ŠOLSKI CENTER CELJE
Srednja šola za kemijo, elektrotehniko in računalništvo

DDoS in keylogger

Računalništvo

Avtor raziskovalne naloge: Benjamin Planko

Mentor raziskovalne naloge: Tilen Sirk

Mestna občina Celje, Mladi za Celje

Celje, 2026

IZJAVA

Mentor/-ica TILEN SIRK v skladu z 21. členom Pravilnika o organizaciji mladinske raziskovalne dejavnosti »Mladi za Celje« Mestne občine Celje, izjavljam, da je v raziskovalni nalogi z naslovom DDOS IN KEYLOGGER, katere avtor/-ica je BENJAMIN PLANKO:

- pri raziskovanju uporabljeno gradivo navedeno v seznamu uporabljene literature,
- da je za objavo fotografij v nalogi pridobljeno avtorjevo dovoljenje in je hranjeno v šolskem arhivu,
- da sme Osrednja knjižnica Celje objaviti raziskovalno nalogo v polnem besedilu na knjižničnih portalih z navedbo, da je raziskovalna naloga nastala v okviru projekta Mladi za Celje,
- da je raziskovalno nalogo dovoljeno uporabiti za izobraževalne in raziskovalne namene s povzemanjem misli, idej, konceptov oziroma besedil iz naloge ob upoštevanju avtorstva in korektnem citiranju,
- da smo pri pripravi raziskovalne naloge upoštevali etična načela in smernice v skladu z veljavnimi pravnimi akti raziskovalnega področja,
- da smo seznanjeni z razpisni pogoji projekta Mladi za Celje.

Celje, 9. 3. 2026

ŠOLSKI CENTER CELJE
SREDNJA ŠOLA ZA KEMIJO, ELEKTROTEHNIKO IN RAČUNALNIŠTVO
CELJE, Pot na Lavo 22

zig šole

Podpis avtorja-jev:

Benjamin Planko

Podpis mentorja – jev:

Sirk

Podpis odgovorne osebe:

Lazuk

POJASNILO

V skladu z 21. členom Pravilnika raziskovalne dejavnosti »Mladi za Celje« Mestne občine Celje je potrebno podpisano izjavo mentorja (-ice), avtorja (-rjev) in odgovorne osebe šole vključiti v končno verzijo raziskovalne naloge, dovoljenje za objavo avtorja (-ice) fotografskega gradiva, katerega ni avtor (-ica) raziskovalne naloge, pa hrani šola v svojem arhivu.

Zahvala

Zahvaljujem se vsem, ki so na kakršenkoli način pomagali pri izdelavi raziskovalne naloge. Z veseljem sem sprejel vsak nasvet in novo idejo, ter upošteval kritike, ki so mi pomagale k izboljšavi naloge.

Posebno zahvalo izrekam mentorju prof. Tilnu Sirku za ves vložen čas, trud in pomoč pri izdelavi raziskovalne naloge.

Uporabljene kratice in ključne besede

DDoS: ang. Distributed Denial-of-Service - kibernetski napad namenjen preobremenitvi strežnikov ali omrežja.

Keylogger: slo. Beležnik tipkanja - programska ali strojna oprema, ki beleži pritiske na tipkovnico in na ta način pridobi zasebne ali zaupne informacije.

Botnet: omrežje okuženih naprav, ki jih napadalec uporabi za izvedbo različnih kibernetskih napadov.

Omrežni promet: količina podatkov, ki se na določen časovni interval prenese preko omrežja med napravami.

UDP: ang. User Datagram Protocol - komunikacijski protokol, idealen za aplikacije, občutljive na čas. Deluje brez vzpostavitve predhodne povezave in zagotavlja dostavo paketa.

IoT: ang. Internet of Things - je omrežje, sestavljeno iz pametnih naprav, predmetov, vozil in aparatov, ki so med seboj povezane preko interneta.

TCP: ang. Transmission Control Protocol - je temeljni omrežni protokol, ki zagotavlja zanesljiv prenos podatkov med računalniki.

ICMP: ang. Internet Control Message Protocol – omrežni protokol uporabljen za pošiljanje diagnostičnih sporočil, obvestil o napakah in preverjanje povezave.

DNS: ang. Domain Name System - je imenik interneta, ki domenska imena pretvori v računalniku berljive IP naslove.

NTP: ang. Network Time Protocol - protokol, ki zagotavlja natančno sinhronizacijo ur računalnikov in omrežnih naprav.

HTTP: ang. HyperText Transfer Protocol - temeljni komunikacijski protokol za prenos informacij na svetovnem spletu.

POST: metoda HTTP protokola, ki od spletnega strežnika zahteva pošiljanje ali sprejemanje informacij.

Povzetek

DDoS napadi preobremenijo ciljni sistem z velikim številom zahtev iz okuženih naprav, zaradi česar storitve postanejo nedostopne uporabnikom. Predstavljene so različne vrste teh napadov (na osnovi prometa, protokolni, aplikacijski ter refleksijski oziroma amplifikacijski) in njihove značilnosti. Opisani so tudi načini zaščite, kot so uporaba DDoS mitigatorjev, omrežnih filtrov, CDN sistemov, preverjanje IP-naslovov in spremljanje prometa.

Drugi del naloge je raziskava keyloggerja, ki je programsko ali strojno orodje za beleženje tipkanja z namenom kraje občutljivih podatkov. Predstavljene so njihove vrste ter tveganja za posameznike in podjetja. Kot zaščita se priporočata uporaba posodobljene varnostne programske opreme ter previdnost pri nameščanju programov in odpiranju sumljivih vsebin.

V tretjem delu naloge je predstavljena simulacija DDoS napada na testno omrežje, s katero je prikazano, kako takšen napad vpliva na delovanje sistema. Simulacija omogoča opazovanje povečanja omrežnega prometa, obremenitve strežnika ter morebitne nedostopnosti storitev. Namen praktičnega dela je bolje razumeti potek napada.

Ključne besede: DDoS, keylogger, botnet, omrežni promet

Abstract

DDoS attacks overload a target system with a large number of requests from infected devices, causing services to become unavailable to users. Different types of these attacks are presented (traffic-based, protocol, application-layer, and reflection or amplification attacks) along with their characteristics. Protection methods are also described, such as the use of DDoS mitigators, network filters, CDN systems, IP address verification, and traffic monitoring.

The second part of the paper focuses on keyloggers, which are software or hardware tools used to record keystrokes in order to steal sensitive information. Their types and the risks they pose to individuals and organizations are discussed. As protection, the use of updated security software and caution when installing programs or opening suspicious content are recommended.

The third part of the paper presents a simulation of a DDoS attack on a test network, demonstrating how such an attack affects system performance. The simulation allows observation of increased network traffic, server load, and possible service unavailability. The purpose of the practical part is to better understand how the attack works.

Keywords: DDoS attack, keylogger, botnet, network traffic

Kazalo

1. Uvod ... 10
1.1 Hipoteze ... 10
1.1.1 Prva hipoteza ... 10
1.1.2 Druga hipoteza ... 10
1.1.3 Tretja hipoteza ... 10
2. Kaj sploh je DDoS napad ... 11
2.1 DDos Volume-Based Attack ... 11
2.2 DDoS napad na protokolni ravni ... 12
2.3 DDoS napad na aplikacijski ravni ... 13
2.4 Refleksijski DDoS napad ... 14
2.5 Amplifikacijski DDoS napad ... 15
2.6 Zaščita pred DDoS napadi ... 15
3. Keylogger ... 18
3.1 Programski keylogger ... 18
3.2 Strojni keylogger ... 18
3.3 Zaščita pred keyloggerjem ... 18
4. Simulacija DDoS napada ... 20
4.1 Programska koda za program namenjen obremenjevanju omrežja ... 20
4.2 Pregled programa ... 30
4.2.1 Dva načina delovanja ... 30
4.2.2 Splošni tok programa ... 31
4.2.3 Importi — uvožene knjižnice ... 31
4.2.4 Socket ... 32
4.2.5 Threading ... 32
4.2.6 Time ... 33
4.2.7 Argparse ... 34

4.2.8 Sys ... 34
4.2.9 Signal ... 35
4.2.10 struct ... 35
4.2.11 from statistics import mean ... 36
4.3 Globalne spremenljivke ... 36
4.4 Funkcija: signal_handler(signum, frame) ... 37
4.4.1 Namen in pomen ... 38
4.4.2 Argumenti ... 38
4.4.3 Kako se registrira ... 39
4.4.4 Podrobno delovanje — korak po korak ... 39
4.5 Funkcija: verify_server_reachable(...) ... 40
4.5.1 Namen in pomen ... 40
4.5.2 Argumenti ... 40
4.5.3 Podroben opis kode ... 41
4.5.4 Kdaj vrne True, kdaj False ... 44
4.6 Funkcija: run_server(bind_ip, port, buffer_size) ... 45
4.6.1 Namen in pomen ... 45
4.6.2 Argumenti ... 45
4.6.3 Podroben opis kode ... 46
4.7 Funkcija: run_client(...) ... 50
4.7.1 Namen in pomen ... 50
4.7.2 Argumenti ... 51
4.7.3 Zakaj 1472 bajtov? ... 52
4.7.4 Podroben opis kode ... 53
4.7.5 Primer izpisa odjemalca ... 56
4.8 Funkcija: main() ... 56
4.8.1 Namen in pomen ... 56

4.8.2 argparse — analiza ukazne vrstice ... 56
4.8.3 Vzajemno izključujoča skupina ... 57
4.8.4 Vsi argumenti ukazne vrstice ... 57
4.8.5 Validacija parametrov ... 59
4.8.6 Registracija signal handlerja ... 59
4.8.7 Zagon izbranega načina ... 60
4.8.8 Vstopna točka programa ... 60
4.9 Povzetek — pregled vseh funkcij ... 61
4.10 Celoten potek programa — od zagona do konca ... 62
5. Prikaz simulacij ... 64
6. Zaključek ... 70
6.1 Potrjevanje oziroma zavračanje hipotez ... 70
6.1.1 Prva hipoteza ... 70
6.1.2 Druga hipoteza ... 70
6.1.3 Tretja hipoteza ... 71
7. Viri ... 72

Kazalo slik

Slika 1 Prikaz obremenjenosti 64

Slika 2 Nadzor hitrosti omrežja 65

Slika 3 Povečanje odzivnega časa 66

Slika 4 Primerjava zakasnitve med različnimi omrežnimi točkami 67

Slika 5 Postopno povečevanje prometa 68

Slika 6 Hitro povečanje prometa oz. DDoS napad 69

Kazalo tabel

Tabela 1 Dva načina delovanja 30
Tabela 2 Globalne spremenljivke 37
Tabela 3 Funkcija Signal-handler argumenti 38
Tabela 4 Funkcija Verify_server_reachable argumenti 41
Tabela 5 Sestavni deli probe paketa 42
Tabela 6 Obravnavanje napak 45
Tabela 7 Funkcija Run_server argumenti 46
Tabela 8 Funkcija Run_client argumenti 52
Tabela 9 Velikost paketa 53
Tabela 10 Funkcija main() argumenti 59
Tabela 11 Povzetek funkcij 61
Tabela 12 Potek programa 63

1. Uvod

V tej seminarski nalogi bom podrobno pogledal, kako delujeta DDoS napad in keylogger. Kako jih lahko zaznamo in kako se lahko pred njimi zaščitimo. Napad z razpršenim zaviranjem storitve (DDoS) je vse pogostejši način napada in postaja vedno bolj sofisticiran, saj napadalci ustvarjajo vedno boljše metode in orodja za izvajanje teh napadov. Keylogger pa je programska oprema ali pa celo naprava, ki snema vnose uporabnika. Namenjen je predvsem za pridobivanje zasebnih ali zaupnih podatkov kot so gesla, šifre in poslovne skrivnosti, ki so kasneje uporabljene v zlorabah in kraji identitete. Keylogger lahko deluje skrito in je neopazen, kar ga naredi zelo nevarnega.

