DOCKER – WPROWADZENIE DO BEZPIECZEŃSTWA

Kamil Jarosiński

Kamil Jarosiński. Konsultant ds. bezpieczeństwa IT w Securitum. Od ponad siedmiu lat przeprowadza testy penetracyjne oraz prowadzi szkolenia z zakresu bezpieczeństwa aplikacji webowych. Pasjonat bezpieczeństwa systemów IT, lubiący dzielić się wiedzą.

Prelegent na konferencjach branżowych, m.in. MSHP, uczestnik programów *bug bounty*.

Podziękowania

Chciałbym serdecznie podziękować mojej kochanej Żonie, dzięki której jestem tu, gdzie jestem. Za Jej wsparcie, możliwość rozwoju, wiarę i opiekę nad całą naszą rodziną.

WSTĘP

Zagadnienie wirtualizacji sięga lat 60. XX wieku, kiedy to IBM stworzyło jej podwaliny w ramach systemu CP/CMS[1]. Pozwalał on na logiczne podzielenie zasobów ogromnych komputerów Mainframe[2]. Mechanizm wirtualizacji ułatwiał efektywniejsze wykorzystanie możliwości oferowanych przez sprzęt. Taka architektura pozwoliła zminimalizować konieczność uruchamiania kolejnych urządzeń, przyczyniając się do redukcji ogólnego zużycia energii. Sama separacja środowisk również była ogromnym plusem, ponieważ błędy związane z jedną wirtualną maszyną nie powodowały problemów z inną instancją.

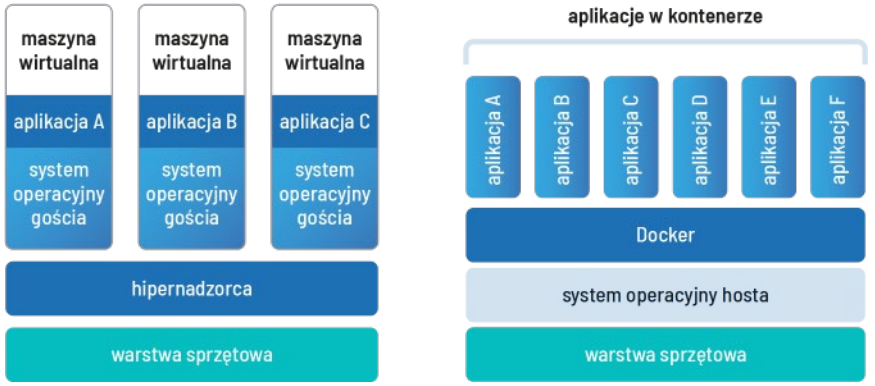
Wirtualizację można podzielić na dwa typy:

1. **pełną wirtualizację** – emuluje cały sprzęt komputerowy; dla maszyny (ang. *virtual machine*, VM) jest nie do odróżnienia, czy została uruchomiona na rzeczywistym sprzęcie czy w wirtualnym środowisku, np. ESXi, hipernadzorca (ang. *hypervisor*) typu 1, VirtualBox, hipernadzorca typu 2*;
2. **parawirtualizację** – działanie maszyn wirtualnych jest zapewniane przez specjalne API hipernadzorcy, które oferuje dostęp do sprzętu dla wywołań wykonywanych przez system operacyjny gościa, nazywanych *hypercalls*, w przypadku instrukcji wrażliwych, przykładowo zarządzanie pamięcią, obsługa przerwań, zarządzanie procesorem (np. Xen, IBM LPAR).

Docker to narzędzie ułatwiające konteneryzację, które pojawiło się w 2013 roku dzięki inicjatywie Solomona Hykesa. Stworzył on wewnętrzny projekt w firmie dotCloud (obecnie Docker, Inc.)[3]. Nadal jest bardzo szeroko stosowany ze względu na liczne zalety w porównaniu z typowymi mechanizmami wirtualizacji. **Dzięki lekkiej architekturze pozwala na efektywniejsze wykorzystanie zasobów systemowych, umożliwiając uruchamianie wielu aplikacji na tej samej infrastrukturze – bez znacznego obciążenia.** Wspomniana lekkość jest związana z tym, że – w odróżnieniu od typowych metod wirtualizacji – nie dochodzi tutaj do symulowania sprzętu.

Docker, **działający jako usługa** (demon, ang. *daemon*) **na hoście**, na którym został uruchomiony, **używa mechanizmów wbudowanych w jądro systemu Linux**, takich jak `cgroups/namespaces`[4], dzięki czemu wszystko opiera się na jednym jądrze.

* Zob. porównanie obu typów hipernadzorcy – np. Amazon Web Services, *What's the Difference Between Type 1 and Type 2 Hypervisors?*, https://aws.amazon.com/compare/the-difference-between-type-1-and-type-2-hypervisors/



Rysunek 1. Porównanie wirtualizacji i konteneryzacji

Na rysunku 1[5] zaprezentowane jest porównanie `wirtualizacji` (po lewej: pełnej), gdzie widoczna jest warstwa sprzętowa (ang. *infrastructure*), na której zainstalowany jest hipernadzorca zarządzający całym procesem, a powyżej niego tworzone są odseparowane maszyny wirtualne gości mające przydzielone zasoby, w ramach których działają aplikacje.

`Docker` (po prawej) jest uruchamiany w ramach infrastruktury, na której musi być zainstalowany system operacyjny hosta (np. Linux). Następnie Docker, wykorzystując jądro hosta, tworzy odseparowane środowiska dla uruchamianych aplikacji. Ze względu na brak konieczności wirtualizacji komponentów systemu gościa można zmniejszyć ilość zużywanych zasobów, aby uruchomić tę samą aplikację.

Docker pozwala na uruchomienie w kontenerze zarówno aplikacji działających na systemie Linux, jak i w ramach systemu Windows. Mimo że w tym rozdziale skupiłem się wyłącznie na kontenerach opartych na Linuksie, opisywane elementy odnoszą się również do systemu firmy Microsoft.

Postaram się tu przedstawić najważniejsze aspekty i najczęściej popełniane błędy, z jakimi miałem okazję się zetknąć jako pentester. Czytelnik tego rozdziału na pewno powinien dobrze znać podstawy Linuksa, żeby podążyć za opisywanymi zagadnieniami.

DOCKERFILE, OBRAZY, REPOZYTORIA I KONTENERY

Rozpoczynając pracę z Dockerem, należy zapoznać się ze sposobem, w jaki uruchamiane są kontenery, z czego budowane są obrazy oraz jak można je przechowywać.

W celu łatwiejszego zrozumienia tematu będę w tym podrozdziale stosował porównania kulinarne, które pomogą wskazać analogię pomiędzy drogą niezbędną do wyprodukowania mrożonej pizzy: od zakupu składników do przygotowania pizzy zgodnie z przepisem, a podstawowymi cegiełkami Dockera.

Dockerfile

Zacznijmy od pliku Dockerfile. Można w nim znaleźć **polecenia, z których będzie budowany obraz**. W tej analogii jest on przepisem zawierającym instrukcję: co i w jakiej

kolejności ma być wykonane, aby wyprodukować mrożoną pizzę. Na listingu 1 znajduje się przykładowy przepis.

Najpierw jednak mała dygresja: **istnieje jeszcze jeden sposób tworzenia obrazów** (który nie wymaga przygotowania pliku Dockerfile), polegający na **uruchomieniu kontenera na podstawie obrazu bazowego i wprowadzeniu stosownych zmian** (np. przeniesienie plików, przygotowanie konfiguracji, doinstalowanie niezbędnego oprogramowania itp.). Następnie tak przygotowany kontener może zostać zapisany do postaci obrazu. Ze względu na brak możliwości łatwego zarządzania zmianami i rozwijaniem raz przygotowanego obrazu nie jest to metoda często stosowana, dlatego zrezygnowałem z dokładniejszego jej opisania.

Listing 1. Przykładowy plik Dockerfile

```
FROM alpine:3.16.0
RUN apk update && apk add --no-cache \
    bash \
    curl
RUN adduser -D limited -s /bin/bash
USER limited:limited
WORKDIR /app
COPY ./app-on-host /app
```

Pierwsze polecenie `FROM alpine:3.16.0` odpowiada za **wskazanie obrazu bazowego**, który będzie podstawą rozwijaną przez kolejne dodawane warstwy (o warstwach piszę szczegółowo poniżej). W przykładzie użyto bardzo popularnej dystrybucji Linux Alpine, która jest zoptymalizowana pod kątem wykorzystania zasobów. Gdy jest to uzasadnione, można również wybrać bardziej wyspecjalizowany obraz, jak nginx (oparty w niektórych wersjach na dystrybucji Debiana). Podczas tworzenia własnych plików Dockerfile trzeba zdecydować, od jakiego obrazu bazowego rozpocznie się rozwój. Obrazy bazowe domyślnie są pobierane z repozytorium obrazów Docker Hub[6], o którym więcej szczegółów można znaleźć w dalszej części tekstu.

WAŻNE

Każdy obraz może mieć przypisany dowolny tag (w omawianym przykładzie: `3.16.0`). Bardzo często istnieje **tag** `latest` – i należy na niego szczególnie uważać, ponieważ jego nazwa może być bardzo myląca. **Nie musi odnosić się do najnowszej wersji oprogramowania.** W przypadku wykonania przez właściciela obrazu aktualizacji obrazu do nowszej wersji, po oznaczeniu jej tagiem z wersją może się zdarzyć, że tag `latest` nie zostanie zaktualizowany, a więc nie będzie wskazywał najnowszej wersji obrazu. Dlatego zaleca się używanie kolejnych wersji obrazów, tak jak pokazano w przykładzie.

Kolejna linia: `RUN apk update && apk add --no-cache bash curl`, **odpowiada za wykonanie polecenia w kontenerze podczas jego budowania**. W tym przypadku zostanie uruchomiona aktualizacja listy menedżera pakietów systemu Alpine oraz, po prawidłowym zakończeniu polecenia, zostanie wykonana instalacja oprogramowania Bash i curl.

Następne dwa polecenia: `RUN adduser -D limited -s /bin/bash` oraz `USER limited:limited`, **dodają nowego użytkownika i przełączają kontekst na jego konto**. Od tego momentu wszystkie kolejne polecenia będą wykonywane przez tego użytkownika.

