Univerzita Hradec Králové
Fakulta informatiky a managementu
Fakulta informatiky

Využití Docker pro bezpečné testování software

Bakalářská práce

Autor: David Louda
Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Mgr. Josef Horálek, Ph.D.

Hradec Králové Duben 2024

Prohlášení:

Prohlašuji, že jsem bakalářskou/diplomovou práci zpracoval/zpracovala samostatně a s použitím uvedené literatury.

V Hradci Králové dne 21.4.2024

David Louda

Poděkování:

Děkuji vedoucímu bakalářské/diplomové práce Mgr. Josefu Horálkovi Ph.D. za metodické vedení práce a ochotu v operativním fungování.

Abstrakt

Tato bakalářská práce se zabývá kontejnerizační technologií Docker se zaměřením na její bezpečnost a testování software.

Teoretická část se zabývá analýzou softwarového řešení Docker. Zkoumá zasasení této technologie v kontextu alternativních řešení, která umožňují provozování aplikací. Teoretická část se dále zabývá charakteristikami vnitřní architektury Dockeru.

V praktické části je Docker porovnán s řešením vrtuálních strojů. V práci je praktická ukázka instalace Dockeru na různých prostředích, konfigurace Dockeru a provoz kontejnerizované aplikace. Poslední část se zabývá bezpečnější variantou instalace rootless Docker. Na této rootless instalaci je opakován test, kdy se testovaný software pokouší o neoprávněný přístup.

Docker byl na základě teoretické i praktické části shledán vhodným pro provoz a testování software ve scénářích, kde je klíčová stabilita aplikačního prostředí a optimalizace zdrojů. Zvláštní pozornost by měla být dále věnována klasické rootful instalaci a interakcí s Docker zdroji třetích stran.

Klíčová slova: Docker, kontejner, image, bezpečnost, operační systém

Abstract

Title: Using Docker for secure software testing

This Bachelor Thesis explores the Docker containerization technology with a focus on its security and software testing.

The theoretical part is analyzing this software solution and puts it into the perspecive of bot alternative closer and more distant solutions. It also focuses on internal architecture and functioning of Docker as well as User options when it comes to working with it.

In the practical part, Docker is compared with the Virtual Machines solution. The thesis also includes a practical demonstration of Docker installation process on different environments with scenarios covering configuration and run of containerized applications. The last section presents practical scenario with the more secure variant of Docker – the so called rootless Docker installation. Retest of running a containerized software that attempts to gain unauthorized access to the root filesystem is demonstrated.

Based on both the theoretical and practical sections, Docker was found to be suitable for running and testing software in scenarios where application environment stability and resource optimization are key requirements. Further, special attention should be paid to classic rootful installation and interactions with third-party Docker resources.

Key words: Docker, container, image, security, operating system

Obsah

Abstrakt 4

Abstract 5

Úvod 8

1. Cíl práce 9

2. Metodika zpracování 10

3. Úvod do Dockeru 11

3.1. Co je Docker – bližší vymezení 11

3.2. Architektura Dockeru 14

3.2.1. Docker klient a Docker daemon (server) 14

3.2.2. Docker registry 15

3.2.3. Docker Images 15

3.2.4. Docker kontejner 15

3.3. Interakce 15

4. Testování software 17

4.1. Typy testování software 17

4.2. Výhody použití Dockeru pro testování 17

5. Bezpečnostní rizika vztahu host - Docker 19

5.1. Bezpečnostní aspekty kontejnerů 19

5.2. SELinux a AppArmor 19

5.3. Best practices pro zabezpečení Docker kontejnerů 20

6. Porovnání Dockeru s alternativami 22

7. Praktická část 26

7.1. Webová aplikace & Webový server 26

7.2. Příprava prostředí 26

7.2.1. Windows & Docker Desktop 26

7.2.2. Linux VM 27

7.2.3. Linux Server 28

7.3. Docker & Docker Compose 29

7.4. Provozování webové aplikace – simulace neúměrné zátěže 31

7.4.1. Zátěžový test webové aplikace na Linux Serveru 32

7.4.2. Zátěžový serveru test webové aplikace na Linux VM 32

7.5. Test bezpečnosti – nebezpečný Docker image 35

7.5.1. Příprava image 35

7.6. Rootless Docker 37
7.6.1. Instalace rootless Dockeru 38
7.6.2. Rootless Docker & Hello World UHK aplikace 39
7.6.3. Teorie v praxi – nebezpečný Docker image & Rootless Docker 39
8. Shrnutí a diskuse výsledků 42
9. Závěry a doporučení 43
10. Seznam použité literatury 44
11. Přílohy 47
12. Zadání práce z IS (eVŠKP) 58

Úvod

Tato bakalářská práce se zabývá kontejnerizačním softwarem, který se jmenuje Docker. Technologie Docker byl a zůstává synonymem pro práci s kontejnerizovanými řešeními, ačkoli v posledních letech jeho monopol slábne. Avšak komunita kolem Dockeru a množství dostupných materiálů zůstává natolik rozmanité, že je zvolen jako ústřední kontejnerizační řešení pro tuto práci.

Docker je v teoretické části této práce analyzován z různých pohledů. Jako první je na Docker nahlíženo z architektonického pohledu. Zde je věnována pozornost komponentám, které jako dílky skládanky tvoří Docker ekosystém. Jedná se o Docker klienty, Docker daemona, registries, images a kontejnery. Následně je definováno testování software a základní kategorie testování. Pozornost je také věnována bezpečnostním prvkům při práci s kontejnery z pohledu uživatele i daemona, který s images a kontejnery pracuje. Poslední sekce teoretické části se věnuje porovnání Dockeru s alternativami, a to jak v rámci kontejnerizační domény (Podman, Unikernel), tak v širší perspektivě, kde je srovnán s procesy nebo virtuálními stroji.

V praktické části je Docker představen na různých operačních systémech. Docker je zde instalován na tři různé operační systémy, Windows 10, Ubuntu Linux server a KaliOS VM provozovanou přes Oracle VM VirutalBox.

Pro testovací účely je připravena webová aplikace s webovým serverem, pro které jsou představeny konfigurační soubory, které uživateli umožňují aplikace provozovat formou kontejnerů.

V další kapitole je proveden zátěžový test, který ukazuje chování Dockeru na různých architekturách, zejména se zaměřuje na práci kontejnerů se zdroji hosta. Poslední část praktické části se věnuje instalaci rootless Dockeru, který je zajímavým rozšířením standartní instalace. Rootless, v překladu „bez roota“, znamená, že ani Docker daemon ani kontejnery neběží pod root uživatelem. Tato forma instalace znamená zvýšenou bezpečnost z pohledu hostitelského OS i kontejnerů navzájem. Na této rootless instalaci je demonstrován pokus o neoprávněný přístup kontejneru k souborovému systému hosta.

Následuje kapitola diskutující a shrnující výsledky, kapitola se závěry a doporučeními a uvedení seznamu použité literatury s přílohami.

1. Cíl práce

Cílem práce je hlubší porozumění kontejnerizačním, popřípadě virtualizačním nástrojům, zejména potom Dockeru, a to z několika pohledů. Jedná se o pohled uživatele, který s těmito technologiemi pracuje a provádí prvotní instalaci a konfiguraci. Dále potom o pohled operačních systémů a jak je v nich Docker integrován. V neposlední řadě je cílem práce ukázat praktické zacházení s Dockerem a kontejnery a ověření znalostí a předpokladů z teoretické části praktickými scénáři.

Tyto reprodukovatelné praktické scénáře simulují zároveň scénáře, při kterých by byl testován software. Popsané mechanismy spojené s touto kontejnerizační službou nabízí čtenáři praktický pohled, jak provozování, respektive testování, softwaru v Dockeru může probíhat. Čtenář je taktéž seznámen s alternativami, a to s celou škálou možností od PC až po individuální proces běžící na zařízení a se silnějšími i slabšími vlastnostmi těchto nástrojů.

2. Metodika zpracování

První, teoretická část práce se zabývá postavením kontejnerizační technologie Docker v dnešním světě softwaru a jejím využitím. V této teoretické části práce sleduje dvě hlavní myšlenky.

První myšlenkou je porozumění Dockeru jako službě, a především jejímu fungování za použití dostupné literatury a dalších zdrojů. Druhou myšlenkou je určité vymezení této služby v porovnání s alternativami a porozumění odlišnostem mezi Dockerem a těmito alternativami.

Tyto informace budou následně sloužit jako vstupy pro praktickou část, ve které budou připravena prostředí a ověřeny předem definované scénáře, které budou následně analyzovány.

Praktická část vede k naplnění hlavního cíle bakalářské práce. Popisuje přípravu Docker prostředí na různých operačních systémech. Obsahuje vývoj softwarového řešení k uskutečnění demonstrativních scénářů. Scénáře jsou v kontrolovaném prostředí uskutečněny a jejich výsledky zanalyzovány v duchu myšlenky „testování software v prostředí Docker“. Na konec jsou praktické scénáře vyhodnoceny proti teoretickým předpokladům a znalostem získaným v teoretické části práce a navržena doporučení pro reálné scénáře.

3. Úvod do Dockeru

Docker byl a zůstává pomyslným pionýrem v softwarovém světě kontejnerizace. Jednalo se o první široce rozšířený nástroj, který byl ke kontejnerizaci softwarových řešení takto využíván a do dnešních dní těží z tohoto pomyslného prvenství formou popularity, sílou komunity a množstvím dostupných materiálů (Osnat, 2020), (Ali, 2023).

Docker je open-source platforma pro kontejnerizaci aplikací. Kontejnerizační technologie umožňuje spouštění aplikací a jejich doprovodných služeb v izolovaných kontejnerech. Tyto kontejnery, obsahující aplikace anebo programy, nejsou závislé na operačním systému hosta, neboť aplikace je v kontejneru obalena všemi nezbytnými nástroji, včetně operačního systému a tvoří tak soběstačnou jednotku.

Na hostitelském stroji běží Docker Engine, což je jedna z dílčích komponent Dockeru, která zajišťuje technický provoz těchto kontejnerů (Enlyft, 2024), (Singh & Singh, 2016). Následující citace z oficiálních stránek Dockeru nastiňuje, jak produkt vnímají jeho strůjci.

„*Docker helps developers bring their ideas to life by conquering the complexity of app development.*“ (Docker, 2023)

Tato citace popisuje primární účel Dockeru, a to jeho schopnost zjednodušit a tedy zrychlit vývoj aplikací, jejich nasazení a testování a další unifikace a zjednodušování kroků v průběhu celého životního cyklu aplikace. Bez nadsázky lze říci, že Docker je schopen zefektivnit významnou část vývojového cyklu softwarového řešení. (AWS, 2024)

3.1. Co je Docker – bližší vymezení

Termín Docker se dle Martin et al (2018) používá v několika odlišných, avšak spjatých významech.

Nejčastěji se Docker používá ve smyslu spjatém s kontejnery a images a jejich používáním za pomoci konfiguračních souborů Dockerfile nebo docker-compose.yml, které obsahují konfiguraci.

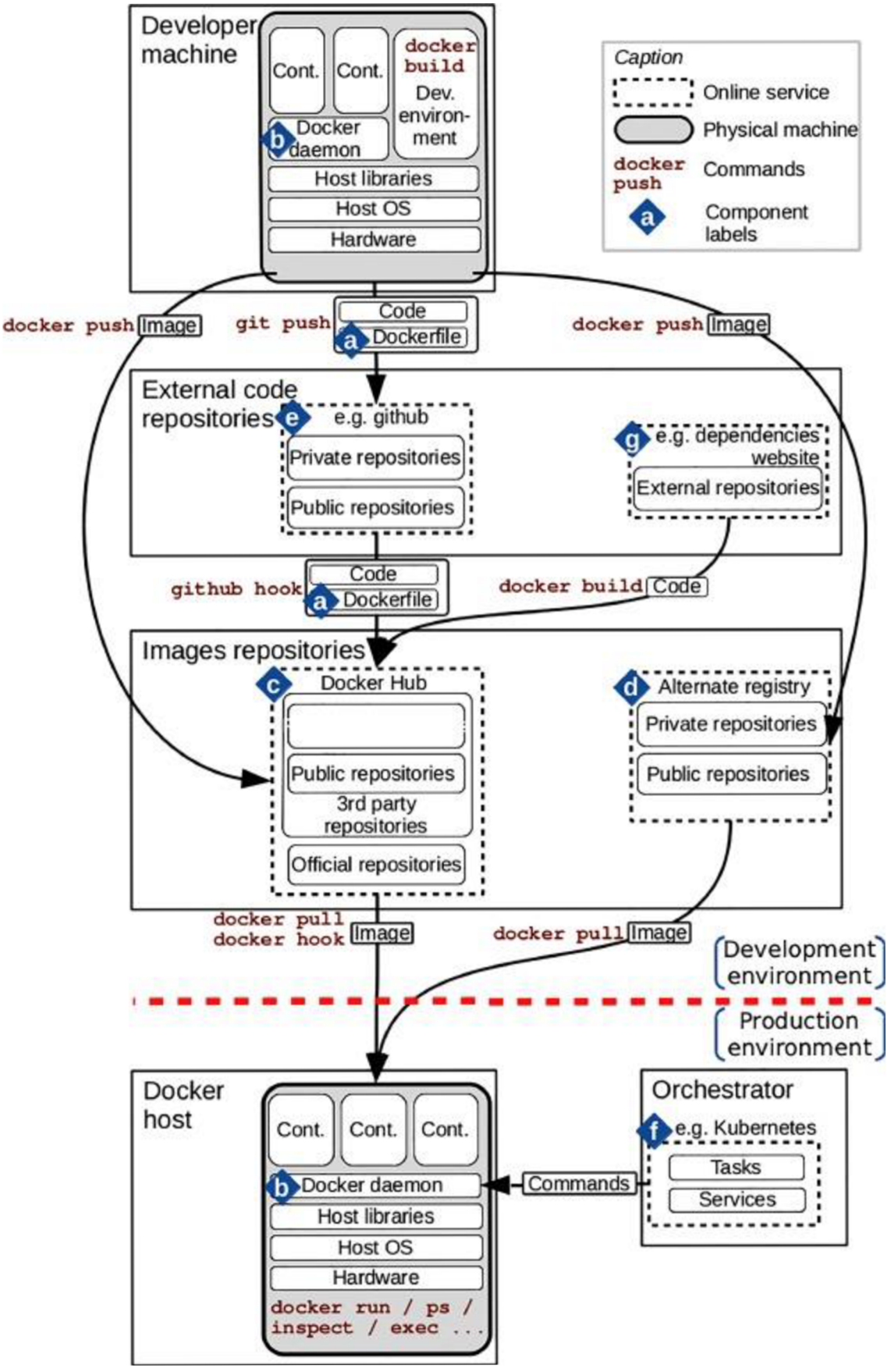
Zároveň se termínem Docker referuje k programům, kterými lze s Dockerem interagovat, popřípadě programům, které vykonají příkazy specifikované uživatelem. Jedná se o Docker Engine neboli také Docker daemon, který je přiblížen v dalších sekcích.

Další význam Dockeru je spjatý s úložišti, která obsahují images jednotlivých aplikací. Tato úložiště se jmenuji registry (plurál registries) a mohou být buď veřejná nebo privátní. Nejznámějším takovým úložištěm je Docker Hub, který je jak privátní, tak primárně veřejné úložiště.

Docker dále odkazuje na proces, kterým se images (pomyslené šablony kontejnerů) sestaví za pomoci konfiguračních souborů a skládáním dalších dílčích images z registries.

Příkladem budiž scénář, ve kterém má image vycházet z nějakého operačního systému, na němž následně poběží aplikace. Tento operační systém bude do výsledného image stažen jako výchozí vrstva a jedná se o image třetí strany. Zároveň Docker může sloužit jako orchestrační nástroj po dobu životního cyklu kontejnerů.

Ilustrace níže zobrazuje ekosystém Dockeru vztahující se k rozboru výše. Blíže se jednotlivým konceptům věnují části práce.



Obr. 1: Docker ekosystém (Martin et al., 2018)

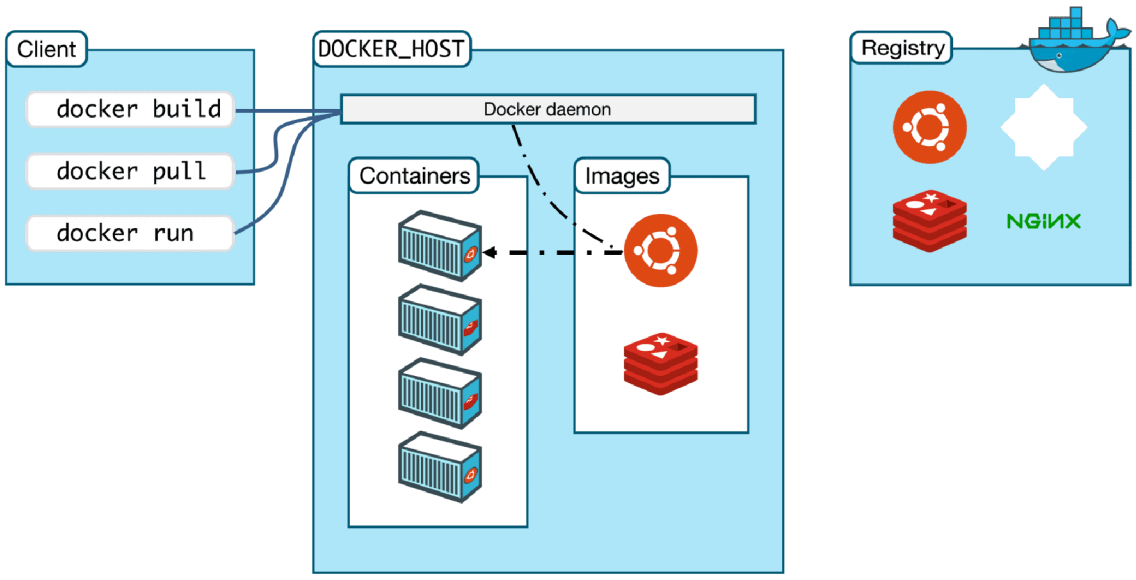
Na schématu výše vidíme ekosystém Dockeru. Schéma znázorňuje vývoj aplikace s využitím Dockeru (body a, b), persistování Docker images v nějakém úložišti (registry) (body c, d) a provoz aplikace (body b, f). Tento ekosystém zobrazující práce s Dockem dává do souvislosti jednotlivé bloky, které jsou běžně označovány jako Docker, avšak refererují pouze k dílčí části ekosystému (Martin et al., 2018).

