

Sem vložte zadání Vaší práce.





**FAKULTA  
INFORMAČNÍCH  
TECHNologiÍ  
ČVUT V PRAZE**

Bakalářská práce

## **Kubernetes klastr pro lámání hesel**

*Tomáš Klas*

Katedra informační bezpečnosti (KIB)

Vedoucí práce: Ing. Jiří Buček, Ph.D.

14. března 2020



---

## Poděkování

Doplňte, máte-li komu a za co děkovat. V opačném případě úplně odstraňte tento příkaz.



---

# Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 14. března 2020

.....

České vysoké učení technické v Praze

Fakulta informačních technologií

© 2020 Tomáš Klas. Všechna práva vyhrazena.

*Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.*

### **Odkaz na tuto práci**

Klas, Tomáš. *Kubernetes klastr pro lámání hesel*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.



---

## Abstrakt

Hlavní náplní práce je nakonfigurování klastru pro lámání hesel. Tento klastr je řízen pomocí technologie Kubernetes. Program využívá ke své správné funkcionalitě kontejnery. Tyto kontejnery jsou tzv. Docker kontejnery. Použité technologie jsou v práci detailně popsány a rozebrány. Dále se práce zabývá rešerží ukládání hesel v současných systémech a tím jak hesla vypadají. Na závěr na klastru bude proveden test různých metod pro lámání hesel. Tyto metody budou popsány a bude analyzováno, jak je klastr efektní a výkonný pro daný typ lámání.

**Klíčová slova** Kubernetes, Ansible, klastr, Docker, distribuované lámání, hesla, hashcat, nasazení.

---

## Abstract

The main goal of the thesis is to setup a cluster managed by kubernetes for password recovery. Next step is to describe used technologies as Docker, Ansible and Hashcat. Thesis contains description of how the passwords are stored and most known attacks to crack them. Successful deployment and password cracking leads to analyzing speed of the cluster and the particular cracking method.

**Keywords** Kubernetes, Ansible, cluster, Docker, distributed cracking, passwords, hashcat, deployment.

---

# Obsah

Úvod	1
<b>1 Cíl práce</b>	<b>3</b>
1.1 Kubernetes	3
1.1.1 Stavební kameny Kubernetes	4
1.1.1.1 Pod	5
1.1.1.2 Control Plane	5
1.1.2 Komponenty nodu	6
1.1.2.1 Kubelet	6
1.1.2.2 Kube-Proxy	6
1.2 Ansible	7
1.2.1 Komponenty	8
1.2.1.1 Control node	8
1.2.1.2 Managed node	8
1.2.1.3 Inventory	8
1.2.1.4 Modules	8
1.2.1.5 Tasks	9
1.2.1.6 Playbooks	9
1.3 Docker	9
1.3.1 Kontejner vs. virtuální počítač	9
1.3.2 Stavební kameny Dockeru	11
1.3.2.1 Jmenné prostory	11
1.3.2.2 Kontrolní skupina	12
1.3.2.3 Docker daemon	13
1.3.2.4 Docker klient	13
1.3.2.5 Docker registr	13
1.3.2.6 Obrazy	13
1.3.3 Systémová kontejnery	13

<b>2 Hesla</b>	<b>15</b>
2.1 Windows . . . . .	15
2.1.1 Popis protokolu . . . . .	16
2.2 Linux . . . . .	16
2.3 Hešovací funkce . . . . .	17
2.3.0.1 MD5 . . . . .	18
2.3.0.2 SHA-1 . . . . .	18
2.4 Útoky na hesla . . . . .	18
2.5 Entropie hesla . . . . .	21
<b>3 Závěr</b>	<b>23</b>
<b>Literatura</b>	<b>25</b>
<b>A Seznam použitých zkratk</b>	<b>27</b>
<b>B Obsah přiloženého CD</b>	<b>29</b>

---

## Seznam obrázků

1.1	Komponenty Kubernetes . . . . .	4
1.2	Kubernetes Pod . . . . .	5
1.3	Kubernetes proxy v user space módu . . . . .	7
1.4	Kubernetes proxy fungující s iptables . . . . .	7
1.5	Docker VM . . . . .	10
1.6	Docker kontejnery . . . . .	11
1.7	Docker architektura . . . . .	12
1.8	Docker kontejnery . . . . .	14
1.9	LXC systémové kontejnery . . . . .	14
2.1	Hesla windows . . . . .	16
2.2	Hesla hešovací funkce . . . . .	17
2.3	Hesla a hešovací funkce MD5 . . . . .	19



---

## Seznam tabulek

1.1	Linuxové jmenné prostory . . . . .	12
2.1	Hašovací funkce . . . . .	17
2.2	SHA-X vlastnosti . . . . .	18





---

# Úvod



# Cíl práce

Cílem práce je sestavit Kubernetes klastr, nasazen bude pomocí technologie Ansible a následně na něm bude spuštěn software na lámání hesel. Rychlost a výsledky lámání budou analyzovány s ohledem na délku hesla a použití daného druhu útoku. Budou popsány běžně používané útoky na hesla a způsob jejich ukládání na různých operačních systémech. Technologie, jež budou použity, budou popsány do hloubky nutné k porozumění daného řešení a následné možné replikace pro jiná řešení.

## 1.1 Kubernetes

Jméno Kubernetes pochází z Řecka a znamená to kormidelník. Projekt zložili Joe Beda, Brendan Burns, a Craig McLuckie, ke kterým se rychle připojili inženýři z Googlu, jako Brian Grant a Tim Hockin. Software byl vydán v roce 2014.

Kontejnery jsou perfektní způsob, jak vytvářet aplikace tak, aby byly lehce rozšiřitelné a dobře spravovatelné. Kubernetes nám následně pomáhá k jejich nasazení a škálování. Co všechno tedy Kubernetes dokáže:

- **Service discovery and load balancing:** Kubernetes propojují kontejnery zkrze DNS a nebo jejich IP adresy. Pokud na jeden kontejner jde mnoho požadavků, přesměrují tyto požadavky na jiný a tímto způsobem balancují provoz a zajišťují stabilitu aplikace.
- **Orchestrace úložiště:** Dovolují automaticky připojovat vzdálené nebo lokální úložiště do klastru a zajistit tak konzistenci dat, zálohu a vysokou dostupnost z více míst.
- **Automated rollouts and rollbacks:** Můžeme popsat požadovaný stav pro naše nasazené kontejnery pomocí Kubernetes a ty automaticky a s kontrolou zajistí tento stav. Můžeme například automatizovat vytváření