Polecenia: `WORKDIR /app` oraz `COPY ./app-on-host /app` **ustawiają katalog roboczy** (jeżeli nie istnieje, zostanie utworzony) oraz kopiują pliki aplikacji z katalogu hosta `./app-on-host` (kopiowany katalog hosta musi znajdować się w tej samej ścieżce co plik Dockerfile) do katalogu `/app` obrazu. Pierwsze polecenie można traktować tak jak polecenie `cd` w systemie Linux, pozwalające na przechodzenie pomiędzy katalogami.

WAŻNE

Dodawanie plików do obrazu zaleca się wykonywać za pomocą polecenia COPY. Polecenie ADD, pomimo bardzo podobnego zachowania, zawiera dodatkowe funkcje, które nie są oczywiste[7], jak rozpakowywanie plików .tar czy wsparcie dla adresów URL.

Pliki Dockerfile mogą być mniej lub bardziej rozbudowane, w związku z tym warto trzymać się najlepszych praktyk[8].

DOBRE PRAKTYKI: PLIK DOCKERFILE

W pracy z plikiem Dockerfile warto korzystać z zasad, które ułatwią tworzenie, modyfikację oraz rozszerzenie już istniejących obrazów:

- kontener powinien wymagać absolutnego **minimum konfiguracji**, w każdej chwili powinna być możliwość zatrzymania go, zniszczenia, ponownego zbudowania i wymiany;
- **pliki zbędne** podczas budowania obrazu można **wykluczyć** za pomocą pliku .dockerignore (działa podobnie jak .gitignore), wskazane pliki z hosta nie będą przenoszone do obrazu;
- nie należy instalować zbędnych pakietów;
- **każdy kontener** powinien realizować **jedną funkcję** (np. być bazą danych PostgreSQL);
- należy **minimalizować liczbę poleceń** w pliku Dockerfile;
- warto stosować **formatowanie wielolinijkowych poleceń** poprzez rozbicie wykonywanych poleceń na kilka linii oraz stosować sortowanie alfabetyczne instalowanych pakietów, co poprawi czytelność Dockerfile;
- wskazane jest **używanie** polecenia COPY **zamiast** ADD;
- zasadne jest stosowanie polecenia USER, które pozwala na określenie, jaki użytkownik w kontenerze będzie wykorzystany do uruchomienia procesów, **ograniczając poziom uprawnień** domyślnego użytkownika root.

Budowanie obrazu

Mając już i rozumiejąc „przepis", można przejść do „przygotowania pizzy i zamrożenia jej", czyli zbudowania obrazu. Będąc w katalogu, w którym znajduje się plik Dockerfile i wszystkie niezbędne pliki aplikacji, można wykonać polecenie z listingu 2, którego zadaniem jest **zbudowanie obrazu na podstawie poleceń znajdujących się w obecnym katalogu w pliku Dockerfile** oraz nadanie obrazowi nazwy `securitum` z tagiem app.

Listing 2. Budowanie obrazu dockera na podstawie Dockerfile

```
$ docker build . -t securitum:app

Sending build context to Docker daemon  18.94kB
Step 1/6 : FROM alpine:3.16.0
 ---> e66264b98777
Step 2/6 : RUN apk update && apk add --no-cache bash curl
```

```
 ---> Using cache
 ---> fad0f3c8ed2a
Step 3/6 : RUN adduser -D limited -s /bin/bash
 ---> Using cache
 ---> 8f3e9b2ee664
Step 4/6 : USER limited:limited
 ---> Using cache
 ---> 6c2e041055fc
Step 5/6 : WORKDIR /app
 ---> Using cache
 ---> e0888ccbb442
Step 6/6 : COPY ./app-on-host /app
 ---> Using cache
 ---> eda6fc35a870
Successfully built eda6fc35a870
Successfully tagged securitum:app
```

Polecenia wskazane w „przepisie" zostaną wykonane, tworząc nowe warstwy (ang. *layers*), osobne dla każdego z poleceń. **Każda kolejna warstwa nakładana jest na wcześniejsze**, czyli przykładowo wykonanie polecenia instalacji dodatkowego oprogramowania spowoduje stworzenie zmian w katalogu przechowującym programy – i właśnie ta różnica zostanie zapisana na tej warstwie (`fad0f3c8ed2a`). Nałożenie kolejnych warstw zawierających zmodyfikowane pliki i ich treść względem poprzedzającej warstwy daje finalnie obraz, do którego miał doprowadzić ten proces.

Dzięki **budowie warstwowej** modyfikacja w dowolnym miejscu „przepisu" (Dockerfile) spowoduje wyłącznie konieczność przeliczenia zmian powyżej tej zmiany, co znacznie przyspiesza wprowadzanie modyfikacji.

Sprawdzenie, z jakich warstw składa się obraz, można zrealizować tak, jak to przedstawiono na listingu 3[9].

Listing 3. Wyświetlanie warstw obrazu `securitum` z tagiem `app`

```
$ docker history securitum:app
IMAGE          CREATED        CREATED BY                                      SIZE
eda6fc35a870   8 hours ago    /bin/sh -c #(nop) COPY
                              dir:d58f590f764b682e1...                        10B
e0888ccbb442   8 hours ago    /bin/sh -c #(nop) WORKDIR/app                   0B
6c2e041055fc   8 hours ago    /bin/sh -c #(nop)
                              USER limited:limited                            0B
8f3e9b2ee664   10 hours ago   /bin/sh -c adduser -D limited -s                4.68kB
fad0f3c8ed2a   10 hours ago   /bin/bash /bin/sh -c apk update
                              && apk add --no-cache curl bash                 6.69MB
e66264b98777   3 weeks ago    /bin/sh -c #(nop) CMD ["/bin/sh"]               0B
<missing>      3 weeks ago    /bin/sh -c #(nop) ADD
                              file:8e81116368669ed3d...                       5.53MB
```

Na listingu 3 widoczne są polecenia, które zostały wykorzystane do zbudowania obrazu. Ostatnie dwa wiersze w tym przypadku są warstwami pochodzącymi z obrazu bazowego.

WAŻNE

Przed uruchomieniem kontenera na podstawie obrazu warto przeanalizować wykonywane polecenia. Wszystkie wywołania komend: `RUN`, `CMD`, `ENTRYPOINT`, powinny być szczegółowo prześledzone. Pozwalają one na:

1. uruchamianie wybranego polecenia podczas budowania obrazu (`RUN`), przykładowo w celu instalacji dodatkowych pakietów;
2. wskazanie polecenia, które ma zostać uruchomione podczas startu kontenera (`CMD`), a które może zostać nadpisane przez osobę uruchamiającą kontener;
3. wskazanie polecenia, które zawsze ma zostać uruchomione podczas startu kontenera (`ENTRYPOINT`), a którego nie można nadpisać. Należy również pamiętać, że pliki znajdujące się już w kontenerze także mogą nieść ze sobą zagrożenie.

Przed przejściem do **uruchomienia pierwszego kontenera** warto zatrzymać się na chwilę i zastanowić – jakie **zagrożenia** mogą się tu znaleźć. Może to być np.:

- **nieaktualne oprogramowanie** zawierające znane podatności,
- **błędy w konfiguracji** używanego oprogramowania,
- **malware** umieszczony w kontenerze lub pobierany w chwili uruchomienia.

Nie powinno się bagatelizować tych zagrożeń, ponieważ według raportu firmy Prevasio[10] (obecnie AlgoSec), w ramach którego przeanalizowano 4 miliony publicznie dostępnych obrazów, 51% miało krytyczne podatności. Wystarczy, że w którymś z plików Dockerfile skorzystano z niebezpiecznego obrazu – i można mieć „intruza" w infrastrukturze, który dodatkowo może atakować znajdujące się w niej kolejne systemy.

Jak w takim razie ograniczyć problemy tego typu?

Należy weryfikować, z jakich warstw składa się obraz, i skanować. Skanowanie pod kątem obecności malware'u należy przeprowadzać za pomocą oprogramowania antywirusowego oraz wykonywać skan Dockerfile lub obrazu za pomocą skanerów takich jak Snyk[11] lub Trivy[12], wykrywających nieaktualne oprogramowanie. Na listingu 4 przedstawiono przykładowy skan wykonany za pomocą skanera Snyk, w zależnościach obrazu widoczna jest lista znalezionych podatności.

Listing 4. Przykładowy skan wykonany za pomocą narzędzia Snyk

```
$ snyk container test nginx:1.23.3-alpine

Testing nginx:1.23.3-alpine...

[...]
✗ Critical severity vulnerability found in curl/libcurl
  Description: Out-of-bounds Write
  Info: https://security.snyk.io/vuln/SNYK-ALPINE317-CURL-5958910
  Introduced through: curl/libcurl@7.87.0-r1, curl/curl@7.87.0-r1
  From: curl/libcurl@7.87.0-r1
  From: curl/curl@7.87.0-r1 > curl/libcurl@7.87.0-r1
  From: curl/curl@7.87.0-r1
  Image layer: 'apk add --no-cache curl ca-certificates'
```

```
  Fixed in: 8.4.0-r0

Organization:      sekurak
Package manager:   apk
Project name:      docker-image|nginx
Docker image:      nginx:1.23.3-alpine
Platform:          linux/amd64
Base image:        nginx:1.23.3-alpine
Licenses:          enabled

Tested 62 dependencies for known issues, found 68 issues.

Base Image           Vulnerabilities  Severity

nginx:1.23.3-alpine  68               2 critical, 15 high, 45 medium, 6 low

[...]
```

Ze względu na budowę warstwową każdego obrazu i obsługę mechanizmu *cache* dla każdej z warstw należy odpowiednio podchodzić do budowania i rozbudowywania pliku Dockerfile, ponieważ zmodyfikowanie wybranej linii w pliku spowoduje konieczność przebudowania wszystkich warstw, poczynając od zmodyfikowanej do ostatniej.

Na listingu 5 przedstawiono przykład stworzenia pliku Dockerfile dla aplikacji Node.js, w którym wprowadzone zostały korzyści płynące z opisanego wcześniej mechanizmu. W pierwszej kolejności przygotowywany jest obraz bazowy, użytkownik zmieniany jest na `node` oraz katalog roboczy ustawiany na `/app`. Następnie do kontenera do katalogu roboczego kopiowane są pliki `package.json` i `package-lock.json`. W kolejnym kroku instalowane są zależności aplikacji wraz z weryfikacją sumy kontrolnej bibliotek. Dopiero po instalacji następuje przekopiowanie plików aplikacji, a przy starcie kontenera, poleceniem `ENTRYPOINT`, zostanie uruchomiona aplikacja.

