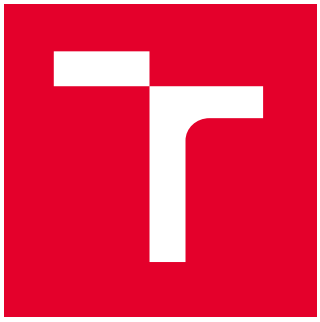


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

POKROČILÝ GENERÁTOR MODERNÍCH SLOW DOS ÚTOKŮ

ADVANCED GENERATOR OF MODERN SLOW DOS ATTACKS

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE

AUTHOR

Bc. Kryštof Trávník

VEDOUCÍ PRÁCE

SUPERVISOR

Ing. Marek Sikora

BRNO 2022

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Informační bezpečnost**

Ústav telekomunikací

Student: Bc. Kryštof Trávník ***ID:*** 193956

Ročník: 2 ***Akademický rok:*** 2021/22

NÁZEV TÉMATU:

Pokročilý generátor moderních Slow DoS útoků

POKYNY PRO VYPRACOVÁNÍ:

V dnešním světě DoS útoků patří pomalé DoS útoky zcela jistě k těm nejsofistikovanějším. Díky napodobování legitimních uživatelů s pomalým Internetovým připojením jsou tyto útoky obtížně detekovatelné a také velmi účinné.

Úkolem této diplomové práce je z dostupných modelů útoků vytvořit pokročilý program pro jejich generování a následné zátěžové testování. Součástí programu bude moderní a intuitivní grafické rozhraní a systém pro sledování a vyhodnocení průběhu útoku. Výsledný program bude umožňovat přehlednou a podrobnou parametrizaci útoků (včetně podpory IPv6, HTTPS a simulací DDoS). Struktura programu bude modulární, aby bylo umožněno jednoduché rozšíření o další typy útoků v budoucnu. Celkem bude program obsahovat moduly pro generování celkem 6 pomalých DoS útoků.

DOPORUČENÁ LITERATURA:

[1] SIKORA, Marek, Radek FUJDIAK, Karel KUCHAR, Eva HOLASOVA a Jiri MISUREC. Generator of Slow Denial-of-Service Cyber Attacks. Sensors. 2021, 21(16). ISSN 1424-8220. DOI:10.3390/s21165473

[2] TRIPATHI, Nikhil a Neminath HUBBALLI. Application Layer Denial-of-Service Attacks and Defense Mechanisms. ACM Computing Surveys. 2021, 54(4), 1-33. ISSN 0360-0300. DOI:10.1145/3448291

Termín zadání: 7.2.2022 ***Termín odevzdání:*** 24.5.2022

Vedoucí práce: Ing. Marek Sikora

doc. Ing. Jan Hajný, Ph.D.
předseda rady studijního programu

ABSTRAKT

Diplomová práce se zabývá vývojem modulárního generátoru pro pomalé útoky na odepření služeb. V teoretické části je detailně rozebrán vývoj protokolu HTTP, útoky na odepření služeb, vybrané pomalé útoky na odepření služeb a nakonec zmíněny některé existující generátory. Do vytvořeného generátoru je pak možno přidávat libovolný počet dalších skriptů, ke kterým je následně vytvořeno grafické rozhraní. Útoky je dále možno rozšířit o distribuovanou verzi za pomocí vytvořených botů, kteří jsou ovládány díky protokolu master-slave navrženém v této práci. Během útoku jsou generovány statistiky, které jsou v reálném čase prezentovány v rámci grafického rozhraní a nebo po skončení samotného testu.

KLÍČOVÁ SLOVA

Pomalé DoS útoky, Slowloris, Slowcomm, Slow Next, SlowDrop, SlowHTTPtest, Generátor, Apache2, Modulární, DDoS, HTTPS

ABSTRACT

This thesis deals with the development of a modular generator for slow denial-of-service attacks. The theoretical part discusses in detail the evolution of the HTTP protocol, denial of service attacks, selected slow denial of service attacks and finally mentions some existing generators. It is possible to add any number of additional scripts to the created generator, to which a graphical interface is automatically created. The attacks can be further extended to a distributed version using the created bots, which are controlled thanks to the master-slave protocol proposed in this work. During the attack, statistics are generated and presented in real time within the graphical interface or after the attack is completed.

KEYWORDS

Slow DoS attacks, Slowloris, Slowcomm, Slow Next, SlowDrop, SlowHTTPtest, Generator, Apache2, Modular, DDoS, HTTPS

Vysázeno pomocí balíčku `thesis` verze 4.07; `http://latex.feec.vutbr.cz`

TRÁVNÍK, Kryštof. *Pokročilý generátor moderních Slow DoS útoků*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2022, 77 s. Diplomová práce. Vedoucí práce: Ing. Marek Síkora,

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Bc. Kryštof Trávník
VUT ID autora:	193956
Typ práce:	Diplomová práce
Akademický rok:	2021/22
Téma závěrečné práce:	Pokročilý generátor moderních Slow DoS útoků

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora*

*Autor podepisuje pouze v tištěné verzi.

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu semestrální práce panu Ing.Markovi Síkorovi, za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Obsah

Seznam obrázků

Seznam výpisů

Úvod

V dnešní době, kdy data putují napříč komunikačními sítěmi v neustále větším měřítku, může jejich nedostupnost či jakékoli zdržení způsobit obrovské finanční ztráty a dokonce lze i očekávat dopad politický a celospolečenský. V prostředí internetu se čím dál tím častěji objevují snahy docílit takových dopadů záměrně za využití různých druhů útoku na odepření služeb. Vzhledem k tomu, že detekce proti běžným útokům na odepření služeb se neustále vyvíjí, vznikají stále nové typy útoků. Jednou ze skupin útoků, která se neustále rozrůstá o nové, sofistikovanější útoky, je skupina Slow Denial of Service (Slow DoS). Její detekce je v běžném síťovém provozu zpravidla komplikovanější a musí se tak využívat složitějších technik, které nejsou ve většině případech dostupné. Tato skutečnost a fakt, že na zrealizování útoků není potřeba velké množství výpočetního výkonu, dělá ze Slow DoS problematiku, které se je z bezpečnostních důvodů potřeba věnovat.

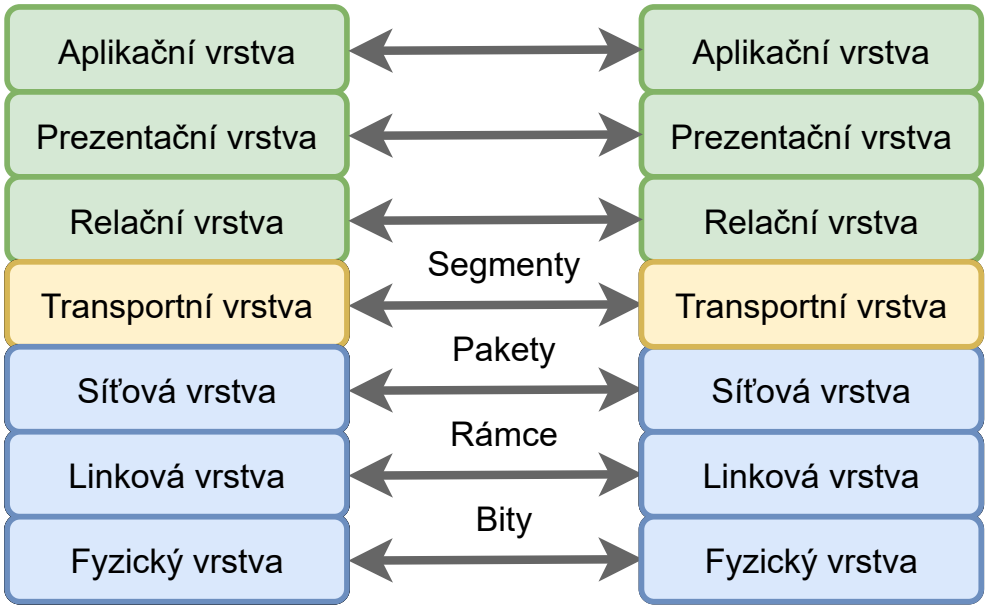
Diplomová práce se v praktické části zabývá vytvořením programové nadstavby na dodané skripty a generátory Slow DoS útoků. Tato nadstavba se skládá z dynamicky generovaného grafického rozhraní, rozšíření dodaných skriptů o funkcionalitu distribuované verze útoků (DDoS), Secure Hypertext Transfer Protocol (HTTPS) a možností z každého útoku vygenerovat statistiky průběhu. Teoretická část pak popisuje jednotlivé komponenty pro pochopení celkové problematiky. V první kapitole se čtenář seznámí se základními síťovými modely a jejich jednotlivými vrstvami. Detailněji je popsána vrstva transportní a převážně pak její protokol Transmission Control Protocol (TCP), který je stěžejní pro tuto práci. V druhé kapitole je pak představen protokol aplikační vrstvy, Hypertext Transfer Protocol (HTTP). V této kapitole se čtenář postupně seznámí s historií protokolu a jeho verzemi. Verze protokolu jsou následně chronologicky popsány a u každé z nich jsou zmíněny jejich výhody oproti verzím předchozím. Ve třetí kapitole jsou obecně popsány útoky na odepření služeb a jejich verze v distribuované podobě. Čtvrtá kapitola se pak již dopodrobna věnuje Slow DoS útokům a následně jsou detailněji rozebrané některé konkretní útoky. V páté kapitole jsou pak představeny některé existující generátory těchto útoků.

1 Základy síťové komunikace

Síťové inženýrství je složitá problematika, která vyžaduje nutnost znalostí z odvětví softwaru, firmwaru a hardwaru. Pro usnadnění je celá koncepce sítí popsána referenčním modelem obsahujícím několik vrstev. Každá vrstva se zabývá nějakým konkrétním úkolem a je nezávislá na všech ostatních vrstvách. Jako celek však téměř všechny síťové úlohy závisí na všech těchto vrstvách. Vrstvy mezi sebou sdílejí data a jsou na sobě závislé pouze při přijímání a odesílání dat. Mezi nejznámější referenční modely patří International Standards Organization Open Systems Interconnection (ISO/OSI) a Transmission Control Protocol/Internet Protocol (TCP/IP) [1].

1.1 OSI/ISO

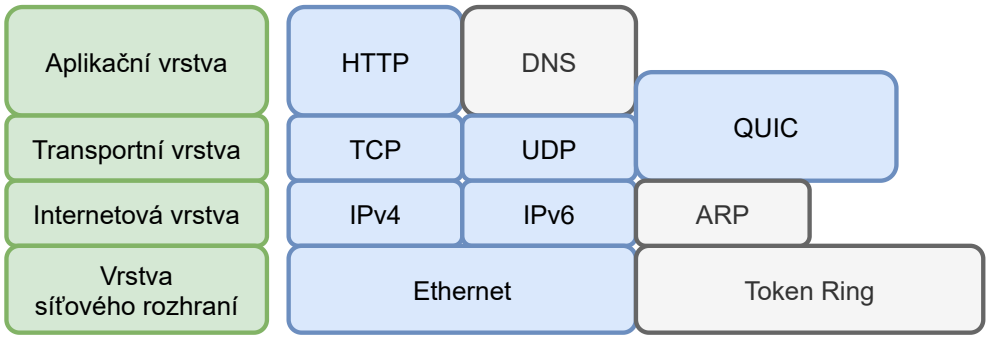
Model OSI je logický a koncepční model, který definuje síťovou komunikaci používanou systémy otevřenými pro vzájemné propojení a komunikaci s jinými systémy. Model rovněž definuje logickou síť a efektivně popisuje přenos síťových paketů pomocí různých vrstev protokolů. Tento model má sedm vrstev a je názorně zobrazen na obrázku 1.1 [2].



Obr. 1.1: Vrstevnatý model OSI/ISO.

1.2 TCP/IP

Protokol TCP/IP pomáhá definovat způsob komunikace v počítačových sítích, založených na protokolu Internet Protocolu (IP). TCP/IP je speciálně navržen jako model, který nabízí vysoce spolehlivý tok bajtů mezi koncovými uživateli přes nespolehlivou internetovou síť. TCP/IP má na rozdíl od OSI/ISO pouze čtyři vrstvy a to z důvodů spojení horních tří vrstev (aplikační, prezentační a relační) do jedné vrstvy aplikační a spodních dvou vrstev (fyzické a linkové) do vrstvy síťového rozhraní. Dále jsou zde vrstvy obdobně jako u OSI/ISO vrstva transportní a internetová (síťová), což lze vidět na obrázku 1.2. Na tomto obrázku jsou také zmíněné některé stěžejní protokoly jednotlivých vrstev [2].



Obr. 1.2: Referenční model TCP/IP a významné protokoly.

1.2.1 Transportní vrstva

Transportní vrstva je druhou vrstvou modelu TCP/IP. Jedná se o vrstvu end-to-end, která slouží k doručování zpráv hostiteli. Je označována jako end-to-end, protože zajišťuje point-to-point[1] spojení mezi zdrojovým a cílovým hostitelem (na rozdíl od hop-to-hop[2]), aby bylo možné spolehlivě poskytovat služby. Jednotkou zapouzdření dat v transportní vrstvě je segment [3]. Standardní protokoly používané na transportní vrstvě jsou User Datagram Protocol (UDP) a Transmission Control Protocol (TCP).

UDP

Protokol UDP je nespojově orientovaný, což znamená, že se předem nenavazuje spojení. UDP posílá data bez jakékoliv další režie a neřeší například jejich ztrátu. Menší

[1] V telekomunikacích se spojením point-to-point rozumí komunikační spojení mezi dvěma koncovými komunikačními body nebo uzly.

[2] Spojením hop-to-hop se rozumí, přenos mezi několika body nebo uzly.

režie na jednu stranu uleví komunikačnímu kanálu, ale logicky není možné garantovat, že data byla doručena úspěšně. UDP se proto používá v situacích, kdy je přijatelná určitá ztráta dat, například při živém přenosu videa/audia, nebo v případech, kdy je rychlost přenosu rozhodujícím faktorem.

TCP

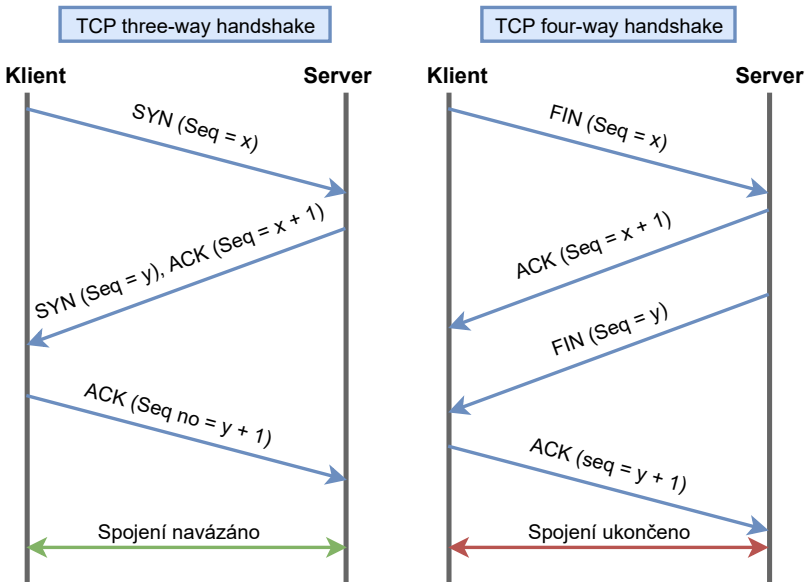
Protokol TCP je na rozdíl od protokolu UDP spojově orientovaný a před jakýmkoliv přenosem musí být navázáno spojení, které je udržováno po celou dobu daného přenosu. Během navázaného spojení je garantováno, že veškerá data budou doručena. Protokol TCP je mimořádně spolehlivý a používá se pro všechny možné účely, od surfování po webu (HTTP) až po posílání e-mailů pomocí Simple Mail Transfer Protocol (SMTP). Při návštěvě webové stránky se protokol TCP používá k zajištění toho, aby dorazil text, obrázky i kód, který je potřebný k vykreslení stránky. Bez protokolu TCP by obrázky nebo text mohly chybět a nebo dorazit v nesprávném pořadí, což by mohlo vést ke špatnému načtení stránky. Spolehlivé spojení také umožňuje sledovat kolik dat bylo přeneseno a po přijetí určitého množství dat potvrdit úspěšné doručení. V případě, že toto potvrzení není obdrženo, tak jsou data odeslána znovu a to od posledního potvrzeného segmentu, což může vést k zatížení přenosového kanálu. K určení, kolik dat se může bezpečně poslat bez nutnosti opakovaní se používá mechanismus **TCP Window Size** neboli velikost okna.

Navázaní a ukončení spojení v TCP

K navázání spojení se používá takzvaný "three–way handshake". K ukončení spojení se pak používá podobný mechanismus nazvaný "four–way handshake". Oba tyto mechanismy lze vidět na schématu 1.3 [4].

TCP Window Size

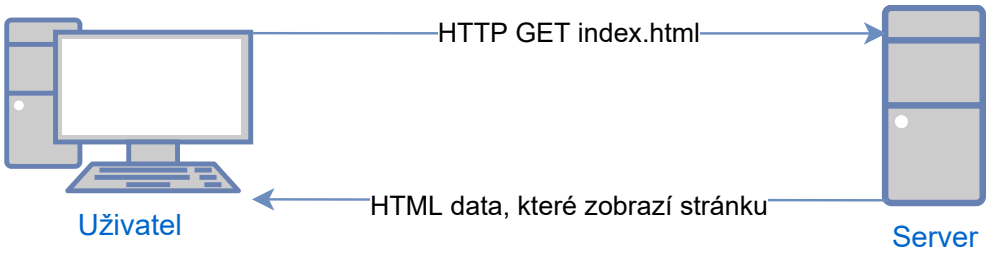
Když příjemce segmentů odešle potvrzení o přijetí, sdělí zároveň odesílateli kolik dat může poslat než obdrží další potvrzení. Dá se tedy říct, že velikost okna v podstatě udává množství vyrovnávací paměti alokované pro příjem dat. Na začátku three-way handshake, jenž je popsán v kapitole 1.2.1, je velikost okna nastavena na menší hodnotu. S každým dalším úspěšným potvrzením se velikost okna po domluvě obou stran zvětšuje. V případě, že potvrzení není obdrženo, tak se velikost okna naopak zmenšuje. Výše popsáný způsob pouze znázorňuje jak velikost okna funguje, v praxi se pro stanovení velikosti okna používají pokročilejší techniky jako například TCP Slow start nebo Random early detection jejichž popis je nad rámec tohoto textu [5].



Obr. 1.3: Three–way handshake pro zahájení komunikace a four–way handshake pro ukončení.

2 Protokol HTTP

HTTP je protokol na aplikační vrstvě, který se používá k přístupu ke zdrojům prostřednictvím sítě World Wide Web (WWW). Výchozí port pro komunikaci HTTP je standardně 80, ale lze jej případně nakonfigurovat dle potřeb. Komunikace HTTP se skládá z komunikace klienta a serveru, kdy klient žádá server o data. Server požadavky zpracuje a klientovi vrátí požadovaná data, viz 2.1. Jedná se o požadavky na webové servery, které jsou používány k návštěvě webových stránek. Přístup na požadovanou webovou stránku, například na adresu www.vut.cz, je zadáván jako Uniform Resource Locator (URL) ve formátu Fully qualified domain name (FQDN)[1].