## 1. CÍL PRÁCE

---

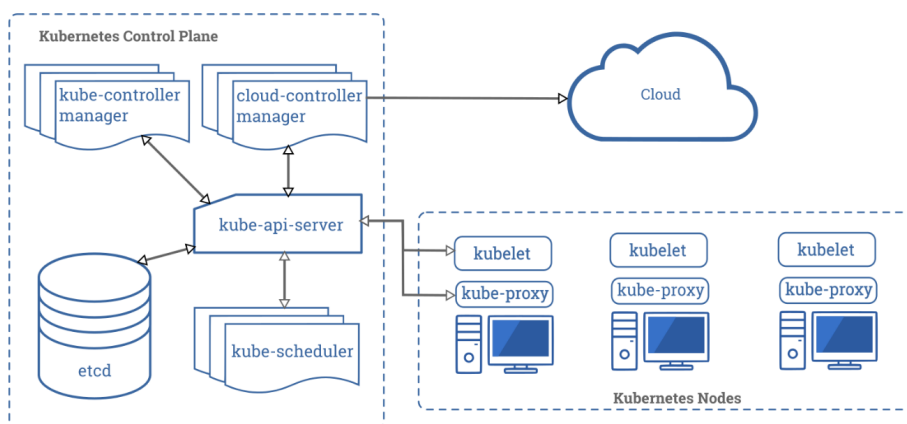
nových kontejnerů, smazání starých a převzetí všech zdrojů jako jsou data a další novým kontejnerem.

- **Automatic bin packing:** Předáme Kubernetes vztvořený klastr s uzly a specifikujeme zdroje, které každý kontejner potřebuje pro své správné fungování. Kubernetes se následně postará o nejlepší rozdělení kontejnerů na uzly tak, aby optimalizoval naše zdroje. To se může nejvíce hodit na cloudových řešení, kde se platí za to, kolik se využívá zdrojů.
- **Self-healing:** Kubernetes restartují kontejnery, které selžou, nahradí je, zahodí pokud přestanou odpovídat na specifikované health checky a přestane je nabízet klientům nebo jiným službám dokud nejsou připravené.
- **Secret and configuration management:** Kubernetes napomáhají ukládání a sdílení či měnění citlivých informací zkrze klastr. Při změně těchto informací tedy nemusíme znovu vytvářet kontejnery a nově všechno předělat. A to bez jejich zveřejňování v konfiguracích.

### 1.1.1 Stavební kameny Kubernetes

V momentě kdy nasadíme Kubernetes, dostaneme klastr, který se skládá z výpočetních uzlů. Takovému uzlu se říká node a na něm se spouští kontejnerizované aplikace. Každý klastr se skládá z alespoň jednoho takového nodu.

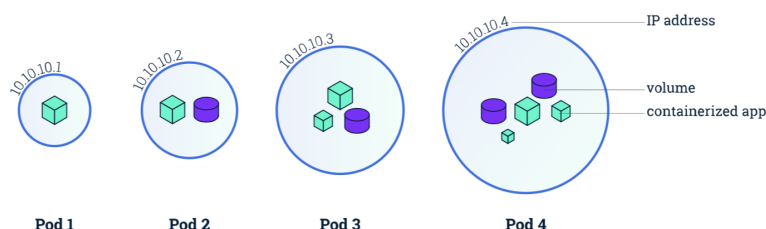
Tyto výpočetní nody poskztují prostor pro pody. Pod si můžeme představit jako balík, ve kterém jsou umístěné kontejnery. V praxi je nejčastěji Control Plane rozdělen na více počítačů, aby se zajistila vysoká dostupnost a maximalizovala se odolnost proti chybám.



Obrázek 1.1: Komponenty Kubernetes

### 1.1.1.1 Pod

Pod je základní stavební jednotka Kubernetes. Je to balíček kontejnerů nebo též samostatný kontejner, který je specifikován v deploymentu a Kubernetes se o něj mají starat. Pro představu je zde obrázek, z čeho se pod skládá. Takových podů může být na jednom nodu několik. To jak se rozhodneme zabalit naši aplikaci do podů je čistě na tom, jak se rozhodneme. Abychom však neporušovali konvence tzv. service architektury. Musíme do jednoho podu dávat služby pouze takové, které dávají smysl. Pro příklad nebudeme do jednoho podu balit databázi s webovým UI. Při škálování by se totiž nejen že replikovala aplikace ale i databáze.



Obrázek 1.2: Z čeho se skládá pod.

### 1.1.1.2 Control Plane

Součástí Control plane řídí a rozhodují co se stane s klastrem, např. řídí plánování, detekci a odpovědi klastru na události, jako jsou start nového podu pokud není splněn požadovaný stav.

Komponenty Control plane mohou být spuštěny na jakémkoliv počítači v klastru. Pro jednoduchost jsou však skripty napsané tak, aby tyto komponenty byly spuštěny na jednom počítači. Samotná aplikace je pak směrována mimo tento počítač.

- **kube-apiserver:** Tato komponenta poskytuje API uživateli. Je to front end pro Control plane. Hlavní implementací je kube-apiserver, ten je navržen, aby se dokázal horizontálně škálovat, to znamená, že se dokáže replikovat do více instancí. Můžeme tedy spustit více instancí a balancovat provoz.
- **etcd:** Skládá se z vysoce dostupné databáze s klíčem a hodnotou pro Kubernetes a všechna klastrová data. Její důležité vlastnosti jsou: jednoduchost, bezpečnost, rychlost, spolehlivost.

- **kube-scheduler**: Tato komponenta kontroluje a vytváří nové pody, pro které zatím nebyl přiřazen žádný node. Je odpovědná za přiřazení takového podu na nějaký node podle požadavků. Vždy musí brát v potaz zdroje, které jsou nutné, hardwarové a softwarové vazby, affinity a anti-affinity specifikace, kde jsou uložena data, vnitřní vytížení a časové termíny.
- **kube-controller-manager**: V této komponentě je spuštěn proces pro kontrolu klastru. Skládá se z více částí, které jsou zkompileovány do jednoho programu aby mohly běžet v jednom procesu, který se o vše stará. A stará se o následující:
  1. **Node Controller**: Odpovědný za upozornění pokud jeden z nodů přestane odpovídat.
  2. **Replication Controller**: Odpovědný za dodržování správného počtu spuštěných aplikací v klastru. Pokud tedy část aplikace spadne je odpovědný za start nové instance.
  3. **Endpoints Controller**: Přidá nová uzly, nebo-li připojí služby a pody do klastru po jejich spuštění.
  4. **Service Account a Token Controllers**: Vytváří výchozí účty a přístupové tokeny k API pro nové namespacey.
- **cloud-controller-manager**: Tento daemon dovoluje připojit cloudové služby. Byl vytvořen pro možnost oddělení cloudových vývojařů od zdrojového kódu Kubernetes. cloud-control-manager je možné připojit k jakémukoli poskytovateli, který implementuje cloudprovider.interface.