Listing 5. Przykładowy plik Dockerfile dla aplikacji Node.js

```
FROM node:20.15.0-alpine3.20
USER node
WORKDIR /app
COPY package.json package-lock.json /app
RUN npm ci
COPY ./src /app/src
ENTRYPOINT node /app/src/index.js
```

Przy zastosowaniu przedstawionej konstrukcji modyfikacja kodu aplikacji spowoduje konieczność przebudowania tylko dwóch ostatnich warstw, a wcześniejsze zostaną pobrane z *cache*. Znacznie przyspiesza to budowanie kolejnych wersji kontenera. W przypadku kiedy zostaną wykonane zmiany w liście bibliotek, oczywiście nastąpi przebudowanie wszystkich warstw – począwszy od tej, która odpowiada za przekopiowanie plików `package.json` oraz `package-lock.json`.

Repozytoria

Przejdźmy do kolejnego miejsca na trasie, jaką ma do przebycia „mrożona pizza". Teraz trzeba ją dostarczyć do klientów za pośrednictwem sklepów. Takim sklepem dla obrazów są repozytoria, jednym z publicznych jest wspomniany już Docker Hub.

Można w nim znaleźć bardzo wiele obrazów, które niestety w wielu przypadkach zawierają błędy bezpieczeństwa w używanym oprogramowaniu[13]. Z tego powodu **najlepiej ograniczyć się jedynie do oficjalnych i zweryfikowanych obrazów**, które można w prosty sposób odfiltrować, zaznaczając opcję DOCKER OFFICIAL IMAGE w wyszukiwarce. Jednak nawet korzystając z wysokiej jakości źródła, nie można czuć się zwolnionym z **obowiązku wykonywania analizy i skanowania**, ponieważ zdarzało się, że atakujący uzyskali możliwość wstrzyknięcia niebezpiecznego kodu do popularnego oprogramowania[14].

Duża grupa firm używających Dockera w swoich projektach posiada własne repozytoria obrazów, w których są przechowywane obrazy bazowe, będące punktem startowym do rozbudowy w kolejnych projektach. Ciekawym dodatkowym zabezpieczeniem przed złośliwie zmodyfikowanym obrazem umieszczonym w repozytorium może być **stosowanie podpisów cyfrowych**[15], które powinny być weryfikowane przed wykonaniem istotnych operacji, takich jak np. uruchomienie. Domyślnie ten mechanizm jest wyłączony.

Uruchamianie kontenera

Ostatnim etapem wędrówki „pizzy" jest „podgrzanie", czyli uruchomienie kontenera za pomocą komendy: `$ docker run -d securitum:app`. W przedstawionym przypadku kontener zostanie uruchomiony w trybie działania w tle (przełącznik `-d`), dzięki czemu istnieje możliwość dalszego wykonywania poleceń na poziomie hosta.

Przy uruchamianiu kontenera istnieje możliwość modyfikowania wielu elementów już zdefiniowanych w obrazie, np.:

- użytkownika, w ramach którego mają być uruchamiane procesy w kontenerze (`--user`),
- ustawianie zmiennych środowiskowych (`-e`),
- zamontowanie dodatkowych katalogów z hosta (`--volume`),
- wskazywanie ustawień sieciowych, które mogą się różnić w zależności od wybranego typu sieci (o sieci piszę poniżej).

Same obrazy powinny być konstruowane w taki sposób, aby w przypadku wyłączenia i ponownego uruchomienia kontenera wszystkie elementy (usługi) nie wymagały ingerencji z zewnątrz.

Z punktu widzenia bezpieczeństwa następnym **ważnym mechanizmem podczas uruchamiania kontenera jest tryb uprzywilejowany** (`--privileged`), który powoduje, że urządzenia dostępne na hoście (katalog `/dev`) zostaną umieszczone w kontenerze oraz nadane zostaną wszystkie *capabilities*. **Jest to bardzo niebezpieczna sytuacja**, ponieważ ucieczka z takiego kontenera jest niezwykle prosta[16], np. w sytuacji zamontowania dysku hosta w trybie odczytu i zapisu.

Zastosowanie nieprawidłowej konfiguracji może pozwolić atakującemu, który uzyskał możliwość wykonywania poleceń w ramach kontenera, wykonywać następnie polecenia

na poziomie hosta, z uprawnieniami demona Dockera (najczęściej root). Ze względu na powyższe **nie zaleca się używania trybu uprzywilejowanego**.

Jednak jeżeli dostęp do konkretnego urządzenia jest niezbędny, należy udostępnić je za pomocą flagi urządzenia (`--device`), choć i w tym przypadku również należy zachować rozsądek i rozważyć, które urządzenie jest udostępniane, oraz wskazać dokładne *capabilities*, jakie powinien mieć kontener (`--cap-drop=ALL --cap-add=CHOWN`).

***Capabilities* w Dockerze pozwalają nadawać lub odbierać zdefiniowane zakresy uprawnień administracyjnych**. W domyślnej konfiguracji użytkownik root ma tylko kilkanaście uprawnień[17]. Na listingu 6 znajduje się przykład uruchomienia kontenera z obrazem Ubuntu, gdzie odebrano wszystkie *capabilities*. Pomimo używania konta administracyjnego nie ma możliwości wykonania aktualizacji pakietów, ponieważ niezbędne *capabilities* nie zostały przydzielone*. Po zakończeniu działania kontenera zostanie on usunięty, ponieważ zastosowano flagę `--rm`.

Listing 6. Uruchomienie kontenera z odebranymi wszystkimi *capabilities*

```
$ docker run --rm --cap-drop=ALL ubuntu bash -c 'id && apt-get update'
uid=0(root) gid=0(root) groups=0(root)
E: setgroups 65534 failed - setgroups (1: Operation not permitted)
E: setegid 65534 failed - setegid (1: Operation not permitted)
E: seteuid 42 failed - seteuid (1: Operation not permitted)
E: setgroups 0 failed - setgroups (1: Operation not permitted)
rm: cannot remove '/var/cache/apt/archives/partial/*.deb': Permission denied
```

JAK DZIAŁA SIEĆ?

Istotnym elementem konfiguracji obrazu jest ustawienie trybu działania sieci[18]. Mamy tu wiele możliwości do wyboru. Zrozumienie detali związanych z poszczególnymi opcjami może okazać się kluczowe dla bezpieczeństwa tworzonego rozwiązania (sam dostęp kontenera do sieci to zbyt mało, jeśli ma być bezpiecznie).

Docker wspiera poniższe tryby działania sieci:

- **none** – pozwala wyłączyć obsługę sieci dla wybranego kontenera (pętla zwrotna nadal będzie działać prawidłowo);
- **bridge** – domyślny tryb działania sieci, jeżeli dany kontener nie posiada wskazanej sieci, zostanie podłączony do domyślnego mostu (ang. *bridge*); nie należy go mylić z trybem mostu z systemów wirtualizujących (np. VirtualBox); w uproszczeniu o tym trybie można myśleć jak o **wirtualnym przełączniku** (ang. *virtual switch*), wszystkie kontenery podłączone do tego samego urządzenia mogą się komunikować ze sobą oraz z hostem, na którym jest tworzony dedykowany interfejs sieciowy;
- **host** – pozwala **pominąć izolację** pomiędzy hostem (systemem, na którym uruchomiona została usługa Docker) a kontenerem; w tej konfiguracji kontener korzysta

* Więcej informacji na temat *capabilities* zob. rozdz.: K. Szafrański, *Linux – zabezpieczanie i utwardzanie konfiguracji*, s. 329.

bezpośrednio z sieci hosta; działa tylko wtedy, gdy Docker został uruchomiony na systemie Linux;

- **overlay** – tryb sieci używany do **komunikacji** pomiędzy kontenerami/Dockerami uruchomionymi na dwóch różnych hostach; używany jest w przypadku konstruowania klastra[19];
- **ipvlan** – jest analogiczny do trybu **bridge** (most) z wirtualizacji, kiedy kontener otrzymuje możliwość komunikacji na warstwie L2 oraz L3 z wybranego interfejsu sieciowego hosta;
- **macvlan** – pozwala na **przypisanie adresu MAC kontenerowi**, sprawiając, że będzie on widoczny jako fizyczne urządzenie; w przypadku kiedy do hosta trafią pakiety z adresem fizycznym kontenera, zostaną one przekierowane do wybranego kontenera;
- **network-plugin** – tryb pozwala na **instalowanie zewnętrznych sterowników sieci** rozszerzających możliwości sieciowe Dockera; przykładowo możliwe jest podłączenie kontenerów do magistrali/sieci CAN.

