FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Kubernetes klastr pro lámání hesel
Student:	Tomáš Klas
Vedoucí:	Ing. Jiří Buček, Ph.D.
Studijní program:	Informatika
Studijní obor:	Bezpečnost a informační technologie
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

Cílem práce je vytvoření výpočetního klastru pro lámání hesel, který bude řízen za pomoci Kubernetes. Jednotlivé uzly budou moci být přidány a odebrány dle potřeby. Uzly se budou připravovat pomocí technologie Ansible. Práce na lámání hesel bude nadřízeným uzlem přidělována výpočetním uzlům (workerům). Ty budou následně ve vytvořeném Docker kontejneru provádět přidělenou práci a vrátí nadřízenému uzlu výsledky lámání. Lámání hesel bude vykonávat software hashcat, který může být optimalizovaný i na grafické karty.
Popište použité technologie (Kubernetes, Ansible, Docker, hashcat) a prozkoumejte existující řešení problému.
Popište běžně používané útoky na hesla a formáty ukládání hesel.
Vytvořte a popište sestavený klastr.
Otestujte vytvořené řešení s různými metodami útoku (hrubou silou, slovníkový, s maskou).
Analyzujte výkonnost jednotlivých metod lámání hesel na použitém klastru vzhledem k délce a složitosti hesla a počtu výpočetních uzlů.

Seznam odborné literatury

Dodá vedoucí práce.

prof. Ing. Pavel Tvrdík, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 12. února 2020

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.

4. června 2020

Poděkování

Rád bych poděkoval svému vedoucímu, panu Ing. Jiřímu Bučkovi, Ph.D., který mě podpořil svým nadšením a ochotnou pomocí. Též bych rád poděkoval své rodině, která mě zajišťovala v průběhu studia a všem svým přátelům, kteří mi byli na blízku. Chtěl bych také poděkovat svým kolegům, od kterých jsem se mnohé naučil.

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 předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, 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 zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). 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 zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 4. června 2020 . . . . . .

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. Samotný program pro lámání využívá ke své správné funkcionalitě mikroslužby, jež jsou zapouzdřeny pomocí technologie Docker. Použité technologie jsou v práci detailně popsány a rozebrány. Dále se práce zabývá ukládáním hesel v současných systémech a tím, jaká hesla se používají a jak nejčastěji vypadají. Na závěr je na klastru proveden test různých metod pro lámání hesel. Tyto metody jsou popsány a je 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 set up a cluster managed by Kubernetes for password recovery. The next step is to describe used technologies like Docker, Ansible, and Hashcat. The thesis contains a description of how the passwords are stored and the most known attacks to crack them. Successful deployment and password cracking lead to analyzing the speed of the cluster and the particular cracking method.

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

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Tak jako my, se svět posouvá dopředu a s ním i používané technologie. Technologie, které se zrodily v nedávných letech a jsou mně i mým kolegům neznámé, jsou však v současné době používané standardně pro veliké firmy, jako je např.: Google, Facebook, Datamole s.r.o.

Nesmíme si tedy dovolit, takové technologie ignorovat. Proto vznikla tato práce. Snaží se dostihnout v praxi používané technologie na vysoké škálování. Proniká do světa mikroslužeb, které spravuje pomocí Kubernetes, což je platforma na správu kontejnerizovaných aplikací.

Problém však nastává, pokud chceme vytvořit klastr čítající stovky počítačů. Tento problém řeší Ansible a jeho schopnost nasadit přesně specifikovaný software vzdáleně na jakýkoliv počítač, ke kterému máme přístupová práva, a připravit tak počítač pro Kubernetes. Ansible je tedy nástroj pro vzdálené nasazení a správu zařízení.

Svět mikroslužeb se neobejde bez správného zapouzdření, které dodá službě vždy stejné prostředí. Služby tedy zabalíme pomocí Dockeru. Výsledkem bude obraz, který je možné pomocí API, které Docker poskytuje spustit na jakémkoliv zařízení, které má Docker nainstalovaný.

Práce tyto technologie rozebere, představí a též vysvětlí pojem mikroslužeb a způsob návrhu takového řešení na jednoduchém příkladě, a to lámání hesel na různé způsoby. Lámání bude provádět různými metodami a bude analyzována rychlost lámání v závislosti na různých faktorech.

Kapitola 1

Kubernetes

Jméno Kubernetes pochází z Řecka a znamená kormidelník. Projekt založili Joe Beda, Brendan Burns, a Craig McLuckie, ke kterým se rychle připojili inženýři z Googlu, jako Brian Grant a Tim Hockin. První verze softwaru byla vydána v roce 2014. [2]

Kubernetes jsou vytvořeny pro lehčí správu mikroslužeb a kontejnerizovaných aplikací. Tento přístup je opakem k monolitické aplikaci. Výhodou je možnost agilního vývoje a vydávání nových verzí pouze pro část aplikace. Pokud je aplikace složena z malých částí, které musí někde běžet a jsou jich pak nasazené stovky, člověk ztratí přehled. [3]

V tuto chvíli se musíme uchýlit k nějakým nástrojům, které pomáhají takové mikroslužby řídit. Mezi takové patří například Nomad, Rancher, nebo Docker Swarm. Výhodou Kubernetes je jejich rozšiřitelnost a deploymenty psané v YAML[1] formátu. Co přesně Kubernetes tedy umí?

- **Service discovery a load balancing**: Kubernetes propojují kontejnery skrze DNS[2], 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. Jelikož Kubernetes klastr, je často služba, zprostředkovávána od cloudových poskytovatelů, jako jsou Azure, nebo AWS, má každá platforma možnost lehce připojit úložistě, které sama poskytuje.

- **Automatic rollouts a rollbacks**: Můžeme popsat požadovaný stav pro naše nasazené kontejnery pomocí Kubernetes a ty automaticky a s kontrolou zajistí tento stav. Můžeme například automatizovat vytváření no-

[1] YAML - Jazyk pro serializaci dat, který je možno číst nejen pomcí stroje.
[2] DNS - Domain Name Systém, slouží k překladu doménových jmen na IP adresy.

vých kontejnerů, smazání starých a převzetí všech zdrojů, jako jsou data a další, nově spuštěným kontejnerem.

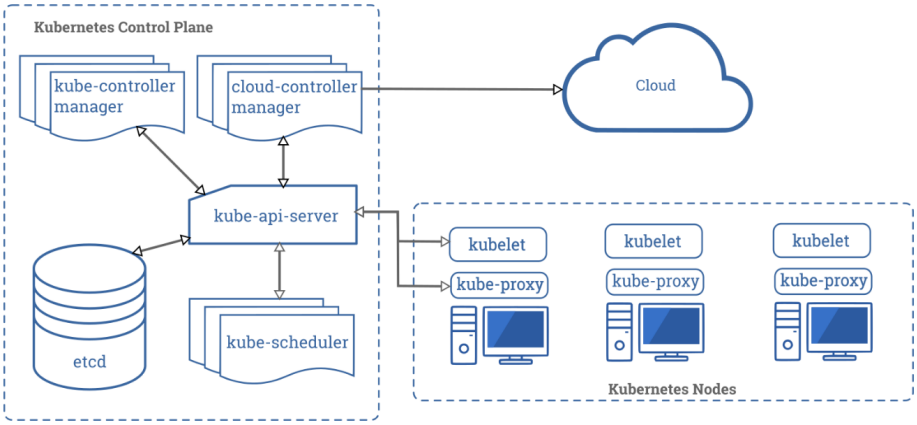
- **Automatic bin packing**: Předáme Kubernetes vytvořený klastr s uzly a specifikujeme zdroje, které každý kontejner potřebuje pro své správné fungování. Kubernetes se následně postará o nejlepší rozdělení kontejnerů na uzly tak, aby optimalizoval naše zdroje. To se může nejvíce hodit na cloudových řešení, kde se platí za to, kolik se využívá zdrojů v danou dobu, nebo pokud jsou naše zdroje limitovány.

- **Self-healing**: Kubernetes restartují kontejnery, které selžou, nahradí je a zahodí pokud přestanou odpovídat na specifikované health checky a přestane je nabízet klientům, nebo jiným službám dokud nejsou připravené.

- **Secrets and configuration management**: Kubernetes napomáhají ukládání a sdílení, či měnění citlivých informací zkrze klastr. Při změně těchto informací tedy nemusíme znovu vytvářet kontejnery a nejsme nuceni tyto informace balit do služeb a obrazů. [2]

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.

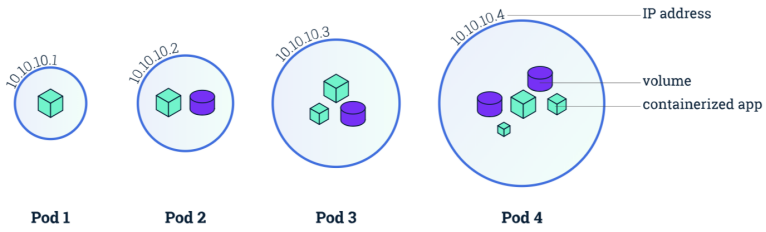
Tyto výpočetní nody poskytují prostor pro pody. Pod si můžeme představit jako balík, ve kterém jsou umístěné kontejnery. V praxi je nejčastěji Control Plane rozdělen na více počítačů, aby se zajistila vysoká dostupnost a maximalizovala se odolnost proti chybám.



Obrázek 1.1: Komponenty Kubernetes [4]

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 1.2, nastíňující, 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 je aplikace navržena. Abychom neporušovali konvence tzv. service architektury, musíme do jednoho podu dávat služby pouze takové, které dávají smysl. Nebudeme do jednoho podu tedy balit databázi s webovým UI[3]. Při škálování by se totiž nejen že replikovala aplikace s UI, ale i databáze a data by pak nemusela být konzistentní.



Obrázek 1.2: Z čeho se skládá pod [5]

[3]UI - User Interface.