Blok Orchestrátor v pravém spodním rohu je nepovinný, jednalo by se o obohacení standartního ekosystému Docker kontejnerů.

3.2. Architektura Dockeru

Docker se skládá ze čtyř hlavních komponent – Docker klient a Docker daemon (aka Docker server), image, kontejnery a registry.

Zajímavost: historicky by k těmto komponentám taktéž patřil container runtime, kterým je v případě Dockeru projekt Containerd. Tento projekt byl však z Dockeru vydělen mimo jiné z důvodu rozšíření Kubernetes. Toto vydělení je výhodné z pohledu minimalizace náročnosti na systémové zdroje, je-li potřeba pouze container runtime Osnat, R. (2020).



Obr. 2: Architektura Dockeru (Kumar, 2022)

3.2.1. Docker klient a Docker daemon (server)

Docker klient a Docker daemon jsou jednou ze čtyř hlavních interních komponent Dockeru (Bashari Rad et al., 2017).

Docker klient a Docker daemon můžeme chápat jako dvě různé aplikace, které spolu komunikují přes RESTové API. Běžně jsou obě instalovány na stejném hostu a zatímco uživatel interaguje s Docker klientem, ten na pozadí předává přes API instrukce Docker daemonu (serveru).

3.2.2. Docker registry

Docker registries jsou úložiště imagů. Registry je centrální místo pro správu a uchovávání imagů. Tyto image jsou poté používány ke dvěma hlavním účelům. Prvním je vytváření vlastních imagů, kde tyto existující image tvoří vrstvy nebo-li stavební bloky nově vznikajících imagů. Druhým účelem je možnost standardizovaného a snadného stažení s následujicím provozováním již existujících aplikací. (Aqua, 2022).

Registry může být veřejné nebo privátní. Největším veřejným úložištěm je Docker Hub. Docker registry funguje analogicky například k version control (VCS) platformám typu GitLab v tom smyslu, že jednotlivé image jsou v repositářích, kde je uživatelé s dostatečným oprávněním mohou upravovat, stahovat je a dále s image pracovat. Analogické je dále i to, že registry může být hostované lokálně nebo může být poskytováno jako internetová služba. Příkladem je Azure Container Registry, které se snadno integruje s Azure DevOps (Microsoft Azure, 2024).

3.2.3. Docker Images

Docker image je připravený objekt, který obsahuje všechny potřebné soubory a konfigurace pro spuštění určité aplikace. Image obsahuje vše, co je potřeba pro vytvoření kontejneru, včetně operačního systému, knihoven, zdrojových kódů aplikace a dalších závislostí. Image slouží jako šablona pro vytváření kontejnerů.

Image, které byly popsány výše, uživatel získává jejich lokálním vytvořením nebo stahováním z registries.

Docker image, tedy šablony pro kontejnery aplikací, mohou být vytvořeny z konfiguračních souborů – Dockerfilů nebo mohou být staženy či být vytvořeny hybridně skládáním. Hybridní skládání znamená, že image základního operačního systému zdockerované aplikace bude stažen z veřejného registry a tento image bude rozšířen o kód aplikace, které na tomto OS v kontejneru poběží (Docker, 2024).

3.2.4. Docker kontejner

Docker kontejner je instance image – například instance běžící aplikace. Kontejner je oddělený od operačního systému hosta a do jisté míry je také oddělen od ostatních kontejnerů, což záleží na předem nakonfigurované úrovni izolace. Kontejner obsahuje všechny potřebné zdroje pro běh dané aplikace, včetně OS.

3.3. Interakce

Interakce s Docker daemonem, který je zodpovědný za vykonávání uživatelem zadaných příkazů, probíhá z pravidla předáváním instrukcí přes API dvěma způsoby. Stejně jako u operačních systémů se jedná buď o příkazovou řádku (Command Line Interface [CLI]) nebo o grafické uživatelské rozhraní (Graphical User Interface [GUI]). Docker daemon je zajímavější z pohledu této práce, neboť různé formy klienstkých řešení pro interakci s Docker daemonem nakonec vyústí ve stejný RESTový API request (Docker, 2024).

Příkladem jsou dvě různé akce z pohledu uživatele – první akcí je uživatelský příkaz „docker ps“ zadaný přes CLI a druhou klik na možnost „Containers“ v GUI Docker Desktopu. Obě tyto akce vedou v GET požadavek na Docker daemona, na API /containers/json, popřípadě s fixní verzí API – například /v1.42/containers/json.

Nejznámějším Docker GUI nástrojem je Docker Desktop, který je však od druhé poloviny roku 2021 zpoplatněn pro organizace od určité velikosti a obratu (Docker, 2023) a objevují se jeho alternativy, jako například Rancher Desktop.

4. Testování software

„*Software testing is the process of evaluating and verifying that a software product or application does what it is supposed to do.* (...)“ (IBM, 2024)

Tato definice testování software a aplikací není příliš detailní, ale vystihuje podstatu testování softwaru.

4.1. Typy testování software

Společnost IBM rozděluje typy testování software následujícím způsobem, který nabízí různé perspektivy na testování software a scénáře, ve kterých je testování žádoucí (IBM, 2024).

- Akceptační testy: Akceptační testy ověřují, zda celý systém funguje, jak má.
- Integrační testy: Integrační testy ověřují, že komponenty systému správně pracují společně.
- Unit testy: Unit testy ověřují, že každá softwarová jednotka funguje podle očekávání. Softwarovou jednotkou může být například funkce, třída nebo rozhraní. Jednotka je nejmenší testovatelná součást aplikace.
- Funkční testování: Kontrola funkcí pomocí emulace business scénářů na základě funkčních požadavků.
- Výkonnostní a zátěžové testování: Testování softwaru při různých pracovních zátěžích. Testování zátěže se používá například k vyhodnocení chování v reálných podmínkách zátěže anebo pro testování maximální zátěže, kterou software vydrží.
- Regresní testování: Regresní testy ověřují, zda nové funkcionality nenarušují nebo nezhoršují funkčnost stávajících.
- Uživatelské testování: Ověřují, jak dobře může zákazník používat systém nebo webovou aplikaci k dosáhnutí cíle.

4.2. Výhody použití Dockeru pro testování

Výhody použití Dockeru pro testování se odvíjí od typu testu a testovaného scénáře. Docker slouží jako virtualizační prvek a také nástroj, kterým lze relativně snadno v různých prostředích připravit standardizované prostředí a scénáře. Docker také implementuje bezpečnostní prvky, které jsou popsané v další kapitole Bezpečnost a může tedy sloužit jako vrstva abstrakce mezi hostitelským prostředím a prostředím, ve kterém bude software testován, tedy prostředí aplikace.

Automatizace testování v Docker kontejnerech

Automatizace je u testování velice žádoucí, neboť šetří čas a zdroje. Automatizovat lze v libovolném rozsahu a od rozsahu a se také odvíjí užitková hodnota použití Dockeru. Protože Docker není jediným automatizačním nástrojem, jeho použití je vhodné tam, kde je Docker nejlepším nástrojem pro danou úlohu. Zatímco u malého Unit testu by byl Docker nadbytečným nástrojem, tak říkajíc „kanón na vrabce“, u většího integračního testu by byl vhodný v kombinaci s dalšími nástroji, jako například Ansible nebo Kubernetes.

Izolované a konstantní prostředí

Následující příklad je Dockerfile, tedy konfigurační soubor, na základě kterého bude sestaven image. V kontejneru, který bude z tohoto image vytvořen, bude spuštěna aplikace se vstupním bodem test_app.py, která bude testována. Spuštění testů by se odvíjelo od jejich formy. V kontejneru by například mohl proběhnout nějaký automatizovaný testovací proces.

```
# Dockerfile

# Základní image s Pythonem v3.9 (již obsahující vlastní OS)

FROM python:3.9


# Přidání pracovního adresáře do kontejneru

WORKDIR /app


# Kopírování aplikace do kontejneru

COPY test_app.py /app


# Instalace závislostí (v tomto případě není potřeba)


# Spuštění testovacího skriptu při startu kontejneru

CMD ["python", "test_app.py"]
```

Celý scénář bychom vykonali příkazem

```
docker build -t app-to-test . && docker run app-to-test
```

Zatímco testovaná aplikace se v čase může měnit, vše ostatní bude zůstávat neměnné.

Rychlost nasazení

Izolované a konstantní prostředí není doména pouze Docker kontejnerů, ale například i zmiňovaných virtuálních strojů. Ovšem rychlost nasazení virtuálních strojů je oproti kontejnerům podstatně nižší, respektive jejich nasazení trvá déle, jelikož Docker kontejnery jsou méně náročné na zdroje viz (Cloud Academy Team, 2023).

5. Bezpečnostní rizika vztahu host - Docker

Podle Zhu & Gehrmann, (2021) a Bui (2015) pozorujeme dramatický nárust užívání kontejnerizačních technologií, zejména potom Dockeru. To vytváří poptávku po bezpečných a efektivních virtualizačních řešeních. Zvýšená efektivita, respektive snížený overhead, znamená v tomto případě kompromis i z bezpečnostního hlediska, kterými se blíže zabývají následující kapitoly.

5.1. Bezpečnostní aspekty kontejnerů

Využití Dockeru obecně snižuje overhead vývojových cyklů a provozování aplikací (například overhead v podobě nákladných I/O operací vzhledem k absenci potřeby emulace, které jsou typické pro virtuální stroje). To se pojí s tendencí provozovat více aplikací na jednom serveru a užší integrací kontejnerů s hostitelským operačním systémem.

Tato užší integrace rozšiřuje attack surface (pomyslný prostor, který je zranitelný), což je důvodem k důkladné analýze rizik (Martin et al., 2018).

Zatímco při používání virtuálních strojů aplikace využívá vymezené zdroje virtuálního stroje, Docker kontejner přistupuje přímo ke zdrojům hosta (využívá LXC-based implementaci, tedy Cgroups a Namespaces pro izolaci) (Soltesz et al., 2007), (Xavier et al., 2013).. Tento rozdíl v přístupu ke zdrojům hostitelského systému znamená riziko pro ostatní aplikace běžící na hostitelském systému stejně jako pro hostitelský systém samotný. Tento předpoklad je ověřen scénáři v praktické části práce.

Martin et al. (2018) s odkazem na analyzované zdroje usuzuje, že bezpečnostním aspektům používání Docker kontejnerů v end-to-end vývojovém cyklu není doposud dostatečně porozuměno s přihlédnutím k diverzitě řešení, která do vývojového cyklu vstupují, ať už se jedná o různé softwarové nástroje, DevOps nástroje nebo řešení poskytovatelů cloudových služeb.

Nicméně kromě výše popsaných pasivních bezpečnostních prvků, které Docker obsahuje, existují i aktivní prvky, které za běhu monitorují chování kontejnerů. Tyto procesy detekují anomálie v chování kontejnerů, například atypickou alokaci zdrojů.

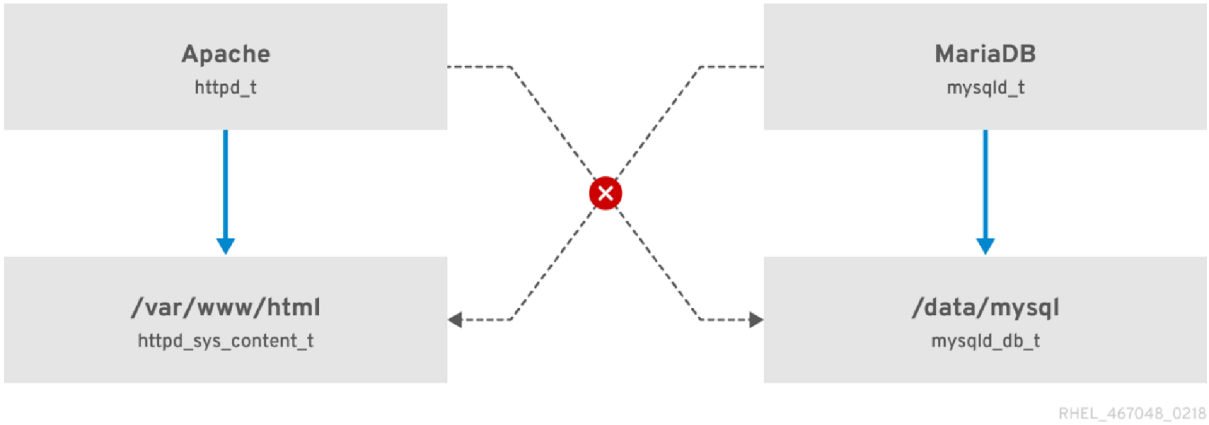
Za zmínku také stojí množství známých zranitelností (CVEs), které je obsaženo v Docker images. V ideálním scénáři by se při objevení nové zranitelnosti (CVE) mělo neprodleně přistoupit k jejímu vyřešení ve zranitelném imagi, což se v praxi často neděje a image zůstávají neupdatované po dlouhou dobu. Tyto zranitelnosti se poté propagují do všech následných imagů, které tento zranitelný image používají (Martin et al., 2018).

5.2. SELinux a AppArmor

SELinux a AppArmor jsou další dva mechanismy na Linuxových systémech, které poskytují vrstvu ochrany pro / před Docker kontejnery. Nejsou však specifické pro použití Dockeru, fungují obecně pro Linuxové procesy. Zároveň, jelikož většina kontejnerů obsahuje Linuxový operační systém, fungují tyto dva mechanismy jak v rámci kontejnerů, tak mezi kontejnery a hostitelským strojem (ComputerNetworkingNotes, 2023).

AppArmor je Mandatory Access Control (MAC) systém pracující s profily jednotlivých programů, jehož hlavním úkolem je udržet zdroje alokované danému profilu v rámci definovaných hodnot. Ve vztahu k Docker kontejnerům to znamená, že pokud si kontejner alokuje více zdrojů, než má daným profilem povoleno, AppArmor zajistí nápravu (Ubuntu, 2024).

Na druhé straně SELinux je mechanismus, který na Linuxových systémech omezuje přístup primárně ke zdrojům typu soubory, linky, složky a podobně. SELinux definuje přístupová oprávnění k těmto zdrojům a dohlíží na jejich dodržování. V případě narušení vynutí nápravu. SELinux má 3 konfigurovatelné módy. Nejstriktnější enforcing mód, laxnější mód permissive a disabled mód, ve kterém je SELinux nečinný. V praxi může SELinux fungovat následovně:



Obr. 3: SELinux mechanismus (Red Hat, 2024)

Ze schématu na obrázku č. 3 je patrné, že jednotlivé procesy mají jasně definovaná přístupová práva tak, jak je u architektury webové aplikace zamýšleno (tj. Webový server nepřistupuje přímo k souborům databáze, a naopak databáze nepřistupuje ke statickým souborům webového serveru). Dodržování pořádku podle schématu výše zaznamenává a vynucuje právě SELinux v módu enforcing.

5.3. Best practices pro zabezpečení Docker kontejnerů

Best practices zabezpečení Docker kontejnerů, jak v rámci daného kontejneru, tak mezi daným kontejnerm a hostem, je komplexní záležitost. Vzhledem k mnoha faktorům, které vstupují do celkové architektury, jde spíše o snahu minimalizovat attack surface, který během procesu vzniká a dále o maximalizaci připravenosti na scénář, ve kterém bude bezpečnost procesu nějakým způsobem narušena (Zhu & Gehrmann, 2021), (Chamoli & Sarishma, 2021).