1.1 Hipoteze

1.1.1 Prva hipoteza

DDoS napadi predstavljajo veliko grožnjo za informacijske sisteme, saj lahko z velikim številom zahtev preobremenijo omrežje ali strežnik in s tem onemogočijo dostop legitimnim uporabnikom do storitve.

1.1.2 Druga hipoteza

Ustrezni varnostni mehanizmi, kot so filtriranje omrežnega prometa, sistemi za zaznavanje napadov in protivirusna zaščita, lahko bistveno zmanjšajo tveganje za uspešen DDoS napad ali okužbo s keyloggerjem.

1.1.3 Tretja hipoteza

Z večanjem uporabe interneta in digitalnih storitev se povečuje tudi število kibernetskih napadov, kot so DDoS napadi in uporaba keyloggerjev.

2. Kaj sploh je DDoS napad

DDoS (Distributed Denial of Service) je kibernetski napad, pri katerem je uporabljeno večje število kompromitiranih sistemov za napad na eno tarčo. Učinek tega napada je, da napadeni sistem ne more delovati normalno, saj ga povečan promet preko omrežja preobremeni z zahtevami ali pa izkoristijo vse njegove vire, kar pomeni, da legitimni uporabniki nimajo več dostopa do storitve. Poznamo več vrst DDoS napadov, kot so napad na osnovni prostornini, protokolni napad in napad na aplikacijski ravni. Število teh napadov se je v zadnjih letih krepko povečalo, saj so relativno enostavni za izvedbo, pred njimi pa ni veliko zaščite, še posebej ranljivi so manjši sistemi in omrežja (Bombal, 2025).

2.1 DDos Volume-Based Attack

Ta tip DDoS napada je namenjen napadu preko omrežja. Trači napada prepreči dostop do omrežja s preobremenitvijo, kar napravi onemogoči opravljanje njenih nalog in onemogoči dostop do storitve uporabnikom. Napad je sestavljen iz različnih korakov.

Prvi korak je infiltracija in kontrola številnih okuženih naprav, kot so telefoni, računalniki in celo naprave IoT. Napadalec za kontroliranje le-teh uporabi zlonamerno programsko opremo ali pa izkoristi znane ranljivosti sistemov. Te naprave nato sestavi v botmreže, ki jo bo kasneje ob napadu tudi uporabil.

Drugi korak je koordinacija dejanj vseh okuženih naprav, napadalec jih pripravi za napad tako, da bodo lahko hkrati pošiljali velike količine podatkov. V ta namen uporabi in vključi različne protokole kot so UDP, TCP in ICMP.

Tretji korak pa je sam napad pri katerem napadalec uporabi bot mrežo, da pošlje veliko količino podatkov iz različnih IP naslovov. To preobremeni ključne naprave v omrežju

predvsem usmerjevalnike, pametna stikala in strežnike. To pa povzroči izgubo paketov, povečan čas odgovora storitve in na koncu celo odpoved storitve.

Napadalec se potem sam odloči kako dolgo bo napad trajal, saj nadzoruje kdaj bo razbremenil omrežje. Napad je po navadi izveden tako, da povzroči čim večjo škodo organizaciji, kateri storitev pripada. V največji nevarnosti za tovrsten napad so predvsem manjša podjetja, ki nimajo omrežij, zmožnih delovanja z veliko količino podatkov in so zato veliko bolj občutljiva za tovrsten napad (Sosafe in CDNetworks, 2025).

2.2 DDoS napad na protokolni ravni

Napad na protokolni ravni ali protokolni DDoS napad deluje na podlagi ranljivosti v samih protokolih, ki so uporabljeni za potovanje podatkov po omrežju. Ta napad je veliko težje zaznati, saj je veliko bolj sofisticiran in izkoristi znane napake v omrežnih protokolih, ki jih uporabljamo za komunikacijo preko omrežja.

Tudi ta napad deluje po korakih. Prvi korak je, da se napadalec odloči kateri protokol bo uporabil oziroma izkoristil za napad. Možnosti so HTTP, HTTPS, DNS, SYN flood (izkoristi ranljivosti v TCP protokolu) in še mnogi drugi.

Drugi korak je, da napadalec generira veliko količino protokolnih zahtevkov, ki morajo biti pravilni in uporabiti pravilne lastnosti. Ti morajo vsebovati pravilne parametre za potovanje po omrežju, saj se te razlikujejo med različnimi protokoli.

Tretji korak pa je potem napad, napadalec sočasno pošlje veliko količino zahtev, ki preobremenijo zmožnosti ciljnega strežnika. Strežnik je prisiljen obdelati vse zahteve, kar pa pomeni, da se čakalni čas za uporabnike podaljša ali pa storitev celo odpove, ker je strežnik tako zelo preobremenjen.

Rezultat takega napada je upočasnitev ali prekinitev storitve, saj lahko strežnik začne zavračati prave zahteve uporabnikov zaradi preobremenitve. Velik problem tega napada pa je tudi možnost polne avtomatizacije, saj lahko napadalec nastavi intervale in čas napada, da ta naredi čim večjo škodo ponudniku storitve.

Protokolni DDoS napadi so veliko nevarnejši od Volume-based napadov, saj jih je veliko težje odkriti in preprečiti, ker uporabljajo dejanske omrežne protokole in ne bot mreže, kjer lahko samo preprečimo dostop IP-ju napadalca. Edini način za zaščito pred njimi je uporaba protokolnih filtrov in blokiranje neobičajno visokega števila protokolnih zahtev (Holdsworth in Kosinski, 2025; CDNetworks, 2025).

2.3 DDoS napad na aplikacijski ravni

DDoS napad na aplikacijski ravni cilja na izkoriščanje ranljivosti v samih aplikacijah, ki se uporabljajo za opravljanje neke storitve. Te napade je zelo težko odkriti in preprečiti, ker uporabljajo dejanske mehanizme aplikacij, s katerimi potem preobremenijo ciljni sistem.

Tudi ta napad je storjen v različnih korakih, prvi korak je izbira specifične aplikacije, ki jo bomo napadli. To so lahko spletne aplikacije, API strežniki ali druge aplikacije.

Drugi korak je ponovno generiranje zahtev, ki morajo biti veljavne in sledi pravilom izbrane aplikacije, To vključuje število HTTP zahtev, POST zahtev ali drugih vrst zahtev.

Ob samem napadu zopet pride do preobremenitve samih zmožnosti strežnika, ki ne more obravnavati tako velike količine zahtev, kar pa pomeni zopet podaljšan reakcijski čas storitve ali celo zrušenje. Zato so uporabljene ranljivosti, ki so najdene v aplikaciji, postavitvi strežnika na katerem se aplikacija nahaja ali pa napake pri

implementaciji aplikacije na strežniku. Pogost primer je napad na avtentikacijskem procesu, ki zlahka preobremeni strežnik, saj ne more hkrati preveriti tako velike količine prijav, kar pa po navadi pripelje do zavračanja prijav legitimnih uporabnikov.

Tudi pri tem napadu je trajanje odvisno od napadalca in njegove pripravljenosti. Ti napadi so največkrat uporabljeni ob največji obremenjenosti sistemov, da imajo čim večji učinek, tako na uporabnike, kot tudi na ponudnika storitve. Pred njimi se zaščitimo tako, da preverimo, če je strežnik pravilno povezan z aplikacijo, ter uporabimo sisteme za zaznavanje večjega prometa zahtev kot običajno, ki jih nato tudi omeji (Sosafe, 2025 in CDNetworks, 2025).

2.4 Refleksijski DDoS napad

Refleksijski napad deluje tako, da napadalec pošlje zahteve s ukradenim IP-naslovom, katerega sistem zazna kot da prihaja iz ciljanega omrežja. Ti zahtevki so usmerjani na tretjo stran, ki je največkrat strežnik, najpogostejša sta DNS in NTP strežnika. Ta stran oziroma strežnik na to zahtevo odgovori in jo pošlje nazaj v omrežje. Tu nastane problem, ko se v omrežju pojavi več enakih IP naslovov, kar pa povzroči preobremenitev v omrežju.

Tovrsten napad je pogosto uporabljen zaradi njegove učinkovitosti in tudi zato, ker je relativno enostaven za izvedbo. Napadalec lahko za dosego tega uporabi veliko različnih protokolov in strežnikov, ki so namenjeni refleksiji zahtev v omrežju, s čimer nato doseže preobremenitev tarčnega omrežja. Za takšen napad so najpogosteje uporabljeni DNS, NTP, SNMP in SSDP protokoli, ki imajo vsak drugačne a dobro poznane ranljivosti, ki jih napadalec lahko izkoristi (Jim Holdsworth in Matthew Kosinski, 2025).

2.5 Amplifikacijski DDoS napad

Amplifikacijski napad je podoben refleksijskemu napadu, vendar se osredotoča na povečanje količine prometa v omrežju. Napadalec pošlje manjšo količino zahtev v ciljano omrežje. Z IP-ja ciljnega omrežja, s katerim pošlje zahteve na strežnike, ki lahko pošiljajo večje število zahtev naenkrat. Primer je, da napadalec pošlje zahtevo na NTP strežnik, ki bo na to zahtevo odgovoril z veliko večjim paketom podatkov. Tako lahko napadalec hitro doseže preobremenitev omrežja saj se količina podatkov v njem hitro povečuje, saj NTP strežnik na zahtevo odgovori z od 50 do 700-krat večjim paketom podatkov od izvorne zahteve. Najpogosteje uporabljeni protokoli za amplifikacijske napade so NTP, Memcached, DNS in SNMP. Vsak od teh protokolov je drugačen od prejšnjega, vsi imajo še vedno dobro poznane slabosti, ki jih napadalci izkoriščajo (Holdsworth in Kosinski, 2025).