### 1.1.2 Komponenty nodu

Node, jak je již uvedeno výše, je v podstatě stroj, který je připojen do klastru. Na tomto nodu mohou běžet pody, což je balík kontejnerů. Níže je popsáno z čeho se tento node skládá a co na něm musí běžet, aby Kubernetes správně operovalo.

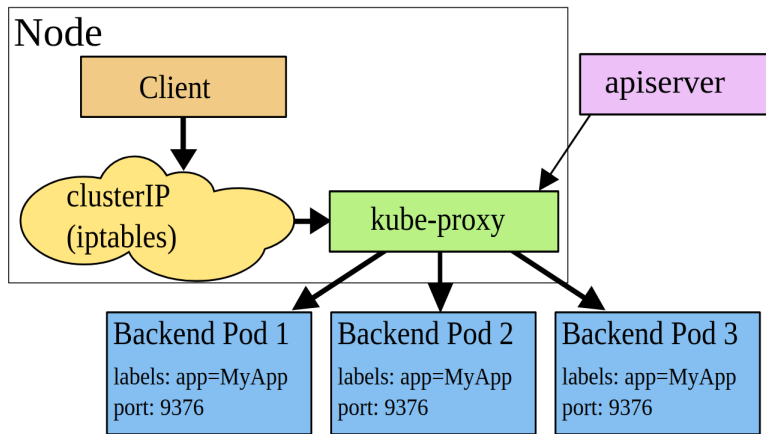
#### 1.1.2.1 Kubelet

Je agent zajišťující běh všech kontejnerů v daném podu. Kubelet převezme množinu požadavků na daný pod, tzv. PodSpecs a zajistí, že všechny kontejnery popsány v těchto specifikacích jsou spuštěné a odpovídají na health checky.

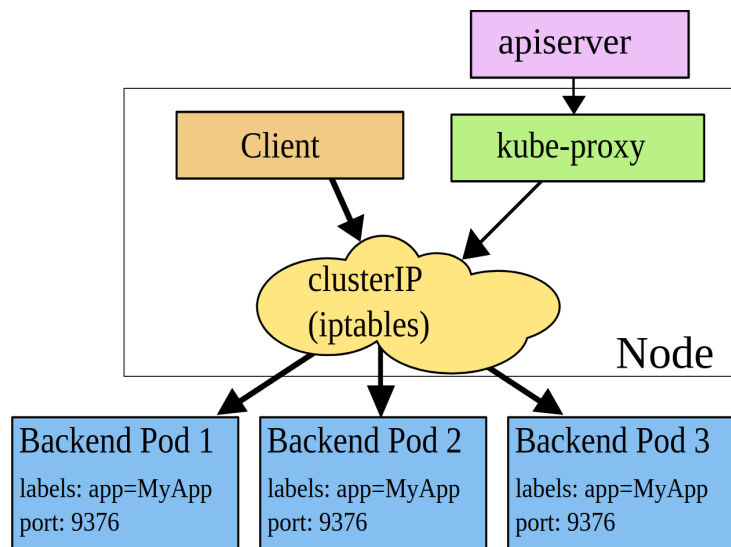
#### 1.1.2.2 Kube-Proxy

Je síťová proxy, která běží na každém nodu. Spravuje síťová pravidla. Zajišťuje tak komunikaci mezi jednotlivými pody a nody uvnitř klastru. Používá

szstémovou vrstvu na filtrování paketů, pokud je dostupná, jinak forwarduje provoz za pomoci svých prostředků. Níže znázorněno na obrázcích.



Obrázek 1.3: Kubernetes proxy - User space proxy mode



Obrázek 1.4: Kubernetes proxy - iptables proxy mode

## 1.2 Ansible

Ansible je automatizační nástroj pro konfiguraci systému, nasazení softwaru, aktualizac. Jeho nejsilnější stránka je nulové výpadky systému při aktualizaci balíčků, nebo automatické nastavovat dané zařízení.

## 1. CÍL PRÁCE

---

Jeho hlavními cíly jsou jednoduchost a nenáročnost. Kód by měl být čitelný i pro lidi, kteří nejsou obeznámeni s programem. Je schopen pokrýt různě velké prostředí od malých podniků až po velice obsáhlou infrastrukturu.

Ansible se připojí na vzdálený počítač pomocí OpenSSH pomocí uživatele, který je současně přihlášen. Na spravovaném počítači není třeba žádný agent. Je možnost nakonfigurovat Ansible, aby pro připojení nepoužíval OpenSSH, ale i kerberos nebo LDAP.

Ansible a jemu podobné nástroje se použijí v případě toho, že máme více serverů, nebo pokud budeme dodržovat trend IaaS. Kdyby měl admin ve firmě nasazovat nebo aktualizovat například sto počítačů, tak na každém stráví 15 minut. To bude 1500 minut a to je 25 hodin. Proto použije Ansible a za 20 minut je hotov.

Pro příklad budeme instalovat Docker na 3 počítače. Na těchto počítačích budeme mít v souboru

keys naše veřejné klíče pro ssh. Na hostovi, ze kterého budeme instalovat je potřeba Ansible. Tedy `sudo apt install ansible`. V souboru `hosts`, který je pro Ansible tzv. inventářem naspecifikujeme stanice, na které budeme instalovat Docker.

### 1.2.1 Komponenty

#### 1.2.1.1 Control node

Jakýkoliv počítač s nainstalovaným Ansible a pythonem, může spouštět příkazy nebo tzv. playbooky. Tento počítač se nazývá control node. Takových můžeme mít klidně více, ne však počítače, které mají nainstalovaný operační systém Windows.

#### 1.2.1.2 Managed node

Je jakékoliv síťové zařízení. Managed nodes můžeme také nazývat jako hosts. Tyto zařízení nemusejí mít nainstalovaný Ansible, ale musejí mít nainstalovaný python. Ansible může být nakonfigurován, aby používal specifikovanou verzi pythonu, pokud není specifikována, spustí se na hostu jeho defaultní.

#### 1.2.1.3 Inventory

Je seznam všech nastavovaných zařízení. Často se nazývá `hostfile`. v tomto souboru nastavujeme skupiny zařízení, jejich IP adresy a další specifikace, například jaký python má daný host použít.