Używanie trybów sieciowych wymaga znajomości ich specyfiki. W poniższym zestawieniu wymieniam ich najbardziej **istotne cechy**:

- **none** – tu brak szczególnych uwag;
- **bridge** – wywołanie polecenia: `docker run -p 9998:9999 alpine` spowoduje podłączenie usługi z kontenera uruchomionej na porcie `9999` do **wszystkich** interfejsów sieciowych hosta na porcie `9998`:
  - jeżeli host ma adres publiczny, spowoduje to, że kontener może być dostępny w Internecie oraz z sieci LAN;
  - konfiguracje reguł zapory sieciowej z łańcucha INPUT[20] nie są aplikowane dla takiego ruchu, ponieważ jest to ruch działający w ramach łańcucha FORWARD;
  - sieć bridge zdefiniowana przez użytkownika automatycznie zapewnia rozwiązywanie nazw DNS pomiędzy kontenerami, umożliwiając dostęp do kontenerów po ich nazwach zamiast adresów IP;
  - używanie sieci bridge definiowanych przez użytkownika zapewnia lepszą izolację sieciową pomiędzy kontenerami, ponieważ należy wskazać podczas uruchomienia lub działania kontenera, z której sieci kontener może korzystać;
  - sieci definiowane przez użytkownika mogą być podłączane/odłączane podczas działania kontenera;
  - kontenery znajdujące się w tej samej sieci są dla siebie widoczne oraz mogą podjąć próbę połączenia na **dowolny port** innego kontenera;
- **host** – w tym trybie sieciowym usługi uruchomione w kontenerze będą **dostępne z poziomu portów hosta**:
  - sposób uruchomienia usługi (np. serwera WWW) w kontenerze może spowodować wystawienie usługi na wszystkich interfejsach hosta; przykładowo, jeśli usługa w kontenerze zostanie uruchomiona na wszystkich interfejsach sieciowych, a host ma publiczny adres IP, spowoduje to, że usługa z kontenera będzie dostępna w Internecie oraz z sieci LAN;

- **overlay** – ten typ sieci działa tylko **dla Dockera będącego częścią klastra** (w trybie `swarm` kilka serwerów jest łączonych w celu uruchamiania na nich kolejnych kontenerów);
  - klaster oparty na czystym Dockerze jest rzadko spotykany, ponieważ częściej używa się Kubernetesa lub OpenShifta;
  - dane aplikacji przekazywane w ramach tej sieci mogą nie być szyfrowane; Docker uruchomiony na systemie Windows, dołączony do klastra, nie może używać szyfrowania;
  - żadne błędy nie są wyświetlane, jednak węzeł (maszyna fizyczna/wirtualna będąca częścią klastra) nie może się komunikować;
- **ipvlan**[21] – o tym typie sieci można myśleć obrazowo jak o fizycznym kablu – z jednej strony wpiętym do switcha/routera, a z drugiej do kontenera, który otrzymuje adresację z sieci; jedyną różnicą jest brak unikatowego adresu MAC przypisanego interfejsowi w kontenerze, ponieważ ten adres jest współdzielony z hostem:
  - mogą pojawić się problemy w przypadku, kiedy serwer DHCP przypisuje adresy IP na podstawie MAC;
  - ten tryb może działać w ustawieniu na warstwie L2 lub L3[22];
  - zastosowanie tego typu sieci powoduje, że reguły zapory sieciowej hosta nie będą stosowane dla pakietów wysyłanych do kontenera;
- **macvlan** – ten tryb działa podobnie do ipvlan, z tą różnicą, że kontener otrzymuje własny adres MAC:
  - realny przypadek użycia tego trybu może pojawić się podczas uruchomienia oprogramowania do sniffowania sieci wewnątrz kontenera, ponieważ pozwala przechwytywać pakiety sieciowe na niższych warstwach modelu OSI;
  - w celu uruchomienia tego trybu karta sieciowa, pod którą podłącza się sieć, musi mieć możliwość obsługi kilku adresów MAC lub przełączenia w tryb nasłuchu (ang. *promiscuous mode*);
  - jeżeli karta sieciowa nie będzie miała możliwości zmiany konfiguracji, pakiety będą odrzucane przez sterownik;
  - przełącznik (ang. *switch*) musi mieć możliwość obsługi kilku adresów MAC na jednym porcie;
  - wykorzystanie tego typu sieci powoduje, że reguły zapory sieciowej hosta nie będą stosowane dla pakietów wysyłanych do kontenera;
- **network-plugin** – należy szczególnie uważać i **weryfikować sterowniki**, ponieważ mogą zawierać niebezpieczny kod.

WOLUMENY I WSPÓŁDZIELENIE DYSKU HOSTA

Rozważmy teraz sytuację, w której **w kontenerze jest uruchomiona baza danych, przechowująca dane aplikacji umieszczonej w drugim kontenerze**. Zakładamy, że każdy kontener powinien uruchamiać tylko jedną usługę. Dzięki takiej konstrukcji można w każdej chwili uruchomić kolejne kontenery, zwiększając wydajność środowiska w przypadku wystąpienia tzw. wąskich gardeł.

Zagadnienie skalowania środowisk należy do tematów związanych np. z Kubernetesem[23] i nie zostanie omówione w tym rozdziale.

Jaki będzie jednak skutek błędu powodującego reset kontenera implementującego usługę bazy danych? **Wszystkie przechowywane dane znikną bezpowrotnie, ponieważ kontener zostanie ponownie uruchomiony, a jego stan będzie odtworzony do tego zapisanego w obrazie.**

Aby sprostać takim sytuacjom, udostępniony został **mechanizm montowania zasobów**, który można podzielić na dwie grupy:

1. **dysk zarządzany** (ang. *volume*) – tworzony przez Docker w formie katalogu domyślnie umieszczonego w `/var/lib/docker/volume`, za pomocą komend, z których pierwsza tworzy nowy dysk zarządzany, a druga, przez parametr `-v`, powoduje podmontowanie go do kontenera:
   ```
   $ docker volume create test
   $ docker run -v test:/dane securitum:lalalatest
   ```
2. **katalog współdzielony** (ang. *bind mounts*) – dowolny katalog hosta wskazany podczas uruchamiania kontenera:
   ```
   $ docker run -v /home/kali/:/dane securitum:lalalatest
   ```

Istnieje jeszcze jedna forma – **tmpfs** dostępna w ramach systemów Linux. Jest to **bardzo szybki magazyn danych**, opierający się na pamięci operacyjnej hosta (należy pamiętać, że dane znikną po wyłączeniu kontenera). W przypadku zamiaru skonfigurowania dokładniejszych ustawień montowania należy użyć flagi `--mount`[24].

Na listingu 7 przedstawiono przykład uruchomienia kontenera z tmpfs, który zostanie umieszczony w katalogu `/app` kontenera z pełnymi uprawnieniami do plików i ustawieniem sticky bit[25] (`tmpfs-mode=1777`) oraz ograniczeniem rozmiaru do maksymalnie jednego megabajta (domyślnie 50% ramu hosta). Przy próbie zapisania większej ilości danych zostanie zwrócony błąd związany z brakiem przestrzeni w takim katalogu.

Listing 7. Uruchomienie kontenera z zamontowanym zasobem typu `tmpfs`

```
$ docker run --mount type=tmpfs,destination=/app,tmpfs-mode=1777,tmpfs-
size=1M securitum:lalalatest bash -c 'cat /dev/random > /app/test || ls -lah
/app/test'
cat: write error: No space left on device
-rw-r--r-- 1 root root 1.0M Jun 29 14:37 /app/test
```

Funkcjonalność **podpinania katalogów hosta** umożliwia zapisywanie danych w zamontowanym zasobie. Po restarcie wykorzystywany jest ten sam katalog, co pozwala na zachowanie danych pomiędzy uruchomieniami kontenera. Dzięki temu nowe informacje, które zostały zapisane w czasie działania aplikacji (np. aktualizacje bazy danych), nie są tracone po wyłączeniu kontenera (nie dotyczy mechanizmu tmpfs).

Kolejną ciekawą konfiguracją, która może być pomocna w niektórych sytuacjach, jest **współdzielenie tej samej przestrzeni dyskowej pomiędzy kontenerami**, dzięki czemu możliwe jest skomunikowanie dwóch kontenerów, które nie mają podpiętych interfejsów sieciowych.

UWAGA

Do konfiguracji zawierającej opisane wyżej elementy należy podejść z ostrożnością. Nieprzemyślane wykorzystanie współdzielenia zasobów, jak chociażby podpięcie głównego katalogu hosta (`/`), **może doprowadzić do modyfikacji plików w systemie hosta**, co w efekcie pozwoli na ominięcie izolacji i przejęcie kontroli nad środowiskiem uruchamiającym Dockera (dokładny opis tego ataku znajduje się w dalszej części rozdziału).

Używając wolumenów, warto rozważyć zastosowanie **flagi** `read-only`, dzięki której ograniczona zostaje możliwość tworzenia/edytowania plików w takim zasobie. Trzeba również zwrócić uwagę na uprawnienia, jakie mają pliki w montowanych katalogach, ponieważ niewskazane jest, aby sekrety miały uprawnienia pozwalające na czytanie pliku/plików przez wszystkich użytkowników systemu (kontenera).

Jeżeli, wbrew zaleceniom[26], w kontenerze jest uruchomionych kilka usług z różnymi uprawnieniami, błąd w jednej z nich może pozwolić na uzyskanie dostępu do sekretu innej usługi.

Ciekawym przypadkiem ataku typu DoS na system hosta z poziomu kontenera jest **wyczerpanie zasobów dyskowych**. Może się tak stać w wyniku braku logicznego podziału dysku na wiele partycji. Atakujący po przejęciu kontroli nad kontenerem (lub odkryciu podatności umożliwiającej zapisywanie danych w podmontowanym wolumenie) może zapełnić pozostałe wolne miejsce. Ochronę przed tego typu atakami stanowi podział dysku na partycje, wykorzystanie mechanizmów takich jak quoty systemu plików czy monitorowanie.

PROBLEMY Z ŻYCIA WZIĘTE

Przejdźmy teraz do przeglądu **najczęściej pojawiających się błędów bezpieczeństwa dotyczących Dockera**, z jakimi można się spotkać podczas testów penetracyjnych.

Pamiętaj o hoście

Docker nie działa w próżni: jest usługą uruchamianą na konkretnym systemie operacyjnym, choćby nawet zwirtualizowanym, dlatego pierwszym elementem, o który **należy zadbać**, jest **host**.

Zbyt często w czasie pentestów spotykałem nieaktualny system operacyjny wykorzystujący jądro zawierające znane błędy bezpieczeństwa (np. dystrybucja Ubuntu z kernelem w wersji 5.4.0-96-generic i podatnością CVE-2021-3715). Jaki będzie efekt takich zaniedbań? Możliwość zaatakowania hosta z poziomu kontenera, czyli, *de facto*, **złamanie jednej z kluczowych zasad: izolacji**.

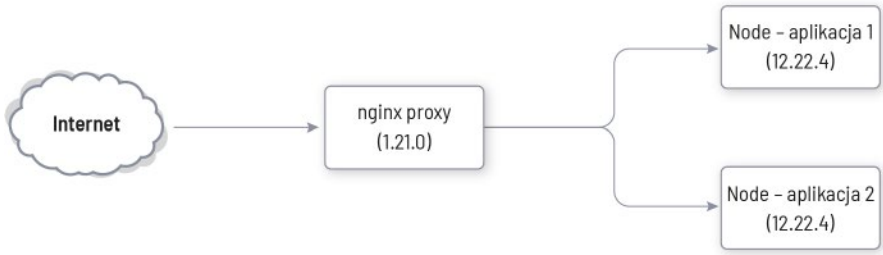
Brak aktualizacji dla samego środowiska Dockera również może skończyć się przejęciem infrastruktury, np. wersja 20.10.7 pozwala na podniesienie uprawnień procesu wewnątrz kontenera (CVE-2022-24769[27]). Znana jest również znacznie bardziej niebezpieczna podatność: CVE-2019-5736[28], pozwalająca na wykonywanie komend[29] z uprawnieniami administratora na hoście poprzez uruchomienie złośliwego obrazu.