Obr. 2.1: Jednoduchá HTTP komunikace.

Protokol HTTP vznikl v roce 1989 až 1991 ve verzi dnes známe jako HTTP/0.9, která obsahovala pouze jeden řádek s jednou metodou GET, která je používána pro získání dat ze serveru. Server na takovou žádost odpovídá pouze s požadovanými daty. Hlavní rozdíl oproti novějším verzím protokolu je absence hlaviček a chybových i informačních kódů. Absence hlaviček v praxi znamená, že je možný přenos pouze souborů typu HTML[2]. Protokol HTTP přinášel v každé další verzi aktualizace, které jsou popsány v následujících podkapitolách [6].

2.1 HTTP/1.0

Protokol HTTP ve verzi 1.0 nabízel oproti předchozí verzi především přidaní konceptu hlaviček, které byly v žádosti od klienta i odpovědi od serveru. Hlavičky přidaly možnost přenosu metadat a protokol se tak stal extrémně flexibilní a rozšířitelný. Díky hlavičce Content–Type tak bylo možné přenášet i jiný formát souboru než HTML. Další důležitou aktualizace bylo přidání chybových a informačních kódů, které umožňovaly webovým prohlížečům poznat, zda byl požadavek úspěšný nebo

[1]Fully qualified domain name (FQDN) je úplné doménové jméno konkrétního počítače nebo hosta na internetu.

[2]Hypertext Markup Language, který slouží k vytváření webových stránek.

neúspěšný a na základě této informace se adaptovat. V neposlední řadě pak byla přenášena verze HTTP, která byla použita. V této verzi byly také přidány další dvě metody POST a HEAD. POST slouží k přenosu dat ve formě bloku (například formulář) od klienta na server. HEAD slouží k získávaní metadat obsažených v hlavičce [6]. Hlavní nevýhodou verze HTTP/1.0 byla nutnost otevření nového TCP spojení pro každý jeden požadavek, což vedlo k velkým problémům s efektivitou.

2.2 HTTP/1.1

HTTP/1.1 vznikl v roce 1999, brzy po vydání předchozí verze, a velmi rychle se stal standardem, který je definován v RFC 2616 a je používán dodnes. Nová verze řešila převážně zmíněné nedostatky předchozí verze, ale přinesla i nové funkce a metody. Spojení již nebylo nutné navazovat po každém jednotlivém požadavku, nýbrž navázání spojení proběhlo pouze na začátku komunikace, což ušetřilo jak čas, tak šířku pásma. Vytvořené spojení přineslo možnost odeslat druhý požadavek na webový server, i když se stále čekalo na odpověď na dříve odeslaný požadavek, což přineslo výrazné snížení latence komunikace. Protokol HTTP/1.1 nově podporoval mechanismy pro vyjednání jazyka, kódovaní a typu. Přibylo záhlaví HOST, které bylo povinné a umožňovalo hostovat různé domény ze stejné IP adresy [6]. V této verzi, byly nově definovány tyto metody [7]:

- PUT - nahrává data na webový server, kde objekt je jméno nahrávaného souboru,
- DELETE - smaže data z webového serveru,
- CONNECT - vytvoří tunel na server identifikovaný cílovým prostředkem,
- OPTIONS - popisuje možnosti komunikace pro webový server,
- TRACE - provede test zpětné smyčky zpráv podél cesty k webovému serveru,
- PATCH - aplikuje částečné úpravy již nahraného souboru.

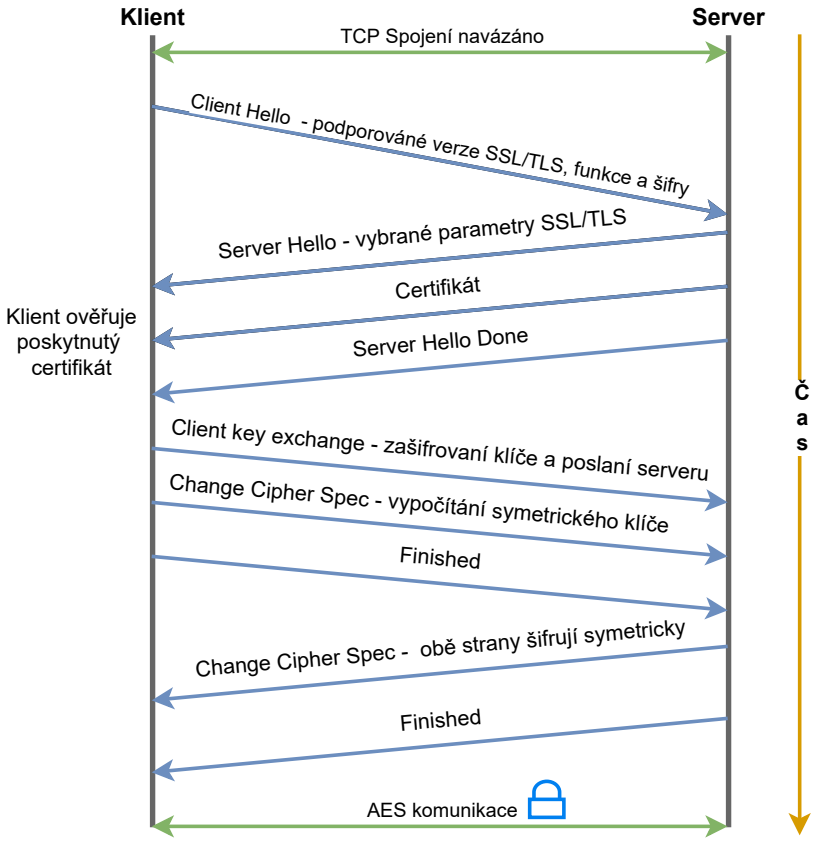
2.2.1 HTTPS

Protokol HTTP byl v původní verzi zamýšlen pouze pro akademické potřeby a nepočítal s použitím pro běžné uživatele nebo firmy. Veškerá data posílána samotným HTTP protokolem tak byla v otevřené podobě a byla tak náchylná na manipulaci od útočníka. Z toho důvodu proběhlo v roce 1994 významné vylepšení protokolu na dnes známé Hypertext Transfer Protocol Secure (HTTPS). Namísto odesílání dat pouze přes základní model TCP/IP vytvořila společnost Netscape Communications nad protokolem TCP další šifrovanou přenosovou vrstvu, Secure Sockets Layer (SSL). Protokol SSL 1.0 nebyl nikdy zveřejněn, ale protokol SSL 2.0 a jeho

nástupce SSL 3.0 už zveřejněny byly a umožnily tak bezpečnou komunikaci pro každého uživatele. Tyto verze šifrovaly a zaručovaly pravost zpráv vyměňovaných mezi serverem a klientem. Protokol SSL 3.0 byl nakonec standardizován jako Transport Layer Security (TLS) [9, 6].

Šifrování výměny klíčů je zajištěno pomocí asymetrické kryptografie a šifrovaní komunikace pak pomocí symetrické kryptografie, viz 2.2. Symetrická kryptografie používá pro šifrovaní a dešifrování stejný klíč. Asymetrická kryptografie používá k šifrování dva různé klíče: [8]

- Soukromý klíč - tento klíč se nachází na webovém serveru a je ve vlastnictví jednotlivých webových stránek. Slouží k dešifrování informací zašifrovaných veřejným klíčem.
- Veřejný klíč - tento klíč je k dispozici všem, kteří chtějí se serverem komunikovat bezpečným způsobem. Informace zašifrované veřejným klíčem lze dešifrovat pouze soukromým klíčem.



Obr. 2.2: Průběh SSL/TLS komunikace.

V dnešní době je používání protokolu HTTPS na webových serverech silně doporučeno.

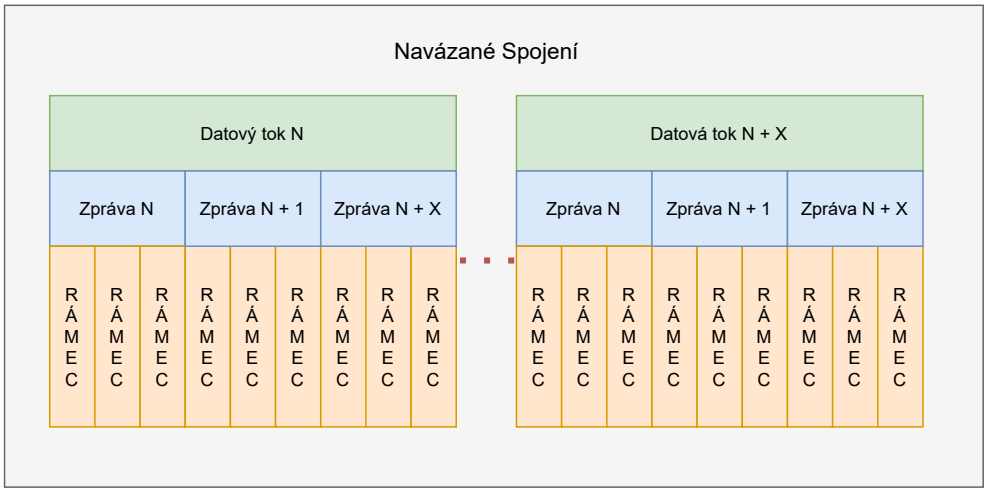
2.3 HTTP/2.0

V průběhu let se webové stránky stávaly mnohem složitější. Některé z nich se staly dokonce samostatnými webovými aplikacemi. Zobrazovalo se více vizuálních prvků a rostla také komplexnost skriptů přidávajících interaktivitu. Pro přenos dat bylo potřeba více a více požadavků HTTP, což způsobilo větší složitost a režii spojení [6].

Protokol HTTP/2 vznikl jako protokol Speedy (SPDY), vyvinutý především ve společnosti Google se záměrem snížit latenci načítání webových stránek pomocí technik, jako je komprese, multiplexování a prioritizace. Když v květnu roku 2015 pracovní skupina httpbis pro hypertextový přenosový protokol v rámci Internet Engineering Task Force (IETF) sestavovala nový standard HTTP/2 setkala se s velkou podporou od nejrozšířenějších prohlížečů (Chrome, Opera, Internet Explorer a Safari). Částečně díky této podpoře prohlížečů došlo od roku 2015 k výraznému rozšíření protokolu, přičemž obzvláště vysoká byla míra rozšíření mezi novými weby [10]. Do července 2016 jej používalo 8,7 % všech webových stránek, což představovalo více než 68 % všech požadavků. Ke konci roku 2021 používá verzi HTTP/2 46.8 % všech webových stránek [11].

2.3.1 Binární rámcová vrstva

Celá táto podkapitola čerpá ze zdroje [12]. Protokol ve verzi 2 opět přináší řadu vylepšení oproti předchozí verzi. Nově je HTTP/2 spíše binární než textový protokol. Rámcová vrstva kóduje požadavky/odpovědi a rozděluje je do menších informačních paketů, což výrazně zvyšuje flexibilitu přenosu dat. Na rozdíl od protokolu HTTP/1.1, který musí využívat více spojení TCP, protokol HTTP/2 vytváří mezi klientem a serverem jediné spojení. V rámci spojení existuje více datových proudů. Každý proud se skládá z několika zpráv ve známém formátu požadavek/odpověď a každá zpráva se rozdělí na menší jednotky nazývané rámce, viz 2.3. Na nejnižší úrovni se komunikační kanál skládá ze svazku binárně kódovaných rámců, z nichž každý je označen určitým datovým tokem. Identifikační značky umožňují rámce během přenosu rozložit a na druhém konci je opět skládat. Rozložené požadavky a odpovědi mohou být posílány paralelně, aniž by blokovaly zprávy za nimi, což je proces nazývaný multiplexování. Multiplexování řeší problém v protokolu HTTP/1.1 tím, že zajišťuje, aby žádná zpráva nemusela čekat na dokončení jiné. To také znamená, že servery a klienti mohou posílat souběžné požadavky a odpovědi, což umožňuje lepší kontrolu a efektivnější správu připojení. Protože multiplexování umožňuje klientovi vytvářet paralelně více proudů, musí tyto proudy využívat pouze jedno spojení TCP. Existence jediného spojení vylepšuje protokol HTTP/1.1 tím, že snižuje paměťovou a výpočetní stopu v celé síti. To vede k lepšímu využití sítě a šířky

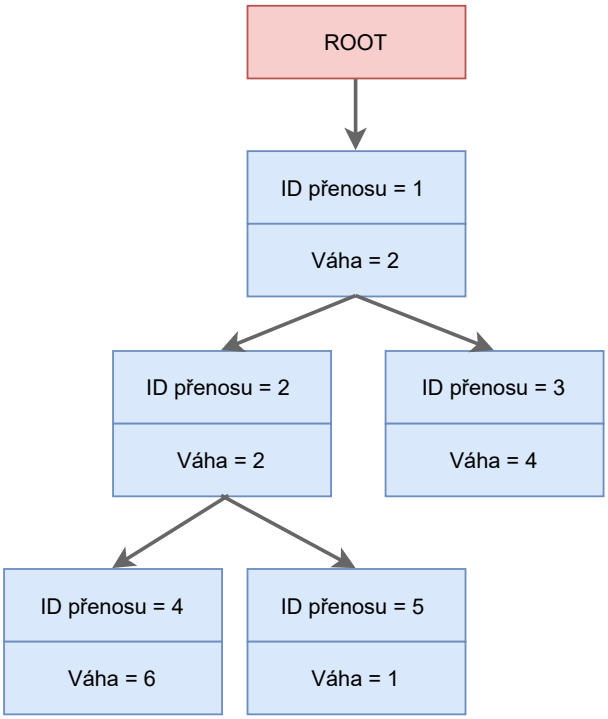


Obr. 2.3: Spojení ve verzi HTTP/2.

pásma, a tím se snižují celkové provozní náklady. Jedno spojení TCP také zlepšuje výkon protokolu HTTPS, protože klient a server mohou opakovaně používat stejnou zabezpečenou relaci pro více požadavků/odpovědí. Ačkoli specifikace protokolu HTTP/2 neukládá povinnost používat vrstvu TLS, mnoho hlavních prohlížečů podporuje protokol HTTP/2 pouze s protokolem HTTPS.

2.3.2 Prioritizace přenosu dat

Prioritizace přenosu dat řeší problém, kdy požadavky soutěží o prostředky webového serveru. Vývojářům umožňuje měnit váhu požadavků pro lepší optimalizaci výkonu aplikace. V předchozí podkapitole bylo zmíněno, že binární rámcová vrstva organizuje zprávy do paralelních datových proudů. Klient může díky prioritizaci odesílat souběžné požadavky na server a upřednostnit určité odpovědi, tím, že každému proudu přiřadí váhu v rozmezí 1 až 256. Vyšší číslo označuje vyšší prioritu. Klient také může uvést závislost jednotlivých proudů na jiných proudech zadáním ID proudu, na kterém závisí. Pokud je identifikátor nadřazeného přenosu vynechán, je považován za závislý na kořenovém přenosu. Server tyto informace použije k vytvoření stromu, který mu umožní určit pořadí, v jakém budou požadavky načítat svá data [13]. Příklad stromu, na základě kterého jsou upřednostněné určité požadavky, lze vidět na následujícím obrázku 2.4. Vývojář nastavuje váhy v požadavcích podle svých potřeb. Například můžete přiřadit nižší prioritu načtení obrázku s vysokým rozlišením.



Obr. 2.4: Strom závislostí a priorit.

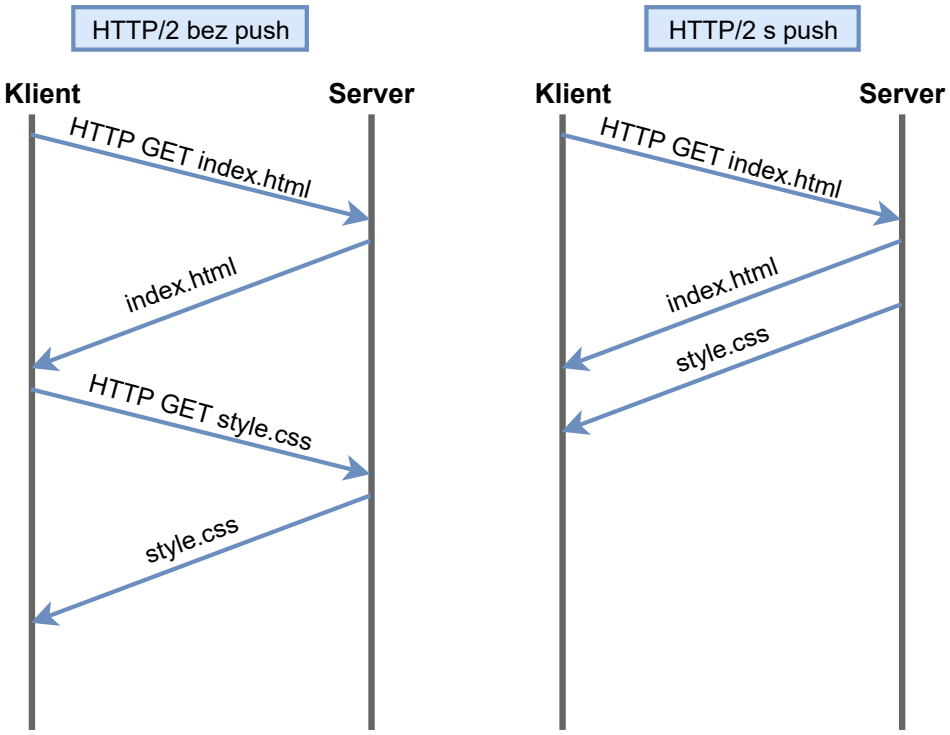
2.3.3 Server push

Vzhledem k tomu, že protokol HTTP/2 umožňuje více souběžných odpovědí na počáteční požadavek GET, může server klientovi odeslat i další data, o které klient nežádá. Tento proces se nazývá server push. Klient se pak sám rozhodne zda bude další data ukládat do mezipaměti nebo je odmítne [15]. Porovnaní komunikace bez využití server push a s využitím, lze vidět na následujícím schéma 2.5 Zde je důležité poznamenat, že ovládaní push serveru je v rukou klienta. Pokud by klient potřeboval server push upravit, nebo ho dokonce zakázat, může kdykoli odeslat rámec SETTINGS a provést kýženou změnu.

2.3.4 Komprese

Komprese se vyskytuje již v předchozí verzi protokolu HTTP, nicméně až v této verzi je možno komprimovat celou zprávu, a to včetně hlavičky. HTTP/2 dokáže oddělit hlavičky od jejich dat, čímž vzniknou rámce hlavičky a dat. Komprese HPACK[3] specifická pro HTTP/2 pak může tento rámec hlavičky komprimovat. Tento algoritmus dokáže zakódovat metadata záhlaví a výrazně zmenší jeho velikost. HPACK navíc dokáže sledovat dříve přenesená pole metadat a dále je komprimovat podle dynamicky měněného indexu sdíleného mezi klientem a serverem. Tuto skutečnost lze

[3]Komprese založena na vyhledávacích tabulkách a Huffmanově kódování [16].