Mezi high-level oblasti, které je vhodné v tomto kontextu identifikovat a zvážit, patří:

- Používání oficiálních a dostatečně důvěryhodných imagů
- Pravidelné aktualizace hostitelského stroje i klientů
- Monitoring chování a událostí a dostatečně detailní logování
- Správně nastavená privilegia podle pravidla nejnižší dostačující
- Aktivní izolační mechanismy popsané výše, jako například AppArmor a SELinux
- Správně nastavené síťové bezpečnostní prvky
- Implementované zálohovací mechanismy a připravenost na krizový scénář

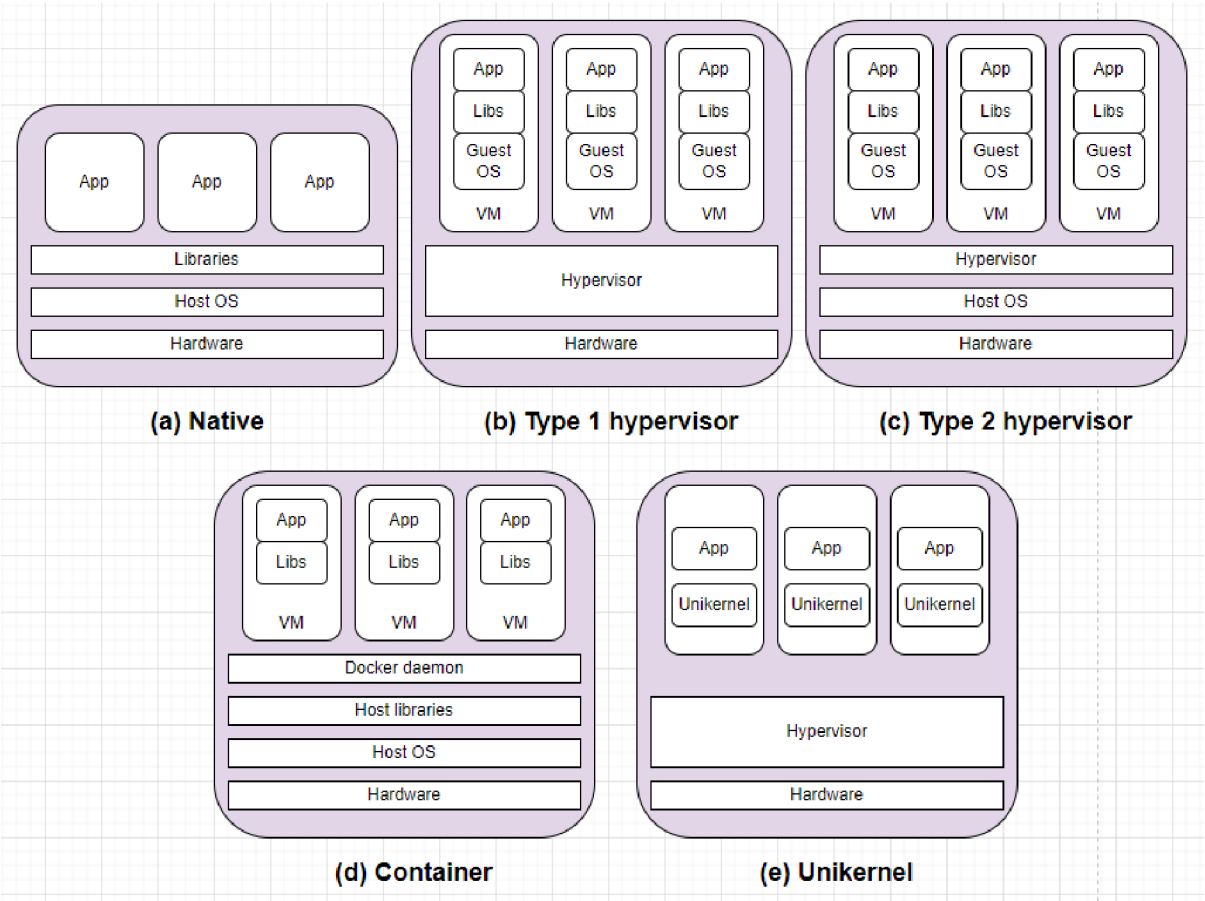
- Obecná pravidla bezpečnosti při používání aplikací jako například vhodné přístupové údaje, znalost problematiky a tak podobně.

6. Porovnání Dockeru s alternativami

Množství publikací na téma Docker začíná způsobem „*with the rapid development of cloud computing technology, containerization technology has drawn much attention from both industry and academia*“ (Zhu & Gehrmann, 2021)

Lze tedy usuzovat, že téma kontejnerizačních technologií je velmi aktuální. Existuje množství řešení, která slouží tomuto účelu. Níže jsou porovnána do jisté míry konkurenční, tedy zastupitelná, řešení, konkrétně virtuální stroje, Docker kontejnery a Podman kontejnery.

Prvně jsou porovnány dvě technologicky vzdálenější technologie, Docker kontejnery a virtuální stroje. Poté budou porovnána dvě kontejnerizační řešení, Docker a Podman.



Obr. 4: Architektury provozu aplikací ve vztahu k hostitelskému OS (Martin et al., 2018)

Obrázek č. 4 zobrazuje různé přístupy k provozování aplikací. V architektuře (a) je zobrazen klasický přístup, kdy aplikace běží přímo na hostitelském systému. Architektury (b) a (c) zobrazují architekturu virtuálních strojů využívajících hypervisorů. Hypervisor (vrstva abstrakce emulující zdroje hosta pro potřeby virtuálních strojů) může běžet buď přímo nad hardwarovými zdroji hosta (b) nebo nad operačním systémem hostitelského stroje (c). Architektura (d) zobrazuje fungování kontejnerizovaných aplikací. Tato architektura je platná primárně pro Docker kontejnery. Podman, což je kontejnerizační alternativa Dockeru, by byl nejvíce podobný právě architektuře typu (d), ale je daemon-less a s kontejnery pracuje formou vlákna v uživatelském prostoru. Architektura (e) zobrazuje Unikernel, další typ

kontejnerizačního přístupu, kdy kontejnery nesdílí kernel operačního systému hosta, ale mají vlastní knihovny, jakýsi minimální operační systém, kterým jim umožňuje fungovat přímo nad hypervisorem (Aleksic, 2023). Ačkoli je Unikernel z pohledu funkčnosti a účelu konkurentem námi porovnávaných kontejnerizačních technologií, je méně rozšířen a kontejnerizace aplikace v Unikernelu je poměrně složitá v porovnání s Dockerem a nebude v této práci dále uvažována.

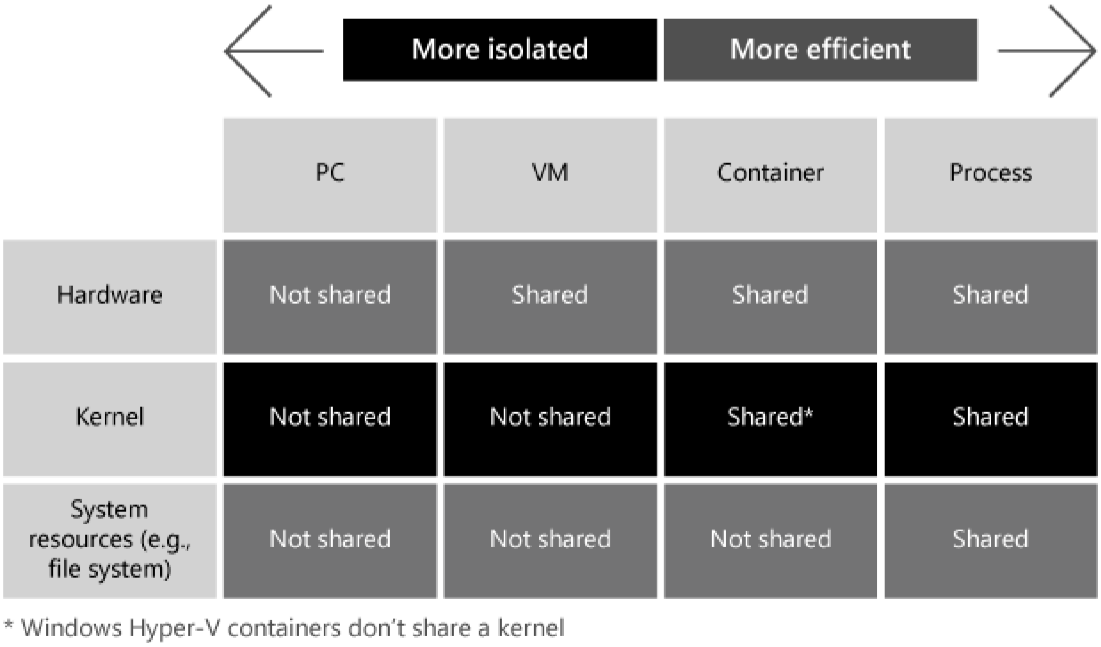
Docker versus virtuální stroje

Docker i virtuální stroje řeší podobný problém a to replikovatelnost, izolaci a unifikaci prostředí, ve kterém běží aplikace. Některé z populárních virtualizačních technologií na bázi virtuálních strojů jsou technologie Hyper-V, Virtual Box nebo Azure VMs (Engine Yard Team, 2022).

Obě architektury přidávají vrstvu ochrany z pohledu hostitelského systému, avšak tato bezpečnostní vrstva, odvíjející se od rozdílných architektur, je nestejná. V případě virtuálních strojů je tato vrstva silnější, neboť za pomoci hypervisoru jsou zdroje hostitelského stroje podle definice VM alokovány a emulovány, zatímco Docker přistupuje ke zdrojům přímo přes hostitelský systém a využívá dynamické alokace (Priya, 2022).

Virtuální stroje mají vlastní operační systém a jsou nezávislé na operačním systému hosta. O to jsou ale náročnější na zdroje. S tím se pojí i rychlost startování a runtime operací. Docker, jako near-native lightweight kontejnerizační aplikace, je obecně rychlejší díky absenci vrstev, přes které runtime operace musí projít.

Bezpečnost těchto řešení se odvíjí mimo jiné od síly vrstvy abstrakce, dělící hostitelský operační systém od kontejneru/virtuálního stroje. Tento vztah platí oběma směry, tj z kontejneru/virtuálního stroje směrem k hostu a z hostu směrem do kontejneru/virtuálního stroje. Objeví-li se například zranitelnost hostitelského operačního systému, jsou Docker kontejnery zranitelnější, nežli virtuální stroje z důvodu užší integrace a naopak. Další citelný rozdíl je v replikovatelnosti těchto řešení. Docker kontejnery lze snáze a rychleji replikovat s využitím méně zdrojů (Priya, 2022), (Engine Yard Team, 2022).



Obr. 5: Matrix sdílení zdrojů různými řešeními (Wwlpublish, 2024)

Obrázek č. 5 znázorňuje kompromis domény kontejnerů. Obecně lze říci, že čím více je objekt vydělen od hostitelského systému, tím je řešení bezpečnější a zároveň těžkopádnější.

Ne vždy je výběr pouze otázkou náročnosti na zdroje nebo bezpečnosti. Jednotlivé přístupy mají další charakteristické rysy, které mohou být vyžadovány. Například Docker kontejnery, ačkoli mohou vycházet například z imagů Windows Server 2022, neposkytnou uživatelům grafické uživatelské rozhraní (Wwlpublish, 2024).

Zajímavostí je, že u Docker kontejnerů běžících na Windows OS (Enterprise 10, Enterprise 11, Pro 10, Pro 11, Windows Server 2019 a 2022) lze specifikovat míru izolace od hostitelského OS. Lze specifikovat například izolační mód hyperv (docker run -it –isolation=hyperv ...), díky kterému jsou kontejnery izolovány stejně, jako VM provozovaná přes Hyper-V. Tento mód je defaultní pro Docker kontejnery na Windows Enterprise a Pro (10, 11) (Wwlpublish, 2024), (Sanner, 2022).

Docker versus Podman

Tyto dvě kontejnerizační technologie jsou do značné míry zastupitelné. Z pohledu vzdálenosti od jádra hostitelského systému jsou stejně izolované a podobně náročné na zdroje. Liší se však svojí vnitřní architekturou. Zatímco Docker používá výše popsaný Docker daemon, který má na starosti interakci se zdroji hostitelského systému a interakci s kontejnery, Podman je daemon-less a kontejnery, kterým se v kontextu Podmana říká pody, vytváří přímo jako uživatelské podprocesy. To znamená, že stejně jako Docker dokáže provozovat root-less kontejnery, tedy kontejnery pod jiným uživatelem, než je root, což snižuje attack-surface daného kontejneru.

Podman v poslední době získává na popularitě, jak toto řešení začíná být vyzrálé. Za zmínku stojí kompatibilita uživatelských příkazů s těmi pro Docker. Dále možnost používání imagů Dockeru díky společnému OCI standardu. Podman, jak jeho jméno napovídá, směřuje více k technologii Kubernetes, kdy v architektuře používá podobné rysy a také jeho zdroje lze snadněji převádět do Kubernetes zdrojů, kdy Podman sám nabízí uživateli možnost generování Kubernetes šablon na základě imagů (Gamela, 2023).

7. Praktická část

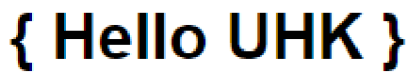
V praktické části této bakalářské práce bude vytvořena webová aplikace, která poběží společně s webovým serverem ve dvou Docker kontejnerech. Webová aplikace bude napsána v Django frameworku, což je rozsáhlá knihovna pro vývoj webových aplikací v pythonu. Jako webový server bude použit Nginx.

Technologie, ve které bude webová aplikace napsána, není pro účel této bakalářské práce podstatná. Bude popsáno, jak byla aplikace provozována ve formě kontejneru a tento projekt dvou kontejnerů bude sloužit jako demonstrativní prostředí pro popis základních principů Dockeru a pro následné analýzy. Zdrojový kód této aplikace je dostupný na adrese https://github.com/Klexus1/hello_world_uhk.git.

Aplikace bude provozována ve třech prostředích. Prvním bude virtuální stroj (VM) s OS Kali Linux provozovaný na Windows OS přes Oracle Virtual Box. Druhým prostředím bude Linuxový server v AWS cloudu. Třetím prostředím je Docker na OS Windows za použití Docker Desktopu. Na dvou z těchto prostředích budou provedeny zátěžové testy a srovnání jejich dopadů na obě prostředí.

7.1. Webová aplikace & Webový server

Jako vzorový software, který může být provozován a testován v Dockeru, byla vybrána webová aplikace. Tato aplikace je napsána v Django frameworku, tedy v pythonu. Konkrétně v pythonu 3.10, Django 3.2.9. Aplikace má pouze landing page s jednoduchým nadpisem.



Obr. 6: Landing page webové aplikace

Tato aplikace je pojmenována hello_world. Aplikace je za webovým serverem, kterým je pro tento účel Nginx.

7.2. Příprava prostředí

7.2.1. Windows & Docker Desktop

Aby mohla být takováto aplikace spuštěna, je třeba mít nainstalovaný Docker. Pro OS Windows 10 je vhodné nainstalovat Docker Desktop. Po nainstalování Docker Desktopu je

Docker dostupný v cestě pod jménem docker. To lze ověřit příkazem „docker –version“, který vrátí odpověď, v tomto případě

[Docker version 20.10.14, build a224086]

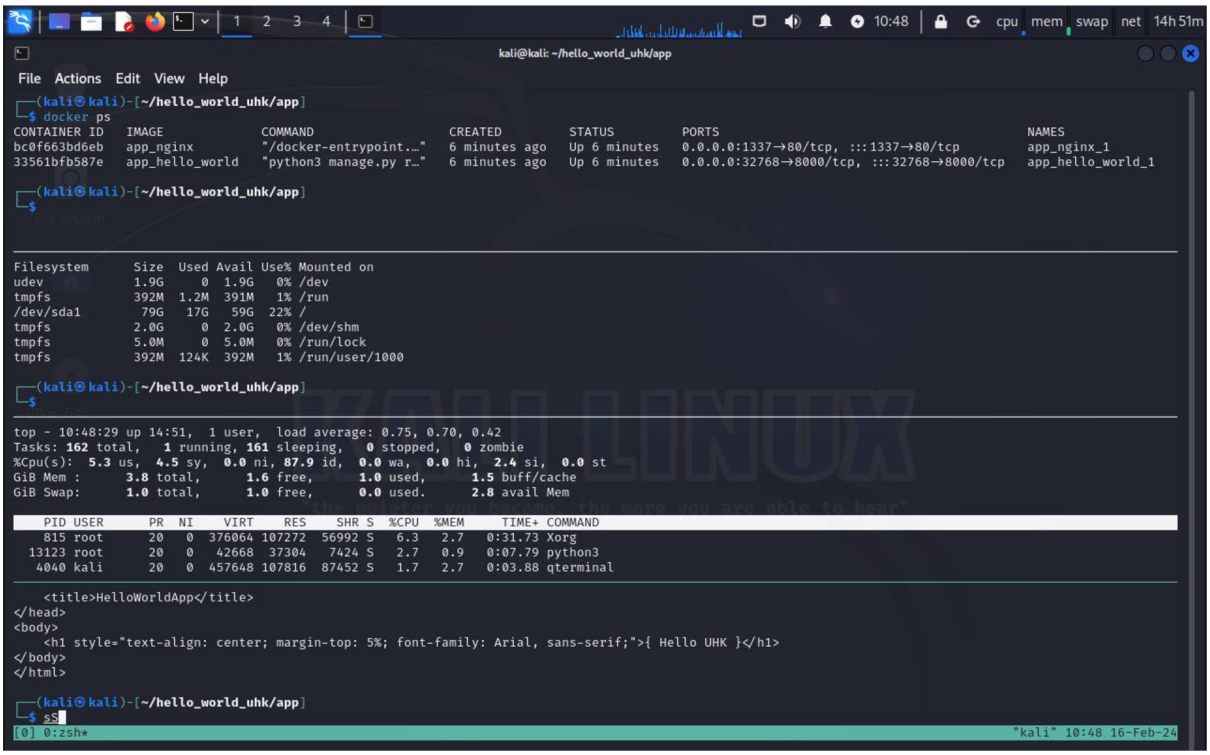
Pokud by se Docker instaloval na Linuxovém OS, bude se instalace lišit podle daného OS. Návod na instalaci je k nalezení na internetu a je taktéž na stránkách Docker docs na adrese https://docs.docker.com/engine/install/. Pro konkrétní Linux OS potom na podstránce, například https://docs.docker.com/engine/install/ubuntu/. Výsledkem by však na všech OS měla být přítomnost nástroje Docker a možnost práce s kontejnery.

7.2.2. Linux VM

Jako Linuxová VM byl zvolen Kali Linux ve verzi 2023.4 běžící v Oracle Virtual Box, který běží na Windows 10 Pro, build 10.0.19044. VM byly přiděleny 4GB RAM, 1 CPU a dynamický storage.