#### 1.2.1.4 Modules

Jsou to jednotlivé části kódu, které bude Ansible spouštět. Každý modul má speciální použití. Vše od správy uživatelů ( `user` ) přes nastavení systému (



systemd ) až k instalování balíčků ( apt, yum ). Můžeme spustit jeden modul v tasku, nebo více v playbooku. Pro přehlednost neuvádím všechny možné moduly, jelikož je jich přes tři tisíce.

#### 1.2.1.5 Tasks

Jsou jednotky, které se musejí provést. Nejčastěji specifikované v deployment souboru.

#### 1.2.1.6 Playbooks

Je seřazený seznam tasků, které se musí vykonat. Ničemu neuškodí pokud se playbook spustí znovu, protože Ansible skontroluje stav daného tasku. Playbooky jsou psané podle konvencí YAMLu.

## 1.3 Docker

Docker je otevřená platforma pro vývoj, dodání a spouštění aplikací. Umožňuje oddělení aplikací od infrastruktury, tedy můžeme dodávat software rychleji a bez problémů, které se váží k různorodosti prostředí, ve kterém aplikace běží. Svou fylozofií jsou velice podobné virtuálním počítačům. Rozdíly mezi těmito různými pohledy na věc budou rozebrány dále v textu.

Docker zprostředkovává platformu pro zabalení aplikace i se všemi jejími závislostmi. Izoluje danou aplikaci od ostatních běžících procesů na daném počítači a zajišťuje tak její bezpečí. Docker kontejner je velice nenáročný na hardware, můžeme jich tedy na daném počítači spustit velice mnoho.

Fylosofie kontejnerů je taková, že každý kontejner je odpovědný pouze za jednu danou část aplikace. Pro příklad máme naši webovou aplikaci. Budeme tedy mít alespoň tři docker kontejnery. Jeden na kterém poběží NGINX a bude zprostředkovávat naši aplikaci uživatelům. Další bude mít naši aplikaci a ve třetím poběží databáze.

Kontejnery fungují tedy jako malé počítače, mají izolované veškeré svoje systémové zdroje ( paměť, procesy, internetové rozhraní ). Díky tomuto mohou být rychle a jednoduše přidány, nebo odebrány.

### 1.3.1 Kontejner vs. virtuální počítač

Virtualizace je odpověď na problém různorodých prostředí mezi vývojáři a zákazníky. Problém ,který virtualizace a kontejnerizace především řeší je různorodost prostředí mezi zákazníkem a dodavatelem softwaru. Při jeho předávání dochází ke změně prostředí, jsou nainstalované jiné verze závislostí a operačního systému a aplikace se může chovat neočekávaně.

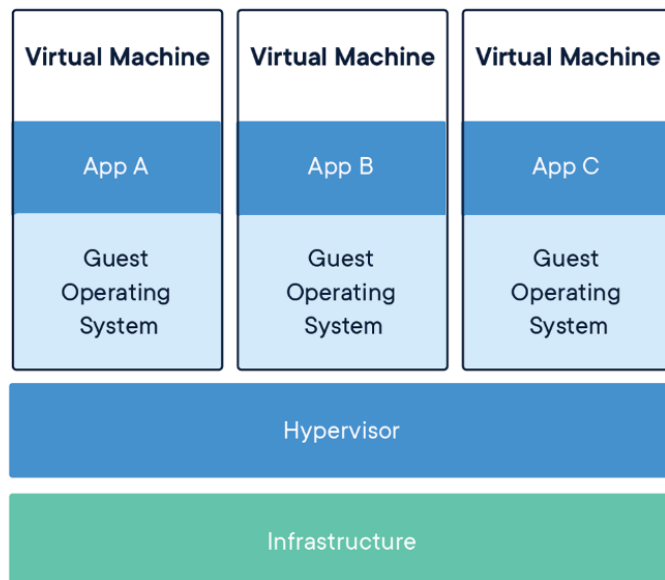
Podíváme se jak se tyto dvě technologie liší a proč se svět žene právě směrem kontejnerizace, když zde již je řešení.

## 1. CÍL PRÁCE

---

Virtuální počítač je regulérní stroj, který běží na daném hostovi. Tento stroj má svůj kernel, svůj operační systém a ke zdrojům přistupuje přes tzv. hypervizor např.: QEMU, nebo VirtualBox. Hypervizor zprostředkovává přístup virtuálního stroje k systémovým zdrojům.

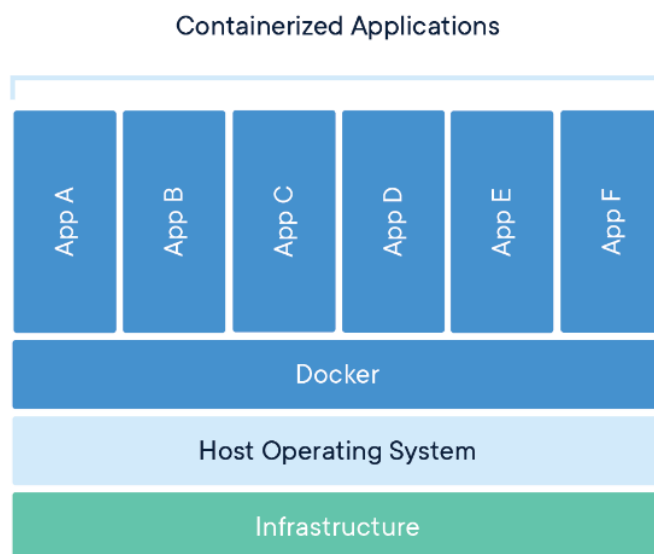
Pro to, aby na hostovi, nebo-li na systému, který má nainstalovaný hypervizor mohlo běžet více virtuálních strojů stačí jedna jeho instance. Nevýhoda tohoto řešení je taková, že se mnoho zdrojů duplikuje. Řekněme, že na hostitelském systému poběží tři aplikace. Každá taková aplikace bude izolovaná od ostatních pomocí virtuálního stroje. Dejme tomu, že to bude databáze, webový server a stroj pro vzdáleného uživatele. Níže uvidíme náskres tohoto řešení.



Obrázek 1.5: Oddělení aplikací za pomoci hypervizoru a virtuálních strojů.

To samé, jako je na obrázku výše se pokusíme realizovat pomocí Dockeru a kontejnerů. Kontejnery, jelikož využívají overlayFS jsou schopny poskytnout jádro operačního systému kontejneru bez zbytečné kopie a využívají copy-on-write funkcionality. Níže uvidíme jak za pomoci jmenných prostorů, kontrolních skupin a overlayFS je tento přístup úspornější a rychlejší než virtualizace.