Obr. 2.5: Rozdíl v komunikaci při použití push.

ilustrovat například na následujících dvou požadavcích 2.1 a 2.2. V případě 2.1 jsou odeslána pole jako method, scheme, host, accept a user–agent, které se v průběhu relace nemění, a proto v požadavku 2.2, který je odeslán ve stejné relaci, je posíláno pouze pole path, které se mění v průběhu komunikace [12, 14].

Výpis 2.1: Záhlaví bez komprese.

```
1 method:      GET
2 scheme:      https
3 host:        vut.cz
4 path:        /portal
5 accept:      /image/jpeg
6 user-agent:  Mozilla/5.0
```

Výpis 2.2: Záhlaví při použití komprese.

```
1 path:        /portal/rozvrh
```

2.4 HTTP/3

S prudkým nárůstem internetového provozu a masivním rozšířením technologií v mobilních zařízeních nezajišťoval protokol HTTP/2 dostatečně plynulé a transparentní prohlížení webu. Protokol nezvládal velké objemy dat a rychlost moderního internetového provozu, zejména s ohledem na stále rostoucí nároky aplikací pracujících v reálném čase. Pouhé čtyři roky po zavedení protokolu HTTP/2 začal vznikat nový standard založený na experimentálním protokolu Quick UDP Internet Connections (QUIC)[4] společnosti Google, HTTP/3. Jeho cílem bylo zvýšit rychlost a bezpečnost interakce uživatelů s webovými stránkami a rozhraními Application programming interfaces (API)[5][17, 18].

Protokol HTTP/3 poskytuje značnou část požadovaných a potřebných zlepšení, jak v oblasti výkonu, tak zabezpečení, které jsou potřeba pro další etapu ve vývoji protokolu HTTP. Navzdory tomu a navzdory skutečnosti, že existuje prokazatelná potřeba vylepšení protokolu HTTP/2, je stále ještě otázkou, zda bude HTTP/3 nakonec skutečně znamenat plošné zlepšení oproti HTTP/2 v jeho dnešní podobě.

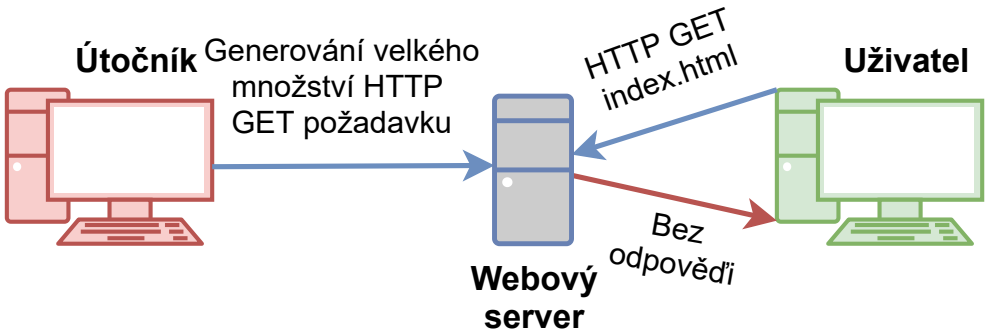
[4]QUIC vznikl jako alternativa k protokolu TCP, TLS a HTTP/2 s cílem zlepšit uživatelský komfort, zejména dobu načítání stránek.

[5]Application programming interfaces zjednodušuje vývoj a vylepšení softwaru tím, že umožní aplikacím snadnou a bezpečnou výměnu dat a funkcí.

3 Útoky na odepření služeb

Útok na odepření služeb, znám také pod zkratkou DoS, má za cíl zpomalit či znepřístupnit internetové služby legitimním uživatelům. Sekundárním cílem může být taky přetížení či zahlcení jednotlivých zařízení nebo celé sítě. Příklad ve formě HTTP DoS útoku lze vidět na obrázku 3.1. Mezi nejčastější cíle patří informační systémy, webové servery kritické infrastruktury jednotlivých zemí, vládních, bankovních institucí, e-shopů a mnohé další. Cílem DoS útoků není přímá krádež nebo ztráta dat, ale nedostupnost dané služby, což má za důsledek vážný finanční, informační, ale i bezpečností dopad na oběť útoku. Dalším negativním dopadem je frustrace uživatelů a poškození dobrého jména poskytovatele služby, což může poškozeného poskytovatele vést k tendenci zamlčet a nenahlásit fakt, že se stal terčem útoku na odepření služeb [20].

Útoků na odepření služeb v posledních měsících přibývá. V prvním pololetí roku 2021 došlo k přibližně 5,4 milionům útoků, což je téměř o 11 % více než ve stejném období předchozího roku [21]. V lednu roku 2021 byl dokonce zaznamenán největší počet útoků DoS v historii, a to 972 000 [21]. Tento nárůst byl důsledkem pandemie COVID-19 v roce 2020, jelikož došlo k přenesení působnosti mnoha organizací do on-line prostoru, aniž by na to byly připraveny [21]. K zatím největšímu útoku na odepření služeb v historii došlo formou distribuovaného útoku v červnu 2020, kdy společnost Amazon Web Services uvedla, že odrazila třídenní distribuovaný útok na odepření služeb (DDoS), který dosáhl maximálního objemu 2,3 terabajtu za sekundu [19].



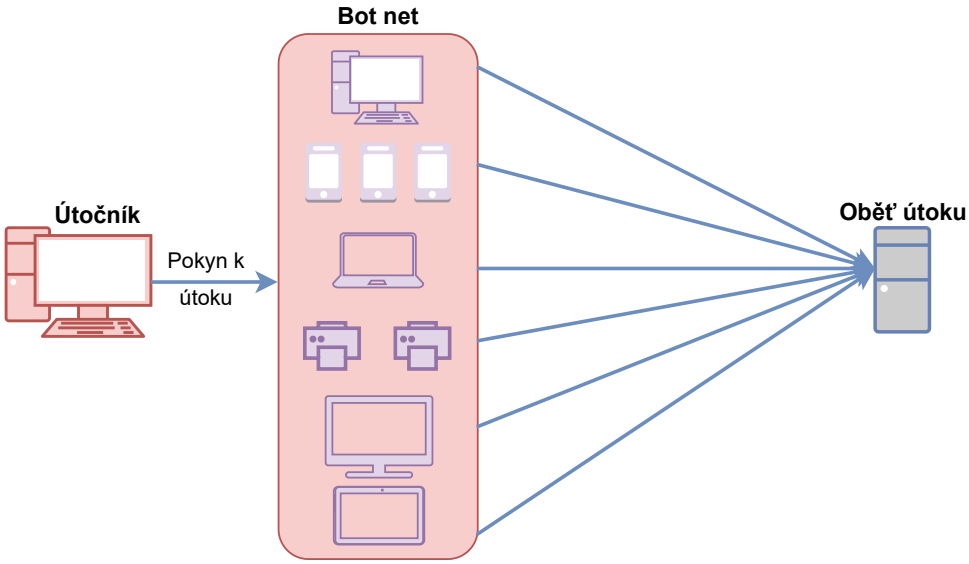
Obr. 3.1: Primitivní DoS útok.

3.1 Distribuované útoky na odepření služeb

Samotný útok na odepření služeb je používán spíše v laboratorních podmínkách než v reálných útocích, a to z důvodu nedostatečného výpočetního výkonu nebo

snazší identifikace zdroje útoku. V novodobých útocích je proto využívána technika distribuované formy DoS neboli DDoS.

Útoky DDoS se provádějí pomocí sítě, která se skládá z počítačů a dalších zařízení (například Internet of Things (IoT)[1]), která byla infikována malwarem[2], což umožňuje jejich vzdálené ovládání útočníkem. Jednotlivá zařízení se označují jako boti (nebo zombie) a skupina botů se pak nazývá botnet. Průběh této techniky je znázorněn na obrázku 3.2 [23]. Útočník může řídit útok zasíláním vzdálených pokynů jednotlivým botům. Útok probíhá tak, že každý bot posílá požadavky na IP adresu cíle, což může způsobit zahlcení serveru nebo sítě a následné odepření služby běžnému uživateli. Protože každý bot je legitimním internetovým zařízením, může být obtížné oddělit útočný provoz od běžného provozu a útočník si navíc může snadněji zachovat anonymitu. DDoS útoky je možno rozdělit do tří základních kategorií. Záplavové útoky, útoky na aplikační vrstvu a útoky na protokoly [22].



Obr. 3.2: Průběh DDoS útoku.

3.1.1 Záplavové útoky

Záplavové útoky jsou ze zmíněných útoků v kapitole 3.1 nejčastější. K provedení takového útoku je využit již zmíněný botnet (zařízení jsou rozmístěny po celém světě), který zaplavuje cíl útoku obrovským provozem. Ohromné množství dat zahltí

[1]Internet of Things označuje síť fyzických objektů, které jsou vybaveny různými technologiemi za účelem připojení a výměny dat s jinými zařízeními a systémy prostřednictvím internetu. Zařízení můžou být běžné chytré senzory v domácnosti i sofistikované průmyslové nástroje.

[2]Malware je škodlivý software, který je určen k poškození, zničení nebo ovládaní počítačů a dalších zařízeních.

veškerou šířku pásma webových stránek. Výsledkem je, že legitimní provoz nemůže projít a běžný uživatel tak nedostane od serveru žádnou odpověď. Útoky založené na objemovém zahlcení se měří v bitech za sekundu (bps) [24].

Příkladem záplavového útoku je UDP flood. Je využit protokol UDP, který jak bylo zmíněno v kapitole 1.2.1 nemusí před zahájením komunikace navázat spojení. UDP flood zahltí náhodné porty cílového zařízení tak, že s přibývajícím počtem přijatých a zodpovězených paketů UDP není systém schopen objem požadavků zpracovat, a přestane proto reagovat [24].

3.1.2 Útoky na protokol

Na rozdíl od záplavových útoků se protokolové útoky zaměřují na vyčerpání zdrojů serveru namísto zahlcení šířky pásma. Zaměřují se také na takzvané "mezilehlé komunikační zařízení", jako jsou například firewally a load balancery. Útočníci zahlcují webové stránky a jejich zdroje tím, že zadávají nelegitimní protokolové požadavky, aby spotřebovali dostupné zdroje. Síla těchto útoků se měří v paketech za sekundu (pps) [24].

Příkladem útoku na protokol je TCP SYN flood, který zneužívá zranitelnosti three–way handshaku protokolu TCP, který byl popsán v kapitole 1.2.1. Útočník odesílá pakety SYN na všechny porty cílového serveru. Server obdrží několik zdánlivě legitimních požadavků na navázání komunikace. Na každý pokus odpoví paketem SYN-ACK. Útočník následně neodešle očekávaný ACK a způsobí, že cílový server bude čekat na potvrzení svého paketu SYN-ACK, který ale nikdy nepřijde. Během čekání nemůže server spojení ukončit odesláním paketu RST a spojení tak zůstane otevřené. Než se spojení ukončí, útočník odešle další paket SYN. Tím zůstává stále větší počet spojení napůl otevřených až nakonec nezbudou žádné zdroje serveru pro legitimního uživatele [25].

3.1.3 Útoky na aplikační vrstvu

Útoky na aplikační vrstvu obecně vyžadují méně prostředků než předchozí dva útoky. Tento typ útoků se zaměřuje na zranitelnosti v jednotlivých webových serverech od firem jako je Apache, Windows nebo OpenBSD. Útoky na aplikační vrstvu vyřazují servery z provozu tím, že provádějí velké množství požadavků, které se zpočátku jeví jako legitimní (napodobují chování reálných uživatelů). Protože se útoky zaměřují pouze na konkrétní aplikační pakety, mohou zůstat nepovšimnuty. Útoky se snaží narušit konkrétní funkce nebo vlastnosti aplikace. Útoky se měří v požadavcích za sekundu (rps). [24] Jedna ze skupin útoků na aplikační vrstvu jsou slow DoS útoky, jež jsou dále podrobněji popsány.

4 Slow DoS útoky

Slow DoS jsou typy útoku na odepření služeb, které se snaží dosáhnout stejného cíle pomocí odlišných technik. Zatímco záplavový útok je založený na generování co největšího počtu bitů za sekundu s využitím rozsáhlé botnetové sítě, slow DoS může být proveden s relativně malou výpočetní silou. Slow DoS může být dokonce proveden z jediného zařízení s průměrným připojením k internetu (útok by byl teoreticky možný provést z mobilního zařízení). Další výhodou slow DoS útoků oproti klasickým záplavovým útokům je jejich obtížnější detekce. Tyto aspekty pak přispívají k rostoucí popularitě slow DoS útoků a k objevovaní stálé sofistikovanějších technik provedení [26].

Útok využívá slabin aplikační vrstvy, která není schopna obsloužit tolik spojení jako například vrstva transportní u záplavového útoku. Cílem útoku je zmocnění se všech dostupných připojení na této vrstvě, což má za důsledek odepření služby legitimnímu uživateli. Útočník se pokouší, aby server zpracovával pouze jeho vlastní požadavky tím, že zaplní kapacitu serveru pouze svými spojeními. Při dosažení úplného zaplnění kapacit můžou nastat dva různé scénáře, na které je potřeba adekvátně reagovat. V prvním případě může vypršet stanovená doba ze strany serveru (takzvaný timeout), která udává jak dlouho server čeká na odpověď klienta než spojení ukončí. Před vypršením limitu tak útočník musí zaslat požadavek "keep–alive", který tento limit restartuje a celý proces se opakuje. U druhé možnosti se vytvořené spojení z nějakého důvodu ukončí (například ztracení paketu keep–alive) a útočník pak musí každé takové spojení nahradit novým spojením a zároveň pokaždé předběhnout regulérního uživatele. Jak získaní veškeré kapacity serveru, tak udržení tohoto stavu musí být dosaženo co možno nejrychleji, ale s použitím co nejmenšího datového provozu. Díky tomu je útok možno provést s minimální vypočtení silou a především obejít detekční mechanismy, které jsou založeny na statistické detekci útoků na odepření služeb [27]. Slow DoS útoky se dále rozdělují na několik dalších podskupin, některé jsou dále popsány a všechny kategorie pochází z literatury [27].

4.1 Slow DoS s opožděnými odpovědmi

Útoky, patřící do této kategorie, posílají cíli zdánlivě legitimní požadavky, které vedou k nuceným výpočetně náročným operacím na straně cíle, za účelem odpovědi. Vzhledem k náročnosti operace dochází ke zpoždění obvyklého času, který je potřeba na odpověď. Odpovědi tak přicházejí později než při standardních provozních podmínkách, a proto jsou takové útoky označované jako Slow DoS s opožděnou odpovědí. Do této kategorie spadá mimo jiné útok Apache range headers [27].

Apache range headers

Tento útok cílí na webový server Apache ve verzi 1. Útok je založen na HTTP parametru "byte range[1]". V případě útoku Apache range headers jsou odesílány žádosti, které si opakovaně vyžadují velké množství dat. Požadavky se vzájemně překrývají, což v praxi znamená, že v prvním požadavku jsou dotázány data 1-10 a v druhém 2-11 místo 11-20. Požadavky tohoto typu pak donutí server vytvořit velký počet kopií požadovaných dat a dojde tak k vyčerpaní prostředků serveru. Vývojáři Apache serveru vydali záplatu a v dnešní době již není možné využít parametru byte range pro provedení toho útoku [27].

4.2 Slow DoS na vyčerpání zdrojů

V této kategorii útočník čeká na ukončení navázaného legitimního spojení, které se snaží okamžitě navázat a obsadit tak celý komunikační kanál. Cílem útoku slow DoS na vyčerpání zdrojů je odhadnout konec legitimního spojení a obsazení tohoto spojení dříve, než ho naváže legitimní uživatel. Pro lepší pochopení tohoto typu útoků je popsán Low-Rate DoS Attack (LoRDAS) [27].

LoRDAS

Útočník se snaží odhadnout nebo předvídat okamžiky, kdy budou serverem uvolněny prostředky. Cílem útoku je zmocnit se uvolněných spojení dříve než to stihne legitimní klient. Odhad se liší od cílové služby oběti, takže například pro službu na streamování videa, lze předpokládat, že k uvolnění prostředku dojde těsně po skončení streamu. Útok se tedy dá provést jen za předpokladu, že útočník odhadne okamžik ukončení spojení, pokud se útočníkovi podaří odhadnout tento okamžik u všech spojení, které má server k dispozici, dojde k vyčerpání zdrojů serveru. Na tomto principu jsou založeny i další útoky spadající do této kategorie [27].

4.3 Slow DoS s dlouhotrvajícími požadavky

Útoky DoS s dlouhotrvajícími požadavky zasílají cíli útoku neúplné požadavky, čímž vyčerpávají jeho zdroje, zatímco cíl čeká na dokončení daného požadavku, nemůže spojení zavřít. Klíčovým bodem Slow DoS s dlouhotrvajícími požadavky je to, že nikdy neodešle poslední prázdný řádek hlavičky, ve snaze udržet spojení naživu co nejdéle a zároveň zprávu rozdělit na několik částí (segmentů), které bude postupně

[1]Parametr byte range je použit když klient žádá server pouze o část požadovaného souboru.

posílat vždy před vypršením limitu. Příkladem útoku s dlouhotrvajícími požadavky je Slowloris a Slow Post [27].

Slowloris

Slowloris funguje tak, že otevírá nová připojení k cílovému webovému serveru a nechává je otevřená tak dlouho, jak je potřeba. Útok nepřetržitě odesílá dílčí požadavky HTTP GET, přičemž žádný z nich není dokončen. Slowloris postupně odesílá hlavičky HTTP pro každý požadavek, ale nikdy jej ve skutečnosti nedokončí. Po vyčerpání všech zdrojů cílového serveru jsou další pokusy o připojení odmítnuty. Slowloris navíc neodesílá nelegitimní pakety a může tak pohodlně obejít Intrusion Detection system (IDS)[2] [27].

Slow Post

Útok Slow POST využívá oproti útoku Slowloris požadavek HTTP POST. Tento typ požadavku se obvykle používá k odesílání dat do HTML formulářů. Při útoku obsahuje pole Content-Length (určuje velikost přenášeného formuláře) vysokou hodnotu, což znamená, že server bude očekávat přijetí velkého množství dat. Záhlaví podvrženého požadavku je pak legitimně ukončeno a požadavek obsahuje data s několika náhodnými znaky. Útočník pak vyčká a před vypršením časovače ukončení spojení opět posílá malý kus dat a udržuje tak co nejvíce spojení, což vede k vyčerpaní dostupných zdrojů serveru [27].

4.4 Slow DoS s dlouhotrvajícími odpovědmi

V případě slow DoS s dlouhotrvajícími odpovědi jsou odesílány legitimní a úplné požadavky. Tyto požadavky jsou ale upravené tak, aby co nejvíce natáhly dobu, kterou server potřebuje na odpověď. Vzhledem k tomu, že odpovědi serveru jsou obsáhlé, může si útočník odpověď číst maximální možnou dobu. Fakt, že jsou požadavky legitimní, logicky ztěžuje detekci a následné odhalení útoku. Nejznámějším zástupcem této podskupiny je útok Slow Read [27].