Pro zprovoznění aplikace na tomto serveru je třeba vykonat následující kroky:

1) sudo apt update && sudo apt upgrade
2) git clone <repository_url> (v tomto případě veřejný repozitář https://github.com/Klexus1/hello_world_uhk.git
3) sudo apt install docker.io
4) apt install docker-compose
5) sudo groupadd -f docker && sudo usermod -aG docker $USER && newgrp docker
6) cd hello_world_uhk/app && docker-compose up -d



Obr. 7: Stav VM během provozování webové aplikace (Detail příloha 1)

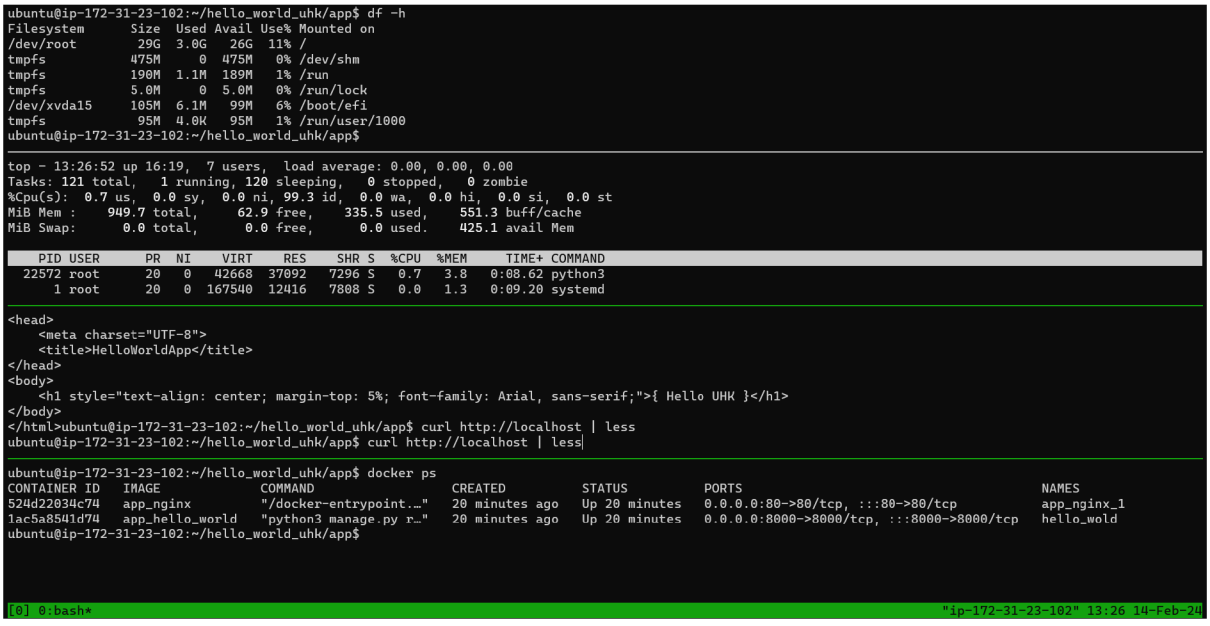
Na obrázku č. 7 výše můžeme vidět stav VM poté, co na ní byl nainstalován docker a rozběhnut projekt s webovou aplikací a webovým serverem. Můžeme vidět, že VM využívá přes 2 GB RAM ze čtyř přidělených a že kontejnery jsou ve stavu Up a webová aplikace, respektive webový server odpovídá na definovaném portu 1337.

7.2.3. Linux Server

Linux server bude v tomto případě AWS t2.micro compute instance. Jedná se o standartní VPS server s operačním systémem Ubuntu jammy 22.04. Server je vytvořen přes webové rozhraní AWS s 30 GB SSD, 1 GB RAM, 1 CPU. Pro zprovoznění aplikace na tomto serveru je třeba vykonat následující kroky:

1. sudo apt update && sudo apt upgrade
2. git clone <repository_url> (v tomto případě veřejný repozitář https://github.com/Klexus1/hello_world_uhk.git
3. apt install docker.io
4. apt install docker-compose
5. sudo groupadd -f docker && sudo usermod -aG docker $USER && newgrp docker
6. cd hello_world_uhk/app && docker-compose up -d

Po těchto krocích máme běžící aplikaci na serveru.



Obr. 8: Stav serveru během provozování webové aplikace (Detail příloha 2)

Obrázek 8 výše zobrazuje stav serveru poté, co na něm byl nainstalován Docker a zprovozněna webová aplikace s webovým serverem ve formě kontejnerů. Scénář je totožný s provozováním tohoto projektu ve VM.

Oproti stavu VM po zprovoznění projektu zde vidíme dva hlavní rozdíly. Jeden čistě konfigurační a to ten, že aplikace, respektive webový server poslouchá na portu 80. Druhý rozdíl je zde ve využité paměti. Vidíme, že server využívá asi 900 MB RAM oproti 2GB, jak tomu bylo v prvním případě, z čehož je více jak polovina využívána pro buffer/cache. Aplikace je v klidovém režimu. Servery jako je tento jsou optimalizované pro podobné projekty mimo jiné minimalizací podpůrných nástrojů, jakými je například GUI nebo určité aplikace tak, aby byl provoz aplikace finančně optimalizován.

7.3. Docker & Docker Compose

Aplikace hello_world i Nginx běží v Dockeru. Obě jsou provozovány za použití společného docker-compose.yml souboru a každá má svůj Dockerfile. Zmíněné soubory vypadají následovně:

docker-compose.yml

```
services:
  hello_world:
    build: ./hello_world
    ports:
      - '8000'


  nginx:
    build: ./nginx
    ports:
      - '1337:80'
    depends_on:
      - hello_world
```

Dockerfile (hello_world app)

```
FROM --platform=$BUILDPLATFORM python:3.10-alpine
WORKDIR /app
```

```
COPY requirements.txt /app
RUN pip3 install -r requirements.txt --no-cache-dir


COPY . /app


ENTRYPOINT ["python3"]
CMD ["manage.py", "runserver", "0.0.0.0:8000"]
```

Dockerfile (Nginx)

```
FROM nginx:latest


RUN rm /etc/nginx/conf.d/default.conf
COPY nginx.conf /etc/nginx/conf.d
```

Aplikace jsou následně spuštěny příkazem

```
docker compose up -d
```

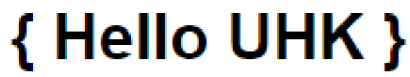
ze složky, ve které se nachází soubor docker-compose.yml. Výsledkem jsou běžící aplikace v Docker kontejnerech. Příkaz

```
docker ps
```

vrací informace o běžících kontejnerech.

```
CONTAINER ID   IMAGE           COMMAND                  CREATED         STATUS         PORTS                     NAMES
075ae94374e8   app_nginx       "/docker-entrypoint.…"   19 minutes ago  Up 19 minutes  0.0.0.0:1337->80/tcp      app_nginx_1
5de242a2db0f   app_hello_world "python3 manage.py r…"   19 minutes ago  Up 19 minutes  0.0.0.0:55913->8000/tcp   app_hello_world_1
```

Jelikož webový server Nginx poslouchá v kontejneru na portu 80, ale na tento port podle definice ze souboru docker-compose.yml dopadají požadavky skrz vystavený port 1337, provolání adresy localhost:1337 nám opět vrací landing page webové aplikace.



Obr. 9: Landing page webové aplikace

7.4. Provozování webové aplikace – simulace neúměrné zátěže

Tato kapitola se zabývá provozováním Docker kontejnerů na různých prostředích. Z pohledu vývojáře se nemusí jednat o rozdílná prostředí. Prostředí jsou zdánlivě stejná v tom smyslu, že na obou běží Docker a kontejnery aplikace. S aplikacemi se zachází stejně a stejná je i výsledná zkušenost uživatele. Tato sekce se zabývá provozováním kontejnerů v detailnějším pohledu a to v kontextu testování nedůvěryhodného softwaru. Pro tento scénář je zásadní vztah hostitelského OS/VM a kontejnerů.

Pro následující dva scénáře, kterými jsou provozování kontejnerů ve VM a provozování kontejnerů přímo na Linuxovém hostu, se zaměříme na přístup kontejnerů ke zdrojům hosta. V tuto chvíli nebudeme uvažovat emulační vrstvu hostitelského OS a VM na něm běžící, ale zaměříme se primárně na přístup kontejnerů ke zdrojům hosta.

Provedeme zátěžový test, při kterém aplikace začne využívat neúměrné množství zdrojů. Konkrétně neúměrné množství paměti tak, že zcela vyčerpá hostitelský systém. Uvažujme tyto testy v následujícím real-world scénáři: Ve firmě běží více Docker kontejnerů na stejném hostu a sdílí zdroje. Vytěžení zdrojů hosta jedním z kontejnerům má dopad i na ostatní kontejnery.

Mohli bychom simulovat nadměrnou zátěž (stress test) s externím původem, jako například velké množství dotazů uživatelů. Takovýto test bychom mohli provést například za pomoci knihovny Locust https://locust.io/. Bylo však zvoleno zatěžování aplikace aplikací samotnou. Takto lze simulovat například chybu v kódu nebo neočekávané spuštění těžby kryptoměny provozovaným softwarem.

7.4.1. Zátěžový test webové aplikace na Linux Serveru

Při spuštění zátěžového testu v kontejneru, který běží přímo na linuxovém serveru, dojde k narušení služby (Denial of service)

```
CONTAINER ID   NAME          CPU %     MEM USAGE / LIMIT     MEM %     NET I/O           BLOCK I/O         PIDS
0fca25d70d49   hello_wold    1.65%     526.9MiB / 949.7MiB   55.48%    4.24kB / 5.09kB   4.07MB / 254kB    4
c88c53508626   app_nginx_1   0.00%     2.387MiB / 949.7MiB   0.25%     8.25kB / 7.25kB   69.6kB / 4.1kB    2
```

Obr. 10: Výstupy [docker stats] při zátěžovém testu - maximální vytížení

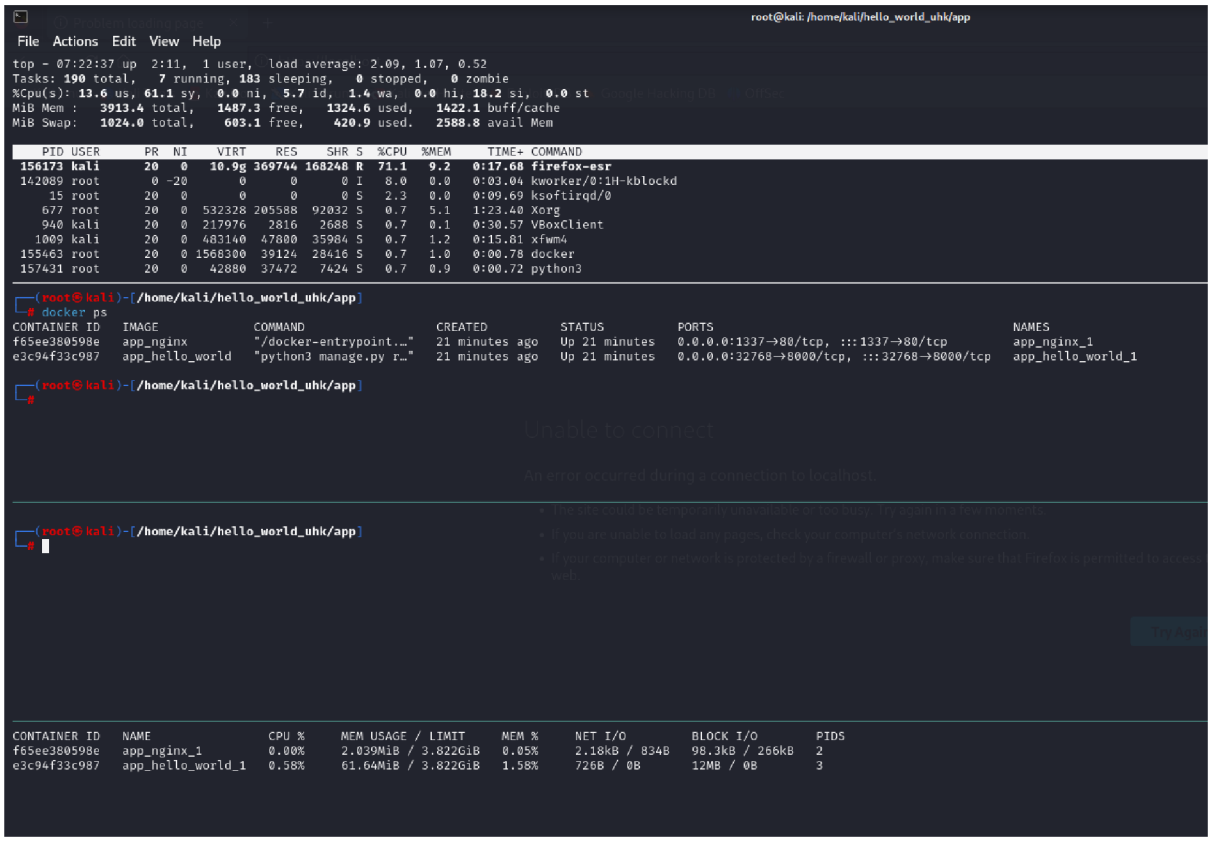
Obr. 11: Odpověď serveru během zátěžového testu

Kromě narušení dané služby je narušený celý chod serveru. Maximální paměť, kterou si webová aplikace během zátěžového testu alokovala, dosáhla 526 MB z celkových 950 MB Ram (Obr. 10). Server se stává irresponzivním, což má dopad i na ostatní služby běžící na tomto serveru (Obr. 11). Příkladem takové služby je služba sshd, která přestane akceptovat nová ssh připojení k serveru.

7.4.2. Zátěžový serveru test webové aplikace na Linux VM

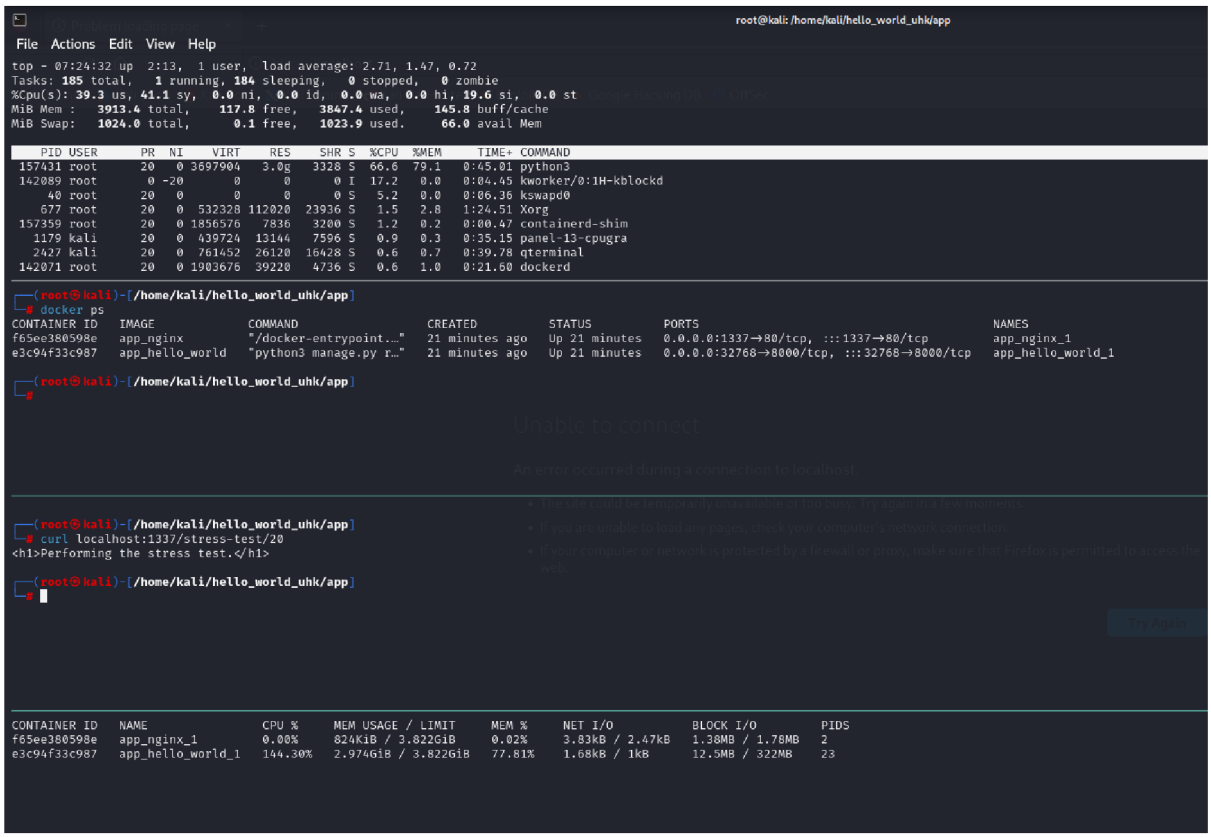
Nyní provedeme stejný zátěžový test ve VM (Kali Linux, 4GB RAM, 1CPU). Jelikož VM má alokované zdroje oddělené od OS hosta, kterým je v tomto případě Windows 10 Pro build 19044, vycházíme z následujícího předpokladu. Při vyčerpání zdrojů VM Docker kontejnerem se stane VM irresponzivní bez dopadu na hostitelský OS Windows.

Před spuštěním testu je stav VM následujicí



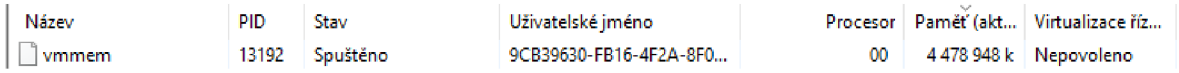
Obr. 12: Klidový stav linuxové VM (Detail příloha 3)

Kontejner aplikace si v klidovém režimu alokuje 61 MB RAM a VM zbývá necelých 1,5 GB volné RAM před spuštěním testu. Po spuštění zátěžového testu je situace následující



Obr. 13: Stav linuxové VM během vrcholu zátěžového testu (Detail příloha 4)

Obr. 13 zachycuje stav VM těsně před tím, než se VM stává irresponsivní. Z celkových 3.9 GB RAM, se kterými VM pracuje, si aplikace alokuje 2,97 GB (77,8%). Ačkoli se VM stává irresponzivní, hostitelský OS Windows není ovlivněn. VM spořádaně vytěžuje pouze ty zdroje, které ji byly alokovány.