Místo, které jsme na hostitelském systému ušetřili však není jediná výhoda. Na tomto příkladu se však rozdíl mezi těmito technologiemi vysvětluje nejlépe. Dalšími výhodami je rychlost spuštění kontejneru a virtuálního stroje. Při spuštění se pouze připojí obraz OS, vytvoří se izolované procesy a popřípadě se omezí i zdroje, které má kontejner využívat. Nehledě na to, že pokud kontejner nemá omezení nebo limit využitých zdrojů alokuje si je dynamicky



Obrázek 1.6: Oddělení aplikací za pomoci Dockeru

oproti virtuálnímu počítači, který si pro sebe naalokuje danou paměť při spuštění.

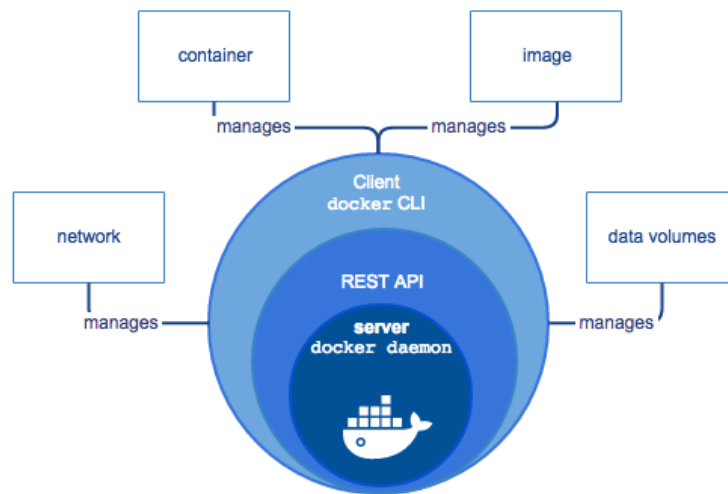
### 1.3.2 Stavební kameny Dockeru

Jak je již uvedeno v předešlé kapitole, Docker využívá vychytávky linuxového kernelu pro svojí funkcionalitu. Díky tomuto perfektně funguje na počítačích, kde běží OS založený na Linuxu. V následujících sekcích budou tyto technologie blíže popsány a bude vysvětlena jejich důležitost.

#### 1.3.2.1 Jmenné prostory

Jmenné prostory zastřešují veškeré zdroje systému tak, že každý proces spuštěn v daném prostoru může používat pouze prostředky, které se váží k tomuto prostoru. Každému procesu se to jeví tak, že má svoje vlastní globální prostředky, které mohou vidět i ostatní procesy z jmenného prostoru, ale ne z jiného. V tabulce 1.1 je možné vidět, jaké jmenné prostory lze v Linuxu nalézt.

Při spuštění kontejneru dojde k vytvoření procesu na hostitelském systému. Procesy dostanou od systému nějaké PID a chovají se jako normální procesy. Pokud se však přihlásíme do kontejneru (command: `docker exec -it name bash`) a podíváme se na procesy běžící v daném kontejneru uvidíme, že procesy mají jiná PID a určitě mají i PID=1. Toto nám umožňuje jmenné prostory.



Obrázek 1.7: Docker a jeho komponenty.

Tabulka 1.1: Linuxové jmenné prostory

Jméno	Popis
Cgroup	Cgroup root adresář
IPC	Systém pro komunikaci procesů, POSIX fronty
Network	Síťové rozhraní, protocols, ports, etc
Mount	Připojená zařízení
PID	ID procesů
User	Uživatelská ID a ID skupin
UTS	Hostname a NIS doménu

Každý kontejner může mít svůj vlastní souborový systém a svoje síťové rozhraní. Vše co můžeme oddělit mezi hostitelem a kontejnery je uvedeno v tabulce výše.

### 1.3.2.2 Kontrolní skupina

Je to vlastnost Linuxového kernelu. Jejich hlavní funkcí je limitovat zdroje. V Dockeru se používají protože dovolují sdílet prostředky mezi hostitelským systémem a dalšími kontejnery.

Často dochází k záměně pojmů mezi kontrolními skupinami a jmennými prostory. Znovu to tedy shrňme. Kontrolní skupin, nebo-li cgroups omezují co můžeme použít a jmenné prostory nebo-li namespaces omezují co jsme schopni vidět v systému.

### 1.3.2.3 Docker daemon

Docker daemon nebo-li dockerd poslouchá dotazy na docker API a spravuje objekty jakou jsou docker obrazy, kontejnery, síť a úložiště. Komunikuje ale i s dalšími daemony, aby byl schopen řídit službu Docker.

### 1.3.2.4 Docker klient

Je to primární cesta, jak komunikovat s Dockerem. Když použijeme příkazy, jako jsou "docker run", klient odešle příkazy daemone zmíněného výše.

### 1.3.2.5 Docker registr

Docker registr je úložiště pro naše Docker obrazy. Bez předchozího nastavení hledá dockerd obrazy, které chceme spustit ve veřejném Docker registru. Obrazy však mohou být dostupné i lokálně, nebo na nějaké jiné službě, např.: gitlab container registry.

Do styku s registrem přicházíme hlavně ve chvílích, kdy provádíme příkazy docker pull, docker push a docker run. Tyto příkazy vždy potřebují znát obraz, který bude spuštěn jako základní vrstva pro nový kontejner, nebo bude stáhnut na lokální počítač, či nasdílen do registru.

### 1.3.2.6 Obrazy

Můžeme si to představit jako šablonu, na které je spuštěn kontejner. Obraz může být složen z vícero obrazů, nebo z nich vycházet.

Pro vytvoření obrazu je třeba soubor Dockerfile. Tento soubor obsahuje jednoduché kroky, které je třeba vykonat pro vytvoření konkrétního obrazu. Např.: jaké použijeme a zveřejníme porty, jaké balíčky chceme ve vytvořeném obrazu mít atd.

