Zvýšenie odolnosti webových aplikácií proti útokom typu DDoS

Bc. Tomáš Hoffer
FIIT STU
Fakulta informatiky a informačných technológií STU
Bratislava, Slovenská republika
xhoffer@stuba.sk

***Abstrakt*—V tejto práci sa venujem problematike ochrany webového servera pred útokmi typu DDoS. V teoretickej časti opisujem útoky DDoS z pohľadu ich definície, taxonómie a známych DDoS útokov. Venujem sa tiež opisu metód detekcie a prevencie voči týmto útokom. Experimentálna časť práce je venovaná ochrane webového servera pomocou servera NGINX, ktorý je využívaný na "load-balancing", "rate-limiting" a "caching". Implementačná časť popisuje implementačné detaily experimentu, ako aj vyhodnotenie účinnosti jednotlivých prevenčných metód. Kombináciou týchto metód je možné dosiahnuť účinnú ochranu proti útokom DDoS.**

***Kľúčové slová*—DDoS útoky, ochrana, NGINX, rate-limiting, load-balancing, caching**

I. Útoky typu DDoS

Útoky DDoS sú rovnako ako útoky DoS zamerané na dostupnosť služieb. Ich cieľom je znemožniť používateľom prístup k danej službe alebo prostriedkom. Na rozdiel od útokov typu DoS ("Denial of Service“) je útok distribuovaný a pakety tak pochádzajú z veľkého množstva hostov. Ochrana pred útokmi typu DDoS nie je triviálna práve z dôvodu náročnej identifikácie zdroja útoku. Motiváciou pre realizáciu DDoS útokov môže byť finančný zisk, pomsta, či poškodenie reputácie.[8]

Typickým cieľom útočníka v prvej fáze DDoS útoku je identifikácia hostov so zraniteľnosťami, pomocou ktorých útočník následne dosiahne ich kompromitáciu. Tieto zariadenia sú nazývané aj "zombie“ stroje. S využitím množstva kompromitovaných "zombie“ strojov útočník dokáže generovať veľký objem požiadaviek a znížiť tak dostupnosť cieľovej služby. Častou technikou, využívanou pri DDoS útokoch je tzv. "IP spoofing“, vďaka ktorému útočník falšuje zdrojové IP adresy v hlavičkách IP paketov, čím dokáže predstierať alebo skryť svoju identitu. Vďaka tejto technike je náročné identifikovať zdroj útoku a zmierniť tak jeho dopad.[9]

A. Taxonómia útokov DDoS

J. Mirkovic a spol.[18] kategorizujú DDoS útoky na základe troch charakteristík: spôsob prípravy útoku, spôsob realizácie útoku a dopad útoku.

1) Spôsob prípravy útoku: Samotná príprava útoku môže byť v rôznej miere automatizovaná. V niektorých prípadoch útočník manuálnym spôsobom hľadá zraniteľnosti v strojoch, ktoré po kompromitácii následne využije ako "zombie" stroje. Manuálna príprava je však v dnešnej dobe ojedinelá. Útočníci častejšie automatizujú úvodnú kompromitáciu "zombie" strojov, vrátane identifikácie zraniteľných zariadení a následného exploitu. Vykonaním príkazu útočník po kompromitácii špecifikuje cieľ útoku a následne útok spustí. Niektoré útoky sú tiež plne automatizované. Cieľ útoku a podmienky spustenia sa pri tomto type útoku nachádzajú v zdrojovom kóde, čím útočník eliminuje potrebu komunikovať s kompromitovanými zariadeniami a komplikuje tak jeho identifikáciu.[18]

Spôsob prípravy sa tiež môže líšiť z pohľadu stratégie hľadania zraniteľných hostov. V prvej fáze prebieha skenovanie hostov s cieľom identifikovať potenciálne zraniteľné zariadenia. Skenovanie môže prebiehať náhodným spôsobom, no častejšie využívanou metódou je tzv. "hitlist scanning". Metóda využíva pred-definovaný zoznam IP adries, získaný z verejne dostupných zdrojov. Metóda "signpost scanning" využíva interakcie medzi jednotlivými hostami na identifikáciu ďalších hostov. Podobne ako vírus typu červ sa šíri po sieti (napr. mailovou komunikáciou). Skenovanie síce prebieha pomalšie, no znižuje pravdepodobnosť jeho odhalenia.[18]

2) Spôsob realizácie útoku: Po kompromitácii "zombie" strojov útočník pomocou týchto strojov generuje veľký objem požiadaviek na cieľový server. Samotný tok požiadaviek je vo väčšine prípadov konštantný a útočník generuje maximálny objem požiadaviek vzhľadom na prostriedky "zombie" stroja. Variabilný tok požiadaviek umožňuje útočníkovi oddialiť detekciu prebiehajúceho útoku vďaka dočasnému zníženiu toku.

Z pohľadu využitých "zombie" strojov môžeme útoky kategorizovať na:

- Útoky s konštantnou množinou agentov: Útočník využíva rovnakú množinu agentov počas celého útoku.
- Útoky s variabilnou množinou agentov: Útočník rozdelí "zombie" stroje do viacerých skupín, ktoré počas priebehu útoku obmieňa a komplikuje tak odvrátenie útoku.

Cieľom útoku môže byť aplikačný server, fyzický stroj, sieťový komponent alebo kľúčová infraštruktúra.[18]

3) Dopad útoku: Z pohľadu dopadu útoku J. Mirkovic a spol.[18] kategorizujú útoky na *disruptívne* a *degradačné*. *Disruptívne* útoky sú realizované s cieľom úplnej eliminácie dostupnosti služby. Najčastejším prípadom sú práve útoky tohto typu. *Degradačné* útoky iba čiastočne degradujú dostupnosť služby, čím útočník z veľkej časti dosiahne svoj cieľ. Útok zväčša trvá dlhšiu dobu a býva ťažšie odhaliteľný.[18]

B. Známe útoky DDoS

Prvý známy DDoS útok bol zaznamenaný v roku 1999 s využitím malvéru Shaft. Jedná sa o prvý útok, v ktorom boli kombinované známe DoS techniky („TCP SYN flood“, „UDP flood“) s distribuovaným a koordinovaným prístupom.[10] Od roku 2000 vznikali čoraz sofistikovanejšie DDoS útoky. Doposiaľ najväčší zaznamenaný útok sa udial v roku 2017, kedy spoločnosť Google čelila útoku o veľkosti 2.54 Tbps.[6] Útok podobných rozmerov zaznamenala aj spoločnosť Amazon v roku 2020.[5]

V roku 2018 čelila DDoS útoku aj spoločnosť GitHub.[15] Útočníci zneužili databázový server Memcached, ktorý spoločnosť využíva na "caching" obsahu. Pomocou techniky IP "spoofing" vygenerovali veľký objem paketov, v ktorých sfalšovali hlavičku odosielateľa. Keďže Memcached databáza využíva protokol UDP, pri komunikácii neprebieha TCP "handshake" a Memcached server zahltil svojimi odpoveďami cieľový server, ktorého IP adresa sa nachádzala v sfalšovaných hlavičkách paketov. Spoločnosť Github však útok zaznamenala už po 10 minútach. Po 20 minútach sa útok podarilo odvrátiť.[15]

Spoločnosť Github bola na útok riadne pripravená najmä z dôvodu podobného útoku, ktorému čelila v roku 2015, kedy útočníci vložili škodlivý kód do vyhľadávacieho nástroja Baidu. Škodlivý kód spôsobil, že prehliadač každého návštevníka stránky vyhľadávača generoval veľký objem HTTP požiadaviek na servery spoločnosti Github.[11]

Známy je tiež útok na poskytovateľa DNS služieb Dyn z roku 2016. Útočníci vytvorili malvér, pomocou ktorého kompromitovali IoT zariadenia, ktoré následne zahltili server požiadavkami. Útok spôsobil nedostupnosť služieb Airbnb, Netflix, PayPal, Visa, Amazon, The New York Times, Reddit, a GitHub po dobu jedného dňa.[11] Dopad známych útokov len potvrdzuje dôležitosť ochrany proti útokom tohto typu.