Příklad deploymentu podu pro lámání hesel:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: computation-unit
spec:
  selector:
    matchLabels:
      run: computation-unit
  replicas: 1
  template:
    metadata:
      labels:
        run: computation-unit
    spec:
      hostname: hashcat-pod
      containers:
      - name: hashcat
        image: |
         registry.gitlab.com/kazyklas/password-cracking-cluster/
         hashcatwrapper:dictionary
```

1.2 Control Plane

Součásti 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. [2]

- **kube-apiserver**: Komponenta poskytuje API[4] uživateli, jako frontend 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 a spolehlivost.

[4]API - Application programming interface, slouží ke komunikace mezi službami.

- **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, specifikace affinity a anti-affinity, umístění uložených dat, vnitřní vytížení a čas.
- **kube-controller-manager**: V této komponentě je spuštěn proces pro kontrolu klastru. Skládá se z více částí, které jsou zkompilovány do jednoho programu, aby mohly běžet v jednom procesu, který se o vše stará. A stará se o následující:
  1. **Node Controller**: Je odpovědný za upozornění, pokud jeden z nodů přestane odpovídat.
  2. **Replication Controller**: Je odpovědný za dodržování správného počtu spuštěných podů v klastru. Pokud tedy část aplikace spadne, je odpovědný za start nové instance.
  3. **Endpoints Controller**: Přidá nové 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 jmenné prostory.
- **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 připoji k jakémukoliv poskytovateli, který implementuje tzv. cloudprovider.interface.

1.3 Komponenty nodu

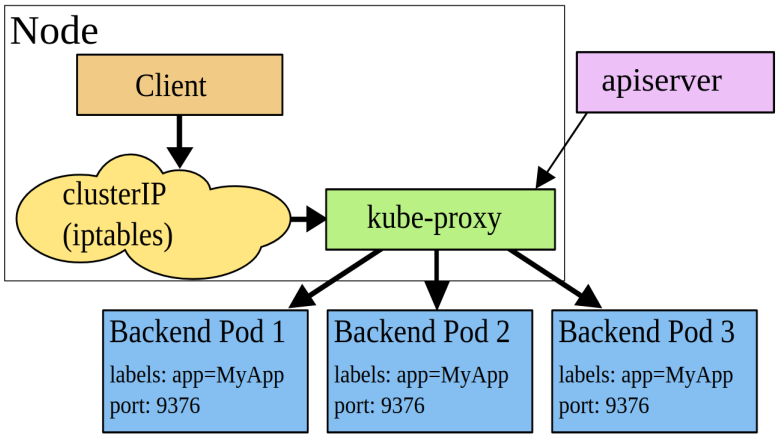
Node je zařízení, které je připojeno do klastru. Na tomto nodu mohou běžet pody, což je balík kontejnerů. Níže je popsáno, z čeho se tento node skládá a co na něm musí běžet, aby Kubernetes správně operovalo.

1.3.1 Kubelet

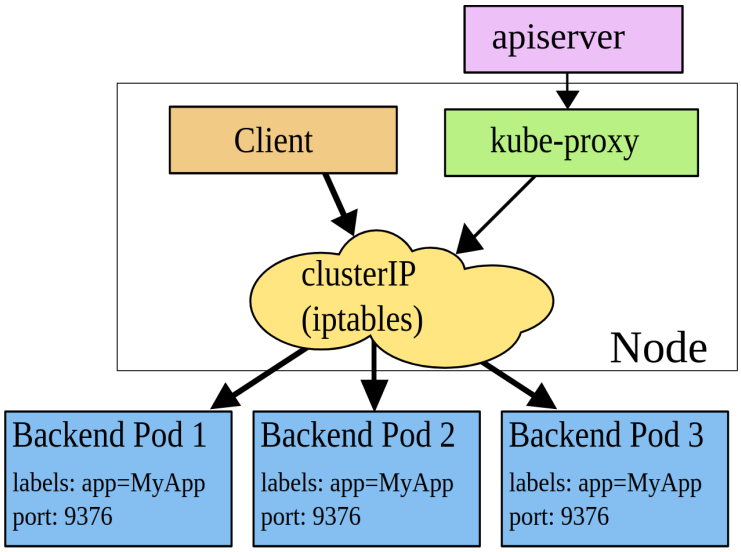
Je agent zajišťující běh všech kontejnerů v daném podu. Kubelet převezme množinu požadavků na daný pod, tzv. PodSpecs a zajistí, že všechny kontejnery popsány v těchto specifikacích jsou spuštěné a odpovídají na health checky.

1.3.2 Kube-Proxy

Je síťová proxy, která spravuje síťová pravidla. Zajišťuje tak komunikaci mezi jednotlivými pody a nody uvnitř klastru. Používá systémovou vrstvu na filtrování paketů, pokud je dostupná, jinak forwarduje provoz za pomoci svých prostředků. Znázorněno na obrázcích 1.3 a 1.4.



Obrázek 1.3: Kubernetes proxy - User space proxy mode [6]



Obrázek 1.4: Kubernetes proxy - iptables proxy mode [7]

1.4 Mikroslužby

Aplikace v Kubernetes se skládají z mikroslužeb. Mikroslužba je část aplikace, která se stará o specifickou funkcionalitu. Opak takového řešení představuje monolitická aplikace, která zajišťuje veškerou funkcionalitu dané aplikace.

Takové řešení má své výhody i nevýhody. Mezi výhody patří například možnost flexibilnějšího agilního vývoje, kde část, nebo samostatný tým vyvíjí část aplikace, neboli jednu mikroslužbu. Při vydání nové verze, kdy tým dokončí vývoj nové funkcionality, je její nasazení mnohem jednodušší, protože se nasadí pouze část aplikace.

Jelikož jsou části aplikace od sebe odděleny, je možné psát různé části v různých jazycích. Z jednoho úhlu pohledu se to může tvářit jako jedna z výhod, ale z pohledu udržování kódu ve více jazycích se může jednat o nevýhodu.

Mikroslužby mezi sebou musí nějakým způsobem komunikovat, typicky s využitím jednoho ze standardních protokolů (REST API přes HTTP[5] nebo HTTPS[6] popř. jiný protokol pro komunikaci), jejich fungování je náročnější na síťové zdroje a tedy mnohem dražší. Též je nutné ve firmě přistupovat k vývoji softwaru velice agilně a je nutné mít v týmu člověka, který se vyzná jak ve zdrojovém kódu, tak v nástrojích na správu mikroslužeb. [3]

[5]HTTP - Hypertext Transfer Protocol.
[6]HTTPS - Hypertext Transfer Protocol Secure.

Kapitola 2

Ansible

Ansible je automatizační nástroj pro konfiguraci systémů, nainstalování softwaru a aktualizací. Jeho nejsilnější stránka je nulový výpadek systému při aktualizaci balíčků nebo automatické nastavení daného zařízení. [8] V praxi se používá i pro aktualizaci konfiguračních souborů, např. aktualizace DNS záznamů na lokálním DNS serveru po přidání nového počítače do sítě. [9]

Hlavními cíli jsou jednoduchost a nenáročnost. Kód by měl být čitelný i pro lidi, kteří nejsou obeznámeni s programem a syntaxí. Je schopen pokrýt různě velké prostředí od malých podniků až po velice obsáhlou infrastrukturu. [8]

Ansible se připojí na vzdálený počítač pomocí OpenSSH a uživatele, který je současně přihlášen. Na spravovaném počítači není třeba žádný agent, pouze Python ve verzi 2.7 a vyšší, který ansible využívá pro spouštění modulů z jednotlivých úloh.

Ansible a jemu podobné nástroje se použijí v případě, kdy se infrastruktura sestává z více serverů, nebo pokud budeme dodržovat trend IaC[7]. Kdyby měl správce ve firmě nasazovat software, 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 na hosta: server.example.fit nainstalujeme Docker za předpokladu, že OS serveru je Ubuntu 18.04, nebo systém, který používá jako balíčkovací systém apt.

```
---
- name: Příklad instalace Docker na hosta
  hosts: server.example.fit
  remote_user: root
  tasks:
  - name: Instalace dockeru
    apt: name=docker.io state=present
```

[7] IaC - Infrastructure as a Code je trend v rozšiřujícím se cloudovém prostředí.

2.1 Control node

Jakýkoliv počítač s nainstalovaným Ansible a Pythonem může spouštět tzv. playbooky. Playbook je množina úloh, které se vykonávají v pořadí, jež jsou definovány a na zařízeních, na kterých se mají vykonat. [10]

Můžeme jich mít v infrastruktuře více. Tyto počítače v současné době však nemohou mít nainstalovaný OS Windows. Pro větší infrastruktury a lepší přehled existuje Ansible Tower od společnosti Red Hat. Ta pomáhá centralizovat playbooky a inventory hostů v jedné webové aplikaci. [10]

2.2 Managed node

Je jakékoliv síťové zařízení, které chceme spravovat. Managed nodes můžeme také nazývat jako hosty. Tato zařízení nemusejí mít nainstalovaný Ansible, ani žádný daemon poslouchající na speciálním portu, ale musejí mít nainstalovaný Python. K managovanému nodu musíme mít přístup přes internet a musíme být schopni se přihlásit na uživatele, pod kterým se mají spustit příkazy z playbooku. [10]

2.3 Iventory

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, nebo do jakých skupin jací hosti patří a pod jakým uživatelem si přejeme přihlásit a pracovat. [11]

2.4 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 instalovaní 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 a jsou k dispozici na oficiálních stránkách. [10]

2.5 Tasks

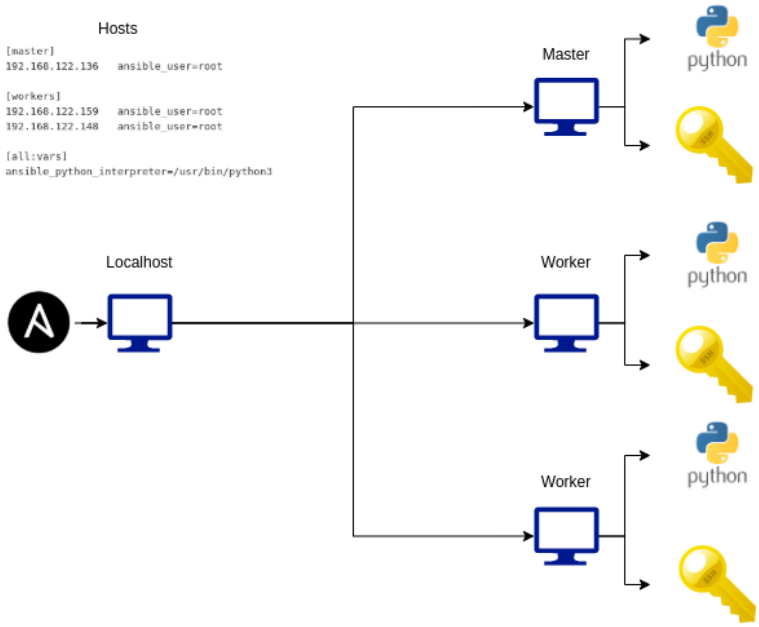
Jsou jednotky, které se musejí provést. Task může být: nainstalovat Docker a následně spustit kontejner na hostu. Takový task se skládá nejméně ze dvou modulů (apt, docker-container). Task se následně zakomponuje do playbooku, ze kterého bude vykonán. [10]

```
- name: Disabling swap partition for K8s.
  shell: swapoff -a

- name: Remove fstab entry for swap partition.
  replace:
    path: /etc/fstab
    regexp: '(.*swap)'
    replace: '#\1'
```

2.6 Playbooks

Je seřazený seznam tasků, které se musí vykonat. Ničemu neuškodí, pokud se playbook spustí znovu, protože Asnible zkontroluje stav daného tasku. Playbooky jsou psané podle konvencí YAMLu. Příklad takového playbooku je níže. [12]

```
---

- hosts: all

  tasks:

    - name: Gather required facts facts
      setup:

    - include: tasks/disable-swap-patition.yml

    - include: tasks/user-creation.yml

    - include: tasks/install-dependancies.yml
```



Obrázek 2.1: Shrnutí Ansible do obrázku

Kapitola 3

Docker

Docker je veřejná 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 filosofií jsou velice podobné virtualním počítačům. [13]

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. Pokud je aplikace běžící v kontejneru náročná na zdroje, je situace samozřejmě jiná. [13]

Filosofie 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[8] a bude zprostředkovávat naši aplikaci uživatelům. Další bude mít naší aplikaci a ve třetím poběží databáze.

Kontejnery fungují tedy jako malé počítače, mají izolované veškeré svoje systémové zdroje (paměť, procesy, internetové rozhraní atd.). Díky tomuto mohou být rychle a jednoduše přidány nebo odebrány. [14]

3.1 Kontejner vs. virtuální počítač

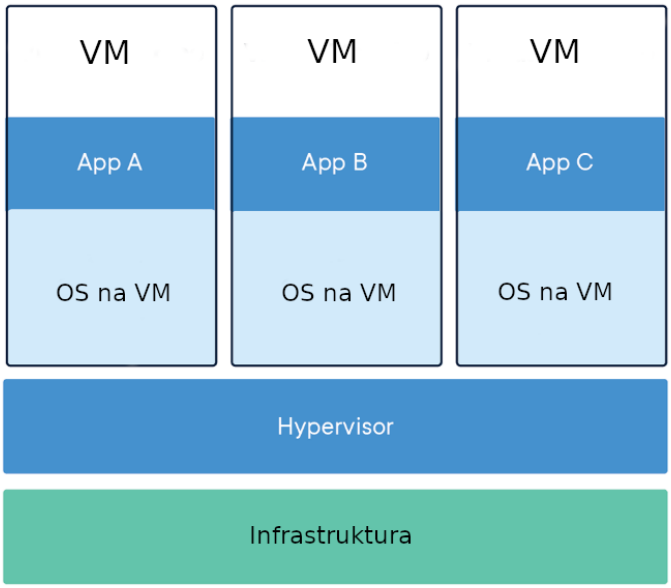
Virtualizace je odpověď na problém různorodých prostředí mezi vývojáři a zákazníky již dlouhou dobu. Problém, který virtualizace a kontejnerizace především řeší, je různorodost prostředí mezi zákazníkem a dodavatelem softwaru. Při jeho předávání dochází ke změně prostředí, jsou nainstalované jiné verze aplikací 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 již existuje řešení.

[8] Webový server a load manager s reverzní proxy.

Virtuální počítač je v podstatě regulérní stroj, který běží na jiném zařízení. Tento stroj má svůj kernel, svůj operační systém a ke zdrojům přistupuje přes tzv. hypervizor např.: QEMU, nebo VirtualBox.

Aby na systému, který má nainstalovaný hypervizor, mohlo běžet více virtuálních strojů, stačí jedna jeho instance. Nevýhoda řešení je ta, že se mnoho zdrojů duplikuje. Řekněme, že na hostitelském systému poběží tři aplikace. Každá aplikace bude izolovaná od ostatních pomocí virtuálního stroje. Dejme tomu, že to bude databáze, webový server a stroj pro vzdáleného uživatele. Nákres takové architektury 3.1.

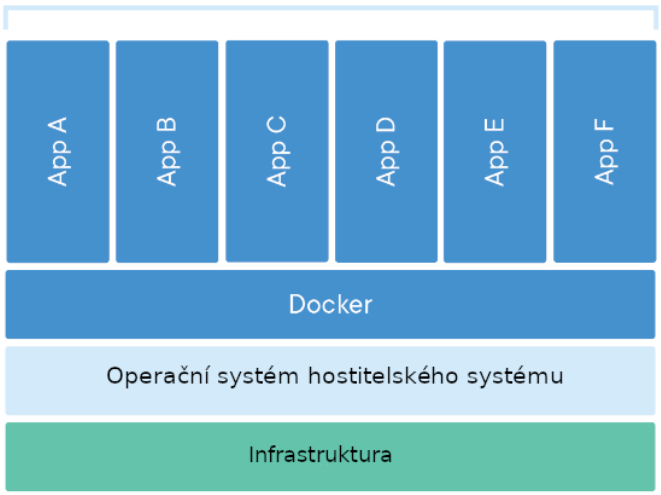


Obrázek 3.1: Oddělení aplikací za pomoci hypervizoru a virtuálních strojů [15]

To samé, jako je na obrázku 3.3, 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. Na obrázku 3.2 uvidíme jak za pomoci jmenných prostorů, kontrolních skupin a overlayFS je tento přístup úspornější než virtualizace.

Místo, které jsme na hostitelském systému ušetřili, není jediná výhoda. Dalšími výhodami je rychlost spuštění kontejneru a virtuálního stroje. Při spuštění se pouze připojí obraz OS[9], vytvoří se izolované procesy a popřípadě se omezí i zdroje, které má kontejner využívat. Pokud kontejner nemá omezení, nebo limit na využití zdrojů, alokuje si je dynamicky. Proti tomu virtuální počítač má zdroje omezené už při spuštění a při potřebě alokovat nové, zde takovou možnost nemá. [14]

[9]OS - Operační systém.



Obrázek 3.2: Oddělení aplikací za pomoci Dockeru [16]

3.2 Stavební kameny Dockeru

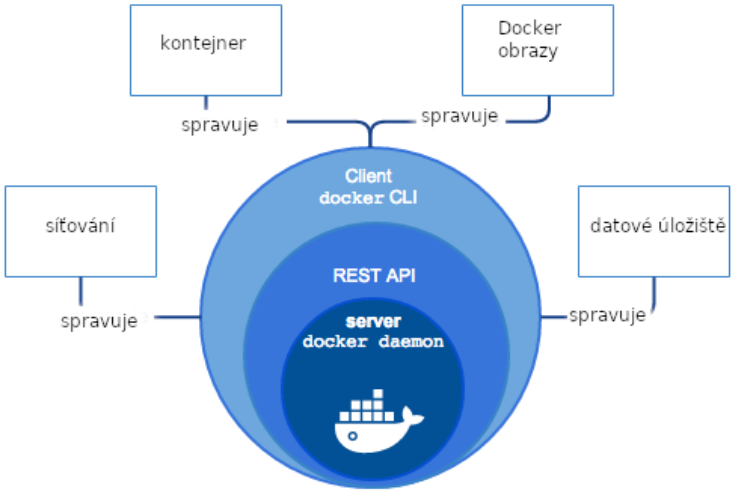
Docker využívá funkcionality linuxového kernelu pro svoji funkcionalitu. Díky tomuto funguje na počítačích, kde běží OS založený na Linuxu. V následujících sekcích budou tyto technologie blíže popsány a bude vysvětlena jejich role v systému.

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 3.1 je možné vidět, jaké jmenné prostory lze v Linuxu nalézt. [18]

Při spuštění kontejneru dojde k vytvoření procesu na hostitelském systému. Procesy dostanou od systému nějaké PID[10] a chovají se jako normální procesy. Pokud se však přihlásíme do kontejneru (command: docker exec -it <nazev kontejneru> bash) a podíváme se na procesy běžící v daném kontejneru uvidíme, že procesy mají jiná PID a určitě mají i PID=1.

[10]PID - Process identifier.



Obrázek 3.3: Docker a jeho komponenty [17]

Tabulka 3.1: Linuxové jmenné prostory

Jméno	**Popis**
Cgroup	Řídí a nastavuje limity a zdroje
IPC	Systém pro komunikaci procesů, POSIX fronty
NET	Síťové rozhraní, protokoly, porty, etc.
MNT	Připojená zařízení, tzv. mounty a volumy
PID	ID procesů a jejich izolace
User	Uživatelská ID a ID skupin
UTS	Hostname a NIS doména systému

Každý kontejner může mít svůj vlastní souborový systém a svoje síťové rozhraní. Vše, co můžeme oddělit mezi hostitelem a kontejnery, je uvedeno v tabulce 3.1.

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 dalšímy kontejnery.

Často dochází k záměně pojmů mezi kontrolními skupinami a jmennými prostory. Znovu to tedy shrňme. Kontrolní skupiny, neboli cgroups, omezují, co můžeme použít. Na druhou stranu jmenné prostory, neboli namespaces, omezují, co jsme schopni vidět v systému. [18]

3.2.3 Docker daemon

Docker daemon, neboli dockerd, poslouchá dotazy na Docker API a spravuje objekty, jako jsou Docker obrazy, kontejnery, síť a úložiště. Komunikuje ale i s dalšími daemony, aby byl schopen řídit službu. [13]