Obr. 14: Zdroje využívané VM ve vrcholu zátěžového testu

Obrázek č. 14 zachycuje využívání zdrojů Windows procesem vmmem, který reprezentuje využívané zmíněnou VM. Ačkoli je proces vmmem zastřešovacím procesem pro sledování zdrojů využívaných virtualizací na daném zařízení (Chen, 2019), v tuto chvíli na Windows host OS neběží jiné virtuální stroje ani virtualizační nástroje (jako Docker, WSL, ..), a lze tedy pojit zdroje reportované procesem vmmem s testovanou VM.

Pokud bychom přemýšleli v kontextu testování software (třetích stran), v tomto scénáři by testovaný software neovlivnil ostatní služby, které host klientům poskytuje. Nicméně tento scénář je velmi specifický a limity na maximální využité zdroje lze aplikovat i na kontejnerová řešení.

7.5. Test bezpečnosti – nebezpečný Docker image

Tato kapitola představí nedodržování doporučených postupů v praxi. Konkrétně proč je důležité prověřovat Docker image třetích stran.

Následující test bude vycházet ze scénáře, ve kterém kontejner namountuje root hostitelského souborového systému. Tato akce sama o sobě nezpůsobí žádnou škodu. Primárním úmyslem nemusí být poškození hosta, k této situaci může dojít z důvodu neznalosti nebo neopatrnosti. Nicméně pokud má kontejner přístup k citlivým částem hostitelského OS, může to mít fatální dopady.

Provedeme následující test. Kontejner explicitně mountuje prostor, ve kterém se nachází mimo jiné privátní klíč uživatele pro přístup na server. Pokud má kontejner mechanismus, jak tuto informaci předat útočníkovi, dochází ke kompromitaci hosta.

7.5.1. Příprava image

Do docker-compose.yml souboru aplikace je přidána následující instrukce

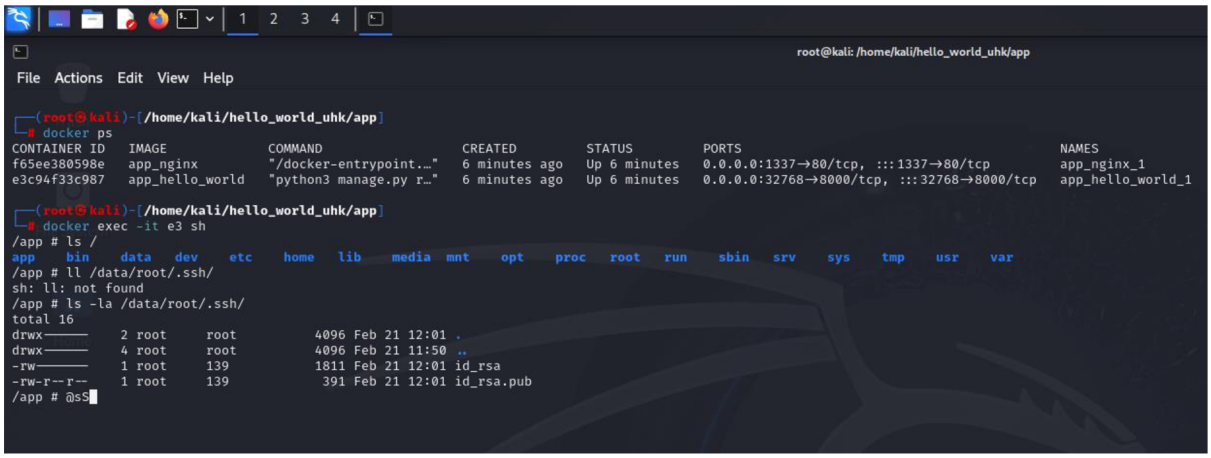
```
volumes:
  - /:/data
```

takže docker-compose.yml nyní vypadá takto.

```
services:
  hello_world:
    build: ./hello_world
    ports:
      - '8000'
    volumes:
      - /:/data


  nginx:
    build: ./nginx
    ports:
      - '1337:80'
    depends_on:
      - hello_world
```

Tato direktiva způsobí, že image hello_world, který bude na základě tohoto docker-compose.yml souboru vytvořen, bude sloužit jako šablona pro kontejnery, které budou do složky /data uvnitř kontejneru mountovat root souborového systému hosta.

Výsledek vypadá takto



Obr. 15: FS hostitelského OS je plně k dispozici kontejneru (Detail příloha 5)

Na obrázku č.15 vidíme stav souborového systému (FS) kontejneru, který byl vytvořen z výše zmíněného image hello_world. Ve složce /data v kontejneru je namountován root FS hostitelského OS, a jelikož kontejner je provozován docker-daemonem, který běží pod uživatelem root, běží i tento kontejner s oprávněními tohoto uživatele. To prakticky znamená plný přístup k souborům v takto namountovaném FS.

Nyní, pokud by tento kontejner odeslal tyto klíče na útočníkovu adresu, je server kompromitován. Při spouštění různých kontejnerů třetích stran navíc standartně nemáme přístup k Dockerfilu / docker-compose souboru, kde bychom zkontrolovali budoucí mounty. Tuto informaci nezískáme ani inspekcí image (například příkazem „docker inspect <image_id>“). Navíc se nejedná o zranitelnost tak, jak jí rozumějí scannery zranitelností, takže není detekována.

Příklad používající trivy vulnerability scanner: https://trivy.dev/

Obr. 16: Vulnerability scan image hello_world (Detail příloha 6)

Na obrázku č. 16 výše vidíme sken vulnerabilit image. Trivy CVE skener reportuje několik CVE, které jsou pro tento image relevantní. Z analýzy image můžeme vidět, že nástroj neodhaluje tuto anomálii v mountech a ukazuje uživateli pouze CVE záznamy vztahující se k softwaru třetích stran. I proto je doporučené používat pouze image důvěryhodných třetích stran.

7.6. Rootless Docker

Standartní mód, ve kterém Docker užívá práv root uživatele, bývá označován jako rootful. Docker daemon může být spuštěn i v takzvaném rootless módu, což znamená, že daemon běží pod jiným uživatelem, než je root. Tento mód zmírňuje potenciální zranitelnost daemona i kontejnerů a zároveň chrání hostitelský OS, pokud by byl některý z aplikačních kontejnerů kompromitován. Rootless kontejnery byla hlavní myšlenka za vznikem Podmana jako alternativního kontejnerizačního řešení. Rootless mód je pro Podman výchozí a jelikož existuje tlak ze strany vývojářů a zákazníků, kteří tento režim preferují, i Docker přidal rootless alternativu (Molloy, 2022), (LHB Community, 2023).

Režim rootless nevyžaduje práva roota ani během instalace Docker démona, pokud jsou splněny předpoklady, kterými jsou pro linuxový server nástroje, jako například newuidmap a newgidmap z balíčku dbus-user-session (Docker Docs, 2024). Při rootless režimu je Docker daemon spouštěn v user namespace uživatele, pro kterého je nakonfigurován. Docker daemon i Docker kontejnery následně běží taktéž v namespace tohoto uživatele a vztahují se na ně omezení práv tohoto uživatele. Takovýto režim prakticky znamená izolaci Docker procesů v rámci daného namespace.

Tento mechanismus má jisté charakteristické rysy a limitace. Mezi charakteristické rysy patří mapování identifikátorů uživatelů a skupin kontejnerů na uživatele, v jehož user

namespace Docker daemon běží. Proto jsou třeba výše zmíněné nástroje (newuidmap a newgidmap). Mezi limitace patří například omezený výčet storage driverů nebo například nemožnost použití AppArmor profilů a určitých síťových prvků.

7.6.1. Instalace rootless Dockeru

Níže následuje praktická ukázka instalace Dockeru a provozování kontejnerů v rootless režimu. Rootless Docker bude nainstalován na Ubuntu 22.04 VM instalovanou přes Oracle Virtual Box na Windows 10 OS. Rootless Docker je instalován takto:

1. apt-get install uidmap -y
2. curl -sSL https://get.docker.com/rootless | sh
3. Přidání dvou proměnných prostředí do ~/.bashrc

   export PATH=/home/$user/bin:$PATH
   export DOCKER_HOST=unix:///run/user/$id/docker.sock

   kde $user je jméno uživatele (zjistitelné přes příkaz whoami), pod kterým instalujeme a $id jeho id (zjistitelné přes příkaz id)

4. systemctl --user start docker

Nyní Unit docker.service běží pod uživatelem, pod kterým byl nainstalován a úspěšně provádíme docker run testovacího kontejneru hello-world.

```
david@david-VirtualBox:~$ systemctl --user status docker
● docker.service - Docker Application Container Engine (Rootless)
     Loaded: loaded (/home/david/.config/systemd/user/docker.service; enabled; vendor preset: enabled)
     Active: active (running) since Tue 2024-03-19 17:29:02 CET; 20min ago
       Docs: https://docs.docker.com/go/rootless/
   Main PID: 11183 (rootlesskit)
      Tasks: 40
     Memory: 86.0M
        CPU: 4.137s
     CGroup: /user.slice/user-1000.slice/user@1000.service/app.slice/docker.service
             ├─11183 rootlesskit --state-dir=/run/user/1000/dockerd-rootless --net=vpnkit --mtu=1500 --slirp4netns-sandbox=auto --s>
             ├─11190 /proc/self/exe --state-dir=/run/user/1000/dockerd-rootless --net=vpnkit --mtu=1500 --slirp4netns-sandbox=auto >
             ├─11203 vpnkit --ethernet /run/user/1000/dockerd-rootless/vpnkit-ethernet.sock --mtu 1500 --host-ip 0.0.0.0
             ├─11216 dockerd
             └─11231 containerd --config /run/user/1000/docker/containerd/containerd.toml

bře 19 17:29:02 david-VirtualBox systemd[826]: Started Docker Application Container Engine (Rootless).
bře 19 17:33:46 david-VirtualBox dockerd-rootless.sh[11231]: time="2024-03-19T17:33:46.887530553+01:00" level=info msg="loading plu>
bře 19 17:33:46 david-VirtualBox dockerd-rootless.sh[11231]: time="2024-03-19T17:33:46.887661465+01:00" level=info msg="loading plu>
bře 19 17:33:46 david-VirtualBox dockerd-rootless.sh[11231]: time="2024-03-19T17:33:46.887673738+01:00" level=info msg="loading plu>
bře 19 17:33:46 david-VirtualBox dockerd-rootless.sh[11231]: time="2024-03-19T17:33:46.890834196+01:00" level=info msg="loading plu>
bře 19 17:33:47 david-VirtualBox dockerd-rootless.sh[11231]: time="2024-03-19T17:33:47.409975960+01:00" level=warning msg="error fr>
bře 19 17:33:47 david-VirtualBox dockerd-rootless.sh[11216]: time="2024-03-19T17:33:47.498373653+01:00" level=info msg="ignoring ev>
bře 19 17:33:47 david-VirtualBox dockerd-rootless.sh[11231]: time="2024-03-19T17:33:47.500622324+01:00" level=info msg="shim discon>
bře 19 17:33:47 david-VirtualBox dockerd-rootless.sh[11231]: time="2024-03-19T17:33:47.503164493+01:00" level=warning msg="cleaning>
bře 19 17:33:47 david-VirtualBox dockerd-rootless.sh[11231]: time="2024-03-19T17:33:47.503531995+01:00" level=info msg="cleaning up>
david@david-VirtualBox:~$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.
```

Obr. 17: Rootless Doocker & hello-world run (Detail příloha 7)

Za pozornost stojí bod 2 instalace:

`curl -sSL https://get.docker.com/rootless | sh`.

Tento příkaz vykoná skript dostupný na této adrese pomocí shell command interpreteru.

Skript tvoří z převážné většiny různé kontroly hostitelského OS, jako například zda-li jsou přítomné nezbytné utility nebo jestli je přítomna očekávaná struktura FS. Následně stahuje potřebné balíčky, kterými jsou docker.tgz a rootless.tgz soubory, které jsou umístěny na určené místo na FS a instalace je dokončena spuštěním skriptu dockerd-rootless-setuptool.sh. Skript dockerd-rootless-setuptool.sh opět provede kontroly hostitelského OS, vytvoří Docker unit file pro systemd a inicialiazuje docker.sock.

7.6.2. Rootless Docker & Hello World UHK aplikace

Opět rozběhneme testovací aplikaci hello_world_uhk kroky:

1) git clone <repository_url> (v tomto případě veřejný repozitář https://github.com/Klexus1/hello_world_uhk.git
2) sudo apt install docker-compose git
3) cd hello_world_uhk/app && docker-compose up -d

```
Successfully built 14faac45c84b
Successfully tagged app_nginx:latest
WARNING: Image for service nginx was built because it did not already exist. To rebuild this image you must use `docker-compose buil
d` or `docker-compose up --build`.
Creating app_hello_world_1 ... done
Creating app_nginx_1       ... done
david@david-VirtualBox:~/hello_world_uhk/app$ docker ps
CONTAINER ID   IMAGE             COMMAND                  CREATED          STATUS         PORTS
    NAMES
6c8a99d102ee   app_nginx         "/docker-entrypoint.…"   10 seconds ago   Up 8 seconds   0.0.0.0:1337->80/tcp, :::1337->80/tcp
    app_nginx_1
465482e81e0b   app_hello_world   "python3 manage.py r…"   12 seconds ago   Up 9 seconds   0.0.0.0:32768->8000/tcp, :::32768->8000/tc
p   app_hello_world_1
```

Obr. 18: Hello World App & Rootless Docker (Detail příloha 8)

Kontejnerizovaná aplikace běží standartně.

```
david@david-VirtualBox:~/hello_world_uhk/app$ docker ps
CONTAINER ID   IMAGE             COMMAND                  CREATED          STATUS         PORTS
   NAMES
6c8a99d102ee   app_nginx         "/docker-entrypoint.…"   9 minutes ago    Up 9 minutes   0.0.0.0:1337->80/tcp, :::1337->80/tcp
   app_nginx_1
465482e81e0b   app_hello_world   "python3 manage.py r…"   9 minutes ago    Up 9 minutes   0.0.0.0:32768->8000/tcp, :::32768->8000/tcp
   app_hello_world_1
david@david-VirtualBox:~/hello_world_uhk/app$ ps aux | grep 465
david       13848  0.0  0.3 1238936 13776 ?       Sl   19:40   0:00 /home/david/bin/containerd-shim-runc-v2 -namespace moby -id 46548
2e81e0bba2ea4a6ad4616f7d3d22be44dcd31f30057c56b7244928eaac4 -address /run/user/1000/docker/containerd/containerd.sock
david       14187  0.0  0.0  20400  2688 pts/0    S+   19:50   0:00 grep --color=auto 465
david@david-VirtualBox:~/hello_world_uhk/app$
```

Obr. 19: Detail procesu kontejneru aplikace (Detail příloha 9)

Pokud se na běžící kontejner podívám blíže příkazem ps aux | grep <id kontejneru>, vidíme, že běží pod uživatelem, pro kterého byl rootless Docker nainstalován a že probíhá mapování id přes user namespace.

7.6.3. Teorie v praxi – nebezpečný Docker image & Rootless Docker

Opět rozšíříme docker-compose.yml soubor o direktivu

```
    volumes:
      - /:/data
```

takže docker-compose.yml nyní vypadá takto.

```
services:
  hello_world:
    build: ./hello_world
    ports:
      - '8000'
    volumes:
      - /:/data


  nginx:
    build: ./nginx
    ports:
      - '1337:80'
    depends_on:
      - hello_world
```

Aplikační kontejner, který bude na základě tohoto docker-compose.yml souboru vytvořen, bude do složky /data uvnitř kontejneru mountovat root souborového systému hosta. Rozdíl je tentokrát v tom, že uživatel root v kontejneru má jen taková práva, jako má uživatel, který kontejner spustil.

Po provedení testu, kde kontejner mountuje root FS hosta, dostáváme následující výsledek.

```
Successfully built 14faac45c84b
Successfully tagged app_nginx:latest
Recreating app_hello_world_1 ... done
app_nginx_1 is up-to-date
david@david-VirtualBox:~/hello_world_uhk/app$ docker ps
CONTAINER ID   IMAGE             COMMAND                  CREATED          STATUS          PORTS
     NAMES
2e142565efa2   app_hello_world   "python3 manage.py r…"   6 seconds ago    Up 5 seconds    0.0.0.0:32769->8000/tcp, :::32769->8000/t
cp   app_hello_world_1
6c8a99d102ee   app_nginx         "/docker-entrypoint.…"   14 minutes ago   Up 14 minutes   0.0.0.0:1337->80/tcp, :::1337->80/tcp
     app_nginx_1
david@david-VirtualBox:~/hello_world_uhk/app$ docker exec -it 2e sh
/app # cd /
/ # ls
app    data   etc    lib    mnt    proc   run    srv    tmp    var
bin    dev    home   media  opt    root   sbin   sys    usr
/ # ls data/
bin         dev         lib         libx32      mnt         root        snap        sys         var
boot        etc         lib32       lost+found  opt         run         srv         tmp
cdrom       home        lib64       media       proc        sbin        swapfile    usr
/ # ls /data/root/
ls: can't open '/data/root/': Permission denied
/ #
```

Obr. 20: Mount rootu FS hosta rootless kontejnerem (Detail příloha 10)

Na obrázku č.20 vidíme, že lze takovýmto kontejnerem mountovat root FS hostitelského OS (/ dir má execute právo pro všechny uživatele). Při pokusu o přístup k souborům, ke kterým uživatel, pod kterým kontejner běží, nemá oprávnění, dostává uživatel root „Permission denied“. Například při pokusu o přístup do složky /data/root odpovídající složce /root hostitelského FS.

Rootless Docker je tedy vhodné rozšíření Dockeru pro účely testování, ale jistě zasluhuje hlubší analýzu a porozumění před tím, než by měl být puštěn na produkčním prostředí. Zajímavostí je, že Podman (kontejnerizační alternativa Dockeru) funguje tímto způsobem ve výchozím stavu.