2.6 Zaščita pred DDoS napadi

Zaščita pred DDoS napadi je zelo zahteven in kompleksen problem, ki vključuje različne tehnike in rešitve. Pred DDoS napadi se lahko le delno zaščitimo, vendar vedno več podjetij združuje različne načine obrambe, da čim bolj zmanjšajo tveganje za napad.

Eden od načinov zaščite pred DDoS napadi je uporaba DDoS mitigatorjev. To je sistem, ki je specializiran za zmanjševanje učinkovitosti DDoS napadov. Deluje tako, da preusmerja promet skozi svoje strežnike, ki filtrirajo in absorbirajo nepravilne ali ponavljajoče se zahteve, preden dosežejo tarčo. Seveda je lahko preobremenjen tudi ta sistem, kar pomeni, da bo kasneje preobremenjeno tudi ciljno omrežje. Vendar ta način dobro zaustavlja manjše DDoS napade.

Naslednji način so omrežni filtri, z njimi si lahko pomagamo, ker zmanjšujejo količino nezaželenega prometa, ki je namenjen strežnikom. Ti filtri lahko zaznajo DDoS napad

in ga uspešno blokirajo, če ima napad dovolj veliko enakih zahtev ali pa enako zaporedje, ki ga filter zazna.

Potem imamo distribucijske sisteme, ki nam omogočajo, da razpršimo omrežni pretok. To dosežemo tako, da se delo porazdeli na več strežnikov v primeru velike obremenitve. Lep primer tega je CDN (Content Delivery Network) storitev, ki nam pomaga pri razpršitvi prometa po čim širšem omrežju.

Nato imamo preverjanje avtentičnosti IP naslovov. Ta način uvede poseben protokol, ki preverja ali so IP naslovi res avtentični ali ne, namenjen je predvsem obrambi pred refleksijskim in amplifikacijskim napadom. Primer takšnega protokola je BCP38.

Zelo pogosto je nadziranje in analiza omrežnega prometa, to pomeni, da stalno nadzorujemo omrežni promet, kar nam lahko pomaga pri zaznavanju nenavadnega skoka prometa na omrežju, ki so glavni pokazatelji DDoS napada. Analitična orodja pa nam pomagajo bolje razumeti, kako izgleda redni promet proti prometu ob DDoS napadu.

Velik vpliv na uspešno obrambe pred napadom ima tudi varnostna arhitektura sistema. Ta uporablja različne načine varnostne arhitekture, ki preusmerja promet na mikrostrežnike in kontejnerje, tako lahko povečamo odpornost na DDoS napade, saj smo povečali fleksibilnost in prilagodljivost omrežja glede na količino prometa.

Dobra ideja je tudi dodajanje rezervnih kapacitet na strežnikih in v omrežjih, ki so namenjene absorbiranju preobremenitev ob napadu, kar nam zagotovi, da lahko omrežje prenese večjo obremenitev, ki bi drugače onesposobila sisteme.

Seveda so v omrežja že vključeni zaščitni protokoli, ki jim lahko pomagamo tudi z optimalno konfiguracijo omrežja. Vendar za njihovo pravilno delovanje, jih je treba redno posodabljati, saj se v njih še vedno skrivajo ranljivosti, ki jih napadalec lahko zlorabi.

Najzanesljivejša pa je seveda uporaba več načinov hkrati, tako, da imamo pokritih čim več varnostnih lukenj in omrežje pripravljeno na morebiten napad, ter odboj le-tega (Holdsworth in Kosinski, 2025; Sosafe, 2025 in CDNetworks, 2025).

3. Keylogger

Keylogger je programska oprema ali strojna naprava, namenjena snemanju in shranjevanju vsakega pritiska na napravi uporabnika. Ti podatki so nato dostopni napadalcu, ki jih lahko uporabi za pridobivanje občutljivih informacij, kot so gesla, PIN kode, osebna sporočila, ali pa poslovne skrivnosti. Keylogger se pogosto uporablja v kibernetskih napadih, saj omogoča napadalcu pridobivanje občutljivih podatkov ne da bi ta potreboval neposreden dostop do računalnika. Ti podatki lahko povzročijo veliko škodo, tako na osebni ravni kot tudi za podjetja. Najpogosteje so uporabljeni za krajo identitete ali pa za izsiljevanje podjetij (Oxford, 2021 in Fortinet, 2025).

3.1 Programski keylogger

Programski keyloggerji so programi, ki se namestijo oziroma okužijo operacijski sistem računalnika in se nahajajo v ozadju tako, da jih uporabnik ne more najti oziroma so prikriti. Delujejo tako, da snemajo vsak pritisk tipkovnice, ki ga naredi uporabnik. Te podatke nato shranijo v skrito datoteko ali pa jih pošljejo direktno napadalcu preko omrežja (Heimdal, 2025 in Fortinet, 2025).

3.2 Strojni keylogger

Strojni keyloggerji so fizični vmesniki, ki se namesti na tipkovnico. Največkrat tako, da uporabnik sploh ne ve, da ima podvojeno tipkovnico oziroma je na njej še naprava, ki beleži vse vnose, ki jih on naredi. Te naprave se največkrat uporabijo na napravah za bančne kartice, kjer je potrebno vpisati PIN kodo. Ker so te fizične naprave se podatki največkrat zapisujejo v pomnilnik katerega mora napadalec fizično odstraniti, da lahko pride do podatkov (Heimdal, 2025 in Fortinet, 2025).

3.3 Zaščita pred keyloggerjem

Zaščita pred keyloggerjem je relativno enostavna. Za preprečitev njegove namestitve, je pomembno, da imamo posodobljena varnostna orodja, kot so antivirusi in varnostne

aplikacije, ki lahko zaznajo zlonamerno kodo ali pa okuženo datoteko. Poleg tega pa je pomembno tudi, da ne nameščamo neznanih aplikacij iz sumljivih strani, ter ne odpiramo prilog v sumljivih sporočilih, ki so najverjetneje phishing napad (Oxford, 2021 in Fortinet, 2025).

4. Simulacija DDoS napada

Kot praktični del naloge sem se odločil razviti program, namenjen postopnemu obremenjevanju strežnika z omrežnim prometom. Cilj tega programa je bil simulirati naraščajočo količino omrežnega prometa ter tako ustvariti nadzorovane pogoje, v katerih je mogoče opazovati odziv strežnika na povečano obremenitev. S takšnim pristopom sem lahko postopoma povečeval število zahtevkov in spremljal, kako se zmogljivost strežnika spreminja pri različnih stopnjah obremenitve. Poleg tega sem spremljal tudi vpliv povečane količine prometa na omrežje, na primer morebitno zasičenost povezav, izgubo paketov in upočasnitev komunikacije. Takšen eksperimentalni pristop mi je omogočil boljše razumevanje, kako se sistemi obnašajo pod visoko obremenitvijo ter kako lahko DDoS napadi vplivajo na delovanje strežniške infrastrukture in omrežja kot celote. Za prikaz delovanja programa pa sem uporabil aplikaciji Grafana in Wireshark.