3.2.4 Docker klient

Je to primární cesta, jak komunikovat s Dockerem. Když použijeme příkazy, jako jsou "docker run", klient odešle příkazy daemonu zmíněného výše. [13]

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 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 spouštěn jako základní vrstva pro nový kontejner, nebo bude stažen na lokání počítač, či nahrán do registru. [13]

3.2.6 Obrazy

Můžeme si je představit jako šablonu, na které je spuštěn kontejner. Jeden 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.

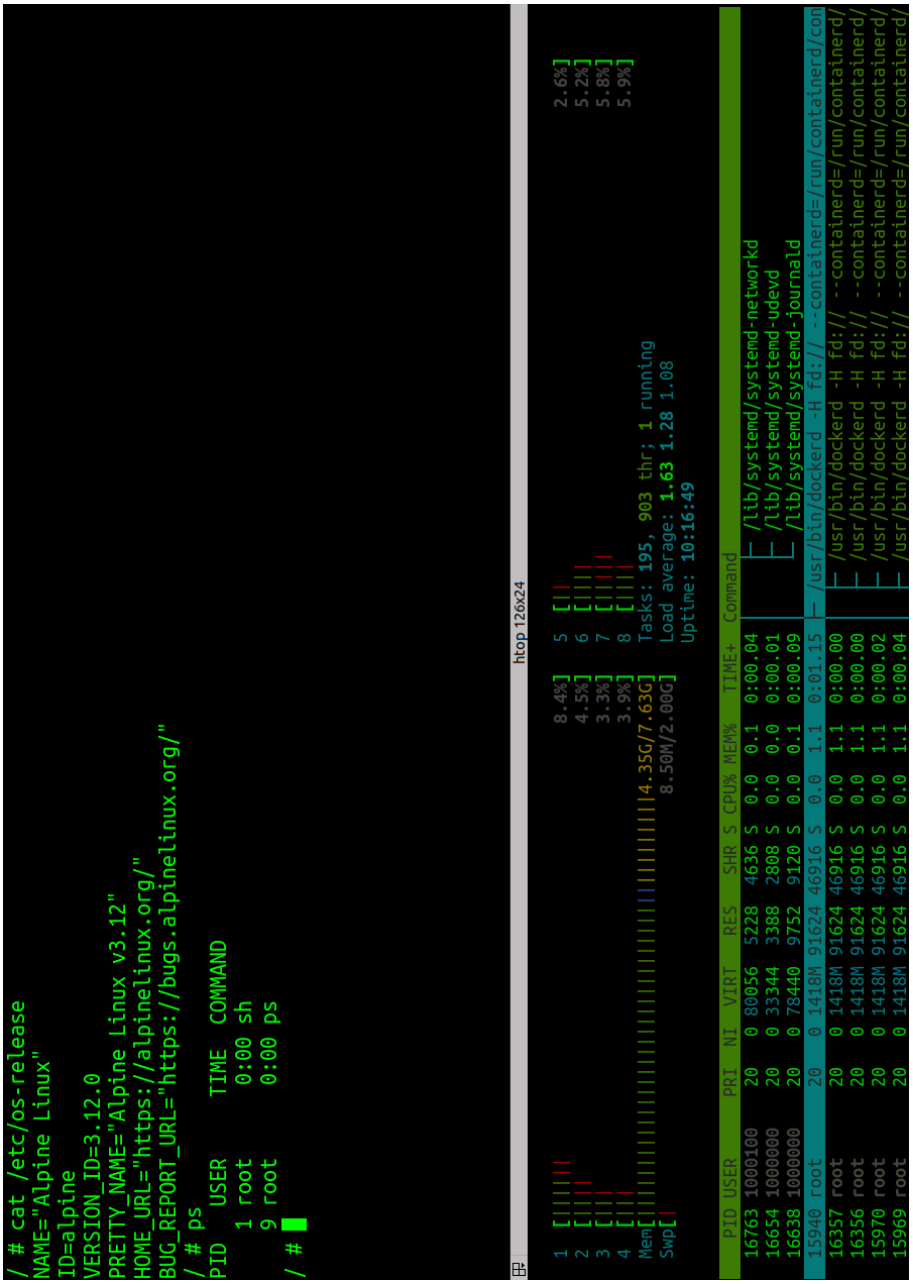
Každý příkaz v Dockerfilu vytvoří na lokálním počítači tzv. vrstvu, kterou při úpravě Dockerfilu mění, nebo předělává pouze pokud byla změněna. [13]

3.3 Systémové kontejnery

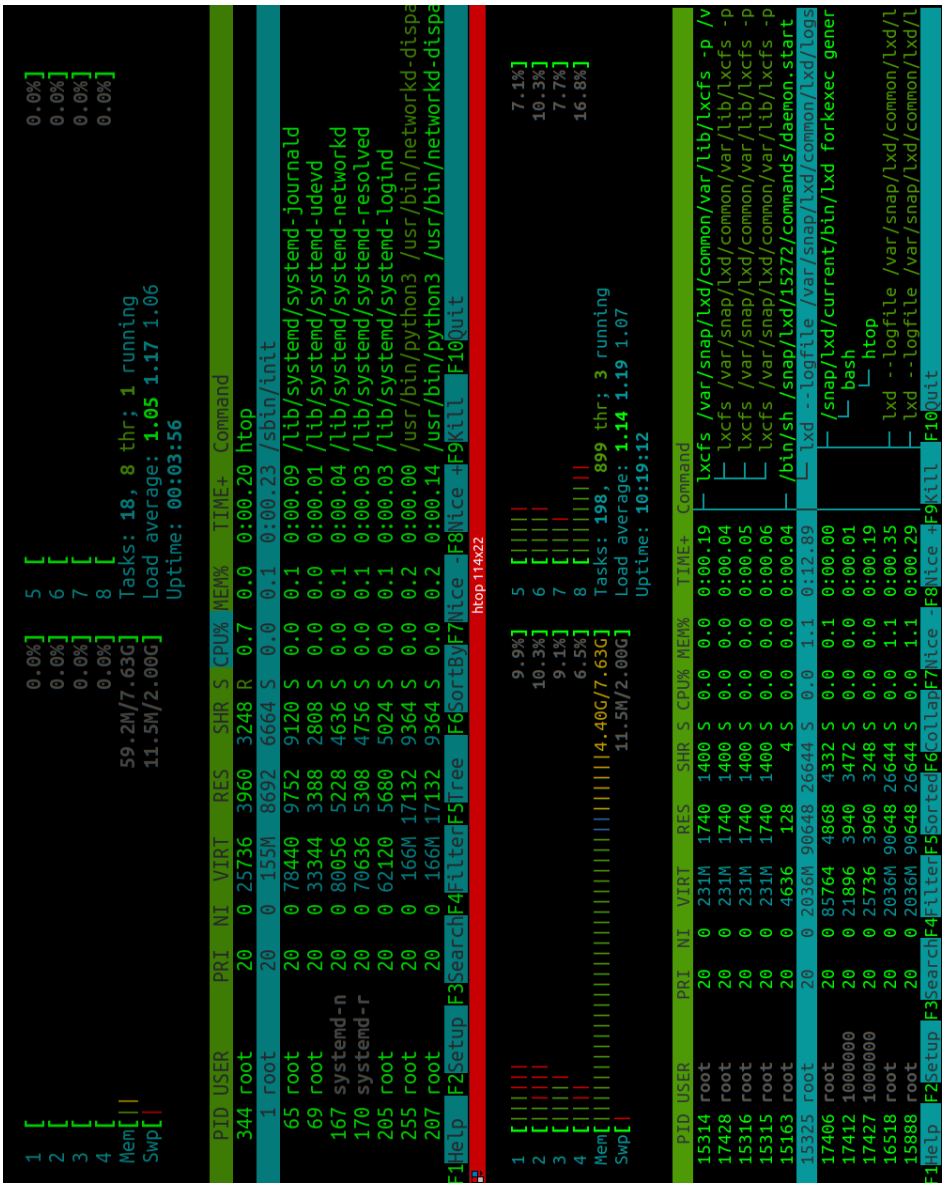
Docker kontejnery nejsou však jediné, které se v produkčním prostředí používají. Patří do tzv. aplikačních kontejnerů. Jejich účel je zpravidla spouštět pouze jeden proces. K takovýmto kontejnerům můžeme ještě přidat kontejnery Rocket. Hlavním rozdílem od Docker kontejnerů je to, že nemají na systému spuštěného daemona.

Dále tady máme systémové kontejnery. Ty jsou používané jako klasické OS a zařízení. Na jednom silném stroji může běžet několik takových kontejnerů. Ty mohou uživateli poskytovat oddělené prostředí od celého serveru a nabídnout mu izolovaný prostor od ostatních, díky zmíněným technologiím.

Kontejnery se pak mohou uživatelům předat jako virtuální počítač v cloudu. Tuto možnost zastřešuje projekt LXC, později LXD. [19]



Obrázek 3.4: Docker a jeho procesy



Obrázek 3.5: LXC a jeho procesy

Kapitola 4

Hesla

Hesla můžeme vidět všude a ne jen v informatice. Pokud se podíváme zpět do historie např. do doby velkého Caesara a jeho šifry, kde je třeba znát číslo, o které se posouvají znaky ve zprávě. Uvidíme, že šifry byly třeba už v dávných dobách a zajišťovaly výhody ve velkých bitvách. [20]

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. [21]

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

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

4.1 Windows

Windows se chovají jinak v doméně a jinak mimo ni. Pokud je počítač v doméně, je preferován autentizační protokol kerberos, nebo jiný autentizační protokol. V doméně je nastaven jeden nebo více autentizačních serverů, přes které se uživatel ověřuje. [22]

V současných Windows Server edicích je implementován Kerberos verze 5. Kerberos v základním nastavení operuje na portu 88 a k šifrování používá symetrickou šifru, tedy k šifrování a dešifrování používá jeden a ten samý klíč. [23]

Kerberos není však jediný protokol, který se používá pro autentizaci v doméně. Pro takové účely existuje ještě protokol NTLM. Mezi nimi je rozdíl

v možnosti pouze jednoho skoku na doménový kontroler v protokolu NTLM, kdežto Kerberos umožňuje širší možnosti.

4.1.1 Kerberos

1. Uživatel se přihlásí. Pošle požadavek v textovém formátu TGT[11]. Zpráva obsahuje ID uživatele, ID požadované služby, klientovu IP adresu a čas, do kdy je požadavek platný.

2. Autentizační server zkontroluje existenci uživatele. Pokud jej najde, vygeneruje náhodný klíč pro spojení mezi uživatelem a serverem. Následně pošle dvě zašifrované zprávy zpět. Jednu s klíčem serveru a jednu s klíčem klienta.

3. Klient rozšifruje zprávu a může se připojit. Informace jsou cachovány lokálně.

4. Pokud uživatel přistupuje na síťové zařízení, pošle své ID, časový otisk a zacachované TGT.

5. TGS[12] rozšifruje informace a poskytne tiket a klíč pro připojení.

6. Klient následně pošle požadavek na server se zašifrovaným tiketem, který dostal v předešlém kroku.

7. Zařízení rozšifruje požadavek a po schválení nabídne své služby. [24]

Podporované šifry v základu od Windows 7 a Windows Serveru 2008 R2 jsou:

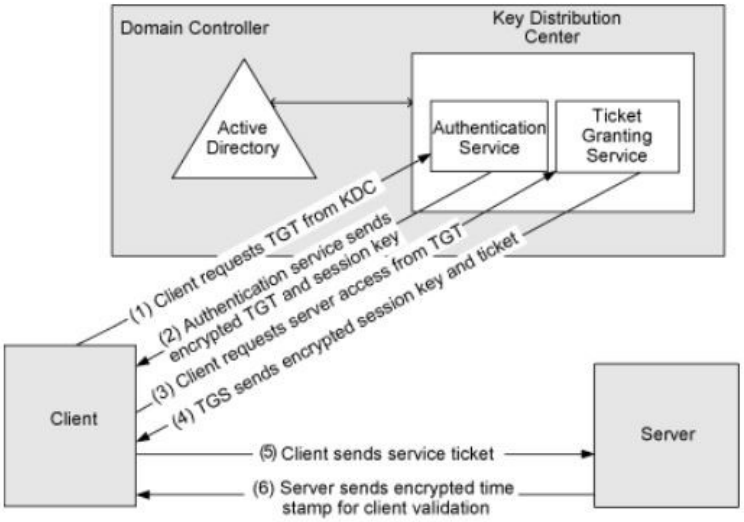
- AES256-CTS-HMAC-SHA1-96
- AES128-CTS-HMAC-SHA1-96
- RC4-HMAC

4.1.2 NTLM

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.),

[11] Ticket-granting ticket - zašifrovaný identifikační soubor.
[12] Ticket granting service - poskytuje uživatelům tikety a TGT.



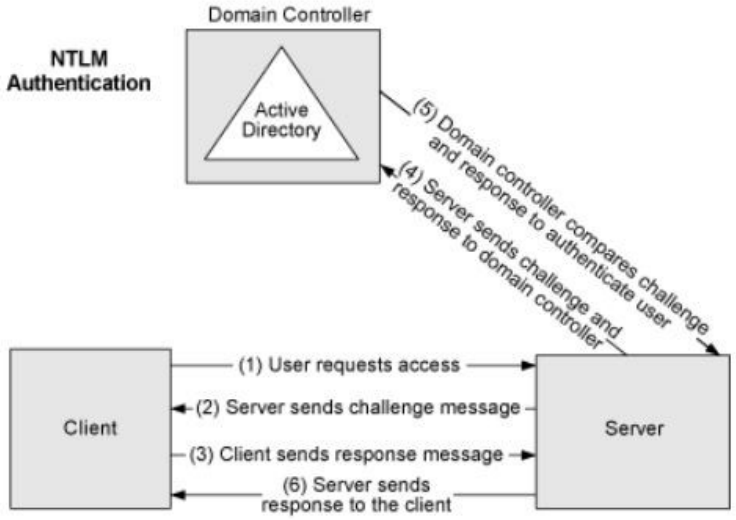
Obrázek 4.1: Kerberos protokol pro autentizaci [25]

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 vypočtení 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. [24]

4.2 Linux

Linux jakožto systém, který se skládá 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 4.1 uvidíme podporované funkce.



Obrázek 4.2: NTLM protokol pro autentizaci [26]

Tabulka 4.1: Hašovací funkce podporované v linuxových systémech

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

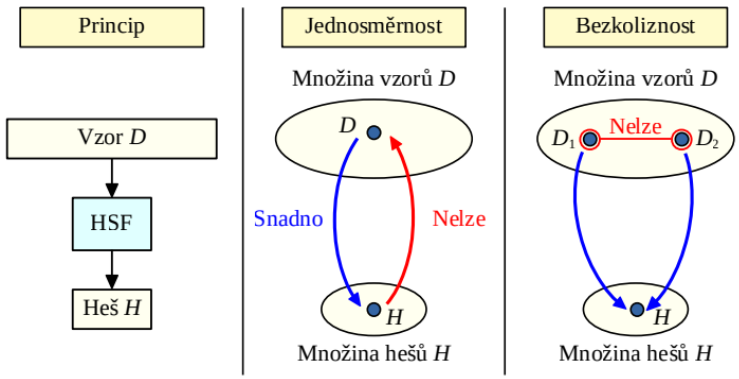
Pro změnu hašovací funkce je potřeba upravit tento soubor `/etc/pam.d/passwd`. Příklad obsahu takového souboru:

```
# V příkazové řádce vypište obsah souboru /etc/pam.d/passwd
cat /etc/pam.d/passwd

#%PAM-1.0

password required pam_unix.so sha512 shadow nullok
```

4.3 Hašovací funkce

Hašovací funkce je kryptografická funkce, která argumentu o libovolné délce přiřazuje tzv. haš H, což je číselná hodnota o pevně stanovené délce (typicky o délce 256 až 512 bitů). [27] Od hašovací funkce se vyžadují dvě vlastnosti:

- **Jednosměrnost**, což znamená, že určení hodnoty haše je pro zadaný vzor výpočetně snadné, avšak určení hodnoty vzoru ze znalosti jeho haše je prakticky nemožné.
- **Bezkoliznost**, což znamená, že je prakticky nemožné nalézt nějakou dvojici různých vzorů takovou, aby jejich haše 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 haš a vznikají tak kolize. [27]



Obrázek 4.3: Hašovací funkce a její vlastnosti [27]

Kryptografické využití hašovacích funkcí:

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

4.3.1 MD5

U MD5 tvoří kontext čtyři 32 bitová slova A, B, C a D. Na obrázku 4.3 vidíme zvětšenou jednu rundu hašování. m_i je jeden 512 bitový blok zprávy. Ten je rozdělen na 16 32bitových slov M_0, $M_1 \ldots M_{15}$, a tato posloupnost je opakována 4x za sebou (v různých permutacích).

V kompresní funkci se kontext "zašifruje"vždy jedním 32bitovým slovem M_i. Poznamenejme, že na místě dílčí funkce F se po 16 rundách střídají 4 různé (nelineární i lineární) funkce (F, G, H, I) a v každé rundě se využívá jiná konstanta K_i. Po 64 rundách dojde ještě k přičtení původního kontextu ($H_i - 1$) k výsledku podle Davies-Meyerovy konstrukce (xor je nahrazen aritmetickým součtem modulo 232). Tak vznikne nový kontext H_i.

Pokud by zpráva M měla jen jeden blok, byl by kontext (A, B, C, D) celkovým výsledkem. Pokud ne, pokračuje se stejným způsobem v hašování druhého bloku zprávy m_2 jakoby s inicializační hodnotou $H_i - 1$. Po zpracování bloku m_N máme v registrech výslednou 128bitovou haš HN. [1]

Pro hašovací funkci se již našel rychlejší způsob, jak vygenerovat kolizi a nesplňuje tak jednu z nutných podmínek, které po takových funkcích vyžadujeme.

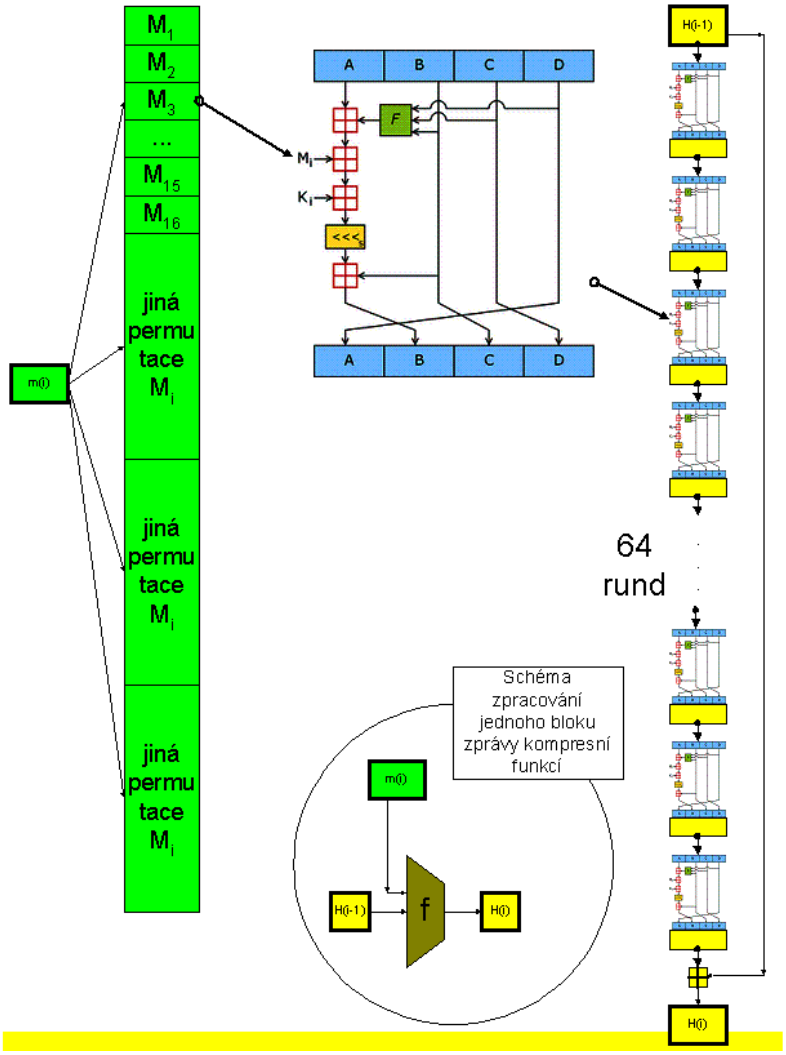
4.3.2 SHA-1

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 umožnuje zpracovat zprávu s maximální délkou $2^64 - 1$ bitů a představuje výstupní hašový kód s délkou 160bitů. Vstupní zpráva M je zpracována po blocích s velikostí 512bitů. Svým vnitřním fungováním je velice podobná MD5. Liší se zejména ve velikosti výstupního řetězce.

Blok zprávy M_i je zpracováván v $4 * 20$ rundách. 160bitový kontext (opět začínáme s inicializačním vektorem IV) je postupně "zašifrováván"32bitovým slovem m_i pro rundy 0, 1 ...15 a pro rundy 16, 18 ...79 32bitovým slovem w_i.

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.[1]

SHA-1, stejně tak jako MD5 se již nepovažuje za bezpečnou kvůli možnosti nalezení kolize v rychlejším čase. Haš byla prolomena.



Obrázek 4.4: Hašovací funkce MD5 [28]

Tabulka 4.2: SHA-X vlastnosti. [1]

	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^{13}	128	192	256

4.4 Útoky na hesla

Útoky na hesla jsou na denním pořádku na většinu z nás. Pokud však ještě nebyly, je otázkou času kdy se tak stane. Při pohledu do blízké minulosti uvidíme, že i velké firmy se s takovými útoky a problémy setkaly. Jsou to např.:

- **Yahoo**, které v roce 2013 a 2014 unikla data milionů uživatelů. V roce 2013 to byl veliký problém, protože hesla nebyla pečlivě hašována. V roce 2014 se však již poučili a hesla zahašovali 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 unikla 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 ledabylostí a to i u takovýchto světových firem. Seznam by mohl pokračovat do nekonečna, protože každým dnem útoků pouze přibývá. [29]

Dále se podíváme na jedny z nejpoužívanějších praktik získávání a lámání hesel.

- **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 číslice či speciální znak. Uživatel nad nimi poté nemusí těžce přemýšlet a vzpomínat si, jako na náhodně vygenerované řetězce. Útočník potřebuje soubor, který obsahuje známá hesla. Takový není těžké najít kdekoliv na internetu. K testovaným útokům pomocí slovníku bude v pozdější části využit soubor `rockyu.txt` se známými hesly. [30]
- **Hrubá síla** již není tak efektivní, jako bývala před lety, kdy se začala používat a nutit po uživatelích hesla s alespoň 8 znaky. Tento útok je ve své podstatě nejjednodušší. Útočník spustí program, který mu generuje náhodné sekvence znaků 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. [30]

- **Odposlech komunikace**. Útočník odposlechne komunikaci. Pokud je taková komunikace nešifrována, je jednoduché z paketu získat heslo (posílání dat přes HTTP bez SSL[14]). Samozřejmě to nemusí být pouze heslo, které se po síti přenáší. Proti takovým útokům se používá například SSL protokol, ten zajistí šifrování komunikace mezi subjekty. [30]
- **Man In the Middle**, nebo-li MITM je množina útoků, kde se útočník dostane do prostoru mezi uživatelem a např.: autentizačním serverem. Rozdíl mezi odposlechem je ten, že se rovnou napojí na linku a směruje provoz přes své zařízení. Mezi takové útoky patří např. útok DNS sniffing. [30]
- **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. [30]
- **Sociální inženýrství** je odvětví útoků, které cílí na neznalost uživatele ohledně těchto útoků, nebo jeho neopatrnost. Mezi takové patří:
    1. **phishing**, který se zaměřuje na kontaktování uživatele a donutí jej kliknout na závadný odkaz, či stáhnout např. key 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,

[14] SSL - Secure Socket Layer.

4.5 Ochrany prolomení a autentizační faktory

Jak by heslo tedy mělo vypadat, aby bylo dostatečně silné a jak se ochránit před většinou z uvedených útoků?

Uvedeme si doporučení NIST SP 800-63B-3 (2017) pro vývojáře, ze kterých bychom si měli odnést jak silné heslo má vypadat:

- minimální délka 8 znaků (generováno člověkem)
- minimální délka 6 znaků (generováno strojem)
- maximální délka alespoň 64 znaků
- podpora všech ASCII znaků
- (dictionary check) kontrola existence hesla ve známých slovnících
- (lockout) alespon 10 pokusů před uzamčením účtu(!)
- žádné požadavky na složitost
- žádná doba expirace
- (hints) žádné nápovědy
- (knowledge-based authentication) žádná znalostní autentizace ("Barva vašeho prvního auta?")
- nepoužívat SMS jako 2FA[15]

Pro zvýšení ochrany se používá tzv. 2FA. Ta přidá další úrověň ochrany v procesu authentikace o další faktor. Mezi základní faktory autentizace patří:

- co vím: heslo, pin (lze uhodnout, nebo zjistit),
- co mám: karta, token (lze ukradnout, zkopírovat),
- co jsem: biometrie (můžeme napodobit).

Jako další úroveň zabezpečení k těmto faktorům můžeme ještě přidat jeden z těchto , abychom zvýšili složitost průniku do systému:

- elektronická karta + PIN,
- heslo + SMS kód,
- heslo + e-mail,
- heslo + token[16],
- heslo + poloha.

[15] 2FA - Dvoufaktorová autentizace.

[16] Token, nebo také OTP (One Time Password).

4.6 Složitost hesel

Důležitým faktorem prolomení hesla je jeho entropie. Ta nám říká, jak nepředvídatelné toto heslo je a jak velká je neurčitost na daný znak. Vzorec pro její výpočet vypadá takto:

$$E = L * log_2(R) \tag{4.1}$$

- E: je výsledná entropie hesla,
- R: množina možných znaků,
- L: počet znaků v daném hesle,
- R^L: počet všech možných hesel z dané množiny znaků a délky hesla,
- $L * log_2(R)$: počet bitů entropie hesla.

Obecně platí, čím větší entropie, tím hůře se dané heslo dá prolomit. Tabulka 4.3 představuje množiny hesel a počet různých znaků, které se dají použít.

Tabulka 4.3: Znakové sady

Sada	**Počet znaků**	**Entropie**
Číslice	10	3.3 b
Hexadecimální znaky	16	4 b
Malá písmena	26	4.7 b
Malá písmena + číslice	36	5.2 b
Malá + velká písmena	52	5.7 b
Malá + velká písmena + číslice	62	6 b
ASCII znaky	94	6.6 b
ASCII + mezera	95	6.6 b
Binárně napsaný byte	256	8 b
Specíální znaky	13	3.7 b
Rozšířené speciální znaky	33	5 b
Diceware[17]	7776	12.9 b

[17]Diceware je metoda generování hesla, pomocí hracích kostek a slovníku (7766 slov), ve kterém jsou slova označena 5ti čísly, od 1 do 6 a heslo se vygeneruje několika hody kostkou.

Jak dlouhá hesla jsou adekvátní v současné době a kam je zařadit?

- E < 28b: velice slabé heslo.
- 28b < E < 35b: stále se nepřibližujeme k cílenému zabezpečení.
- 36b < E < 59b: zde už se pohybujeme v přijatelných heslech.
- 60b < E < 127b: velice silné heslo.
- 128b < E: přehnaně silné heslo.

4.7 Solení hesla

Sůl je přidaný bezpečnostní proces do hašování hesla, je to náhodná sekvence znaků, která se přidá před zahašováním na konec hesla. To zajistí různé výstupy pro stejná hesla a zlepší ochranu proti útoku pomocí tzv. rainbow tables[18]. Sůl je nejčastěji uložena s heslem v jedné databázi, aby systém věděl jakou sůl použít. [31]

[18]Rainbow table je předpočítaná dabáze haší s nejpoužívanějšími hesly.

Kapitola 5

Jiná řešení

Lámání hesel je známý problém a stejně tak distribuované lámání na více počítačů. Uvedu řešení, která mě zaujala a od kterých jsem se inspiroval.

Od těchto řešení se má práce liší hlavně v tom, jaké technologie jsou použity a v jakém jazyce jsou služby napsané. Žádné totiž nepoužívá Kubernetes pro správu všech služeb, ze kterých se jejich řešení skládá. Nejsou tedy škálovatelné a lehce rozšiřitelné o nové funkce. Jejich downtime pro aktualizace bude řádově větší a nespolehlivý.

Aplikace nejsou zabalené v Dockeru, takže se nedá předpokládat stále stejné prostředí a pro aplikace bude třeba doinstalovat různé verze knihoven, nebo jiného softwaru, které mohou na serverech vytvořit nekompatibilitu mezi verzemi. Stejně tak se tedy předpokládá, že na serveru poběží jen tyto aplikace a to nemusí být optimální rozdělení zdrojů mezi služby a zbytečně se budou plýtvat finance na nevyužitý HW[19].

5.1 Hashtopolis

Je multipratformní client-server nástroj pro distribuování úloh pro hashcat na více počítačů. Jejich hlavním cílem je robustnost, přenositelnost a podpora více uživatelů. Skladá se ze dvou částí a to:

- agenta napsaného v C#, nebo Pythonu,
- serveru napsaného v PHP.

Se serverem se komunikuje pomocí API, nebo pomocí GUI[20]. Pro to aby byl politicky korektní se přejmenoval v roce 2018 z Hashtopussy na Hashtopolis. Projekt může být k nalezení na `https://github.com/s3inlc/hashtopolis`.

[19] Hardware - veškeré fyzické a technické vybavení počítače.
[20] GUI - Graphical User Interface.

5.2 CrackLord

Poskytuje škálovatelný a distribuovaný systém pro lámání hesel ale i pro jiné úkoly. Více než klastr na lámání hesel je to spíše cesta jak balancovat různé úkoly. Dostává tak jak CPU tak GPU z různých systémů do jedné fronty pomocí dvou služeb:

- Resource,
- Queue.

Nezrychlí úlohy ve frontě, tím, že by je paralelizoval, ale jednoduše zařídí jejich distribuci přes všechny zdroje. Stejně tak, jako má aplikace. Používá různé nástroje psané v Go, které je možné rozšířit. Informace a repozitář k projektu na `https://github.com/jmmcatee/cracklord`.

5.3 Distributed Hash Cracker

Distrubovaný lámač hesel napsaný v Go. Inspirovaný kurzem Carnegie Mellon University. Jeho autor se prioritně snažil rozšířit si znalosti v jazyce Go.

Aplikace je rozdělena do třech částí:

- Request Client, což je část, která předává požadavky centrálnímu serveru,
- Central Server zpracovává úlohy, výpočetní jednotky a distribuci práce,
- Worker Client počítá a spouští úlohy jež dostal od Central Serveru.

5.4 Gocrack

Vychází ze stejného kurzu jako Distributed Hash Cracker, který jej uvádí ve svých zdrojích. Je též napsaný v Go a skládá se ze stejných částí.

Jako ostnatní aplikace a řešení problému distribuovaného lámání nepracuje s Kubernetes a dalšími nástroji pro orchestraci. Neřeší zapouzdření aplikace, ani nastavení zařízení, na kterých poběží. Práce je k nalezení na `https://github.com/henrykhadass/gocrack`

Kapitola 6

Vytvoření klastru na lámání hesel

Klastr označujeme jako spojení více počítačů za jedním účelem. Tyto počítače se následně mohou tvářit jako jeden samotný počítač. Výhoda takového řešení je, že centralizujeme výpočetní výkon přes více zařízení a násobíme tak naše zdroje, zlevňujeme náklady za více slabších počítačů oproti jednomu silnému. [32] Klastry mohou sloužit pro více účelů:

- Výpočetní klastr
- Klastr s vysokou dostupností
- Rozložení zátěže
- Úložný klastr
- Gridový klastr

Kubernetes klastr je množina výpočetních nodů, které spouští kontejnerizované aplikace. [33]

6.1 Realizovaný klastr

Realizovaný klastr se skládá z virtuálních počítačů hostovaných z `https://one.fit.cvut.cz/` (následně kvůli nedostatku zdrojů byl přesunut na DigitalOcean 6.2). Počítačů v klastru je připojeno konktrétně šest. Jeden z nich je nastaven jako master a poskytuje sdílené úložiště pro data z databáze pro workery pomocí NFS[21]. Master též poskytuje Grafanu pro monitorování všech zařízeních v klastru.

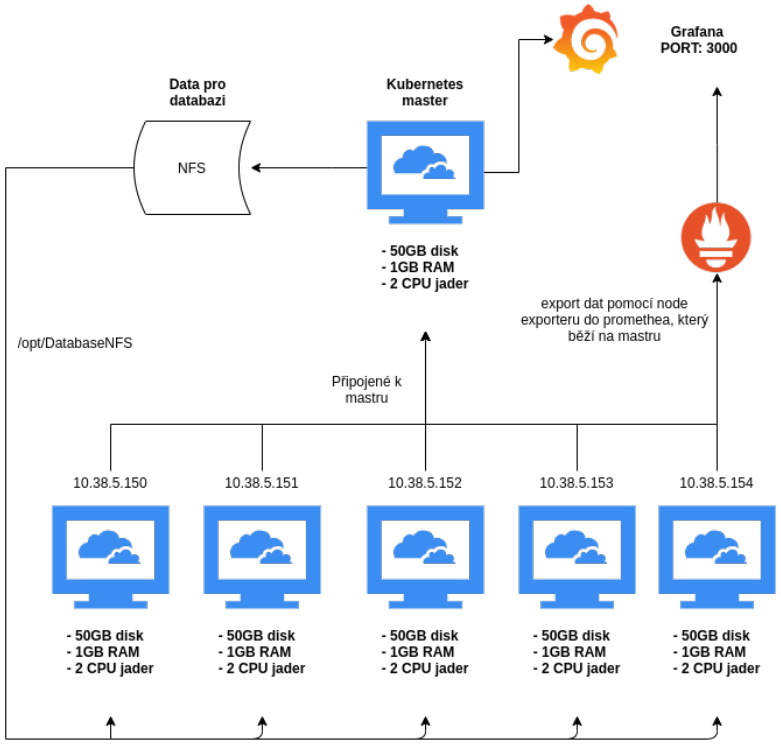
Operační systém běžící na všech počítačích je Ubuntu 20.04. Každý má k dispozici (zdroje na DigitalOcean):

[21] NFS - Network Filesystem, slouží ke sdílení dat po síti.

- 50GB místa na disku (160GB),
- 1GB velikost paměti RAM[22] (8GB),
- 2 CPU[23] jádra (4 jádra).

Na každém je nainstalován **node-exporter**, který zveřejňuje metriky zařízeních a pomáhá monitorovat klastr. Tyto metriky jsou zpracovány aplikací **Prometheus** na mastru a ten tyto data poskytuje **Grafaně**, která slouží k jejich přehlednější vizualizaci.

Všechna zařízení mají připojené sdílené úložistě s daty databáze. Data jsou sdílena z mastra pomocí NFS, kontejnery s databází tak mají konzistetní data kdykoliv se vypnou, nebo restartují na jakémkoliv nodu.



Obrázek 6.1: Realizovaný klastr.

[22]RAM - Random Access Memory.
[23]CPU - Central Processing Unit

6.2 Nastavení a nasazení klastru

Veškeré instalace a nastavení provádí Ansible. Na počítači, ze kterého klastr chceme nastavovat, potřebujeme přístup na všechna zařízení a tato zařízení musejí být též propojené a síťově přístupné jedno pro druhé.

Důležité úlohy jež Ansible provádí:

- nastaví a nainstaluje sdílené úložiště,
- zakáže swap (zpomaluje a zhoršuje optimalizaci Kubernetes),
- vytvoří uživatele s právy pro použití příkazu kubectl,
- nainstaluje závislosti pro Kubernetes,
- nainstaluje a spustí aplikace pro monitorování,
- na mastra stáhne repozitář s deploymentem,
- připojí zařízení k mastru.

Na komponenty se kopírují i specifické konfigurační soubory pro aplikace, které je potřebují. Jsou k nalezení ve složce config. K těmto patří například soubory pro systemd a zaručení spuštění služeb. Konfigurace pro monitorování klastru pro aplikaci Prometheus a další.

Adresářová struktura pro nasazování pomocí Ansible vypadá typicky následovně:

```
deploy.yml ..... soubor s úlohami a cíly spuštění
config ..... adresář který obsahuje nastavení některých služeb
    grafana.ini
    prometheus.service
    prometheus.yml
    exports .1 hosts ..... specifikace nastavovaných zařízení
README.adoc ..... stručný popis adresáře, nástroje a použití
tasks ..... adresář s úkoly, jež se mají vykonat
```

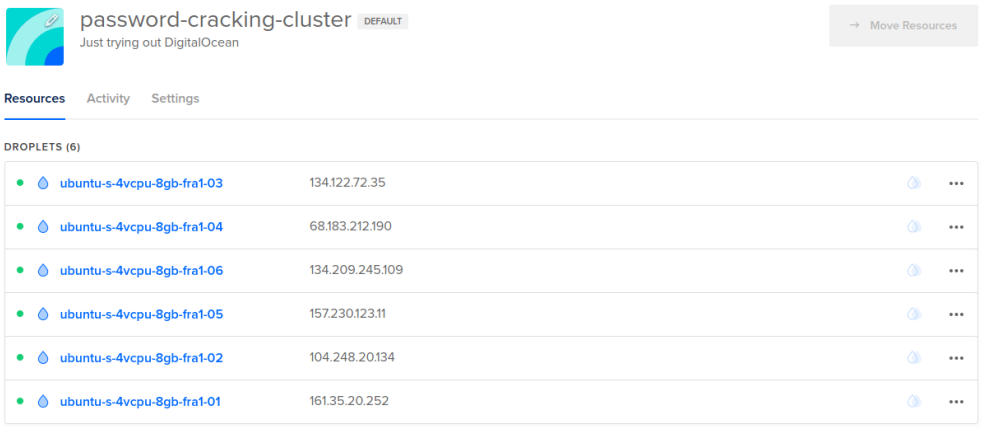
Aplikace se spouští ručně po příhlášení na master a to z důvodu větší kontroly nad jejími komponenty a spouštění. Zejména pro to, že Docker image jsou dostupné ze služby třetí strany (Gitlab). V předchozích krocích se deploymenty pro Kubernetes stáhly společně s repozitářem a je možné aplikaci spustit postupným deploymentem: **kubectl apply -f deployment-<služba>**. Při prvním spuštění je třeba vytvořit databázi pomocí kontejneru na kterém běží. Skript na vytvoření je v přiložených zdrojových kódech, nebo v repozitáři, který se stáhne na mastera.

Následně pomocí příkazu uvidíme informace o naší aplikace a všech komponentách: **kubectl get all**.

6.3 Digital Ocean

Z důvodu nedostatku zdrojů a podcenění alokace virtuálních zařízení bylo nutno zrealizovat silnější klastr. Tzn. klastr který se skládá ze silnějších zařízení. Využil jsem akce jež DigitalOcean[24] poskytuje a při registraci jsem obdržel kredit na spuštění několika virtuálních počítačů, které nazývají Droplet. Tímto bylo možné otestovat vytvořené nasazení i na cloudové řešení, které je používáno v reálném světě mnohými společnostmi. Zakládají si na jednoduchosti a rychlosti.

DigitalOcean poskytuje i tzv. managed kubernetes klastry. K takovému klastru není přístup na jednotlivé zařízení v něm připojené. Poskytovatel zveřejní jen configurační soubor pro komunikaci s klastrem pomocí Kubernetes API.



Obrázek 6.2: Digital Ocean.

[24]DigitalOcean - cloudový poskytovatel služeb

Kapitola 7

Návrh aplikace na lámání hesel

Aplikace se skládá ze čtyř mikroslužeb. Z publicservice, která zprostředkovává přístup uživateli do databáze hesel a dovoluje mu přidávat haše k řešení. Privateservice, jež slouží ke komunikaci mezi službami Computationunit a databází 7.1.

Mikroslužby mezi sebou komunikují pomocí síťových protokolů. Komunikace pomocí HTTP a API se lehce implementuje a poskytuje potřebné vlastnosti pro realizaci služeb a jejich kominukace. K implementaci API jsem použil knihovnu FastApi, která poskytuje pokročilé funkce bez složité implementace. Též poskytuje webové rozhraní pro admina, nebo jiného správce s popisy implementovaných funkcí.

Aplikace je rozdělena do čtyř služeb. Publicservice pro komunikaci uživatele s aplikací, Privateservice, sloužící pro komunikaci mezi Computationunit a databází.

7.1 Databáze

Služba Databaservice, je služba zpřístupňující databázi, která v ní běží a slouží pouze pro ukládání dat a dovoluje k nim přistupovat mikroslužbám Publicservice a Privateservice.

Jako databázi jsem vybral PostgreSQL. Je pomalejší než MySQL, ale za to podporuje mnoho složitějších funkcí, jako jsou např. složitější datové typy. Důležité je, že pokud se bude v budoucnu aplikace nějakým způsobem rozšiřovat, nebude nutné převádět data z jedné databáze do druhé. Aplikace složená z mikroslužeb by na tyto možnosti měla myslet a být na ně náležitě připravena.

Databáze obsahuje jednu tabulku jménem hashes. Sloupce použité v databázi:

- primkey INT PRIMARY KEY (vygeneruje na základě posledního PKI[25]),

[25] Primary Key

- hash TEXT (získá od uživatele),
- type TEXT (získá od uživatele),
- solving TEXT (indikátor probíhající práce),
- result TEXT (výsledek po prolomení),
- tested INT (velikost hesla pro které má být hash testován),
- dictionary TEXT (indikátor zda bylo heslo testováno slovníkem).

7.2 Publicservice

Slouží ke komunikaci uživatele s aplikací. Uživatel může prostřednictvím této služby nahrát do databáze soubor haší, tento soubor musí být textový. Haše mohou být typu MD5 nebo SHA-1 a SHA-512 (ze souboru **/etc/shadow**). To, jakého jsou typu, musí uživatel uvést před jejich nahráním. Služba poté vrátí, zda se vložení do databáze zdařilo. Tato služba může též uživateli poskytnout informace o tom, jaké haše jsou již prolomené, nebo na kterých se právě pracuje. Dotazy provádí prostřednictvím API, které služba poskytuje. Více ohledně API na obrázku 7.1.

API rozhraní, které Publicservice poskytuje:

- **/files** - z formuláře nahraje data do DB[26],
- **/solved** - v JSON[27] formátu vrátí již prolomené haše,
- **/solved/result/{result}** - vrátí haš k danému heslu,
- **/solved/hash/{primkey}** - vrátí result k daném haši,
- **/all** - vrátí veškerá data z DB,
- **/ready** - slouží pro Kubernetes a readiness check,
- **/health** - slouží pro Kubernetes a sledování, jestli služba odpovídá.

7.3 Privateservice

Tato služba slouží pro komunikaci mezi službami Computationunit a databází. Computation unit se ptá, zda existuje v databázi práce, která by mu mohla být přidělena. Privateservice mu odpoví. V případě pozitivní opovědi sestávající se z volných haší, v opačném vrátí "Fail". Computationunit na obdržené haše odešle Privateservice kontrolu zda může na těchto haších začít pracovat.

[26] Databáze
[27] JSON - JavaScript Object Notation.

Pokud byla jiná instance rychlejší a stihla začít na haši již pracovat, Private service zakáže práci. Poté co Computationunit haš vyřeší, odešle službě požadavek na aktualizaci databáze s výsledkem. Jinak předá službě nezdar, služba aktualizuje záznam v databázi a přidá velikost hesla pro další lámání. API Privateservice je uvedeno v obrázku 7.1.

API rozhraní, které poskytuje Privateservice:

- **/delete-work/{primkey}** - odstraní provádění práce na haši v DB,
- **/not-found/primkey/{passwdlen}** - volá Computationunit pokud nenajde haš v rozsahu, ve kterém hledá,
- **/not-found-dictionary/primkey/{passwdlen}** - volá Computationunit v módu slovíku, pokud nenajde haš,
- **/get-work** - Computationunit se dotazuje po práci v módu masky,
- **/get-work-dictionary** - Computationunit se dotazuje po práci v módu slovníku,
- **/start-work/{primkey}** - Computation signalizuje službě začátek lámání dané haše, pokud již byla přiřazena před tím, služba vrátí "Fail" a Computationunit na haši nepracuje,
- **/work-done/{primkey}/{result}** - heslo bylo nalezeno a odesláno službě
- **/ready** - slouží pro Kubernetes a readiness check,
- **/health** - slouží pro Kubernetes a sledování, jestli služba odpovídá na jednoduché dotazy.

7.4 Computationunit

Tato služba slouží jako obal nad aplikací HashCat, která se stará o samotné lámání. Tedy služba se stará o dotazy pro práci služby Privateservice. Následně potvrdí začátek práce na dané haši a předá práci programu. Program po prolomení svůj výsledek zapíše do souboru, ze kterého služba výsledek přečte, a pošle požadavek na aktualizaci řešení k dané haši Privateservice. Pokud nebyla haš prolomena použije jinou funkci API Privateservice.

Computationunit je v současné době schopen lámat MD5, SHA-1 a SHA-512 (ve formátu ze souboru **shadow**) haše. Může lámat v několika módech, a to slovníkem, maskou a hrubou silou. Maska může být optimalizována dle potřeby při vytváření Docker obrazů.

Na klastru je Computationunit vždy spuštěn s několika instancemi, které lámou pomocí slovníku a několika instancemi, které lámou pomocí masky.

Na každou metodu je vytvořen obraz, který se automaticky spustí s danou metodou. Liší se v otagování[28] v docker repozitory.

7.5 Hashcat

Hashcat je typ nástroje určen na lámání hesel. Je multiplatformní, tzn. nejen že běží jak na Linuxu, nebo Windows, tak podporuje veškerá zařízení, která podporují OpneCL runtime (GPU[29], CPU, FPGA ...). Dokáže lámat více než 200 různých typů haší. Podporuje útoky maskou, bruteforce, combination (spojení a kombinace kandidátů) a kombinaci útoku s maskou a se slovníkem. [34]

HashCat se stále vylepšuje a podporuje lépe GPU než jeho konkurence John the Ripper. Proto jsem zvolil tento program pro lámání. V případě změny hardwaru na GPU bude jeho integrace daleko snazší. Též vývoj generátoru slov se stále zlepšuje a zrychluje.

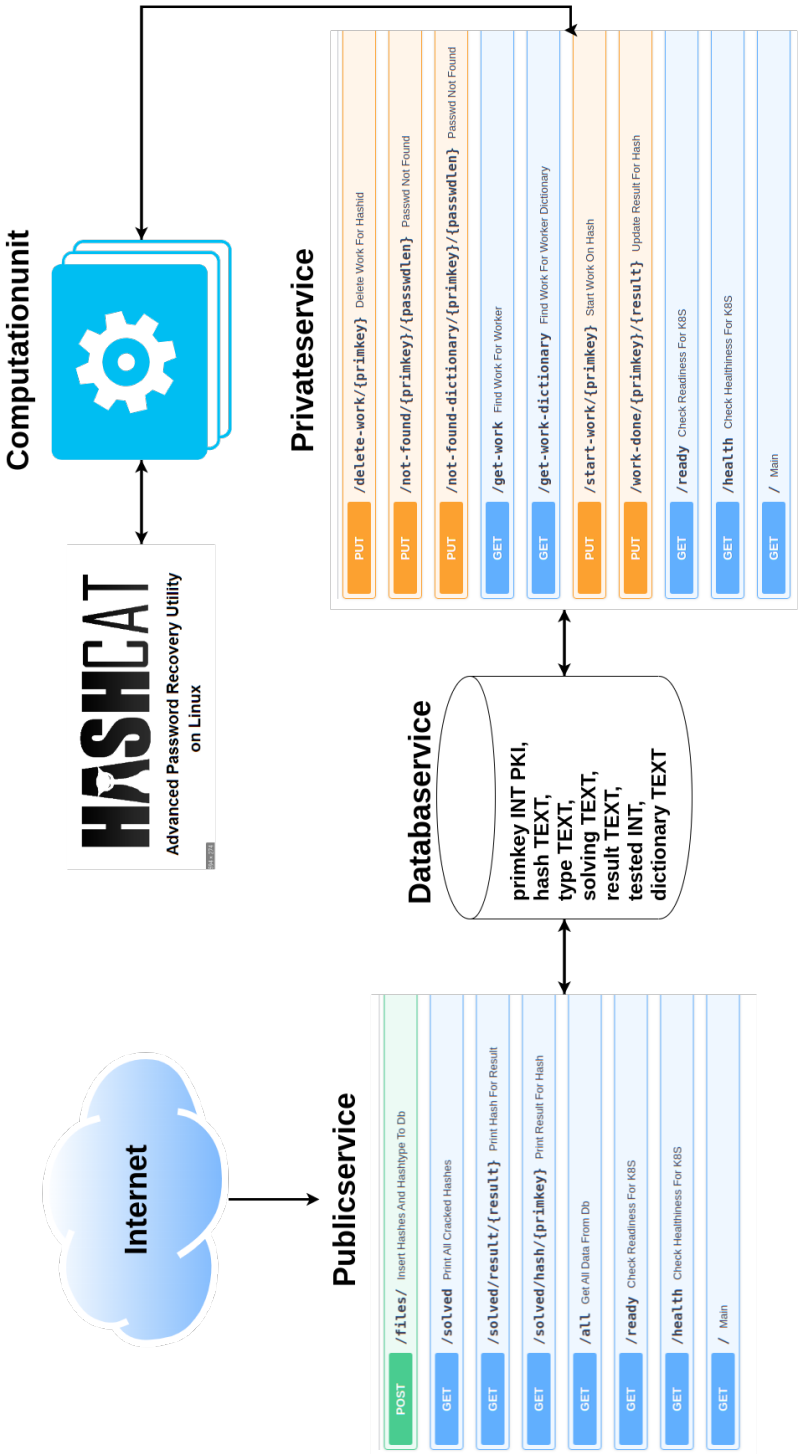
HashCat je spouštěm pomocí Computationunit. Jsou mu předány parametry, se kterými se má spustit (typ haše, výstupní soubor, soubor s haši a délka hesla). Délka hesla pro jedno lámání je vždy nastavena od Computationunit. Když se podíváme v klastru na logy jednotlivých podů, HashCat má přehledný interface, ve kterém je vidět v jakém stavu se lámání nachází.