WAŻNE

Należy pamiętać o aktualizacji zarówno systemu operacyjnego, jak i oprogramowania uruchomionego na hoście. Warto również przeprowadzić utwardzanie hosta (ang. *hardening*).

Nieaktualne kontenery

Brak aktualizacji jest poważną bolączką środowisk umieszczonych w kontenerach. Załóżmy, że środowisko zostało zbudowane w lipcu 2021 roku (wersje oprogramowania pobrane z Docker Hub) według schematu z rysunku 2.



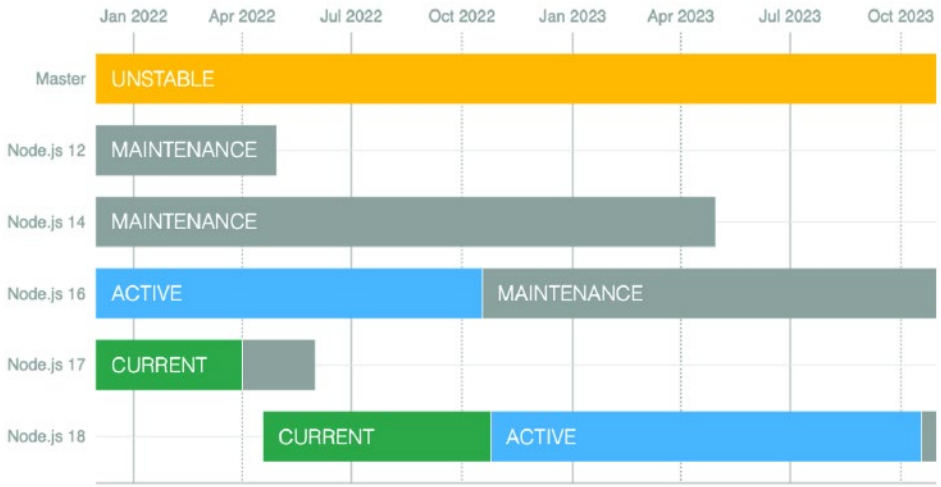
Rysunek 2. Przykładowe środowisko wykorzystujące kontenery

Jeśli obrazy kontenerów nie będą aktualizowane (np. w marcu 2022 roku), okaże się, że w infrastrukturze znajduje się wiele nieaktualnego oprogramowania, zawierającego błędy bezpieczeństwa.

Zamieszczone poniżej zestawienie zawiera listę błędów CVE dla obrazów wykorzystanych w przedstawionym środowisku:

- nginx (wersja: 2.21.0) – brak podatności[30],
- Node (wersja: 12.22.4):
  - CVE-2021-3672,
  - CVE-2021-22931,
  - CVE-2021-22940,
  - CVE-2021-32803,
  - CVE-2021-32804,
  - CVE-2021-22959,
  - CVE-2021-22960,
  - CVE-2021-44531,
  - CVE-2021-44532,
  - CVE-2021-44533,
  - CVE-2022-21824,
  - CVE-2022-0778.

Po dziewięciu miesiącach uzbierało się całkiem sporo luk. Dodatkowo okazuje się, że Node skończył wsparcie[31] tej konkretnej wersji w kwietniu 2022 roku (rysunek 3).



Rysunek 3. Informacje o wsparciu dla Node

Kontenery z uprawnieniami administratora

Błąd związany z uprawnieniami administratora wynika z domyślnych ustawień, jakie ma Docker. Uruchamiając nowy kontener, należy pamiętać, że procesy w kontenerze będą działać w kontekście użytkownika root. Na listingu 8 przedstawiono uruchomienie kontenera na podstawie obrazu `ubuntu:22.04`, w kontenerze uruchomiono polecenie `id`. W wyniku jego działania w kontenerze procesy są uruchamiane z uprawnieniami roota.

Listing 8. Uruchomienie kontenera na podstawie obrazu ubuntu:22.04

```
# Polecenie uruchomione w wierszu poleceń hosta
$ docker run --rm -it ubuntu:22.04 id
uid=0(root) gid=0(root) groups=0(root)
```

Odnosząc się do konfiguracji z wcześniejszego przypadku, można stwierdzić, że dowolna instancja Node po przeprowadzeniu ataku RCE umożliwi wykonanie kodu w kontenerze na uprawnieniach administratora: atakujący będzie w stanie uruchamiać polecenia wewnątrz kontenera, przeszukiwać zasoby, przesyłać pliki czy wykorzystać dowolną technikę (np. *reverse shell*) do połączenia z serwerem C2 w celu łatwiejszego operowania przejętym kontenerem. Dostęp do zdalnych zasobów czy systemów dostępnych z poziomu kontenera może pozwolić np. na pivoting i dalszą eksplorację infrastruktury organizacji.

Skutki ataku można ograniczyć w bardzo prosty sposób: **wystarczy uruchomić proces wewnątrz kontenera z najniższymi potrzebnymi uprawnieniami** (listing 9).

Listing 9. Uruchomienie procesu w kontekście użytkownika www-data

```
# Polecenie uruchomione w wierszu poleceń hosta
$ docker run --rm --user www-data -it ubuntu:22.04 id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Dodatkowo warto rozważyć uruchomienie usługi (ang. *daemon*) Dockera z obniżonymi uprawnieniami (ang. *rootless mode*)[32].

Płaska sieć

Problem ten dotyczy często nawet złożonych systemów. Bywa, że administratorzy nie zmieniają ustawień sieci, która jest zwykle konfigurowana w taki sposób, by **wszystkie kontenery znajdowały się w tym samym jej segmencie**. Niejednokrotnie wynika to z domyślnej konfiguracji sieci, która niekoniecznie jest poprawna z punktu widzenia bezpieczeństwa.

Standardowo po uruchomieniu kontener otrzymuje własny stos sieciowy i jest podpinany do mostu. Sterownik sieciowy Dockera zarządza adresacją IP kontenerów. Dzięki niemu maszyny uzyskują możliwość komunikacji pomiędzy sobą.

Warto prześledzić ten proces na listingach 10 i 11.

Listing 10. Uruchomienie pierwszego kontenera

```
# Polecenie uruchomione w wierszu poleceń hosta
$ docker run --rm -it --name "kali1" kalilinux/kali-rolling bash -c "apt-get
update && apt-get install -y ncat && ncat -lvp 31337"
```

Listing 11. Weryfikacja adresu IP kontenera uruchomionego w domyślnej sieci bridge

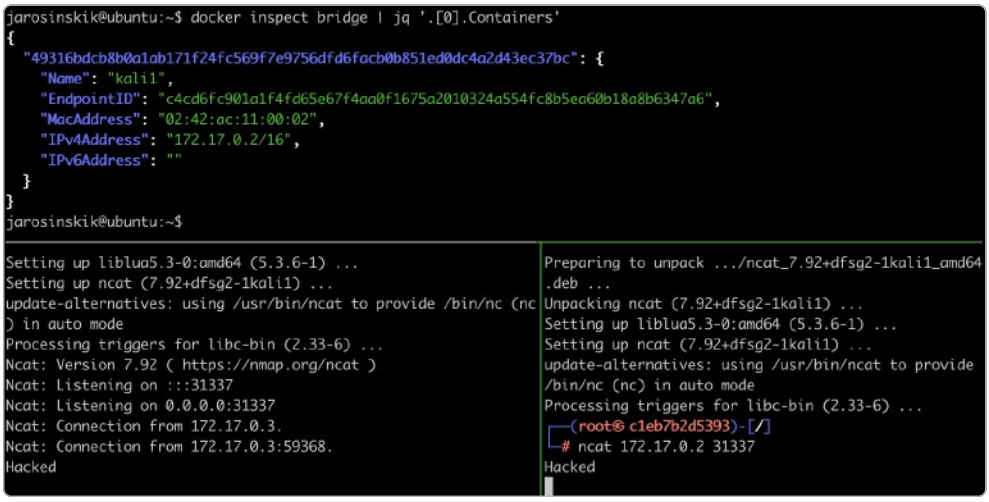
```
# Polecenie uruchomione w wierszu poleceń hosta
$ docker inspect bridge | jq '.[0].Containers'
{

  "49316bdcb8b0a1ab171f24fc569f7e9756dfd6facb0b851ed0dc4a2d43ec37bc": {
    "Name": "kali1",
    "EndpointID":
"c4cd6fc901a1f4fd65e67f4aa0f1675a2010324a554fc8b5ea60b18a8b6347a6",
    "MacAddress":  "02:42:ac:11:00:02",
    "IPv4Address": "172.17.0.2/16",
    "IPv6Address": ""
  }
}
```

Następnie uruchamiany jest kolejny kontener i odbywa się próba nawiązania połączenia z pierwszym (listing 12).

Listing 12. Uruchomienie powłoki w kontenerze kali2

