

POLITECHNIKA POZNAŃSKA

WYDZIAŁ INFORMATYKI

INSTYTUT INFORMATYKI

Praca dyplomowa magisterska

**Analiza bezpieczeństwa mechanizmu
wirtualnych kontenerów Docker**

Jakub Nurski

Promotor:

dr inż. Michał Szychowiak

Poznań, 2019

Spis treści

Spis treści	2
1 Wstęp	5
1.1 Wirtualizacja środowiska przetwarzania	5
1.2 Cel i zakres pracy	5
1.3 Struktura pracy	6
2 Wirtualizacja systemów operacyjnych	6
2.1 Maszyny wirtualne	7
2.2 Kontenery	8
2.3 Unikernela	10
2.4 Porównanie	11
3 Ekosystem Dockera	11
3.1 Specyfikacje Dockera	12
3.2 Elementy wewnętrzne Dockera	14
3.3 Demon Dockera	15
3.4 Docker Hub	16
3.5 Systemy operacyjne dedykowane Dockerowi	16
3.6 Docker dla systemów operacyjnych nieopartych na Linuxie	17
3.6.1 Windows	17
3.6.2 MacOS	18
4 Architektura zabezpieczeń kontenerów Dockera	18
4.1 Izolacja	18
4.2 Utwardzanie jądra systemu operacyjnego gospodarza	19
4.2.1 SELinux	20
4.2.2 AppArmor	21
4.3 Wykorzystanie mechanizmów kryptograficznych	22
5 Powszechnie przypadki użycia Dockera	23
5.1 Zalecany przypadek użycia	23
5.2 Rozpowszechniony przypadek użycia	24
5.3 Przypadek użycia dostawców chmurowych	25

6 Analiza Dockera zorientowana na podatności	28
6.1 Model agresora	28
6.2 Identyfikacja podatności	30
6.3 Podatności w konfiguracji	30
6.3.1 Podatności	30
6.3.2 Ataki	31
6.3.3 Zapobieganie atakom	31
6.4 Docker jako menadżer pakietów	35
6.4.1 Podatności	35
6.4.2 Ataki	35
6.4.3 Zapobieganie atakom	36
6.5 Zautomatyzowane rurociągi CI/CD	36
6.5.1 Podatności	36
6.5.2 Ataki	38
6.6 Podatności wewnętrz obrazu	39
6.6.1 Podatności	39
6.6.2 Ataki	39
6.6.3 Zapobieganie atakom	39
6.7 Podatności usługi Docker	40
6.7.1 Podatności	40
6.7.2 Ataki	41
6.8 Podatności jądra Linuxa	41
6.9 Ocena podatności	41
6.10 Alternatywy dla Dockera	41
7 Realizacja aplikacji	43
7.1 Motywacja	43
7.2 Wykorzystane narzędzia	44
7.2.1 Berkeley Packet Filter	44
7.2.2 Extended Berkeley Packet Filter	44
7.2.3 BPF Compiler Collection	45
7.2.4 Python3.7	45
7.2.5 Docker SDK for Python	46
7.3 Schemat działania	46
7.3.1 Parsowanie argumentów linii poleceń	46

7.3.2	Uruchomienie kontenera	47
7.3.3	Śledzenie wywołań systemowych	48
7.3.4	Analiza wywołań systemowych	51
7.3.5	Zwolnienie zasobów i zakończenie programu	51
7.4	Wyniki przykładowej analizy	52
8	Zakończenie	53
8.1	Zebrane doświadczenia	53
8.2	Podsumowanie	54
9	Bibliografia	55
A	Dodatek A	60

1. Wstęp

1.1. Wirtualizacja środowiska przetwarzania

W ostatnich dwóch dekadach nastąpił gwałtowny rozwój obszaru technologii wirtualizacji, które umożliwiają partycjonowanie fizycznego systemu komputerowego w wiele odizolowanych środowisk wirtualnych. Istniejące technologie wirtualizacji oferują znaczne korzyści szybko napędzające ich rozwój. Jednym z najbardziej powszechnych powodów stosowania wirtualizacji jest tworzenie chmurowych centrów obliczeniowych, z którymi na co dzień styczność ma wielu programistów. Amazon AWS, Microsoft Azure czy Google Cloud Platform to tylko niektórzy z popularnych dostawców chmurowych wykorzystujących wirtualizację.

Rosnące wykorzystanie wirtualizacji stworzyło zapotrzebowanie na technologie, które zapewnią wydajne, skalowalne i bezpieczne środowiska serwerowe. W przeciągu poprzednich lat pojawiło się wiele rozwiązań, które mogą być zakwalifikowane do jednej z dwóch głównych kategorii: wirtualizacji bazującej na konteneryzacji i wirtualizacji bazującej na nadzorcach. Potrzeba coraz krótszych cykli rozwoju oprogramowania, ciągłych aktualizacji oraz zmniejszania kosztów w infrastrukturach bazujących na chmurze obliczeniowej spowodowała, że z tych dwóch kategorii to wirtualizacje bazujące na kontenerach zyskały ogromną popularność. Oferują one większą elastyczność niż maszyny wirtualne oraz wydajność zbliżoną do natywnie uruchumionego systemu operacyjnego.

Docker jest obecnie najpopularniejszym na rynku rozwiązaniem spośród wszystkich mechanizmów kontenerowych. W szczególności, Docker jest kompletnym narzędziem pozwalającym na konteneryzację i dostarczanie oprogramowania. Łączy on pojęcie wirtualizacji systemu operacyjnego z tematami przenośności oprogramowania, modularizacji systemu oraz wersjonowania. Tym samym idealnie wpisuje się w obecnie popularną "filozofię" tworzenia oprogramowania DevOps. Jednakże, wirtualizacja oparta na kontenerach posiada również wady, a istotna część z nich dotyczy bezpieczeństwa tych rozwiązań.

1.2. Cel i zakres pracy

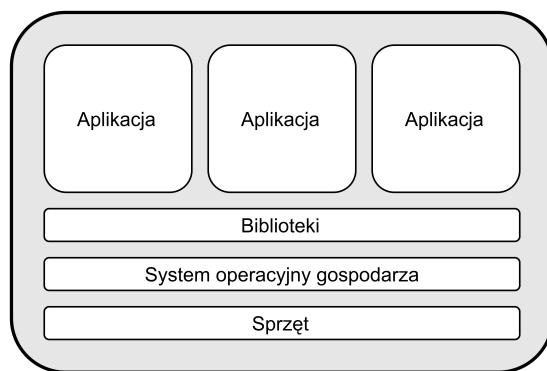
Celem pracy jest analiza rozwiązań wirtualizacji systemów operacyjnych z naciskiem na aspekt bezpieczeństwa przy wykorzystaniu środowiska wirtualizacji Docker. Zwieńczeniem analizy jest projekt i implementacja aplikacji wspomagającej zautomatyzowaną analizę kontenerów Docker pod kątem wykrywania przydziału zbyt dużej ilości uprawnień danemu kontenerowi Dockera. Stworzenie takiej aplikacji motywowane jest brakiem odpowiednich rozwiązań dostępnych na rynku, pomimo tak dużej popularności środowiska Docker oraz krytycznego znaczenia

prawidłowego przydziału uprawnień w zarządzaniu bezpieczeństwem.

1.3. Struktura pracy

Wstęp pracy przedstawia kontekst w jakim umiejscowiona jest tematyka związana ze środowiskiem narzędzia Docker. Rozdział drugi przedstawia najpopularniejsze metody wirtualizacji systemów operacyjnych, a także porównuje istniejące alternatywy kontenerów Dockera. Kolejne dwa rozdziały opisują poszczególne części ekosystemu Dockera oraz architekturę zastosowanych w nich zabezpieczeń. Rozdział piąty omawia najpowszechniejsze przypadku użycia Dockera, które rozdział szósty łączy z komponentami ekosystemu Dockera w celu przedstawienia analizy podatności. Ostatni rozdział przedstawia implementację analizatora uprawnień kontenerów Dockera.

2. Wirtualizacja systemów operacyjnych

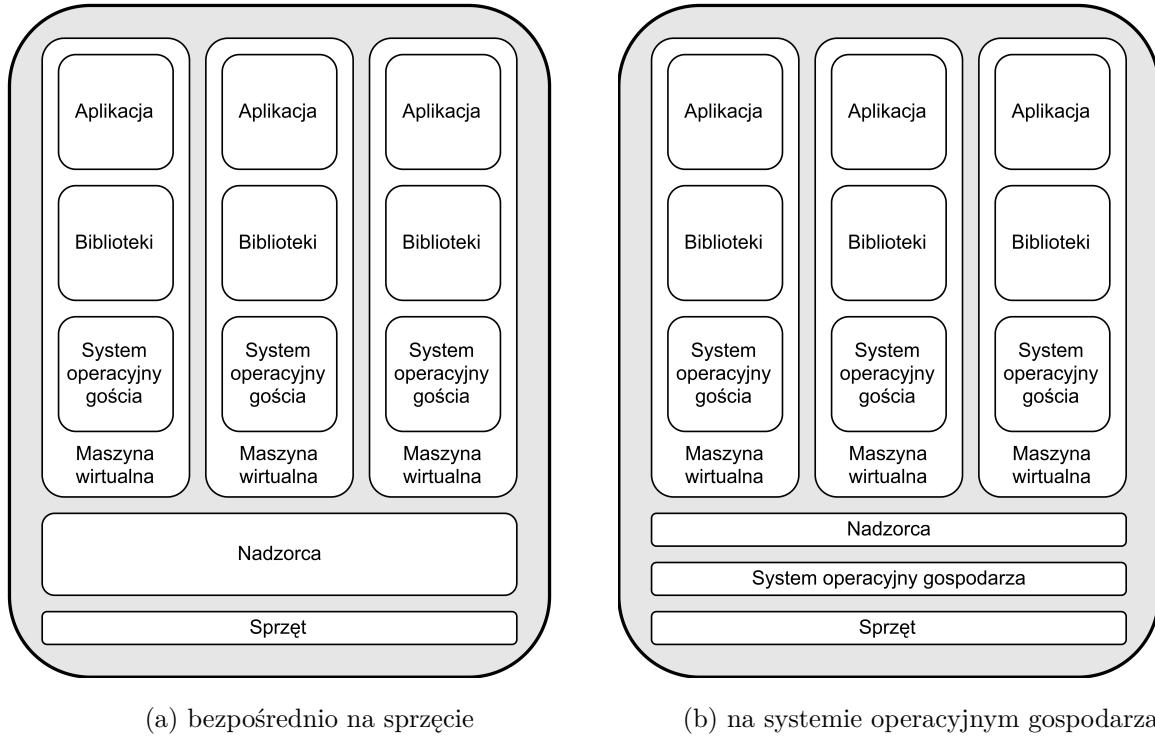


Rysunek 2.1: Schemat infrastruktury uruchomionej natywnie

Rozdział ten przedstawia dostępne rozwiązania wirtualizacji systemów operacyjnych. Zastosowania chmurowe zwykle wykorzystują wirtualizację jako element zabezpieczeń, umożliwiając łatwiejsze zarządzanie bezpieczeństwem złożonego klastra lub infrastruktury chmurowej. Poprawnie skonfigurowana architektura usprawnia monitorowanie poprawności systemu gościa i warstwy wirtualizacji oraz chroni je przed większością rodzajów ataków, zachowując jednocześnie pełną przejrzystość dla dostawcy i użytkownika usługi. Co więcej, pozwala na lokalne reagowanie na naruszenia bezpieczeństwa oraz powiadamianie o takich wydarzeniach.

Możliwe jest także stworzenie architektury chmurowej całkowicie w oparciu o rozwiązania Open Source. Wyniki analiz pokazują, że takie podejście są skuteczne, a kompromis pomiędzy wydajnością i bezpieczeństwem akceptowalny [LOMB 2010]. Jednakże, biorąc pod uwagę pewne podstawowe ograniczenia, takie jak narzucona wydajność, elastyczność i skalowalność, pojawiły się alternatywne rozwiązania dla wirtualizacji w postaci kontenerów i unikerneli.

2.1. Maszyny wirtualne



Rysunek 2.2: Schemat infrastruktury uruchomionej na maszynach wirtualnych z nadzorcą

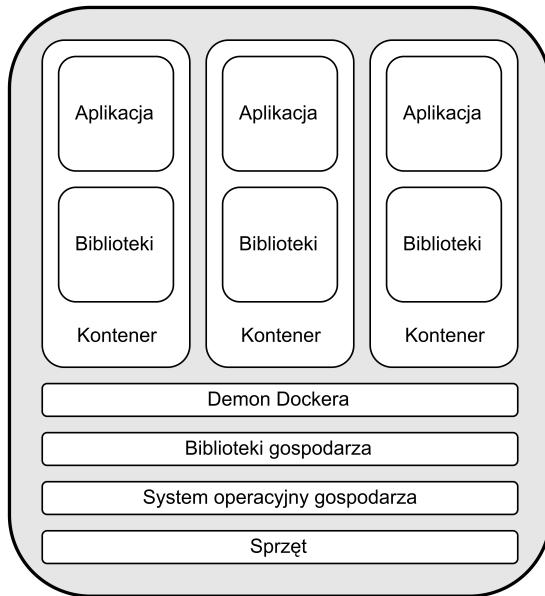
Maszyny wirtualne są najczęstszym sposobem wirtualizacji przetwarzania w chmurze: są w pełni funkcjonalnym systemem operacyjnym, działającym na emulowanej warstwie sprzętowej, zapewnianej przez leżącego u podstaw nadzorę. Nadzorca może działać bezpośrednio na sprzęcie (np. Xen)(rys. 2.2a) lub na systemie operacyjnym gospodarza (np. KVM)(rys. 2.2b). Maszynę wirtualną można klonować, instalować w ciągu kilku minut i uruchamiać w ciągu kilku sekund, co pozwala na tworzenie całego stosu technologicznego za pomocą skoncentrowanych narzędzi. Jednak obecność dwóch systemów operacyjnych (gospodarza i gościa) wraz z dodatkową, wirtualną warstwą sprzętową wprowadza znaczne narzuty w wydajności. Sprzętowa obsługa wirtualizacji radykalnie zmniejsza przytoczony narzut, ale wydajność jest daleka od operowania bezpośrednio na sprzęcie, szczególnie w przypadku operacji wejścia/wyjścia.

Wyniki badań wyraźnie pokazują, że wydajność nadzorcy działającego na systemie operacyjnym gospodarza znacznie się poprawiła w ciągu ostatnich kilku lat. Niemniej jednak szczególnie wydajność operacji na dysku twardym może nadal stanowić wąskie gardło w przypadku niektórych rodzajów aplikacji [MORA 2015].

Podobna sytuacja dotyczy nadzorcy działającego bezpośrednio na sprzęcie. W jego przypadku za słabą wydajność operacji na dysku odpowiadają parasterowniki, które pomimo wielu usprawnień wprowadzanych od kilku lat nie są w stanie osiągnąć wyników zbliżonych do innych

rozwiązań [XAVI 2013].

2.2. Kontenery



Rysunek 2.3: Schemat infrastruktury uruchomionej na kontenerach

Kontenery (rys. 2.3) zapewniają prawie, że natywną wydajność w przeciwieństwie do wirtualizacji [MORA 2015][XAVI 2013] z dodatkową możliwością płynnego uruchamiania wielu aplikacji na tej samej maszynie. Nowe instancje kontenerów można tworzyć niemal natychmiastowo aby sprostać szczytowi zapotrzebowania klientów. Kontenery istniały od dawna w różnych formach, które różnią się poziomem zapewnianej izolacji. Na przykład, *BSD jails* [KAMP 2010] i *chroot* można uznać za wczesną formę technologii kontenerowej. Najnowsze rozwiązania kontenerowe oparte na systemie Linux, bazują w dużej mierze na wsparciu jądra, biblioteki przestrzeni użytkownika zapewniającej interfejs do wywołań systemowych i aplikacji dla użytkownika rozwiązania kontenerowego. Istnieją dwie główne implementacje kontenerów: implementacja oparta na LXC, z wykorzystaniem narzędzi *cgroups* i *namespaces* oraz łatka jądra o nazwie OpenVZ. Spis poszczególnych technologii kontenerowych przedstawia tabela 2.1.

Kontenery można wykorzystać w środowisku wielu dzierżawców (ang. multi-tenant environment), wykorzystując w ten sposób współdzielenie zasobów w celu zwiększenia średniego zużycia sprzętu. Cel ten osiąga się poprzez współdzielenie jądra z maszyną gospodarza. W przeciwieństwie do maszyn wirtualnych, kontenery nie uruchamiają własnego jądra, ale działają bezpośrednio na jądrze gospodarza. Skracą to ścieżkę wykonywania wywołań systemowych poprzez usunięcie warstwy jądra gościa i wirtualnej warstwy sprzętowej. Dodatkowo, kontenery mogą współdzielić zasoby oprogramowania (np. biblioteki) z gospodarzem, unikając powielania kodu. Brak dodatkowego jądra i niektórych bibliotek systemowych (udostępnianych przez go-

Technologia	Podstawa	Biblioteka	Zależności jądra	Dodatkowe zależności
LXC	LXC	liblxc	cgroups, namespaces, capabilities	golang
LXD	LXC	liblxc	cgroups, namespaces, capabilities	LXC, golang
Docker	LXC	libcontainer	cgroups, namespaces, capabilities, kernel w wersji 3.10+	iptables, perl, AppArmor, sqlite, golang
Rocket	LXC	AppContainer	cgroups, namespaces, capabilities, kernel w wersji 3.8+	cpio, golang, squashfs, gpg
Warden	LXC	<i>brak</i>	cgroups, namespaces	debootstrap, rake
OpenVZ	OpenVZ	libCT	niestandardowa aktualizacja jądra	specific components: CRIU, ploop, VCMMD

Tabela 2.1: Rozwiązania kontenerowe

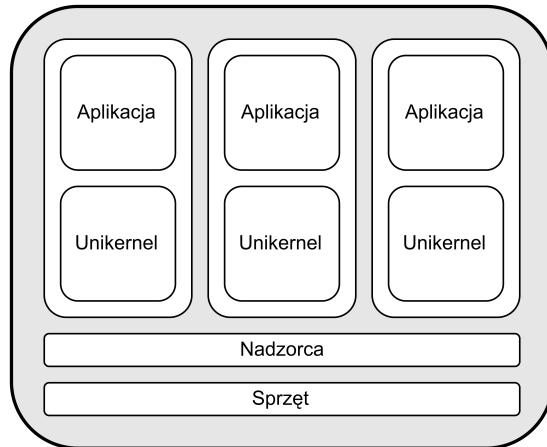
spodarza) sprawia, że kontenery są bardzo lekkie (rozmiary obrazów mogą się zmniejszyć do kilku megabajtów). Dzięki temu proces uruchamiania jest bardzo szybki (nawet poniżej jednej sekundy [ZHEN 2015]).

Pozwala to na ponowne uruchamianie kontenerów na żądanie lub szybkie ich przenoszenie. Jak pokazują niektóre przykłady zastosowań technologii kontenerowych możliwe jest tworzenie aplikacji, które dynamicznie uruchamiają i usuwają kontenery wykorzystywane do wykonywania krótkich zadań. Wydajność takiego rozwiązania jest jednak ograniczona i przy systemach wykorzystujących tysiące kontenerów koszty abstrakcji zaczynają być wyższe niż zyski wynikające z tak rozproszonej architektury [ZHEN 2015].