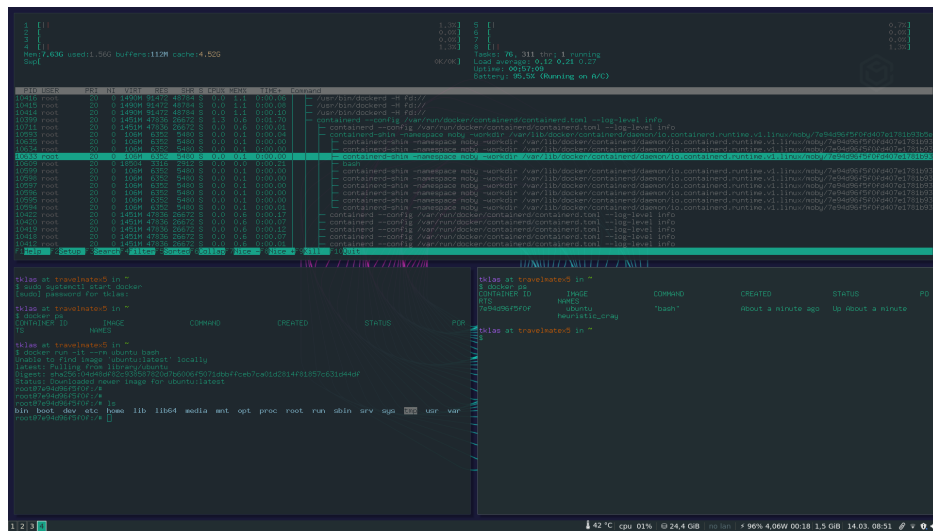
Každý příkaz v Dockerfilu vytvoří na lokálním počítači tzv. vrstvu, kterou při úpravě Dockerfilu mění nebo předělává pouze pokud byla změněna.

## 1.3.3 Systémová kontejnery

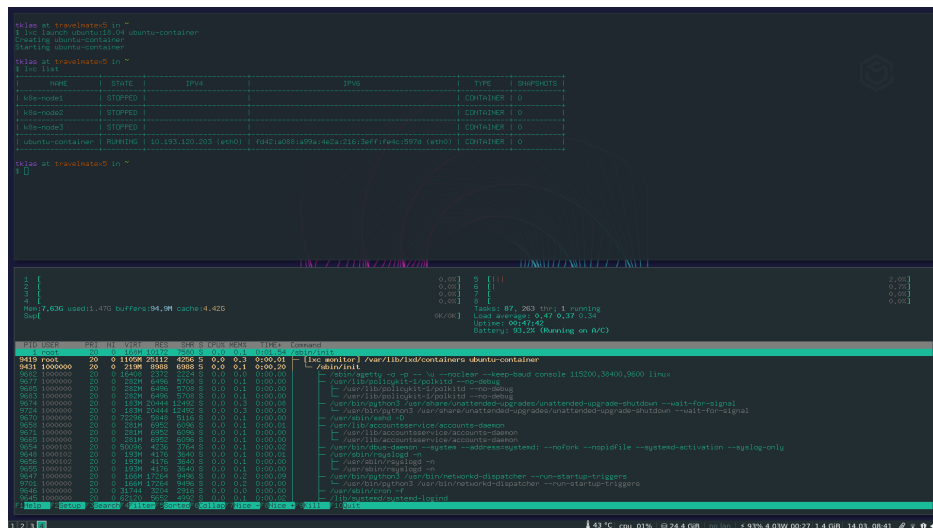
Docker kontejnery nejsou však jediné, které se v produkčním prostředí používají. Patří do tzv. aplikačních kontejnerů. Jejich účel je zpravidla spouštět pouze jeden proces. K takovýmto kontejnerům můžeme ještě přidat kontejner z Rocket. Hlavním rozdílem je to, že rtk nemá na systému spuštěného daemona jako má např. Docker. Při spuštění se tedy pod běžícím spustí další.

Dále tady máme systémové kontejnery. Ty jsou používány jako klasické OS. Na jednom silném stroji může běžet několik takových kontejnerů a ty mohou uživateli poskytovat oddělené prostředí od celého serveru a nabídnout mu izolovaný prostor od ostatních pomocí výše zmíněných technologií. Tuto možnost zastřešuje projekt LXC později LXD.

## 1. CÍL PRÁCE



Obrázek 1.8: Docker a jeho procesy.



Obrázek 1.9: LXC a jeho procesy.

## Hesla

Hesla můžeme vidět všude a ne jen v informatice. Pokud se podíváme zpět do historie např. do doby velkého Caesara a jeho šifry, ke které je třeba znát číslo, o které se posouvají znaky ve zprávě. Jak tedy můžeme vidět, hesla neslouží pouze k naší autentizaci vůči nějaké službě či serveru. Můžeme je také použít k podepsání citlivých dokumentů jako je třeba příloha e-mailu. Následně pak nemůžeme popřít jeho poslání. Tomuto se říká elektronický podpis.

Hesla však mají nejednu nevýhodu. Útočník může s naším, nebo i bez našeho vědění odhalit naše heslo a tím nám narušit naše soukromí. Hesla mohou také být v systémech, které používáme uložena nepatřičným způsobem, jako je například čistý text bez použití žádných ochranných prostředků.

Hesla též mohou ze systému uniknout. V tomto případě, pokud byla hesla uložena nepříčným způsobem nemusí se potenciální útočník nějak přemáhat, aby uživatele kompromitoval. Proto se zaměříme na to jak mohou a jak skutečně jsou uložena.

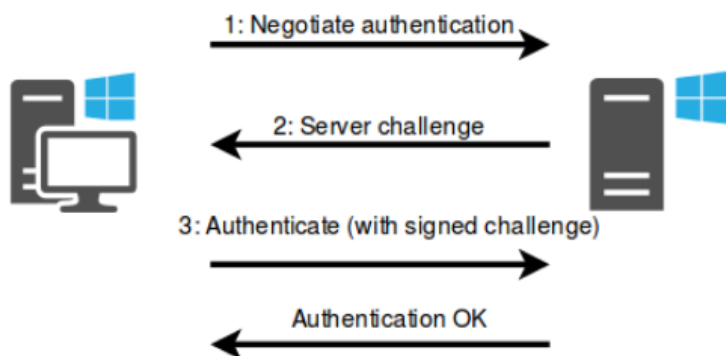
### 2.1 Windows

Windows se chovají jinak v doméně a jinak mimo ní. Pokud je počítač v doméně je preferován autentizační protokol kerberos. V doméně je nastaven jeden nebo více autentizačních serverů přes které se uživatel ověřuje. V současných Windows Server edicích je implementován Kerberos verze 5. Kerberos v základní nastavení operuje na portu 88 a k šifrování používá symetrickou šifru. Pokud počítač není nastaven aby se autentikoval pomocí protokolu Kerberos používají Windows šifrování NTLM. Tento protokol autentizace není oficiálně dokumentován, byl však popsán v projektu Samba. Současná nejnovější verze je označena NTLMv2.