C. Detekcia útokov DDoS

Predpokladom pre úspešnú detekciu prebiehajúceho DDoS útoku je porozumenie bežnej sieťovej prevádzky. Bez dôsledného porozumenia sieťovej prevádzky by sme totiž nedokázali vyhodnotiť prítomnosť útoku. Meranie počtu HTTP požiadaviek, ktoré počas bežnej prevádzky server obsluhuje pomôže nie len pri škálovaní služby počas dňa, ale aj pri nastavení tzv. "alerting"-u. Ďalšou významnou metrikou môže byť napr. latencia (dĺžka trvania požiadavky), využitie HW prostriedkov servera alebo počet chybových odpovedí.[20] Po pekročení úrovne bežnej sieťovej prevádzky môže systém upozorniť administrátora a umožniť tak včasnú mitigáciu útoku. Nápomocným môže byť tiež porozumenie DDoS útokov z minulosti a ich typických charakteristík.[1, 8] Vo všeobecnosti môžeme detekčné mechanizmy kategorizovať do dvoch základných skupín - detekcia na základe príznakov a detekcia na základe anomálií.[18, 8]

1) Detekcia na základe príznakov: Detekčné mechanizmy postavené na tomto prístupe využívajú databázu prízakov typických pre DDoS útoky a monitorujú prítomnosť týchto príznakov v sieťovej komunikácii.[18] Databáza však musí byť pravidelne aktualizovaná. Nevýhodou tohoto prístupu je, že dokáže odhaliť len známe útoky z minulosti, ktoré sa nachádzajú v databáze. Akúkoľvek variáciu týchto útokov, rovnako aj nové útoky už mechanizmus neodhalí. Typickým príkladom nástroja využívajúceho túto detekčnú metódu je nástroj Snort.[24] Typickým príznakom môže byť napr. vysoký počet otvorených TCP spojení. [18, 8]

2) Detekcia na základe anomálií: Predpokladom pre úspešnú detekciu na základe anomálií je monitorovanie a porozumenie bežnej sieťovej prevádzky, ktorému som sa venoval v podkapitole I-C. Na základe týchto informácií je možné vytvoriť model bežnej sieťovej prevádzky, ktorý je pavidelne porovnávaný s modelom aktuálnej prevádzky. Model aktuálnej prevádzky však musí byť pravidelne aktualizovaný, najmä v heterogénnych prostrediach, pre ktoré sú výkyvy typické. V prípade diskrepancie (anomálie) sa môže jednať o DDoS útok a systém tak upozorní administrátora. Výhodou tohoto prístupu oproti detekcii na základe príznakov je, že dokáže odhaliť nové typy útokov. Nevýhodou je ale vyšia miera falošnej pozitivity. Nastavenie správnych prahových hodnôt (angl. "threshold") pre detekciu anomálie je tiež netriviálnym problémom.[18, 8]

D. Prevencia voči útokom DDoS

Napriek náročnosti zabrániť útokom DDoS existujú techniky prevencie, pomocou ktorých dokážeme znížiť dopad takýchto útokov alebo ich úplne eliminovať. Základom prevencie je eliminácia tzv. „zero-day“ zraniteľností pred tým, ako ich objaví a zneužije útočník. Súčasťou prevencie voči útokom DDoS je aj definícia plánu obnovy prevádzky, zahŕňajúceho návody, školenie zamestnancov, procesy a komunikačný plán.[1, 8] V nasledujúcich podkapitolách sa však zameriam len na technický aspekt prevencie voči DDoS útokov.

1) Rate-limiting: Prvá reakcia na útok DoS/DDoS by mala byť utomatizovaná, pretože ľudský zásah trvá podstatne dlhší čas a charakter útoku sa môže rýchlo meniť.[19] "Rate-limiting" spočíva v zahodení prichádzajúcich požiadaviek s určitou pravdepodobnosťou tak, aby priepustnosť nepresiahla maximálnu priepustnosť servera. "Rate-limiting" môže byť realizovaný na viacerých vrstvách. Kým na úrovni smerovača môžeme obmedzovať počet prichádzajúcich paketov (sieťová vrstva), na úrovni proxy servera dokážeme obmedzovať počet prichádzajúcich HTTP požiadaviek (aplikačná vrstva). Okrem samotného "rate-limiting"-u je účinným mechanizmom tiež obmedzenie počtu otvorených TCP spojení (angl. "connection limiting") alebo obmedzenie priepustnosti pri sťahovaní súborov (angl. "bandwidth limiting").[16]

Metóda však nerozlišuje medzi požiadavkami od skutočných používateľov a požiadavkami od útočníka. "Rate-limiting" teda môže mať za následok dočasné zníženie dostupnosti služby, čo môže byť práve cieľom útočníka. V kombinácii s ostatnými metódami prevencie však môže táto metóda poskytnúť účinnú ochranu servera pred zahltením. [19, 3]

2) Vyrovnávanie záťaže: Vyrovnávanie záťaže je v súčasnosti vo veľkej miere využívané pri distribuovaných výpoč-

toch v prostredí cloudu. Pomocou rôznych navrhnutých algoritmov pre vyrovnávanie záťaže systémy dosahujú vyššiu priepustnosť, škálovateľnosť a v neposlednom rade dostupnosť.[2]

Hoci je vyrovnávanie záťaže primárne určené na zvýšenie priepustnosti webových služieb, môže slúžiť aj ako metóda prevencie DDoS útokov. Zvýšením priepustnosti systému dokážeme znížiť dopad prípadného DDoS útoku, pretože útočník na zahltenie potrebuje vyvinúť väčší objem požiadaviek. Algoritmy vyrovnávania záťaže môžeme rozdeliť do dvoch skupín – statické a dynamické. Statické prístupy sú vhodné do homogénnych a stabilných prostredí, v ktorých neočakávame významné výkyvy. Zohľadňujú rôzne statické atribúty systému ako napr. CPU, pamäť RAM, veľkosť úložiska a priepustnosť siete. Ich nevýhodou však je, že nedokážu pružne reagovať na zmeny počas prevádzky. Dynamické prístupy zohľadňujú rôzne atribúty systému pred aj po uvedení do prevádzky, napr. aktuálne vyťaženie uzlov. Sú náročnejšie na implementáciu z dôvodu potreby monitorovania týchto údajov v reálnom čase. Sú však vhodnejšie do heterogénnych prostredí, v ktorých očakávame časté výkyvy.[2]

Pri výbere algoritmu pre dynamické vyrovnávanie záťaže je potrebné zohľadniť niekoľko faktorov:

- *Priestorová distribúcia uzlov:* Pri vyrovnávaní záťaže je dôležité zohľadniť aj vzdialenosť medzi jednotlivými uzlami a systémom na vyrovnávanie záťaže. Niekedy môže byť vhodnejšie smerovať požiadavku na uzol, ktorý síce je viac vyťažený, no leží bližšie k systému na vyrovnávanie záťaže. Najmä v prípade pomalej siete.
- *Zložitosť algoritmu:* Čím jednoduchší je algoritmus, tým menej prostriedkov spravidla potrebuje. Nízka zložitosť algoritmu je zväčša preferovanou voľbou.
- *Eliminácia jedného bodu zlyhania:* V prípade zlyhania systému na vyrovnávanie záťaže môže dôjsť k zlyhaniu celého systému. Samotný systém je teda lákavým terčom útokov a je preto vhodné nasadiť ho tiež v distribuovanom režime.