7.5.1 OpenCL runtime

S OpenCL jsem však narazil na problém, který vidím jako velice dobré zmínit. Na nových procesorech (generace 8+) byl problém s tím, že Intel měl chybu v ovladači a knihovnách pro OpenCL a HashCat s tímto nedokázal pracovat. Program jde spustit i s touto chybou, avšak výsledky nebyly správné a často program přestal pracovat, nebo spadl bez výsledků. Tato vina není na straně SW, ale na straně výrobců čipů. Vlákno na StackExchange, které toto rozebírá `https://security.stackexchange.com/questions/179516/hardware-compatibility-with-hashcat-in-windows-10-64-bit`. HashCat sám při použití ovladače pro Intel vypíše varování o nespolehlivosti OpneCL.

[28] Otagování obrazu znamená, označit jej verzí, nebo speciálním názvem pro rozlišení od druhých.

[29] GPU - Graphics Processing Unit.



Obrázek 7.1: Architektura aplikace

Kapitola 8

Analýza hesel a výkonu klastru

Tato kapitola se zabývá rychlostí lámání hesel dle jejich délky, haše a výpočetních uzlů. I když klastr podporuje i lámání SHA-512 haše, budeme v některých testech počítat pouze s haší MD5 a SHA-1 kvůli složitosti SHA-512.

Jelikož klastr paralelizuje na úrovni velikosti hesel a ne na optimalizaci měnění abeced a rozložení jedné haše na více uzlů, bude se zkoumat množina hesel. Klastr tak slouží více k distribuci a uchování práce mezi instancemi Computationunit podu. Podobně jako Cracklord uveden v kapitole 5.

8.1 Prolomení jednoho hesla

Heslo budeme pro první test lámat pouze jedno, bude 6ti znakové a složené z malých písmen. Bude to heslo **hodiny**. Uživatelé mají tendenci k tomu, si za heslo volit to, co okolo sebe vidí. Tím se entropie hesla sníží, jelikož následující znak, nemusí být vždy náhodný, ale může být předvídatelný. Můžeme jej předpovědět podle použité abecedy nebo použitého jazyka. Pokud, by bylo použito, 6ti znakové heslo, vytvořené náhodným nebo pseudonáhodným generátorem, který by generoval pouze z malých písmen, bude horní hranice entropie:

$$E = L * log_2(R) = 6 * log_2(26) = 28.2b \tag{8.1}$$

K tomuto heslo zvolím další hesla, která se nemusejí nacházet ve slovníku a následně pro ně upravím masku, abych vysvětlil důležitost si o heslu zjistit základní informace. Tím je myšleno jeho velikost a znaková sada, která byla použita k jeho vytvoření. K útoku maskou budu používat pouze malá písmena pro první dvě hesla a číslice pro poslední, aby byla demonstrována rychlost po optimalizaci na konkrétní hesla. K útoku hrubou silou budu používat a-Z, 0-9 a speciální znaky. Nula v tabulce indikuje nenalezení hesla, heslo nebude nalezeno jen pokud bude použit útok slovníkem a heslo se v něm nebude nacházet.

Je nutné podotknout, že vteřinové rozdíly mohou být způsobeny též HashCatem, který si inicializuje a alokuje paměť a kontroluje zařízení před spuštěním. Testy na jednotlivá hesla byla testována na master nodu a na původním klastru na 6.1. Zdroje byly limitovány a lámání bylo zpomaleno přepínáním kontextu mezi běžícími procesy a jinými elementy. V tabulce 8.1 je ukázka prolomení hesla v závislosti na jeho velikosti, zvolené sadě, typ útoku a zahašování pomocí MD5. V další tabulce 8.2 jsou uvedeny rychlosti pro zahašování pomocí SHA-1. Detailnější test lámání dle velikosti v tabulce 8.3.

Tabulka 8.1: Hesla hašovaná MD5 haší.

heslo	**maskou**	**slovníkem**	**hrubou silou**
hodiny	23.064s	12.213s	167.542s
asbzxc	29.811s	0	143.341s
123482	20.344s	0	110.070s
voda	17.850s	5.747s	7.837s

V příkladu, kdy se láme množina hesel kubernetes přiřadí pody s Computationunit instancí do volných zařízení a veškerý výkon připadne na lámání. Uvidíme tedy zlepšení oproti lámání i pouze jednoho hesla, pouze však řádech sekund.

Tabulka 8.2: Hesla hašovaná SHA-1 haší.

heslo	**maskou**	**slovníkem**	**hrubou silou**
hodiny	27.479s	10.331s	227.259s
asbzxc	20.170s	0	193.418s
123482	22.653s	0	210.491s
voda	14.998s	5.223s	13.792s

8.2 Prolomení množiny hesel podle instancí

Na klastru vytvoříme několik instancí Computationunit, pokaždé několik v módu slovníkového útoku a několik v módu útoku maskou. Maska je definována na malá písmena a číslice. Klastr zkusí hesla prolomit pomocí slovníku jako první krok. Hashcat je totiž schopen optimalizovat lámání více haší najednou. Prolomení všech zadaných pomocí slovníku nám může ušetřit mnoho času a je to první test, který na heslo použijeme, protože je časově relativně málo náročný v porovnání s lámáním alespoň 7mi místného hesla. Pokud nebudou prolomena slovníkovým útokem, budou dále lámána pomocí masky.

Množina hesel se skládá ze 4 hesel ze slovníku a 4 hesel, jež se v něm nenacházejí. Dále budeme zkoumat, jak se klastr zrychlí, pokud bude více computationunit instancí s maskou. Hesla, jež budu lámat:

- tamps01,
- heaven,
- alexandra,
- 55555,
- dauiz,
- bqru,
- asdvcx,
- ab.

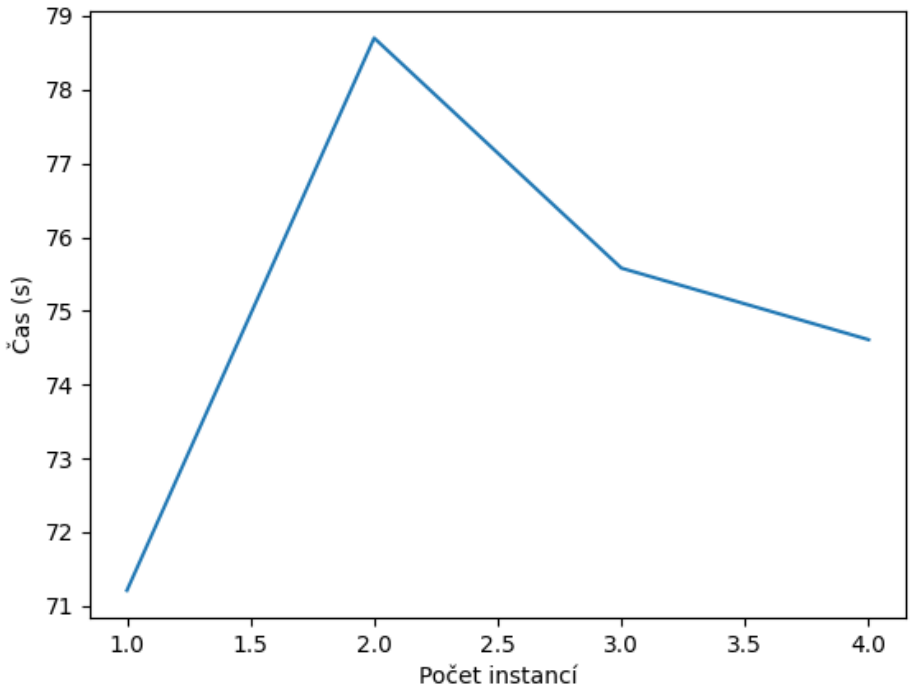
Hesla budou zahašována jak SHA-1, tak MD5 haší. A budou najednou nahrána na klastr. Klastr využije větší počet compoutationunt instancí lámajících pomocí slovníku v případě, kdy uživatelé rychle přidávají nová hesla, zrychlení se projeví tedy až při nasazení a větším počtu uživatelů. Jelikož přidáme haše jednorázově budeme měřit rychlost pro přidanou množinu. Klastr je též optimalizován na rozdělování práce a ne pro paralelizované lámání jedné haše s generováním různých abeced.

Pro práci s Kubernetes je dobré mít na master nodu více než 1GB paměti, protože Kubernetes se starají o hodně věcí a spotřebují postřehnutelné množství zdrojů 1. Doporučuji pro mastera vybrat tedy silnější z dostupných zařízení pokud se bude veškeré řízení Kubernetes centralizovat.

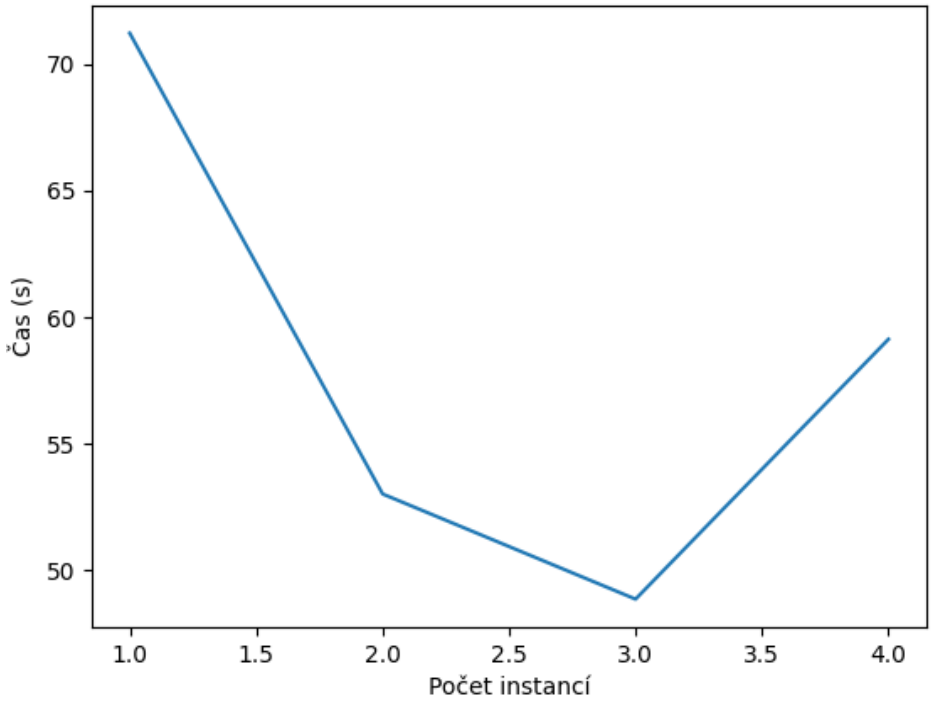
Originální klastr při spuštění více computationunit instancí přestával dobře pracovat a to kvůli nedostatku paměti RAM. Další testy jsou provedeny na klastru poskytnutého cloudovou službou DigitalOcean, zde jsem vytvořil klastr s 6 virtuálními počítači. Rozdíl mezi jejich a mým řešením je hlavně ten, že jejich virtuální počítače, které jsem přidal do klastru mají 8GB paměti, více o klastru 6.

Tabulka 8.3: Lámaná hesla zahašována sha1 a md5 haší v závislosti na počtu instancí Computationunit v daném módu lámání.

slovníkem & maskou	**čas**
1 & 1	1m 11.21s
1 & 2	0m 53.02s
1 & 3	0m 48.87s
1 & 4	0m 59.13s
2 & 1	1m 18.70s
3 & 1	1m 15.58s
4 & 1	1m 14.61s
2 & 2	0m 48.11s
3 & 3	0m 49.11s
4 & 4	0m 55.78s



Obrázek 8.1: Graf zvyšování rychlosti v závislosti na instancích slovníkového útoku.



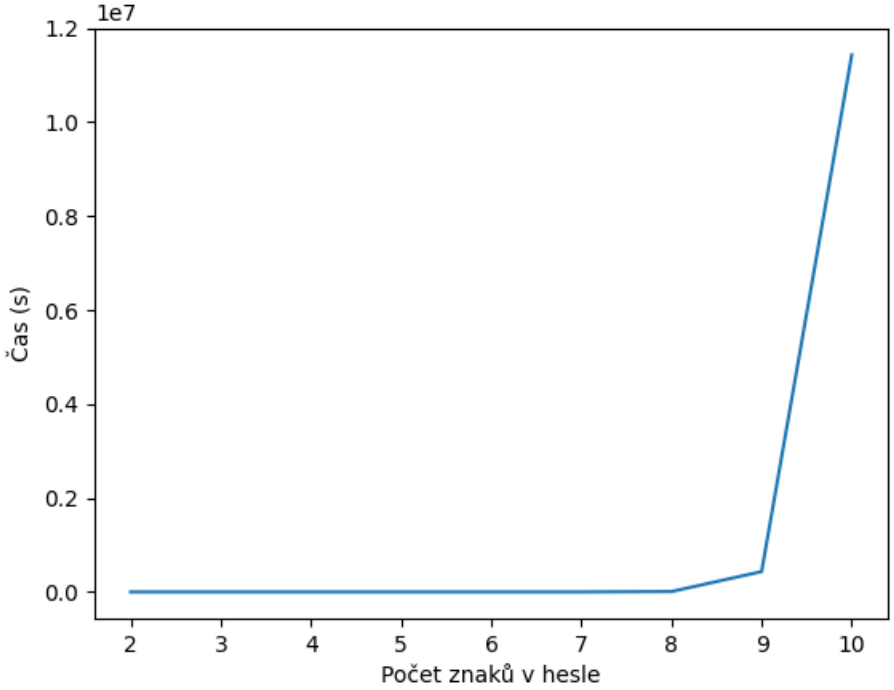
Obrázek 8.2: Graf zvyšování rychlosti v závislosti na instancích útoku maskou.

8.3 Prolomení hesla podle velikosti

Klastr není optimalizován na lámání jednoho hesla s generováním abacedy a znásledné paralelizace. Rozděluje práci pro Computationunit instance a čeká na jeho prolomení. Lámání hesla podle velikosti bude tak spíše ukázka zvyšující se složitosti při přidání jednoho znaku. Maska s jež budu hesla lámat bude stejná jako v předešlém případě. V tabulce 8.4 jsou vidět výsledky.

Tabulka 8.4: Lámaná hesla zahašována sha1 a md5 haší

znaku	**čas MD5**
2	0m5.429s
3	0m7.830s
4	0m11.626s
5	0m13.241s
6	1m 15.58s
7	4m2.939s
8	2 hours, 18 mins
9	5 days, 0 hours
10	132 days, 7 hours



Obrázek 8.3: Graf zvyšování se složitosti lámání v závislosti na velikosti hesla.

Kapitola 9

Závěr

Úkolem této práce, bylo seznámit se s novými technologiemi a pomocí nich vytvořit klastr na lámání hesel. Použité technologie byly rozebrány a byly vysvětleny všechny potřebné pojmy tak, aby čtenář byl schopen pochopit, jak tyto technologie fungují a z čeho se skládají. Klastr byl nasazen a spuštěn se základními útoky (slovníkovým, maskou a hrubou silou).

V práci je rozebráno, jak jsou hesla uložená na operačních systémech Linux a Windows a jak probíhá jejich ověření. Je vysvětlena složitost a solení hesla. Práce rozebírá nejpoužívanější útoky a jak se proti nim bránit. Útoky jsou časté a bezpečnost je v současné době velký problém a trpí jimi i velké firmy.

Klastr je popsán a je znázorněno, jak byl sestaven a spravován. Při jeho vytvoření se narazilo na problém nedostatku zdrojů a bylo za potřebí použít cloudové služby, pro poskytnutí silnějších zařízeních. Výhodou tohoto problému bylo otestování nasazení do různých prostředí. Služby, které jsou použity, jsou zabaleny v Docker obrazech a klastr je možné automaticky nasadit, na jakékoliv zařízení, běžící na OS Ubuntu 18.04 a novější.