Slow Read

Při útoku Slow Read se posílá velké množství legitimních požadavků HTTP na server a následné odpovědi se zpracovávají co nejpomaleji daný server dovolí. Pomalého čtení se docílí stanovením co nejmenší hodnoty pro vyrovnávací paměť. Při běžné

[2]Intrusion Detection system je zařízení nebo softwarová aplikace, která monitoruje síť nebo systém a detekuje podezřelé aktivity nebo porušování stanovených pravidel.

komunikaci má většina přijatých paketů velikost 1 448 bajtů avšak při Slow Read je server donucen nastavit malou velikost okna a posílá pakety o velikosti v řádech jednotek až desítek bajtů. Jak bylo řečeno v kapitole 1.2.1, hodnota okna je nastavena pomocí počátečního paketu SYN v three–way handshake. Důsledek útoku jsou dlouho trvající odpovědi, kdy v případě vícenásobných spojení dojde k nedostupnosti služeb [27].

4.5 Slow DoS nezávislé na protokolu aplikační vrstvy

Poslední kategorie se vyznačují nezávislostí na protokolu. V praxi to znamená, že útok je možno provést jak na nejčastěji zmiňovaný protokol HTTP, tak například na protokol File Transfer Protocol (FTP)[3] a jiné další. Do této kategorie spadají mimo jiné i útoky jako Slowcomm a Slow Next.

Slowcomm

Princip útoku je založený na útoku Slowloris 4.3. Útočník odešle co největší množství neúplných požadavků na server, který musí čeká na zbytek dat, což vede k obsazení všech jeho zdrojů a legitimní uživatelé tak nemůžou být obslouženi. Útočník pak udržuje pomocí zmíněné techniky navázaná spojení otevřená, viz 4.1. Na rozdíl od Slowlorise je Slowcomm schopen monitorovat navázané spojení. Pokud se spojení zavře, dokáže Slowcomm reagovat a spojení znovu otevřít. Slowcomm je mnohem efektivnější, nedostupnost serveru může teoreticky trvat do nekonečna. Vzhledem k sofistikovanosti útoku je detekce útoku opět o něco těžší než u Slowlorise [28].

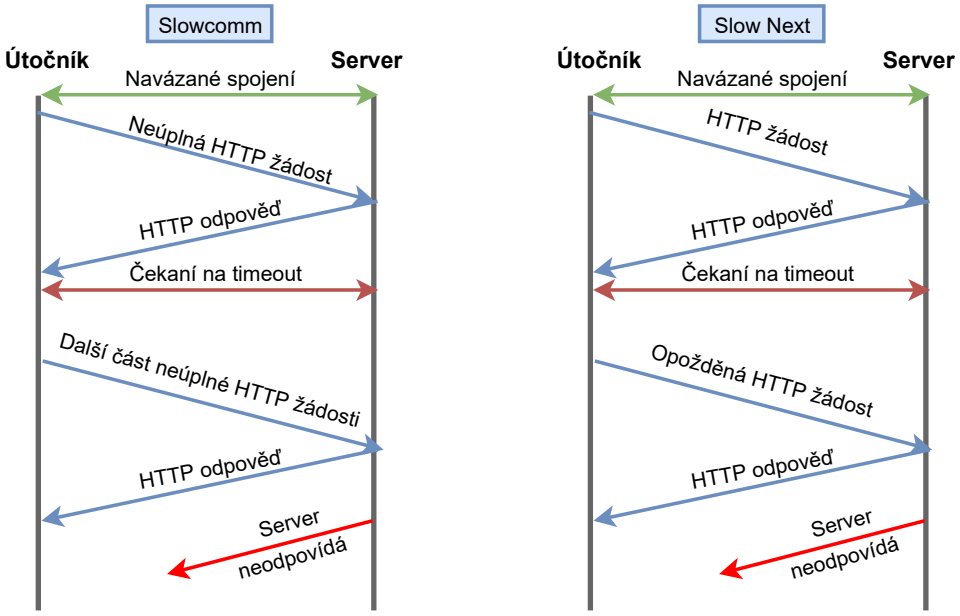
Slow Next

Slow Next zneužívá časový interval mezi odpovědí serveru a dalším požadavkem klienta v rámci vytvořeného spojení. V tomto útoku odešle útočník velké množství požadavků na server, který na ně odpovídá. Server pak chvíli čeká, zda útočník v rámci každého vytvořeného spojení požádá o další data. Útočník se odmlčí na stanovený čas a po jeho vypršení pošle další požadavky v rámci všech spojení tak, aby spojení zůstala otevřená po celou dobu, viz 4.1. Výsledkem útoku je opět vyčerpaní prostředků serveru pro běžné uživatele. Komunikace opět obsahuje legitimní pakety, a je proto snazší obejít detekční systémy [28].

[3]FTP (File Transfer Protocol) je síťový protokol pro přenos souborů mezi počítači.

SlowDrop

Útok SlowDrop patří mezi relativně nové (popsán v roce 2019) a komplexnější útoky než výše popsané. Útočník zde opět používá profil legitimního uživatele a zranitelnosti protokolu TCP. Útočník pak simuluje chování uživatele připojeného přes nespolehlivý kanál (například přes slabé bezdrátové připojení). Myšlenka útoku SlowDrop spočívá v opakovaném požadavku na určitý prostředek na serveru (nejlépe co největší) a následném náhodném zahazovaní přijatých paketů od serveru. Zahazování paketů má simulovat legitimní případy v komunikaci, které můžou reálně nastat. Například při slabém bezdrátovém připojení nebo pomalém internetovém přenosu. V praxi běžně používané Intrusion Detection system (IPS)[4] nebo firewally mohou neúmyslně odpojit legitimního uživatele, což útok dělá ještě více účinější [28].



Obr. 4.1: Schéma útoku Slowcomm a Slow Next.

[4]Intrusion Detection system (IPS) je rozšíření systému IDS, který může kromě monitorování sítě aktivně přijímat potřebná opatření k zabránění vzniku útoku.

5 Generátory Slow DoS útoků

V následující kapitole jsou obecně popsány nejznámější generátory některých útoků, z nichž některé jsou zmíněny v předchozím textu. V dostupných zdrojích je výčet těchto generátorů obsáhlý, ale nejednotný, chybí jejich systematický přehled. Záměrem této práce, je proto mimo jiné i snaha o vytvoření jednotného generátoru s přehledným grafickým rozhraním, který poskytne administrátorům možnost snazšího testování jejich služeb.

5.1 SlowHTTPtest

SlowHTTPTest je veřejně dostupný nástroj pro simulaci některých útoků typu Denial of Service na aplikační vrstvě s využitím malého datového proudu. Využívá k tomu převážně protokol HTTP. Dá se využít k otestování webového serveru na zranitelnost DoS nebo jen k zjištění, kolik paralelních připojení server zvládne. SlowHTTPTest funguje na většině nejrozšířenějších operačních systémech jako je Linux a Mac OS. Ke spouštění na Microsoft Windows používá Cygwin[1], který běží na Dockeru[2]. Mezi nejznámější implementované útoky patří:

- Slowloris,
- Slow HTTP POST,
- Slow Read,
- Apache Range Header.

SlowHTTPTest je užitečný nástroj a i když se dá využít k reálným útokům, tak se dnes používá spíše jako testovací nástroj pro administrátory serverů, kteří cíleně testují výdrž serveru proti jednotlivým útokům a následně zkoumají výsledky, které nástroj dokáže zobrazit jak v textové, tak grafické podobě [29].

5.2 Slowloris

Slowloris je bezplatný open source[3] nástroj s kódem, který je volně dostupný na Githubu [30]. Pomocí tohoto nástroje lze provést útok typu Denial of Service. Jedná se o framework napsaný v jazyce Python. Tento nástroj umožňuje vyčerpat prostředky serveru za použití již zmíněného útoku Slowloris 4.3. Další velmi podobný nástroj, který opět využívá převážné útok Slowloris je například **PyLoris** [31].

[1] Prostředí podobné Unixu, které slouží jako rozhraní příkazového řádku pro Microsoft Windows

[2] Softwarový framework pro vytváření, spouštění a správu kontejnerů v cloudu.

[3] Termín open source označuje něco, co mohou lidé upravovat a sdílet, protože jeho návrh je veřejně přístupný (v tomto případě kód programu).

5.3 Packet Cannon

Packet Cannon je generátor vyvinutý v rámci bakalářské práce na Vysokém učením technickém v Brně a je opět volně přístupný na Githubu. Generátor je napsán v programovacím jazyce C a jako jeden z prvních má grafické rozhraní, které je vytvořené v subsystému WPF [4]. V tomto generátoru je možné zvolit útoky Slowloris, Slow Port a Slow Read a následně zvolit parametry pro každý jednotlivý útok. Packet Cannon navíc přichází se simulovaným DDoS útokem, který je však možné spustit pouze v lokální síti, která nemá ochranu proti Address Resolution Protocol (ARP) spoofingu. Více informací o tomto generátoru je možné nalézt zde [33].

5.4 SlowDosGen

Generátor opět vyvinutý v rámci bakalářské práce na Vysokém učením technickém v Brně, ale v tomto případě není veřejně dostupný na Githubu, ale je pouze volně ke stažení v rámci této bakalářské práce [32]. Generátor je napsán v Pythonu 3.0 a funguje formou skriptu a je ovládán pomocí parametrů. Generátor obsahuje útoky Slowcomm a Slow Next. SlowDosGen je rozdělen do dvou samostatných útoků, které jsou použity v rámci generátoru vytvořeného v této práci [32].

5.5 SlowDrop DoS attack

Další z generátorů vyvinutých v rámci diplomové práce na Vysokém učením technickém v Brně nabízí implementaci útoku SlowDrop. Opět se jedná o skriptovací generátor, který je ovládán pomocí několika parametrů, které jsou potřebné pro provedení daného útoku. Generátor je napsán v jazyce Python ve verzi 3.0 a je také součástí generátoru vyvíjeného v této semestrální práci [34].

[4] Knihovna pro tvorbu grafického rozhraní v programovacím jazyce C.

5.6 Porovnaní zmíněných generátorů

V následující podkapitole je formou tabulky 5.1 znázorněno co jednotlivým generátorům chybí a nebo čím naopak disponují v porovnaní s generátorem nevrženým v této práci.

Tab. 5.1: Porovnaní různých generátorů s navrhovaným generátorem.

Nástroj/Funkcionalita	Modulární	GUI	Dynamicky generované GUI	DDoS	HTTPS	Statistiky v reálném čase	CSV statistiky	Grafy	IPv6
SlowHTTPtest	Ne	Ne	Ne	Ne	Ano	Ano	Ano	Ano	Ano
Packet Canon	Ne	Ano	Ne	Ano	Ne	Ne	Ne	Ne	Ne
SlowDosGen	Ne	Ne	Ne	Ne	Ne	Ne	Ne	Ne	Ne
Slow Drop	Ne	Ne	Ne	Ne	Ne	Ne	Ne	Ne	Ne
Slowloris	Ne	Ne	Ne	Ne	Ne	Ne	Ne	Ne	Ne
Navrhovaný generátor	Ano	Ano	Ano	Ano	Ne	Ano	Ano	Ano	Ne

6 Praktická část semestrální práce

V teoretické části práce bylo cílem seznámení se s vývojem protokolu HTTP a jeho přínosy a nedostatky v daných verzích, dále pak obecné studium útoků na odepření služeb a podrobnější popis slow DoS útoků. V kapitole 5 pak byly popsány jednotlivé generátory a to jak ty, jež jsou rozšířené napříč komunitou zabývající se ochranou proti různým formám útoků na odepření služeb, tak generátory, které byly vyvinuté na půdě Vysokého učení technického v Brně. V praktické části diplomové práce jsou získané znalosti využity pro vývoj nového generátoru, který si klade za cíl obsáhnout funkce dříve známých a popsaných generátorů. Generátor byl vyvíjen a testován v laboratorním prostředí, které je popsáno v podkapitole 6.1.

6.1 Laboratorní prostředí

V této kapitole je detailněji popsáno prostředí, ve kterém se navrhovaný generátor vyvíjí a převážně testuje. Laboratorní prostředí se skládá ze tří hlavních komponent. Jedná se o webový server, legitimního uživatele a útočníka. Všechna tato zařízení jsou vytvořena pomocí virtualizace v nástroji VMware a jedná se o různé distribuce operačního systému Linux v 64-bitové verzi. Zařízení se také nachází ve stejné interní virtuální síti, která je překládána prostřednictvím network address translation (NAT).

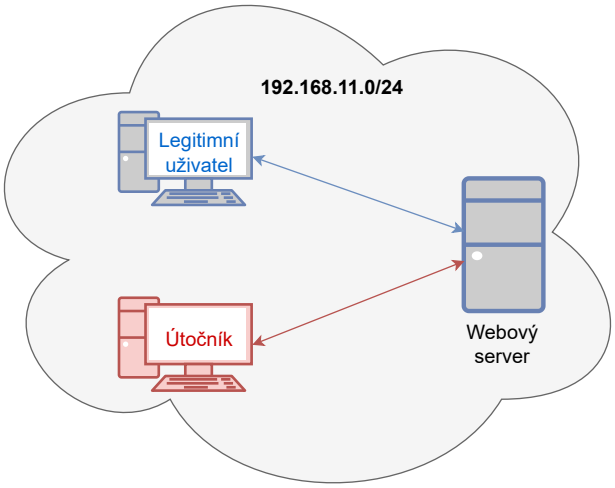
Legitimní uživatel využívá distribuci Linuxu Ubuntu ve verzi 20.04.2, má přidělenou paměť RAM 6 GB a slouží jako simulace běžného provozu pro přístup na webový server. Webový server je vytvořen na verzi Ubuntu 20.04.2 s přidělenou pamětí 4 GB RAM bez grafického rozhraní a běží na něm webový server Apache ve verzi 2.4.41, který je konfigurován ve výchozím nastavení, což znamená:

- Timeout je nastaven na 300 sekund,
- KeepAlive je povolen,
- maximální počet spojení je 100,
- KeepAliveTimeout je nastaven na 5 sekund,
- MaxRequestWorkers je nastaven na 150.

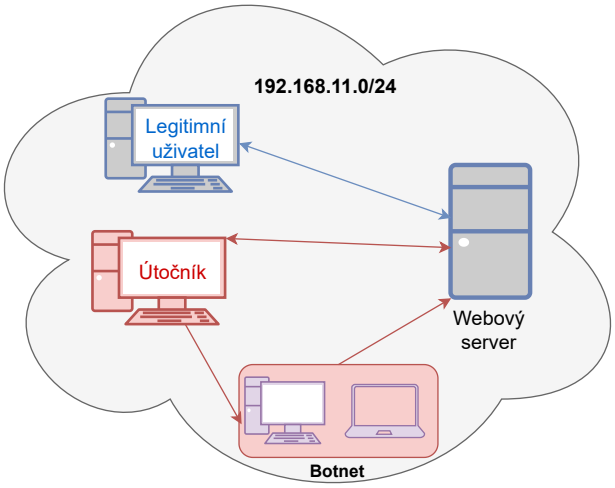
Další komponentou je útočník, který používá linuxovou distribuci Kali, ve verzi 2021.3 a je mu přiděleno 8 GB paměti RAM. V rámci scénáře s využitím sítě botnet se v síti dále nachází libovolný počet zařízení vytvořených (pomocí virtualizace[1]) jak s operačním systémem v distribuci Kali, tak Ubuntu. Schéma laboratorního prostředí lze rozdělit na scénář, kdy útočník použije pouze jedno zařízení, což lze

[1] Oba dva scénáře byly testovaný i v reálné lokální síti.

vidět na obrázku 6.1, a dále na scénář, kdy útočník využije více zařízení, která se budou chovat jako součást sítě botnet, což lze opět vidět na obrázku 6.2.



Obr. 6.1: Ukázka laboratorního prostředí s jedním zařízením.



Obr. 6.2: Ukázka laboratorního prostředí se sítí botnet.

6.2 Generátor Slow (D)DoS útoků

Generátor Slow DoS útoků je vytvořen v jazyce Python verze 3.9 a jako vývojové prostředí byl použit pyCharm. Grafické rozhraní je vytvořeno pomocí modulu tkinter, který je přímo navržen pro programovací jazyk Python a je zároveň i šířen pod licencí Python License. Původně bylo grafické rozhraní vyvíjeno v toolkitu PyQt5, ale vzhledem ke snadnějšímu dynamickému generovaní grafických prvků bylo od toolkitu upuštěno, a to i za cenu ztráty výrazně větší atraktivnosti GUI jako takového. Tkinter však umožňuje vytvoření instance třídy ttk, která umožní přistupovat k volitelným stylům, které jsou jak oficiálně předvytvořeny pro jednotlivé operační systémy (lze zjistit metodou *style.theme_names()*), tak je možno využít volně dostupných stylů vytvořených třetími stranami.

V rámci této práce byl využit styl **Azure theme** od uživatele **rdbende**, který je volně dostupný pod licencí MIT[2]. Azure byl původně vyvinut pro operační systém Mac OS, nicméně většina jeho upravených a i kompletně nových komponentů se dá použít i pro operační systém Linux, využívaný v této diplomové práci. Jedním ze stěžejních cílů práce je vytvoření grafického rozhraní, které bude vykreslováno co možná nejvíce automaticky, aby byla zajištěna modularita daného řešení. V následující podkapitole bude popsán návrh modelu, který tuto funkcionalitu co nejvíce umožní.

[2]Licence MIT je svobodná licence, která vznikla na Massachusettském technologickém institutu.

V této diplomové práci jsou v generátoru obsaženy tyto útočné skripty:

- SlowDrop,
- SlowDosGen,
  - Slow Next,
  - Slowcomm.
- SlowHTTPtest,
  - Slowloris,
  - Slow HTTP POST,
  - Slow Read,
  - Apache Range Header.
- Slowloris.

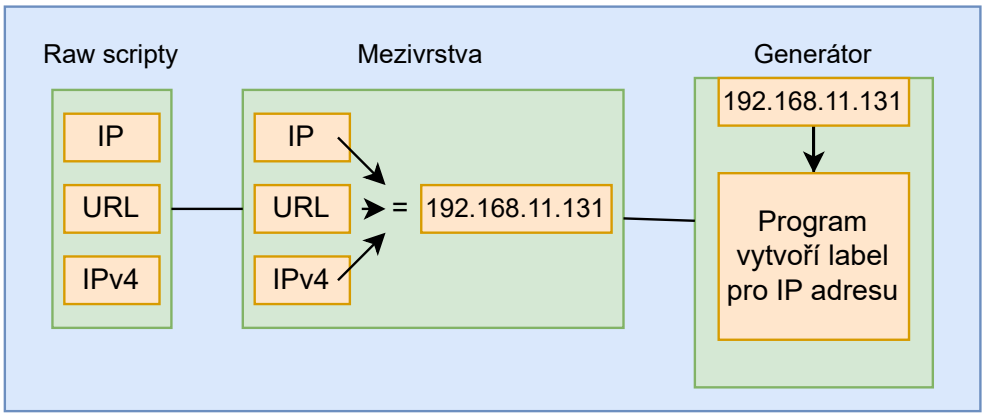
Avšak je potřeba zmínit, že díky řešení, které je zmíněné v kapitole 6.3.1 je možné velmi jednoduše přidávat další skripty s různými útoky a zachovat jejich veškerou funkcionalitu.