4.1 Programska koda za program namenjen obremenjevanju omrežja

```
import socket
import threading
import time
import argparse
import sys
import signal
import struct
from statistics import mean


# Global flag za zaustavitev programa
shutdown_requested = threading.Event()


def signal_handler(signum, frame):
    """Ctrl + C, Za Zaustavitev"""
    print("\n[!] Shutdown requested, stopping...")
    shutdown_requested.set()
```

```python
def verify_server_reachable(target_ip: str, port: int, timeout: float = 2.0,
retries: int = 3) -> bool:
    """
    Z ECHO pogleda ali je strežnik dosegljiv.
    Če se strežnik odziva vrne TRUE, drugače vrne nedosegljivo.
    """
    probe_sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    probe_sock.settimeout(timeout)

    # Določi zgradbo za probe sock in izbere port
    probe_sock.bind(('0.0.0.0', 0))
    local_port = probe_sock.getsockname()[1]

    # Preveri nastavitev porta
    probe_marker = b'STRESSTEST_PROBE'
    timestamp = struct.pack('!d', time.time())
    probe_packet = timestamp + probe_marker + b'\x00' * (64 - len(timestamp) -
len(probe_marker))

    for attempt in range(1, retries + 1):
        try:
            print(f"[VERIFY] Probe attempt {attempt}/{retries} to
{target_ip}:{port}...")

            # Pošlje poziv na IP
            probe_sock.sendto(probe_packet, (target_ip, port))

            # Počaka na odgovor (strežnik vrne odgovor na poziv)
            data, addr = probe_sock.recvfrom(1024)

            #  Preveri če je pravi okvir, marker in timer
            if len(data) >= 8:
                recv_time = time.time()
```

```
                try:
                    send_time = struct.unpack('!d', data[:8])[0]
                    rtt = (recv_time - send_time) * 1000
                    print(f"[VERIFY] Server responded from {addr[0]}:{addr[1]}
(RTT: {rtt:.3f} ms)")
                except:
                    print(f"[VERIFY] Server responded from
{addr[0]}:{addr[1]}")

                probe_sock.close()
                return True

        except socket.timeout:
            print(f"[VERIFY] Timeout - no response")
            continue
        except Exception as e:
            print(f"[VERIFY] Error: {e}")
            continue

    probe_sock.close()
    return False


def run_server(bind_ip: str, port: int, buffer_size: int = 65535):
    """
    Nastavitve za postavitev serverja
    """
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    try:
        sock.bind((bind_ip, port))
    except OSError as e:
        print(f"[ERROR] Cannot bind to {bind_ip}:{port} - {e}")
        sys.exit(1)
```

```python
print(f"[SERVER] Listening on {bind_ip}:{port}")
print("[SERVER] Press Ctrl+C to stop")

packets_received = 0
bytes_received = 0
start_time = time.time()
last_report_time = start_time
latencies = []

# Določi čas za periodično zaustavitev skripte
sock.settimeout(0.5)

while not shutdown_requested.is_set():
    try:
        data, addr = sock.recvfrom(buffer_size)
        recv_time = time.time()

        # Preveri verification in izmeri measurement(Echo back to sender)
        try:
            sock.sendto(data, addr)
        except:
            pass  # Ignorira echo errors

        # Poveča dovoljeni čas ob preobremenitvi (prvih 8 Bajtov podvoji)
        if len(data) >= 8:
            try:
                send_time = struct.unpack('!d', data[:8])[0]
                latency = (recv_time - send_time) * 1000  # ms
                latencies.append(latency)
            except:
                pass

        packets_received += 1
        bytes_received += len(data)
```

```
        except socket.timeout:
            continue
        except Exception as e:
            if not shutdown_requested.is_set():
                print(f"[ERROR] {e}")

    # Prikaže zadnje prikaze v kodi
    duration = time.time() - start_time
    print(f"\n[SERVER] Statistics:")
    print(f"  Duration: {duration:.2f}s")
    print(f"  Packets received: {packets_received:,}")
    print(f"  Bytes received: {bytes_received:,} ({bytes_received/1e6:.2f}
MB)")
    print(f"  Packets/sec: {packets_received/duration:,.0f}" if duration > 0
else "  Packets/sec: N/A")
    print(f"  Throughput: {bytes_received*8/duration/1e6:.2f} Mbps" if
duration > 0 else "  Throughput: N/A")
    if latencies:
        print(f"  Avg latency: {mean(latencies):.3f} ms")
        print(f"  Min latency: {min(latencies):.3f} ms")
        print(f"  Max latency: {max(latencies):.3f} ms")

    sock.close()


def run_client(target_ip: str, port: int,
               start_pps: int = 100,
               max_pps: int = 1000000,
               step_pps: int = 1000,
               step_duration: float = 5.0,
               packet_size: int = 1472,
               skip_verify: bool = False):

    # Preveri dostopnost serverja
```

```
    if not skip_verify:
        print("[CLIENT] Verifying server connectivity...")
        if not verify_server_reachable(target_ip, port):
            print(f"\n[ERROR] Server {target_ip}:{port} is not responding.")
            print("[ERROR] Please check:")
            print("  - Server is running with: python3 stress_test.py --server
--port", port)
            print("  - No firewall blocking UDP port", port)
            print("  - Target IP address is correct")
            print("\nUse --skip-verify to bypass this check (not
recommended)")
            sys.exit(1)
        print("[CLIENT] Server verified - starting stress test\n")
    else:
        print("[CLIENT] Skipping connectivity verification (--skip-verify)\n")

    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # Pripravi prikaze z timestamp
    payload_data = b'X' * (packet_size - 8)

    print(f"[CLIENT] Target: {target_ip}:{port}")
    print(f"[CLIENT] Starting at {start_pps} pps, max {max_pps} pps")
    print(f"[CLIENT] Step: +{step_pps} pps every {step_duration}s")
    print(f"[CLIENT] Packet size: {packet_size} bytes")
    print("[CLIENT] Press Ctrl+C to stop\n")

    current_pps = start_pps
    total_packets = 0
    total_bytes = 0
    start_time = time.time()
    step_start_time = start_time

    while not shutdown_requested.is_set() and current_pps <= max_pps:
        # Kalkulator intervala
```

```
        interval = 1.0 / current_pps

        step_packets = 0
        step_bytes = 0
        step_start = time.time()

        # Pošilja po nastavljenem intervalu
        while (time.time() - step_start < step_duration and
               not shutdown_requested.is_set()):

            # Naredi pakete z najnovejšimi timestamp-i(using struct for float
encoding)
            timestamp = time.time()
            packet = struct.pack('!d', timestamp) + payload_data

            try:
                sock.sendto(packet, (target_ip, port))
                total_packets += 1
                step_packets += 1
                total_bytes += len(packet)
                step_bytes += len(packet)
            except Exception as e:
                print(f"\n[ERROR] Send failed: {e}")
                shutdown_requested.set()
                break

            # Natančna kontrola intervala
            next_send = time.time() + interval
            while time.time() < next_send and not shutdown_requested.is_set():
                if next_send - time.time() > 0.001:
                    time.sleep(0.0005)

        # Statistike ob intervalu
        step_elapsed = time.time() - step_start
        actual_pps = step_packets / step_elapsed if step_elapsed > 0 else 0
```

```
        print(f"[STEP] Target: {current_pps:,} pps | "
              f"Actual: {actual_pps:,.0f} pps | "
              f"Sent: {step_packets:,} pkts | "
              f"Rate: {step_bytes*8/step_elapsed/1e6:.2f} Mbps")

        # Naslednji paketek
        current_pps += step_pps

    # Zadnji prikaz
    duration = time.time() - start_time
    print(f"\n[CLIENT] Final Statistics:")
    print(f"  Duration: {duration:.2f}s")
    print(f"  Total packets: {total_packets:,}")
    print(f"  Total bytes: {total_bytes:,} ({total_bytes/1e6:.2f} MB)")
    print(f"  Average rate: {total_packets/duration:,.0f} pps" if duration > 0
else "  Average rate: N/A")
    print(f"  Average throughput: {total_bytes*8/duration/1e6:.2f} Mbps" if
duration > 0 else "  Average throughput: N/A")

    sock.close()


def main():
    parser = argparse.ArgumentParser(
        description='UDP Network Stress Test - Gradual Load Generator',
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog="""
Examples:
  # PC 1 - Start server:
  python3 stress_test.py --server --bind 0.0.0.0 --port 5000

  # PC 2 - Run client (with automatic verification):
  python3 stress_test.py --client --target 192.168.1.10 --port 5000

  # Skip verification (if server doesn't echo back):
```

```
  python3 stress_test.py --client --target 192.168.1.10 --skip-verify

  # Fast ramp-up test:
  python3 stress_test.py --client --target 192.168.1.10 --start-pps 1000
                          --max-pps 500000 --step-pps 5000 --step-time 3
        """
    )

    mode = parser.add_mutually_exclusive_group(required=True)
    mode.add_argument('-s', '--server', action='store_true',
                      help='Run in server mode (receive packets)')
    mode.add_argument('-c', '--client', action='store_true',
                      help='Run in client mode (send packets)')

    # Možnosti strežnika
    parser.add_argument('--bind', default='0.0.0.0',
                        help='IP to bind server (default: 0.0.0.0)')

    # Klient možnosti
    parser.add_argument('--target', '-t',
                        help='Target server IP (required for client)')
    parser.add_argument('--start-pps', type=int, default=100,
                        help='Starting packets/sec (default: 100)')
    parser.add_argument('--max-pps', type=int, default=1000000,
                        help='Maximum packets/sec (default: 1000000)')
    parser.add_argument('--step-pps', type=int, default=1000,
                        help='Packets/sec increase per step (default: 1000)')
    parser.add_argument('--step-time', type=float, default=5.0,
                        help='Seconds per step (default: 5)')
    parser.add_argument('--packet-size', type=int, default=1472,
                        help='UDP payload size in bytes (default: 1472, max
65507)')
    parser.add_argument('--skip-verify', action='store_true',
                        help='Skip server connectivity verification (not
recommended)')
```

```python
    # Common možnosti
    parser.add_argument('-p', '--port', type=int, default=5000,
                        help='UDP port (default: 5000)')

    args = parser.parse_args()

    # Preveri argumente
    if args.client and not args.target:
        parser.error("--target is required when using --client")

    if not (64 <= args.packet_size <= 65507):
        parser.error("--packet-size must be between 64 and 65507")

    # Naloži signal handler
    signal.signal(signal.SIGINT, signal_handler)

    # Požene izbrani mode
    if args.server:
        run_server(args.bind, args.port)
    else:
        run_client(
            args.target,
            args.port,
            start_pps=args.start_pps,
            max_pps=args.max_pps,
            step_pps=args.step_pps,
            step_duration=args.step_time,
            packet_size=args.packet_size,
            skip_verify=args.skip_verify
        )


if __name__ == '__main__':
    main()
```

4.2 Pregled programa

UDP Testni obremenjevalnik omrežja je orodje za testiranje zmogljivosti omrežnih stikal (switch) in usmerjevalnikov. Deluje tako, da postopoma povečuje obremenitev omrežja in s tem ugotavlja, pri kateri hitrosti prenos paketov postane nezanesljiv.

Program temelji na protokolu UDP, ki je — za razliko od TCP — brez vzpostavitve povezave in brez potrditve dostave. To ga naredi idealnega za stresno testiranje, saj ne ustvarja dodatnega prometa za potrjevanje oddanih oziroma sprejetih paketov.

4.2.1 Dva načina delovanja

Način	Opis	Kako zagnati
Strežnik (server)	Sprejema pakete, jih odzvanja nazaj in meri statistike	python3 stress_test.py --server --port 5000
Odjemalec (client)	Pošilja pakete z naraščajočo hitrostjo in meri propustnost	python3 stress_test.py -- client -- target 192.168.1.10

Tabela 1 Dva načina delovanja

4.2.2 Splošni tok programa

1. Uporabnik zažene program z argumenti iz ukazne vrstice.

2. main() prebere argumente, preveri njihovo veljavnost in registrira Ctrl+C handler.

3. Glede na izbrani način se zažene run_server() ali run_client().

4. Odjemalec najprej pokliče verify_server_reachable() — preveri, da strežnik sploh posluša.

5. Odjemalec začne pošiljati pakete z naraščajočo hitrostjo v korakih.

6. Strežnik sprejema pakete, jih odzvanja nazaj in beleži statistike.

7. Ob pritisku Ctrl+C signal_handler varno ustavi oba.

8. Ob koncu se izpišejo podrobne statistike (paketi, bajti, Mbps, latenca).

4.2.3 Importi — uvožene knjižnice

Na začetku vsake Python datoteke so importi — ukazi, ki uvozijo potrebne knjižnice. Vsaka knjižnica doda programu določene zmožnosti. Brez importov bi morali vse pisati sami od začetka.

4.2.4 Socket

import socket

Standardna Python knjižnica za omrežno komunikacijo. Omogoča ustvarjanje, odpiranje in zapiranje omrežnih vtičnic (socket). Vtičnica je programski objekt, ki predstavlja eno stran omrežne povezave.

V tem programu se uporablja za:

- Ustvarjanje UDP vtičnic: socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
- Pošiljanje paketov: sock.sendto(data, (ip, port))
- Sprejemanje paketov: sock.recvfrom(buffer_size)
- Vezavo strežnika na vrata: sock.bind((ip, port))

4.2.5 Threading

import threading

Knjižnica za vzporedno izvajanje kode z nitmi (threads). Nit je lahek proces, ki se izvaja znotraj istega programa.

V tem programu se threading ne uporablja za vzporedne niti, temveč za poseben objekt threading.Event(), ki deluje kot skupna zastavica (flag) med različnimi deli programa.

shutdown_requested = threading.Event()

Event ima dve stanji:

- shutdown_requested.is_set() → True = zaustavi se
- shutdown_requested.set() → nastavi na True
- shutdown_requested.clear() → ponastavi na False

4.2.6 Time

import time

Knjižnica za delo s časom. Je ključna za dve stvari:

1. Merjenje latence: time.time() vrne trenutni čas v sekundah (npr. 1704067200.123456). Z odštevanjem dveh časov dobimo zamudo.