[2]

3) Caching: Caching je technika umožňujúca zníženie záťaže samotného servera pri poskytovaní obsahu klientovi. Obsah je pre klientov v čo najvyššej miere pred-pripravený a uložený vo vyrovnávacej pamäti s cieľom minimalizácie potreby výpočtových prostriedkov pri jeho poskytovaní.[26]

Pokiaľ klient navštívi web stránku, server musí skompilovať jej obsah a odoslať ho klientovi. Kompilácia obsahu je však výpočtovo náročná operácia a pri veľkom množstve požiadaviek môže dôjsť k zahlteniu servera. "Caching" je vhodný najmä v situáciach, keď sa poskytovaný obsah často nemení. Typickým príkladom je statický obsah webových stránok, ako napr. html, css, fonty, obrázky, či videá. Ďalším častým príkladom využitia je "caching" odpovedí databázového servera. Pokiaľ by server mal vykonať dopyt na databázu s každou prichádzajúcou požiadavkou, mohlo by dôjsť k zahlteniu databázového servera a ukladanie obsahu do "cache" pamäte môže byť efektívnym nástrojom ochrany. Pokiaľ je však konzistencia dát prioritou, nemusí byť "caching" vhodným riešením. Je preto dôležité prehodnotiť, ktorý obsah bude do tejto pamäte ukladaný. Ďalším možným využitím je "caching" v prípade nedostupnosti služby. Pokiaľ volaná služba dočasne prestane odpovedať, systém môže vrátiť odpoveď uloženú v cache pamäti a zachovať tak vysokú dostupnosť. [26]

4) Siete CDN: Siete CDN sú využívané pre distribúciu webového obsahu naprieč geografickým územím s cieľom v čo najvyšej miere priblížiť obsah konečnému používateľovi.[13] Čím bližšie sa fyzický server nachádza, tým kratší čas trvá sieťový prenos a načítanie stránky. Výhodou CDN sietí je, že vo väčšine prípadov ukladajú obsah do vyrovnávacej pamäte. Účinnosti "caching"-u pri ochrane proti DDoS útokom sa bližšie venujem v podkapitole I-D3. Poskytujú tiež filtrovacie mechanizmy pomocou WAF, ktorým je venovaná podkapitola I-D5. Servery sú na CDN sieťach tiež replikované, čím je zvýšená ich dostupnosť pomocou vyrovnávania záťaže, opísaného v podkapitole I-D2. Siete CDN teda kombinujú viacero mechanizmov, čím sa stávajú účinným nástrojom na ochranu pred DDoS útokmi.[13]

5) Inšpekčné systémy WAF a IDS: WAF ("Web application firewall") a IDS ("Intrusion detection system") sú stavové detekčné systémy, ktoré analyzujú prichádzajúcu a odchádzajúcu sieťovú komunikáciu s cieľom detekcie a prevencie potenciálnych hrozieb. Analýza pri systémoch IDS/IPS môže prebiehať na základe bázy pravidiel, ktoré popisujú známe útoky z minulosti. Využívané sú tiež metódy strojového učenia, najmä neutónové siete.[4]

Systémy WAF operujú na aplikačnej vrstve OSI modelu a analyzujú prichádzajúce HTTP požiadavky. Typicky chránia webové a aplikačné servery pred útokmi typu XSS, CSRF, SQL injection a DoS. WAF slúži ako "reverse-proxy", cez ktorú musí prejsť každá HTTP požiadavka pred tým, ako ju prijme server. WAF tiež využíva bázu pravidiel a jej výhodou je možnosť rýchlej úpravy pravidiel v závislosti od aktuálnej potreby. V prípade podozrenia na prebiehajúci DOS/DDoS útok je tak možné využiť WAF na "rate-limiting" (viac v podkapitole I-D1) alebo úplné blokovanie vybraných hostov. Cloudová služba Cloudflare WAF[7] dokonca poskytuje ochranu proti známym "zero-day" zraniteľnostiam, ochranu proti OWASP Top 10 zraniteľnostiam, metódy strojového učenia na detekciu potencialnych hrozieb, či ochranu pred zneužitím ukradnutých údajov.[7] Predpokladom pre úspenosť týchto systémov je však ich správne umiestnenie a nastavenie. Dôležitá je tiež kombinácia s ostatnými prevenčnými technikami.[4]

6) Captcha: Akronym "Captcha" v angl. jazyku znamená "Completely Automated Public Turning Test to tell Computers and Humans Apart".[21] Metóda je využívaná na odlíšenie botov od ľudí v internetovom priestore. Môže teda slúžiť ako efektívna metóda prevencie, ktorá pri DDoS útoku odlíši "zombie" stroje od skutočných používateľov. Požiadavky od "zombie" strojov sú zablokované a nedôjde tak k zahlteniu servera. Základným prvkom metódy je jednoduchý test, o ktorom existuje vysoký predpoklad, že dokáže byť vyriešený len človekom. Pokiaľ používateľ správne vyrieši daný test, nástroj ho považuje za človeka a jeho aktivita nebude bloko-

vaná.[21, 23]

Typické príklady testu zahŕňajú napríklad:

- Porušený text na obrázku, ktorý používateľ prepíše do textového okna. Text je často zložený z rôznych fontov, veľkostí písma, farby a orientácie znakov.
- Jednoduchá matematická úloha.
- Rozpoznanie objektov na obrázku alebo označenie všetkých obrázkov, ktoré obsahujú zadaný objekt.

Nástroje "captcha" môžu byť účinnou metódou prevncie pokiaľ je vhodne nastavená náročnosť úlohy. Úloha nesmie byť príliš jednoduchá aby ju nebolo možné vyriešiť pomocou OCR, algoritmov na rozpoznávanie objektov, či umelej inteligencie. Úloha však nemôže byť ani príliš náročná aby neodradila samotného používateľa.[21, 23]

II. Motivácia a cieľ experimentu

Cieľom experimentu je ochrana webového servera pred simulovaným útokom typu DDoS. Implementácia efektívnej ochrany proti DDoS útokom môže byť časovo aj finančne náročná. Hlavnou motiváciou projektu bolo nájdenie riešenia, ktoré nie je náročné na implementáciu a zároveň poskytuje efektívnu ochranu pred DDoS útokmi. Podmienkou bola tiež distribúcia pod "open-source" licenciou. Na základe analýzy metód prevencie, opísaných v kapitole I-D, som sa rozhodol využiť "load-balancing", "rate-limiting" a "caching". Všetky tieto metódy totiž poskytuje webový server NGINX v open-source verzii. Na úrovni proxy vrstvy v podobe NGINX servera teda dokážeme implementovať hneď niekoľko ochranných mechanizmov, čo bolo hlavným dôvodom výberu NGINX servera. Cieľom experimentu je overenie účinnosti spomenutých metód pri ochrane webového servera proti útokom typu DDoS.