```
# Polecenie uruchomione w wierszu poleceń hosta
$ docker run --rm -it --name "kali2" kalilinux/kali-rolling bash -c
"apt-get update && apt-get install -y ncat && bash"
```

Na rysunku 4 widać potwierdzenie nawiązania połączenia pomiędzy kontenerami.



Rysunek 4. Nawiązanie połączenia między kontenerami (góra: host, lewa kolumna: kali1, prawa kolumna: kali2)

Ograniczenie takiego zachowania jest możliwe dzięki umieszczeniu kontenerów w osobnych mostach[33], konfigurowanych przez użytkownika. Najpierw należy stworzyć nowe sieci (listing 13).

Listing 13. Utworzenie dwóch nowych sieci typu most

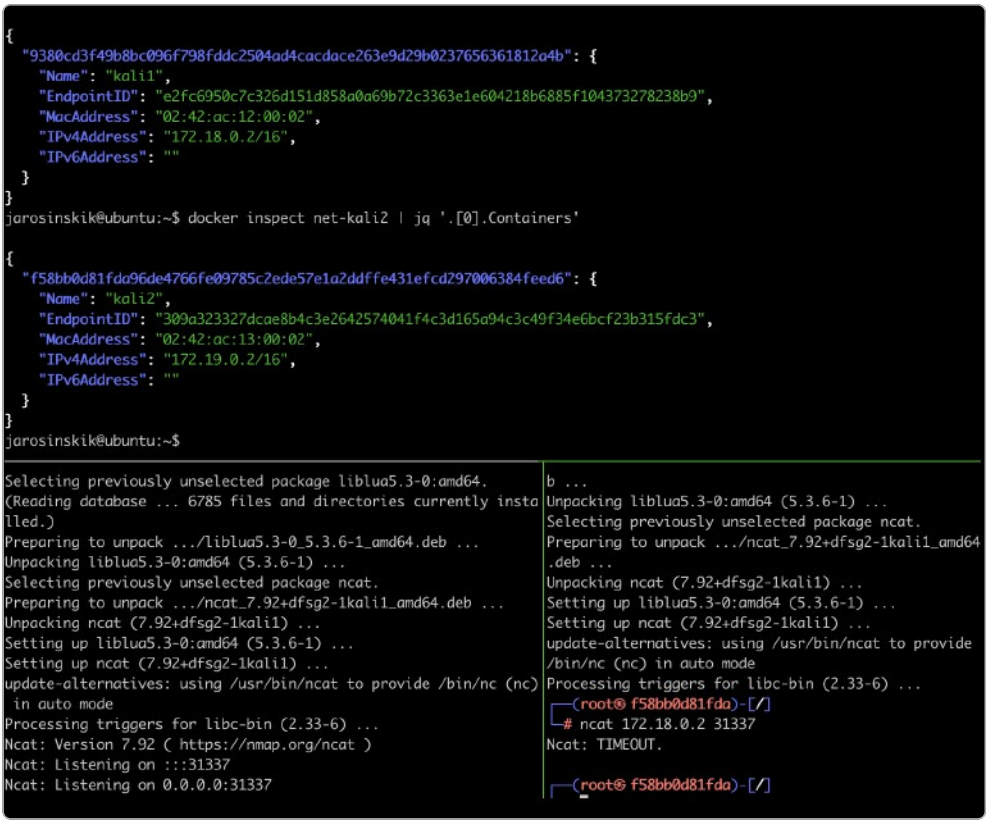
```
# Polecenie uruchomione w wierszu poleceń hosta
$ docker network create net-kali1
1ffe4e570b2f4c090ca16f6c85a1eb15f0a6f13461fd08026367729d7ea85405
$ docker network create net-kali2
fdb0ece963ae8227089c49c921c2c35614f4483b6995ac0bebbb5d4eabfd2282
```

Następnie należy uruchomić kontenery ze wskazaniem, w jakiej sieci mają się znaleźć (listing 14).

Listing 14. Uruchomienie kontenerów w osobnych sieciach

```
# Polecenie uruchomione w wierszu poleceń hosta
$ docker run --rm -it --name "kali1" --network "net-kali1" kalilinux/kali-
rolling bash -c "apt-get update && apt-get install -y ncat && ncat -lvp
31337"
$ docker run --rm -it --name "kali2" --network "net-kali2" kalilinux/kali-
rolling bash -c "apt-get update && apt-get install -y ncat && bash"
```

W efekcie kontenery zostały umieszczone w osobnych sieciach i nie mają komunikacji sieciowej między sobą (rysunek 5).



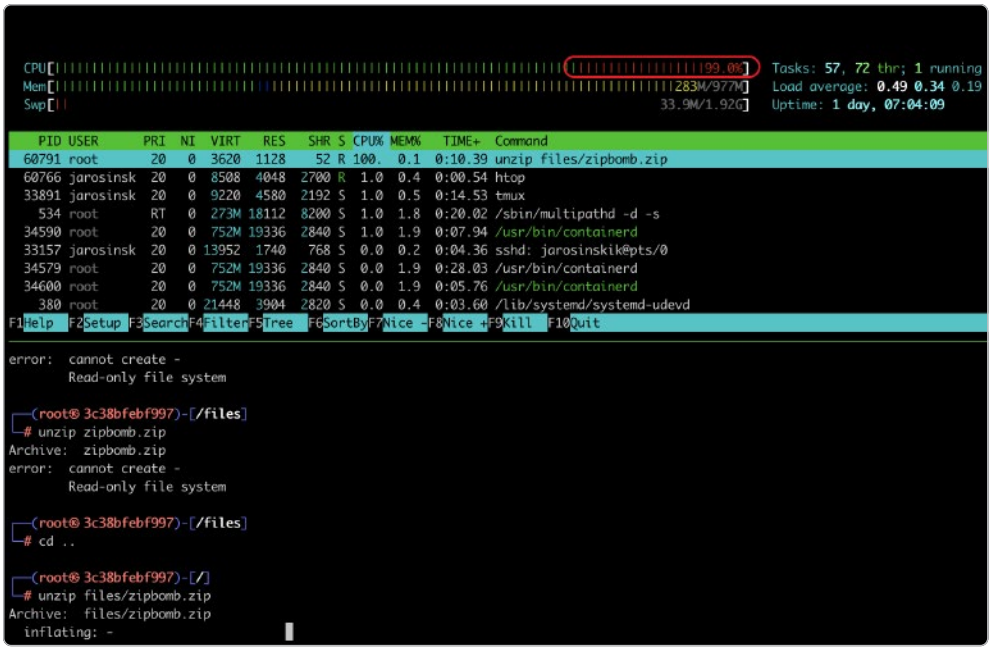
Rysunek 5. Brak możliwości nawiązania połączenia pomiędzy kontenerami uruchomionymi w osobnych sieciach

Alternatywnie można zastosować `flagę --icc=false` na poziomie demona[34], wyłączając komunikację między kontenerami.

Brak ograniczeń zasobów

Wyobraźmy sobie sytuację, że mamy kontener z aplikacją, którą ktoś zaatakował i uzyskał dostęp do kontenera. Jeżeli dostęp do zasobów nie został określony, spowoduje to, że kontener będzie w stanie wykorzystać wszystkie zasoby pamięciowe/CPU hosta.

Na rysunku 6 znajduje się przykład ataku *zip bomb*[35], kiedy w trakcie rozpakowywania archiwum .zip zajmowana jest cała dostępna pamięć operacyjna i obciążany jest procesor. Przy odpowiednio przygotowanym pliku ten proces może wyczerpać całą przestrzeń dyskową.



Rysunek 6. Przykład ataku *zip bomb* (góra: host, dół: kontener)

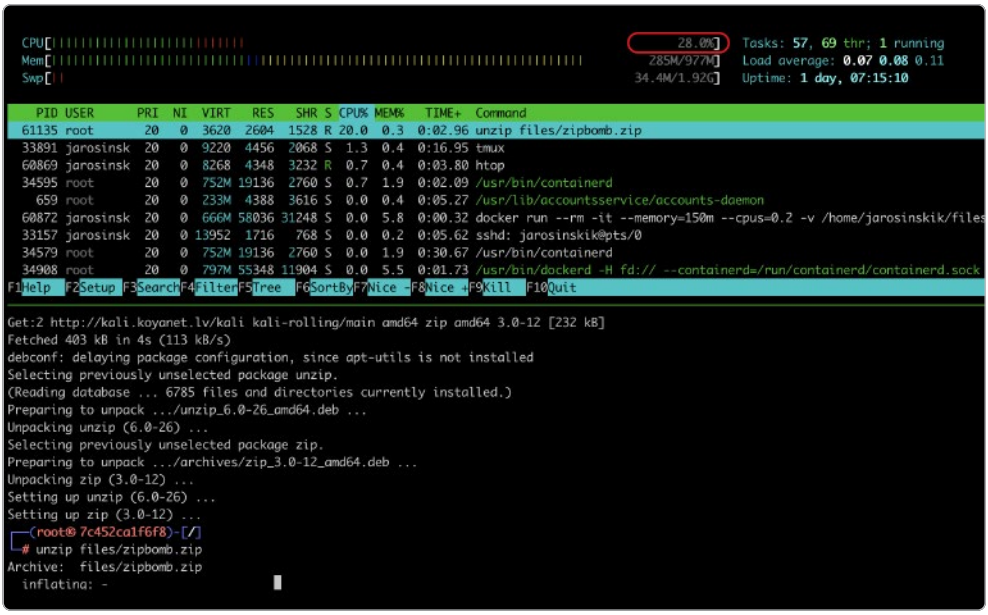
Jak w takim razie ograniczyć zasoby?

Rozwiązanie jest bardzo proste: należy skonfigurować ograniczenia nakładane na kontener za pomocą odpowiednich parametrów[36] (listing 15).

Listing 15. Konfigurowanie ograniczeń nakładanych na kontener

```
# Polecenie uruchomione w wierszu poleceń hosta
$ docker run --rm -it --memory="150m" --cpus="0.2"  -v /home/jarosinskik/
files/:/
files:ro  kalilinux/kali-rolling bash -c "apt-get update && apt-get install
-y zip && bash"
```

W przypadku z listingu 15 ograniczono dostęp do pamięci do poziomu 150 MB oraz do 20% wykorzystania procesora. Docker do ograniczenia zasobów wykorzystuje cgroups[37] (dla więcej niż jednego rdzenia, np. dla trzech, przeznaczenie 100% mocy wymaga podania wartości 3). Przykład ataku *zip bomb* z zastosowaniem ograniczeń CPU i MEM przedstawiony jest na rysunku 7.



Rysunek 7. Przykład ataku *zip bomb* z zastosowaniem ograniczeń CPU i MEM (góra: host, dół: kontener)

Uprawnienia R/W na plikach hosta

Bardzo często przy uruchamianiu jakiejś usługi w kontenerze należy przekazać jej pliki konfiguracyjne znajdujące się na hoście (ewentualnie konfiguracja może znaleźć się w obrazie kontenera). **Przekazanie konfiguracji opiera się na podpięciu wybranego pliku z hosta do kontenera (listing 16).**

Listing 16. Podmontowanie wybranego pliku z hosta do kontenera