Działanie wielu takich kontenerów – domyślnie nieświadomych siebie nawzajem pomimo współużytkowania jądra gospodarza – wymaga jednak dodatkowej izolacji. Popularność Dockera jako jednego z przykładów rozwiązania kontenerowego, w połączeniu z rozszerzonymi uprawnieniami na uruchamianych maszynach, czyni go celem o wysokiej wartości dla atakującego. Dlatego też w kolejnych rozdziałach praca koncentruje się na podatnościach, na które jest narażony,

i możliwych środkach zaradczych.

2.3. Unikernel



Rysunek 2.4: Schemat infrastruktury uruchomionej na unikernelach

Unikernel (rys. 2.4) składają się z bardzo lekkich systemów operacyjnych, specjalnie zaprojektowanych do działania na maszynie wirtualnej. Unikernela zostały pierwotnie zaprojektowane do działania na nadzorcach Xen. Są bardziej wydajne niż klasyczna maszyna wirtualna gdyż skracają ścieżkę wykonywania wywołań systemowych: nie korzystają z parasterowników, a nadzorca nie emuluje warstwy sprzętowej. Interakcję osiąga się za pomocą specjalnego interfejsu programistycznego. Umożliwia on optymalizacje, które były niemożliwe w przypadku starszego modelu maszyn wirtualnych, gdzie niezmodyfikowane jądro działało na emulowanym sprzęcie. Jednakże, te modyfikacje uzależniają unikernele od nadzorcy, na którym działają.

W chwili obecnej większość rozwoju unikerneli jest związana z nadzorcą Xen. Ponadto, unikernela są zwykle zaprojektowane do uruchamiania programów napisanych w określonym języku programowania. W związku z tym osadzają one tylko biblioteki potrzebne dla konkretnego języka i pod niego są zoptymalizowane (np. HaLVM dla Haskell, OSv dla Java, C, C++, Ruby i Node.js, LING dla Erlang). Wszystkie powyższe zmiany zmniejszają narzut w stosunku do maszyn wirtualnych [XEN 2019].

Wyniki analiz pokazują, że unikernela w rzeczywistości osiągają lepszą wydajność niż maszyny wirtualne, a ponadto rozwiązuje niektóre problemy związane z bezpieczeństwem, na jakie narażone są kontenery. Zalety dotyczące bezpieczeństwa wynikają z tego, że aplikacje działające w unikernalach nie współpracują z jądrami systemu operacyjnego. Badania stwierdzają, że implementacje unikerneli nie są wystarczająco dojrzałe, aby można je było szeroko rozpowszechnić w produkcji. Próby stworzenia środowisk produkcyjnych wymagały dużej ilości optymalizacji pod konkretne rozwiązania, a brak porządnego narzędzia deweloperskiego znacznie wydłużał czas

rozwoju. Uważa się jednak unikernie za poważnych konkurentów dla kontenerów w dłuższej perspektywie czasu, zarówno ze względów bezpieczeństwa, jak i ze względu na ich bardzo krótki czas uruchamiania [KUEN 2017][MADH 2013].

Unikernie mogą w przyszłości oferować znaczne korzyści dla użytkowników biznesowych, którzy z reguły starają się zmniejszać koszty poprzez minimalne wydatki na rozwiązania bezpieczeństwa i prywatności danych. Unikernie poprzez swoją idempotentność działania ograniczają liczbę błędów wykonywanych przez deweloperów, a tym samym łagodzą koszty deweloperskie. Ponadto, dzięki redukcji narzutów wirtualizacji pozwalają na lepsze wykorzystanie zasobów, prowadząc do jeszcze większych oszczędności w obszarze infrastruktury [DUNC 2017].

2.4. Porównanie

Każda z powyższych alternatyw zapewnia inny kompromis pomiędzy wieloma czynnikami: wydajnością, izolacją, czasem rozruchu, przestrzenią dyskową, zależnością od systemu operacyjnego i dojrzałością. Większość z tych czynników jest związana z wydajnością, ale w grę wchodzi również bezpieczeństwo i dojrzałość rozwiązań. Zgodnie ze względnym znaczeniem tych parametrów i zależnie od zamierzonego zastosowania każda z alternatyw może być poprawnym wyborem.

Z jednej strony tradycyjne maszyny wirtualne pochłaniają dużo zasobów, a ich uruchamianie i wdrażanie przebiega wolno. Zapewniają jednak silną izolację, której doświadczeno w produkcji od wielu lat. Z drugiej strony, kontenery są lekkie, bardzo szybkie we wdrożeniu i rozruchu oraz, zmniejszając izolację, nakładając mniejszy narzut. Unikernie starają się osiągnąć kompromis między maszynami wirtualnymi, a kontenerami, zapewniając szybkie i lekkie środowisko wykonawcze działające na nadzorcy. Nadal jest to jednak rozwiązanie eksperymentalne.

Dalsza część pracy skupia się wyłącznie na kontenerach jako obecnie najpopularniejszym rozwiąaniu wirtualizacji systemów operacyjnych.

3. Ekosystem Docker'a

Pojęcie Docker jest powiązane z kilkoma znaczeniami. Po pierwsze jest to specyfikacja obrazów kontenera i środowiska wykonawczego, w tym plików *Dockerfile*, umożliwiających powtarzalny proces budowania (rys. 3.1a). Ponadto, jest to także oprogramowanie, które implementuje tę specyfikację (demon Docker, o nazwie Docker Engine, rys. 3.1b), centralny rejestr, w którym programiści mogą przechowywać i udostępniać swoje obrazy (Docker Hub, rys. 3.1c) oraz inne nieoficjalne rejstry (rys. 3.1d), wraz ze znakiem towarowym (Docker Inc.). Proces rozwoju oprogramowania zakłada przetrzymywanie kodu w zewnętrznym repozytorium (rys. 3.1e) oraz wykorzystanie zewnętrznego serwisu rurociągu CI/CD (rys. 3.1g). Definicja rurociągu łą-

czy zdefiniowany kod z zewnętrznymi zależnościami (rys. 3.1f) i buduje obraz Dockera, który jest publikowany we wspomnianych już rejestrach. Do zarządzania cyklem życia infrastruktury operacyjnej można wykorzystać orkiestratora (rys. 3.1h).

Projekt Docker został napisany w języku Go i wydany po raz pierwszy w marcu 2013 roku. Od tego czasu doznał gwałtownego wzrostu popularności i został powszechnie przyjęty przez przemysł IT jako domyślne rozwiązanie kontenerowe.

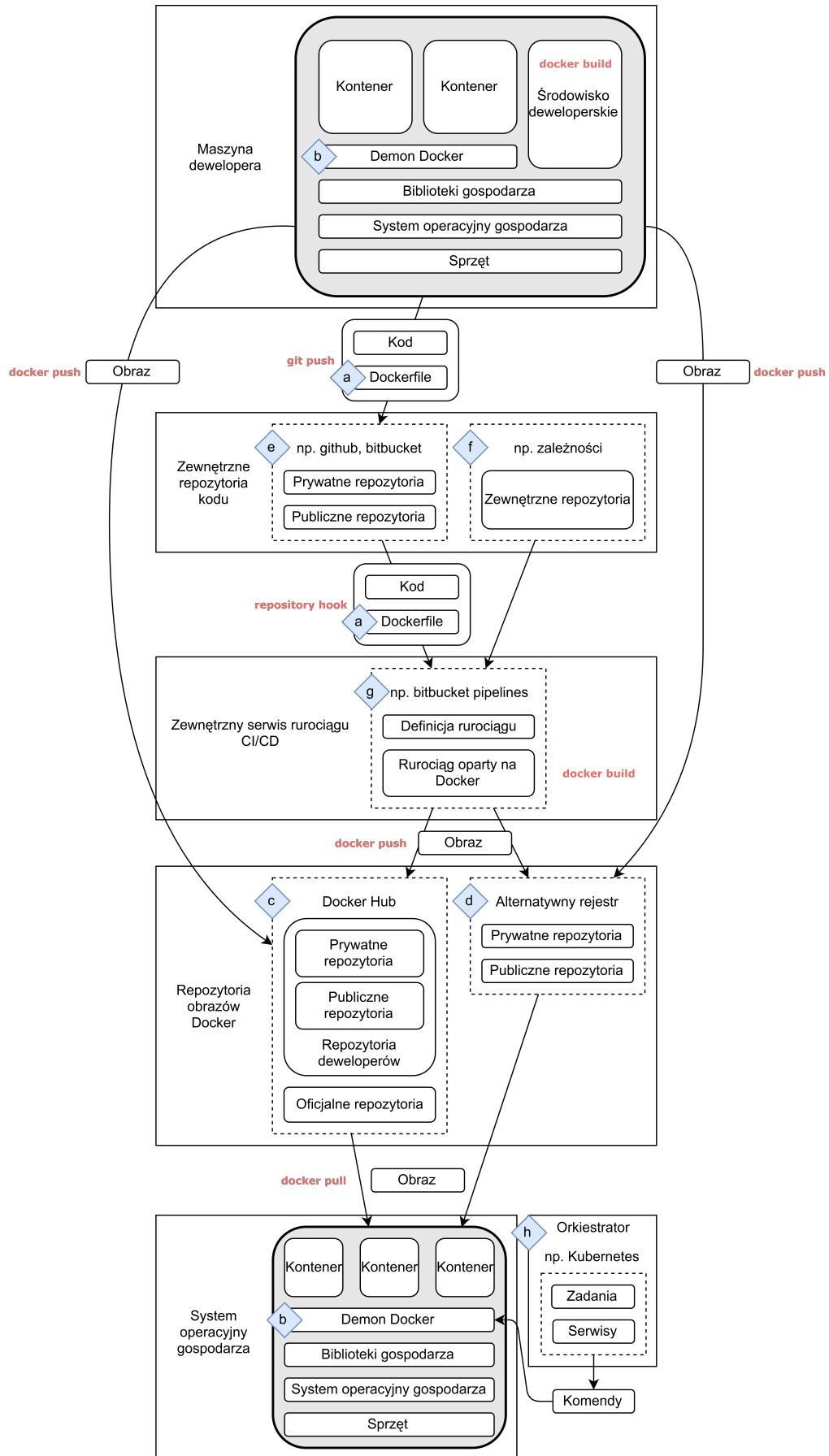
3.1. Specyfikacje Dockera

Zakres specyfikacji obejmuje obrazy kontenera i środowisko wykonawcze. Obrazy Dockera składają się z zestawu warstw wraz z metadanymi w formacie JSON. Są one przechowywane w `/var/lib/docker/<driver>/`, gdzie `<driver>` oznacza używany sterownik pamięci (np. AUFS, BTRFS, VFS, Device Mapper, OverlayFS). Każda warstwa zawiera modyfikacje wprowadzone w systemie plików w stosunku do poprzedniej warstwy, zaczynając od obrazu podstawowego (zazwyczaj lekkiej dystrybucji Linuxa). W ten sposób obrazy są zorganizowane w drzewa, a każdy obraz ma rodzica, z wyjątkiem obrazów podstawowych, które są korzeniami drzew (rys. 3.2). Ta struktura umożliwia wysyłanie w obrazie tylko modyfikacji specyficznie związanych z tym obrazem (dane właściwe aplikacji). Dlatego jeśli wiele obrazów na systemie operacyjnym gospodarza dziedziczy po tym samym obrazie podstawowym lub ma te same zależności, zostaną one pobrane tylko raz z rejestru. Ponadto, jeśli pozwala na to lokalny sterownik pamięci (z systemem plików typu *union*, tj. systemem plików tylko do odczytu i dodatkową warstwą zapisu), będzie on przechowywany tylko raz na dysku, prowadząc do znacznych oszczędności zasobów dyskowych. Szczegółowa specyfikacja obrazów Dockera i kontenerów znajduje się w [MOBY 2018].

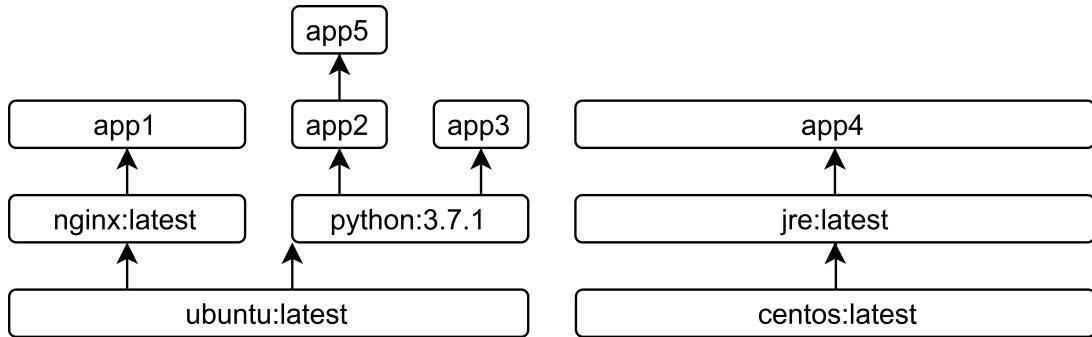
Metadane obrazów zawierają informacje o samym obrazie (np. identyfikator, suma kontrolna, tagi, rejestr, autor), o jego rodzicu (identyfikator) wraz z opcjonalnymi parametrami środowiska wykonawczego, które posiadają wartości domyślne (np. przekierowanie portów, konfiguracja *cgroups*). Parametry te można nadpisać w czasie uruchamiania przekazując ich wartości do komendy `docker run`.

Budowanie obrazów można wykonać na dwa sposoby. Możliwe jest uruchomienie kontenera z istniejącego obrazu (`docker run`), wykonanie modyfikacji i instalacji wewnątrz kontenera, zatrzymanie kontenera, a następnie zapisanie stanu kontenera jako nowego obrazu (`docker commit`). Proces ten jest zbliżony do klasycznej instalacji maszyny wirtualnej, ale musi być wykonywany przy każdej przebudowie obrazu (np. w celu aktualizacji). Z racji, iż obraz podstawowy jest znormalizowany, sekwencja poleceń jest dokładnie taka sama.

W celu automatyzacji powyższego procesu, pliki *Dockerfile* pozwalają określić obraz podstawowy i sekwencję poleceń, które należy wykonać w celu zbudowania obrazu, wraz z innymi



Rysunek 3.1: Przegląd ekosystemu Docker. Strzałki pokazują ścieżkę kodu



Rysunek 3.2: Przykład drzew dziedziczenia obrazów

opcjami specyficznymi dla obrazu (np. odsłonięte porty, komenda startowa). Obraz jest następnie budowany za pomocą komendy *docker build*, w wyniku czego powstaje kolejny ustandaryzowany i otagowany obraz, który można uruchomić lub wykorzystać jako obraz podstawowy dla kolejnego obrazu. Szczegółowa specyfikacja pliku *Dockerfile* jest dostępna w [DOCK 2019g].

3.2. Elementy wewnętrzne Dockera

Kontenery Dockera bazują na tworzeniu opakowanego i kontrolowanego środowiska wewnątrz systemu operacyjnego gospodarza, w którym dowolny kod może (docelowo) być bezpiecznie uruchomiony. Izolacja ta zostaje osiągnięta głównie dzięki dwóm cechom jądra: przestrzeniom nazw (*namespaces*) i grupom kontrolnym (*cgroups*). Warto zaznaczyć, że te funkcjonalności nie istniały w jądrze Linuxa od zawsze i pojawiły się w wersji 2.6.24 [CORB 2017]. Obecnie w jądrze znajduje się 7 różnych przestrzeni nazw, z których każda dotyczy określonego aspektu systemu operacyjnego [LINU 2019]:

- Cgroup – zapewnia wirtualizację grup kontrolnych dla procesu. Każda przestrzeń grupy kontrolnej posiada własne *root directory*, które reprezentuje punkty bazowe jej zasad kontrolnych
- IPC (inter-process communication) – zapewnia wyizolowane kolejki komunikatów POSIX, komunikację System V, pamięć współdzieloną, itd.
- Network – zapewnia zasoby sieciowe. Każda z przestrzeni nazw zawiera cały stos sieciowy: interfejsy, tablice routingu, *iptables*, gniazda sieciowe
- Mount – zapewnia osobne punkty montowania dysków dla każdej z przestrzeni nazw
- PID (process identifier) – zapewnia odrębne drzewa PID, które są względne dla każdej z przestrzeni nazw. Każdy PID w przestrzeni nazw jest mapowany na unikalne PID w przestrzeni globalnej co pozwala na posiadanie tego samego lokalnego PID przez wiele procesów w różnych przestrzeniach nazw.

- User – zapewnia odrębny widok na użytkowników i grupy: UID (user id), GID (group id), uprawnienia plików
- UTS (Unix timesharing system) – zapewnia osobną nazwę hosta i domeny NIS dla każdej z przestrzeni nazw

Każda z przestrzeni nazw ma własne obiekty wewnętrzne jądra związane z jej typem i zapewnia przetwarzanie lokalnej instancji niektórych ścieżek w systemach plików */proc* i */sys*. Na przykład przestrzenie nazw Network mają własny katalog */proc/net*. Dokładną listę ścieżek izolowanych dla przestrzeni nazw podano w [LINU 2019].

Nowe przestrzenie nazw mogą być tworzone przez wywołania systemowe *clone()* i *unshare()*, a procesy mogą zmieniać swoje obecne przestrzenie nazw za pomocą *setns()*. Procesy dziedziczą przestrzenie nazw po procesie rodzica. Każdy kontener jest tworzony w ramach własnych przestrzeni nazw. Oznacza to, że po uruchomieniu głównego procesu (punktu wejścia kontenera) wszystkie procesy potomne kontenera są ograniczone do przestrzeni nazw kontenera.

Grupy kontrolne (*cgroups*) to mechanizm jądra, który ogranicza wykorzystanie zasobów przez proces lub grupę procesów. Zapobiegają one przed zawłaszczeniem przez proces wszystkich dostępnych zasobów i wygładzeniu innych procesów, a tym samym kontenerów. Kontrolowane zasoby obejmują czas użytkowania procesora, pamięć RAM, przepustowość sieci i operacje wejścia/wyjścia.

3.3. Demon Dockera

Oprogramowanie Docker (rys. 3.1b) działa jako demon na komputerze gospodarza. Demon może uruchamiać kontenery, kontrolować ich poziom izolacji (grupy kontrolne, przestrzenie nazw, uprawnienia *capabilities* i profile SELinux/Apparmor), monitorować je w celu wyzwalania poleceń (np. restart) i otwierać powłoki wewnętrz kontenera w celach administracyjnych. Może zmieniać reguły *iptables* na systemie operacyjnym gospodarza i tworzyć interfejsy sieciowe. Jest również odpowiedzialny za zarządzanie obrazami kontenerów: pobieranie i publikowanie obrazów w zdalnym rejestrze (np. Docker Hub), budowanie obrazów z plików Dockerfile, podpisywanie ich, itp. Sam demon działa jako użytkownik *root* (z pełnymi uprawnieniami) w systemie gospodarza i jest zdalnie sterowany przez gniazdo unixowe. Właściciel tego gniazda lub członek grupy, do której gniazdo jest przypisane, może zarządzać kontenerami na systemie gospodarza za pomocą polecenia *docker*. Alternatywnie, demon może nasłuchiwać na klasycznym gnieździe TCP, umożliwiając zdalne administrowanie kontenerami bez potrzeby dodatkowej powłoki na hoście.