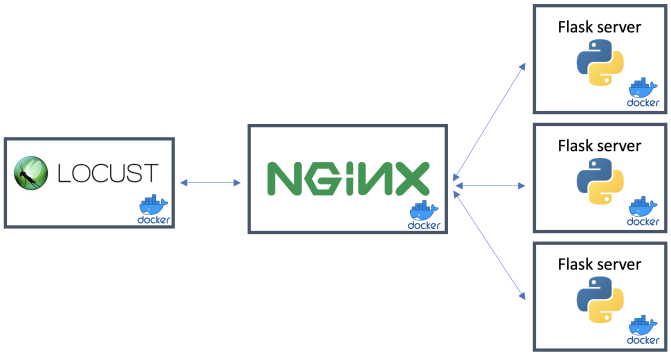
III. Návrh riešenia

Pre účely realizácie experimentu je potrebné vytvoriť testovacie prostredie, ktoré je v čo najvyššej miere podobné reálnemu produkčnému prostrediu. Dôležitou charakteristikou testovacieho prostredia je teda použitie produkčných serverov a konfigurácií. Samozrejmosťou je aj virtualizácia jednotlivých prvkov aby sme v čo najvyšej miere zabránili vzájomnému súpereniu o prostriedky (CPU, pamäť RAM, priestor na disku). Obrázok 1 znázorňuje hlavné prvky testovacieho prostredia - webový server flask, NGINX server a nástroj Locust, slúžiaci na simuláciu DDoS útoku. Počas všetkých experimentov je pomocou nástroja Locust simulovaných 500 paralelných klientov ("zombie" strojov), ktoré generujú požiadavky s cieľom zahltenia webového servera. Proxy server NGINX slúži ako ochranný mechanizmus.

IV. Implementácia riešenia

A. Webový server

Dôležitým prvkom prostredia je samotný webový server, ktorý je cieľom simulovaného DDoS útoku. Pre implementáciu servera som sa rozhodol využiť jazyk Python a rámec flask[12] vo verzii 2.0.3. Flask je jednoduchý rámec na tvorbu webových aplikácii v jazyku Python s využitím špecifikácie



Obr. 1. Testovacie prostredie

```
import socket
from flask import Flask, Response

app = Flask(__name__)
hostname = socket.gethostname()
local_ip = socket.gethostbyname(hostname)

@app.route('/')
def hello_world():
  resp = Response("Hello from " + local_ip)
  return resp

if __name__ == '__main__':
  app.run()
```

Kód 1. Zdrojový kód webového servera

WSGI ("Web server gateway interface")[27]. WSGI definuje rozhranie medzi rámcom na tvorbu webových aplikácii a webovým serverom. Väčšina dnešných rámcov podporuje WSGI štandard a zjednodušuje tak nasadenie na ľubovoľný WSGI server. Kód 1 spúšťa webový server, ktorého odpoveďou je text "Hello from", nasledovaný IP adresou odpovedajúceho servera.

Flask poskytuje jednoduchý server na vývojárske účely. Tento server však nie je vhodný na produkčné nasadenie vzhľadom na jeho nízku priepustnosť. Pre produkčné nasadenie som sa rozhodol použiť webový server gunicorn[14] z dôvodu jednoduchého nasadenia, nízkej náročnosti na prostriedky a podpory WSGI štandardu.

Server je virtualizovaný s využitim nástroja Docker. Nasledujúci Dockerfile súbor(Kód 2) definuje jeho Docker obraz. Využívajúc python verziu 3.10 je aplikácia spustená na "localhost"-e a počúva na porte 5000. Posledný príkaz inštaluje závislosti potrebné na spustenie servera, definované v súbore *requirements.txt*.

B. Server NGINX ako ochrana pred DDoS útokmi

Docker obraz pre server NGINX je definovaný Dockerfile súborom v Kóde 3. Vychádzajúc z obrazu pre OS Ubuntu, pomocou príkazu *apt install* nainštalujeme potrebné systémové

```
FROM python:3.10-alpine
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1
EXPOSE 5000
COPY . .
RUN pip install -r requirements.txt
```

Kód 2. Dockerfile súbor webového servera

```
FROM ubuntu:18.04

RUN apt update
RUN apt install libpcre3 libpcre3-dev zlib1g \
    zlib1g-dev libssl-dev build-essential \
    wget tar -y
RUN wget \
    https://nginx.org/download/nginx-1.21.6.tar.gz
RUN tar -zxvf nginx-1.21.6.tar.gz

# Copy configuration
RUN rm -f /etc/nginx/nginx.conf
COPY nginx.conf /etc/nginx/

RUN mkdir /cache

WORKDIR nginx-1.21.6
RUN ./configure --sbin-path=/usr/bin/nginx   \
  --conf-path=/etc/nginx/nginx.conf   \
  --error-log-path=/var/log/nginx/error.log   \
  --http-log-path=/var/log/nginx/access.log   \
  --pid-path=/var/run/nginx.pid

RUN make
RUN make install

EXPOSE 8081
CMD ["/usr/bin/nginx", "-g", "daemon off;"]
```

Kód 3. Dockerfile súbor NGINX servera

knižnice. Prikazom *wget* stiahneme NGINX vo verzii 1.21.6 a následne ho rozbalíme. Po skopírovaní konfiguračného súboru a vytvorení adresára pre "cache" spustíme príkaz *configure*, ktorým vytvoríme konfiguráciu servera pred sustením. Príkazmi *make* a *make install* spustíme "build" a vytvoríme spustiteľný binárny súbor. Server počúva na porte 8081 a spustený je posledným príkazom CMD.

C. Nástroj Locust

Nástroj Locust[17] umožňuje generovanie veľkého objemu HTTP požiadaviek distribuovaným spôsobom. Jeho výhodou je jednoduchá škálovateľnosť pri nasadení. *Master* proces je zodpovedný za orchestráciu jednotlivých úloh, ktoré deleguje na *worker* procesy. Počet *worker* procesov je konfigurovateľný a jediným praktickým obmedzením je tak počet CPU jadier na stroji. Nástroj dynamicky upravuje generovanú záťaž vzhľadom na vyťaženie servera. Po istom čase teda konverguje k maximálnej priepustnosti, ktorú server poskytuje pri čo najnižšej miere chybovosti. Pomáha teda pri odhaľovaní limitov v priepustnosti. Nástroj vykonáva testovací kód umiestnený v súbore *locustfile.py* (Kód 4). V našom prípade sa

```
from locust import HttpUser, task

class DemoUser(HttpUser):
    @task
    def test(self):
        self.client.get("/")
```

Kód 4. Testovací kód súboru locustfile.py

```
web1:
    build: ./app
    command: gunicorn \
       --bind 0.0.0.0:5000 app:app
    expose:
      - 5000
    cpu_count: 1
```

Kód 5. Služba web_N v súbore *docker-compose.yml*

jedná o jednoduché volanie HTTP požiadaviek na koreňovú URL.

D. Nasadenie

Testovacie prostredie je nasadené na jednom fyzickom stroji s využitím nástroja Docker-compose. Stroj disponuje 16CPU a 24Gb RAM. Súbor *docker-compose.yml* obsahuje definíciu všetkých služieb testovacieho prostredia.