### 2.1.1 Popis protokolu

Protokol se řídí sekvencí výzva-odpověď, která vyžaduje, aby mezi klientem a serverem byly vyměněny celkem tři zprávy (three-way-handshake):

1. Klient odešle zprávu obsahující informace o klientem podporovaných nebo požadovaných funkcích (velikosti kryptovacích klíčů, požadavek na vzájemnou autentizaci atd.)
2. Server odpoví zprávou obsahující podobné informace o serverem podporovaných nebo požadovaných funkcích (čímž klient dokáže vybrat vhodné autentizační parametry) a - nejdůležitější část - náhodnou výzvu (8 bytů, tzv kryptografická sůl).
3. Klient pak použije výzvu ze zprávy, uživatelské jméno a heslo k výpočtení odpovědi. Volba výpočetní metody je závislá na autentizačních parametrech dohodnutých předešlou zprávou, nicméně obecně lze říci, že aplikuje hašovací funkce MD4 nebo MD5 a šifrování DES.



Obrázek 2.1: NTLM protokol pro autentizaci.

## 2.2 Linux

Linux jakožto systém, který se skládá pouze ze souborů má též hesla uložena v jednom souboru `/etc/shadow`. Hesla v linuxu bývala dříve uložena společně v `/etc/passwd`. Kvůli bezpečnosti se však oddělila a v souboru byl otisk nahrazen písmenem `x`. Hesla jsou uložena pomocí zvolené hašovací funkce. V tabulce níže uvidíme podporované funkce.



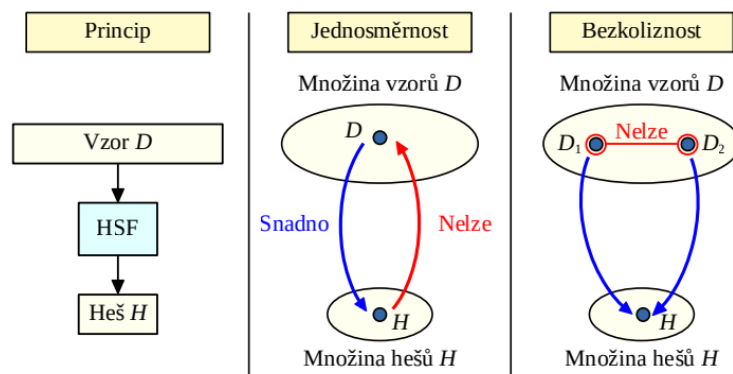
Tabulka 2.1: Hašovací funkce

ID	Funkce
1	MD5
2a	Blowfish
5	SHA-256
6	SHA-512

## 2.3 Hešovací funkce

Hešovací funkce je kryptografická funkce, která argumentu o prakticky libovolné délce přiřazuje tzv. heš  $H$ , což je číselná hodnota o pevně stanovené délce (typicky o délce 256 až 512 bitů). Od hešovací funkce se vyžadují dvě specifické vlastnosti:

- **Jednosměrnost**, což znamená, že určení hodnoty heše je pro zadaný vzor výpočetně snadné, avšak určení hodnoty vzoru ze znalosti jeho heše je prakticky nemožné.
- **Bezkoliznost**, což znamená, že je prakticky nemožné nalézt nějakou dvojici různých vzorů takovou, aby jejich heše byly stejné. V této souvislosti je zapotřebí si uvědomit, že počet číselných posloupností libovolné délky (tj. počet vzorů) je vždy větší než počet posloupností jediné možné délky. Z toho pak plyne, že mnoho vzorů musí mít stejný heš a vznikají tak kolize. Požaduje se však, aby nalezení kolize bylo prakticky nemožné.



Obrázek 2.2: Hešovací funkce a její vlastnosti.

Kryptografické využití hešovacích funkcí:

- Kontrola integrity (kontrola shodnosti velkých souborů a dat).

## 2. HESLA

---

- Automatické dešifrování (souboru, disku apod.).
- Ukládání a kontrola přihlašovacích hesel.
- Prokazování autorství.
- Jednoznačná identifikace dat (jednoznačná reprezentace vzoru, digitální otisk dat, jednoznačný identifikátor dat, to vše zejména pro digitální podpisy).
- Prokazování znalosti.
- Autentizace původu dat.
- Nepadělatelná kontrola integrity.
- Pseudonáhodné generátory, derivace klíčů.

### 2.3.0.1 MD5

### 2.3.0.2 SHA-1

Algoritmus SHA (Secure Hash Algorithm) byl zavedený NIST a publikovaný v normě FIPS 180 (1993). Revidovaná verze SHA-1 v normě FIPS-180-1 (1995). SHA-1 umožňuje zpracovat zprávu s zpráva M je zpracována po blocích z velikostí 512b. Svým vnitřním fungováním je velice podobná MD5. Liší se zejména v velikosti výstupního řetězce.

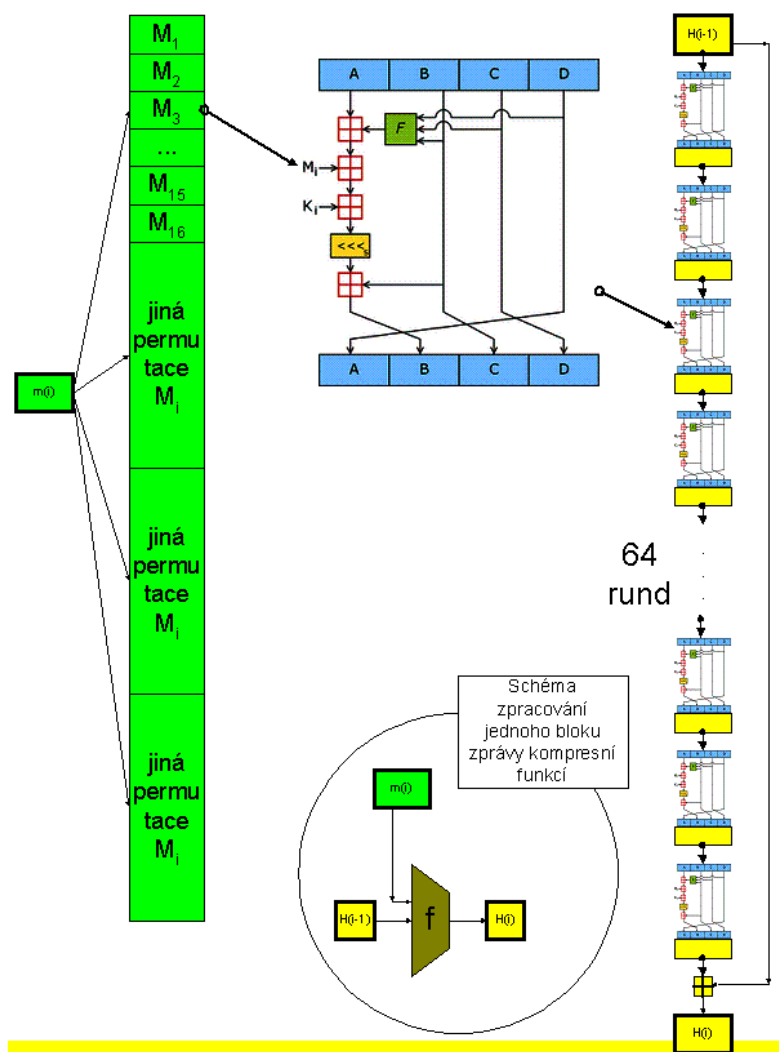
Při inovaci standardu FIPS 180-1 na FIPS 180-2 byly zavedeny 3 nové hašovací algoritmy: SHA-256, SHA-384 a SHA-512. Jejich rozdíly je možné vidět v tabulce.