3.4. Docker Hub

Docker Hub (rys. 3.1c) to rejestr online, który umożliwia programistom przesyłanie i udostępnianie ich obrazów Docker w celu pobrania przez innych użytkowników. Programiści mogą założyć bezpłatne konto, na którym wszystkie rejestyry są publiczne, lub konto płatne, umożliwiające tworzenie prywatnych rejestrów. Istnieją również oficjalne rejestyry, dostarczone bezpośrednio przez Docker Inc.: "wyselekcyjowany zestaw repozytoriów Docker rejestrów w Docker Hub" [DOCK 2019j]. Oficjalne rejestyry dostarczają najczęściej używane obrazy podstawowe służące do budowy kontenerów.

Docker Hub pozwala również programistom Dockera sprzedawać swoje obrazy wśród użytkowników Dockera. Sklep zawiera bezpłatne i open-sourcowe obrazy, a także oprogramowanie sprzedawane bezpośrednio przez wydawców. Każdy użytkownik może zostać wydawcą co zapewnia lepszą widoczność, możliwość uzyskania certyfikowanych znaków jakości Docker oraz skorzystania ze wsparcia licencyjnego Docker Hub (w celu ograniczenia dostępu do ich oprogramowania w zależności od rodzaju użytkowników). Co ważne, Docker Hub zapewnia wydawcom kanał komunikacji z klientami, pozwalając na powiadamianie użytkowników w przypadku naruszenia zabezpieczeń lub aktualizacji obrazu [DOCK 2019f].

3.5. Systemy operacyjne dedykowane Dockerowi

Oprócz pakietu Docker dla głównych dystrybucji systemu Linux opracowano szereg dedykowanych dystrybucji przenaczonych do uruchamiania Dockera lub innych rozwiązań kontenerowych.

- CoreOS – dystrybucja dedykowana kontenerom. Może obsługiwać zarówno Dockera, jak i Rocketa, dla którego został zaprojektowany. Rocket jest forkiem Dockera, który obsługuje kontenery bez użycia wyspecjalizowanego demona. Tym samym, w przeciwieństwie do monolitycznego projektu Dockera, interakcją z ekosystemem i komplikacjami obrazów zarządzają inne narzędzia w CoreOS. System operacyjny integruje się docelowo z Kubernetes aby koordynować klastry kontenerów na wielu hostach [PURR 2015].
- RancherOS – system operacyjny w całości oparty na Dockerze, przeznaczony do uruchamiania kontenerów Docker. Proces inicjalizujący również jest demonem Dockera, a usługi systemowe działają w kontenerach. Jedną z tych usług jest kolejny demon Dockera, który uruchamia kontenery na poziomie użytkownika. Wszystkie zainstalowane aplikacje w systemie działają w kontenerach Dockera, dzięki czemu obraz systemu jest bardzo lekki. Dystrybucja zapewnia dodatkowo programy przydatne w produkcyjnych wdrożeniach kontenerów [ROSL 2018].

- Project Atomic – dystrybucja, która używa menadżerów pakietów z trzech większych dystrybucji (CentOS, Fedora, RHEL) i tym samym pozwala na wybór swojej dystrybucji bazowej. Poza wbudowanym wsparciem dla Dockera i Kubernetes implementuje funkcjonalność tranzakcyjnego systemu operacyjnego, tzn. pozwala na wycofanie aktualizacji do stanów poprzednich. Ponadto, domyślnie definiuje rygorystyczne reguły SELinux w celu zapewnienia zwiększonego bezpieczeństwa urachamianych kontenerów [ROSL 2018].

3.6. Docker dla systemów operacyjnych nieopartych na Linuxie

Oprogramowanie Docker można uruchomić również na dwóch systemach operacyjnych nieopartych na Linuxie: Windows i MacOS. W obydwu przypadkach demon Dockera uruchamiany jest wewnątrz maszyny wirtualnej Linuxa uruchomionej w systemie gospodarza. Powoduje to gorszą wydajność kontenerów co wynika z dodatkowego narżenia warstwy wirtualizacji. Powyzsze systemy operacyjne od kilku lat próbują wprowadzać specyficzne mechanizmy mające na celu przybliżenie wydajności kontenerów Dockera do tej osiąganej na systemie Linux, a także ukrycie wszelkich zmian wynikających z odmiennych architektur systemów operacyjnych.

Maszyna wirtualna początkowo wykorzystana do wirtualizacji systemu Linux to Docker Machine (poprzednia nazwa Boot2Docker). Narzędzie to pozwala na przygotowanie środowiska i silnika Docker Engine na nielinuowych systemach operacyjnych. Docker Machine był jedynym sposobem na uruchomienie Dockera na systemach Windows i MacOS aż do wersji 1.12. W chwili obecnej dostępne są specjalne wydania natywnych aplikacji Dockera: Docker Desktop for Windows i Docker Desktop for Mac. Natywność tych aplikacji polega na optymalizacji maszyny wirtualnej i silnika Docker pod określony system operacyjny [DOCK 2019c].

3.6.1 Windows

Docker Desktop for Windows może być zainstalowany tylko w systemie Windows w wersji 10 Pro i w najnowszych wersjach serwerowych (2016+). W celu wirtualizacji wykorzystuje technologię Hyper-V co pozwala na osiąganie lepszej wydajności niż uruchomienie maszyny wirtualnej w programie VirtualBox lub VMWare. Pozwala również na dostosowanie klasycznych ustawień maszyn wirtualnych takich jak przydział rdzeni procesora, pamięci, rozmiaru dysku, zapory sieciowej czy współdzielenia katalogów. Ponadto, domyślnie gromadzi i przesyła statystyki użytkowania [DOCK 2019i].

Docker w wersji na systemy Windows pozwala na tworzenie obrazów Dockera, które działają tylko i wyłącznie na jądrze systemu Windows. Obrazy podstawowe tego typu można również znaleźć w Docker Hub. O ile obrazy bazujące na Linuxie można uruchomić w Dockerze na wszystkich systemach, o tyle obrazy bazujące na systemie Windows mogą być tylko na nim

uruchomione. Tworzy to pewnego rodzaju niezgodność z założeniami wirtualizacji systemów operacyjnych, zgodnie z którymi rozwiązanie wirtualizacyjne powinno być ogólne.

3.6.2 MacOS

Docker Desktop for Mac wykorzystuje specjalnie przygotowanego nadzorę wirtualizacji *hyperkit* oraz współdzielony system plików *osxfs*. Od samego początku oprogramowanie posiadało problemy wydajnościowe związane z wspomnianym systemem plików. Wersja 17.04 Dockera wprowadziła dodatkowe opcje montowania dla systemu plików *osfxs*, które zwiększyły wydajność operacji dyskowych nawet 3,5-krotnie. Nadal jednak Docker w wersji dla systemu MacOS pozostaje najwolniejszym sposobem uruchomienia infrastruktury Docker [DOCK 2019h][SCHM 2017].

4. Architektura zabezpieczeń kontenerów Dockera

4.1. Izolacja

Kontenery Dockera w kwestii zabezpieczeń opierają się wyłącznie na funkcjach jądra systemu Linux, w tym na przestrzeniach nazw, grupach kontrolnych, utwardzaniu jądra i uprawnieniach (*capabilities*). Izolacja przestrzeni nazw i redukcja uprawnień są domyślnie włączone. Jednakże, ograniczenia grup kontrolnych muszą być skonfigurowane dla poszczególnych kontenerów poprzez opcje *--cgroup-parent* podczas uruchamiania kontenera. Domyślna konfiguracja izolacji jest stosunkowo ścisła. Jedyną wadą jest to, że wszystkie kontenery współużytkują ten sam most sieciowy, umożliwiając ataki typu *ARP poisoning* między kontenerami na tym samym systemie operacyjnym gospodarza.

Zabezpieczenia kontenera można obniżyć za pomocą opcji, podanych przy uruchomieniu kontenera, dając tym samym kontenerowi rozszerzony dostęp uprawnień do niektórych części gospodarza. Poniższe opcje pozwalają na zwiększenie uprawnień kontenera. Większość z nich polega na wyłączeniu izolacji wynikającej z systemu przestrzeni nazw:

- *--cap-add=<CAP>* – dodanie jednego z uprawnień
- *--privileged* – dostęp do wszystkich urządzeń gospodarza, jak i konfiguracje AppArmor i SELinux dające te same uprawnienia jakie mają procesy gospodarza
- *--cgroup-parent=<parent>* – dołączenie kontenera do określonej grupy kontrolnej
- *--ipc=*

- shareable – prywatna przestrzeń nazw IPC, z możliwością współdzielenia z innymi kontenerami
- container: <name or ID> – dołączenie do przestrzeni nazw IPC innego kontenera z opcją *shareable*
- host – użycie przestrzeni nazw IPC gospodarza
- --network=host – użycie stosu sieciowego gospodarza
- --device=<host-device-path>:<container-device-path> – dostęp i możliwość zamontowania określonego urządzenia
- --pid=
 - container: <name or ID> – dołączenie do przestrzeni nazw PID innego kontenera
 - host – użycie przestrzeni nazw PID gospodarza
- --uts=host – użycie przestrzeni nazw UTS gospodarza

Opcje te umożliwiają kontenerom interakcje z innymi kontenerami lub systemem gospodarza w zamian za wprowadzenie ewentualnych luk w zabezpieczeniach. Na przykład, opcja *--net=host* powoduje, że Docker nie umieszcza kontenera w osobnej przestrzeni nazw Network, a zatem daje kontenerowi pełny dostęp do stosu sieciowego hosta (umożliwiając rekonfigurację, *network sniffing*, itp.).

Zabezpieczenia można dodatkowo konfigurować globalnie za pomocą opcji przekazywanych do demona Docker. Obejmuje to opcje obniżające bezpieczeństwo, takie jak opcja *--insecure-register*, wyłączająca sprawdzanie certyfikatu TLS dla danego rejestru. Z drugiej strony, dostępne są opcje zwiększące bezpieczeństwo, takie jak opcja *--icc=false*, która zabrania komunikacji sieciowej pomiędzy kontenerami i zmniejsza szanse na opisane wcześniej ataki typu *ARP poisoning*. Uniemożliwa jednak ona prawidłowe działanie aplikacji wielokontenerowych, dlatego też jest rzadko używana [DOCK 2019d].

4.2. Utwardzanie jądra systemu operacyjnego gospodarza

Utwardzanie jądra za pomocą modułów bezpieczeństwa Linux to sposób na egzekwowanie ograniczeń związanych z bezpieczeństwem. Takie dodatkowe zabezpieczenia mogą być przydatne w momencie przeniknięcia złośliwego oprogramowania do kontenera i eskalacji (ucieczki) do systemu operacyjnego gospodarza. Obecnie obsługiwane są AppArmor, SELinux (tylko i wyłącznie dla dystrybucji RedHat), Seccomp i GRSEC z dostępnymi profilami domyślnymi. Profile te są bardzo ogólne, dlatego też domyślne utwardzanie chroni gospodarza przed kontenerami, ale

nie kontenerów przed innymi kontenerami. Rozwiążanie tego problemu leży po stronie programistów, którzy muszą sami określić profile/polityki w zależności od indywidualnych zapotrzebowień systemu [MOBY 2019][MILL 2012].

W celu użycia dodatkowej polityki bezpieczeństwa w ramach kontenera należy w trakcie jego uruchomienia przekazać odpowiedni argument wraz z opcją `--security-opt`. Taka możliwość istnieje w Dockerze od wersji 1.3 (październik 2013) co pokazuje, że wsparcie dla dodatkowego zabezpieczenia kontenerów istniało prawie od początku projektu Docker. W późniejszych latach zabrakło wewnętrznego rozwoju tego typu rozwiązań i obecny stan niewiele różni się od tego sprzed kilku lat.

4.2.1 SELinux

SELinux został pierwotnie opracowany przez Narodową Agencję Bezpieczeństwa Stanów Zjednoczonych i implementuje system obowiązkowej kontroli dostępu (ang. MAC, Mandatory Access Control). Decyzje dotyczące kontroli dostępu są podejmowane na podstawie kontekstu bezpieczeństwa przypisanego do zasobów, łącząc kontrolę dostępu opartą na rolach (ang. Role Base Access Control), wymuszanie typów kontroli (ang. Type Enforcement) oraz zabezpieczenia wielopoziomowe (ang. Multi Lever Security). Reguły określają domenę, z którą program jest powiązany, i definiują, w jaki sposób procesy w poszczególnych domenach mogą uzyskiwać dostęp do zasobów oznaczonych określonymi typami. Reguły SELinuxa mogą być bardzo złożone i są zdefiniowane w postaci kontekstów bezpieczeństwa, które są stosowane jako etykiety zasobów [SCHR 2011].

Przestrzenie nazw nie wpływały na kontrolę dostępu SELinuxa, co oznacza, że istnieje tylko jeden zbiór zasad dla wszystkich kontenerów Dockera w systemie. Opcja `--security-opt` pozwala na określenie etykiety użytkownika oraz roli i typu etykiety:

- `--security-opt=label:user:<USER>`
- `--security-opt=label:role:<ROLE>`
- `--security-opt=label:type:<TYPE>`

Dystrybucja Linuxa o nazwie RedHat jako jedna z niewielu dostarcza domyślną politykę SELinuxa dla Dockera. Bazuje ona na polityce przygotowanej dla maszyn wirtualnych *libvirt*. Specjalna etykieta *svirt_sandbox_file_t* pozwala na dodanie kontenerom uprawnień do danego zasobu. Polityka definiuje wiele reguł zwiększających bezpieczeństwo systemu gospodarza i kontenerów poprzez ograniczenie kontenerom uprawnień takich jak [RED 2016]:

- zapis do katalogów innych niż `/etc` i `/usr`
- dostęp do portów TCP
- zapis do plików typu `corefile`
- odczyt i zapis do powłok systemowych
- nasłuchiwanie na wywołania systemowe procesów
- dynamiczne ładowanie modułów jądra
- odczyt systemowego generatora liczb losowych
- korzystanie z pamięci współdzielonej

Warto zaznaczyć, że tworzenie i modyfikacja polityk SELinuxa wymaga wiedzy dziedzionatej z zakresu bezpieczeństwa systemów operacyjnych i dla większości programistów pracujących z Dockerem będzie wyzwaniem. Twórcy Dockera nie ułatwiają w żaden sposób tworzenia takich polityk chociażby poprzez dostarczanie domyślnych polityk dla popularniejszych systemów operacyjnych. Muszą tym zajmować się sami twórcy dystrybucji (np. Red Hat, Inc.).

4.2.2 AppArmor

AppArmor również implementuje system obowiązkowej kontroli dostępu jednakże używa prostszego modelu niż SELinux. AppArmor pozwala na zdefiniowanie profilu dla każdego z ograniczonych programów z listą uprawnień do zasobów. Proste abstrakcje takie jak `dbus`, `kde` lub `nameservice` pozwalają na grupowanie niskopoziomowych cech programu w celu zdefiniowania profilu. Istnieje wiele programów z interfejsem graficznym ułatwiających generowanie profili. Profile AppArmor mogą być długie i szczegółowe odzwierciedlając złożoność aplikacji i różnorodność dystrybucji, na których ma być uruchamiony program. Narzędzie to jest przedstawiane przez twórców, Canonical Ltd., jako prostsza alternatywa w stosunku do SELinuxa [SCHR 2011].

Profile AppArmor oddziałują na poszczególne kontenery, a nie demona Dockera co pozwala na definiowane osobnych profili dla każdego kontenera. Opcja `--security-opt=<PROFILE>` pozwala na określenie nazwy profilu, który ma zostać użyty w momencie uruchomienia kontenera. Domyślny profil dostarczany przez Dockera definiuje następujące reguły [MOBY 2019]:

- pozwala na zabicie kontenera przez demona Dockera
- pozwala na wysyłanie sygnałów pomiędzy kontenerami

- blokuje zapis do większości plików i podkatalogów w katalogu `/proc`, z wyróżniającym się wyjątkiem `/proc/sys/kernel`
- blokuje montowanie woluminów danych
- blokuje nasłuchiwanie na wywołania systemowe procesów

Istnieje również narzędzie `bane`, które pozwala na definiowanie profili przygotowanych bezpośrednio pod kontenery Dockera w prostszym formacie plików TOML. Narzędzie automatycznie transformuje zdefiniowaną konfigurację w profil AppArmor, zapisuje go na dysku oraz wykonuje obowiązkowe parsowanie przy użyciu `apparmor_parser`. Znacznie ułatwia to pracę z tworzeniem profili dla kontenerów aplikacyjnych [BANE 2019].

4.3. Wykorzystanie mechanizmów kryptograficznych

Obrazy Dockera pobrane ze zdalnego rejestru są weryfikowane za pomocą funkcji skrótu, zaś połączenie z rejestrem jest nawiązywane poprzez TLS (chyba, że wymuszono połączenie niezabezpieczone). Ponadto, począwszy od wersji 1.8 wydanej w sierpniu 2015r., architektura Docker Content Trust pozwala deweloperom podpisywać swoje obrazy przed opublikowaniem ich do rejestru.

Content Trust opiera się na The Update Framework. Jest to framework zaprojektowany w celu usunięcia wad menedżera pakietów. Może odratować system po przejęciu klucza (`key compromise`) i złagodzić ataki typu *replay attack* poprzez osadzenie wygasających znaczników czasowych w podpisanych obrazach. Jego wadą jest jednak złożone zarządzanie kluczami. W rzeczywistości implementuje infrastrukturę klucza publicznego, w której każdy programista jest właścicielem klucza głównego ("offline key"), używanego do podpisywania ("signing keys") obrazów Docker. Klucze do podpisywania są współużytkowane przez każdy podmiot, który chce opublikować obraz. Należy w to również włączyć automatyzowanie podpisów w łańcuchach CI/CD co powoduje dostęp do kluczy przez podmioty zewnętrzne. Problemem jest także dystrybucja (licznych) kluczy głównych [SAMU 2010][CAPP 2008].

Demon Dockera jest zdalnie sterowany poprzez gniazdo. Domyślnym gniazdem używanym do sterowania demonem jest gniazdo unixowe, zlokalizowane w `/var/run/docker.sock` i należące do `root:docker`. Dostęp do tego gniazda umożliwia pobieranie i uruchamianie dowolnego kontenera w trybie uprzywilejowanym, zapewniając w ten sposób dostęp użytkownika `root` do systemu gospodarza. W przypadku gniazda unixowego członek grupy `docker` może uzyskać uprawnienia użytkownika `root`. W przypadku gniazda TCP dowolne połączenie z tym gniazdem pozwala na korzystanie z uprawnień użytkownika `root`. Dlatego też, połączenie musi być zabezpieczone przy pomocy TLS (`--tlsverify`). Umożliwia to zarówno szyfrowanie, jak i uwierzytelnianie dwóch stron

połączenia, jednak dodaje problem zarządzania dodatkowymi certyfikatami. Warto zaznaczyć, że opcja `--tlsverify` jest domyślnie wyłączona.

5. Powszechnie przypadki użycia Dockera

Większość dyskusji na temat bezpieczeństwa kontenerów porównuje je do maszyn wirtualnych, zakładając zatem, że obie technologie są równoważne pod względem projektowym. Chociaż równoważność z maszynami wirtualnymi jest celem niektórych technologii kontenerowych (np. OpenVZ), najnowsze "lekkie" rozwiązania kontenerowe, takie jak Docker, zostały zaprojektowane aby osiągnąć zupełnie inne cele niż te uzyskiwane przez maszyny wirtualne [PETA 2014]. Dlatego też, w ramach pracy wyróżnione zostaną trzy przypadki użycia jako podłoż do analizy podatności Dockera.