Služba web_N definuje gunicorn webový server, počúvajúci na porte 5000. Služba má priradené 1 CPU a v prostredí sa nachádzajú 3 takéto služby. (Kód 5)

Služba *nginx* definuje NGINX server, ktorý počúva na porte 8081, má priradené 2 CPU a jej spustenie prebehne v okamihu, keď sú spustené služby $web[1...3]$. (Kód 6)

Služba *locust-master* definuje master proces, orchestrujúci locust úlohy. Proces počúva na porte 8089 a spúšťa spomínaný súbor *locustfile.py*. Proces má priradené 1 CPU. (Kód 7)

Služba *locust-worker* definuje samotný worker proces, vykonávajúci locust úlohy. Proces má tiež priradené 1 CPU, nakoľko nedokáže využiť viac CPU jadier. (Kód 8)

Spustenie testovacieho prostredia je vykonané príkazom *docker-compose up –build –scale locust-worker=10*. Príkaz

```
nginx:
    build: ./nginx
    container_name: nginx
    restart: always
    ports:
      - 8081:8081
    depends_on:
      - web1
      - web2
      - web3
    cpu_count: 2
```

Kód 6. Služba *nginx* v súbore *docker-compose.yml*

```
locust-master:
    image: locustio/locust
    ports:
      - "8089:8089"
    volumes:
      - ./:/mnt/locust
    command: -f /mnt/locust/locustfile.py \
        --master \
        --config=/mnt/locust/locust.conf
    cpu_count: 1
```

Kód 7. Služba *locust-master* v súbore *docker-compose.yml*

```
locust-worker:
    image: locustio/locust
    volumes:
      - ./:/mnt/locust
    command: -f /mnt/locust/locustfile.py \
        --worker\
        --master-host locust-master
    cpu_count: 1
```

Kód 8. Služba *locust-worker* v súbore *docker-compose.yml*

vytvorí virtualizované prostredie obsahujúce 3 repliky webového servera, server NGINX, locust master proces a 10 replík locust-worker procesov.

V. Ochrana webového servera pred útokmi DDoS pomocou servera Nginx

A. NGINX Load-balancing

1) Algoritmus Round-robin: Server NGINX poskytuje viacero algoritmov na vyrovnávanie záťaže. Predvoleným algoritmom je algoritmus Round-robin, ktorý rovnomerne rozdeľuje záťaž medzi jednotlivé repliky servera. Súbor *nginx.conf* obsahuje nastavenie vyrovnávania záťaže medzi 3 repliky servera s využitím tohto algoritmu (Kód 9).

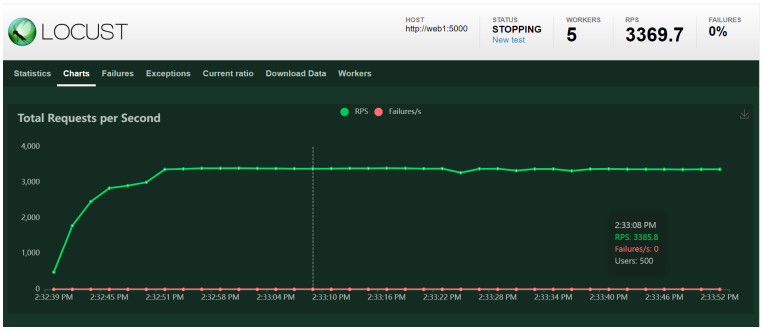
Priepustnosť samotného webového servera bez využitia vyrovnávania záťaže dosiahla 3369 požiadaviek za sekundu pri 4% chybovosti. Graf na Obr. 2 znázorňuje priebeh testu pomocou nástroja locust. Po zapnutí vyrovnávania záťaže medzi 3 replikami s využitím algoritmu Round-robin dosiahlo riešenie priepustnosť 7649 požiadaviek za sekundu pri rovnakej miere chybovosti (Obr. 3). Môžeme teda konštatovať, že vyrovnávanie záťaže úspešne zvýšilo priepustnosť servera.

2) Algoritmus Least-connected: Ďalším algoritmom, ktorý NGINX poskytuje je algoritmus Least-connected. Algoritmus funguje na princípe dynamického škálovania, zohľadňujúc tak aktuálne vyťaženie jednotlivých replík. NGINX interne eviduje informácie o aktuálnom vyťažení jednotlivých replík, napr. latencie odpovedí, počet otvorených TCP spojení alebo počet "timeout"-ov. Vďaka týmto informáciam pomocou algoritmu Least-connected dokáže smerovať prichádzajúce požiadavky na tie repliky, ktoré sú v danom okamihu najmenej vyťažené.[25]

```
events {}
http {
 upstream localhost {
    server web1:5000;
    server web2:5000;
    server web3:5000;
 }
 server {
    listen 8081;
    server_name localhost;

    location / {
       proxy_pass http://localhost;
    }
  }
}
```

Kód 9. Súbor *nginx.conf* s konfiguráciou vyrovnávania záťaže pomocou algoritmu Round-robin



Obr. 2. Priebeh testu nad webovým serverom bez využitia vyrvnávania záťaže.

Pre zapnutie algoritmu Least-connected je potrebná nasledujúca úprava v súbore *nginx.conf* (Kód 10).

Pre účely experimentu je tiež potrebné upraviť webový server. Ako som opísal v podkapitole IV-A, aktuálny server realizuje jednoduchú operáciu, v ktorej vráti textovú odpoveď. Keďže nevykonáva žiaden náročný výpočet, dopyt na databázu, či akúkoľvek asynchrónnu operáciu, je vysoko

Obr. 3. Priebeh testu nad webovým serverom s využitím vyrovnávania záťaže algritmom Round-robin.

```
upstream localhost {
    least_conn;
    server web1:5000;
    server web2:5000;
    server web3:5000;
}
```

Kód 10. Súbor *nginx.conf* s konfiguráciou vyrovnávania záťaže pomocou algoritmu Least-connected

```
@app.route('/')
def hello_world():
  resp = Response("Hello from " + local_ip)
  if bool(random.getrandbits(1)):
    time.sleep(0.5)
  return resp
```

Kód 11. Zdrojový kód webového servera po úprave

pravdepodobné, že latencie všetkých požiadaviek budú takmer rovnaké. Rozdiel medzi algoritmami Round-robin a Least-connected by sa teda neprejavil, nakoľko všetky repliky vykonávajú rovnakú úlohu, ktorá im zaberie rovnaký čas. Túto hypotézu som overil aj experimentom, v ktorom bol rozdiel v prepustnosti len 300 požiadaviek za sekundu v prospech algoritmu Least-connected.

Jeden z troch webových serverov som teda upravil nasledujúcim spôsobom (Kód 11). Server s 50% pravedpodobosťou zavolá operáciu *sleep* s dĺžkou 500ms. Operáciou tak simuluje náročný výpočet, dopyt na databázu, či inú časovo náročnú operáciu. V testovacom prostredí sa teda nachádzajú 2 repliky pôvodného servera a 1 replika upraveného(pomalšieho) servera.

Systém s vyrovnávaním záťaže algoritmom Round-robin dosiahol priepustnosť len 10 požiadaviek za sekundu, keďže 1/3 požiadaviek bola smerovaná na pomalý server a server sa tak stal úzkym hrdlom. (Obr. 4)

Algoritmus Least-connected naopak odhalil, že server odpovedá pomalšie a smeroval naň len malý objem požiadaviek. Rozdiel v priepustnosti oproti alogritmu Round-robin je signifikantný. (Obr. 5)

Obr. 4. Priebeh testu nad webovým serverom s využitím vyrovnávania záťaže algritmom Round-robin. Algoritmus smeruje 1/3 záťaže na pomalý server.