6.3 Modulární návrh grafického rozhraní

V první fázi návrhu grafického rozhraní bylo prvotně potřeba vymyslet, jakým způsobem bude program získávat od jednotlivých skriptů, či případně dalších komplexnějších programů potřebné proměnné a další informace, které bude nutno případně vykreslit. Dále bylo nutno prozkoumat jednotlivé skripty či programy a zjistit v čem se podobají, a nebo naopak liší, a to především pro využití dostatečné modularity. V ideálním případě by všechny skripty byly napsány podle jednotného vzoru a stačilo by pouze nasadit jakési obecné grafické rozhraní, které by bylo postaveno například na výpisu nápovědy těchto ideálních skriptů, kdy by vypsané argumenty byly automaticky převedeny do grafického rozhraní, a to pokaždé stejně. Již při zkoumání dodaných skriptů bylo jasné, že bude platit pravidlo „co autor, to jiný přístup“. Pro jednoduchou demonstraci, skript A obsahoval argument -ip, což značilo IP adresu cíle a druhý skript označoval stejný argument -URL.

Vzhledem k výše zmíněnému problému vznikla potřeba jakési mezivrstvy, která by umožnila převedení lišících se hodnot (například již zmíněných argumentů -ip a -URL) tak, aby vyvíjený program uměl vytvořit kýžené grafické rozhraní. Při návrhu takovéto mezivrstvy bylo potřeba navrhnout její formát a také způsob, jakým bude uživatel s takovou mezivrstvou nakládat. Jednoduchý model logiky takové mezivrstvy lze vidět na modelu na obrázku 6.3 Pro formát popisu mezivrstvy se nabízely v podstatě dvě možnosti. První možnost byl formát YAML[3], který nabízí trochu snazší čitelnost pro člověka, ale zároveň menší výkon než další zvažovaný

[3]YAML (Ain't Markup Language) je formát pro serializaci strukturovaných dat.



Obr. 6.3: Návrh modelu s mezivrstvou.

formát JSON[4]. Hlavní výhodou kromě již zmíněného výkonu JSONu je jeho mnohem snazší strojová čitelnost, která souvisí i s lepší dekódovatelností dat v Pythonu, které bude v tomto návrhu dále použito. Z výše zmíněných důvodů byl proto pro formát mezivrstvy vybrán JSON.

Ve způsobu, jak bude s JSONem uživatel nakládat (nyní už víme že mezivrstva bude reprezentována formátem JSON), se opět nabízela dvě řešení. První možnost byla JSON vždy vložit do daného skriptu, což by znamenalo, že uživatel by musel JSON upravovat přímo ve zdrojovém kódu skriptu a druhá možnost pak byla vytvořit složku v rámci projektu, ve kterém by se nacházely pouze soubory ve formátu JSON se jménem jednotlivých skriptů a uživatel by tak upravoval/vytvářel pouze dedikovaný soubor, který by si následně program našel a vykreslil dle něj grafické rozhraní (více níže). Rozhodnutí nakonec padlo na dedikovanou složku s JSON soubory.

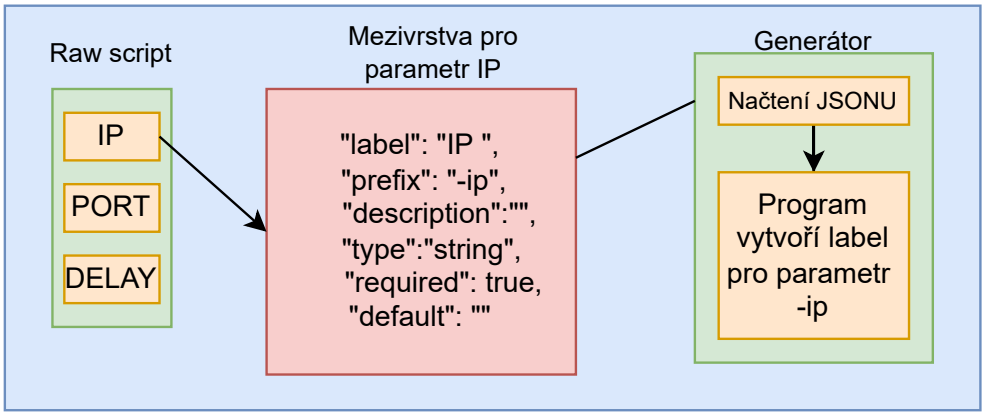
6.3.1 Konfigurační soubor JSON

V předchozí podkapitole byl popsán důvod a výhody, proč byl nakonec vybrán formát JSON. V této podkapitole bude převeden model zmíněný na obrázku 6.3 do prakticky využívaného modelu a dále popsán jeden z konfiguračních souborů, využívaných v této práci.

V modelu na obrázku 6.3 se popisuje počáteční problém, kdy jsou dodány různé skripty a parametry jsou označeny pokaždé jiným klíčovým slovem. V návrhu modelu, který lze vidět na obrázku 6.4, se již počítá s řešením, které bylo popsáno v předchozí podkapitole, což znamená, že místo více skriptů se v tomto modelu pracuje pouze s jedním skriptem a pro tento ukázkový případ pouze s jedním jeho

[4]JSON (JavaScript Object Notation) je způsob zápisu dat nezávislý na počítačové platformě, určený pro přenos dat, která mohou být organizována v polích nebo agregována v objektech.

argumentem. Ve výpise 6.1 lze vidět část (celý konfigurační soubor je příliš velký)



Obr. 6.4: Návrh modelu s konfiguračním souborem JSON pro parametr IP.

konfiguračního souboru skriptu SlowDoSGen. Konfigurační soubor je ve formátu JSON a Python na něj nahlíží jako na vnořené slovníky[5]. Na výpise 6.1 lze vidět základní (v průběhu práce budou popsány další) předdefinované objekty, se kterými bude dále pracovat program vyvíjený v této práci a pro další popis zdrojového kódu je důležité tyto objekty z výpisu 6.1 zmínit a popsat:

- description - vyskytuje se na dvou úrovních slovníku a v prvním případě (řádek č. 2) popisuje název konkrétního útoku jako takového a v druhém případě (například řádek č. 6) popisuje konkrétní účel daného parametru.
- detail_description - na řádku č. 3 je detailní popis skriptu přidávaného skriptu, který je uživateli zobrazen ve framu „Choose scripts“ což lze vidět na obrázku 6.5.
- label - vyskytuje se například na řádku č. 7 a slouží ke stanovení názvu, který bude vykreslen pro daný parametr.
- prefix - důležitý objekt, který se nachází na řádku č. 9 popisuje název parametru, který se reálně používá pro definici hodnoty daného skriptu.
- type - nachází se na řádku č. 10 a popisuje typ, o který se jedná aby program dokázal s takovým typem patřičně naložit.
- required - jak již název napovídá, tento objekt vyskytující se na řádku č. 15 nabývá boolean hodnotu a udává, zda je daný parametr povinný pro fungování daného skriptu, typicky je to u každého skriptu hodnota IP adresy.
- default - objekt nacházející se na řádku č. 23 vyplňuje předefinovanou hodnotu přímo do grafického rozhraní.

[5]Slovníky se používají k ukládání datových hodnot v párech klíč:hodnota, a jsou to kolekce, které jsou uspořádané, měnitelné a neumožňují duplicity.

Výpis 6.1: Část konfigurační souboru JSON pro skript slowDoSGen.

```
{
"description": "SlowDoSGen",
"detail_description": "Detailní popis skriptu (V tomto případě
   SlowDosGen)",

    "params": [
        {
            "label": "Attack",
            "description":"C = Slowcomm OR N = Slow Next",
            "prefix": "-a",
            "type":"options",
            "options":[
                "C",
                "N"
            ],
            "required": true
        },
        {
            "label": "IP address",
            "prefix": "-ip",
            "description":"Target IP address or URL",
            "type":"string",
            "required": true,
            "default": "192.168.1.1"
        },
        {
            "label": "Connection",
            "prefix": "-c",
            "description":"Number of connections to established",
            "type":"numeric",
            "required": true,
            "default": "10"
        },
        {
            "label": "Port",
            "prefix": "-p",
            "description":"destination port",
            "type":"numeric",
            "required": true,
            "default": "80"
        }
    ]
}
```

6.3.2 Práce s konfiguračním souborem ve zdrojovém kódu

V této podkapitole bude detailně popsáno samotné fungování vyvíjeného programu při práci s konfiguračním souborem a také s komponenty knihovny tkinter. Na výpise 6.2 je nejdříve definován slovník *disovery_list* a pole *main_list*. Následně je na na řádku č. 5 zavolána funkce *find_scripts*, kterou lze vidět na výpise 6.3, která jednoduše prohledá dedikovanou složku scripts, jež je lokalizována jako podsložka ve stromové struktuře, odkud je hlavní program spouštěn, vytvoří seznam (list) všech souborů, které končí koncovkou .json, následně je vypíše do konzole a vrátí je volané funkci z výpisu 6.2. Seznam nalezených souborů je pak v cyklu postupně zpracováván voláním funkce *discover_scripts* z výpisu 6.4, která se pokusí načíst každý z nalezených souborů JSON a zkontroluje, jestli je soubor čitelný (možný problém s poškozeným souborem, či správnými právy ke čtení souboru) a jestli soubor obsahuje validní JSON formát. Pro tento účel se používá vestavěná podpora JSON formátu v jazyce Python, která je následně v dalších částech i hojně používána pro serializaci a deserializi objektů při síťové komunikaci mezi jednotlivými komponentami představeného řešení simulace. Pokud je soubor v pořádku načten, jsou adekvátně aktualizovány proměnné *disovery_list* a *main_list*.

V další části program načítá soubor configuration.json, který je povinně očekáván v adresáři, ze kterého je spouštěn hlavní program a obsahuje zejména seznam IP adres a portů botů pro samotnou DDoS simulaci.

Výpis 6.2: Začátek programu pro vykreslení GUI.

```
1 ;# discovery list is the dictonary of the scripts self discoveries
2 discovery_list = {}
3 # list is the simple list of the scripts discovered and correctly
     parsed (JSONs)
4 main_list = []
5 for s in find_scripts():
6     j = discover_script(s)
7     if j != None:
8         discovery_list[get_description(j)] = j
9         list.append(get_description(j))
```

Výpis 6.3: Funkce find_scripts.

```
1 def find_scripts() -> list:
2     print("Searching for scripts ...")
3     scripts_dir = f"{os.path.dirname(os.path.realpath(__file__))}/
         scripts/"
4     scripts = glob.glob(f"{scripts_dir}*.json")
5     print(f"Scripts found: {scripts}")
6     return scripts
```

Výpis 6.4: Funkce discover_script.

```
def discover_script(script):
    try:
        json_file = open(script)
    except:
        print("File could not be read", script)
        return
    try:
        j = json.load(json_file)
    except:
        print("JSON is not valid or file could not be read", script
            )
        return
    j["script"] = script
    return j
```

6.3.3 Práce s načtením konfiguračního souboru ve zdrojovém kódu

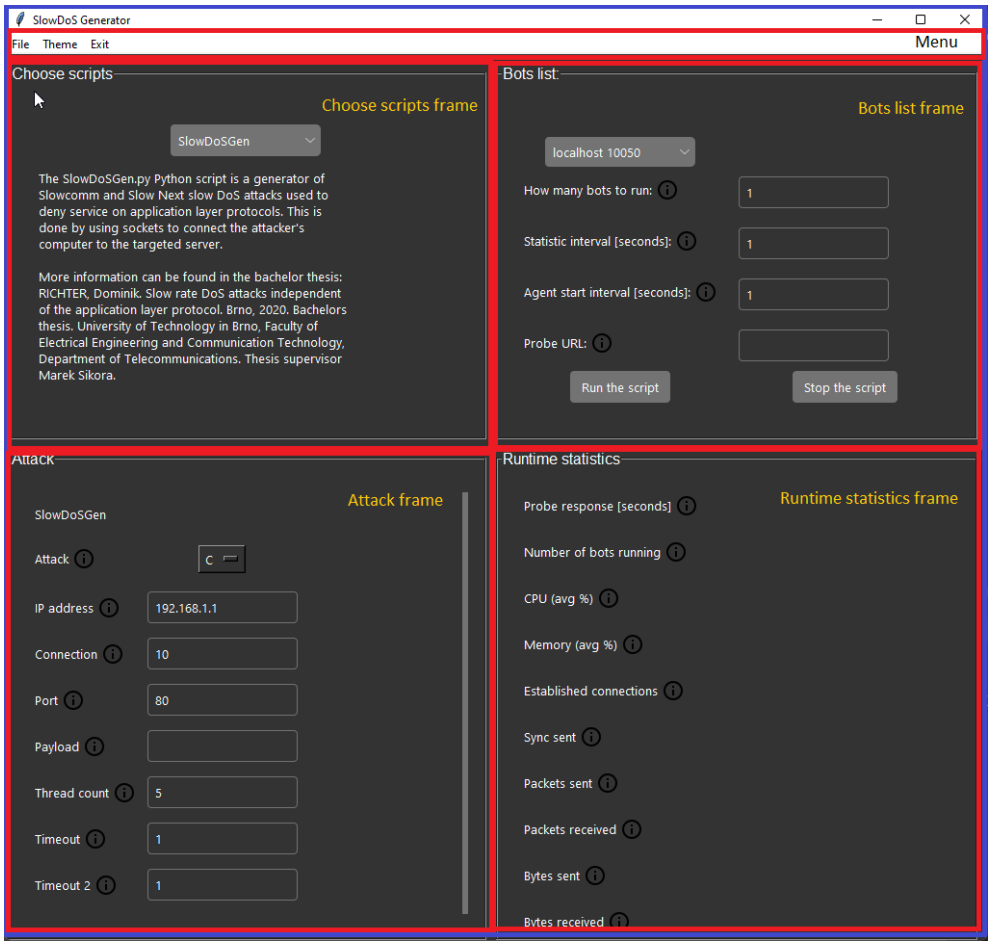
GUI je logicky rozděleno do několika logických částí, které jsou implementovány jako objekty typu frame, což je znázorněno na obrázku 6.5 a podrobněji popsáno v kapitole 6.8.1. Stěžejní částí frontend řešení je schopnost GUI dynamicky a programově zpracovat načtené definice, jak bylo popsáno v kapitole 6.3.2. Pokud byla v předchozím běhu programu nalezena validní konfigurace, obsahuje combobox základní popis jednotlivých skriptů, načtených z kliče *description* a je pro něj vyrenderováno příslušné GUI.

Samotný rendering probíhá ve funkci *populate_frame*, která dostává jako vstupní parametry slovník input a odkaz na frame, do kterého se mají grafické prvky renderovat. V první části funkce se prochází případné „child“ prvky z předchozí definice a volá se metoda *destroy*, čímž se frame resetuje do výchozího stavu.

Výpis 6.5: Funkce populate_frame.

```
def populate_frame(input: dict, frame: ttk.Frame):
    global submit_run
    global cancel_run
    for widget in frame.winfo_children():
        widget.destroy()
    s = json.loads(json.dumps(input))

    desc = ttk.Label(frame, text=s["description"])
    varList = populate_vars(s['params'], frame)
    i = 0
    for p in s['params']:
```



Obr. 6.5: Základní rozdělení GUI.

```
        if "target" in p:
            s["target"] = i
        if "label" in p:
            l = ttk.Label(frame, text=p["label"],
                          image=_photo, compound=tk.RIGHT)
            button1_ttp = CreateToolTip(l, text=p["description"])
        if p["type"] == "options":
            o = p["options"]
            t = tk.OptionMenu(frame, varList[i], *o)
            varList[i].set(o[0])
        else:
            if "value" in p:
                st = 'readonly'
            else:
                st = 'normal'
            t = ttk.Entry(frame, textvariable=varList[i], state=st)
            t.grid(row=1+i, column=1)
        i = i + 1
```

Dále se vytváří kopie ze slovníku input a načítá se do proměnné *s*. Pak je volána funkce *populate_vars*, která je znázorněna na výpise 6.6, jejímiž vstupními argumenty jsou slovník s popisem parametrů a frame. Funkce vrací seznam proměnných typu StringVar[6], jež reprezentují konkrétní vstup a jsou navázány na příslušný widget (prvek GUI). Pokud existují klíče *default*, nebo *value*, jsou generované proměnné inicializovány na příslušnou hodnotu.

Výpis 6.6: Funkce populate_vars.

```
def populate_vars(v, frame):
    lv = []
    i = 0
    for p in v:
        val = None
        if "default" in p:
            val = p["default"]
        if "value" in p:
            val = p["value"]
        if p["type"] == "numeric":
            lv.append(StringVar(frame, value=val))
        elif p["type"] == "string":
            lv.append(StringVar(frame, value=val))
        elif p["type"] == "options":
            lv.append(StringVar(frame, value=val))
        i = i+1
    return lv
```

[6]StringVar pomáhá efektivněji spravovat hodnoty widgetů v tkinteru.

Poté se v cyklu zpracovávají prvky pole *params* proměnné *s*, přičemž každý takový prvek obsahuje zobecněný popis parametru volaného skriptu a je pro něj renderována příslušná GUI prezentace (widget) a jsou nastaveny jeho další vlastnosti. Nejdříve se testuje, jestli daný parametr reprezentuje cíl útoku, což je vyjádřeno existencí klíče *target*. Pokud ano, je index (pořadí) tohoto parametru vložen jako klíč *target* do proměnné s. Přesný význam a použití klíčového slova target je vysvětleno v další části práce.

Následně se zjišťuje, jestli daný parametr obsahuje klíč *label* a pokud ano, renderuje se jako popis pro daný vstup, zároveň se vytvoří tooltip[7] s detailnějším vysvětlením. Následuje logická podmínka, která detekuje, jestli se daný parametr volí se seznamu možných parametrů a v takovém případě je renderováno *OptionMenu*, nebo se jedná o volný vstup, pak je renderováno textové pole. Pro oba případy je widget referencován k příslušnému objektu StringVar, získaného dříve popsanou funkcí *populate_vars*. Pokud je parametr opatřen klíčem *value*, předpokládá se, že jde o pevně definovanou hodnotu a textové pole je jen ke čtení.

6.3.4 Výhody renderingu oproti statické definici představené v semestrální práci

Hlavní výhodou renderingu GUI na základě obecné definice oproti statické definici, představené v semestrální práci je úplné oddělení kódu a vstupních dat, což bylo jedním z úkolů diplomové práce. Tato skutečnost ušetří obrovské množství kódu, které je nutno napsat a případně pro další rozšíření i pochopit. Pro porovnání se semestrální prací, kdy bylo GUI převážně statické a vytvořené vždy na míru jednotlivému skriptu dosahoval počet řádků pouze na vytvoření základních komponentů zhruba 400 řádků. Zatímco v diplomové práci bylo řádků potřeba pouze okolo 40 na základní komponenty. Na tento fakt navazuje další výhoda dynamického renderingu GUI a to jeho modularita, která de facto umožňuje jednoduše přidat neomezené množství skriptů pomocí již zmíněného konfiguračního souboru JSON. Poslední výhodou je pak modularita této části práce, která umožňuje přenositelnost logiky vytvořeného kódu a částečně i kód jako takový, na jakýkoliv další projekt v Pythonu, který bude vyžadovat podobně vytvořené grafické rozhraní.