2. Kontrolo hitrosti: time.sleep(sekunde) začasno ustavi program. Busy-wait zanka pa čaka brez spanja za višjo natančnost.

```
# Primer merjenja časa:

start = time.time()

# ... operacija ...

elapsed = time.time() - start  # čas v sekundah
```

4.2.7 Argparse

import argparse

Knjižnica za razčlenjevanje argumentov iz ukazne vrstice. Namesto da bi ročno brali sys.argv, argparse samodejno razčleni, validira in dokumentira vse argumente.

Primer: ko uporabnik vpiše:

python3 stress_test.py --client --target 192.168.1.10 --port 5000 --max-pps 100000

argparse to razčleni v objekt z atributi: args.client=True, args.target='192.168.1.10', args.port=5000 itd.

4.2.8 Sys

import sys

Dostop do sistemskih funkcij Python interpreter-ja. V tem programu se uporablja samo za:

sys.exit(1)

Ta ukaz takoj zaustavi program in vrne izhodno kodo 1 (napaka) operacijskemu sistemu. Izhodni koda 0 pomeni uspeh, karkoli drugega pa napako.

4.2.9 Signal

import signal

Knjižnica za upravljanje s sistemskimi signali. Signali so sporočila, ki jih operacijski sistem pošlje procesom — npr. ko pritisnemo Ctrl+C, OS pošlje signal SIGINT (Signal Interrupt) programu.

Brez signal knjižnice bi Ctrl+C takoj in brutalno ubil program. Z njo pa lahko ujamemo signal in se lepo poslovimo:

signal.signal(signal.SIGINT, signal_handler)

To pove Pythonu: ko pride SIGINT (Ctrl+C), pokliči funkcijo signal_handler namesto privzetega obnašanja (takojšnje zaustavitve).

4.2.10 struct

import struct

Knjižnica za pakiranje in razpakiranje binarnih podatkov. Omogoča pretvorbo Python vrednosti (int, float, string) v surove bajte in nazaj — kar je potrebno za vključitev časovnega žiga v vsak paket.

Format '!d' pomeni:

- ! → big-endian byte order (omrežni standard)
- d → double (64-bitno decimalno število, 8 bajtov)

```
# Pakiranje (čas → bajti):
struct.pack('!d', time.time())   # vrne 8 bajtov
# Razpakiranje (bajti → čas):
struct.unpack('!d', data[:8])[0] # vzame prve 8 bajtov, vrne float
```

4.2.11 from statistics import mean

Iz standardne knjižnice statistics uvozi samo funkcijo mean() za izračun aritmetičnega povprečja. Uporablja se za povprečno latenco:

```
mean(latencies)  # npr. mean([1.2, 0.8, 1.5]) → 1.167
```

Namesto 'import statistics' in klicanja statistics.mean() se uvozi samo ta ena funkcija — bolj elegantno in berljivo.

4.3 Globalne spremenljivke

```
shutdown_requested = threading.Event()
```

Ta vrstica je napisana na vrhu datoteke, izven vseh funkcij — je globalna spremenljivka. To pomeni, da je dostopna iz vseh funkcij v datoteki kot argument.

threading.Event() ustvari poseben objekt, ki deluje kot on/off stikalo:

Metoda	Opis	Primer
.set()	Nastavi zastavico na True (zaustavi)	shutdown_requested.set()
.is_set()	Preveri, ali je argument True	while not shutdown_requested.is_set():
.clear()	Ponastavi argument na False	shutdown_requested.clear()
.wait(timeout)	Čaka dokler ni nastavljena	shutdown_requested.wait(1.0)

Tabela 2 Globalne spremenljivke

4.4 Funkcija: signal_handler(signum, frame)

```
def signal_handler(signum, frame):
    print("\n[!] Shutdown requested, stopping...")
    shutdown_requested.set()
```

4.4.1 Namen in pomen

Ko uporabnik pritisne Ctrl+C, operacijski sistem pošlje signal SIGINT procesu. Brez posebne obravnave bi Python takoj prekinil izvajanje — vtičnice bi ostale odprte, statistike ne bi bile izpisane.

signal_handler to prepreči: nastavi zastavico shutdown_requested, vse zanke jo opazijo in se same varno zaključijo, nato pa se izpišejo končne statistike.

4.4.2 Argumenti

Argument	Tip	Kdo ga poda	Opis
signum	int	Python samodejno	Številka prejetega signala. SIGINT = 2, SIGTERM = 15.
frame	frame obj.	Python samodejno	Objekt, ki opisuje stanje programa v trenutku signala (katera vrstica se je izvajala, lokalne spremenljivke itd.).

Tabela 3 Funkcija Signal-handler argumenti

4.4.3 Kako se registrira

V funkciji main() je naslednja vrstica:

```
signal.signal(signal.SIGINT, signal_handler)
```

To pove Pythonu: namesto privzetega obnašanja ob Ctrl+C pokliči signal_handler. Brez te vrstice bi signal_handler sploh nikoli ne bil klican.

4.4.4 Podrobno delovanje — korak po korak

1. Uporabnik pritisne Ctrl+C.

2. OS pošlje signal SIGINT procesu Python.

3. Python pokliče registrirani signal_handler(2, <frame>).

4. Izpiše se: "[!] Shutdown requested, stopping..."

5. shutdown_requested.set() nastavi zastavico na True.

6. Vse zanke, ki preverjajo 'while not shutdown_requested.is_set()', se začnejo zaključevati.

7. run_server() ali run_client() izpišeta končne statistike.

8. Program se varno zaključi.

4.5 Funkcija: verify_server_reachable(...)

def verify_server_reachable(target_ip: str, port: int, timeout: float = 2.0, retries: int = 3) -> bool:

4.5.1 Namen in pomen

Preden odjemalec začne pošiljati na tisoče paketov na sekundo, mora vedeti, ali strežnik sploh posluša. Ta funkcija pošlje en testni (probe) paket in čaka na echo odgovor. Če strežnik odzvanja, je vse v redu. Če ne, program sporoči napako in se zaustavi.

Brez te preveritve bi odjemalec nemara pošiljal pakete v prazno — na napačen IP, na izklopljeni strežnik ali skozi požarni zid.

4.5.2 Argumenti

Argument	Tip	Privzeto	Opis
target_ip	str	—	IP naslov strežnika (npr. '192.168.1.10')
port	int	—	Vrata na katera pošljemo probe paket

Argument	Tip	Privzeto	Opis
timeout	float	2.0	Koliko sekund čakamo na vsak odgovor. Če v tem času ni odgovora → timeout izjema.
retries	int	3	Koliko poskusov naredimo skupaj preden ugotovimo, da strežnik ni dosegljiv.

Tabela 4 Funkcija Verify_server_reachable argumenti

Vrne: True če strežnik odgovori, False če ni odgovora po vseh poskusih.

4.5.3 Podroben opis kode

Korak 1: Ustvaritev probe vtičnice

```
probe_sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

probe_sock.settimeout(timeout)

probe_sock.bind(('0.0.0.0', 0))

local_port = probe_sock.getsockname()[1]
```

Ustvari se začasna UDP vtičnica samo za to preverjanje. settimeout(2.0) pomeni: če v 2 sekundah ni odgovora, sproži socket.timeout izjemo. bind(('0.0.0.0', 0)) prosi OS, da

sam izbere prosta vrata — parameter 0 pomeni 'izberi kar ti misliš da je primerno'. getsockname() vrne IP in vrata, ki jih je OS izbral.

Korak 2: Sestavljanje probe paketa

```
probe_marker = b'STRESSTEST_PROBE'

timestamp = struct.pack('!d', time.time())

probe_packet = timestamp + probe_marker + b'\x00' * (64 - len(timestamp) - len(probe_marker))
```

Probe paket je dolg natančno 64 bajtov in ima tri dele:

Del paketa	Velikost	Vsebina	Namen
Časovni žig	8 bajtov	Trenutni čas kot double	Za izračun RTT ob odgovoru
Marker	16 bajtov	b'STRESSTEST_PROBE'	Identifikacija probe paketa
Ničle	40 bajtov	b'\x00' * 40	Zapolnitev do 64 bajtov skupaj

Tabela 5 Sestavni deli probe paketa

Korak 3: Zanka poskusov

```
for attempt in range(1, retries + 1):
    print(f"[VERIFY] Probe attempt {attempt}/{retries} ...")
    probe_sock.sendto(probe_packet, (target_ip, port))
    data, addr = probe_sock.recvfrom(1024)
```

Zanka se ponovi do 3-krat (privzeto). V vsakem poskusu pošljemo probe paket in čakamo na odgovor. recvfrom(1024) bo čakal največ timeout sekund (2.0). Strežnik odzvanja vsak prejet paket — torej bo odgovor naš probe paket.

Korak 4: Izračun RTT

```
recv_time = time.time()
send_time = struct.unpack('!d', data[:8])[0]
rtt = (recv_time - send_time) * 1000  # * 1000 = pretvorba v milisekunde
```

Ko prejmemo odgovor, preberemo časovni žig iz prvih 8 bajtov. RTT (Round Trip Time) je čas od pošiljanja paketa do prejema odgovora — vključuje pot tja, obdelavo na strežniku in pot nazaj.

```
[VERIFY] Server responded from 192.168.1.10:5000 (RTT: 0.423 ms)
```

Korak 5: Obravnava napak

```
except socket.timeout:
    print('[VERIFY] Timeout - no response')
    continue
except Exception as e:
    print(f'[VERIFY] Error: {e}')
    continue
```

socket.timeout se sproži, če v timeout sekundah ni odgovora. continue nadaljuje z naslednjim poskusom. Če vse 3 poizkuse presežemo brez odgovora, se zanka konča in funkcija vrne False.

4.5.4 Kdaj vrne True, kdaj False

Situacija	Rezultat	Posledica
Strežnik odgovori v timeout sekundah	True	Odjemalec začne s testiranjem
Strežnik ne odgovori po vseh retries poskusih	False	Program izpiše napako in se zaustavi (sys.exit(1))

Situacija	Rezultat	Posledica
Napačen IP naslov	False	Paket se izgubi, timeout, False
Požarni zid blokira UDP port	False	Paket se izgubi, timeout, False

Tabela 6 Obravnavanje napak

4.6 Funkcija: run_server(bind_ip, port, buffer_size)

def run_server(bind_ip: str, port: int, buffer_size: int = 65535):

4.6.1 Namen in pomen

Zažene UDP strežnik, ki posluša na določenih vratih. Ko prejme paket, ga nemudoma odzvanja nazaj pošiljatelju (echo). Hkrati beleži statistike: število paketov, prenesene bajte in latenco. Ko se zaustavi, izpiše zbrane statistike.

4.6.2 Argumenti

Argument	Tip	Privzeto	Opis
bind_ip	str	—	IP naslov za poslušanje. '0.0.0.0' pomeni 'sprejmi na vseh omrežnih vmesnikih'

Argument	Tip	Privzeto	Opis
port	int	—	Vrata, na katerih strežnik posluša (npr. 5000)
buffer_size	int	65535	Največja velikost paketa za sprejemanje. 65535 je absolutni maksimum UDP paketa.