5.1. Zalecany przypadek użycia

Twórcy Dockera zalecają podejście oparte na mikrousługach, co oznacza, że kontener musi obsługiwać pojedynczą usługę działającą w jednym procesie lub demona tworzącego procesy-dzieci. Dlatego też, kontener Docker nie jest traktowany jak maszyna wirtualna: nie posiada menedżera pakietów, procesu inicjującego ani demona SSH do zarządzania nim. Wszystkie zadania administracyjne (zatrzymanie kontenera, restart, tworzenie kopii zapasowych, aktualizacje, itd.) muszą być wykonywane za pośrednictwem gospodarza. Oznacza to, że administrator kontenera posiada dostęp na poziomie użytkownika *root* do systemu gospodarza. Docker został zaprojektowany do izolowania aplikacji, które w innym przypadku działałyby bezpośrednio w systemie gospodarza, więc zakłada się, że wspomniane uprawnienia zostały przyznane. Izolacja procesów (przestrzeń nazw) i zarządzanie zasobami (grupy kontrolne) sprawiają, że wdrażanie aplikacji Docker zapewnia większe bezpieczeństwo w porównaniu z nieużywaniem żadnej z technologii kontenerowych.

Główną zaletą Dockera jest łatwość wdrażania aplikacji. Został zaprojektowany w celu całkowitego oddzielenia płaszczyzny kodu od płaszczyzny danych. Obrazy Dockera można budować w dowolnym miejscu za pomocą ogólnego pliku konfiguracyjnego (*Dockerfile*), który określa kroki budowania obrazu z obrazu podstawowego. Ten ogólny sposób budowania obrazów sprawia, że proces generowania obrazów i powstałe obrazy są prawie niezależne od systemu gospodarza. Jądro systemu jest (zgodnie z definicją) jedynym elementem, który może wpływać na proces generowania – żadne oprogramowanie zainstalowane na systemie gospodarza nie powinno wpływać na tworzony obraz. Zgodnie z oficjalnymi zaleceniami można wymienić sprawdzone w produkcji przypadki użycia Dockera [SULE 2016]:

- uproszczenie konfiguracji

- zarządzanie rurociągiem CI/CD
- produktywność deweloperów
- izolacja aplikacji
- konsolidacja aplikacji
- możliwości debugowania
- architektura nastawiona na wielu najemców (multi-tenancy)
- szybkie wdrożenia

5.2. Rozpowszechniony przypadek użycia

Zgodnie z raportem Forrester Consulting, jednym z głównych powodów przyjęcia kontenerów jest zwiększenie wydajności programistów, a nie faworyzowanie architektury mikrousług co byłoby zgodnie z zalecanyim przypadkiem użycia [FORR 2017]. W rzeczywistości dwa najbardziej popularne obrazy w Docker Hub to Ubuntu i CentOS czyli dwa obrazy kontenerów zorientowanych na bycie maszyną wirtualną [DOCK 2019b]. Niektórzy administratorzy lub programiści używają Dockera do wdrażania kompletnych środowisk wirtualnych i ich regularnej aktualizacji, wykorzystując kontenery tak jakby były maszynami wirtualnymi. Ogranicza to zadania administracyjne systemu do absolutnego minimum (np. *docker pull*) i tym samym jest wygodne dla użytkownika. Wprowadza jednak również kilka implikacji dla bezpieczeństwa systemu.

Po pierwsze, osadzenie większej ilości oprogramowania niż rozmiar dla którego zaprojektowano kontenery, zwiększa powierzchnię ataku obrazów kontenera. Nadmiarowe pakiety i biblioteki mogą prowadzić do luk w zabezpieczeniach, których w przeciwnym wypadku można było uniknąć. Co więcej, obraz większych rozmiarów sprawia, że zarządzanie kontenerami jest bardziej złożone i prowadzi do marnowania zasobów (np. wymagana większa przepustowość sieci i potrzebne miejsce do ich przechowywania, więcej procesów w kontenerach). Kiedy kontenery zawierają wystarczająco dużo oprogramowania, aby uruchomić pełny system operacyjny (demon do tworzenia logów, demon SSH, a czasem nawet proces inicjujący), kuszące jest wykonywanie zadań administracyjnych z poziomu samego kontenera. Jest to całkowicie sprzeczne z założonym projektem Dockera. Co więcej, niektóre z zadań administracyjnych wymagają dostępu użytkownika *root* do kontenera. Jeszcze inne czynności administracyjne (np. montowanie dysków w kontenerze) mogą wymagać dodatkowych uprawnień (*capabilities*), które domyślnie są odebrane przez Dockera. Wszystkie powyższe działania mają tendencję do zwiększania powierzchni ataku. Aktywująą większą ilość kanałów komunikacji pomiędzy gospodarzem, a kontenerami oraz

między samymi kontenerami, zwiększąc ryzyko ataków, takich jak eskalacja uprawnień [COLE 2016].

Dzięki przyspieszeniu cykli tworzenia oprogramowania, na które pozwala Docker, programiści nie są w stanie utrzymywać każdej wersji swojego produktu i wspierają najczęściej tylko tę najnowszą (tag *latest* w rejestrach Docker). W rezultacie stare obrazy są nadal dostępne do pobrania, mimo tego, że nie były aktualizowane przez setki dni i mogą prowadzić do luk w zaabezpieczeniach [SHU 2017]. Badanie [BANY 2015] wykazało, że ponad 30% obrazów w Docker Hub zawiera zagrożenia o wysokim poziomie CVE (Common Vulnerabilities and Exposures), a do 70% zawiera zagrożenia o wysokim lub średnim poziomie CVE.

Warto zaznaczyć, że niektórych obrazów nie da się nawet całkowicie odtworzyć. Chociaż plik *Dockerfile* jest publicznie dostępny w Docker Hub to może zawierać instrukcję (np. *ADD start.sh*), która kopiuje skrypt startowy z komputera twórcy obrazu do obrazu i uruchamia go, a skrypt jednocześnie nie pojawia się w pliku *Dockerfile*.

5.3. Przypadek użycia dostawców chmurowych

Największy dostawcy usług chmury publicznej umożliwiają uruchomienie kontenerów Docker przy pomocy wielu, różnorodnych funkcji. Amazon AWS (Amazon Web Services), Microsoft Azure i Google Cloud Platform nie są jedynymi dostawcami, jednak produją oni w świecie biznesu i sumarycznie odpowiadają za prawie 60% udziału w rynku. Jednak to Amazon wiedzie prym w dziedzinie usług chmurowych posiadając 47.8% udziału w rynku. Dokładne dane przedstawia tabela 5.1 [GART 2019].

Firma	Udział (%) w 2018r.
Amazon	47.8
Microsoft	15.5
Alibaba	7.7
Google	4.0
IBM	1.8
Inne	23.2

Tabela 5.1: Ogólnoświatowy udział w rynku usług chmurowych IaaS [GART 2019]

Funkcjonalności dostarczane przez powyższych dostawców w dziedzinie konteneryzacji różnią się od siebie, jednak posiadają pewne części bazowe, które są wspólne dla wszystkich z nich. Architektura i metoda działania zostanie opisana na bazie usług AWS, a następnie wykazane zostaną podobieństwa i różnice w stosunku do pozostałych dwóch dostawców.

Amazon AWS	Microsoft Azure	Google Cloud Platform
Elastic Cloud Computing	Azure Virtual Machine	Compute Engine
Elastic Container Registry	Azure Container Registry	Container Registry
Identity and Access Management	Azure Active Directory	Cloud Identity and Access Management
Elastic Container Service	Azure Container Instances	Google Kubernetes Engine

Tabela 5.2: Podobieństwa w usługach oferowanych przez dostawców chmurowych

Amazon oferuje trzy usługi związane z uruchamianiem kontenerów Docker. Pierwsza z nich to AWS EC2 czyli Elastic Cloud Computing. Usługa ta pozwala użytkownikom tworzyć maszyny wirtualne, na których następnie należy samemu zainstalować oprogramowanie Docker aby ostatecznie móc uruchomić kontener. Rozwiążanie to w niczym nie różni się od rozwiązań innych dostawców chmurowych czy nawet serwerowni oferujących hostowanie maszyn wirtualnych. Jednakże, z racji uruchomienia maszyny w ramach chmury AWS użytkownik otrzymuje dostęp do szeregu innych funkcjonalności oferowanych przez tego dostawcę. Tym samym, bez żadnej dodatkowej konfiguracji, maszyna wirtualna korzysta z wbudowanych zabezpieczeń, takich jak: zapora sieciowa, sieć prywatna, automatyczne szyfrowanie ruchu pomiędzy usługami, ochrona przed atakami DoS, szyfrowanie danych, zarządzanie użytkownikami mającymi dostęp do zasobów, a także ich kluczami oraz monitorowanie wszystkich zasobów. Z punktu widzenia bezpieczeństwa Dockera rozwiązanie nie różni się niczym od uruchomienia Dockera na normalnej maszynie – EC2 nie zapewnia profili AppArmor czy SELinux innych niż te domyślne, a także zezwala na wszystkie opcje, które udostępnia Docker (zwiększenie uprawnień, współdzielenie przestrzeni nazw).

Drugą usługą, raczej pomniejszą, jest AWS ECR czyli Elastic Container Registry. Amazon udostępnia własny rejestr obrazów Dockera działający na tej samej zasadzie co oficjalny Docker Hub. Ułatwia to pracę z kontenerami w ramach infrastruktury AWS, a także odciąża programistów od tworzenia własnej infrastruktury przechowywania obrazów. Integracja z AWS IAM (Identity and Access Management) zapewnia wysoce bezpieczną kontrolę nad przechowywanymi zasobami, która byłaby ciężka do osiągnięcia w przypadku tworzenia rejestru od zera.

ECS czyli Elastic Container Service skupia się tylko i wyłącznie na kontenerach Dockera oraz tworzeniu z nich klastrów. Usługa ta wykorzystuje wcześniej wspomniane maszyny wirtualne EC2, ukrywając jednocześnie przed użytkownikiem przedstawiony proces instalacji Dockera. Co więcej, ECS instaluje na każdej z maszyn agenta ECS czyli orkiestratora pozwalającego na tworzenie klastrów kontenerów Docker. Użytkownik z kolei, wykorzystując przygotowane przez siebie obrazy Dockera definiuje usługi (ang. service) składające się z zadań (ang. task), które będą uruchomione w ramach klastra. Dla każdej z usług można zdefiniować statyczną liczbę

maszyn wirtualnych i zadań lub przygotować politykę dynamicznego skalowania usługi, która rozdzieli zadania pomiędzy maszyny wirtualne. Taka polityka pozwoli również na odpowiednie reagowanie na obciążenie usługi i zgodnie z zapotrzebowaniem będzie uruchamiać lub wyłączać dodatkowe maszyny wirtualne. Komunikacja pomiędzy użytkownikiem, a demonem oraz demonem i maszynami wirtualnymi odbywa się przy pomocy specjalnego interfejsu programistycznego (opakowanego również w interfejs linii poleceń i interfejs graficzny), który korzysta tylko i wyłącznie z szyfrowanego połączenia. Za każdym razem gdy w ramach klastra uruchamiana jest nowa maszyna wirtualna generuje się nowa para kluczy szyfrujących.

Microsoft Azure nie oferuje żadnej usługi przeznaczonej bezpośrednio pod tworzenie klastrów kontenerów Dockera. Oficjalne dokumentacje proponują używanie usługi Azure AKS (Azure Kubernetes Service), która pozwala na zarządzanie środowiskami Kubernetes. Tym samym przypomina w dużym stopniu Amazon ECS jednak nie współpracuje tylko i wyłącznie z kontenerami Dockera.

Kubernetes jest stworzonym przez Google systemem służącym do automatycznego wdrożenia, zarządzania i skalowania aplikacji kontenerowych. Umożliwia automatyczne tworzenie klastra maszyn wirtualnych, na których instalowany i konfigurowany jest Docker. Kontenery są pogrupowane w pody (ang. pods): zbiory kontenerów o tej samej przestrzeni nazw Network (tym samym interfejsy sieciowe i adresy IP) oraz opcjonalnie grupy kontrolne, co umożliwia bezpośrednią komunikację między nimi. Silnie sprzężone kontenery (wiele mikrousług tworzących tę samą aplikację) zwykle działają w tym samym podzie. Pody są podstawową jednostką orkiestratora Kubernetes, podobnie jak maszyny wirtualne dla klasycznych infrastruktur chmurowych. Pody są automatycznie tworzone i umieszczane na maszynach wirtualnych w klastrze, zgodnie z konfiguracją冗余 i dostępności określoną przez kontrolery replikacji (ang. replication controllers). Te kontrolery same są częścią usług: podmiotów, które definiują globalne parametry aplikacji (mapowanie portów zewnętrznych, itp.). Wszystkie węzły w klastrze uruchamiają demona *kubelet*, który kontroluje lokalnego demona Docker, a centralny węzeł nadzoruje orkiestrację.

Część poradników Microsoft Azure poleca również używanie oprogramowania Docker Swarm do tworzenia klastrów kontenerów Docker. Docker Swarm jest "natywnym" rozwiązaniem oprogramowania Docker pozwalającym na zarządzanie klastrem. Oferuje on podobne funkcjonalności do Kubernetes jednak z racji swojego ścisłego powiązania z architekturą Dockera pozwala dodatkowo na łatwe odkrywanie usług (ang. service discovery) oraz aktualizację i wersjonowanie kontenerów wewnątrz klastra.

Instancja Docker Swarm reprezentuje kластер węzłów (ang. node), które mogą być maszy-

nami fizycznymi lub wirtualnymi. Istnieją dwa typy węzłów: menedżer i węzeł roboczy. Menedżerowie utrzymują stan klastra, a węzły robocze są rzeczywistymiinstancjami kontenerów. Gdy użytkownik wdraża usługę w klastrze, menedżer klastra ma obowiązek zaplanować ją jako jedno lub więcej zadań uruchamianych niezależnie od siebie w węzłach roboczych. Zadanie to atomowa jednostka planowania w klastrze, która jest uruchamiania w ramach kontenera. Docker Swarm zapewnia dwa typy wdrożeń dla usług: replikowane (ang. replicated) i globalne (ang. global). Korzystając z replikowanej usługi, użytkownik musi określić liczbę identycznych zadań, które chce uruchomić. Z kolei, usługa globalna uruchamia jedno zadanie dla każdego węzła roboczego. Za każdym razem, gdy użytkownik dodaje nowy węzeł do klastra, orkiestrator tworzy zadanie, a planista (ang. scheduler) przypisuje zadanie do nowego węzła.

Google Cloud Platform podobnie do Microsoft Azure nie implementuje własnego rozwiązania do tworzenia klastrów kontenerów Docker, zamiast czego proponuje używanie wyżej wspomnianych rozwiązań: Kubernetes i Docker Swarm.

Wymienieni wyżej dostawcy usług chmurowych posiadają podobną infrastrukturę (tabela 5.2), a ich podejścia do tworzenia klastrów i orkiestratorzy mają odpowiadające sobie główne elementy (tabela 5.3).

Amazon ECS	Kubernetes	Docker Swarm
Instancja EC2	Węzeł	Węzeł
Agent ECS	kubelet	Menedżer
Zadanie	Pod	Zadanie
Usługa	Kontroler replikacji	<i>brak</i>
Definicja zadania	Usługa	Usługa
Klaster	Klaster	Swarm

Tabela 5.3: Odpowiadające sobie terminy w Amazon ECS, Kubernetes i Docker Swarm

6. Analiza Dockera zorientowana na podatności

6.1. Model agresora

Biorąc pod uwagę opis ekosystemu Dockera i przypadki użycia, rozważane są dwie główne kategorie agresorów: bezpośredni i pośredni.

Bezpośredni agresor może przechwytywać, blokować, generować lub modyfikować komunikację sieciową i systemową. Jego celem są bezpośrednio maszyny produkcyjne. Lokalnie lub zdalnie może zaatakować:

- kontenery produkcyjne (np. z publicznie dostępnej usługi kontenerowej uzyskuje uprawnienia użytkownika *root* na powiązanym kontenerze lub z zaatakowanego kontenera wykonuje atak typu DoS na kontenerach znajdujących się w tym samym systemie operacyjnym gospodarza)
- produkcyjny system operacyjny (np. z zaatakowanego kontenera uzyskuje dostęp do krytycznych plików systemowych gospodarza – atak typu *container escape*)
- produkcyjne demony Dockera (np. z zaatakowanego systemu operacyjnego gospodarza obniża domyślne ustawienia zabezpieczeń Dockera i uruchamia kontener)
- sieć produkcyjna (np. z zaatakowanego systemu operacyjnego gospodarza przekierowuje ruch sieciowy)

Pośredni agresor ma te same możliwości co bezpośredni, ale dodatkowo wykorzystuje słabości ekosystemu Dockera (np. repozytoria kodu i rejestrów obrazów) aby dotrzeć do środowiska produkcyjnego.

W zależności od fazy ataku zidentyfikować można następujące cele: kontenery, system operacyjny gospodarza, kontenery "sąsiedzi" czyli takie, które współdzielą system gospodarza, repozytoria kodów, rejetry obrazów oraz sieć.

W celu ataku na środowisko kontenerowe, rozważany zostanie podzbiór wszystkich potencjalnych wektorów ataku:

- kontenery Docker
- repozytoria kodów
- rejstry obrazów

Podzbiór został określony w powyższy sposób ze względu na silne powiązanie z publicznie dostępnymi usługami i interfejsami. Pozostałe, pominięte wektory ataku mogą obejmować:

- system operacyjny gospodarza
- sieć administracyjną (służącą do zarządzania systemem)
- fizyczny dostęp do maszyn

6.2. Identyfikacja podatności

W następnych podrozdziałach zostaną przeanalizowane główne komponenty ekosystemu Docker co pozwoli na ujawnienie części z ich powierzchni ataku. Stosując podejście góra-dół (ang. top-down) zidentyfikowane zostanie pięć kategorii podatności, każda związana z inną warstwą ekosystemu. Aby wzbogacić analizę wykorzystane zostaną wcześniej już przytoczone typowe przypadki użycia, znane kategorie podatności (np. CVE — Common Vulnerabilities and Exposures [MITRE 2019]) oraz skala wzmocnień systemu (np. SELinux).