6.3.5 Problémy při řešení dynamicky načítané konfigurace

Největším problémem při řešení dynamicky načítané konfigurace bylo především nalézt správnou míru zobecnění, tak aby byl formát vstupu vhodně strukturovaný

[7]Zpráva, která se zobrazí po umístění kurzoru na ikonu, obrázek, hypertextový odkaz nebo jiný prvek v grafickém uživatelském rozhraní.

a umožnil popsat všechny nutné parametry a zároveň nebyl příliš komplikovaný.

Tím, že byl zvolen formát JSON, může se jevit jako problém dodržení striktního formátu konfigurace, který JSON vyžaduje k tomu, aby byla následně možná deserializace. Existuje řada validátorů a nástrojů[8], které tento problém pomáhají vyřešit a ověřit si správný formát. Jelikož je ve standardním formátu JSONu akceptována čárka za posledním elementem pole, ale Python tento formát implementace neakceptoval a docházelo k chybě s načtením při načítaní, kterou dělalo problém identifikovat.

6.4 Rozšíření o HTTPS

Ne všechny skripty v současné sadě skriptů s útoky, podporují útok na TLS (SSL) endpoint pomocí protokolu HTTPS. Oficiální slowHTTPtest tuto možnost samozřejmě obsahuje, ale další skripty, napsané v jazyce Python ne. Cílem bylo zanalyzovat, jakým způsobem by se tyto skripty o možnost komunikace s HTTPS rozšířit daly a prakticky tuto možnost implementovat.

Vzhledem k faktu, že se velmi často při zneužívání zranitelnosti web serverů používá nestandardní komunikace, jsou všechny zmiňované skripty postaveny na použití interního modulu sockets jazyka Python, který realizuje komunikaci na transportní vrstvě modelu OSI. Jazyk Python nabízí poměrně snadno použitelný modul ssl, využívající OpenSSL knihovnu k přístupu k TLS komunikaci a to jako „wrapper“ stávajícího socketu, takže jeho další použití je transparentní. Tento přístup byl prakticky implementován a otestován ve scriptu *slowLoris.py*. Na výpisu 6.7 lze vidět původní verzi zdrojového kódu, bez úpravy HTTPS, která je naopak zobrazena na výpise 6.8. Logika úpravy kódů by pak byla u všech skriptů velmi podobná.

Výpis 6.7: Původní verze bez HTTPS.

```
1 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
2 s.settimeout(4)
3 s.connect((ip,int(port)))
```

Výpis 6.8: Upravená logika s HTTPS.

```
1 init_s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
2 init_s.settimeout(4)
3 s = ssl.wrap_socket(init_s)
4 s.connect((ip,int(port)))
```

[8] V této práci využíván `https://jsonformatter.curiousconcept.com`.

6.5 Realizace DDoS rozšíření

Kromě výše popsané frontendové části, která umožňuje parametrizovat a spustit (případně zastavit) DoS útok jako celek, bylo nutné navrhnout a vyvinout backendovou část, která na základě požadavku z frontendové části bude samotný běh řídit, škálovat, sbírat data o běhu a ukládat jejich výsledky k pozdější analýze a tento koncept dále ještě rozšířit o koncept, kde bude možné útoky rozšířit o variantu DDoS a zároveň zachovat veškeré funkcionality, které jsou využity pří útoku DoS. Pro rozšíření o distribuovanou verzi byly na stole v podstatě dvě možnosti, kdy každá přinášela značné kompromisy především k zadání této diplomové práce.

První zvažovanou možností bylo vytvoření simulace na základě ARP spoofingu (tato simulace DDoS útoku je detailně popsána v práci [33]), kde je de facto lokální sít přesvědčena o existenci dalších botů s jinými IP adresami v této síti, které následně DDoS útok na cíl simulují. V tomto případě, se dosáhne úplné simulace, ale pouze za předpokladu, že útoky, které jsou přidány do programu vyvíjeného v této práci jsou v plné moci vývojáře. Vzhledem k tomu, že jsou do programu přidávány i útoky třetích stran, které jsou vyvinuty v jiných programovacích jazycích, by došlo ke ztrátě modularity, jelikož by byl pokaždé nutný zásah do kódu daného útoku.

Druhá možnost byla vymyšlena v rámci této diplomové práce a jedná se o řešení založené na principu master-slave, která je reálně používána při provádění distribuovaného DoS útoku a inspirace tohoto řešení pochází z programu JMeter[9]. V rámci diplomové práce byla nakonec vybrána druhá varianta, tedy model master-slave, o kterém je více řečeno jak z hlediska návrhu a logiky, tak z hlediska samotné implementace v následujících podkapitolách.

6.5.1 Koncept návrhu řešení

Koncept realizovaného řešení vychází ze samotné podstaty DDoS, kdy je stejný útok realizovaný stejným skriptem z libovolného množství počítačů v síti. Byl tedy zvolen model, kdy master modul komunikuje po TCP/IP síti s libovolným počtem běžících botů, přičemž umožňuje opakovaně spouštět různé, nebo různě parametrizované skripty, a dostávat zpět data z těchto botů.

Zde je tenká hranice mezi tím, co lze nazývat simulací a co je reálným DDoS, ale bylo vycházeno z předpokladu, ze pokud se jedná o laboratorní situaci, jejíž účelem není pouze daný útok provést a dosáhnout odepření služeb napadeného serveru, případně clusteru serverů, nýbrž celý proces řídit, komunikovat s jednotlivými instancemi, reálně skript spouštět, zpracovávat výkonnostní data z těchto instancí

[9] Apache JMeter je projekt Apache, který lze použít jako nástroj pro testování zátěže pro analýzu a měření výkonu různých služeb se zaměřením na webové aplikace.

v reálném čase a následně post-proces získat a uložit další výstupy, lze takový koncept jako simulaci označit. Navíc je možné tuto laboratorní simulaci realizovat instancemi botů, bežícími v kontejnerech, nebo za pomocí cloudových služeb. Předpokladem pro úspěšné provedení DDoS je, že bot proces běží, je dostupný master modul po TCP/IP síti, a že má bot lokálně k dispozici spouštěný skript.

6.5.2 Popis protokulu master-slave

Jak již bylo zmíněno, master modul komunikuje během simulace se sítí slaves (v této práci bude dále slave nazýván botem). Bylo proto nutné navrhnout protokol této komunikace tak, aby splňovala potřebnou funkčnost. Jako základ komunikačního protokolu byl opět zvolen JSON formát a to pro již několikrát zmiňovanou rozšířenost, obecnou použitelnost a s tím rovněž spojenou spolehlivě ověřenou implementaci v jazyce Python. Z posledně jmenovaného plyne rovněž nativní podpora pro serializaci a deserializaci objektů. Jazyk Python umožňuje tato operace pro základní datové type provádět velmi snadno, čehož bylo využito při definici protokolu. Jedná se v podstatě o JSON interpretaci Python objektů typu dictionary (slovník).

Komunikace je vždy inicializována z modulu master a je typu žádost-odpověď. Detaily implementace z hlediska algoritmu budou popsány v následujících kapitolách, nyní však následuje popis protokolu samotného.

Protokol definuje sadu požadavků, které, jak již bylo zmíněno, vždy inicializuje modul master. Momentálně jsou implementovány:

- **ping**,
- **run**,
- **status**,
- **stop**,
- **details**,
- **exit**.

Ping

Účelem požadavku ping je zjistit, zda bot v pořádku běží a na daný požadavek odpoví *unix timestampem* s aktuálním časem na počítači, na kterém běží. Rovněž je u všech odpovědí přítomen klíč agent, který identifikuje bot. **požadavek:**

```
1 {"command": "ping"}
```

odpověď:

```
1 {"result": <unix_timestamp>,
2  "agent": {
3     <IP>,
4     <port>
5 }}
```

Run

Požadavek *run* obsahuje všechna data, která potřebuje bot k tomu, aby spustil daný skript s útokem. V prvé radě je to povinný klíč *cmd_to_run*, který obsahuje pole s příkazem samotným a jeho argumenty, dále nepovinné klíče *target* a *post_process*. Target obsahuje řetězec s identifikací cíle daného útoku. De facto to může být URL, host, nebo IP adresa. Klíč *post_process* obsahuje objekt s klíčem *cmd* s příkazem, který se má provést po ukončení útoku a pole řetězců download se seznamem souborů, které se mají po ukončení stáhnout do řídícího programu (viz 6.9). Bot odpoví objektem *response*, který obsahuje klíč *message*, případně PID (Process IDentifier), nebo error. Klíč *message* identifikuje stav po pokusu o spuštění skriptu na botu. Může nabývat hodnot:

- AlreadyRunning - pokud je na botu již nějaký skript spuštěn,
- Started - pokud spuštění proběhlo v pořádku,
- Failed - pokud spuštění selhalo.

V případě úspěšného spuštění je klíč *PID* naplněn PID spuštěného procesu, naopak pokud se spuštění nepodaří, klíč *error* obsahuje detaily dané chyby.

požadavek:

```
{"command": "run",
 "cmd_to_run": [
   <command>,
   <arg1>,
   ...
   <argn>
 ],
 "target": <string>,
 "post_process": {
   "cmd": <string>,
   "download": [
            <string>,
            ...
            <string>]
}}
```

odpověď:

```
{"result": <unix_timestamp>,
 "agent": {
    <IP>,
    <port>
}}
```

Status

Požadavek *status* slouží k vyžádání aktuálního stavu skriptu, běžícího na botu. Bot odpoví objektem *response*, který obsahuje klíč *message* s aktuálním statusem a aktuálními výkonnostními statistikami v klíči *current_stats*. Klíč *message* může nabývat hodnot:

- IsRunning - pokud skript aktuálně běží,
- NotRunning - pokud skript již neběží (skončil, nebo byl ukončen),
- NeverRunning - pokud skript neběží a ještě nebyl učiněn pokus o jeho spuštění.

Jestliže je proces ve stavu IsRunning, jsou v klíči uloženy *current_stats* hodnoty:

- timestamp - aktuální čas v okamžiku odečtení statistik,
- cpu - CPU, spotřebované spuštěným skriptem v procentech,
- memory - paměť používaná běžícím skriptem,
- connections - aktuální stav connections pro daný skript. Obsahuje seznam rozdělený podle stavu connection (odpovídající stavům příkazu netstat, tzn. ESTABLISHED, SYN_SENT, TIME_WAIT),
- memory - nicstat - statistiky síťového rozhraní (bytes_sent, bytes_recv, packets_sent, packets_recv, errin, errout, dropin, dropout). Jde o relativní

údaje od startu skriptu (v okamžiku start se uloží aktuální stav a ten je pak odečten při každém volání požadavku status).

požadavek:

```
{"command": "status"}
```

odpověď:

```
{"response": {
  "message": <string>,
  "current_stats": {
  "timestamp": <unix_timestamp>,
  "cpu": <float>,
  "memory": <float>,
  "connections": {
    <string>: <int>,...
}
  "nicstat": [
   <value>,...
  ],
}
},
 "agent": {
    <IP>,
    <port>
}}
```

Stop

Požadavek stop slouží k ukončení běžícího skriptu. Odpověď obsahuje klíč *message*, jenž může nabývat hodnoty:

- Terminated - pokud je proces s běžícím skriptem úspěšně zastaven,
- NotRunning - pokud skript již neběží (skončil, nebo byl ukončen),
- NeverRunning - pokud skript neběží a ještě nebyl učiněn pokus o jeho spuštění.

požadavek:

```
{"command": "stop"}
```

odpověď:

```
{"message": <string>,
 "agent": {
    <IP>,
    <port>
}}
```

Details

Požadavek details slouží především k získání a přenesení dat po zastavení skriptu.

Odpověď obsahuje klíč *message*, který může nabývat hodnot:

- NoData - pokud skript neběží a ještě nebyl učiněn pokus o jeho spuštění,
- IsRunning - pokud aktuálně běží,
- Finished - pokud byl skript již ukončen.

V případě, kdy má klíč *message* hodnotu Finished, je vložen objekt *process_details*, který obsahuje:

- PID - Process ID, které měl proces skriptu během svého běhu,
- returncode - return code procesu po jeho ukončení,
- stdout - obsah standardního výstupu procesu,
- stdout - stderr - obsah standardního chybového výstupu procesu,
- stdout - download - pole obsahů souborů, vyžádaného požadavkem *run.post_process.download*.

požadavek:

```
{"command": "details"}
```

odpověď:

```
{"message": <string>,
 "process_details": {
   "PID": <int>,
   "returncode": <int>,
   "stdout": <string>,
   "stderr": <string>,
   "download": [<string>,...]
}
 "agent": {
    <IP>,
    <port>
}}
```

Exit

Požadavek exit jednoduše ukončuje běh bota. Odpověď obsahuje klíč *message* s hodnotou *exit*.

požadavek:

```
{"command": "exit"}
```

odpověď:

```
{"message": "exit",
 "agent": {
    <IP>,
```

```
4       <port>
5   }}
```

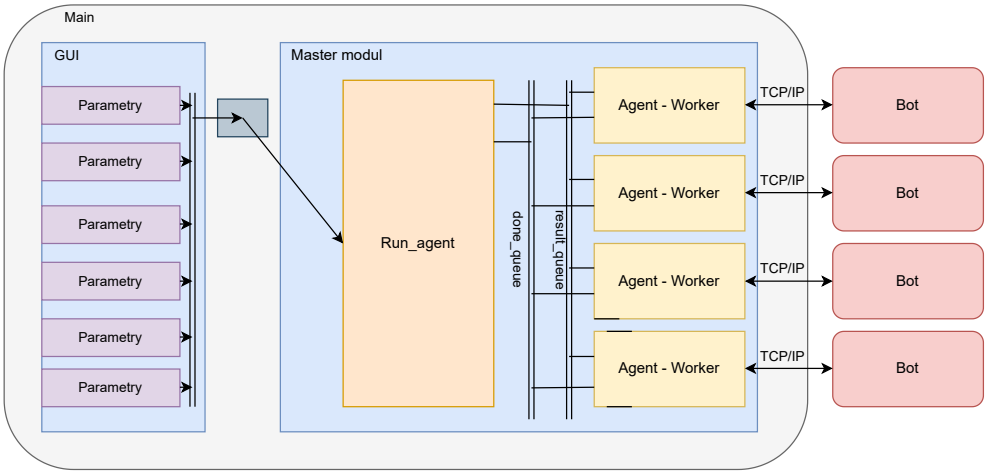
6.5.3 Problémy při řešení protokolu master-slave

Problémem při řešení protokolu master-slave bylo především najít sadu požadavků a jejich odpovědí tak, aby byla pokryta všechna požadovaná a navržená funkčnost. Dále pak bylo nutno vyřešit problém, který se netýkal logické stránky protokolu, ale spíše komunikační. Jedna z prvních verzí programu, který implementoval popisovaný protokol fungovala bezchybně při testování na lokálním síťovém rozhraní (localhost), ale vykazovala chybovost při pozdějším testování na několika botech na jiných počítačích v síti. Analýza ukázala, že dochází k chybám při deserializaci odpovědí do objektů JSON a to proto, že nebyly přeneseny celé. Tento problém byl následně identifikován jako důsledek faktu, že byl zvolen typ komunikace na relační vrstvě modelu OSI, pomocí implementace sockets v jazyce Python a tudíž při komunikaci s vyšší latencí docházelo k předčasnému ukončení relace.

Problém byl nejdříve řešen snahou o nastavení parametru timeout na straně sockets a velikostí bufferu, což v mnoha případech problém zcela eliminovalo, nikoliv však spolehlivě a definitivně, a to především u většího množství přenášených dat. Nakonec bylo finálním řešením přidání velikosti přenášených dat v bytech jako prefix samotné JSON reprezentace. Tato velikost je známa na straně odesílatele v okamžiku serializace objektů jazyka Python do formátu JSON a tudíž může být takto použita. Přijímající strana si tedy pak nejdříve načte očekávanou velikost přijímaných dat a poté pokračuje v komunikaci, dokud nejsou všechna data přenesena.

6.6 Programový model modulu master

Modul master je Python modul, obsahující kód k řízení DDoS útoku, to znamená především komunikaci se sítí botů, agregací přicházejících dat ze sítě a ukládání výkonnostních dat a výstupu z jednotlivých botů. Jako vstup dostává modul master skript, který má spustit společně se všemi jeho argumenty, list botů, na kterých má daný skript spustit, komunikační fronty (queues), kterými je synchronizována komunikace mezi moduly GUI a master a další argumenty, které jsou podrobně popsány v dalších kapitolách. Na obrázku 6.6 je graficky znázorněn programový model stěžejní části modulu master. Pro komunikaci s boty se používá nativní modul multiprocessing jazyka Python. Pro každou instanci bota je vytvořen *subprocess* a pomocí odkazu na dvojici objektů fronty, které jsou předány každému subprocesu, probíhá komunikace a synchronizace mezi jednotlivými subprocesy a řídícím procesem modulu master. Každý subproces řídí komunikaci s jedním botem a realizuje



Obr. 6.6: Model modulu master.

celý algoritmus. Průběžné výsledky jsou pak posílány prostřednictvím fronty (*result_queue*) zpět řídícímu procesu, druhou zmíněnou frontou *(done_queue)* naopak posílá řídící modul subprocesu požadavek na ukončení běhu skriptu na botu, pokud neskončí sám. Další detaily implementace včetně komentářů kódu jsou popsány v následující kapitole.

6.6.1 Popis algoritmu

Jak již bylo nastíněno, modul master obsahuje kód řídícího modulu a dále pak kód pro řízení jednotlivých botů.

run_agents

Řídící kód modulu se nachází ve funkci *run_agents*. Vstupními parametry funkce jsou:

- *agent_list*: list - pole adres botů, na kterých se má spustit skript,
- *status_queue*: *multiprocessing.Queue* - fronta, kterými řídící proces komunikuje s modulem GUI,
- *agg_results_queue*: *multiprocessing.Queue* - fronta, kterou řídící proces posílá modulu GUI průběžná agregovaná performance data,
- *task*: dict - skript s argumenty, který má být spuštěn,
- *results_path*: str - cesta souborového systému, kde mají být ukládána posbíraná data,
- *status_period*: int - interval v sekundách, definující, jak často mají být sbírána průběžná data,
- *start_period* : int - interval v sekundách definující dobu, během které je skript na všech botech spuštěn.

V první části funkce se inicializují fronty, které budou použity ke komunikaci se subprocesy a na základě vstupního parametru *start_period* a počtu botů se spočítá zpoždění mezi starty jednotlivých botů. Poté se v cyklu prochází list adres botů a pro každého z nich se spustí subproces s funkcí *result_queue* s příslušnými argumenty a odkaz je vložen do seznamu *procs*.