Tabela 7 Funkcija Run_server argumenti

4.6.3 Podroben opis kode

Korak 1: Ustvaritev in konfiguracija vtičnice

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

sock.bind((bind_ip, port))
```

socket.socket() ustvari novo UDP vtičnico. SO_REUSEADDR je možnost, ki dovoli ponovni zagon strežnika na istih vratih takoj po zaustavitvi — brez te možnosti bi dobili napako 'Address already in use', ker OS drži vrata zasedena za kratek čas po zaprtju. bind() poveže vtičnico z določenim IP in vrati.

Korak 2: Inicializacija spremenljivk za statistike

```
packets_received = 0

bytes_received = 0

start_time = time.time()

latencies = []
```

Pred začetkom zanke se inicializirajo spremenljivke za beleženje statistik. latencies je seznam, v katerega se sproti dodajajo izmerjene latence vsakega paketa.

Korak 3: Timeout za odzivnost na zaustavitev

```
sock.settimeout(0.5)
```

Strežnik ne more kar čakati na paket za vedno — vsake 0.5 sekunde mora preveriti, ali je bila zahtevana zaustavitev (shutdown_requested.is_set()). settimeout(0.5) povzroči, da recvfrom() vrne socket.timeout izjemo po 0.5 sekundah, če ni prispel noben paket.

Korak 4: Glavna zanka sprejemanja

```
while not shutdown_requested.is_set():

    try:

        data, addr = sock.recvfrom(buffer_size)
```

```
        recv_time = time.time()
```

Zanka se vrti, dokler ni zahtevana zaustavitev. recvfrom() vrne dva podatka: data = vsebina paketa (bajti), addr = tuple (IP, port) pošiljatelja.

Korak 5: Echo — ozvanjanje paketa nazaj

```
try:
    sock.sendto(data, addr)
except:
    pass  # Ignoriraj napake pri ozvanjanju
```

Celoten prejet paket (vključno s časovnim žigom) se pošlje nazaj na naslov pošiljatelja. To je ključno za dve stvari: odjemalec lahko meri RTT (latenca), in verify_server_reachable() dobi odgovor na probe paket.

Korak 6: Izračun latence iz paketa

```
if len(data) >= 8:
    try:
        send_time = struct.unpack('!d', data[:8])[0]
        latency = (recv_time - send_time) * 1000
        latencies.append(latency)
```

```
    except:
        pass
```

Iz prvih 8 bajtov vsakega paketa se prebere časovni žig, ki ga je odjemalec zapisal ob pošiljanju. Razlika med recv_time in send_time je enosmerna latenca (one-way delay) — čas, ki ga je paket potreboval od odjemalca do strežnika.

Korak 7: Ažuriranje statistik

```
packets_received += 1
bytes_received += len(data)
```

Korak 8: Obravnava timeoutov in napak

```
except socket.timeout:
    continue  # Normalno — preverimo shutdown in nadaljujemo
except Exception as e:
    if not shutdown_requested.is_set():
        print(f"[ERROR] {e}")
```

socket.timeout je pričakovana in normalna izjema — zgodi se vsake 0.5 sekunde in nam da priložnost preveriti shutdown zastavico. Druge izjeme se izpišejo samo, če zaustavitev ni bila zahtevana (sicer bi ob normalnem zapiranju dobili lažne napake).

Korak 9: Končne statistike

```
duration = time.time() - start_time

print(f"  Duration: {duration:.2f}s")

print(f"  Packets received: {packets_received:,}")

print(f"  Throughput: {bytes_received*8/duration/1e6:.2f} Mbps")

print(f"  Avg latency: {mean(latencies):.3f} ms")
```

Na koncu se izpišejo vse zbrane statistike. bytes_received*8/duration/1e6 pretvori bajte v megabite na sekundo (Mbps): *8 = bajti → biti, /1e6 = biti → megabiti.

4.7 Funkcija: run_client(...)

def run_client(target_ip, port, start_pps, max_pps, step_pps, step_duration, packet_size, skip_verify):

4.7.1 Namen in pomen

Zažene UDP odjemalca, ki pošilja pakete na strežnik z naraščajočo hitrostjo. Začne pri start_pps paketov/sekundo in vsake step_duration sekunde poveča hitrost za step_pps, dokler ne doseže max_pps. Na ta način testiramo omrežje postopno — iščemo točko, pri kateri omrežje ne zdrži več celotne obremenitve.

4.7.2 Argumenti

Argument	Tip	Privzeto	Opis
target_ip	str	—	IP naslov strežnika (obvezno)
port	int	—	Ciljna vrata
start_pps	int	100	Začetna hitrost v paketih/sekundo (packets per second)
max_pps	int	1.000.000	Največja hitrost — ko jo dosežemo, test konča
step_pps	int	1000	Za koliko pps povečamo hitrost na vsakem koraku
step_duration	float	5.0	Koliko sekund trajamo na vsakem koraku

Argument	Tip	Privzeto	Opis
packet_size	int	1472	Velikost UDP paketa v bajtih (privzeto = max za ethernet brez fragmentacije)
skip_verify	bool	False	Preskoči verify_server_reachable() preveritev

Tabela 8 Funkcija Run_client argumenti

4.7.3 Zakaj 1472 bajtov?

Privzeta velikost paketa 1472 bajtov ni naključna:

Plast	Velikost	Kumulativno
Ethernet MTU (Maximum Transmission Unit)	1500 bajtov	1500
IP header (odštejemo)	– 20 bajtov	1480

Plast	Velikost	Kumulativno
UDP header (odštejemo)	– 8 bajtov	1472
UDP payload (naš paket)	= 1472 bajtov	1472

Tabela 9 Velikost paketa

Paketi večji od 1472 bajtov bi bili na omrežju fragmentirani — razdeljeni na več kosov. Fragmentacija zmanjša učinkovitost in povzroča dodatne težave pri testiranju.

4.7.4 Podroben opis kode

Korak 1: Preverjanje strežnika

```
if not skip_verify:
    if not verify_server_reachable(target_ip, port):
        sys.exit(1)
```

Pokliče verify_server_reachable(). Če strežnik ne odgovori in skip_verify ni True, se program zaustavi. skip_verify opcija obstaja za primere, ko strežnik ne odzvanja paketov (npr. specializiran merilnik prometa).

Korak 2: Sestavljanje predloge paketa

```
payload_data = b'X' * (packet_size - 8)
```

Pripravi se del paketa brez časovnega žiga — zapolnitev iz znakov 'X'. Odštejemo 8 bajtov za časovni žig, ki bo dodan pred vsakim pošiljanjem. Ta del se ne spreminja med pošiljanjem, kar je učinkovitejše od ponovnega ustvarjanja celega paketa.

Korak 3: Zunanja zanka — koraki hitrosti

```
current_pps = start_pps
while not shutdown_requested.is_set() and current_pps <= max_pps:
    interval = 1.0 / current_pps
    # ... notranja zanka za step_duration sekund ...
    current_pps += step_pps
```

Zunanja zanka se ponavlja za vsak korak hitrosti. interval je čas med dvema paketoma v sekundah. Npr. pri 1000 pps je interval = 0.001 sekunde (1 ms). Ko korak mine, se current_pps poveča za step_pps.

Korak 4: Notranja zanka — pošiljanje paketo pri fiksni hitrosti

```
while (time.time() - step_start < step_duration and
        not shutdown_requested.is_set()):
```

```
    timestamp = time.time()
    packet = struct.pack('!d', timestamp) + payload_data
    sock.sendto(packet, (target_ip, port))
```

Notranja zanka pošilja pakete pri trenutni hitrosti (current_pps) za step_duration sekund. Pred vsakim paketom se ustvari svež časovni žig in doda pred payload_data.

Korak 5: Natančno časovno krmiljenje (busy-wait)

```
next_send = time.time() + interval
while time.time() < next_send and not shutdown_requested.is_set():
    if next_send - time.time() > 0.001:
        time.sleep(0.0005)
```

Navadni time.sleep() ni dovolj natančen za visoke hitrosti (napaka do 15 ms na Windowsih). Busy-wait zanka preverja čas v zanki. Ko je preostali čas > 1 ms, spimo 0.5 ms — zmanjša CPU porabo. Ko je preostali čas < 1 ms, vrtimo zanko brez spanja — maksimalna natančnost.

Korak 6: Poročilo vsakega koraka

```
actual_pps = step_packets / step_elapsed
print(f"[STEP] Target: {current_pps:,} pps | Actual: {actual_pps:,.0f} pps | ...")
```

Po vsakem koraku se izpiše: ciljana hitrost, dejanska hitrost, število poslanih paketov in propustnost v Mbps. Razlika med target in actual pps razkrije, kdaj sistem ne zmore več slediti zahtevani hitrosti.

4.7.5 Primer izpisa odjemalca

[STEP] Target: 1.000 pps | Actual: 999 pps | Sent: 4.997 pkts | Rate: 11.35 Mbps

[STEP] Target: 2.000 pps | Actual: 1.998 pps | Sent: 9.991 pkts | Rate: 22.70 Mbps

[STEP] Target: 50.000 pps | Actual: 47.823 pps | Sent: 239.115 pkts | Rate: 543.02 Mbps

Ko actual_pps začne znatno zaostajati za target, smo dosegli mejo zmogljivosti.

4.8 Funkcija: main()

def main():

4.8.1 Namen in pomen

Vstopna točka programa — tukaj se vse začne. main() je odgovorna za: razčlenjevanje argumentov ukazne vrstice (argparse), validacijo parametrov, registracijo signal_handler in zagon izbranega načina (strežnik ali odjemalec).

4.8.2 argparse — analiza ukazne vrstice

parser = argparse.ArgumentParser(description='UDP Network Stress Test ...')

ArgumentParser je objekt, ki opisuje celoten vmesnik ukazne vrstice. Ko pokličemo parser.parse_args(), prebere sys.argv (argumente ob zagonu), jih razčleni in vrne objekt z atributi.

4.8.3 Vzajemno izključujoča skupina

```
mode = parser.add_mutually_exclusive_group(required=True)

mode.add_argument('-s', '--server', action='store_true')

mode.add_argument('-c', '--client', action='store_true')
```

add_mutually_exclusive_group zagotovi, da je mogoče izbrati samo --server ali --client, ne oboje hkrati. required=True zagotovi, da mora biti eden od njiju obvezno podan.

4.8.4 Vsi argumenti ukazne vrstice

Argument	Tip	Privzeto	Opis
-s / --server	bool (flag)	False	Zažene strežniški način
-c / --client	bool (flag)	False	Zažene odjemalski način
--bind	str	'0.0.0.0'	IP za vezavo strežnika

Argument	Tip	Privzeto	Opis
--target / -t	str	None	IP strežnika (obvezno za odjemalca)
-p / --port	int	5000	UDP vrata (skupna za strežnik in odjemalca)
--start-pps	int	100	Začetna hitrost v pps
--max-pps	int	1.000.000	Maksimalna hitrost v pps
--step-pps	int	1000	Povečanje na korak
--step-time	float	5.0	Trajanje koraka v sekundah
--packet-size	int	1472	Velikost paketa v bajtih (64–65507)

Argument	Tip	Privzeto	Opis
--skip-verify	bool (flag)	False	Preskoči preverjanje dosegljivosti

Tabela 10 Funkcija main() argumenti

4.8.5 Validacija parametrov