Wyróżnione kategorie są wymienione od najodleglejszej do najbliższej (ang. remote to local) względem produkcyjnego systemu kontenerów Docker. Analiza zakłada minimalną (domyślną) konfigurację:

- Podatności w konfiguracji
- Podatności w procesie dystrybucji obrazu
 - Docker jako menadżer pakietów
 - Zautomatyzowane rurociągi CI/CD
- Podatności wewnętrz obrazu
- Podatności bezpośrednio powiązane z Dockerem
- Podatności jądra Linuxa

6.3. Podatności w konfiguracji

Domyślna konfiguracja Dockera jest względnie bezpieczna ponieważ zapewnia izolację pomiędzy kontenerami i ogranicza dostęp kontenerów do gospodarza. Kontener jest umieszczony we własnej przestrzeni nazw i grupie kontrolnej, a także posiada tylko następujące uprawnienia (*capabilities*) [DOCK 2019d]: **SETPCAP, MKNOD, AUDIT_WRITE, CHOWN, NET_RAW, DAC_OVERRIDE, FOWNER, FSETID, KILL, SETGID, SETUID, NET_BIND_SERVICE, SYS_CHROOT, SETFCAP**

6.3.1 Podatności

Użycie niektórych opcji w trakcie uruchomienia demona Dockera lub przekazanych w komendzie uruchamiającej kontener może zapewnić kontenerom rozszerzony dostęp do gospodarza. Przykładowo:

- Montowanie wrażliwych folderów z systemu gospodarza w kontenerze

- Konfiguracja TLS zewnętrznych rejestrów obrazów Dockera
- Zmiana uprawnień do gniazda kontrolnego Dockera
- Zarządzanie grupami kontrolnymi innych kontenerów
- Opcje bezpośrednio zapewniające kontenerom rozszerzony dostęp do gospodarza (*--privileged*, dodatkowe uprawnienia)

6.3.2 Ataki

Przykładowo opcja *--uts=host* umieszcza kontener w przestrzeni nazw UTS gospodarza co pozwala kontenerowi odczytać i zmienić nazwę oraz domenę hosta. Opcja *--cap-add=<CAP>* nadaje kontenerowi określone uprawnienia, czyniąc go potencjalnie bardziej szkodliwym dla gospodarza. Dzięki opcji *--cap-add=SYS_ADMIN* kontener może ponownie zamontować podkatalogi */proc* i */sys* oraz zmienić parametry jądra systemu gospodarza, co prowadzi do potencjalnych luk w zabezpieczeniach, takich jak wyciek danych lub odmowa usługi.

Poza przedstawionymi opcjami uruchomienia kontenera również kilka konfiguracji po stronie gospodarza może ułatwić drogę do ataku. Nawet domyślne właściwości pozwalają w niektórych przypadkach na atak typu DoS. Docker pozwala na wybór sterownika pamięci dyskowej, a część z nich (np. AUFS) nie ogranicza użycia dysku przez kontenery. Kontener z woluminem pamięci może go zapełnić i wpływać na inne kontenery na tym samym systemie, a nawet na sam system gospodarza. Jeśli pamięć Dockera, znajdująca się w */var/lib/docker* nie jest zamontowana na osobnej partycji doprowadzi to do całkowitego zawieszenia działania systemu. Wspomniany już sterownik AUFS jest domyślnym sterownikiem dla Dockera w przypadku Ubuntu 14.04 i Dockera w wersji niższej niż 18.06 (aktualizacja z sierpnia 2018).

6.3.3 Zapobieganie atakom

W celu ograniczenia szkodliwych opcji, które mogą prowadzić do uzyskania dostępu do gospodarza przez kontener, Center for Internet Security przygotowało specyfikację testu o nazwie Docker Benchmark [CENT 2018]. Przedstawia ona zalecenia dotyczące wszystkich części środowiska Docker z podziałem na dwie kategorie: "Scored" oraz "Not scored". W trakcie ewaluacji zaleceń tylko te oznaczone jako "Scored" wpływają na ostateczny wynik testu. Dodatkowo, każde z zaleceń posiada poziom:

- "Level 1" – zalecenia są relatywnie łatwe do spełnienia i zapewniają zwiększyony poziom bezpieczeństwa, nie wpływając przy tym negatywnie na wydajność systemu

- "Level 2" – zalecenia są przeznaczone dla środowisk, w których bezpieczeństwo jest najważniejsze i dopuszczalne jest poświęcenie w tym celu wydajności systemu

Zalecenia te podzielone są na sześć kategorii:

Konfiguracja systemu gospodarza

Kategoria określa zalecenia dotyczące bezpieczeństwa, których należy przestrzegać, aby przygotować system gospodarza kontenerów Dockera co tworzy solidne i bezpieczne podstawy do konteneryzacji aplikacji. Wśród zaleceń znajduje się wspomniane już wcześniej utwardzanie jądra Linuxa, ale także utrzymywanie aktualnej wersji Dockera, upewnienie się, że tylko zaufani użytkownicy mają dostęp do demona Dockera oraz stworzenie osobnej partycji dyskowej dla kontenerów. Ponadto, zaleca się wykorzystanie narzędzia *auditctl* do wykonywania audytów na demonie Dockera i powiązanych z nim plikach. Wyniki audytu powinny być przechowywane w logach, które są okresowo analizowane. Pliki bądź katalogi, które należy poddać audytom to:

- *docker.service*
- *docker.socket*
- */etc/docker*
- */etc/default/docker*
- */etc/sysconfig/docker*
- */etc/docker/daemon.json*
- */usr/bin/containerd*
- */usr/sbin/runc*
- */var/lib/docker*

Konfiguracja demona Dockera

Zalecenia w tej kategorii zmieniają konfigurację i zabezpieczają demona Dockera, a tym samym wpływają na wszystkie kontenery uruchomione w systemie. Zalecane jest zbieranie logów na poziomie *info* (*--log-level="info"*) oraz przechowywanie ich z wykorzystaniem scentralizowanego, zewnętrznego systemu. Należy wymusić uwierzytelnianie przy użyciu protokołu TLS (*--tlsverify*, *--tlscacert*, *--tlscert*, *--tlskey*), a także zainstalować plugin dodający wymóg autoryzacji podczas interakcji z demonem Dockera. W celu ograniczenia zasobów systemu, powinien

zostać nałożony limit przy pomocy narzędzia *ulimit* oraz ustanowiona domyślna grupa kontrolna dla kontenerów Dockera (*--cgroup-parent=default*). Aktywacja wsparcia dla przestrzeni nazw użytkowników (*--userns-remap=default*) pozwala na zwiększenie izolacji.

Okresowo powinno sprawdzać się czy demon Dockera nie korzysta z niezabezpieczonych rejestrów obrazów. Zaleca się również zablokowanie komunikacji sieciowej pomiędzy kontenerami z wykorzystaniem domyślnego interfejsu sieciowego typu *bridge* (*--icc=false*) oraz wyłączenie możliwości zwiększenia uprawnień przez kontenery poprzez użycie *suid* lub *sgid* (*--no-new-privileges*). Należy upewnić się, że demon nie korzysta z niebezpiecznego sterownika pamięci dyskowej AUFS (*--storage-driver aufs*), a także sprawdzić czy nie są aktywne żadne z funkcjonalności eksperymentalnych.

Pliki konfiguracyjne demona Dockera

Kategoria definiuje zasady dotyczące uprawnień dostępu do plików i katalogów powiązanych z demonem Dockera. Zawarte w nich są wrażliwe dane konfiguracyjne, które powinny być odpowiednio zabezpieczone w celu poprawnego działania demona. Zalecane uprawnienia przedstawia tabela 6.1

Plik / katalog	Właściciel	Grupa	Uprawnienia
docker.service	root	root	644
docker.socket	root	root	644
/etc/docker	root	root	755
certyfikaty rejestrów	root	root	444
certyfikaty TLS	root	root	444
/var/run/docker.sock	root	docker	660
daemon.json	root	root	644
/etc/default/docker	root	root	644
/etc/sysconfig/docker	root	root	644

Tabela 6.1: Zalecani właściciele, grupy i uprawnienia plików demona Dockera

Obrazy kontenerów i pliki Dockerfile

Obrazy podstawowe i pliki Dockerfile określają podstawy działania kontenerów. Wykorzystanie odpowiednich obrazów podstawowych jest bardzo ważne podczas budowania infrastruktury opartej o kontenery. Należy korzystać tylko z zewnętrznych obrazów, które są oznaczone etykietą *trusted*, a do pobrania i dystrybucji obrazów wykorzystywać Content Trust. Wewnątrz

obrazu powinny być instalowane tylko i wyłącznie niezbędne paczki. Polecenia aktualizacji wydawane menadżerom paczek nie mogą być jedyną komendą warstwy obrazu gdyż powoduje to ich cachowanie (ewentualnie można skorzystać z opcji `--no-cache`). Co więcej, okresowo należy skanować obrazy pod kątem podatności zainstalowanych paczek i natychmiastowo aplikować aktualizacje bezpieczeństwa.

Każdy kontener powinien korzystać z użytkownika innego niż użytkownik `root` (dyrektywa `USER`) oraz definiować sposób na okresowe sprawdzanie stanu kontenera (dyrektywa `HEALTH-CHECK`). Dyrektywa `ADD` ma możliwość pobrania plików z zewnętrznego źródła i rozpakowania ich co powoduje dodatkowe podatności, dlatego też zaleca się aby zastąpić wszystkie dyrektywy `ADD` dyrektywami `COPY`. Ostatecznie, żadne wrażliwe dane uwierzytelniające (ang. secrets) nie powinny znaleźć się w obrazie.

Uruchamianie kontenerów

Sposób w jaki uruchamiane są kontenery niesie ze sobą wiele implikacji związanych z bezpieczeństwem. Niektóre z opcji mogą powodować narażenie systemu gospodarza i innych kontenerów. Ważne jest zatem aby zweryfikować z jakimi parametrami są uruchamiane poszczególne kontenery.

W ramach tej kategorii zostały przedstawione zalecenia, które powinny zostać aplikowane i dostosowywane w zgodzie z obowiązującą polityką bezpieczeństwa organizacji. W szczególności, należy:

- jeśli to możliwe, skorzystać z AppArmor lub SELinux
- ograniczyć uprawnienia kontenerów do minimum i nie korzystać z opcji `--privileged`
- nie montować wrażliwych katalogów systemu gospodarza w kontenerach
- nie uruchamiać demona SSH wewnętrz kontenerów
- otwierać tylko niezbędne porty
- nie współdzielić interfejsu sieciowego systemu gospodarza z kontenerami
- nałożyć odpowiednie ograniczenia na użycie pamięci i czasu procesora
- montować system plików kontenera w trybie tylko do odczytu
- zdefiniować politykę restartu kontenera w przypadku błędów i ograniczyć liczbę restartów do 5

- nie współdzielić przestrzeni nazw PID, User, IPC i UTS systemu gospodarza z kontenerami
- potwierdzić korzystanie z grup kontrolnych
- nie montować gniazda demona Dockera wewnętrz kontenerów

Działania organizacyjne

Organizacje powinny rozszerzać swoje polityki bezpieczeństwa uwzględniając specyficzną naturę konteneryzacji. Zalecenia z tej kategorii działają głównie w kwestii rozwijania zbioru "dobrych praktyk" wewnętrz zespołów deweloperskich zarządzających maszynami produkcyjnymi.

Obrazy Dockera otagowane inną etykietą niż *latest* służą głównie w celach awaryjnych kiedy trzeba szybko wycofać wdrożenie do którejś z poprzednich wersji. Zaleca się nieprzetrzymywania dużej ilości obrazów na jednym systemie gospodarza. Powinno definiować się metodyki pracy, które uwzględniają usuwanie zbędnych, przestarzałych obrazów. Podobna sytuacja dotyczy przechowywania większej liczby kontenerów, co powoduje niepotrzebne zużycie zasobów systemu gospodarza. Biorąc pod uwagę szybkość uruchomienia nowego kontenera nie powinno przechowywać się żadnych zatrzymanych kontenerów. Może to prowadzić do pomyłki lub niewłaściwej konfiguracji wynikającej z błędu ludzkiego. Liczba zapisanych obrazów i kontenerów powinna stanowić niezbędnego minimum. Polecenie *docker system prune -a* pozwala usunąć wszystkie nieużywane zasoby Dockera, t.j. obrazy, kontenery, interfejsy siecowe i woluminy.

6.4. Docker jako menadżer pakietów

6.4.1 Podatności

Architektura Docker Hub jest podobna do repozytorium pakietów z demonem Dockera działającym jako menedżer pakietów. Dlatego też Docker jest podatny na te same słabości co menedżerowie pakietów. Luki te obejmują przetwarzanie, przechowywanie i dekompresję potencjalnie złośliwego kodu wykonywanego przez demona Dockera z uprawnieniami użytkownika *root*. Ten kod może zostać zmodyfikowany przez twórcę (złośliwy obraz) lub podczas przesyłania (na przykład w wyniku opcji *--insecure-register* podanej demonowi Dockera, która umożliwia atak typu man-in-the-middle pomiędzy rejestrem, a systemem).

6.4.2 Ataki

Ataki na menedżerów pakietów są możliwe [CAPP 2008], jeśli osoba atakująca kontroluje część sieci pomiędzy systemem gospodarza, a repozytorium. Udany atak pozwoliłby agresorowi na umieszczenie złośliwych obrazów w systemie gospodarza. Prowadzi to do naruszenia bezpieczeństwa obrazów, które mogą wykorzystać luki w procesie ich przetwarzania. Po pierwsze,

ponieważ obrazy są skompresowane, specjalnie spreparowany obraz zawierający ogromny plik wypełniony śmieciowymi danymi (np. samymi zerami) wypełniłby urządzenie pamięci masowej gospodarza powodując odmowę usługi (atak typu zipbomb). Dodatkowo, z racji, że obrazy są dekompresowane w systemie plików gospodarza, w przeszłości możliwe były ataki typu path traversal (CVE-2014-9356, CVE-2018-15664 [RED 2019]). Wykorzystanie jednej z tych luk sprawiło, że dekomprezja obrazów (wykonywana jako użytkownik *root*) odbywająca się za pomocą bezwzględnych dowiązań symbolicznych umożliwiła zastąpienie plików binarnych gospodarza plikami binarnymi z obrazu. Inne możliwe ataki obejmują wstrzykiwanie kodu do obrazów (code injection) lub odtwarzanie starych obrazów zawierających znane luki w zabezpieczeniach (replay attack).

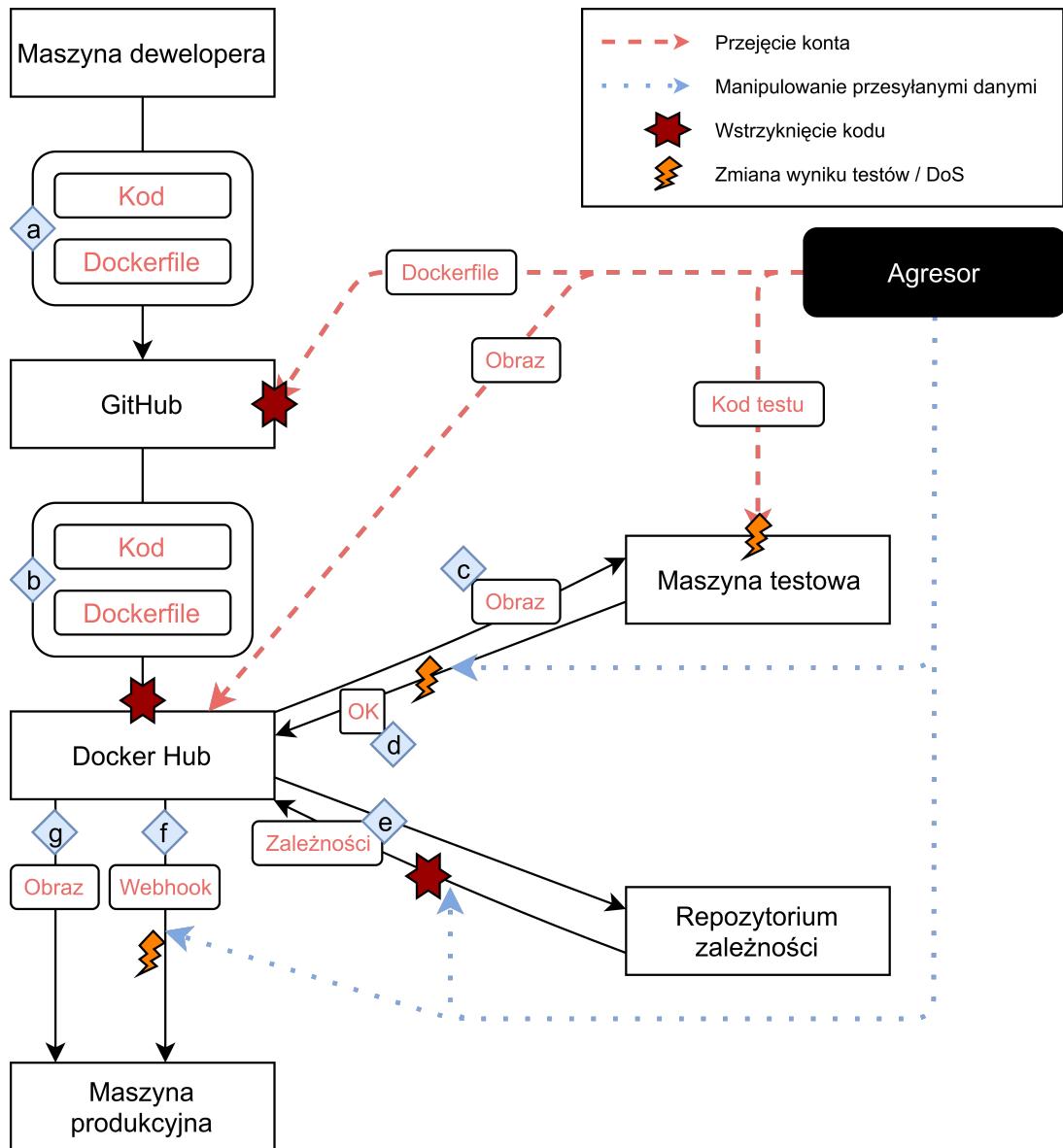
6.4.3 Zapobieganie atakom

Przed wersją Dockera 1.8 jedyną ochroną było użycie TLS podczas połączenia z rejestrzem, które można wyłączyć. W wersji 1.8 Docker wprowadził Content Trust, architekturę do podpisywania obrazów w taki sam sposób, w jaki pakiety są podpisywane przez menedżerów pakietów. Poza polepszeniem bezpieczeństwa zrodziło to także dwa problemy. Po pierwsze, Content Trust można wyłączyć, przekazując opcję *--disable-content-trust* do demona Dockera. Zapewne jest to wygodna opcja dla prywatnych rejestrów jednak stanowi lukę w zabezpieczeniach. Po drugie, proces podpisu obrazu wymaga zaufania do deweloperów, co jest możliwe tylko dzięki ujednoliceniu podpisu obrazu. Co więcej, to rozwiązanie nie jest skalowalne, ponieważ wymaga od tysięcy programistów podpisywania rejestrów własnym kluczem. Poza zwyczajnym wyzwaniem technicznym jakie stanowi dystrybucja kluczy jest to problem zaufania i skali.

6.5. Zautomatyzowane rurociągi CI/CD

6.5.1 Podatności

Zautomatyzowane wdrożenia i webhooki proponowane przez Docker Hub są kluczowym elementem tego procesu dystrybucji. Prowadzą do rurociągu, w którym każdy element ma pełny dostęp do kodu ostatecznie trafiającego na produkcję. Coraz więcej elementów jest również hostowanych w chmurze obliczeniowej. Aby zautomatyzować to wdrożenie, Docker proponuje automatyczne budowanie obrazów w Docker Hub, uruchamiane zdarzeniem z zewnętrznego repozytorium kodu (np. GitHub, BitBucket)(rys. 6.1b). Następnie wysyłane jest żądanie HTTP do gospodarza Dockera dostępnego publicznie w Internecie aby powiadomić go, że nowy obraz jest dostępny. Powoduje to pobranie obrazu z rejestrów i kontener uruchamia się ponownie na nowym obrazie (rys. 6.1f, rys. 6.1g). W tym typie rurociągu zatwierdzenie kodu w repozyto-



Rysunek 6.1: Rurociąg CI/CD automatycznie wdrażający kod przy użyciu usług GitHub, Docker Hub, zewnętrznej maszyny testowej i repozytorium zależności

rium (rys. 6.1a) zapoczątkuje budowę nowego obrazu (rys. 6.1b), który zostanie automatycznie uruchomiony w produkcji (rys. 6.1f, rys. 6.1g). Przed ostatecznym wdrożeniem można dodać opcjonalne kroki testowe, które potencjalnie mogą być uruchamianie u jeszcze innego dostawcy (np. GitHub Circle CI, BitBucket Pipelines). W tym przypadku Docker Hub wysyła wpierw żądanie HTTP do maszyny testowej (rys. 6.1c), która następnie pobiera obraz, uruchamia testy i wysyła wyniki do Docker Hub za pomocą adresu URL wywołania zwrotnego (rys. 6.1d). Sam proces budowania często pobiera zależności z repozytoriów podmiotów zewnętrznych (rys. 6.1e), czasem przez niezabezpieczony kanał (którym można manipulować). Cały potok kodu pokazano na rysunku 6.1.

