Επίσης σημαντική θα ήταν η επέκταση της παραγωγής κανόνων κατά την φάση training του συστήματος, ώστε να περιλαμβάνει κανόνες για τα αρχεία που είναι απαραίτητα για τον εκάστοτε container, οι οποίοι θα παράγονται δυναμικά μέσω έξυπνων αλγορίθμων, με στόχο την βελτιστοποίηση του αριθμού των παραγόμενων κανόνων και της αυστηρότητας του παραγόμενου προφίλ. Τέλος, θα μπορούσε να υλοποιηθεί ένας μηχανισμός που θα επιτρέπει το πακετάρισμα των προφίλ εκτέλεσης, όπως αυτό προέκυψε από την διαδικασία εκπαίδευσης, προσθήκη συμβατή με την φιλοσοφία των διάφορων μηχανισμών για container orchestration που έχουν αναπτυχθεί, όπως το Kubernetes και το Mesos.

Σημαντική συνεισφορά στον τομέα της ασφάλειας των containers, θα ήταν η υλοποίηση ενός αντίστοιχου μηχανισμού για άλλες container πλατφόρμες, ιδίως για machine containers, όπως το LXC, ενώ, όπως προαναφέρθηκε, η γενίκευση της υλοποίησης για οποιαδήποτε OCI συμβατή υλοποίηση container θα ήταν αρκετά εύκολη, αλλά και ωφέλιμη.

Εκτός από τα παραπάνω, θα ήταν χρήσιμο να συγκριθεί η ασφάλεια που προσφέρεται από το docker-sec με αντίστοιχες υλοποιήσεις που χρησιμοποιούν SELinux. Ακόμη, ωφέλιμο θα ήταν να εξεταστούν πτυχές όπως η ευκολία υλοποίησης, η ευκολία χρήσης και η επιδόσεις των δύο συστημάτων.

6 ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] N. RALEIGH, «Linux Container Deployment Plans Grow, While Concerns about Container Security and Certification Linger, Red Hat Survey Finds» Red Hat, Inc., 19 June 2017.

[2] Wikemedia Foundation, «Wikepedia» https://en.wikipedia.org/wiki/Operating-system-level_virtualization.

[3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer και J. Wilkes, «Borg, Omega, and Kubernetes» *ACM Queue,* τόμ. 14, p. 24, 2016.

[4] J. Beda, «Cotnainers At Scale» 22 May 2014. https://speakerdeck.com/jbeda/containers-at-scale.

[5] T. P. Morgan, «Inside eBay's Shift To Kubernetes And Containers Atop OpenStack» 12 November 2015. https://www.nextplatform.com/2015/11/12/inside-ebays-shift-to-kubernetes-and-containers-atop-openstack/.

[6] U. Cohen, « Containers, microservices, and orchestrating the whole symphony » 12 December 2014. https://opensource.com/business/14/12/containers-microservices-and-orchestrating-whole-symphony.

[7] A. Kanso, H. Huang και A. Gherbi, «Can Linux Containers Clustering Solutions offer High Availablity?» 2016.

[8] J. Tee, «Don't let unfounded Docker fears deter container technology adoption» 12 July 2016. http://www.theserverside.com/news/450300160/Dont-let-unfounded-Docker-fears-deter-container-technology-adoption.

[9] S. J. Bigelow, «Virtual security tactics for Type 1 and Type 2 hypervisors» July 2013. http://searchservervirtualization.techtarget.com/answer/Virtual-security-tactics-for-Type-1-and-Type-2-hypervisors.

[10] M. Luft, «Analysis of Hypervisor Breakouts» 20 May 2013. https://insinuator.net/2013/05/analysis-of-hypervisor-breakouts/.

[11] Microsoft, «Microsoft Security Bulletin MS15-068 - Critical» Microsoft, 14 July 2015. https://technet.microsoft.com/en-us/library/security/ms15-068.aspx.

[12] K. Páral, «KVM DISK PERFORMANCE: IDE VS VIRTIO» 12 September 2012. https://kparal.wordpress.com/2012/09/12/kvm-disk-performance-ide-vs-virtio/.

[13] G. Danti, «KVM VirtIO paravirtualized drivers: why they matter» 8 January 2014. http://www.ilsistemista.net/index.php/virtualization/42-kvm-virtio-paravirtualized-drivers-why-they-matter.html?limitstart=0.

[14] G. J. Popek και R. P. Goldberg, «Formal Requirements for virtualizable third generation architectures,» *Communications of the ACM*, τόμ. 17, αρ. 7, p. 10, 1974.

[15] S. Meier, IBM Systems Virtualization: Servers,Storage, and Software, ibm.com/redbooks, 2008.

[16] S. E. Hallyn, «[PATCH] namespaces: utsname: implement CLONE_NEWUTS flag,» 02 October 2006. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=071df104f808b8195c40643dcb4d060681742e29.

[17] Linux, «Linux 3.8,» 18 February 2013. https://kernelnewbies.org/Linux_3.8#head-fc2604c967c200a26f336942caee2440a2a4099c.

[18] B. Cheswick, « An Evening with Berferd: In Which a Cracker is Lured, Endured, and Studied» σε *USENIX Summer Conference*, San Francisco, California, 1991.