8. Shrnutí a diskuse výsledků

Teoretická část práce vymezila pojem Docker a kontext, ve kterém se tato kontejnerizační technologie využívá. Architektura Dockeru byla blíže analyzována a rozpadnuta na čtyři hlavní bloky s popisem vazeb mezi těmito stavebními prvky. Testování software bylo představeno jako fáze vývojového cyklu aplikace a jak zapojení Dockeru v této fázi může zefektivnit vývojový cyklus. Lze říci, že při vhodném scénáři lze použití Dockeru vést k optimalizaci nejen HW zdrojů, ale i lidských zdrojů a časové náročnosti testovací aktivity.

Zvláštní kapitola zde byla věnována bezpečnosti při práci s Dockerem, neboť existují prvky, které vstupují do vývojového cyklu aplikace bez větší pozornosti, avšak mohou představovat značné riziko. Konkrétně se jedná například o vrstvy Docker imagů nebo celé image, které mohou být buď kompromitovány nebo představovat bezpečnostní riziko. Ačkoli Docker implementuje řadu bezpečnostních mechanismů v klasickém, takzvaném rootful módu, attack surface zůstává značný, jelikož jednotlivé kontejnery běží s rootovským oprávněním a jedna z nejúčinnějších ochran v tomto módu je nakonec dodržování best practices při práci s Docker zdroji.

V praktické části docházelo především k ověření předpokladů, které vznikly v teoretické části. V první části se jednalo o demonstrativní instalace Dockeru v různých prostředích, provozování webové aplikace sestávající ze dvou kontejnerů. V další části byl proveden zátěžový test na linuxovém serveru a na linuxové VM. Výsledky odpovídaly teoretickým předpokladům, jež vycházely z odlišnosti architektur scénářů. Poslední praktická ukázka sestávala z instalace rootless Dockeru, tedy instalace Dockeru tak, aby běžel pod jiným uživatelem, než je běžný root. Následně byl proveden test, kdy se kontejner pokusil přistoupit k souborům způsobem, jež byl přinejmenším nestandartní a bylo mu v tom zamezeno právě podstatou rootless Docker režimu.

9. Závěry a doporučení

Tato práce je pouhým představením světa Dockeru a konceptů, které se Dockeru blíže či více vzdáleně týkají. Každý jednotlivý koncept, ať už se jedná o kontejnery, vztah hostitelského OS a kontejnerů nebo rozšíření Dockeru, tzv. Rootless Docker, by vyžadoval mnohem hlubší analýzu, aby mu bylo porozuměno.

Tato práce tyto koncepty z praktických důvodů představuje, ale nezabývá se jejich vnitřním fungováním nebo analýzou zdrojového kódu, kterážto by byla nezbytná pro informovaná rozhodnutí v praxi.

Jelikož vznikají kontejnerizační alternativy Dockeru, jako například Podman, na Docker bude vznikat tlak jednak ze strany bezpečnosti (například rootless Docker by mohl být v blízké budoucnosti standardem), tak i ze strany kompatibility různých rozhraní a standardů. Příkladem může být virtualizační standard OCI, který odráží trend dnešní doby, a to že do vývojového cyklu aplikací vstupuje čím dál více nástrojů a jejich kompatibilita rozhraní se stává klíčovým faktorem při jejich volbě.

Z teoretické části také vyplývá, že uživatelé Dockeru si jsou vědomi nedostatečného porozumění bezpečnostním rizikům, které při každodenním používání Dockeru vznikají a apelují na úsilí, které by mělo být tímto směrem vyvíjeno. V praktické části byl této problematice věnován scénář, který při nestandartním mountování FS hosta do kontejneru způsobuje neomezený přístup kontejneru k FS hostitelského OS. Navíc se nejedná o klasickou formu zranitelnosti, neboť tento scénář může být v určitých situacích žádoucí, a tedy nebyl odhalen skenery zranitelností, na které se mnozí spoléhají při analýze imagů.

Praktická část dále ověřila teoretický předpoklad rozdílného vztahu zdrojů hosta a kontejnerů v různých přístupech architektur. Docker kontejnery mají při standartní (tzv. rootful) instalaci v podstatě neomezený přístup ke zdrojům hosta, kterého mohou z různých důvodů učinit irresponzivního, což vede k narušení fungování ostatních služeb, který tento server poskytuje. Doporučení vyplývající z tohoto testu platí pro scénáře, kdy server poskytuje klientům více nezávislých služeb, jako například při provozu více aplikací na jednom serveru. Toto doporučení zní zvážit separaci jednotlivých služeb na různé servery, popřípadě instalaci rootless Docker rozšíření k zajištění maximální kvality služeb z pohledu zákazníka. Každá z těchto možností sebou nese jistá úskalí (optimalizace zdrojů, jiné chování řešení), ale za cenu zvýšené bezpečnosti a zlepšení klientských služeb.

V kontextu testování software je na Docker vhodné nahlížet jako na jednu z cest, která má ovšem značné výhody oproti klasickému testování, kdy je připraveno nové prostředí, sestávající například ze skupiny serverů. Docker, nehledě na typ testů, přináší aplikacím uniformní prostředí, které se nemění v čase a je automaticky vytvořeno na základě konfiguračních souborů. Značné zefektivnění přinese především ve scénářích, kde je aplikace potřeba testovat například na různých operačních systémech nebo se vzájemně se vylučujícími službami nebo verzemi téže služby oddělením aplikačních prostředí formou soběstačných kontejnerů.

10. Seznam použité literatury

Ali, M. (2023) *Mastering docker and Kubernetes for Machine Learning Applications: Unleashing the power of Containerization*, *DataCamp*. Available at: https://www.datacamp.com/tutorial/containerization-docker-and-kubernetes-for-machine-learning (Accessed: 12 March 2024).

Osnat, R. (2020) *A brief history of containers: From the 1970s till now*, *Aqua*. Available at: https://www.aquasec.com/blog/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016/ (Accessed: 12 March 2024).

Entlyft (2024) Docker commands 20.9% market share in virtualization platforms, Entlyft. Available at: https://enlyft.com/tech/products/docker (Accessed: 10 March 2024).

Singh, S. and Singh, N. (2016) 'Containers & Docker: Emerging roles & future of cloud technology', 2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT) [Preprint]. doi:10.1109/icatcct.2016.7912109.

About Docker (2023) Docker. Available at: https://www.docker.com/company/ (Accessed: 10 March 2024).

AWS (no date) What is Docker? | AWS, What is Docker? Available at: https://aws.amazon.com/docker/ (Accessed: 10 March 2024).

Martin, A. et al. (2018) 'Docker ecosystem – vulnerability analysis', Computer Communications, 122, pp. 30–43. doi:10.1016/j.comcom.2018.03.011.

Kumar, A. (2022) Docker Architecture: Docker resource isolation: Lifecycle, Cloud Training Program. Available at: https://k21academy.com/docker-kubernetes/docker-architecture-docker-engine-components-container-lifecycle/ (Accessed: 10 March 2024).

Bashari Rad, B., John Bhatti, H. and Ahmadi, M. (2017) An introduction to docker and analysis of its performance. Available at: https://www.researchgate.net/publication/318816158_An_Introduction_to_Docker_and_Analysis_of_its_Performance (Accessed: 10 March 2024).

Aqua (2022) Docker registries, Aqua. Available at: https://www.aquasec.com/cloud-native-academy/docker-container/docker-registry/ (Accessed: 10 March 2024).

Microsoft Azure (2024) Azure Container Registry, Microsoft Azure. Available at: https://azure.microsoft.com/en-us/products/container-registry (Accessed: 10 March 2024).

Docker Documentation (2024) Create a base image, Docker Documentation. Available at: https://docs.docker.com/build/building/base-images/ (Accessed: 10 March 2024).

Docker (2024) Containers, Docker Engine API v1.43 reference. Available at: https://docs.docker.com/engine/api/v1.43/#tag/Container (Accessed: 10 March 2024).

Docker (2023) Docker faqs, Docker. Available at: https://www.docker.com/pricing/faq/ (Accessed: 10 March 2024).

IBM (2024) What is software testing and how does it work?, IBM. Available at: https://www.ibm.com/topics/software-testing (Accessed: 10 March 2024).

Team, C.A. (2023) Docker vs. Virtual Machines: Differences you should know, Cloud Academy. Available at: https://cloudacademy.com/blog/docker-vs-virtual-machines-differences-you-should-know/. (Accessed: 10 March 2024).

Zhu, H. and Gehrmann, C. (2021) 'LIC-SEC: An Enhanced Apparmor Docker Security Profile Generator', Journal of Information Security and Applications, 61, p. 102924. doi:10.1016/j.jisa.2021.102924.

Bui, T. (2015) 'Analysis of Docker Security', Aalto University School of Science [Preprint]. doi:https://arxiv.org/pdf/1501.02967v1.pdf.

Soltesz, S. et al. (2007) 'Container-based operating system virtualization', Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 [Preprint]. doi:10.1145/1272996.1273025.

Xavier, M.G. et al. (2013) 'Performance evaluation of container-based virtualization for high performance computing environments', 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing [Preprint]. doi:10.1109/pdp.2013.41.

ComputerNetworkingNotes (2023) Selinux explained with examples in easy language, ComputerNetworkingNotes. Available at: https://www.computernetworkingnotes.com/linux-tutorials/selinux-explained-with-examples-in-easy-language.html (Accessed: 10 March 2024).

Ubuntu (no date) Security - apparmor | ubuntu, How to create an AppArmor Profile. Available at: https://ubuntu.com/server/docs/security-apparmor (Accessed: 10 March 2024).

Red Hat (no date) Chapter 1. getting started with Selinux Red Hat Enterprise Linux 8, Red Hat Customer Portal. Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/using_selinux/getting-started-with-selinux_using-selinux (Accessed: 10 March 2024).

Zhu, H. and Gehrmann, C. (2021a) 'LIC-SEC: An Enhanced Apparmor Docker Security Profile Generator', Journal of Information Security and Applications, 61, p. 102924. doi:10.1016/j.jisa.2021.102924.

Chamoli, S. and Sarishma (2021) 'Docker Security: Architecture, threat model, and best practices', Advances in Intelligent Systems and Computing, pp. 253–263. doi:10.1007/978-981-16-1696-9_24.

Engine Yard Team (2022) Docker vs Virtual Machines explained, EngineYard. Available at: https://www.engineyard.com/blog/docker-vs-virtual-machines-explained/ (Accessed: 10 March 2024).

Aleksic, M. (2023) Unikernel vs. containers: What's the difference?: Phoenixnap KB, Knowledge Base by phoenixNAP. Available at: https://phoenixnap.com/kb/unikernel-vs-container (Accessed: 10 March 2024).

Bala Priya (2022) Docker vs Virtual Machine (VM) – key differences you should know, freeCodeCamp.org. Available at: https://www.freecodecamp.org/news/docker-vs-vm-key-differences-you-should-know/ (Accessed: 10 March 2024).

Wwlpublish (2024) List the differences between containers and VMS - training, Training | Microsoft Learn. Available at: https://learn.microsoft.com/en-us/training/modules/run-containers-windows-server/3-list-differences-between-containers-vms (Accessed: 10 March 2024).

Wwlpublish (2024a) List the differences between containers and VMS - training, Training | Microsoft Learn. Available at: https://learn.microsoft.com/en-us/training/modules/run-containers-windows-server/3-list-differences-between-containers-vms (Accessed: 10 March 2024).

Wwlpublish (2024a) Define windows server and hyper-V containers and isolation modes - training, Training | Microsoft Learn. Available at: https://learn.microsoft.com/en-us/training/modules/run-containers-windows-server/4-define-hyper-v-containers-isolation-modes (Accessed: 10 March 2024).

Sanner, E. (2022) Docker bootcamp - container isolation modes / blogs / perficient, Perficient Blogs. Available at: https://blogs.perficient.com/2022/10/10/docker-bootcamp-container-isolation-modes/ (Accessed: 10 March 2024).

Gamela, A. (2023) Podman vs Docker: What are the differences?, Blog | Imaginary Cloud. Available at: https://www.imaginarycloud.com/blog/podman-vs-docker/ (Accessed: 10 March 2024).

Chen, R. (2019) What is this VMMEM program that is using up all my CPU and memory?, The Old New Thing. Available at: https://devblogs.microsoft.com/oldnewthing/20180717-00/?p=99265 (Accessed: 15 March 2024).

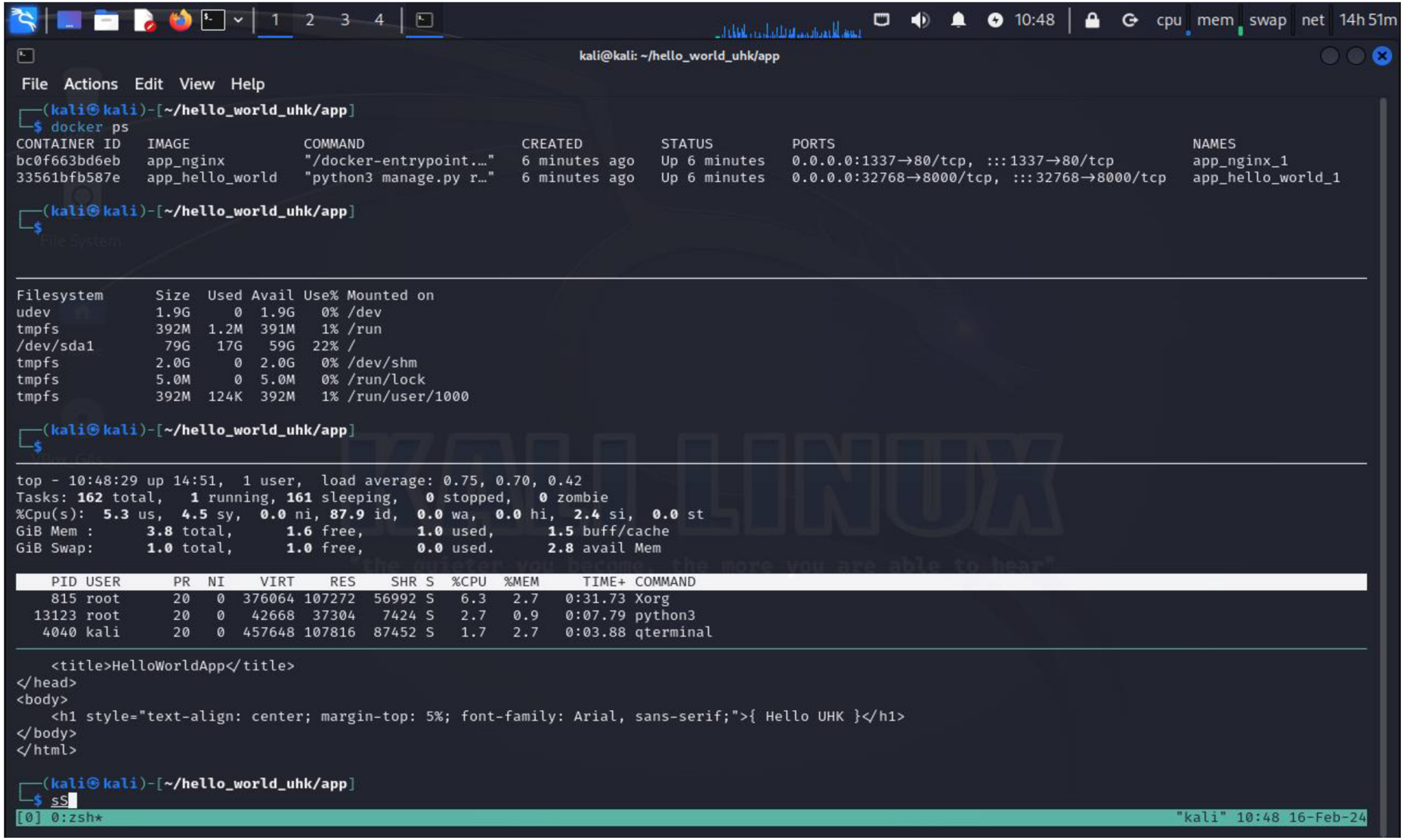
Docker Docs. (2024, February 26). Run the docker daemon as a non-root user (rootless mode). Docker Documentation. https://docs.docker.com/engine/security/rootless/

LHB Community (2023) How to do a rootless docker installation?, Linux Handbook. Available at: https://linuxhandbook.com/rootless-docker/ (Accessed: 19 March 2024).