6.5.2 Ataki

W zaprezentowanej architekturze próby ataków obejmują przechwytywanie kont, manipulowanie komunikacją sieciową (w zależności od zastosowania TLS) oraz ataki z wykorzystaniem osób wtajemniczonych (insider attack). Taka konfiguracja dodaje zewnętrzne kroki pośrednie do ścieżki, którą musi przebyć kod. Dostawcy usług mają własne systemy uwierzytelniania i powierzchnie ataku, ogólnie zwiększając globalną powierzchnię ataku.

Przykładowo, zakładając scenariusz, w którym skonfigurowana jest powyższa architektura agresor może uzyskać dostęp do konta w usłudze GitHub (choćby przy pomocy socjotechniki). Prowadzi to do wykonania złośliwego kodu na zbiorze maszyn produkcyjnych w przeciągu kilku minut od przejęcia konta. Wyniki takiego ataku mogą być brzemienne w skutkach jeśli na zmiany w obrazie nasłuchuje wiele klastrów maszyn produkcyjnych. Badania pokazują, że implementacje podobnych architektur wdrażają kod z repozytorium na maszyny produkcyjne w czasie poniżej 5 minut [BEZE 2016].

Warto zaznaczyć, że choć zagrożenia dotyczące repozytorium kodu są niezależne od Dockera, automatyczne wdrożenie kodu do produkcji znacznie zwiększa liczbę zainfekowanych komputerów – nawet jeśli złośliwy kod zostanie usunięty w ciągu kolejnych kilku minut. Atak może się również odbyć na poziomie konta Docker Hub, a konsekwencje będą takie same. Przejęcie konta nie jest nowym problemem, ale powinno być traktowane z coraz większą troską w związku z mnożeniem się kont u różnych dostawców. Ponieważ do zainfekowania kodu dochodzi na zaatakowanym komputerze, TLS jest bezużyteczny. W rzeczywistości złośliwy kod jest „bezpiecznie” dystrybuowany przez TLS do różnych repozytoriów. Co więcej, z powody szyfrowania komunikacji w protokole, czuwające, ewentualne systemy IDS (Intrusion Detection System) i IPS (Intrusion Prevention System) niczego nie wykryją. Można, co prawda, skonfigurować je do działania w trybie man-in-the-middle i deszyfrowania ruchu sieciowego, jednak prowadzi to do kolejnych potencjalnych luk w zabezpieczeniach.

Ponadto, chociaż transfer kodu jest zwykle zabezpieczony za pomocą protokołu TLS, to może nie dotyczyć to wywołań API uruchamiających budowanie i wywołań zwrotnych. Manipulowanie tymi danymi może prowadzić do błędnych wyników testów, niechcianych restartów kontenerów, itp. Co więcej, taka konfiguracja nie jest zgodna ze schematem Content Trust, ponieważ kod jest przetwarzany przez podmioty zewnętrzne znajdujące się pomiędzy deweloperem, a środowiskiem produkcyjnym. Content Trust zapewnia środowisko, w którym zaufany jest jeden podmiot (osoba lub organizacja, która podpisała obrazy). Podczas gdy w niniejszym przypadku zaufanie jest podzielone na kilka podmiotów zewnętrznych, z których każdy może narazić obrazy.

6.6. Podatności wewnątrz obrazu

6.6.1 Podatności

Podczas indeksowania Docker Hub wykazano, że 36% oficjalnych obrazów zawiera luki CVE o wysokim poziomie zagrożenia, a 64% zawiera luki o średnim lub wysokim poziomie zagrożenia. Wartości te spadają odpowiednio do 23% i 47% dla obrazów oznaczonych jako *latest* [BANY 2015]. Pomimo iż tak oznaczone obrazy są najczęściej pobieranymi w Docker Hub, zawierają one znaczną liczbę luk w zabezpieczeniach, w tym niektóre z ostatnio popularnych klas takich jak Shellshock i Heartbleed.

Promowana przez Docker metodyka DevOps pozwala programistom samemu pakować swoje aplikacje, łącząc w ten sposób środowiska programistyczne i produkcyjne, a tym samym potencjalnie wprowadzając luki. W ostatecznej wersji obrazu Docker mogą pozostać deweloperskie wersje pakietów lub narzędzia programistycznych zwiększać tym samym jego powierzchnię ataku (np. narzędzie do debugowania).

Stworzone obrazy często zawierają nieaktualne wersje pakietów ponieważ ich obraz podstawowy (np. *ubuntu* lub *centos*) jest przestarzały lub ponieważ proces komplikacji pobiera nieaktualny kod z jakiegoś zdalnego repozytorium. Ogrom komplikacji obrazów – praktycznie jedna dla każdego zatwierdzenia kodu w repozytorium projektu – prowadzi do utrzymywania się przestarzałych obrazów, wciąż dostępnych w repozytoriach. Z kolei szybkie cykle programowania zwykle koncentrują się tylko i wyłącznie na najnowszych wersjach.

6.6.2 Ataki

Wykorzystanie takich luk w zabezpieczeniach jest istotne w kontekście ataku z zewnątrz. Możliwe są klasyczne metody ataków na aplikację, pod warunkiem, że kontener ujawnia punkt ataku (otwarty port sieciowy, dane wejściowe, itp.). Ponadto, obrazy zbudowane z zewnętrznych repozytoriów kodu, tj. obrazy, które pobierają dane z innych repozytoriów podczas procesu komplikacji (rys. 6.1e), są zależne od tego repozytorium i bezpieczeństwa połączenia użytego do pobrania tych danych. Te repozytoria – nie zawsze oficjalne – są kolejnym punktem ataku typu wstrzykiwanie kodu.

6.6.3 Zapobieganie atakom

Docker Hub korzysta z Docker Security Scanning. Użytkownicy mogą skanować obrazy w prywatnych repozytoriach, aby sprawdzić, czy są wolne od znanych luk w zabezpieczeniach. Skanowanie przechodzi przez każdą warstwę obrazu, identyfikując komponenty oprogramowania

i obliczając z nich funkcję skrótu SHA. Następnie są one porównywane z bazą danych CVE w celu uzyskania informacji o znanych lukach bezpieczeństwa. Całkowite skanowanie trwa od 1 do 24 godzin, w zależności od wielkości ocenianych obrazów. Udzielone wsparcie jest ograniczone ze względu na koszt usługi tylko do użytkowników Docker EE (Enterprise Edition). Co więcej, jeśli luka nie jest częścią bazy danych, skanowanie nie może jej ujawnić, przez co usługa nie reaguje na nowo pojawiające się luki [DOCK 2019k].

6.7. Podatności usługi Docker

6.7.1 Podatności

Podatności w Docker i *libcontainer* dotyczyły głównie ataków na system gospodarza lub na bezpośrednio atakowany kontener:

- uzyskanie dostępu do demona Dockera (CVE-2019-15752, CVE-2016-3697, CVE-2015-3627, CVE-2014-3499)
- uruchomienie arbitralnego kodu wewnętrz kontenera (CVE-2019-5736, CVE-2014-9357, CVE-2014-6407)
- Denial of Service (CVE-2017-14992, CVE-2016-6595)
- uzyskanie dodatkowych uprawnień wewnętrz kontenera (CVE-2016-8867, CVE-2014-6408)
- pozyskanie informacji o systemie gospodarza (CVE-2015-3630)
- atak typu Directory Traversal w systemie gospodarza (CVE-2018-15664)

Ponieważ procesy kontenerowe często działają z uprawnieniami użytkownika *root*, mają one dostęp do odczytu i zapisu na całym systemie plików gospodarza w momencie gdy dojdzie do ucieczki z kontenera. W ten sposób mogą nadpisywać pliki binarne gospodarza, co prowadzi do wykonania dowolnego kodu z uprawnieniami administratora.

Najnowsza podatność, która pojawiła się w Dockerze w wersji 19.03.0 (CVE-2019-14271) została załatwiona aktualizacją do wersji 19.03.1. Aktualizacja bezpieczeństwa została opublikowana tego samego dnia, w którym pojawiły się raporty dotyczące podatności. Pokazuje to, że zespół pracujący nad Dockerem bardzo szybko reaguje na pojawiające się podatności. Jednakże, zaaplikowanie aktualizacji nadal jest zależne od użytkownika, co najczęściej powoduje znaczne wydłużenie czasu, w którym podatność jest otwarta na ataki.

6.7.2 Ataki

Oprócz przestrzeni nazw jądra, grup kontrolnych, zmniejszania uprawnień kontenerów i ograniczeń w montowaniu woluminów danych, obowiązkowa kontrola dostępu (ang. Mandatory Access Control) może wymuszać ograniczenia w przypadku nietypowego wykonywania aplikacji. Takie podejście jest widoczne w domyślnym profilu AppArmor dla Dockera. Jednakże, wspomniana domyślna konfiguracja mogłaby być bardziej restrykcyjna. Z reguły polityki Apparmor zwykle definiują białe listy, jawnie deklarując zasoby, do których każdy proces może uzyskać dostęp. Jednocześnie odmawiają jakiegokolwiek innego dostępu [CHOI 2017]. Jednak domyślny profil Dockera zapewnia kontenerom pełny dostęp do urządzeń sieciowych, systemów plików wraz z pełnym zestawem uprawnień i zawiera tylko niewielką listę dyrektyw odmawiających dostępu, składających się z czarnej listy.

6.8. Podatności jądra Linuxa

Z racji iż kontenery działają na tym samym jądrze co gospodarz to są one podatne na ataki na jądro. Atak dający pełne uprawnienia użytkownika *root* do kontenera może pozwolić osobie atakującej na ucieczkę z kontenera i narażenie systemu gospodarza (powodując naruszenie izolacji i integralności, a także ujawnienie danych).

6.9. Ocena podatności

Ocena poziomu zagrożenia wyjaśnionych podatności jest zależna od każdego z przedstawionych przypadków użycia (tabela 6.2). Zamiast koncentrować się na konkretnym przypadku zastosowania z określona aplikacją, przeanalizowany został ekosystem Dockera w powiązaniu z typowymi przypadkami użycia. Takie podejście jest również zgodne z metodologią NIST, która definiuje kontekst jako środowisko podejmowania decyzji na bazie ryzyka i wymiar, który należy wziąć pod uwagę podczas przeprowadzania oceny podatności na zagrożenia [NATI 2012].

Ocena skupia się wyłącznie na różnicę między przypadkami użycia, przy czym wszystkie inne wymiary (zagrożenia i środki zapobiegawcze) są równe. Szeroko rozpowszechniony przypadek użycia (używanie kontenerów jako maszyn wirtualnych) ujawnia najwięcej podatności. Co więcej, niezależnie od rozważanych przypadków użycia, podatności jądra Linuxa i podatności bezpośrednio związane z Dockerem mają podobny poziom zagrożenia.

6.10. Alternatywy dla Dockera

Od kilku lat, równolegle do kontenerów, rozwijane są również unikernelle. Chociaż nie są one jeszcze zbyt powszechnie używane w produkcji, rozwiązują problem izolacji poprzez uruchamianie własnego jądra, specjalnie zoptymalizowanego dla jednej aplikacji. Osiągają wydajność

Kategoria podatności	Zalecany przypadek użycia	Rozpowszechniony przypadek użycia	Przypadek użycia dostawców chmurowych
Podatności w konfiguracji	Umiarkowany Domyślna konfiguracja Dockera jest względnie bezpieczna. Możliwe obniżenie bezpieczeństwa konfiguracji.	Bardzo wysoki Bardzo prawdopodobna niebezpieczna konfiguracja.	Wysoki Kontenery w podzie współdzielą przestrzeń nazw Network z Kubernetes.
Podatności w procesie dystrybucji obrazu	Bardzo wysoki Promowana metodyka DevOps. Użycie automatyzacji na każdym kroku w celu skrócenia cyklu rozwoju oprogramowania	Umiarkowany Kontery używane jako maszyny wirtualne powodując mniejszą ilość ciągłej integracji	Wysoki Automatyzacja każdej warstwy w celu ciągłego wdrażania zmian
Podatności wewnętrz obrazu	Umiarkowany Domyślnie odsłonięta ograniczona powierzchnia ataku	Bardzo wysoki Bardzo prawdopodobne użycie ciężkich obrazów z powierzchnią ataku większą od obrazów mikroserwisów	Umiarkowany W zależności od obrazu mogą tu wystąpić przypadki użycia zalecane i rozpowszechnione
Podatności bezpośrednio powiązane z Dockerem	Podobny poziom zagrożenia we wszystkich przypadkach użycia		
Podatności jądra Linuxa	Podobny poziom zagrożenia we wszystkich przypadkach użycia		

Tabela 6.2: Względna skala ocen poziomu zagrożenia w trzech przypadkach użycia

zbliżoną lub nawet lepszą niż kontenery, a ich bardzo szybki rozruch pozwala na uruchomienie unikernela w celu spełnienia określonego żądania. Dalszy rozwój technologiczny może spowodować, że w nadchodzących latach unikernele staną się poważnym konkurentem dla kontenerów [MADH 2015].

7. Realizacja aplikacji

7.1. Motywacja

Analiza podatności pokazała, że większość wektorów ataku na ekosystem Docker pochodzi z błędów konfiguracji demona i kontenerów Docker, a także z błędów ludzkich związanych z wielousługową metodą wdrażania oprogramowania. Wsparcie ze strony twórców Dockera pozwala na szybkie łatanie podatności środowiska. Jednakże aplikowanie aktualizacji bezpieczeństwa nadal jest zależne od użytkownika końcowego.

Błędy konfiguracyjne mogą pojawić się na wielu etapach pracy z ekosystemem Docker. Pomijając zewnętrzne zależności można wyróżnić takie elementy środowiska jak pliki konfiguracyjne demona Docker, uprawnienia powiązanych z nim plików i katalogów systemu gospodarza oraz opcje przekazane przez użytkownika w trakcie uruchomienia demona. Ponadto, konfiguracja kontenerów jest podatna na błędy wynikające z wykorzystania nieodpowiednich dyrektyw w plikach Dockerfile lub opcji użytych w trakcie uruchamiania kontenerów. Polityka bezpieczeństwa powinna brać pod uwagę scenariusz w jakim jest aplikowana i być odpowiednio dostosowywana w celu osiągnięcia zamierzonych celów.

Popularność Dockera dała wielu programistom dostęp do bardzo złożonego narzędzia jakim są kontenery. Jednakże, nie każdy użytkownik Dockera jest specjalistą od spraw bezpieczeństwa systemów komputerowych. Wiąże się to z tworzeniem podatności wynikających z nieznajomości pewnych aspektów wykorzystywanej technologii. Celem twórców wszystkich elementów ekosystemu Dockera powinno być wsparcie użytkowników w kwestiach zwiększania bezpieczeństwa tworzonych przez nich aplikacji kontenerowych.

Wspomniany w poprzednich rozdziałach Docker Benchmark [CENT 2018] został przeanalizowany przez twórców Dockera w celu stworzenia narzędzia weryfikującego wykorzystanie zalecanych dobrych praktyk. Wynikiem tej pracy jest implementacja skryptu Docker Bench for Security. W sposób statyczny analizuje on konfigurację demona i kontenerów Dockera sprawdzając spełnienie zaleceń oznaczonych w Docker Benchmark jako "Scored" (88/115 wszystkich zaleceń) [DOCK 2019a].

W ramach pracy zaprojektowano aplikację wspomagającą zautomatyzowaną analizę kontenerów Dockera pod kątem wykrywania przydziału zbyt dużej ilości uprawnień. W porównaniu

do skryptu Docker Bench for Security [DOCK 2019a] aplikacja analizuje wykorzystanie uprawnień w trakcie działania kontenera. Wynikiem uruchomienia aplikacji jest raport przedstawiający porównanie zadeklarowanych w konfiguracji uprawnień kontenera z rzeczywiście użytymi uprawnieniami. Tym samym, daje to użytkownikowi możliwość zmniejszenia wektora ataku, którym są nadmiarowe uprawnienia. W kolejnych podrozdziałach aplikacja jest zwana również analizatorem.

7.2. Wykorzystane narzędzia

7.2.1 Berkeley Packet Filter

Filtry BPF wywodzą się z systemu BSD (Berkeley Software Distribution), jednak obecnie dostępne są w większości dystrybucji opartych o system operacyjny Unix. Filtry zapewniają interfejs warstwy łącza danych w postaci pseudo-urządzeń. Procesy przestrzeni użytkownika mogą uruchamiać program filtrujący, określający jakie pakiety mają zostać przez niego otrzymane.

Filtry implementuje się w postaci kodu bajtowego dla maszyny wirtualnej BPF (maszyna wirtualna BPF nie ma nic wspólnego z opisywanymi wcześniej maszynami wirtualnymi, służącymi do wirtualizacji systemów operacyjnych). Kompilator nakłada pewne ograniczenia dotyczące kodu bajtowego BPF: skoki są możliwe tylko w przód, zabronione jest używanie pętli, a wyjściowy kod bajtowy może zawierać maksymalnie 4096 linii kodu. Wszystko to ma na celu zapewnienie zakończenia programu filtra, który jest uruchamiany dla każdego pakietu przychodzącego i wychodzącego z danego gniazda [MCCA 1993].

7.2.2 Extended Berkeley Packet Filter

Extended Berkeley Packet Filter rozszerza zastosowanie maszyny wirtualnej BPF poprzez umożliwienie uruchomienia filtra na zdarzeniach innych niż przepływ pakietów. Przy pomocy eBPF możliwe jest również śledzenie (ang. tracing) operacji wejścia/wyjścia, systemu plików, opóźnień systemu, użycia procesora, zmian stosu oraz co ważne, wywołań systemowych. Wszystkie zdarzenia są podzielone na 4 główne kategorie:

- kprobes – dynamiczne śledzenie jądra
- uprobes – dynamiczne śledzenie przestrzeni użytkownika
- tracepoints – statyczne śledzenie jądra
- perf_events – próbkowanie okresowe i Performance Monitoring Counters

Tym samym filtry eBPF świetnie nadają się do tworzenia narzędzi, które w przyszłości mają zastąpić obecnie istniejące moduły jądra. Ma to na celu postępującą transformację jądra Linuxa

do architektury mikrojądra, w której większość funkcjonalności zdefiniowana i uruchamiana jest w przestrzeni użytkownika. Takie podejście eliminuje część z wad modułów, t.j. potrzebę rekompilacji jądra po ich instalacji oraz możliwość zatrzymania jądra po błędzie modułu [GREG 2019].