```
# Polecenie uruchomione w wierszu poleceń hosta
$ cat config.conf
Super secret
$ docker run --rm -v /home/jarosinskik/config.conf:/app/config.conf -it
ubuntu:22.04 bash -c 'echo "Hacked" >> /app/config.conf'
$ cat config.conf
Super secret
Hacked
```

Jak widać na listingu 16, kontener mógł zmodyfikować plik znajdujący się na hoście, więc możliwa jest nieuprawniona zmiana działania usługi w kontenerze. Jeżeli konfiguracja będzie zamontowana również do pozostałych kontenerów, w nich też nastąpi zmiana (wprawdzie najczęściej wymagany jest do tego restart usługi, jest to jednak tylko kwestią czasu).

Jakie konsekwencje mogłyby się pojawić, gdyby z jakiegoś powodu został podmontowany główny katalog hosta (`/`) lub katalog konfiguracji (`/etc`)? Atakujący otrzymałby możliwość modyfikowania plików hosta.

Jak więc należy montować pliki w przypadku, gdy nie jest wymagana ich edycja z poziomu kontenera? Należy zastosować, jak na listingu 17, **etykietę tylko do odczytu** (ang. *read-only*).

Listing 17. Podmontowanie wybranego pliku z hosta do kontenera w trybie *read-only*

```
# Polecenie uruchomione w wierszu poleceń hosta
$ cat config.conf
Super secret
$ docker run --rm -v /home/jarosinskik/config.conf:/app/config.conf:ro -it
ubuntu: 22.04 bash -c 'echo "Hacked" >> /app/config.conf'
bash: line 1: /app/config.conf: Read-only file system
$ cat config.conf
Super secret
```

Tym razem w odpowiedzi pojawia się błąd informujący o fakcie, że plik jest wyłącznie do odczytu. Można również rozważyć włączenie trybu tylko do odczytu na poziomie całego kontenera (wszystkich plików kontenera). Dzięki temu atakujący, nawet z uprawnieniami administracyjnymi w kontenerze, nie będzie w stanie edytować żadnych plików (listing 18).

Listing 18. Włączenie trybu tylko do odczytu na poziomie całego kontenera (wszystkich plików kontenera)

```
# Polecenie uruchomione w wierszu poleceń hosta
$ docker run --rm --read-only -it ubuntu:22.04 bash -c 'id; echo "test" > ~
/test.txt'
uid=0(root) gid=0(root) groups=0(root)
bash: line 1: /root/test.txt: Read-only file system
```

UWAGA

W przypadku podmontowania wolumenu (ang. *volume*) lub katalogu/pliku hosta bez dodania etykiety `:ro` edycja takich plików będzie możliwa, pomimo powyższego trybu na poziomie całego kontenera.

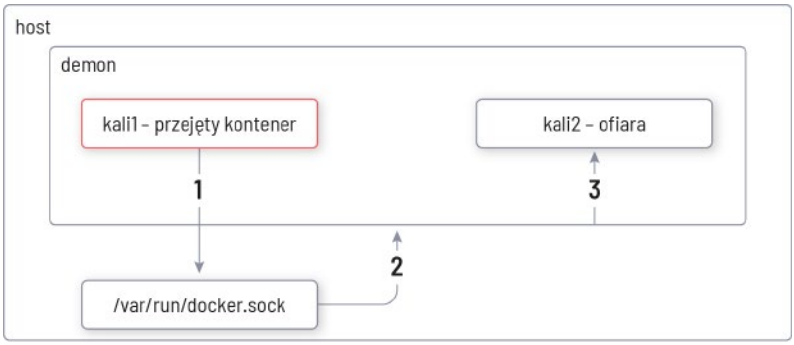
Dostęp do Docker API

Podczas uruchamiania usługi Dockera w domyślnej konfiguracji tworzony jest UNIX socket (`/var/run/docker.socket`)[38], który umożliwia wysyłanie demonowi komendy do wykonania, np.: stworzenie nowego kontenera, zmiany w sieci, wykonanie polecenia na wybranym kontenerze.

Jeżeli konfiguracja montuje wyżej wymieniony plik, a kontener został przejęty przez atakującego z możliwością pisania do pliku, atakujący zyskuje możliwość kontroli nad hostem z uprawnieniami, jakie posiada użytkownik, za pomocą którego uruchomiony został demon Dockera (najczęściej root).

Dlaczego? Dzięki temu plikowi możliwe jest stworzenie nowego kontenera z zamontowanym głównym katalogiem hosta, pliku z kluczami SSH itp. i uzyskanie dostępu do powłoki systemowej hosta.

W celach demonstracyjnych wykonam polecenie w systemie operacyjnym drugiego kontenera znajdującego się w środowisku przedstawionym na rysunku 8.



Rysunek 8. Architektura atakowanego środowiska

Odbywa się to w trzech krokach:

1. **Krok 1** – potwierdzenie obecności pliku gniazda (ang. *socket*). Typowa lokalizacja to `/var/run/docker.sock`, plik ten może się jednak znaleźć również w innych miejscach, dlatego należy go szukać. Po zidentyfikowaniu pliku można spróbować wysłać polecenie pobierające informacje na temat kontenerów zarządzanych przez demona.
2. **Krok 2** – rozpoczęcie komunikacji z gniazdem.

Listing 19. Wylistowanie uruchomionych kontenerów za pomocą gniazda demona Dockera

```
# Polecenie wykonane w kontenerze przejętym przez atakującego
$ curl --unix-socket /var/run/docker.sock http://localhost/containers/json

[{"Id":"dbc2b25527245e92be697ade7e399d84b7d19efd8e203184d21c1c1f330c21bd",
"Names":["/kali1"],[...],"Mounts":[{"Type":"bind","Source":"/var/run/docker.
sock","Destination":"/var/run/docker.sock","Mode":"ro","RW":false,"Propagati
on":"rprivate"}],"IPAddress":"172.17.0.2",[...]}
{"Id":"92a5b1b13c9b70c8982cf76777f0701cda5c11dd506699d19e3b14607218342a",
"Names":["/kali2"],[...],"IPAddress":"172.17.0.3",[...]}]
```

Interesujące jest, że nawet mimo zamontowania pliku w trybie tylko do odczytu możliwe jest wykonywanie poleceń (z listingów 19–21). Mając już identyfikator kontenera ofiary, można przygotować zapytanie tworzące polecenie do wykonania (listing 20).

Listing 20. Zapytanie tworzące polecenie do wykonania

```
# Polecenie wykonane w kontenerze przejętym przez atakującego
$ curl -H "Content-Type: application/json" -d '{"AttachStdout": true,"Cmd":
["ifconfig"]}' --unix-socket /var/run/docker.sock 'http://localhost/containe
rs/92a5b1b13c9b70c8982cf76777f0701cda5c11dd506699d19e3b14607218342a/exec'

{"Id":"8ecb65b7c41db457972b43938f1231a2e5782add7a6b7d8ee6988c9156b8ebf0"}
```

W odpowiedzi API zwraca identyfikator polecenia, który należy wywołać, aby komenda została wykonana.

3. **Krok 3** – wywołanie polecenia w kontenerze ofiary (listing 21).

Listing 21. Wywołanie polecenia w kontenerze ofiary

```
# Polecenie wykonane w kontenerze przejętym przez atakującego
$ curl -H "Content-Type: application/json" -d '{"Tty": true}' --unix-socket
/var/run/docker.sock 'http://localhost/exec/8ecb65b7c41db457972b43938f1231a2
e5782add7a6b7d8ee6988c9156b8ebf0/start'

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.3  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:ac:11:00:03  txqueuelen 0  (Ethernet)
        RX packets 106  bytes 257683 (251.6 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 95  bytes 5496 (5.3 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Finalnie udało się wykonać polecenie w kontenerze, który był celem ataku. Potwierdzeniem skuteczności tego działania jest zdobyty adres IP. Ze względu na zagrożenie, jakie niesie ze sobą API, należy szczególnie dbać o bezpieczeństwo tego zasobu oraz **nie montować go do kontenerów**.

(Nie)bezpieczne repozytorium obrazów

Kolejnym elementem, który jest szczególnie istotny z punktu widzenia bezpieczeństwa rozwiązania opartego na Dockerze, są obrazy kontenerów. Należy je dokładnie sprawdzić, zanim zostaną zbudowane i uruchomione, ponieważ uruchamianie oprogramowania z repozytorium bez weryfikacji, np. z publicznego Docker Huba, może spowodować wykonanie niebezpiecznego kodu w kontenerze/hoście.

Atak taki polega na umieszczeniu w repozytorium obrazu zawierającego polecenia do wykonania. W poniższym przypadku pobierane było oprogramowanie do wydobywania kryptowalut. Obrazy zostały nazwane `alpine/alpine2`, co miało skłonić ofiary do ich pobrania (alpine[39] to minimalistyczny obraz linuksowy, bardzo często stanowiący punkt wyjścia do dalszej rozbudowy).

Użytkownicy, którzy padli ofiarą oszustwa, korzystali ze spreparowanego złośliwego obrazu. Po jego zbudowaniu, kompilacji zależności i uruchomieniu rozpoczynał się proces wydobywania kryptowaluty mającej zasilić portfel przestępcy, który umieścił obraz w publicznym rejestrze. Więcej szczegółów można znaleźć w opisie tego przypadku autorstwa Augusta Remillana[40].

Jak się bronić przed atakami tego typu?

Należy sprawdzać, co znajduje się w pliku z obrazem. Oczywiście można taki **skan wykonać automatami**, np. skanerem Snyk lub Trivy, a nawet zintegrować je z własnym procesem wytwarzania[41].

Chciałbym też zaznaczyć, że nawet posiadając prywatne repozytoria, **warto przeprowadzać** takie **weryfikacje**, ponieważ różne osoby mogą mieć dostęp do repozytorium. Ci użytkownicy lub serwer przechowujący obrazy także mogą zostać zaatakowani.

Interesującym pomysłem jest zastosowanie **podpisów cyfrowych dla obrazów opublikowanych w repozytorium**. Aby jednak podpis był weryfikowany, wymagane jest ustawienie zmiennej środowiskowej na kliencie pobierającym obraz: **`DOCKER_CONTENT_TRUST=1`** (domyślną wartością jest **`0`**).