```
if args.client and not args.target:
    parser.error('--target is required when using --client')
if not (64 <= args.packet_size <= 65507):
    parser.error('--packet-size must be between 64 and 65507')
```

argparse ne more vedeti, da je --target obvezen samo za odjemalca — to preverimo ročno. Velikost paketa mora biti med 64 (minimalen smiseln UDP paket) in 65507 (absolutni maksimum UDP paketa: 65535 - 20 IP header - 8 UDP header).

4.8.6 Registracija signal handlerja

```
signal.signal(signal.SIGINT, signal_handler)
```

Registrira signal_handler za SIGINT (Ctrl+C). Ta klic mora biti pred zagonom run_server() ali run_client() — sicer Ctrl+C ne bi bil pravilno obravnavan.

4.8.7 Zagon izbranega načina

```
if args.server:
    run_server(args.bind, args.port)
else:
    run_client(args.target, args.port, start_pps=args.start_pps, ...)
```

Glede na izbrani način se pokliče ustrezna funkcija z vsemi parametri iz ukazne vrstice.

4.8.8 Vstopna točka programa

```
if __name__ == '__main__':
    main()
```

Ta vzorec je standardni Python idiom. Ko Python izvede datoteko, nastavi spremenljivko __name__:

- Če je datoteka zagnana direktno (python3 stress_test.py) → __name__ == '__main__' → main() se pokliče.

- Če je datoteka uvožena kot modul (import stress_test) → __name__ == 'stress_test' → main() se NE pokliče.

To pomeni, da bi kodo tega programa lahko uvozili v drug Python program brez stranskih učinkov — main() se ne bi samodejno zagnal

4.9 Povzetek — pregled vseh funkcij

Funkcija	Namen	Vrne
signal_handler(signum, frame)	Obravnava Ctrl+C, nastavi shutdown zastavico	—
verify_server_reachable(...)	Pošlje probe paket, preveri dosegljivost strežnika	bool
run_server(bind_ip, port, buffer_size)	Strežnik: sprejema pakete, echo, statistike	—
run_client(target_ip, port, ...)	Odjemalec: pošilja z naraščajočo hitrostjo	—
main()	Vstopna točka: argumenti, validacija, zagon	—

Tabela 11 Povzetek funkcij

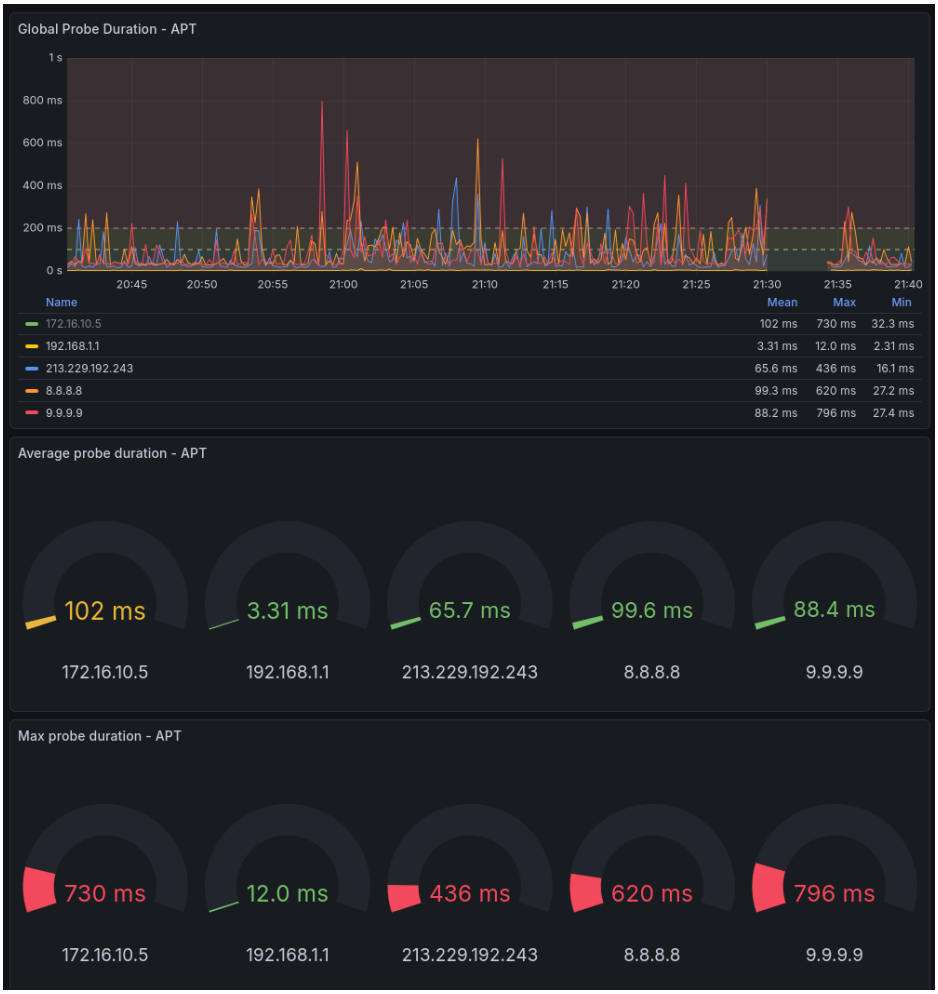
4.10 Celoten potek programa — od zagona do konca

#	Korak	Funkcija
1	Uporabnik zažene program z argumenti	—
2	main() razčleni in validira argumente	main()
3	Registrira se signal_handler za Ctrl+C	signal.signal()
4a	[STREŽNIK] Vtičnica se odpre, strežnik začne poslušati	run_server()
4b	[ODJEMALEC] Pošlje probe paket, čaka na odgovor	verify_server_reachable()
5b	[ODJEMALEC] Začne stopenjsko pošiljanje paketov	run_client()
6	Ob Ctrl+C: signal_handler nastavi shutdown zastavico	signal_handler()

#	Korak	Funkcija
7	Vse zanke se varno zaključijo	run_server() / run_client()
8	Izpišejo se končne statistike	run_server() / run_client()

Tabela 12 Potek programa

5. Prikaz simulacij

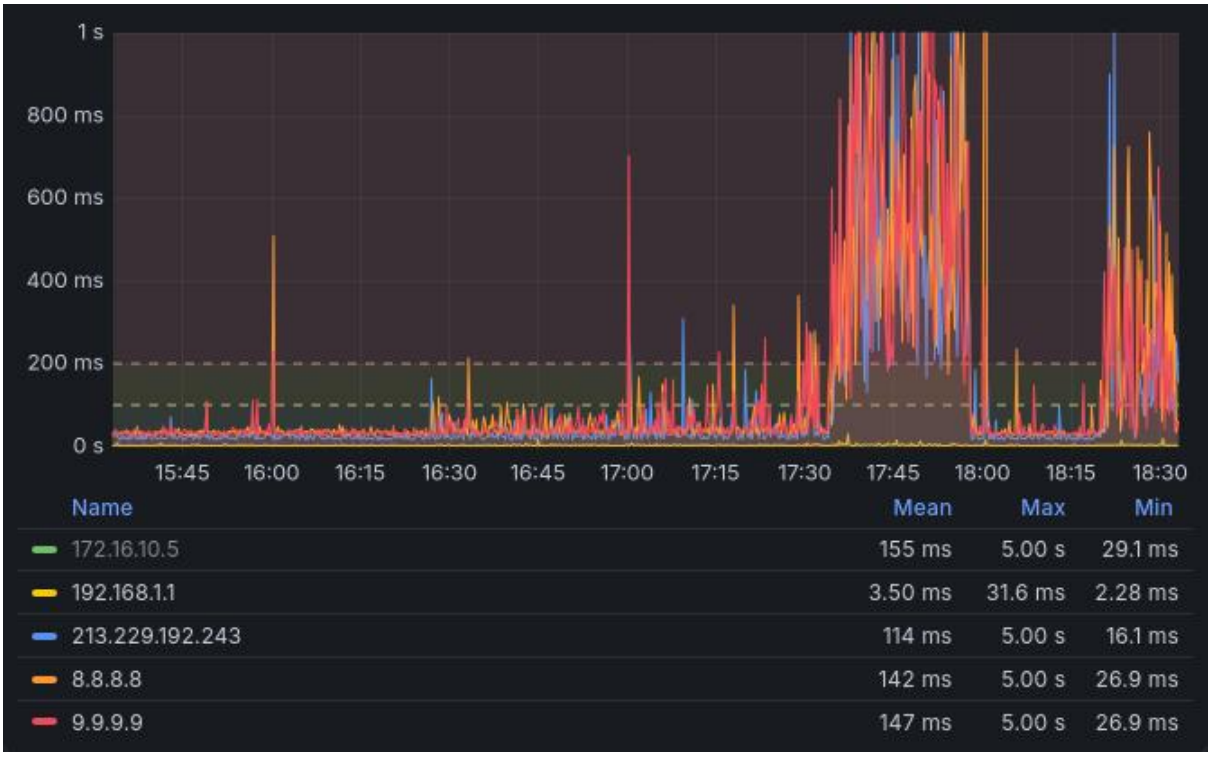


Slika 1 Prikaz obremenjenosti

Slika 1 prikazuje graf globalnega trajanja poizvedb (probe duration) za več različnih strežnikov oziroma naslovov IP. Graf prikazuje spremembe odzivnega časa skozi čas ter povprečne, maksimalne in minimalne vrednosti zakasnitve. Iz prikazanih podatkov je mogoče opaziti občasne skoke v odzivnem času, kar kaže na povečano obremenitev omrežja ali strežnika.

Slika 2 Nadzor hitrosti omrežja

Na sliki 2 je prikazan nadzor hitrosti prenosa podatkov v omrežju, vključno s hitrostjo prenosa (download), oddaje podatkov (upload) in zakasnitvijo (ping). Graf prikazuje spreminjanje hitrosti prenosa skozi čas ter povprečne, maksimalne in minimalne vrednosti prenosa. Takšni podatki omogočajo analizo zmogljivosti omrežne povezave med testiranjem obremenitve strežnika.

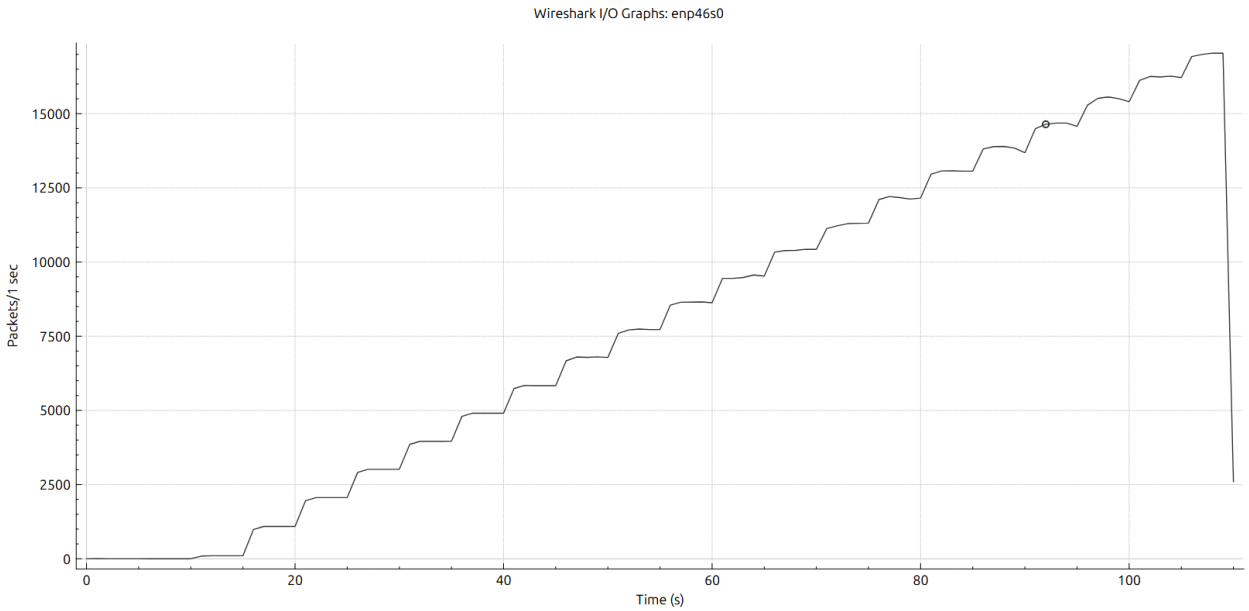


Slika 3 Povečanje odzivnega časa

Graf na sliki 3 prikazuje spremembe odzivnega časa več različnih strežnikov med obdobjem povečane omrežne aktivnosti. Opaziti je mogoče izrazito povečanje zakasnitve in večje število nihanj v odzivnem času, kar kaže na vpliv večje obremenitve ali simuliranega napada na stabilnost omrežnih storitev.

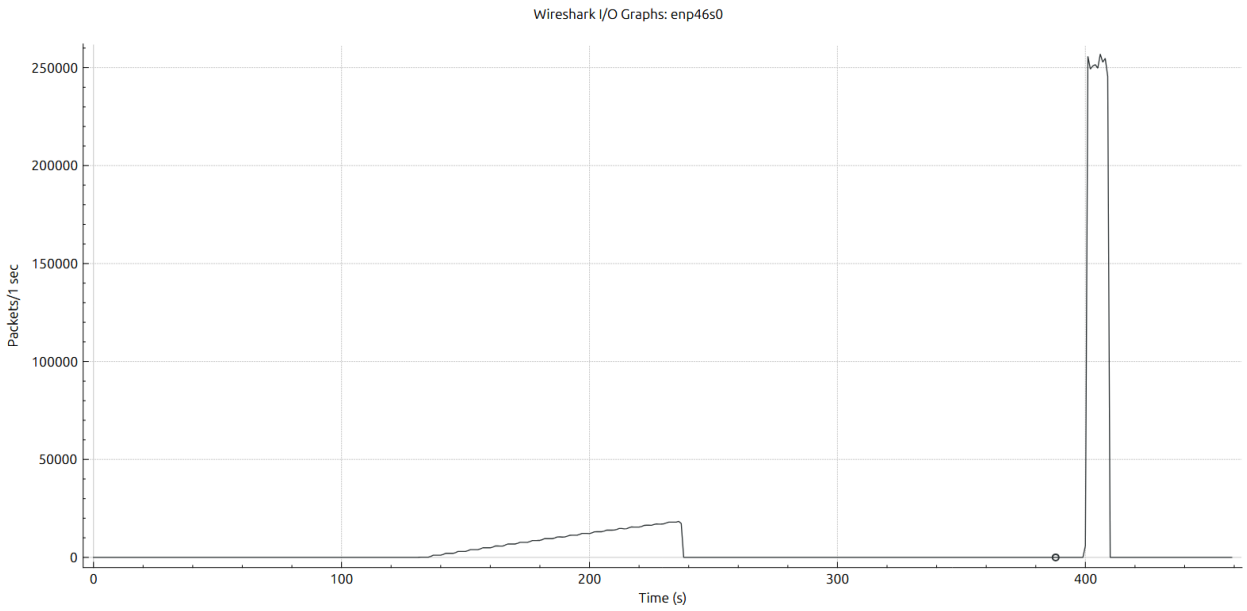
Slika 4 Primerjava zakasnitve med različnimi omrežnimi točkami

Na sliki je prikazan nadzor nad zakasnitvijo omrežnih poizvedb med več različnimi ciljnimi strežniki. Prikazane so povprečne, maksimalne in minimalne vrednosti trajanja poizvedb za različne IP naslove. Takšna primerjava omogoča boljši vpogled v stabilnost in zmogljivost posameznih omrežnih poti ter identifikacijo morebitnih ozkih grl v omrežju.