Programy eBPF wymagają jądra Linuxa w wersji przynajmniej 4.1, a wraz z kolejnymi wersjami jądra pojawiają się dodatkowe funkcjonalności. Przykładowo, wersja 4.7 jądra pozwala na statyczne śledzenie jądra przy pomocy rozwiązania *tracepoints*. Co więcej, pisanie programów eBPF nie różni się wiele od pisania filtrów BPF – składnia kodu bajtowego jest bardzo niskopoziomowa oraz posiada wspomniane wcześniej ograniczenia. Powoduje to wolną adopcję tego rozwiązania w przenoszeniu modułów jądra na programy przestrzeni użytkownika [JACK 2018].

7.2.3 BPF Compiler Collection

Berkeley Packet Filter Compiler Collection, w skrócie BCC, w znacznym stopniu ułatwia pisanie programów eBPF poprzez opakowanie kodu bajtowego BPF wewnętrz programów napisanych w języku C++ oraz udostępnienie interfejsów programistycznych w językach Lua i Python. Dodatkowo, repozytorium BCC zawiera dużą ilość przykładów i wbudowanych narzędzi (ponad 150) ułatwiających nowym programistom rozpoczęcie tworzenia własnych programów.

BCC dodatkowo analizuje kod programów napisanych w C++ dzięki czemu gwarantuje, że końcowy kod bajtowy BPF będzie poprawny. To narzędzie ma na celu zapewnienie interfejsu, który pozwoli na tworzenie tylko i wyłącznie poprawnych programów eBPF, zachowując równocześnie dostęp do wszystkich funkcjonalności. Ponadto, minimalizuje czas spędzony na przygotowaniu i komplikacji kodu bajtowego BPF. Umożliwia tym samym skupienie się na tworzeniu docelowej aplikacji [BCC 2019].

7.2.4 Python3.7

Python jest interpretowanym językiem programowania wysokiego poziomu z dynamiczną semantyką. Wbudowane wysokopoziomowe struktury w połączeniu z dynamicznym typowaniem i wiązaniem czynią go idealnym kandydatem do szybkiego rozwoju oprogramowania, a także tworzenia skryptów oraz łączenia istniejących komponentów większego systemu [PYTH 2019a].

Wersja 3.7 Pythona dostarcza wiele funkcjonalności przydatnych w tworzeniu analizatora. Ulepszone annotacje typów i wbudowany w bibliotekę standardową moduł *dataclasses* ułatwia tworzenie czystego kodu. Dodanie słów kluczowych *async* i *await* sprawia, że Python natywnie wspiera asynchronousność. Stworzona aplikacja w dużym stopniu wykorzystuje również moduł *threading* udostępniający klasę *Thread* dla programów wielowątkowych oraz moduł *queue* im-

plementującą synchronizowaną kolejkę [PYTH 2019b].

7.2.5 Docker SDK for Python

Docker SDK for Python jest biblioteką języka Python opakowującą interfejs programistyczny silnika Dockera. Umożliwia ona osiągnięcie wszystkiego na co pozwala również komenda *docker*, jednak z poziomu aplikacji Pythonowej, t.j. tworzenie obrazów, uruchomienie i zarządzanie kontenerami, wsparcie dla Docker Swarm, itp. W celu komunikacji z demonem Dockera należy stworzyć obiekt klasy *DockerClient*, który wspiera operacje synchroniczne oraz asynchroniczne [DOCK 2019e].

7.3. Schemat działania

Działanie analizatora można przedstawić na podstawie uproszczonego algorytmu składającego się z 5 kroków:

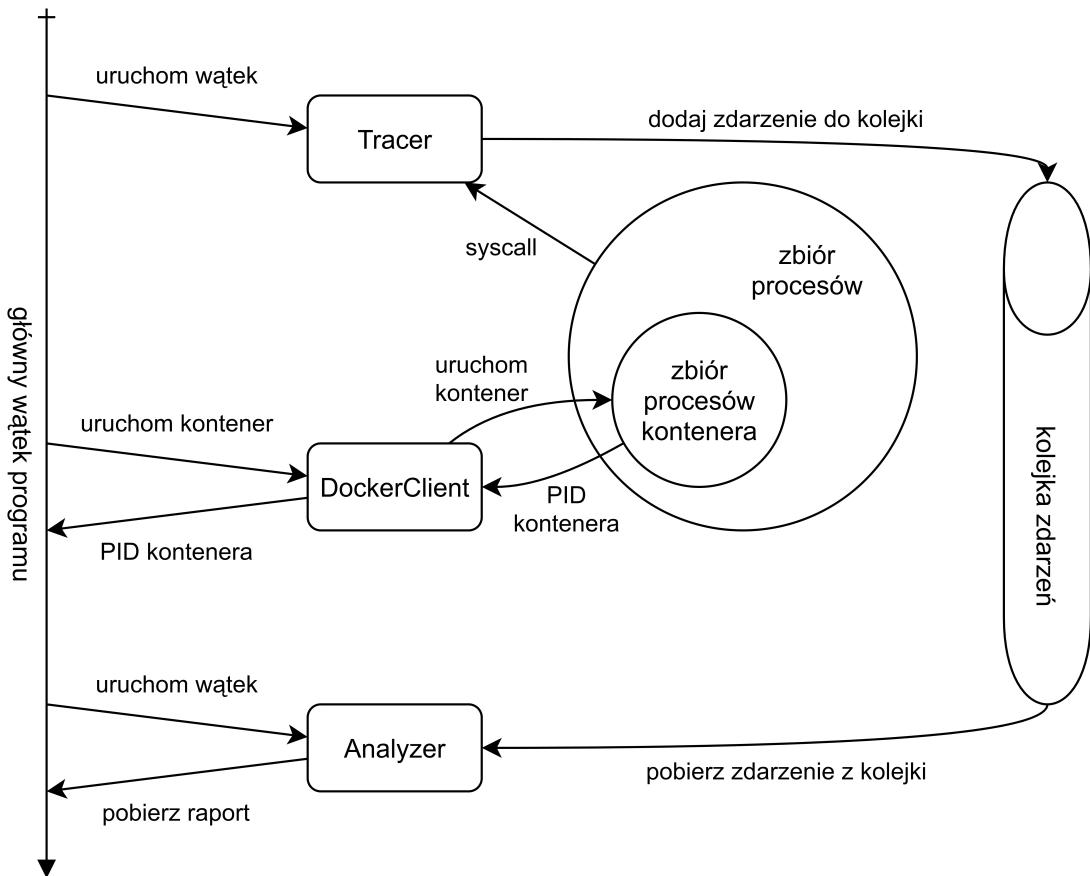
1. parsowanie argumentów linii poleceń
2. rozpoczęcie śledzenia wywołań systemowych
3. uruchomienie kontenera
4. rozpoczęcie analizy wywołań systemowych
5. zwolnienie zasobów i zakończenie programu

Uproszczony schemat przepływu sterowania i przepływu danych w analizatorze został przedstawiony na rysunku 7.1. Kolejne podrozdziały opisują bardziej szczegółowo poszczególne kroki.

7.3.1 Parsowanie argumentów linii poleceń

Analizator uruchamia się z linii poleceń przy pomocy polecenia *python3.7 main.py <image>*. Program akceptuje również te same opcje, które można przekazać do polecenia *docker run* w trakcie uruchamiania kontenera.

Klasa *Parser* implementuje obsługę opcji narzędzia Docker przy pomocy klasy *ArgumentParser* z modułu *argparse*. Niemożliwe było wykorzystanie oryginalnego kodu źródłowego interfejsu linii poleceń Dockera, gdyż tak samo jak silnik i demon Dockera, jest on napisany w języku Go. Wzięty pod uwagę był również parser narzędzia Docker Compose napisanego w Pythonie, który pozwala na definicję wielokontenerowych aplikacji Dockera w formacie pliku YAML. Niestety, Docker Compose pozwala na zadeklarowanie dodatkowej konfiguracji, która jest niedostępna w ramach polecenia uruchomienia kontenera. Co więcej, format zakłada inną składnię



Rysunek 7.1: Schemat przepływu sterowania i przepływu danych w analizatorze

opcji, więc wymagane byłoby mapowanie pomiędzy standardami Docker SDK for Python, a Docker Compose. Tym samym, najprostszym rozwiązaniem było stworzenie parsera od zera.

7.3.2 Uruchomienie kontenera

Kontener zostaje uruchomiony w głównym wątku programu poprzez obiekt klasy *ContainerManager*, która opakowuje obiekt klasy *DockerClient* z modułu *docker*. Opcje uzyskane z parsera zostają przekazane do wywołania metody *run*, która zwraca obiekt klasy *Container*. Obiekty tej klasy reprezentują stan kontenera w danym momencie czasu. Wywołanie metody *reload* powoduje wczytanie obecnego stanu kontenera. Początkowo, zwrócony obiekt nie posiada globalnego identyfikatora procesu punktu wejściowego kontenera (zwanego dalej, dla uproszczenia **identyfikatorem procesu kontenera**). Dlatego też, metoda *start* obiektu klasy *ContainerManager* odświeża obiekt kontenera do momentu uzyskania identyfikatora procesu kontenera, a następnie zwraca tę wartość.

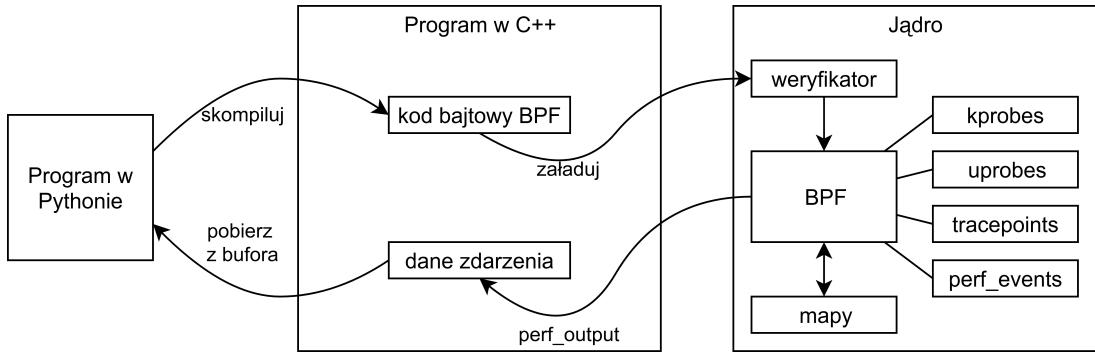
7.3.3 Śledzenie wywołań systemowych

Śledzenie wywołań systemowych odbywa się w osobnym wątku, który jest uruchamiany poprzez obiekt klasy *CapabilitiesTracer*. Metoda *start* wykorzystuje klasę *BPF* z modułu *bcc* narzędzia BCC w celu skompilowania programu eBPF napisanego w języku C++. Program eBPF zawiera w sobie funkcję *kretprobe__cap_capable*, która poprzez BCC zostaje załadowana do modułu BPF w jądrze. Dzięki temu, zostanie ona wywołana po każdym wywołaniu funkcji systemowej *cap_capable*, która służy do sprawdzenia czy dany proces posiada określone uprawnienie. Funkcja *kretprobe__cap_capable* poprzez argumenty posiada dostęp do kontekstu wywołania systemowego *cap_capable*: uprawnienia i stanu procesu, którego zapytanie dotyczy oraz wartości zwróconej z funkcji.

BCC udostępnia funkcję *bpf_get_current_task*, która zwraca strukturę zawierającą informacje dotyczące procesu, w którym doszło do wywołania systemowego. W strukturze tej znajduje się identyfikator procesu (*.pid*) oraz wskaźnik na strukturę tego samego typu dotyczącą rodzica (*.parent*). Iterując po wskaźnikach w góre drzewa procesów można sprawdzić czy wywołanie dotyczyło jednego z procesów kontenera. Jednakże, ograniczenia programów eBPF zabraniają wykorzystania pętli w kodzie bajtowym. Dlatego też skorzystano z metody odwijania pętli (ang. loop unrolling, loop unwinding), która zamienia pętlę na określoną liczbę następujących po sobie bloków wewnętrznych pętli. Oznacza to również, że liczba "iteracji" musi być z góry określona i ograniczona. Tym samym, istnieje możliwość wywołania systemowego *cap_capable* wewnętrz kontenera, które nie zostanie wykryte przez program eBPF.

Komunikacja programu eBPF z analizatorem odbywa się poprzez obiekt tablicy stworzony przy pomocy funkcji *BPF_PERF_OUTPUT* udostępnianej przez BCC. Obiekt korzysta z bufora *perf_output*, który pozwala na emitowanie zdarzeń z przestrzeni jądra do przestrzeni użytkownika poprzez wywołanie na nim metody *perf_submit*. Obiekt klasy *CapabilitiesTracer* najpierw definiuje wywołanie zwrotne (ang. callback) poprzez metodę *open_perf_buffer* klasy *BPF*, a następnie oczekuje na zdarzenia przy pomocy metody *perf_buffer_poll*. Załadowanie programu eBPF i przepływ danych przedstawione są na rysunku 7.2.

Jak już zostało wspomniane, program eBPF zostaje wykonany dla każdego wywołania systemowego *cap_capable*. Należy określić sposób w jaki odfiltrowane zostaną tylko te wywołania, które dotyczą procesów kontenera – wyamaga to identyfikatora procesu kontenera. Zostały rozważone dwa poniższe warianty.



Rysunek 7.2: Wykorzystanie narzędzia BCC w analizatorze

Filtrowanie na poziomie programu eBPF

Identyfikator procesu kontenera zostaje przekazany do programu eBPF przed komplikacją poprzez dyrektywę `#define`. Dzięki temu, program w trakcie iteracji w góre drzewa procesów może wykryć czy wywołanie dotyczyło kontenera i tylko w takim przypadku wyemitować zdarzenie. W tym wariantie identyfikator procesu kontenera musi być znany jeszcze przed rozpoczęciem śledzenia. Oznacza to pewien okres czasu, w którym kontener już działa, a jego wykorzystanie uprawnień nie jest śledzone.

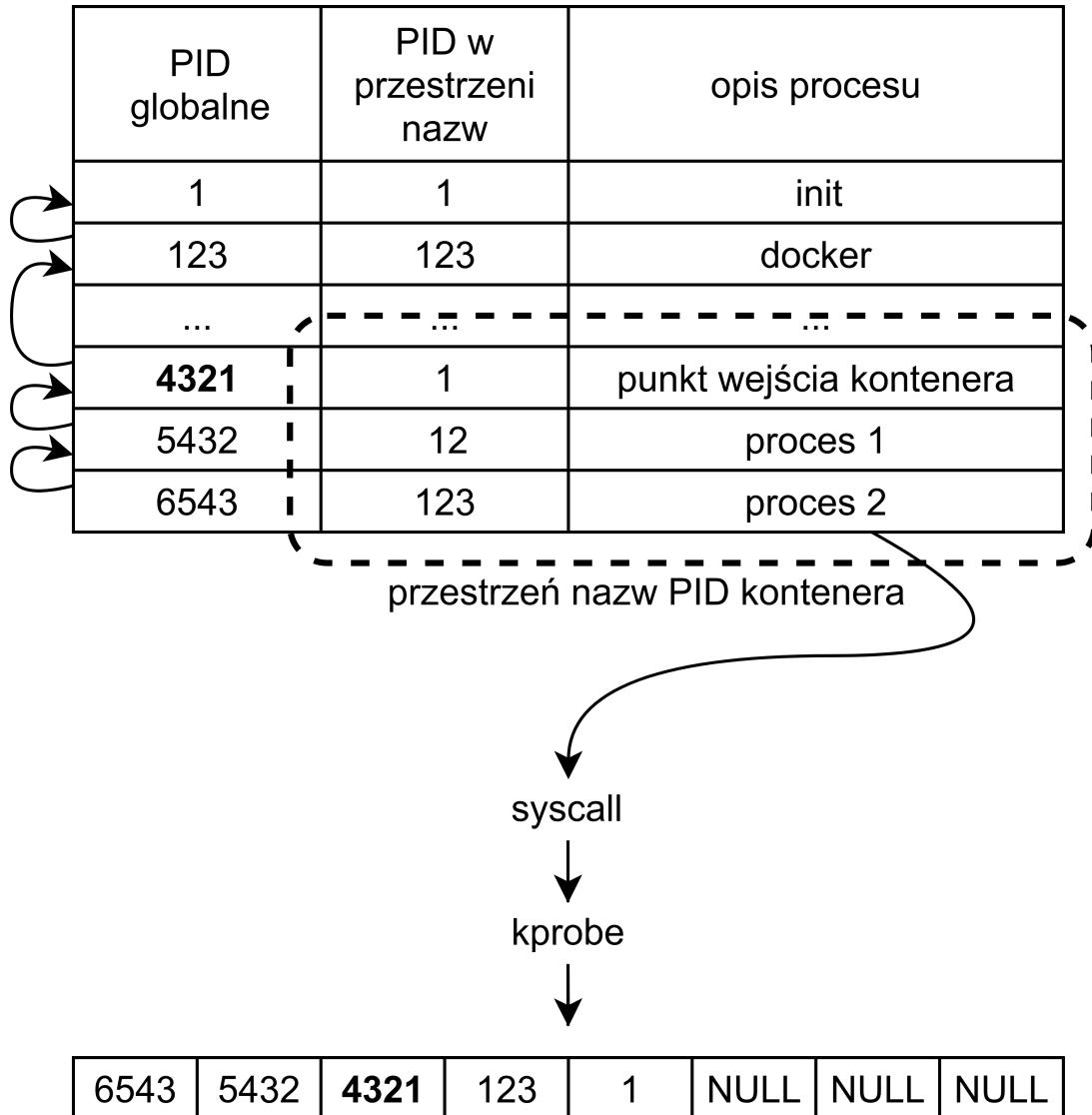
Filtrowanie na poziomie analizy zdarzeń

Każde wywołanie systemowe `capable` powoduje wyemitowanie zdarzenia. W celu filtrowania na poziomie analizy zdarzeń, zdarzenie musi również zawierać całą gałąź drzewa procesów aż do procesu inicjującego (identyfikator procesu = 1). Gałąź drzewa procesów zapisywana jest w postaci tablicy identyfikatorów procesów, gdzie każdy element oznacza identyfikator procesu, a elementy ułożone są w kolejności od dołu gałęzi drzewa. Należy przypomnieć, iż z racji ograniczeń programów eBPF, cała gałąź drzewa procesów może nie zostać zapisana w tablicy. Rysunek 7.3 obrazuje metodę powstawania tablicy.

Wybór wariantu

Wybór pomiędzy dwoma wariantami opierał się na kompromisie pomiędzy szybkością analizy, a utratą początkowych wywołań systemowych kontenera. Warto zaznaczyć, że drugi wariant również może utracić część wywołań z powodu zbyt dużej ich ilości. Jądro może nie uruchomić programu eBPF dla zbyt szybko pojawiających się wywołań systemowych lub bufor `perf_output` może się zapełnić prowadząc do utraty zdarzeń. Dotyczy to również wariantu pierwszego, jednak w mniejszym stopniu, gdyż ten wariant programu eBPF produkuje mniejszą ilość zdarzeń.

Ostateczna implementacja analizatora wykorzystuje filtrowanie na poziomie analizy zda-



Rysunek 7.3: Schemat powstawania tablicy gałęzi drzewa procesów

rzeń. Pierwsze momenty od uruchomienia kontenera polegają najczęściej na inicjalizacji wewnętrznych usług, co w dużej mierze prowadzi do wywołań systemowych. Informacja o uprawnieniach wykorzystywanych w trakcie inicjalizacji jest według autora relatywnie ważniejsza od możliwej utraty informacji z późniejszych wywołań systemowych.