Tabulka 2.2: SHA-X vlastnosti

	SHA-1	SHA-256	SHA-384	SHA-512
Délka haš. kódu	160	256	384	512
Délka zprávy	<264	<264	<2128	<2128
Velikost bloku	512	512	1024	1024
Velikost slova	32	32	64	84
Počet rund	80	80	80	80
Bezpečnost v bitech	80	128	192	256

## 2.4 Útoky na hesla

Útoky na hesla jsou na denním pořádku na většinu z nás. Pokud se koukneme do blízké minulosti uvidíme, že firmy jako jsou např.:



Obrázek 2.3: Hešovací funkce MD5.

- **Yahoo**, které v roce 2013 a 2014 unikly data milionů uživatelů. V roce 2013 to byl veliký problém, protože hesla nebyla pečlivě hešována. V roce 2014 se však již poučili a hesla zahešovali bcrypt algoritmem.
- **Marriott International**, útok se uskutečnil v roce 2014. Byl však objeven až v roce 2018. Během této doby bylo kompromitováno okolo 500 miliónů uživatelů.
- **eBay**, roku 2014 unikly jména, adresy, data a hesla přes 145 miliónům uživatelů. Útok vyžadoval po zákazníkovi změnu hesla, ale přidal i kolonku na citlivé informace, které byly ukládány.

- **Sony's PlayStation Network**, která byla napadena v roce 2011. Ztráty se odhadují na 171 miliónů dolarů. 12 miliónů kreditních karet, které byly uloženy bez jakéhokoliv zabezpečení bylo kompromitováno.

Jak lze vidět i v 21 století se hesla a další citlivé informace ukládají s velkou neopatrností a ledabílostí a to i u takových to známých firem. Seznam by mohl pokračovat do nekonečna, protože každým dnem útoků jen přibývá.

Dále se podíváme na jedny z nejpoužívanějších praktik získávání a lámání hesel. V praktické části bude použit program Hashcat na provádění některých z níže uvedených útoků.

- **Slovníkový útok** využívá znalosti toho, že uživatelé tíhnou k používání hesel, jež jsou dobře známá slova. Pokud je politika použitého hesla přísnější, na začátku bude velké písmeno a na konci číslovka. Nemusí nad nimi poté těžce přemýšlet, jako nad náhodně vygenerovanými řetězci. Útočník potřebuje soubor, který obsahuje známá hesla. Takový není těžké najít kdekoli na internetu. K testovaným útokům pomocí slovníku bude v pozdější části využit soubor rockyou.txt.
- **Hrubá síla** již není tak efektivní, jako bývala před lety, kdy se začali používat a nutit po uživateli hesla s alespoň 8 znaky. Tento útok je ve své podstatě nejjednodušší. Útočník spustí program, který mu generuje náhodné sekvence čísel a písmen ze sady, která byla použita při psaní hesla a zkouší veškeré možnosti. Jednoho dne se tedy určitě dostane na správnou posloupnost, ta se však s každým přidaným znakem exponenciálně vzdaluje.
- **Odposlech komunikace.** Útočník odposlechne komunikaci. Pokud je taková komunikace nešifrována je jednoduché z paketu získat heslo. Samozřejmě to nemusí být pouze heslo, které se po síti přenáší. Proti takovým útokům se používá například SSL (Secure Sockets Layer) protokol, ten zajistí šifrování komunikace mezi subjekty.
- **Man In the Middle**, nebo-li MITM útok. Útočník se dostane do prostoru mezi uživatelem a např.: autentizačním serverem. Rozdíl mezi odposlechem je ten, že se rovnou napojí na linku a směřuje provoz přes své zařízení. Takový útok je např. DNS sniffing.
- **Key logger attack.** Útočník nainstaluje software na hostitelský systém, který ukládá veškeré zmáčknutí klávesnice a následně data získá a analyzuje. Nemusí se však jednat pouze o SW, existují i USB rozbočovače, které se připojí mezi klávesnici a USB port na systému.
- **Sociální inženýrství** je odvětví útoků, které cílí na neznalost uživatele, nebo jeho neopatrnost. Mezi takové patří:

1. **Phishing**, se zaměřuje na kontaktování uživatele a donutí jej kliknout na závadný odkaz, či stáhnout např. kez logger skrze e-mail.
2. **Baiting**, útočník nechá na chodníku kde se oběť pohybuje pohozené médium s útočným softwarem a počká až jej oběť sebere a dá jej do svého systému.
3. **Quid quo pro**, kdy se útočník vydává za technického zaměstnance a interaguje s obětí a donutí ji mu poskytnout cestu do systému.

## 2.5 Entropie hesla



---

## Závěr

V práci byly rozebrány a vysvětleny všechny potřebné pojmy tak, aby čtenář byl schopen pochopit jak fungují technologie použité k sestavení klastru. Klastř byl nasazen a spuštěn s danými útoky. Klastř je popsán a jejasně znázorněno jak byl sestaven a spravován. Výsledky lámání hesel jsou analyzovány a okomentovány. Na přiloženém médiu jsou zdrojové kódy potřebné pro lokální replikaci a otestování.

[1]1





---

## Literatura

- [1] D. Merkel, “Docker: lightweight linux containers for consistent development and deployment,” *Linux journal*, vol. 2014, no. 239, p. 2, 2014.



## Seznam použitých zkratk

**GUI** Graphical user interface

**XML** Extensible markup language



## Obsah přiloženého CD

	readme.txt.....	stručný popis obsahu CD
	exe .....	adresář se spustitelnou formou implementace
	src	
	impl.....	zdrojové kódy implementace
	thesis .....	zdrojová forma práce ve formátu L <sup>A</sup> T <sub>E</sub> X
	text .....	text práce
	thesis.pdf .....	text práce ve formátu PDF
	thesis.ps .....	text práce ve formátu PS