Obr. 5. Priebeh testu nad webovým serverom s využitím vyrovnávania záťaže algritmom Least-connected. Systém dosiahol priepustnosť 6027 požiadaiek za sekundu.

```
➜  ~ siege -v -r 5 -c 2 http://localhost:8081
** SIEGE 4.1.2
** Preparing 2 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200     0.01 secs:      21 bytes ==> GET  /
HTTP/1.1 200     0.01 secs:      21 bytes ==> GET  /
HTTP/1.1 200     0.00 secs:      21 bytes ==> GET  /
HTTP/1.1 200     0.00 secs:      21 bytes ==> GET  /
HTTP/1.1 200     0.01 secs:      21 bytes ==> GET  /
HTTP/1.1 200     0.01 secs:      21 bytes ==> GET  /
HTTP/1.1 200     0.00 secs:      21 bytes ==> GET  /
HTTP/1.1 200     0.00 secs:      21 bytes ==> GET  /
HTTP/1.1 200     0.00 secs:      21 bytes ==> GET  /
HTTP/1.1 200     0.00 secs:      21 bytes ==> GET  /

Transactions:                     10 hits
Availability:                 100.00 %
Elapsed time:                   0.02 secs
```

Obr. 6. Priebeh testu nad systémom s vypnutým "rate-limiting"-om. Všetky požiadavky sú úspešne obslúžené.

B. NGINX Rate-limiting

Pre účely demonštrácie účinnosti "rate-limiting"-u som sa rozhodol použiť nástroj Siege[22]. Nástroj umožňuje generovanie HTTP požiadaviek a simuláciu viacerých paralelných klientov.

Prepínač "-c" definuje počet klientov a prepínač "-r" definuje počet požiadaviek, ktoré každý klient volá. Požiadavky sú volané paralelne. Na Obr. 6 vidíme priebeh testu s 2 klientami, volajúcimi 5 požiadaviek. Keďže je "rate-limiting" vypnutý, všetky požiadavky sú ihneď obslúžené a dostupnosť služby je 100%.

"Rate-limiting" je na serveri NGINX konfigurovaný v súbore *nginx.conf* (Kód 12). Nastavením *limit_req_zone* vytvoríme zónu *myratelimit*. IP adresy klientov, z ktorých prichádzajú HTTP požiadavky evidujeme z dôvodu pamäťovej efektivity v binárnom formáte. Veľkosť pamäte vyhradenej pre túto tabuľku obmedzujeme na 10mb, čo by malo stačiť pre uloženie 2,5mil. IP adries. "Rate-limit" je nastavený na 1 požiadavku za sekundu.

Pri rovnakom teste na Obr. 7 vidíme, že obslúžená bola len jedna požiadavka. Po prijatí prvej požiadavky bol dosiahnutý "rate-limit" 1req/s. Keďže boli všetky požiadavky zavolané

```
http {
 limit_req_zone $binary_remote_addr \
   zone=myratelimit:10m rate=1r/s;
 ...

 server {
    ...
    location / {
      limit_req zone=myratelimit;
      ...
    }
  }
}
```

Kód 12. Súbor *nginx.conf* s konfiguráciou zapnutého "rate-limiting"-u

```
→  ~ siege -v -r 2 -c 5 http://localhost:8081

** SIEGE 4.1.2
** Preparing 5 concurrent users for battle.
The server is now under siege...
HTTP/1.1 503     0.00 secs:     197 bytes ==> GET  /
HTTP/1.1 503     0.00 secs:     197 bytes ==> GET  /
HTTP/1.1 503     0.00 secs:     197 bytes ==> GET  /
HTTP/1.1 503     0.00 secs:     197 bytes ==> GET  /
HTTP/1.1 200     0.01 secs:      21 bytes ==> GET  /
HTTP/1.1 503     0.01 secs:     197 bytes ==> GET  /
HTTP/1.1 503     0.01 secs:     197 bytes ==> GET  /
HTTP/1.1 503     0.01 secs:     197 bytes ==> GET  /
HTTP/1.1 503     0.01 secs:     197 bytes ==> GET  /
HTTP/1.1 503     0.00 secs:     197 bytes ==> GET  /

Transactions:                      1 hits
Availability:                  10.00 %
```

Obr. 7. Priebeh testu nad systémom so zapnutým "rate-limiting"-om. Po dosiahnutí "rate-limit"-u sú ostatné požiadavky odmietnuté.

paralelne, pred prijatím ostatných požiadaviek ešte neubehla 1 sekunda a NGINX ich odmietol. Úspešne sme tak ochránili webový server pred útokom typu DOS/DDoS. Nevýhodou tohto riešenia je však zníženie dostupnosti služby na 10%.

Práve zníženie dostupnosti je v praxi javom, ktorý je potrebné v čo najvyššej miere eliminovať. Pre tento účel NGINX podporuje nastavenie *burst*. Nastavenie vykonáme v súbore *nginx.conf* (Kód 13).

Nastavením parametra *burst* na hodnotu *N* NGINX vytvorí FIFO "queue" o veľkosti *N*, do ktorej putujú požiadavky po dosiahnutí "rate-limit"-u. Požiadavky, ktoré sa zmestia do "queue" teda nie sú odmietnuté, ale obslúžené neskôr, rešpektujúc nastavený "rate-limit".

Obr. 8 znázorňuje správanie NGINX servera pri nastavení parametra *burst* na hodnotu 5. Považujem za dôležité pripomenúť, že na poradí požiadaviek nezáleží, nakoľko boli volané paralelne. Prvá úspešná (modrá) požiadavka bola obslúžená ihneď s trvaním 0.0s. Nasledujúcich 5 úspešných (modrých) požiadaviek putovalo do queue, z ktorej boli postupne obslúžené v jedno-sekundových intervaloch tak, aby bol

```
http {
 ...
 server {
 ...
    location / {
      ...
      limit_req zone=myratelimit burst=5;
    }
  }
}
```

Kód 13. Súbor *nginx.conf* s konfiguráciou zapnutého "rate-limiting"-u a nastavenia *burst*

```
→  ~ siege -v -r 1 -c 10 http://localhost:8081
** SIEGE 4.1.2
** Preparing 10 concurrent users for battle.
The server is now under siege...
HTTP/1.1 503     0.00 secs:     197 bytes ==> GET  /
HTTP/1.1 503     0.00 secs:     197 bytes ==> GET  /
HTTP/1.1 503     0.00 secs:     197 bytes ==> GET  /
HTTP/1.1 503     0.00 secs:     197 bytes ==> GET  /
HTTP/1.1 200     0.00 secs:      21 bytes ==> GET  /
HTTP/1.1 200     1.01 secs:      21 bytes ==> GET  /
HTTP/1.1 200     2.00 secs:      21 bytes ==> GET  /
HTTP/1.1 200     3.01 secs:      21 bytes ==> GET  /
HTTP/1.1 200     4.00 secs:      21 bytes ==> GET  /
HTTP/1.1 200     5.00 secs:      21 bytes ==> GET  /

Transactions:                      6 hits
Availability:                  60.00 %
Elapsed time:                   5.00 secs
```

Obr. 8. Priebeh testu nad systémom so zapnutým "rate-limiting"-om a parametrom *burst* s hodnotou 5.

dodržaný "rate-limit" 1req/s. Zvyšné požiadavky sa do queue nezmestili, a tak boli NGINX serverom ihneď odmietnuté. Pri zachovaní "rate-limitu" sme v tomto prípade zvýšili dostupnosť služby na 60% oproti pôvodným 10% a úspešne sme ochránili webový server pred útokom DDoS.