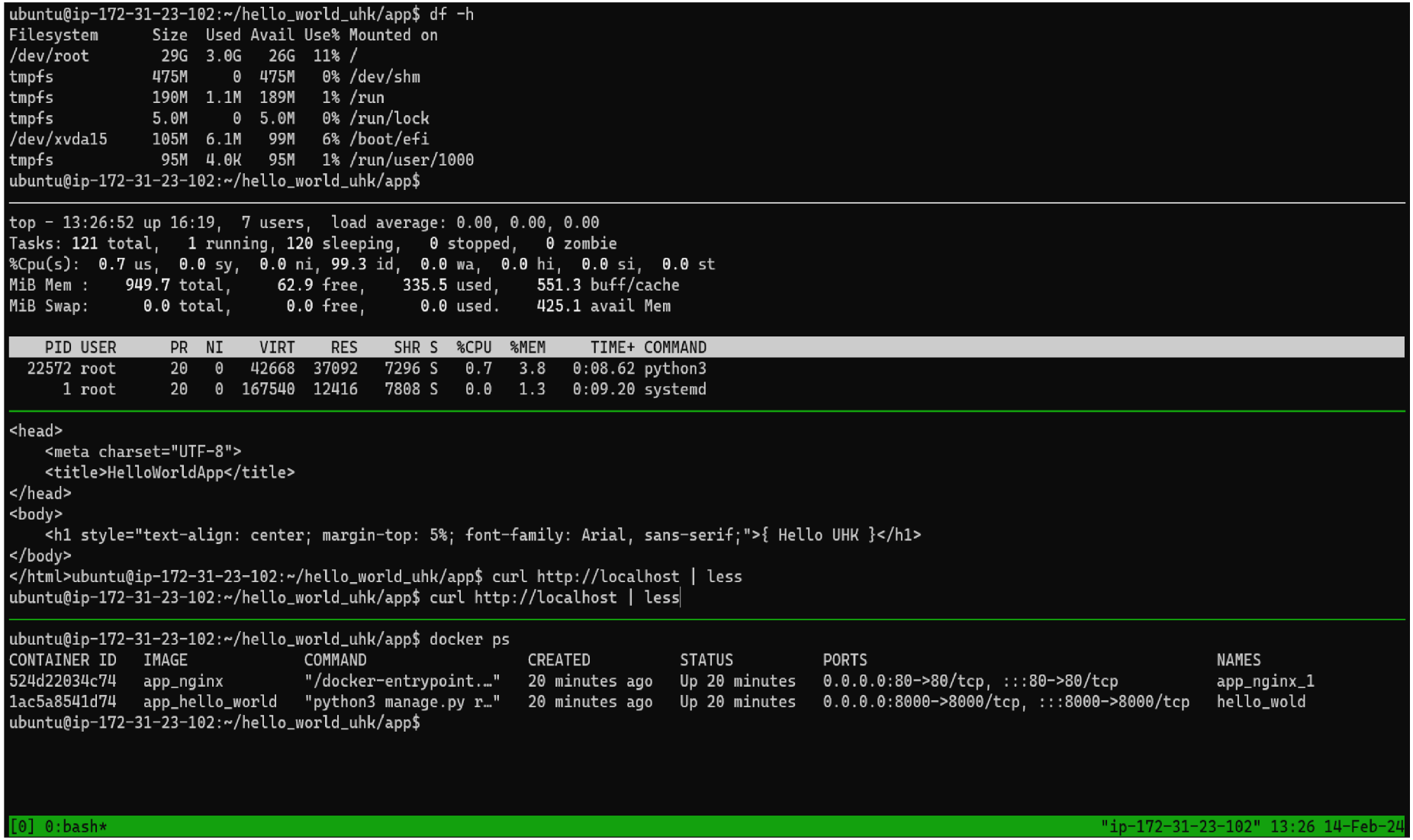
Molloy , J. (2022) How to run Rootless Docker Containers, Liquid Web. Available at: https://www.liquidweb.com/kb/how-to-docker-rootless-containers/ (Accessed: 19 March 2024).

11. Přílohy

Příloha 1:


1 2 3 4
10:48
cpu mem swap net 14h 51m
kali@kali: ~/hello_world_uhk/app
File Actions Edit View Help
(kali㉿kali)-[~/hello_world_uhk/app]
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
bc0f663bd6eb app_nginx "/docker-entrypoint.…" 6 minutes ago Up 6 minutes 0.0.0.0:1337→80/tcp, :::1337→80/tcp app_nginx_1
33561bfb587e app_hello_world "python3 manage.py r…" 6 minutes ago Up 6 minutes 0.0.0.0:32768→8000/tcp, :::32768→8000/tcp app_hello_world_1
(kali㉿kali)-[~/hello_world_uhk/app]
$
Filesystem Size Used Avail Use% Mounted on
udev 1.9G 0 1.9G 0% /dev
tmpfs 392M 1.2M 391M 1% /run
/dev/sda1 79G 17G 59G 22% /
tmpfs 2.0G 0 2.0G 0% /dev/shm
tmpfs 5.0M 0 5.0M 0% /run/lock
tmpfs 392M 124K 392M 1% /run/user/1000
(kali㉿kali)-[~/hello_world_uhk/app]
$
top - 10:48:29 up 14:51, 1 user, load average: 0.75, 0.70, 0.42
Tasks: 162 total, 1 running, 161 sleeping, 0 stopped, 0 zombie
%Cpu(s): 5.3 us, 4.5 sy, 0.0 ni, 87.9 id, 0.0 wa, 0.0 hi, 2.4 si, 0.0 st
GiB Mem : 3.8 total, 1.6 free, 1.0 used, 1.5 buff/cache
GiB Swap: 1.0 total, 1.0 free, 0.0 used. 2.8 avail Mem
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
815 root 20 0 376064 107272 56992 S 6.3 2.7 0:31.73 Xorg
13123 root 20 0 42668 37304 7424 S 2.7 0.9 0:07.79 python3
4040 kali 20 0 457648 107816 87452 S 1.7 2.7 0:03.88 qterminal
<title>HelloWorldApp</title>
</head>
<body>
<h1 style="text-align: center; margin-top: 5%; font-family: Arial, sans-serif;">{ Hello UHK }</h1>
</body>
</html>
(kali㉿kali)-[~/hello_world_uhk/app]
$ sS
[0] 0:zsh*
"kali" 10:48 16-Feb-24

Příloha 2:



```
ubuntu@ip-172-31-23-102:~/hello_world_uhk/app$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/root        29G  3.0G   26G  11% /
tmpfs           475M     0  475M   0% /dev/shm
tmpfs           190M  1.1M  189M   1% /run
tmpfs           5.0M     0  5.0M   0% /run/lock
/dev/xvda15     105M  6.1M   99M   6% /boot/efi
tmpfs            95M  4.0K   95M   1% /run/user/1000
ubuntu@ip-172-31-23-102:~/hello_world_uhk/app$
```

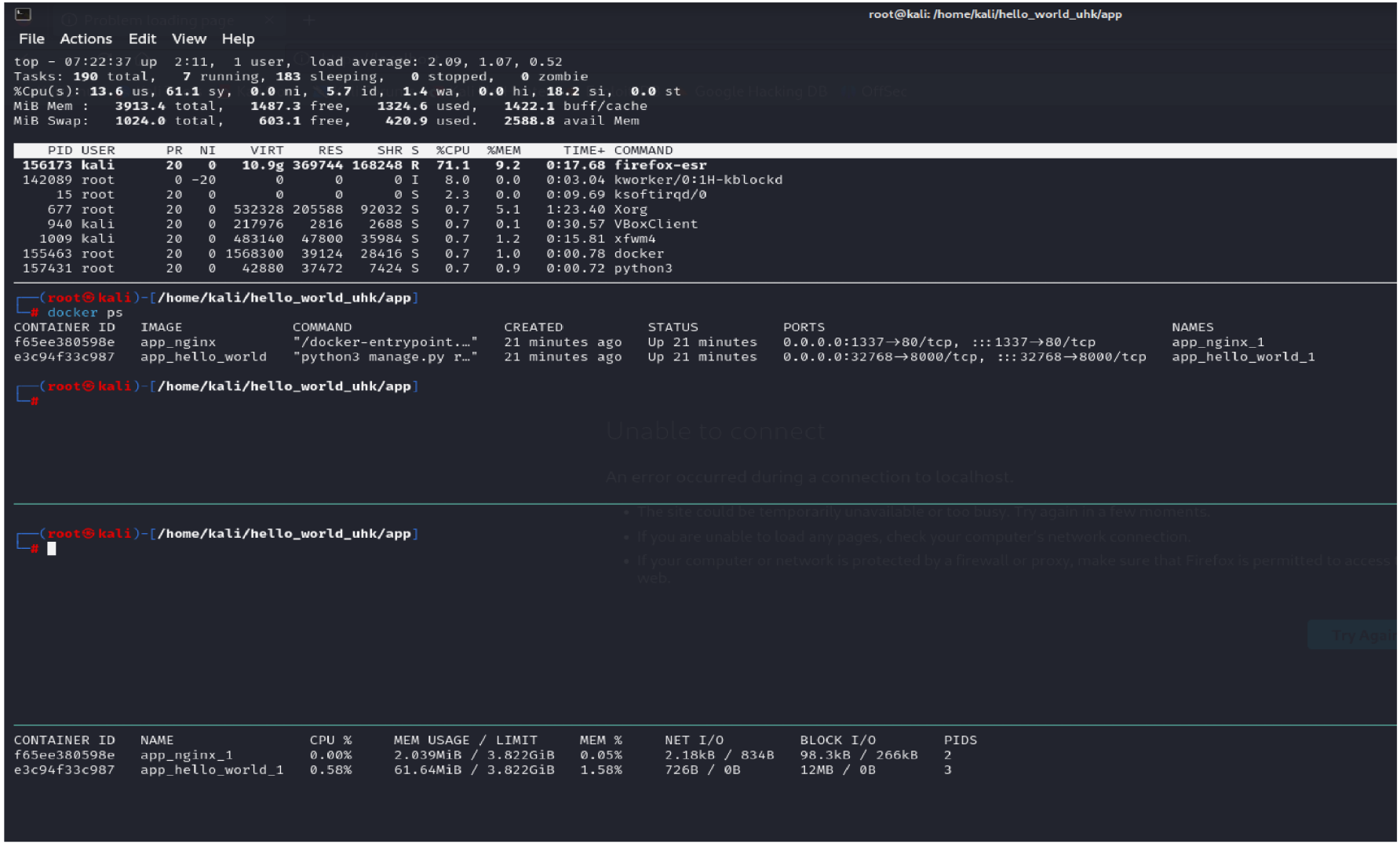
```
top - 13:26:52 up 16:19,  7 users,  load average: 0.00, 0.00, 0.00
Tasks: 121 total,   1 running, 120 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.7 us,  0.0 sy,  0.0 ni, 99.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :    949.7 total,     62.9 free,    335.5 used,    551.3 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.    425.1 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  22572 root      20   0   42668  37092   7296 S   0.7   3.8   0:08.62 python3
      1 root      20   0  167540  12416   7808 S   0.0   1.3   0:09.20 systemd
```

```
<head>
    <meta charset="UTF-8">
    <title>HelloWorldApp</title>
</head>
<body>
    <h1 style="text-align: center; margin-top: 5%; font-family: Arial, sans-serif;">{ Hello UHK }</h1>
</body>
</html>ubuntu@ip-172-31-23-102:~/hello_world_uhk/app$ curl http://localhost | less
ubuntu@ip-172-31-23-102:~/hello_world_uhk/app$ curl http://localhost | less
```

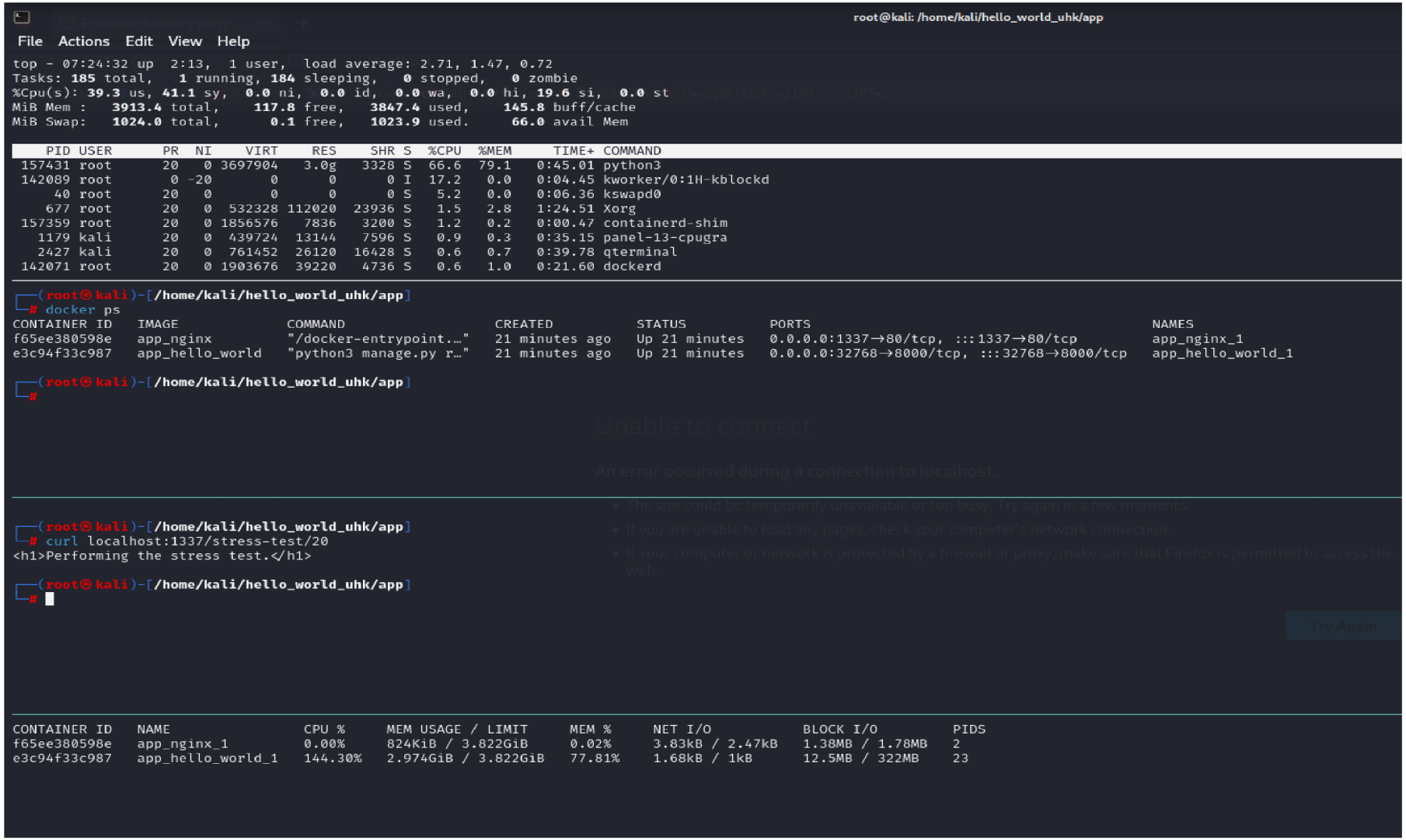
```
ubuntu@ip-172-31-23-102:~/hello_world_uhk/app$ docker ps
CONTAINER ID   IMAGE             COMMAND                  CREATED          STATUS          PORTS                                        NAMES
524d22034c74   app_nginx         "/docker-entrypoint.…"   20 minutes ago   Up 20 minutes   0.0.0.0:80->80/tcp, :::80->80/tcp            app_nginx_1
1ac5a8541d74   app_hello_world   "python3 manage.py r…"   20 minutes ago   Up 20 minutes   0.0.0.0:8000->8000/tcp, :::8000->8000/tcp    hello_wold
ubuntu@ip-172-31-23-102:~/hello_world_uhk/app$
```

```
[0] 0:bash*                                                                    "ip-172-31-23-102" 13:26 14-Feb-24
```

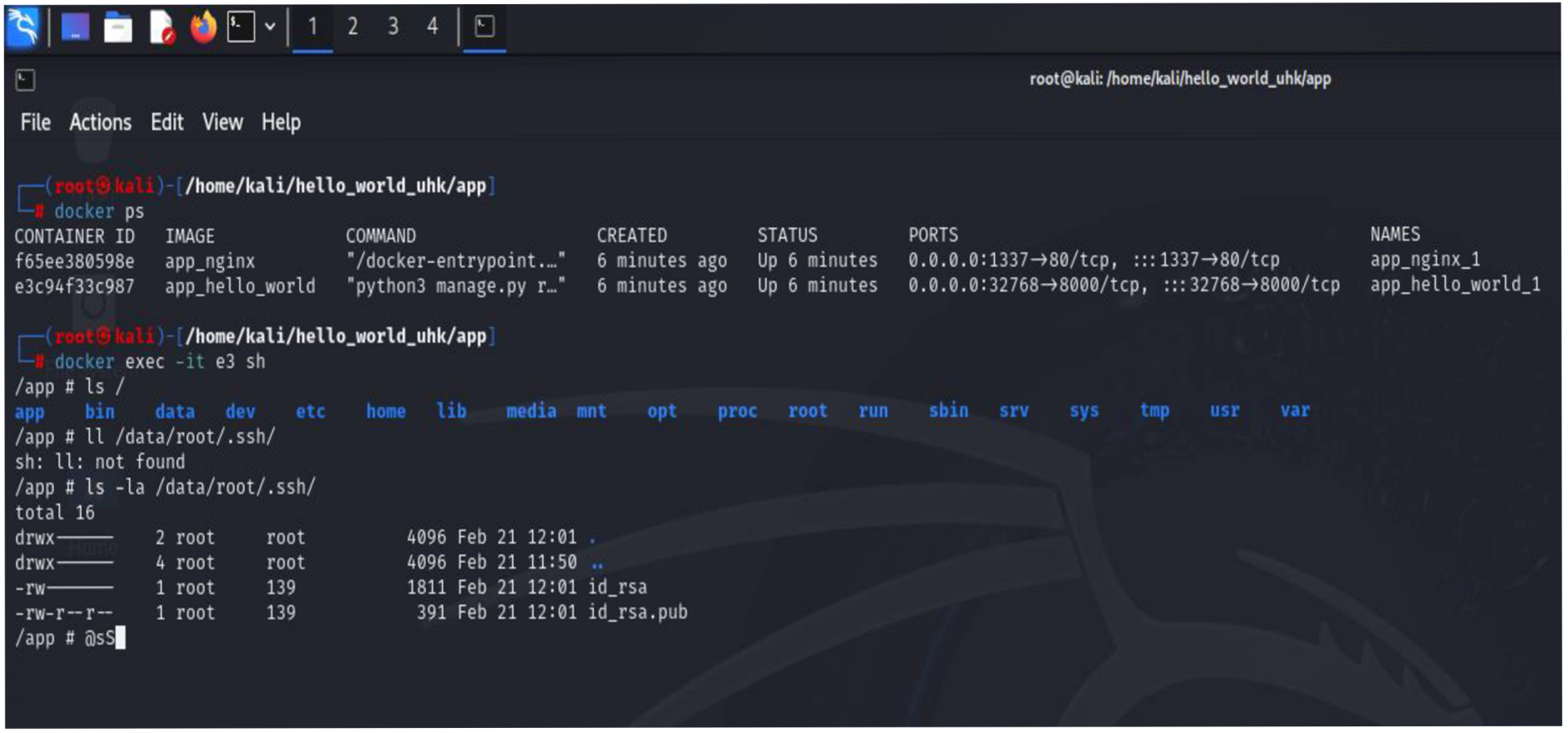
Příloha 3:


root@kali: /home/kali/hello_world_uhk/app
File Actions Edit View Help
top - 07:22:37 up 2:11, 1 user, load average: 2.09, 1.07, 0.52
Tasks: 190 total, 7 running, 183 sleeping, 0 stopped, 0 zombie
%Cpu(s): 13.6 us, 61.1 sy, 0.0 ni, 5.7 id, 1.4 wa, 0.0 hi, 18.2 si, 0.0 st
MiB Mem : 3913.4 total, 1487.3 free, 1324.6 used, 1422.1 buff/cache
MiB Swap: 1024.0 total, 603.1 free, 420.9 used. 2588.8 avail Mem
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
156173 kali 20 0 10.9g 369744 168248 R 71.1 9.2 0:17.68 firefox-esr
142089 root 0 -20 0 0 0 I 8.0 0.0 0:03.04 kworker/0:1H-kblockd
15 root 20 0 0 0 0 S 2.3 0.0 0:09.69 ksoftirqd/0
677 root 20 0 532328 205588 92032 S 0.7 5.1 1:23.40 Xorg
940 kali 20 0 217976 2816 2688 S 0.7 0.1 0:30.57 VBoxClient
1009 kali 20 0 483140 47800 35984 S 0.7 1.2 0:15.81 xfwm4
155463 root 20 0 1568300 39124 28416 S 0.7 1.0 0:00.78 docker
157431 root 20 0 42880 37472 7424 S 0.7 0.9 0:00.72 python3
(root㉿kali)-[/home/kali/hello_world_uhk/app]
docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
f65ee380598e app_nginx "/docker-entrypoint.…" 21 minutes ago Up 21 minutes 0.0.0.0:1337→80/tcp, :::1337→80/tcp app_nginx_1
e3c94f33c987 app_hello_world "python3 manage.py r…" 21 minutes ago Up 21 minutes 0.0.0.0:32768→8000/tcp, :::32768→8000/tcp app_hello_world_1
(root㉿kali)-[/home/kali/hello_world_uhk/app]
#
(root㉿kali)-[/home/kali/hello_world_uhk/app]
#
CONTAINER ID NAME CPU % MEM USAGE / LIMIT MEM % NET I/O BLOCK I/O PIDS
f65ee380598e app_nginx_1 0.00% 2.039MiB / 3.822GiB 0.05% 2.18kB / 834B 98.3kB / 266kB 2
e3c94f33c987 app_hello_world_1 0.58% 61.64MiB / 3.822GiB 1.58% 726B / 0B 12MB / 0B 3