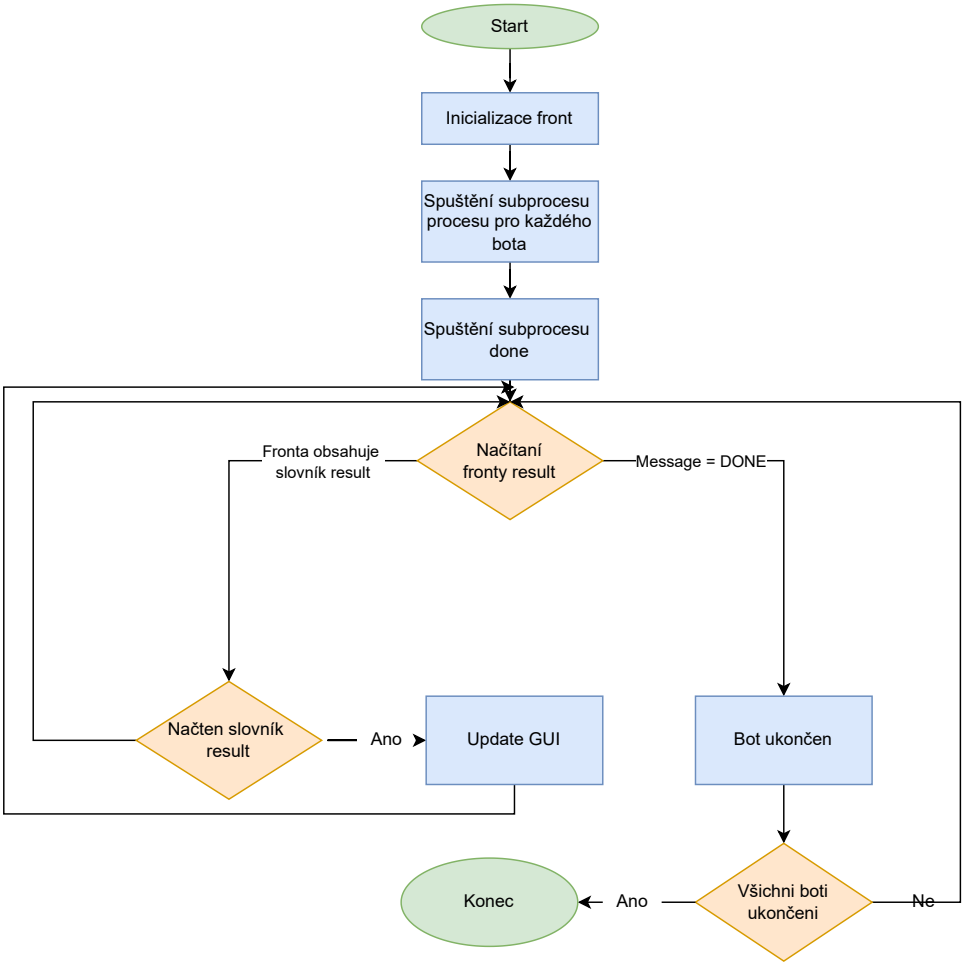
Rovněž je vložen inicializační záznam do slovníku *status_list*. Tento objekt má jako klíč adresu bota a jeho hodnotou je slovník, který obsahuje výkonnostní data, sbíraná během běhu skriptu na botech. Po proběhnutí cyklu následuje spuštění dalšího subprocesu (done), který je spuštěn s funkcí *done_func* a bude popsána podrobněji dále. Následuje nekonečný cyklus, který postupně vyčítá obsah *result_queue*. Zde (a i v dalších částech kódu) se využívá vlastnosti jazyka Python, který jako jiné dynamicky typované jazyky, umožňuje zpracovávat libovolné typy a struktury proměnných. Fronta *result_queue* je totiž použita nejenom pro sběr výkonnostních dat, ale bot ji používá i pro slovník s klíčem *message*. Pokud má klíč *message* hodnotu DONE, znamená to, že daný subproces, reprezentovaný funkcí *agent_worker*, ukončil svoji úlohu. To vede ke smazání jeho záznamu ve slovníku status_list a seznamu agent_list. Poté se zjišťuje, jestli je seznam botů již prázdný, což de facto znamená, že všechny instance subprocesu s funkcí *agent_worker* ukončily svoje úlohy, což vede k přerušení cyklu. V další části cyklu se testuje, zda se z fronty načetl slovník s klíčem *result*. Pokud ano, volá se funkce *update_gui*, která se stará o agregaci průběžných performance dat z jednotlivých botů a jejich zaslání řídícímu modulu GUI. Jakmile je výše popsaný cyklus přerušen, provede se ještě ukončení subprocesu done, pokud je to třeba a funkce končí. Zjednodušený diagram této funkce je zobrazen na obrázku 6.7.

agent_worker

Kód pro obsluhu komunikace mezi řídícím procesem a jednotlivými boty se nachází ve funkci *agent_worker*. Vstupními parametry jsou:

- *server_address*: tuple - reprezentuje IP a port bota,
- *command*: dict - skript s argumenty, který má být spuštěn,
- *result_queue*: *multiprocessing.Queue*,
- *done*: *multiprocessing.Queue*,
- *path_result*: str - cesta souborového systému, kam mají být ukládána posbíraná data,
- *period*: float - interval v sekundách definující dobu, během které je skript na všech botech spuštěn.

Veškerá síťová komunikace funkce *agent_worker* popsaná v následujících částech je ošetřena blokem *Try-Except*, aby nenarušila algoritmus spouštění skriptů a umožnila



Obr. 6.7: Diagram funkce *run_agents*.

korektní ukončení subprocesu, pokud dojde k chybě, kterou nelze ignorovat. Funkce *send_message*, která realizuje samotnou síťovou komunikaci, je popsána v dalším textu. V první fázi funkce se pomocí požadavku ping (6.5.2) zjišťuje, zda je daný bot dostupný. Pokud tomu tak není, je vypsána hláška do konzole a zároveň je do fronty *result_queue* zaslán slovník s klíčem *message* a hodnotou DONE, jehož význam je popsán v (6.6.1).

Poté následuje zaslání požadavku stop (6.5.2), aby byla vyloučena možnost, že bot stále provádí jakoukoliv aktivitu z nějaké předchozí relace. Pokud volání požadavku stop nezpůsobí výjimku, je v zápětí zaslán požadavek run (6.5.2) a provedena kontrola, zda proběhl na straně bota v pořádku. Pokud ne, je adekvátně funkce ukončena, včetně synchronizace pomocí zaslání slovníku s klíčem *message* a hodnotou DONE, jak bylo popsáné výše, a vypsání podrobnosti chyby do konzole. Další část funkce řeší samotný běh skriptu na botu. Jde vlastně o další nekonečný cyklus, který probíhá dokud běh skriptu samotného neskončí, nebo není zaslán požadavek o jeho přerušení z řídící vrstvy.

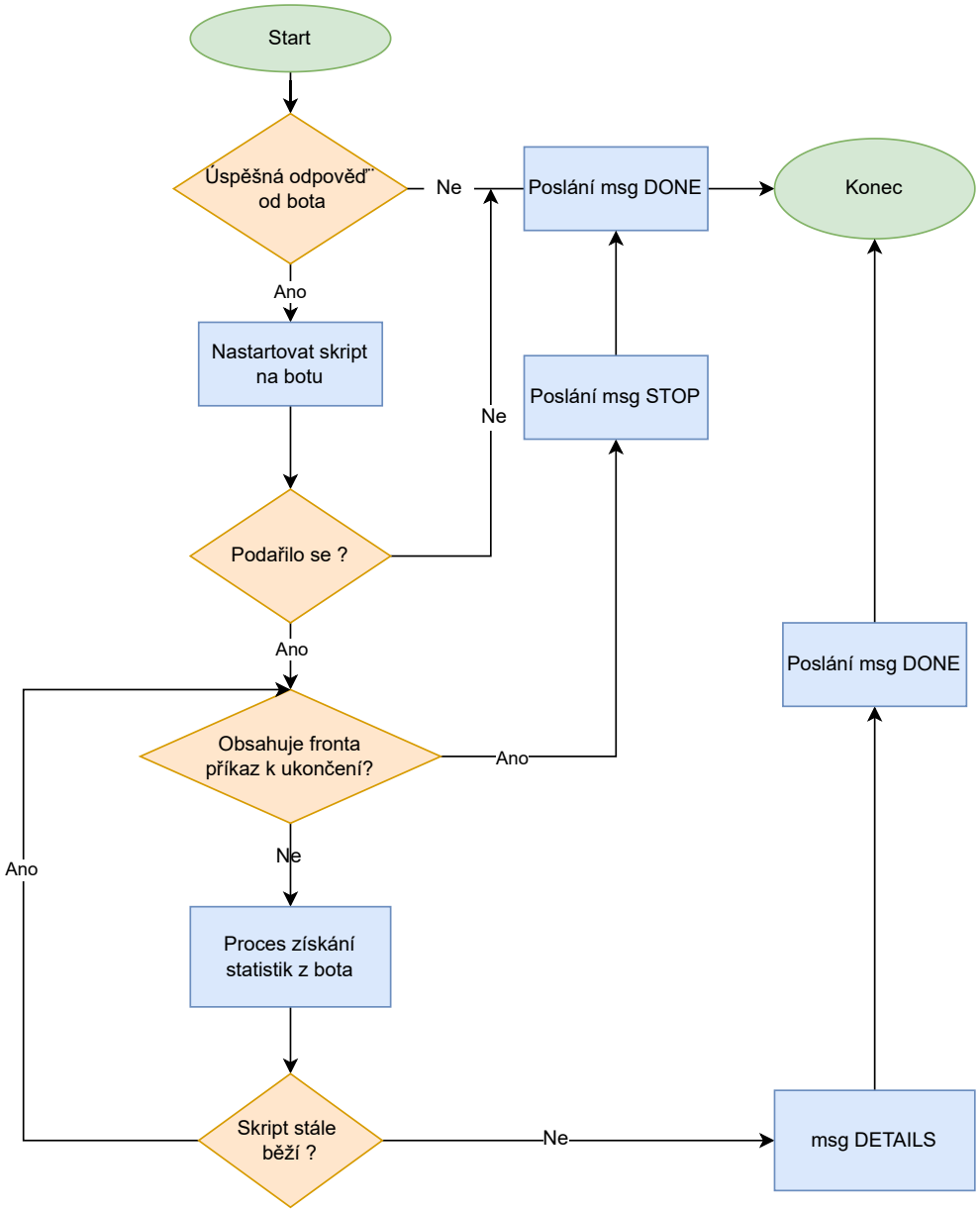
Periodické načítání stavu situace na botech probíhá ve dříve definovaném intervalu (uloženém v proměnné period). To je realizováno voláním funkce *time.Sleep()*. Následuje kontrola fronty done. Zde je použito tzv. non-blocking čtení fronty, kdy běh programu není blokován, pokud ve frontě není žádná zpráva (defaultně je tomu naopak, tzn. metoda get blokuje program, dokud není zpráva načtena). Jelikož non-blocking čtení vyvolává v případě prázdné fronty výjimku Empty, musí být adekvátně tato výjimka ošetřena (except Empty: pass). V případě, že fronta done zprávu obsahuje, je zaslán požadavek details (6.5.2), následně požadavek stop (6.5.2) a funkce je ukončena. Pokud není zaznamenán požadavek na ukončení, je zaslán požadavek status (6.5.2). Výsledek požadavku status je zpracován ve funkci *process_status* popsané níže. Funkce vrací bool hodnotu určující, zda skript na botu stále běží. Pokud tomu tak není (*is_running = False*), je (podobně jako při žádosti o přerušení z řídící vrstvy), zaslán požadavek details, výsledek zpracován a funkce ukončena. Na obrázku 6.8 lze vidět diagram této funkce.

Funkce process_status

Vstupní paramatery:

- *path_result*: str,
- *status*: dict,
- *result_queue*: *multiprocessing.Queue*,
- *shift*: int.

Funkce *process_status* jednoduše zasílá do fronty *result_queue* aktuální stav výkonových statistik.



Obr. 6.8: Diagram funkce *agent_worker*.

Funkce send_message

Vstupní parametry:

- *server_address*: tuple - adresa a port cíle komunikace,
- *cmd*: dict - objekt typu slovník.

Funkce *send_message* realizuje síťovou komunikaci s botem. Jak již bylo zmíněno, komunikace je realizována použitím modulu sockets. Globální proměnná *buffer_size* určuje velikost bufferu, nastavenou defaultně na 1 kB. Poté následuje získání socketu s AddressFamily AF_INET a SocketKind Stream. Po serializaci a enkodování vstupního parametru typu slovník, je tento parametr zaslán na bota pomocí funkce *sockhelper.send_msg*, principiálně popsané v kapitole 6.5.3 a odpověď je serializována a funkcí vrácena.

Funkce process_details

Vstupní parametry:

- *path_result*: str - cesta v adresářové struktuře, kam se mají zapsat detaily o běhu skriptu na botech,
- *details*: dict - slovník s detaily.

Funkce *process_details* obdrží jako vstup slovník details, definovaný v protokolu master-slave a zapíše jeho obsah do souborů. Název výsledných souborů je vygenerován tak, že vždy začíná IP adresou a portem bota, následovaný samotným názvem, Vždy je ukládán standardní výstup, standardní chybový výstup, případně soubory, definované volitelným klíčem *downloads*.

Funkce update_gui

Vstupní parametry:

- *status*: dict - slovník s aktuálními performance statistikami,
- *results_queue*: *multiprocessing.Queue* - odkaz na frontu pro zaslání agregovaných statistik.

Funkce *update_gui* zpracovává statistiky jednotlivých botů a pomocí podpůrné funkce *get_actual_stats* přepočítává agregované statistiky tak, že vrací průměrné hodnoty pro CPU a využití paměti a sumy spojení a síťových rozhraních.

Funkce done_func

Vstupní parametry:

- *done*: *multiprocessing.Queue*,
- *status_queue*: *multiprocessing.Queue*,
- *agent_list*: list.

Funkce done má jednoduchou, ale pro synchronizaci subprocesů důležitou roli. Volá metodu get pro frontu *status_queue* v blocking módu. Jakmile je načtena jakákoliv zpráva, projde seznam reprezentující seznam aktuálně aktivních botů a pošle jim zprávu DONE, která zabezpečí jejich korektní ukončení (viz kapitola 6.6.1)

6.7 Popis programu bot

Bot je koncipován jako síťový server, implementující funkčnost definovanou protokolem master-slave, viz 6.5.2. Při návrhu modelu a způsobu implementace byl kladen důraz na použití nativních modulů jazyka Python. Jediným externím modulem je psutil[10].

Jelikož se jedná o program, který by měl být samostatně spouštěn z příkazové řádky, je vstupním bodem top-level prostředí main. Program vyžaduje 2 argumenty. IP a port, na kterém se má pokusit otevřít port a provádět akce požadované modulem master. Pokud je spuštěn bez parametrů, defaultně se pokusí otevřít socket localhost:10050.

6.7.1 Popis algoritmu

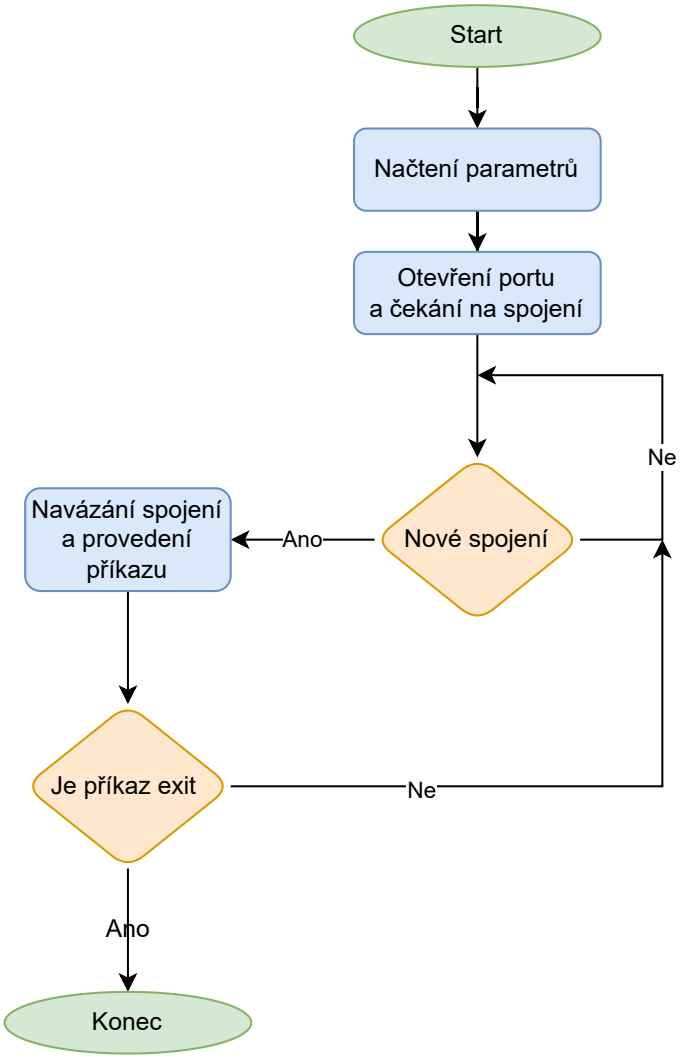
Funkce run_server

Vstupní parametry:

- *host* - IP adresa,
- *port* - port.

Funkce *run_server* je fundamentální částí bota a je znázorněna na obrázku 6.9 prostřednictvím diagramu. Nejdříve se pokusí otevřít socket určený vstupními parametry a v nekonečném cyklu provádět akce, požadované modulem master. V cyklu se jednoduše čeká na přicházející požadavek, který je zpracován (co se týče síťové úrovně) způsobem podrobněji popsaným v kapitole 6.5.3. Jakmile je požadavek obdržen, je učiněn pokus o jeho deserializaci do slovníku command a pak předán ke zpracování funkcí *process_command*. Funkce vrátí příslušný slovník, který je následně vložen do výsledného objektu tak, aby splňoval definici protokolu (viz 6.5.2) a posléze je serializován do formátu JSON a zakódován do pole bytů a zaslán zpět modulu mastera. Pokud je detekován požadavek exit, je program ukončen. V opačném případě se cyklus vrací na začátek a očekává další požadavek. Celý kód je

[10]psutil (process and system utilities) je multiplatformní knihovna pro získávání informací o běžících procesech a využití systému (procesor, paměť, disky, síť, senzory) v jazyce Python. Je užitečná především pro monitorování systému, profilování a omezování prostředků procesů a správu běžících procesů.



Obr. 6.9: Diagram funkce *run_server*.

důsledně opatřen cykly *Try-Except*, aby nemohlo dojít k jeho ukončení neočekávanou chybou a byl co nejvíce stabilní. Hlavní cyklus je ošetřen rovněž *finally* částí, tak aby vždy došlo k uvolnění síťových prostředků, pokud program z jakýchkoliv důvodů končí.

Funkce process_command

Vstupní parametry:

- *cmd*: dict - slovník s požadavkem.

Funkce *process_command* funguje jako dispečer funkce, která podle klíče *command* slovníku s požadavkem provede rozdělení do funkcí, které podporované požadavky implementují.

Funkce run_command

Vstupní parametry:

- *cmd*: dict - slovník s požadavkem.