Wywołanie zwrotne odczytujące zdarzenia z bufora przekształca je w obiekt klasy *CapabilityEvent*, a następnie dodaje do kolejki zdarzeń typu *Queue* z modułu *queue*. Klasa *Queue* implementuje wymagane zabezpieczenia synchronizacyjne dzięki czemu może być użyta w aplikacjach wielowątkowych. Ponadto, przepływ elementów odbywa się w porządku FIFO (ang. first-in, first-out).

7.3.4 Analiza wywołań systemowych

Analiza wywołań systemowych również odbywa się w osobnym wątku, który jest uruchamiany poprzez obiekt klasy *CapabilitiesAnalyzer*. Obiekt tej klasy wykorzystuje konfigurację odczytaną z obiektu klasy *Container* w celu zdefiniowania zbioru uprawnień udzielonych kontenerowi. Obiekt pasywnie czeka na zdarzenia pojawiające się w kolejce. Dla każdego ze zdarzeń sprawdza jego gałąź identyfikatorów procesów. Jeśli zdarzenie dotyczy procesu wewnętrz kontenera to w zależności od tego, czy uprawnienie zostało udzielone przez jądro czy nie, dodaje je do odpowiedniego zbioru,

7.3.5 Zwolnienie zasobów i zakończenie programu

Program analizatora kończy się w momencie zakończenia pracy kontenera bądź wymuszenia zakończenia przez użytkownika. Jeśli to użytkownik wymusił zakończenie programu kontener zostaje zastopowany. Główny wątek analizatora następnie wysyła sygnał do wątków wewnętrz obiektów klas *CapabilitiesTracer* i *CapabilitiesAnalyzer* aby zakończyły działanie. Obiekt klasy *CapabilitiesTracer* kończy śledzenie wywołań systemowych natychmiastowo, zaś obiekt klasy *CapabilitiesAnalyzer* najpierw odczytuje i analizuje wszystkie zdarzenia pozostałe w kolejce, a następnie wyświetla raport dotyczący uprawnień.

Raport bazuje na opcjach wpływających na uprawnienia, przekazanych do kontenera w trakcie uruchomienia analizatora:

- *--cap-add*
- *--cap-drop*
- *--privileged*

oraz na uprawnieniach, które procesy kontenera próbowaly użyć w trakcie działania – wywołania systemowe *cap_capable*. Raport składa się z 4 sekcji, przedstawiających zbiory uprawnień:

- ”declared” – przypisane do kontenera uprawnienia
- ”granted” – przypisane do kontenera uprawnienia, z których skorzystał w trakcie analizy
- ”declared but not granted” – przypisane do kontenera uprawnienia, z których nie skorzystał w trakcie analizy
- ”not granted” – uprawnienia, z których kontener chciał skorzystać w trakcie analizy, ale nie miał ich przypisanych

Głównym celem analizy jest zbiór uprawnień "declared but not granted", który obrazuje zjawisko udzielenia nadmiarowych uprawnień kontenerowi (ang. over permissioning). Oczywiście, analiza była dokonywana w trakcie działania kontenera, co oznacza, że najprawdopodobniej nie wszystkie ścieżki wykonania kodu zostały wykorzystane. Jednakże, ten zbiór uprawnień zawiera potencjalnych kandydatów do użycia z opcją `--cap-drop`. Ciekawy również jest ostatni zbiór uprawnień – "not granted" – zawarte w nim uprawnienia próbowały zostać wykorzystane przez kontener jednak nie posiadał on do tego zgody.

7.4. Wyniki przykładowej analizy

Analizator został uruchomiony dla obrazu `postgres:latest`, a kontener poza domyślnymi uprawnieniami otrzymał dodatkowe uprawnienie `SYS_TIME` oraz zostało mu odebrane uprawnienie `KILL`:

```
python3.7 main.py --cap-add SYS_TIME --cap-drop KILL postgres:latest
```

Zwrócony został poniższy raport:

Container declared capabilities:

```
AUDIT_WRITE  
CHOWN  
DAC_OVERRIDE  
FOWNER  
FSETID  
MKNOD  
NET_BIND_SERVICE  
NET_RAW  
SETFCAP  
SETGID  
SETPCAP  
SETUID  
SYS_CHROOT  
SYS_TIME
```

Container granted capabilities:

```
CHOWN  
DAC_OVERRIDE  
FOWNER  
FSETID
```

```
MKNOD  
SETGID  
SETPCAP  
SETUID
```

Container declared but not granted capabilities (over permissioning):

```
AUDIT_WRITE  
NET_BIND_SERVICE  
NET_RAW  
SETFCAP  
SYS_CHROOT  
SYS_TIME
```

Container not granted capabilities (under permissioning):

```
SYS_ADMIN
```

W celu poprawy bezpieczeństwa warto byłoby przeanalizować użycie przez kontener następujących uprawnień: *AUDIT_WRITE*, *NET_BIND_SERVICE*, *NET_RAW*, *SETFCAP*, *SYS_CHROOT*, *SYS_TIME* i w miarę możliwości odebrać mu te, które są przez niego niewykorzystywane.

8. Zakończenie

8.1. Zebrane doświadczenia

W trakcie realizacji pracy autor zdobył liczne doświadczenia wynikające z analizy, projektowania i implementacji programu. Dogłębna analiza całego ekosystemu Dockera pozwoliła na pozyskanie wiedzy przydatnej nie tylko z kwestii widzenia bezpieczeństwa, ale także ogólnie pojętego, zaawansowanego wykorzystania narzędzia Docker. Ponadto, projektowanie analizatora wymagało spędzenia wielu godzin na analizie kodu samego jądra Linuxa co pozwoliło na zapoznanie się technikami programistycznymi wykorzystanymi w tak zaawansowanym projekcie. Ostatecznie, implementacja programu zakładała połączenie w całość kilku projektów i języków projektowania. Doprowadziło do potrzeby zapoznania się z wieloma dokumentacjami, sięgającymi nawet 30 lat wstecz.

8.2. Podsumowanie

Popularność Dockera, tam samo jak i innych znanych projektów informatycznych, sprawiła, że stał się on częstym celem ataków. Docker pozyskał również wielu użytkowników nieswiadomych zagrożeń wynikających z nieprawidłowego wykorzystania tego narzędzia. Istniejące wsparcie ze strony twórców wszystkich głównych części ekosystemu Dockera umożliwia naprawę błędów i ciągły rozwój zabezpieczeń. Efekty konteneryzacji wybiegają jednak poza zamknięty świat usług Dockera, co powoduje istnienie poważnych wektorów ataku w postaci serwisów zewnętrznych. Jednakże, jak pokazała analiza, większość istniejących luk jest zależna w dużym stopniu od błędu ludzkiego, a kultura szybkiego wdrażania oprogramowania, oparta na filozofii DevOps, jeszcze bardziej uwypukla te wady.

9. Bibliografia

- [BANE 2019] bane Repository Contributors, *bane - Custom & better AppArmor profile generator for Docker containers.*, 2019, URL: <https://github.com/genuine-tools/bane> [dostęp 21.09.2019].
- [BANY 2015] Banyan Team, *Over 30% of Official Images in Docker Hub Contain High Priority Security Vulnerabilities*, 2015, URL: <https://www.banyansecurity.io/blog/analyzing-docker-hub/> [dostęp 17.09.2019].
- [BCC 2019] BCC contributors, *BCC Readme*, 2019, URL: <https://github.com/iovisor/bcc/README.md> [dostęp 28.09.2019].
- [BEZE 2016] Cor-Paul Bezemer i in., *How is Performance Addressed in DevOps? A Survey on Industrial Practices*, 2016, URL: <https://arxiv.org/pdf/1808.06915.pdf> [dostęp 19.09.2019].
- [CAPP 2008] Justin Cappos i in., *A Look In the Mirror: Attacks on Package Managers*, 2008, URL: https://ssl.engineering.nyu.edu/papers/cappos_mirror_ccs_08.pdf [dostęp 15.09.2019].
- [CENT 2018] Center for Internet Security, *Docker Benchmark*, 2018, URL: <https://learn.cisecurity.org/benchmarks> [dostęp 18.09.2019].
- [CHOI 2017] Sung Choi i in., *Trustworthy Design Architecture: Cyber-Physical System*, 2017, URL: <https://www.osti.gov/servlets/purl/1431786> [dostęp 20.09.2019].
- [COLE 2016] Mike Coleman, *Containers are not VMs*, 2016, URL: <https://blog.docker.com/2016/03/containers-are-not-vms/> [dostęp 17.09.2019].
- [CORB 2017] Jonathan Corbet, *Notes from a container*, 2017, URL: <https://lwn.net/Articles/256389/> [dostęp 15.09.2019].
- [DOCK 2019a] Docker Bench for Security contributors, *Docker Bench for Security*, 2019, URL: <https://github.com/docker/docker-bench-security> [dostęp 28.09.2019].
- [DOCK 2019b] Docker Inc., *Docker Hub*, 2019, URL: <https://hub.docker.com/search/?q=&type=image> [dostęp 17.09.2019].
- [DOCK 2019c] Docker Inc., *Docker Machine Overview*, 2019, URL: <https://docs.docker.com/machine/overview/> [dostęp 21.09.2019].
- [DOCK 2019d] Docker Inc., *Docker run reference*, 2019, URL: <https://docs.docker.com/engine/reference/run/> [dostęp 15.09.2019].

- [DOCK 2019e] Docker Inc., *Docker SDK for Python*, 2019, URL: <https://docker-py.readthedocs.io/en/stable/client.html> [dostęp 29.09.2019].
- [DOCK 2019f] Docker Inc., *Docker Technology Partner Program Guide*, 2019, URL: <https://www.docker.com/sites/default/files/d8/2019-07/Docker-Technology-Partner-Program-Guide-July-2019.pdf> [dostęp 15.09.2019].
- [DOCK 2019g] Docker Inc., *Dockerfile reference*, 2019, URL: <https://docs.docker.com/engine/reference/builder/> [dostęp 15.09.2019].
- [DOCK 2019h] Docker Inc., *Get started with Docker Desktop for Mac*, 2019, URL: <https://docs.docker.com/docker-for-mac/> [dostęp 21.09.2019].
- [DOCK 2019i] Docker Inc., *Get started with Docker for Windows*, 2019, URL: <https://docs.docker.com/docker-for-windows/> [dostęp 21.09.2019].
- [DOCK 2019j] Docker Inc., *Official Images on Docker Hub*, 2019, URL: https://docs.docker.com/docker-hub/official_images/ [dostęp 15.09.2019].
- [DOCK 2019k] Docker Inc., *Scan images for vulnerabilities*, 2019, URL: <https://docs.docker.com/v17.12/datacenter/dtr/2.4/guides/user/manage-images/scan-images-for-vulnerabilities/> [dostęp 20.09.2019].
- [DUNC 2017] Bob Duncan, Andreas Happe i Alfred Bratterud, *Cloud Cyber Security: Finding an Effective Approach with Unikernels*, 2017, URL: <https://www.intechopen.com/books/advances-in-security-in-computing-and-communications/cloud-cyber-security-finding-an-effective-approach-with-unikernels> [dostęp 26.09.2019].
- [FORR 2017] Forrester Consulting, *Containers: Real Adoption And Use Cases In 2017*, 2017, URL: https://downloads.dell.com/solutions/cloud-solution-resources/Forrester%20Containers%20Real%20Adoption%20Dell%20EMC%20White%20Paper_SPi_R6.pdf [dostęp 17.09.2019].
- [GART 2019] Gartner Inc., *Gartner Says Worldwide IaaS Public Cloud Services Market Grew 31.3% in 2018*, 2019, URL: <https://www.gartner.com/en/newsroom/press-releases/2019-07-29-gartner-says-worldwide-iaas-public-cloud-services-market-grew-31point3-percent-in-2018> [dostęp 17.09.2019].
- [GREG 2019] Brendan Gregg, *Linux Extended BPF (eBPF) Tracing Tools*, 2019, URL: <http://www.brendangregg.com/eBpf.html#eBPF> [dostęp 28.09.2019].

- [JACK 2018] Joab Jackson, *Linux Technology for the New Year: eBPF*, 2018, URL: <https://thenewstack.io/linux-technology-for-the-new-year-ebpf/> [dostęp 28.09.2019].
- [KAMP 2010] Poul-Henning Kamp i Robert N. M. Watson, *Jails: Confining the omnipotent root.*, 2010, URL: <http://phk.freebsd.dk/pubs/sane2000-jail.pdf> [dostęp 14.09.2019].
- [KUEN 2017] Simon Kuenzer i in., *Unikernels Everywhere: The Case for Elastic CDNs*, 2017, URL: <http://delivery.acm.org/10.1145/3060000/3050757/p15-Kuenzer.pdf> [dostęp 14.09.2019].
- [LINU 2019] Linux manpages contributors, *Namespaces - overview of Linux namespaces*, 2019, URL: <http://man7.org/linux/man-pages/man7/namespaces.7.html> [dostęp 15.09.2019].
- [LOMB 2010] Flavo Lombardi i Roberto Di Pietro, *Secure virtualization for cloud computing*, 2010, URL: <http://www.csc.villanova.edu/~nadi/csc8580/S11/AnushaUppalapati.pdf> [dostęp 14.09.2019].
- [MADH 2013] Anil Madhavapeddy i in., *Unikernels: Library Operating Systems for the Cloud*, 2013, URL: <http://mort.io/publications/pdf/asplos13-unikernels.pdf> [dostęp 14.09.2019].
- [MADH 2015] Anil Madhavapeddy i in., *Jitsu: Just-In-Time Summoning of Unikernels*, 2015, URL: <https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-madhavapeddy.pdf> [dostęp 20.09.2019].
- [MCCA 1993] Steven McCanne i Van Jacobson, *The BSD Packet Filter: A New Architecture for User-level Packet Capture*, 1993, URL: <https://www.usenix.org/legacy/publications/library/proceedings/sd93/mccanne.pdf> [dostęp 28.09.2019].
- [MILL 2012] Adam Miller i Lei Chen, *Securing Your Containers*, 2012, URL: <http://worldcomp-proceedings.com/proc/p2012/SAM9702.pdf> [dostęp 15.09.2019].
- [MITR 2019] MITRE Corporation, *Docker Vulnerability Statistics*, 2019, URL: <https://www.cvedetails.com/product/28125/Docker-Docker.html> [dostęp 18.09.2019].
- [MOBY 2018] Moby contributors, *Docker Image Specification v1.2.0*, 2018, URL: <https://github.com/moby/moby/blob/master/image/spec/v1.2.md> [dostęp 15.09.2019].

- [MOBY 2019] Moby contributors, *Docker default AppArmor profile*, 2019, URL: <https://github.com/moby/moby/blob/master/contrib/apparmor/main.go> [dostęp 15.09.2019].
- [MORA 2015] Roberto Morabito, Jimmy Kjällman i Miika Komu, *Hypervisors vs. Lightweight Virtualization: A Performance Comparison*, 2015, URL: https://www.researchgate.net/profile/Roberto_Morabito/publication/273756984_Hypervisors_vs_Lightweight_Virtualization_A_Performance_Comparison/links/550a83660cf26198a63afb10/Hypervisors-vs-Lightweight-Virtualization-A-Performance-Comparison.pdf [dostęp 14.09.2019].
- [NATI 2012] National Institute of Standards and Technology, U.S. Department of Commerce, *Guide for Conducting Risk Assessments*, 2012, URL: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-30r1.pdf> [dostęp 20.09.2019].
- [PETA 2014] Jérôme Petazzoni, *Why you don't need to run SSHd in your Docker containers*, 2014, URL: <https://blog.docker.com/2014/06/why-you-dont-need-to-run-sshd-in-docker/> [dostęp 16.09.2019].
- [PURR 2015] John Purrier, *What is Rocket and How It's Different Than Docker*, 2015, URL: <https://www.ctl.io/developers/blog/post/what-is-rocket-and-how-its-different-than-docker/> [dostęp 15.09.2019].
- [PYTH 2019a] Python Software Foundation, *What is Python? Executive Summary*, 2019, URL: <https://www.python.org/doc/essays/blurb/> [dostęp 29.09.2019].
- [PYTH 2019b] Python Software Foundation, *What's New In Python 3.7*, 2019, URL: <https://docs.python.org/3/whatsnew/3.7.html> [dostęp 29.09.2019].
- [RED 2016] Red Hat Atomic Host Documentation Team, *Red Hat Enterprise Linux Atomic Host7 Container Security Guide*, 2016, URL: http://ftp.newtek.com/pub/reads/Atomic_Host-7-Container_Security_Guide-en-US.pdf [dostęp 21.09.2019].
- [RED 2019] Red Hat Inc., *Red Hat CVE Database*, 2019, URL: <https://access.redhat.com/security/security-updates/#/cve> [dostęp 19.09.2019].
- [ROSL 2018] Jonas Rosland, *Container OS Comparison*, 2018, URL: <https://blog.codeship.com/container-os-comparison/> [dostęp 15.09.2019].

- [SAMU 2010] Justin Samuel i in., *Survivable Key Compromise in Software Update Systems*, 2010, URL: <https://www.freehaven.net/~arma/tuf-ccs2010.pdf> [dostęp 15.09.2019].
- [SCHM 2017] Michael Schmid, *Docker on Mac Performance: Docker Machine vs Docker for Mac*, 2017, URL: <https://stories.amazee.io/docker-on-mac-performance-docker-machine-vs-docker-for-mac-4c64c0afdf99> [dostęp 21.09.2019].
- [SCHR 2011] Z.C. Schreuders, T. McGill. i C. Payne, *Empowering End Users to Confine Their Own Applications: The Results of a Usability Study Comparing SELinux, AppArmor and FBAC-LSM*, 2011, URL: http://eprints.leedsbeckett.ac.uk/546/1/empowering_end_users.pdf [dostęp 21.09.2019].
- [SHU 2017] Rui Shu, Xiaohui Gu i William Enck, *A Study of Security Vulnerabilities on Docker Hub*, 2017, URL: <https://www.enck.org/pubs/shu-codasp17.pdf> [dostęp 17.09.2019].
- [SULE 2016] Aater Suleman, *8 Proven Real-World Ways to Use Docker*, 2016, URL: <https://www.airpair.com/docker/posts/8-proven-real-world-ways-to-use-docker> [dostęp 16.09.2019].
- [XAVI 2013] Miguel G. Xavier i in., *Performance Evaluation of Container-based Virtualization for High Performance Computing Environments*, 2013, URL: https://www.researchgate.net/profile/Fabio_Rossi10/publication/261266481_Performance_Evaluation_of_Container-Based_Virtualization_for_High_Performance_Computing_Environments/links/5500c3200cf2aee14b581172/Performance-Evaluation-of-Container-Based-Virtualization-for-High-Performance-Computing-Environments.pdf [dostęp 14.09.2019].
- [XEN 2019] Xen Project Wiki contributors, *Unikernel*, 2019, URL: <https://wiki.xenproject.org/wiki/Unikernels> [dostęp 14.09.2019].
- [ZHEN 2015] Chao Zheng i Douglas Thain, *Integrating Containers into Workflows: A Case Study Using Makeflow, Work Queue, and Docker*, 2015, URL: https://cri-lab.net/wp-content/uploads/2018/03/Docker_ecosystem---_Vulnerability_analysis.pdf [dostęp 14.09.2019].

A. Dodatek A

Płyta CD z kompletnym kodem źródłowym analizatora.