Příloha 4:


root@kali: /home/kali/hello_world_uhk/app
File Actions Edit View Help
top - 07:24:32 up 2:13, 1 user, load average: 2.71, 1.47, 0.72
Tasks: 185 total, 1 running, 184 sleeping, 0 stopped, 0 zombie
%Cpu(s): 39.3 us, 41.1 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 19.6 si, 0.0 st
MiB Mem : 3913.4 total, 117.8 free, 3847.4 used, 145.8 buff/cache
MiB Swap: 1024.0 total, 0.1 free, 1023.9 used. 66.0 avail Mem
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
157431 root 20 0 3697904 3.0g 3328 S 66.6 79.1 0:45.01 python3
142089 root 0 -20 0 0 0 I 17.2 0.0 0:04.45 kworker/0:1H-kblockd
40 root 20 0 0 0 0 S 5.2 0.0 0:06.36 kswapd0
677 root 20 0 532328 112020 23936 S 1.5 2.8 1:24.51 Xorg
157359 root 20 0 1856576 7836 3200 S 1.2 0.2 0:00.47 containerd-shim
1179 kali 20 0 439724 13144 7596 S 0.9 0.3 0:35.15 panel-13-cpugra
2427 kali 20 0 761452 26120 16428 S 0.6 0.7 0:39.78 qterminal
142071 root 20 0 1903676 39220 4736 S 0.6 1.0 0:21.60 dockerd
(root㉿kali)-[/home/kali/hello_world_uhk/app]
docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
f65ee380598e app_nginx "/docker-entrypoint.…" 21 minutes ago Up 21 minutes 0.0.0.0:1337→80/tcp, :::1337→80/tcp app_nginx_1
e3c94f33c987 app_hello_world "python3 manage.py r…" 21 minutes ago Up 21 minutes 0.0.0.0:32768→8000/tcp, :::32768→8000/tcp app_hello_world_1
(root㉿kali)-[/home/kali/hello_world_uhk/app]
#
(root㉿kali)-[/home/kali/hello_world_uhk/app]
curl localhost:1337/stress-test/20
<h1>Performing the stress test.</h1>
(root㉿kali)-[/home/kali/hello_world_uhk/app]
#
CONTAINER ID NAME CPU % MEM USAGE / LIMIT MEM % NET I/O BLOCK I/O PIDS
f65ee380598e app_nginx_1 0.00% 824KiB / 3.822GiB 0.02% 3.83kB / 2.47kB 1.38MB / 1.78MB 2
e3c94f33c987 app_hello_world_1 144.30% 2.974GiB / 3.822GiB 77.81% 1.68kB / 1kB 12.5MB / 322MB 23

Příloha 5:


1 2 3 4
root@kali: /home/kali/hello_world_uhk/app
File Actions Edit View Help
┌──(root㉿kali)-[/home/kali/hello_world_uhk/app]
└─# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
f65ee380598e app_nginx "/docker-entrypoint.…" 6 minutes ago Up 6 minutes 0.0.0.0:1337→80/tcp, :::1337→80/tcp app_nginx_1
e3c94f33c987 app_hello_world "python3 manage.py r…" 6 minutes ago Up 6 minutes 0.0.0.0:32768→8000/tcp, :::32768→8000/tcp app_hello_world_1
┌──(root㉿kali)-[/home/kali/hello_world_uhk/app]
└─# docker exec -it e3 sh
/app # ls /
app bin data dev etc home lib media mnt opt proc root run sbin srv sys tmp usr var
/app # ll /data/root/.ssh/
sh: ll: not found
/app # ls -la /data/root/.ssh/
total 16
drwx------ 2 root root 4096 Feb 21 12:01 .
drwx------ 4 root root 4096 Feb 21 11:50 ..
-rw------- 1 root 139 1811 Feb 21 12:01 id_rsa
-rw-r--r-- 1 root 139 391 Feb 21 12:01 id_rsa.pub
/app # @sS

Příloha 6:

```
┌──(root㉿kali)-[/home/kali/hello_world_uhk/app]
└─# docker ps
CONTAINER ID   IMAGE             COMMAND                  CREATED          STATUS          PORTS                                                  NAMES
f65ee380598e   app_nginx         "/docker-entrypoint.…"   16 minutes ago   Up 16 minutes   0.0.0.0:1337→80/tcp, :::1337→80/tcp                    app_nginx_1
e3c94f33c987   app_hello_world   "python3 manage.py r…"   16 minutes ago   Up 16 minutes   0.0.0.0:32768→8000/tcp, :::32768→8000/tcp              app_hello_world_1

┌──(root㉿kali)-[/home/kali/hello_world_uhk/app]
└─# trivy image app_hello_world
2024-02-21T07:14:48.045-0500    INFO    Need to update DB
2024-02-21T07:14:48.046-0500    INFO    DB Repository: ghcr.io/aquasecurity/trivy-db
2024-02-21T07:14:48.046-0500    INFO    Downloading DB ...
43.05 MiB / 43.05 MiB [-------------------------------------------------------------------------------------------------------------------------]
2024-02-21T07:14:56.646-0500    INFO    Vulnerability scanning is enabled
2024-02-21T07:14:56.646-0500    INFO    Secret scanning is enabled
2024-02-21T07:14:56.646-0500    INFO    If your scanning is slow, please try '--scanners vuln' to disable secret scanning
2024-02-21T07:14:56.647-0500    INFO    Please see also https://aquasecurity.github.io/trivy/dev/docs/scanner/secret/#recommendation for faster secret detection
2024-02-21T07:15:05.049-0500    INFO    Detected OS: alpine
2024-02-21T07:15:05.055-0500    INFO    Detecting Alpine vulnerabilities ...
2024-02-21T07:15:05.100-0500    INFO    Number of language-specific files: 1
2024-02-21T07:15:05.101-0500    INFO    Detecting python-pkg vulnerabilities ...

app_hello_world (alpine 3.19.1)

Total: 2 (UNKNOWN: 0, LOW: 0, MEDIUM: 1, HIGH: 1, CRITICAL: 0)

┌─────────┬─────────────────┬──────────┬────────┬───────────────────┬───────────────┬──────────────────────────────────────────────────────────────┐
│ Library │ Vulnerability   │ Severity │ Status │ Installed Version │ Fixed Version │                            Title                             │
├─────────┼─────────────────┼──────────┼────────┼───────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ libexpat│ CVE-2023-52425  │ HIGH     │ fixed  │ 2.5.0-r2          │ 2.6.0-r0      │ expat: parsing large tokens can trigger a denial of service  │
│         │                 │          │        │                   │               │ https://avd.aquasec.com/nvd/cve-2023-52425                   │
│         ├─────────────────┼──────────┤        │                   │               ├──────────────────────────────────────────────────────────────┤
│         │ CVE-2023-52426  │ MEDIUM   │        │                   │               │ expat: recursive XML entity expansion vulnerability          │
│         │                 │          │        │                   │               │ https://avd.aquasec.com/nvd/cve-2023-52426                   │
└─────────┴─────────────────┴──────────┴────────┴───────────────────┴───────────────┴──────────────────────────────────────────────────────────────┘
2024-02-21T07:15:05.207-0500    INFO    Table result includes only package filenames. Use '--format json' option to get the full path to the package file.

Python (python-pkg)

Total: 1 (UNKNOWN: 0, LOW: 0, MEDIUM: 1, HIGH: 0, CRITICAL: 0)

┌──────────────────┬─────────────────┬──────────┬────────┬───────────────────┬───────────────┬──────────────────────────────────────────────────────────────┐
│     Library      │ Vulnerability   │ Severity │ Status │ Installed Version │ Fixed Version │                            Title                             │
├──────────────────┼─────────────────┼──────────┼────────┼───────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ pip (METADATA)   │ CVE-2023-5752   │ MEDIUM   │ fixed  │ 23.0.1            │ 23.3          │ pip: Mercurial configuration injectable in repo revision     │
│                  │                 │          │        │                   │               │ when installing via pip                                      │
│                  │                 │          │        │                   │               │ https://avd.aquasec.com/nvd/cve-2023-5752                    │
└──────────────────┴─────────────────┴──────────┴────────┴───────────────────┴───────────────┴──────────────────────────────────────────────────────────────┘
```

Příloha 7:

```
david@david-VirtualBox:~$ systemctl --user status docker
● docker.service - Docker Application Container Engine (Rootless)
     Loaded: loaded (/home/david/.config/systemd/user/docker.service; enabled; vendor preset: enabled)
     Active: active (running) since Tue 2024-03-19 17:29:02 CET; 20min ago
       Docs: https://docs.docker.com/go/rootless/
   Main PID: 11183 (rootlesskit)
      Tasks: 40
     Memory: 86.0M
        CPU: 4.137s
     CGroup: /user.slice/user-1000.slice/user@1000.service/app.slice/docker.service
             ├─11183 rootlesskit --state-dir=/run/user/1000/dockerd-rootless --net=vpnkit --mtu=1500 --slirp4netns-sandbox=auto --s>
             ├─11190 /proc/self/exe --state-dir=/run/user/1000/dockerd-rootless --net=vpnkit --mtu=1500 --slirp4netns-sandbox=auto >
             ├─11203 vpnkit --ethernet /run/user/1000/dockerd-rootless/vpnkit-ethernet.sock --mtu 1500 --host-ip 0.0.0.0
             ├─11216 dockerd
             └─11231 containerd --config /run/user/1000/docker/containerd/containerd.toml

bře 19 17:29:02 david-VirtualBox systemd[826]: Started Docker Application Container Engine (Rootless).
bře 19 17:33:46 david-VirtualBox dockerd-rootless.sh[11231]: time="2024-03-19T17:33:46.887530553+01:00" level=info msg="loading plu>
bře 19 17:33:46 david-VirtualBox dockerd-rootless.sh[11231]: time="2024-03-19T17:33:46.887661465+01:00" level=info msg="loading plu>
bře 19 17:33:46 david-VirtualBox dockerd-rootless.sh[11231]: time="2024-03-19T17:33:46.887673738+01:00" level=info msg="loading plu>
bře 19 17:33:46 david-VirtualBox dockerd-rootless.sh[11231]: time="2024-03-19T17:33:46.890834196+01:00" level=info msg="loading plu>
bře 19 17:33:47 david-VirtualBox dockerd-rootless.sh[11231]: time="2024-03-19T17:33:47.409975960+01:00" level=warning msg="error fr>
bře 19 17:33:47 david-VirtualBox dockerd-rootless.sh[11216]: time="2024-03-19T17:33:47.498373653+01:00" level=info msg="ignoring ev>
bře 19 17:33:47 david-VirtualBox dockerd-rootless.sh[11231]: time="2024-03-19T17:33:47.500622324+01:00" level=info msg="shim discon>
bře 19 17:33:47 david-VirtualBox dockerd-rootless.sh[11231]: time="2024-03-19T17:33:47.503164493+01:00" level=warning msg="cleaning>
bře 19 17:33:47 david-VirtualBox dockerd-rootless.sh[11231]: time="2024-03-19T17:33:47.503531995+01:00" level=info msg="cleaning up>
david@david-VirtualBox:~$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.
```

Příloha 8:

```
Successfully built 14faac45c84b
Successfully tagged app_nginx:latest
WARNING: Image for service nginx was built because it did not already exist. To rebuild this image you must use `docker-compose buil
d` or `docker-compose up --build`.
Creating app_hello_world_1 ... done
Creating app_nginx_1       ... done
david@david-VirtualBox:~/hello_world_uhk/app$ docker ps
CONTAINER ID   IMAGE             COMMAND                  CREATED          STATUS         PORTS
    NAMES
6c8a99d102ee   app_nginx         "/docker-entrypoint.…"   10 seconds ago   Up 8 seconds   0.0.0.0:1337->80/tcp, :::1337->80/tcp
    app_nginx_1
465482e81e0b   app_hello_world   "python3 manage.py r…"   12 seconds ago   Up 9 seconds   0.0.0.0:32768->8000/tcp, :::32768->8000/tc
p   app_hello_world_1
```

Příloha 9:

```
david@david-VirtualBox:~/hello_world_uhk/app$ docker ps
CONTAINER ID   IMAGE             COMMAND                  CREATED         STATUS         PORTS
   NAMES
6c8a99d102ee   app_nginx         "/docker-entrypoint.…"   9 minutes ago   Up 9 minutes   0.0.0.0:1337->80/tcp, :::1337->80/tcp
   app_nginx_1
465482e81e0b   app_hello_world   "python3 manage.py r…"   9 minutes ago   Up 9 minutes   0.0.0.0:32768->8000/tcp, :::32768->8000/tcp
   app_hello_world_1
david@david-VirtualBox:~/hello_world_uhk/app$ ps aux | grep 465
david      13848  0.0  0.3 1238936 13776 ?       Sl   19:40   0:00 /home/david/bin/containerd-shim-runc-v2 -namespace moby -id 46548
2e81e0bba2ea4a6ad4616f7d3d22be44dcd31f30057c56b7244928eaac4 -address /run/user/1000/docker/containerd/containerd.sock
david      14187  0.0  0.0  20400  2688 pts/0    S+   19:50   0:00 grep --color=auto 465
david@david-VirtualBox:~/hello_world_uhk/app$
```

Příloha 10:

```
Successfully built 14faac45c84b
Successfully tagged app_nginx:latest
Recreating app_hello_world_1 ... done
app_nginx_1 is up-to-date
david@david-VirtualBox:~/hello_world_uhk/app$ docker ps
CONTAINER ID   IMAGE             COMMAND                  CREATED          STATUS          PORTS
     NAMES
2e142565efa2   app_hello_world   "python3 manage.py r…"   6 seconds ago    Up 5 seconds    0.0.0.0:32769->8000/tcp, :::32769->8000/t
cp   app_hello_world_1
6c8a99d102ee   app_nginx         "/docker-entrypoint.…"   14 minutes ago   Up 14 minutes   0.0.0.0:1337->80/tcp, :::1337->80/tcp
     app_nginx_1
david@david-VirtualBox:~/hello_world_uhk/app$ docker exec -it 2e sh
/app # cd /
/ # ls
app    data   etc    lib    mnt    proc   run    srv    tmp    var
bin    dev    home   media  opt    root   sbin   sys    usr
/ # ls data/
bin         dev         lib         libx32      mnt         root        snap        sys         var
boot        etc         lib32       lost+found  opt         run         srv         tmp
cdrom       home        lib64       media       proc        sbin        swapfile    usr
/ # ls /data/root/
ls: can't open '/data/root/': Permission denied
/ #
```

12. Zadání práce z IS (eVŠKP)

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor:	**David Louda**
Studium:	I2100711
Studijní program:	B1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název bakalářské práce:	**Využití Docker pro bezpečné testování software**
Název bakalářské práce AJ:	

Cíl, metody, literatura, předpoklady:

Cílem práce je představit principy Dockeru a jeho využití pro bezpečné testování software. V teoretické části autor představí principy Dockeru, jeho implementace a využití pro bezpečné testování. V pratkické části autor sestaví minimálně laboratorní úlohy od istalace Dockeru, přes jeho ukázkové použití na předem definovaných scénářích.

Zadávající pracoviště:	Katedra informačních technologií, Fakulta informatiky a managementu
Vedoucí práce:	doc. Mgr. Josef Horálek, Ph.D.
Datum zadání závěrečné práce:	15.10.2021