[19] J. Corbet, «Notes from a container,» lwn.net, 29 October 2007. [Ηλεκτρονικό]. Available: https://lwn.net/Articles/256389/. [Πρόσβαση 23 February 2017].

[20] L. Torvalds, «[PATCH 12/18] shared mount handling: bind and rbind» 16 November 2005. http://yarchive.net/comp/linux/pivot_root.html..

[21] Linux , «CHROOT(2) Linux Programmer's Manual» Linux man-pages project, 12 December 2016. http://man7.org/linux/man-pages/man2/chroot.2.html.

[22] M. Kerrisk, «Anatomy of a user namespaces vulnerability» 20 March 2013. https://lwn.net/Articles/543273/.

[23] M. Kerrisk, «Mount namespaces and shared subtrees» lwn.net, 8 June 2016. https://lwn.net/Articles/689856/.

[24] «SVIPC(7) Linux Programmer's Manual,» http://man7.org/linux/man-pages/man7/svipc.7.html..

[25] D. Vyukov, «CVE-2015-7613» Common Vulnerabilities and Exposures, 01 October 2015. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7613.

[26] J.-T. L. Bigot, «Introduction to Linux namespaces - Part 2: IPC» 18 December 2013. https://blog.yadutaf.fr/2013/12/28/introduction-to-linux-namespaces-part-2-ipc/.

[27] M. Kerrisk, «Namespaces in operation, part 2: the namespaces API» LWN.net, 8 January 2013. https://lwn.net/Articles/531381/.

[28] Linux man-pages project, «PID_NAMESPACES(7) Linux Programmer's Manual» 12 December 2016. http://man7.org/linux/man-pages/man7/pid_namespaces.7.html.

[29] J. Corbet, «Network namespaces» 30 January 2007. https://lwn.net/Articles/219794/.

[30] M. Larabel, «CGroup Namespaces Support Set For Linux 4.6 Kernel» 20 March 2016. https://www.phoronix.com/scan.php?page=news_item&px=CGroup-Namespaces-Linux-4.6.

[31] A. Kali, «CGroup Namespaces» 13 October 2014. https://lwn.net/Articles/616099/.

[32] Linux man-pages project, «CGROUPS(7) Linux Programmer's Manual» 12 December 2016. http://man7.org/linux/man-pages/man7/cgroups.7.html.

[33] A. Lutomirski, «[RFC] capabilities: Ambient capabilities» 12 March 2015 . https://lwn.net/Articles/636533/.

[34] M. Mattetti, A. Shulman-Pelegy, Y. Allouchey, A. Corradi, S. Dolevz και L. Foschini, «Security hardening of linux containers and their workloads» 2015.

[35] J. Rudenberg, «Docker Image Insecurity» 23 December 2014. https://titanous.com/posts/docker-insecurity.

[36] Cluster HQ, DevOps.com, «The Current State Of Container Usage, Identifying and eliminating bariers to adoption» Cluster HQ, 2015.

[37] ClusterHQ, DevOps.com, «Container Market Adoption,» ClusterHQ, 2016.

[38] jancorg, «LibContainer Overview» 3 January 2015. http://jancorg.github.io/blog/2015/01/03/libcontainer-overview/.

[39] T. Jernigan, «Docker 1.11 et plus: Engine is now built on runC and containerd» 17 May 2016. https://medium.com/@tiffanyfayj/docker-1-11-et-plus-engine-is-now-built-on-runc-and-containerd-a6d06d7e80ef#.hq47m0l6i.

[40] D. Walsh, «Secure Your Containers with this One Weird Trick» 17 October 2016. http://rhelblog.redhat.com/2016/10/17/secure-your-containers-with-this-one-weird-trick/.

[41] Common Vulnerabilities and Exposures, «CVE-2016-9962» 15 December 2016. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9962.

[42] S. Barlev, Z. Basil, S. Kohanim, R. Peleg, S. Regev και A. Shulman-Peleg, «Secure yet usable: Protecting servers and Linux» *Ibm Journal of Research and Development*, pp. 1-12, 2016.

[43] A. Grattafiori, «Understanding and Hardening Linux Containers» 29 June 2016. https://www.nccgroup.trust/us/our-research/understanding-and-hardening-linux-containers/.

[44] Center for Internet Security, «CIS Docker 1.13.0 Benchmark» 19 January 2017. https://benchmarks.cisecurity.org/tools2/docker/CIS_Docker_1.13.0_Benchmark_v1.0.0.pdf.

[45] «Docker Bench for Security» May 2015. https://github.com/docker/docker-bench-security.

[46] E. Bacis, S. Mutti, S. Capelli και S. Paraboschi, «DockerPolicyModules: Mandatory Access Control for Docker containers» 2015.

[47] J. Rosland, «Container OS Comparison,» 02 July 2015. https://blog.codeship.com/container-os-comparison/.

[48] A. Konovalov, «Linux kernel: CVE-2017-6074: DCCP double-free vulnerability (local root)» 22 Febryary 2017. http://seclists.org/oss-sec/2017/q1/471.

[49] CVE details, http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33