NGINX ďalej poskytuje možnosť vynútenia obsluhy prvých N požiadaviek z "queue" bez čakania. Tento jav sa nazýva 2-fázový "rate limiting" a je možné ho nastaviť v súbore *nginx.conf* pomocou parametra *delay* (Kód 14).

Na Obr. 9 vidíme priebeh experimentu po nastavení parametra *burst* na hodnotu 5 a *delay* na hodnotu 3. Prvá úspešná požiadavka bola obslúžená v rámci "rate-limit"-u 1req/s. Nasledujúcich 5 požiadaviek bolo presunutých do "queue". Prvé 3 požiadavky z "queue" boli obslúžené okamžite(ich latencia je 0.01s). Zvyšné 2 požiadavky boli obslúžené tak, aby bol dodržaný nastavený "rate-limit". Ich latencie sú preto 1.01s a 2.01s. Zvyšné požiadavky sa do "queue" nezmestili, a tak ich server NGINX odmietol. Oproti predchádzajúcej konfigurácii sme zachovali dostupnosť na úrovni 60%, ale znížili sme celkový čas obslúženia všetkých požiadaviek z 5s na 2.01s.

```
http {
...
 server {
    ...
    location / {
       ...
       limit_req zone=myratelimit \
         burst=5 delay=3;
    }
  }
}
```

Kód 14. Súbor *nginx.conf* s konfiguráciou zapnutého "rate-limiting"-u s parametrami *burst* a *delay*

```
➜  ~ siege -v -r 1 -c 10 http://localhost:8081
** SIEGE 4.1.2
** Preparing 10 concurrent users for battle.
The server is now under siege...
HTTP/1.1 503     0.01 secs:     197 bytes ==> GET  /
HTTP/1.1 503     0.01 secs:     197 bytes ==> GET  /
HTTP/1.1 503     0.01 secs:     197 bytes ==> GET  /
HTTP/1.1 503     0.01 secs:     197 bytes ==> GET  /
HTTP/1.1 200     0.01 secs:      21 bytes ==> GET  /
HTTP/1.1 200     0.01 secs:      21 bytes ==> GET  /
HTTP/1.1 200     0.01 secs:      21 bytes ==> GET  /
HTTP/1.1 200     0.01 secs:      21 bytes ==> GET  /
HTTP/1.1 200     1.01 secs:      21 bytes ==> GET  /
HTTP/1.1 200     2.01 secs:      21 bytes ==> GET  /

Transactions:                      6 hits
Availability:                  60.00 %
Elapsed time:                   2.01 secs
```

Obr. 9. Priebeh testu nad systémom so zapnutým "rate-limiting"-om a parametrami *burst*=5 a *delay*=3

C. NGINX Caching

Podkapitola I-D3 je venovaná účinnosti "caching"-u pred zahltením servera. Na úrovni NGINX servera je možné v "cache" pamäti ukladať odpovede na všetky typy HTTP požiadaviek. "Caching" je opäť nastaviteľný v súbore *nginx.conf* (Kód 15).

V parametri *proxy_cache_path* nastavíme adresár */cache* ako cieľový adresár pre ukladanie zapamätaných odpovedí s využitím dvoj-úrovňového adresovania. Nastavenie je pomenované *my_cache* a pre "cache" pamäť je vyhradených 10mb diskového priestoru. NGINX však po prekročení obmedzenia pokračuje alokáciou ďalšieho priestoru a tak je vhodné nastaviť aj horné ohraničenie na 1Gb aby sme predišli zahlteniu diskového priestoru NGINX servera. Položky v pamäti, ktoré neboli za posledných 60min pristúpené sú z "cache" pamäte automaticky odstránené. Nastavením *proxy_cache* zapneme vytvorenú "cache". NGINX tiež podporuje vynútenie vrátenia odpovede uloženej v "cache" pamäti v prípade, že server neodpovedá alebo odpovedá s 5XX status kódom. Týmto spôsobom predídeme zníženiu dostupnosti služby počas DOS/DDoS útoku.

```
http {
...
 proxy_cache_path /cache levels=1:2 \
   keys_zone=my_cache:10m max_size=1g\
   inactive=60m use_temp_path=off;
...
 server {
    ...
    location / {
       proxy_cache my_cache;
       proxy_cache_use_stale error \
         timeout http_500 http_502 \
         http_503 http_504;
       ...
    }
  }
}
```

Kód 15. Súbor *nginx.conf* s konfiguráciou zapnutého "caching"-u

```
@app.route('/')
def hello_world():
  ...
  resp.headers['Cache-Control'] \
    = 'max-age=604800'
  return resp
```

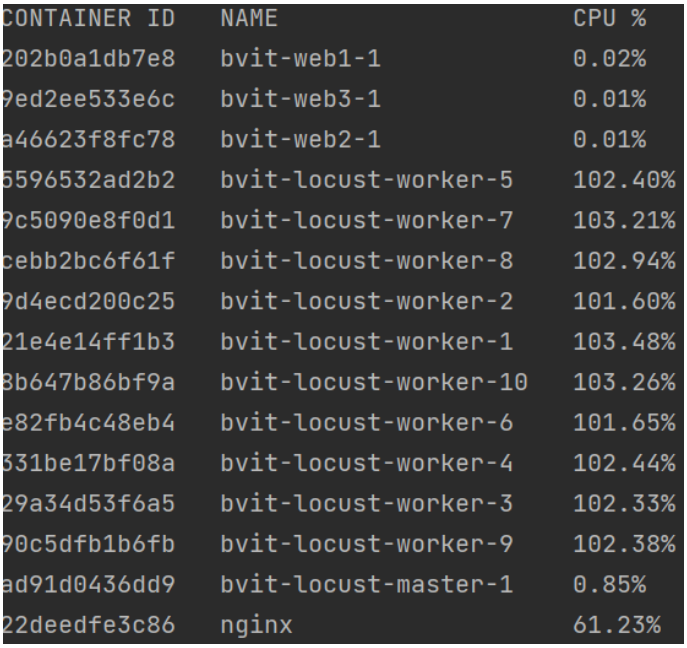
Kód 16. Zdrojový kód webového servera po pridaní HTTP hlavičky *Cache-Control*

NGINX automaticky do "cache" pamäte ukladá odpovede, ktoré obsahujú HTTP hlavičky *Expires* alebo *Cache-Control*. Zdrojový kód servera je teda potrebné úraviť tak, aby aby odpovede obsahovali jednu z týchto hlavičiek (Kód 16).

Obr. 2 znázorňuje priebeh testu nad 1 replikou servera pri vypnutom "cache"-ingu. V teste dosiahol server priepustnosť 3369 požiadaviek za sekundu pri 4% chybovosti. Na Obr. 10 vidíme, že pri zapnutom "caching"-u server dosiahol priepustnosť až 11336 požiadaviek za sekundu.

Pri analýze vyťaženia CPU jednotlivými Docker kontajnermi na Obr. 11 vidíme, že webové servery *web1..3* nie sú vôbec vyťažené, pretože neobsluhujú žiadne požiadavky.

Obr. 10. Priebeh testu nad serverom so zapnutým "caching"-om.