Výsledky lámání hesel jsou analyzovány a okomentovány. Klastr neposkytuje rapidní zrychlení lámání jednoho haše, ale soustředí se na rozdělení a škálování práce. Paralelizace na lámání hesel je konstruována na velikost hesla kvůli optimalizaci programu HashCat použitého pro výpočet haší. Haš se po přidání do aplikace pomocí Publicservice přístupné uživateli nahraje do databáze. První útok na ní provede Computationunit v módu slovníkového útoku. Pokud haš nebude prolomena, bude následně lámána pomocí masky, a postupně se bude zvyšovat velikost masky, pomocí které se ji budeme snažit prolomit.

Na přiloženém médiu jsou zdrojové kódy, potřebné pro lokální replikaci a otestování. Doporučené je spouštět řešení v cloudovém prostředí. Kód je též dostupný v repozitáři na serveru gitlab, který byl využit i jako docker repozitory.

Literatura

[1] Paar, C.; Pelzl, J.; Preneel, B.: *Understanding cryptography: a textbook for students and practitioners*. Springer, 2010.

[2] The Kubernetes Authors: What is Kubernetes? https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, March 2020, [Online; accessed April 15, 2020].

[3] Tišňovský, P.: Mikrosluzby: moderni aplikace vyuzivajici znamych konceptu - Root.cz. https://www.root.cz/clanky/mikrosluzby-moderni-aplikace-vyuzivajici-znamych-konceptu/, May 2019, [Online; accessed April 15, 2020].

[4] The Kubernetes Authors: Components of Kubernetes. https://d33wubrfki0l68.cloudfront.net/7016517375d10c702489167e704dcb99e570df85/7bb53/images/docs/components-of-kubernetes.png, April 2020, [Online; accessed April 15, 2020].

[5] Aykan, F.: How to run a container on Kubernetes. https://medium.com/@aykanferhat/running-our-first-container-on-kubernetes-80041d659633, December 2018, [Online; accessed April 15, 2020].

[6] The Kubernetes Authors: Service Userspace Overview. April 2020, [Online; accessed April 15, 2020]. Dostupné z: https://d33wubrfki0l68.cloudfront.net/e351b830334b8622a700a8da6568cb081c464a9b/13020/images/docs/services-userspace-overview.svg

[7] The Kubernetes Authors: Service Iptables Overview. https://d33wubrfki0l68.cloudfront.net/27b2978647a8d7bdc2a96b213f0c0d3242ef9ce0/e8c9b/images/docs/

services-iptables-overview.svg, April 2020, [Online; accessed April 15, 2020].

[8] Red Hat, Inc.: Ansible Documentation. https://docs.ansible.com/ansible/latest/index.html, April 2020, [Online; accessed April 15, 2020].

[9] Emeth, E.; Snyder, G.; Hein, T.; aj.: *Unix And Linux System Administration Handbook*. Addison Wesley, 2017, ISBN 9780134277554, 915-948 s.

[10] Red Hat, Inc.: Ansible Concepts. https://docs.ansible.com/ansible/latest/user_guide/basic_concepts.html, April 2020, [Online; accessed April 15, 2020].

[11] Red Hat, Inc.: How to build your inventory. https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html#intro-inventory, April 2020, [Online; accessed April 15, 2020].

[12] Red Hat, Inc.: Intro to Playbooks. https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html#about-playbooks, April 2020, [Online; accessed April 15, 2020].

[13] Docker, Inc.: Docker Overview. https://docs.docker.com/get-started/overview/, 2020, [Online; accessed April 15, 2020].

[14] Emeth, E.; Snyder, G.; Hein, T.; aj.: *Unix And Linux System Administration Handbook*. Addison Wesley, 2017, ISBN 9780134277554, 915-948 s.

[15] Docker, Inc.: Virtual Machine Solution. https://www.docker.com/sites/default/files/d8/2018-11/container-vm-whatcontainer_2.png, 2020, [Online; accessed April 15, 2020].

[16] Docker, Inc.: What is a Container. https://www.docker.com/resources/what-container, 2020, [Online; accessed April 15, 2020].

[17] Docker, Inc.: Engine Components. https://docs.docker.com/engine/images/engine-components-flow.png, [Online; accessed April 15, 2020].

[18] Raghu, B.: *Mastering Linux Kernel Development*. Pack Publishing Limited, třetí vydání, 2017, ISBN 9781785883057.

[19] Taylor, T.: Application Containers vs. System Containers. https://www.sumologic.com/blog/application-containers-vs-system-containers-understanding-difference/, October 2020, [Online; accessed April 15, 2020].

[20] Savarese, C.; Hart, B.: Caesar Cipher. http://www.cs.trincoll.edu/~crypto/historical/caesar.html, April 2010, [Online; accessed April 15, 2020].

[21] Peterka, J.: *Bajecny svet elektronickeho podpisu*. CZ. NIC, 2011, ISBN 9788090424838.

[22] NTLM Protocol. https://cs.wikipedia.org/wiki/NTLM, February 2019, [Online; accessed April 15, 2020].

[23] Kerberos Protocol. https://en.wikipedia.org/wiki/Kerberos_(protocol), April 2020, [Online; accessed April 15, 2020].

[24] Microsoft: NTLM vs KERBEROS. [Online; accessed April 15, 2020]. Dostupné z: https://answers.microsoft.com/en-us/msoffice/forum/all/ntlm-vs-kerberos/d8b139bf-6b5a-4a53-9a00-bb75d4e219eb

[25] Microsoft: Kerberos Protocol Picture. [Online; accessed April 15, 2020]. Dostupné z: https://filestore.community.support.microsoft.com/api/images/c67335e9-6bad-405e-a2a7-91c400818fba?upload=true

[26] Microsoft: NTLM Protocol Picture. [Online; accessed April 15, 2020]. Dostupné z: https://filestore.community.support.microsoft.com/api/images/45bc59ef-a2e7-4a75-a129-8be12a01dd16?upload=true

[27] Burda, K.: *Kryptografie okolo nás*. CZ.NIC, z.s. p.o., 2019, ISBN 9788088168522, 24-26 s.

[28] Klíma, V.: Hašovací funkce, principy, příklady a kolize. http://crypto-world.info/klima/2005/cryptofest_2005.htm, March 2005, [Online; accessed May 15, 2020].

[29] Swinhoe, D.: The 14 biggest data breaches. https://www.csoonline.com/article/2130877/the-biggest-data-breaches-of-the-21st-century.html, March 2020, [Online; accessed April 15, 2020].

[30] OneLogin, Inc.: 6 Types of Password Attacks. https://www.onelogin.com/learn/6-types-password-attacks, 2020, [Online; accessed April 15, 2020].

[31] Arias, D.: Adding Salt to Hashing: A Better Way to Store Passwords. https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/, May 2018, [Online; accessed May 15, 2020].

[32] Mark, B.: *Cluster Computing White Paper*. University of Portsmouth, UK, December 2000.

[33] Red Hat, Inc.: What is a Kubernetes cluster. https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-cluster, 2020, [Online; accessed April 15, 2020].

[34] Alvarez Technology Group, Inc.: What Should I Know About Hashcat. https://www.alvareztg.com/what-should-i-know-about-hashcat/, 2020, [Online; accessed April 15, 2020].

Příloha **A**

Seznam použitých zkratek

API Application programming interface

App Aplikace

CPU Central processing unit

FIPS Federal Information Processing Standards

FPGA Fiels-programmable gate array

GPU Graphics processing unit

HTTP Hypertext transfer protocol

HTTPS Hypertext transfer protocol secure

IaC Infrastructure as Code

ID Identifikátor

IP Internet protocol address

NIST National Institute of Standards and Technology

OS Operační systém

PID Process identifier

RAM Random access memory

VM Virtual machine (virtuální počítač)

GB Gigabyte

DO DigitalOcean

OTP One Time Password

NIS Network Information Service

IPC Inter-process communication

JSON JavaScript Object Notation

Příloha B

Obsah přiloženého CD

- `Ansible` ..... adresář s deploymentem a vším co je potřeba k nasazení
  - `config` ....adresář s konfiguračními soubory pro nasazované aplikace
  - `deploy.yml`
  - `hosts` ..... inventory pro Ansible
  - `README.adoc`
  - `tasks` ..... úlohy
    - `clone-repo.yml`
    - `disable-swap-patition.yml`
    - `install-dependancies.yml`
    - `install-run-exporter.yml`
    - `install-run-grafana.yml`
    - `install-run-prometheus.yml`
    - `join-cluster.yml`
    - `mount-shared-storage.yml`
    - `setup-master-node.yml`
    - `setup-nfs-server.yml`
    - `setup-node.yml`
    - `user-creation.yml`
- `Docs` .....adresář s textem k práci
  - `main.tex` ..... hlavní soubor pro text
  - `Pictures` ..... složka s obrazky pro text
- `Kuberntes` .....deploymenty pro služby
  - `deployment-computationunit-dict.yaml`
  - `deployment-computationunit-mask.yaml`
  - `deployment-databaseservice.yaml`
  - `deployment-privateservice.yaml`
  - `deployment-publicservice.yaml`
- `src` ..... soubory se sluzbami a potřebnými soubory pro Docker
  - `Databaseservice`
    - `Dockerfile` ..... vytvoří obraz s Databaseservice
  - `Hashcatwrapper`
    - `main.py`
    - `Dockerfile` .....vytvoří obraz s Computationunit
  - `Privateservice`
    - `main.py`
    - `Dockerfile` ..... vytvoří obraz s Privateservice
  - `Publicservice`
    - `main.py`
    - `Dockerfile` .....vytvoří obraz s Publicservice
- `README.adoc` ..... Popis repozitáře