Slika 5 Postopno povečevanje prometa

Na sliki 5 je prikazan potek omrežnega prometa med simulacijo postopnega povečevanja obremenitve strežnika. Vidno je enakomerno naraščanje števila paketov na sekundo, ki se začne z približno 100 paketi na sekundo in se v stopnjah povečuje do približno 16.000 paketov na sekundo.



Slika 6 Hitro povečanje prometa oz. DDoS napad

Na sliki 6 je prikazano hitro in izrazito povečanje omrežnega prometa, ki je posledica simuliranega napada tipa DDoS. Graf prikazuje število paketov na sekundo skozi čas, pri čemer je v začetnem delu opaziti zelo nizko količino prometa. V določenem trenutku pa pride do nenadnega in zelo velikega porasta prometa, kjer število paketov na sekundo močno naraste na več kot 250.000 paketov. Takšen skok predstavlja situacijo, v kateri veliko število odjemalcev hkrati pošilja zahteve proti strežniku, s čimer se simulira obremenitev, značilna za DDoS napad.

6. Zaključek

DDoS napadi so postali vse pogostejši in vse bolj dodelani, saj jih napadalci nenehno razvijajo. Vse pogostejša postajajo orodja za obremenjevanje in onesposabljanje lokalnih omrežji in večjih tarč. Obramba pred njimi pa je zelo zahtevna in kompleksna, ker ni enostavne rešitve. Možnost napada lahko samo zmanjšamo z uporabo DDoS motivatorjev, omrežnih filtrov in sistemi za razdelitev obremenitve izhodne kode ter razprševanje omrežnega prometa. Keylogger pa je že dolgo uporabljen za krajo zaupnih podatkov, saj lahko napadalec izsiljuje podjetja ali posameznike. Tudi ta kibernetski napad postaja vse pogostejši zaščita pred njim pa postaja vse bolj učinkovita.

6.1 Potrjevanje oziroma zavračanje hipotez

6.1.1 Prva hipoteza

Prva hipoteza trdi, da DDoS napadi predstavljajo veliko grožnjo za informacijske sisteme, saj lahko preobremenijo omrežje ali strežnik in s tem onemogočijo dostop legitimnim uporabnikom. Na podlagi razlage različnih vrst DDoS napadov (volume-based, protokolni in aplikacijski napadi) lahko ugotovimo, da ti napadi dejansko povzročajo preobremenitev sistema in s tem nedostopnost storitve. Zato prva hipoteza drži.

6.1.2 Druga hipoteza

Druga hipoteza predpostavlja, da lahko ustrezni varnostni mehanizmi zmanjšajo tveganje za uspešen DDoS napad ali okužbo s keyloggerjem. V nalogi so predstavljeni različni načini zaščite, kot so filtriranje omrežnega prometa, sistemi za zaznavanje napadov, protivirusna zaščita in porazdelitev omrežnega prometa. Ti mehanizmi sicer omejijo možnost napada a ga ne morejo preprečiti. Zato druga hipoteza drži le delno.

6.1.3 Tretja hipoteza

Tretja hipoteza pravi, da se z večanjem uporabe interneta in digitalnih storitev povečuje tudi število kibernetskih napadov. Ker se vedno več storitev seli na internet in je zato nanj povezanih vedno več naprav, imajo napadalci tudi več potencialnih tarč. Posledično se število napadov, kot so DDoS napadi in uporaba keyloggerjev, povečuje. Zato tretja hipoteza drži.

7. Viri

[1] Bombal, D. (3. marec 2025). DDoS Attacks (HTTP/2, DNS, Hacktivist) // Real World Technical Analysis:

Pridobljeno iz YouTube. Nazadnje obiskano: 15.12.2025 ob 16.35

https://www.youtube.com/watch?v=t2jKcA1OyBE

[2] CDNetworks (27. avgust 2025). *Types of DDoS Attacks*..

Nazadnje obiskano: 10.2.2026 ob 21.55

https://www.cdnetworks.com/blog/cloud-security/types-of-ddos-attacks/

[3] Fortinet (2025). *What Is A Keylogger? Definition And Types*..

Nazadnje obiskano: 19.2.2026 ob 19.45

https://www.fortinet.com/resources/cyberglossary/what-is-keyloggers

[4]Heimdal (25. december 2025). *Types of Keyloggers: Definition, Examples and Prevention*.. //

Nazadnje obiskano: 7.2.2026 ob 17.30

https://heimdalsecurity.com/blog/what-is-a-keylogger/

[5] Jim Holdsworth in Matthew Kosinski (2025). *What is a distributed denial-of-service (DDoS) attack?*..

Nazadnje obiskano: 21.12.2025 ob 18.20

https://www.ibm.com/think/topics/ddos

[6]Oxford University Press (16. februar 2021).*How Keyloggers Work and How To Defeat Them*..

Nazadnje obiskano: 31.1.2026 ob 11.50

https://pmc.ncbi.nlm.nih.gov/articles/PMC7928946/

[7] Sosafe (22 May 2025). *What is a DDoS attack?*..

Nazadnje obiskano: 17.1.2026 ob 14.15

https://sosafe-awareness.com/glossary/ddos-attacks/