CONTAINER ID	NAME	CPU %
202b0a1db7e8	bvit-web1-1	0.02%
9ed2ee533e6c	bvit-web3-1	0.01%
a46623f8fc78	bvit-web2-1	0.01%
5596532ad2b2	bvit-locust-worker-5	102.40%
9c5090e8f0d1	bvit-locust-worker-7	103.21%
cebb2bc6f61f	bvit-locust-worker-8	102.94%
9d4ecd200c25	bvit-locust-worker-2	101.60%
21e4e14ff1b3	bvit-locust-worker-1	103.48%
8b647b86bf9a	bvit-locust-worker-10	103.26%
e82fb4c48eb4	bvit-locust-worker-6	101.65%
331be17bf08a	bvit-locust-worker-4	102.44%
29a34d53f6a5	bvit-locust-worker-3	102.33%
90c5dfb1b6fb	bvit-locust-worker-9	102.38%
ad91d0436dd9	bvit-locust-master-1	0.85%
22deedfe3c86	nginx	61.23%

Obr. 11. Vyťaženie CPU jednotlivými Docker kontajnermi.

Worker procesy *locust-worker1..10* sú vyťažené na 100%, čo znamená, že sme dosiahli limit systému pri generovaní HTTP požiadaviek. Na generovanie väčšieho množstva by sme potrebovali viac CPU jadier, umožňujúcich spustenie viac *worker* procesov. Napriek tomu je NGINX vyťažený len na 61%. Je teda pravdepodobné, že server dokáže poskytnúť ešte vyššiu priepustnosť, ktorú však v našich podmienkach nedokážeme nasimulovať. S využítim "caching"-u sme ochránili webový server pred útokom DOS/DDoS so zachovaním vysokej dostupnosti a priepustnosti.

VI. Zhrnutie

V tejto práci som sa venoval opisu DDoS útokov, ako aj rôznych techník ich detekcie a prevencie. Časový plán práce som splnil s výnimkou miernej odchýlky pri tvorbe dokumentácie v posledných týždňoch semestra. V experimentálnej časti práce som sa venoval ochrane webového servera pred útokmi typu DDoS. S využitím servera NGINX som demonštroval účinnosť "load-balancing"-u, "rate-limiting"-u a "caching"-u pri ochrane proti útokom typu DDoS. Simulácie prebiehali v testovacom prostredí, ktoré v čo najvyššej miere pripomína reálne produkčné prostredie (produkčné servery a konfigurácie, kontajnerizácia). Napriek úspešnej demonštrácii účinnosti týchto metód považujem za dôležité spomenúť, že žiadna z týchto metód neposkytne úplnú ochranu pred DDoS útokom. "Load-balancing" a "caching" sú účinné len v prípade, že útočník nedokáže generovať väčší objem požiadaviek, než je priepustnosť systému po aplikácii týchto mechanizmov. Priepustnosť však vďaka týmto metódam dokážeme signifikantne zvýšiť a môžu tak poskytnúť istú úroveň ochrany. "Rate-limiting" za cenu mierneho zníženia dostupnosti tiež pomôže predísť zahlteniu webového servera. Na účinnú ochranu proti DDoS útokom neexistuje univerzálna metóda. Kombináciou viacerých metód však môžeme dopad týchto útokov v čo najvyššej možnej miere eliminovať.

Použité zdroje a literatúra

[1] *10 Best Practices to Prevent DDoS Attacks*. URL: https://securityscorecard.com/blog/best-practices-to-prevent-ddos-attacks.

[2] *10 Best Practices to Prevent DDoS Attacks*. URL: https://securityscorecard.com/blog/best-practices-to-prevent-ddos-attacks.

[3] Mohammed N Alenezi, Martin J Reed, and Mohammed A Alhomidi. "Selective Windowed Rate Limiting for DoS Mitigation". In: *2017 9th IEEE-GCC Conference and Exhibition (GCCCE)*. IEEE. 2017, pp. 1–6.

[4] Akashdeep Bhardwaj et al. "Three tier network architecture to mitigate ddos attacks on hybrid cloud environments". In: *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies*. 2016, pp. 1–7.

[5] Catalin Cimpanu. *AWS said it mitigated a 2.3 Tbps DDoS attack, the largest ever*. URL: https://www.zdnet.com/article/aws-said-it-mitigated-a-2-3-tbps-ddos-attack-the-largest-ever/.

[6] Catalin Cimpanu. *Google says it mitigated a 2.54 Tbps DDoS attack in 2017, largest known to date*. URL: https://www.zdnet.com/article/google-says-it-mitigated-a-2-54-tbps-ddos-attack-in-2017-largest-known-to-date/.

[7] *Cloudflare: What is WAF?* URL: https://www.cloudflare.com/en-gb/learning/ddos/glossary/web-application-firewall-waf/.

[8] G Dayanandam et al. "DDoS Attacks—Analysis and Prevention". In: *Innovations in Computer Science and Engineering*. Ed. by H S Saini et al. Singapore: Springer Singapore, 2019, pp. 1–10. ISBN: 978-981-10-8201-6.

[9] Rashmi V. Deshmukh and Kailas K. Devadkar. "Understanding DDoS attack & its effect in cloud environment". In: *Procedia Computer Science* 49.1 (2015), pp. 202–210. ISSN: 18770509. DOI: 10.1016/j.procs.2015.04.245. URL: http://dx.doi.org/10.1016/j.procs.2015.04.245.

[10] Sven Dietrich and Neil Long. "Analyzing distributed denial of service tools: The shaft case". In: *14th Systems Administration Conference (LISA 2000)*. 2000.

[11] *Famous DDoS attacks - Cloudflare*. URL: https://www.cloudflare.com/en-gb/learning/ddos/famous-ddos-attacks/.

[12] *Flask webpage*. 2022. URL: https://flask.palletsprojects.com/en/2.1.x/.

[13] Yossi Gilad et al. "CDN-on-Demand: An affordable DDoS Defense via Untrusted Clouds." In: *NDSS*. 2016.

[14] *Gunicorn website*. URL: https://gunicorn.org/.

[15] Sam Kottler. *Github - February 28th DDoS Incident Report*. URL: https://github.blog/2018-03-01-ddos-incident-report/.

[16] *Limiting Access to Proxied HTTP Resources*. URL: https://docs.nginx.com/nginx/admin-guide/security-controls/controlling-access-proxied-http/.

[17] *Locust.io - An open source load testing tool.* URL: https://locust.io/.

[18] Jelena Mirkovic and Peter Reiher. "A taxonomy of DDoS attack and DDoS defense mechanisms". In: *ACM SIGCOMM Computer Communication Review* 34.2 (2004), pp. 39–53.

[19] Jarmo Mölsä. "Effectiveness of rate-limiting in mitigating flooding DOS attacks." In: *Communications, Internet, and Information Technology*. Citeseer. 2004, pp. 155–160.

[20] *Most Important Server Monitoring Metrics to Consider*. URL: https://www.dnsstuff.com/most-important-server-monitoring-metrics-to-consider.

[21] Baljit Singh Saini and Anju Bala. "A review of bot protection using CAPTCHA for web security". In: *IOSR Journal of Computer Engineering* 8.6 (2013), pp. 36–42.

[22] *Siege tool repository*. URL: https://github.com/JoeDog/siege.

[23] Ved Prakash Singh and Preet Pal. "Survey of different types of CAPTCHA". In: *International Journal of Computer Science and Information Technologies* 5.2 (2014), pp. 2242–2245.

[24] *Snort website*. URL: https://www.snort.org/.

[25] *Using NGINX as HTTP Loadbalancer*. URL: http://nginx.org/en/docs/http/load_balancing.html.

[26] Jia Wang. "A survey of web caching schemes for the internet". In: *ACM SIGCOMM Computer Communication Review* 29.5 (1999), pp. 36–46.

[27] *WSGI specification*. URL: https://wsgi.readthedocs.io/en/latest/.