PODSUMOWANIE

Mam nadzieję, że udało mi się przedstawić najczęstsze problemy związane z używaniem Dockera. Nie są to wszystkie obszary, na których warto się skupić, dlatego w linkach pozostawiam dodatkowe materiały, które warto wykorzystać do uzupełnienia wiedzy na ten temat. Poniżej zamieszczam też listę kontrolną, która może pomóc w ocenie stanu infrstruktury wykorzystującej kontenery Dockera.

✓ CHECKLISTA: KONFIGURACJA KONTENERÓW

Prostym sposobem na sprawdzenie, czy **kontener został poprawnie skonfigurowany**, jest udzielenie odpowiedzi na poniższe pytania:

1. Czy host został utwardzony (jaki jest jego hardening*)?
2. Czy został użyty tryb *rootless*?
3. Czy API Docker nie zostało udostępnione w sieci (TCP)[42]?
4. Czy dostęp do API Docker został ograniczony do użytkowników uprzywilejowanych?
5. Czy plik `/var/run/docker.sock` nie został zamontowany do kontenera?
6. Czy system operacyjny i oprogramowanie hosta są w najnowszej (najbardziej aktualnej) wersji?
7. Czy stosowane są najlepsze praktyki tworzenia pliku Dockerfile?
8. Czy obrazy kontenerów są pobierane z zaufanego źródła?

* Więcej nt. hardeningu Linuksa zob. rozdz. K. Szafrański, *Linux – zabezpieczanie i utwardzanie konfiguracji*, s. 229.

9. Czy weryfikacja podpisów obrazów została włączona?
10. Czy obrazy kontenerów są weryfikowane pod kątem ewentualnych zagrożeń?
11. Czy obrazy kontenerów są w najnowszej wersji?
12. Czy kontenery używają konta użytkownika z ograniczonymi uprawnieniami?
13. Czy wprowadzono odpowiednią segmentację sieci używanej przez kontenery?
14. Czy zasoby przydzielone kontenerowi zostały ograniczone do niezbędnego minimum (CPU, RAM)?
15. Czy kontenery nie działają w trybie uprzywilejowanym (`--privileged`)?
16. Czy z kontenerów usunięto nadmiarowe możliwości (`--cap-drop ALL`)?
17. Czy kontenery są uruchamiane z politykami `AppArmor/SELinux`?
18. Czy montowane wolumeny/katalogi w kontenerze posiadają flagę zezwalającą wyłącznie na odczyt, jeżeli jest taka możliwość?
19. Czy uprawnienia na podmontowanych zasobach są prawidłowo ograniczone (pliki poufne nie powinny dawać możliwości odczytywania ich przez wszystkich użytkowników)?

Dalsze poszerzanie wiedzy

Zainteresowanych tematem Czytelników zachęcam do zapoznania się w pierwszej kolejności z poniższą literaturą:

- Docker Docs, *Docker security*, https://docs.docker.com/engine/security/
- OWASP Cheat Sheet Series, *Docker Security Cheat Sheet*, https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html
- Osnat R., *Top 22 Docker Security Best Practices: Ultimate Guide*, Aqua Blog, July 1, 2021, https://blog.aquasec.com/docker-security-best-practices
- Segura T., *Docker Security Best Practices [cheat sheet included]*, GitGuardian, December 20, 2022, https://blog.gitguardian.com/how-to-improve-your-docker-containers-security-cheat-sheet/
- Webscale, *Deepen your technical knowledge*, https://www.section.io/engineering-education/best-practices-to-secure-a-docker-container/
- Center for Internet Security (CIS), *CIS Benchmarks*, https://learn.cisecurity.org/benchmarks

PRZYPISY

wdbit2.sekurak.pl/r10/

REPOZYTORIUM

wdbit2.sekurak.pl/l/r10

1 *CP/CMS* [w:] *Wikipedia, the Free Encyclopedia*, https://en.wikipedia.org/wiki/CP/CMS
2 *Mainframe computer* [w:] *Wikipedia, the Free Encyclopedia*, https://en.wikipedia.org/wiki/Mainframe_computer
3 Docker, https://www.docker.com/press-release/dotcloud-inc-now-docker-inc/
4 Docker, *Use containers to Build, Share and Run your applications*, https://www.docker.com/resources/what-container/; van Kalken S., *What Are Namespaces and cgroups, and How Do They Work?*, NGINX Community Blog, July 21, 2021, https://www.nginx.com/blog/what-are-namespaces-cgroups-how-do-they-work/
5 Za: Docker, *Use containers to Build, Share and Run your applications*, https://www.docker.com/resources/what-container/
6 Docker Hub, https://hub.docker.com/
7 Docker Docs, *ADD or COPY* [w:] *Building best practices. Use multi-stage builds*, https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#add-or-copy
8 Docker Docs, *Building best practices. Use multi-stage builds*, https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
9 Zob też: Docker Docs, *Docker image history*, https://docs.docker.com/reference/cli/docker/image/history/
10 Prevasio, *Industry's First Dynamic Analysis of 4 million Publicly Available Docker Hub Container Images. Operation Red Kangaroo*, 2020, https://www.algosec.com/wp-content/uploads/pdf/Prevasio-Red-Kangaroo.pdf
11 Snyk, *Docker Security Scanning Guide 2023*, https://snyk.io/learn/docker-security-scanning/
12 Gregório G. (geyslan) i in. (Aqua Security), *Trivy*, GitHub, https://github.com/aquasecurity/trivy
13 Wist K. i in., *Vulnerability Analysis of 2500 Docker Hub Images* [w:] *Advances in Security, Networks, and Internet of Things. Proceedings from SAM'20, ICWN'20, ICOMP'20, and ESCS'20. Conference proceedings*, Berlin 2021, s. 307 – 327, https://arxiv.org/pdf/2006.02932
14 Seals T., *Docker Hub Hack Affects 190K Accounts, with Concerning Consequences*, Vulners, April 29, 2019, https://vulners.com/threatpost/THREATPOST:B047BB0FECBD43E30365375959B09B04
15 Docker Docs, *Content trust in Docker*, https://docs.docker.com/engine/security/trust/
16 Polop C., *Docker Breakout / Privilege Escalation*, HackTricks, last updated: June 5, 2024, https://book.hacktricks.xyz/linux-hardening/privilege-escalation/docker-breakout/docker-breakout-privilege-escalation#capabilities-abuse-escape
17 Docker Docs, *Runtime privilege and Linux capabilities* [w:] *Running containers*, https://docs.docker.com/engine/reference/run/#runtime-privilege-and-linux-capabilities
18 Zob. też Keith Ch. (NetworkChuck), *Docker networking is CRAZY!! (you NEED to learn it)*, YouTube, August 6, 2022, https://www.youtube.com/watch?v=bKFMS5C4CG0
19 Zob. też: Docker Docs, *Swarm mode overview*, https://docs.docker.com/engine/swarm/
20 Russell R. i in., *iptables(8) – Linux man page*, die.net, https://linux.die.net/man/8/iptables
21 Bhardwaj R., *MacVLAN vs IPvlan: Understand the difference*, IP With Ease, https://ipwithease.com/macvlan-vs-ipvlan-understand-the-difference/
22 Docker Docs, *IPvlan network driver*, https://docs.docker.com/network/drivers/ipvlan
23 Kubernetes, *Production-Grade Container Orchestration*, https://kubernetes.io/
24 Docker Docs, *Mounts*, https://docs.docker.com/build/guide/mounts/; zob. też: Docker Docs, *tmpfs mounts*, https://docs.docker.com/storage/tmpfs/
25 Carrigan T., *Linux permissions: SUID, SGID, and sticky bit*, Red Hat, October 15, 2020, https://www.redhat.com/sysadmin/suid-sgid-sticky-bit
26 *Decouple applications* [w:] Docker Docs, *Building best practices*, https://docs.docker.com/build/building/best-practices/#decouple-applications
27 McGowan D. (dmcgowan), *Default inheritable capabilities for linux container should be empty*, GitHub, March 24, 2022, https://github.com/containerd/containerd/security/advisories/GHSA-c9cp-9c75-9v8c
28 Sarai A., *CVE-2019-5736: runc container breakout (all versions)*, SecLists.org, February 12, 2019, https://seclists.org/oss-sec/2019/q1/119
29 Avrahami Y., *Breaking out of Docker via runC – Explaining CVE-2019-5736*, Unit 42, February 21, 2019, https://unit42.paloaltonetworks.com/breaking-docker-via-runc-explaining-cve-2019-5736/
30 NGINX, *nginx security advisories*, https://nginx.org/en/security_advisories.html
31 nodejs.org, *Node.js Releases*, https://nodejs.org/en/about/releases/
32 Docker Docs, *Run the Docker daemon as a non-root user (Rootless mode)*, https://docs.docker.com/engine/security/rootless/
33 Docker Docs, *Differences between user-defined bridges and the default bridge*, [w:] *Bridge network driver*, https://docs.docker.com/network/bridge/#differences-between-user-defined-bridges-and-the-default-bridge
34 Docker Docs, *Daemon CLI (dockerd)*, https://docs.docker.com/engine/reference/commandline/dockerd/

35 Sajdak M. (ms), *42kB zip rozpakowuje się do 4,5 petabajta. Potraktował atakujących tego typu zip bombą*, sekurak.pl, 6 lipca 2017, https://sekurak.pl/42kb-zip-rozpakowuje-sie-do-45-petabajta-potraktowal-atakujacych-tego-typu-zip-bomba/
36 Docker Docs, *Runtime options with Memory, CPUs, and GPUs*, https://docs.docker.com/config/containers/resource_constraints/
37 Kerrisk M., *cgroups(7) – Linux manual page*, man7.org, December 22, 2023, https://www.man7.org/linux/man-pages/man7/cgroups.7.html
38 Docker Docs, *Develop with Docker Engine API*, https://docs.docker.com/engine/api/
39 Docker Hub, *Alpine*, https://hub.docker.com/_/alpine
40 Remillano A., *Malicious Docker Hub Container Images Used for Cryptocurrency Mining*, Trend Micro, 19 agosto 2020, https://www.trendmicro.com/vinfo/it/security/news/virtualization-and-cloud/malicious-docker-hub-container-images-cryptocurrency-mining
41 Snyck User Docs, *Snyk CI/CD integrations*, https://docs.snyk.io/scm-ide-and-ci-cd-workflow-and-integrations/snyk-ci-cd-integrations
42 *Daemon socket option* [w:] Docker Docs, *Daemon CLI (dockerd)*, https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-socket-option