Tato funkce implementuje požadavky run. Na začátku lze vidět deklaraci globálních proměnných. Ty jsou nutné pro sdílení stavu mezi jednotlivými požadavky. Poté se zjišťuje, jestli už není spuštěn jiný skript, který dosud nebyl ukončen. Pokud ano, je vytvořen slovník s odpovědí „AlreadyRunning“ (6.5.2).

Následuje test, zda se skript nachází v podsložce scripts, což je považováno za defaultní stav, a pokud tomu tak je, opatří se jméno skriptu relativní cestou. Pokud ne, není v tuto chvíli vyhlášena chyba, název zůstává beze změny a je tudíž ještě počítáno s možností, ze se daný skript nachází v cestě PATH (toho se například může využít při volání slowHTTPtest, jež je součástí řady linuxových distribucí).

V dalších blocích se ošetřují situace, kdy je samotný skript ke spuštění ještě doplněn rozšiřujícími nepovinnými parametry. První z nich, target, reprezentuje cíl útoku tak, aby bylo možné identifikovat síťové rozhraní, přes které bude skript s cílem útoku komunikovat. Díky tomu mohou být zasílány statistiky pouze z tohoto konkrétního rozhraní. Pokud parametr target chybí, jsou tato data zasílána sečtená za všechna síťová rozhraní daného počítače. Pro identifikaci rozhraní na základě parametru target byla napsána funkce *try_find_if*, která bude krátce popsána dále v textu. Do globální proměnné nic_init je pak vložen aktuální stav sčítačů v okamžiku těsně před spuštěním skriptu. Pokud požadavek obsahuje parametr *post_process*, je tento parametr uložen do globální proměnné k pozdějšímu zpracování.

Následuje samotné spuštění skriptu pomocí nativního modulu jazyka Python subprocess s parametry stdout a stderr (více v kapitole 6.8.3). Pokud spuštění selže,

je zpět zaslána odpověď „Failed“ s detailem chyby. Pokud je skript spuštěn v pořádku, je zaslána odpověď „Started“ a do globální proměnné *process_details* je uložen odkaz na detaily spuštěného procesu.

Funkce details_command

Vstupní parametry:

- *cmd*: dict - slovník s požadavkem.

Tato funkce implementuje požadavek details. Ten, jak bylo popsáno v 6.5.2, slouží především k zaslání dat po ukončení běhu procesu a je koncipován tak, že by měl být spuštěn pouze jednou. Samotná funkce začíná voláním *get_process_details*. Tato funkce detekuje, jestli proces stále běží, nebo již byl ukončen a dle toho zasílá příslušná data. Pro běžící proces je to v podstatě jen PID. Pokud je zjištěno, že proces již skončil, jsou zaslána rovněž údaje o návratovém kódu a obsah bufferů se standardním výstupem a standardním chybovým výstupem. Obsah obou těchto bufferů je zároveň očištěn o ANSI escape sequence[11], tak aby byl čitelný pro člověka. Pokud existuje v globální proměnné postprocess klíč *cmd*, je proveden. Stejně tak, pokud existuje klíč *download*, jsou všechny soubory načteny a vloženy do odpovědi.

Funkce stop_command

Vstupní parametry:

- *cmd*: dict - slovník s požadavkem.

Jak z názvu vyplývá, účelem funkce *stop_command* je spolehlivé zastavení běžícího procesu. Funkce začíná prozkoumáním globální proměnné *running_command*. V případě, že je rovna None, je zaslána odpověď „NeverRunning“ a funkce končí. V opačném případě a pokud metoda *poll()* je rovna None (metoda *poll* vrací None v případě, ze proces běží a návratový kód, pokud již běh procesu skončil), je kaskádovitě učiněn pokus o jeho zastavení a to tak, ze se nejdříve posílá signál SIGINT, pokud není úspěšný tak následuje pokus terminate (SIGTERM) a pokud ani ten není úspěšný, tak kill (SIGKILL). Ten je považován za vždy úspěšný, protože další možnost, jak proces ukončit, už neexistuje. Následně je zaslána odpověď „Terminated“. V případě, že žádný proces neběží, je zaslána odpověď „NotRunning“.

Funkce status_command

Vstupní parametry:

- *cmd*: dict - slovník s požadavkem.

[11]Metoda pro řízení formátování, barev a dalších vlastností výstupu na obrazovkové textové terminály pomocí signalizace v přenosovém kanálu.

Tato funkce vrací odpověď, která obsahuje výkonnostní statistiky běžícího procesu a některé další síťové statistiky. Kromě již dříve popsané logice dle stavu procesu, je její podstatnou částí volání funkce *get_stats*, která de facto tuto funkčnost implementuje. Využívá se modulu psutil. Kromě jednoduchého načtení údajů o CPU a Memory procesu je v části shromažďující síťové statistiky vidět, že je pro každé spojení vrácen její status a rovněž, že jsou zasílány čítače síťového rozhraní dle toho, zda známe konkrétní rozhraní, či ne.

Funkce try_find_if

Vstupní parametry:

- *h*: str - řetězec identifikující cíl útoku

Funkce *try_find_if* se nachází v modulu netutil a slouží k určení síťového rozhraní, kterým se na daném počítači bude komunikovat s cílem. Vychází se z úvahy, že daný řetězec může být:

- URL,
- hostname,
- IP adresa.

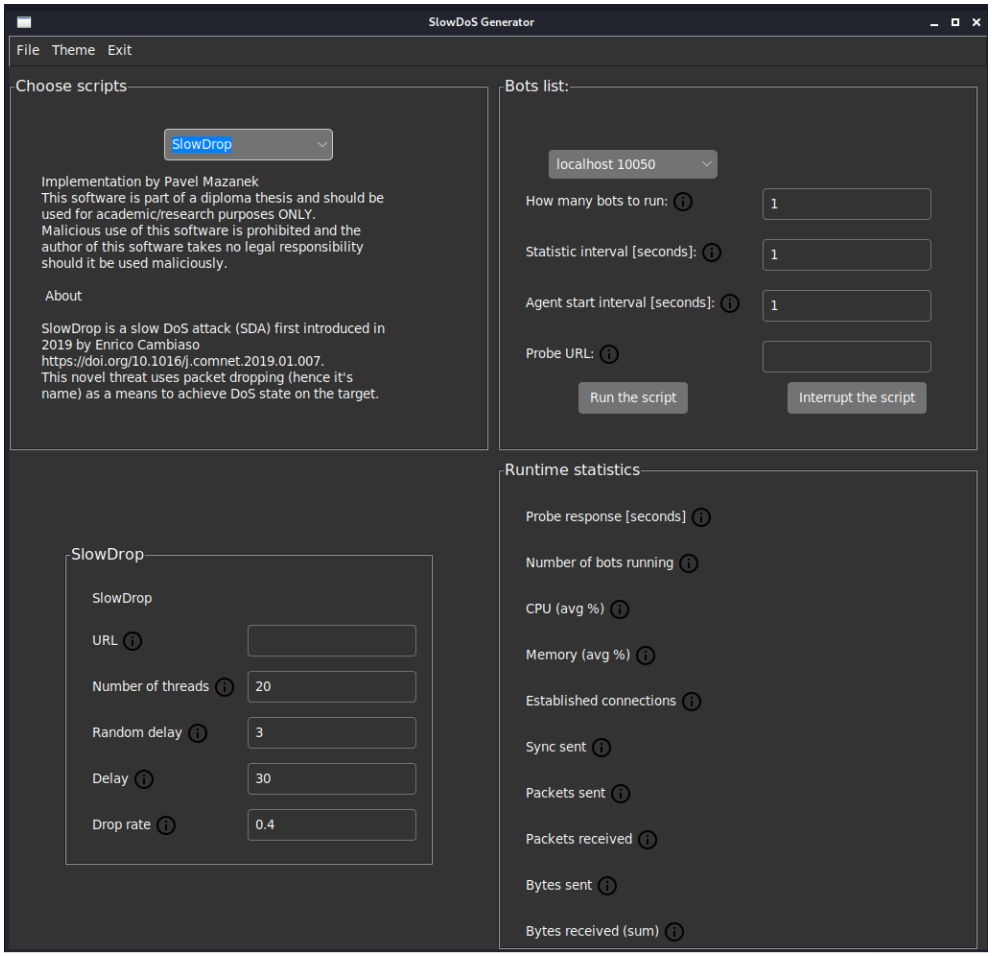
Nejdříve tedy testujeme řetězec jako URL funkcí *host_from_url*. Jestliže je to URL, vrací host část URL, pokud ne, vstupní řetězec je vrácen beze změny. Dále se pokusíme o překlad řetězce pomocí funkce *ip_from_hostname*. Nakonec pokud je překlad úspěšný, voláme funkci *interface_go_out*, která využívá nástroj linuxu ip a vrací kýžené jméno síťového rozhraní.

6.8 Grafické rozhraní a jeho interakce

Grafické rozhraní, je jak již bylo zmíněno, vytvořeno pomocí knihovny tkinter, která je nejvíce využívána. Tkinter umožňuje uživateli vybrat určitý styl, kterým jsou jednotlivé komponenty ovlivněny, může se jednat o oficiální distribuce na míru danému operačnímu systému nebo o distribuce od třetích stran, které jsou dostupné pod různými licencemi pro volné užití ve vlastních projektech. V případě navrhovaného generátoru byl zvolen styl **Azure-ttk-theme**. V následujících podkapitolách bude obecně popsáno rozvržení jednotlivých komponentů v grafickém rozhraní a dále popsány funkce, které jsou uživateli viditelné pří průběhu programu a také funkce, které uživateli umožňují určitou interakci s navrhovaným programem.

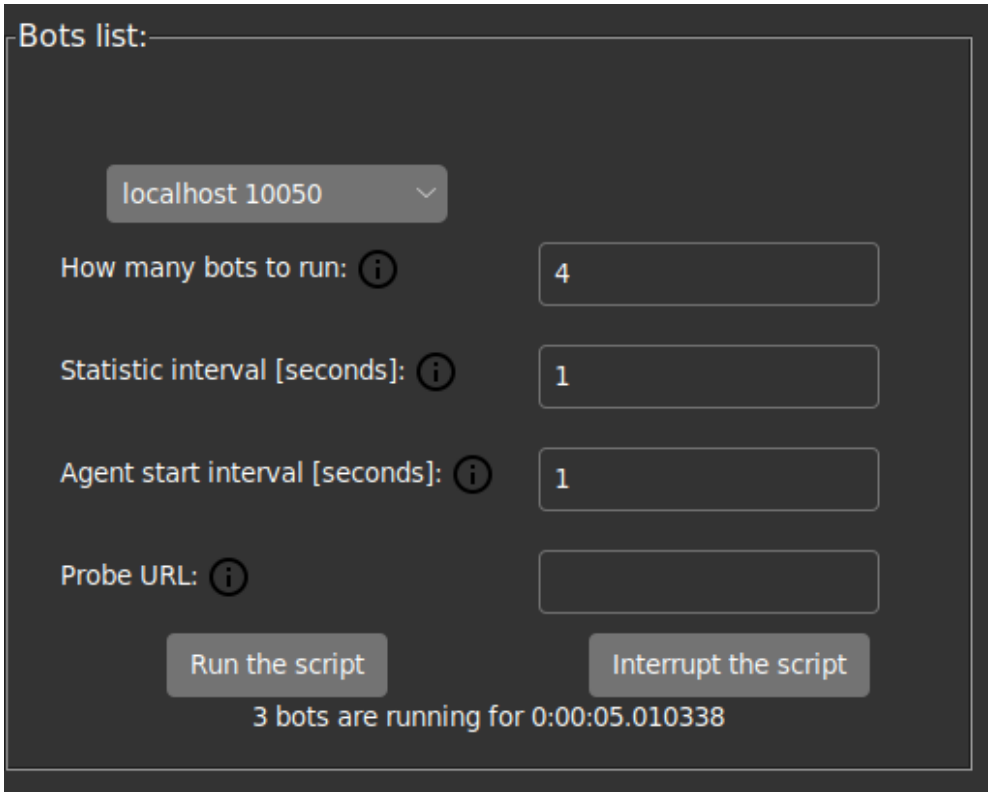
6.8.1 GUI

Po spuštění programu se uživateli zobrazí grafické rozhraní, což lze vidět na obrázku 6.10. Logika grafického rozhraní spočívá v jednom hlavním okně main, ve



Obr. 6.10: Grafické rozhraní po spuštění.

kterém jsou dále vloženy čtyři frames (každý z nich je ohraničen) a horní lišta menu (detailněji znázorněno na obrázku 6.5). Frames jsou vidět na obrázku 6.10 a jsou pojmenovány jako „Choose scripts“, Název vybraného skriptu (v ilustračním případě se jedná o skript SlowDrop), „Bots list“ a „Runtime statistics“. První zmíněný frame, se skládá z jednoho combo boxu (jehož logika je popsána v kapitole 6.3.3) a autorského popisu daného skriptu. Frame s názvem momentálně vybraného skriptu je generován vždy dynamicky a liší se tak s každým vybraným útokem (znovu popsáno v kapitole 6.3.3). „Bots list“ frame je generován staticky, a je proto stejný pro každý zvolený útok. Frame má na starost nastavovaní jednotlivých botů, kteří jsou pro daný scénář k dispozici, dále se v tomto framu nachází samotné ovládání programu a také výpis o aktuálně prováděných úkonech (například začátek programu, průběh programu, chyby v programu, informace o průběhu a informace o ukončení programu), což lze lépe vidět na obrázku 6.11. V posledním framu „Runtime statis-

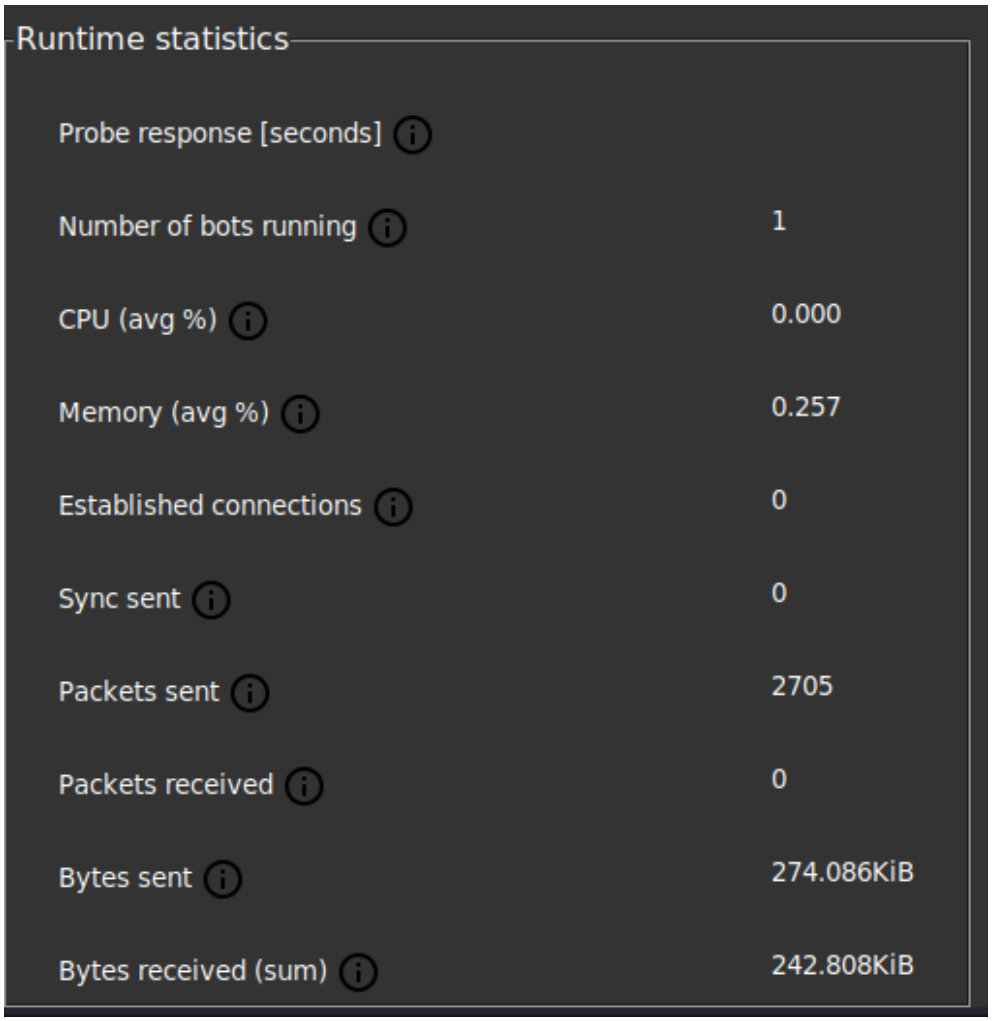


Obr. 6.11: Detail na frame Bots list a vypisování.

tics“ jsou zobrazeny data a statistiky, které program sbírá, zpracovává a následně v reálném čase zobrazuje měnící se statistiky uživateli. Tato data jsou sbírána především pomocí knihovny psutil, kterou je potřeba ručně doinstalovat a její funkčnost byla zmíněna v kapitole 6.7 (více o požadavcích na spuštění programu, lze nalézt

v příloze této práce) a déle pomocí nástroje netstat[12]. Možný stav tohoto framu, lze vidět na obrázku 6.12.

Poslední nepopsanou částí programu je horní menu, které se nachází na horní liště programu. Uživatel zde může vybrat File, Theme nebo Exit. Položka File obsahuje možnost importu a exportu (popsáno v následující kapitole), položka Theme pak umožňuje uživateli měnit z tmavého módu na světlý mód a položka Exit, která jednoduše ukončuje celý program.



Obr. 6.12: Detail na frame Runtime statistic.

6.8.2 Funkce Import/Export

Jelikož se jedná o nástroj testovací, je žádoucí možnost exportování zajímavých parametrů útoku do externího souboru a jeho následné importování do programu pro

[12]Zobrazuje aktivní spojení TCP, porty, na kterých počítač naslouchá, statistiky Ethernetu, směrovací tabulku IP, statistiky IPv4 (pro protokoly IP, ICMP, TCP a UDP).

další analýzu. Logika těchto funkcí je založena na souboru ve formátu JSON, který je nejdřív zaplněn všemi vyplněnými parametry daného skriptu a nastavení botů a následně je uživateli zobrazeno dialogové okno pro výběr místa uložení. Pro opětovné nahrání (import) parametrů se uživateli opět zobrazí dialogové okno, ve kterém následně zvolí kýžený soubor ve formátu JSON, soubor se zpracuje a parametry jsou nahrány do dedikovaných polí.

6.8.3 Ukládání statistik

Stejně jak je potřeba archivovat parametry pro další využití, je potřeba zaznamenat veškeré statistiky, které program vyprodukuje (jedná se o stejné statistiky, které jsou v reálném čase zobrazeny ve framu Runtime statistics zmíněného v kapitole 6.8.1). V tomto případě se jevil jako vhodný formát dat CSV, který byl již využit i v rámci semestrální práce. Po každém běhu útoku je ve složce results vytvořená podsložka s názvem daného skriptu a v této složce je dále vytvořena podsložka s datem a časem spuštění útoku, ve které se nachází veškeré získané statistiky. Kromě souboru CSV se zde ještě nachází textové soubory s výpisem daného skriptu do konzole. Jedná se buď o takzvaný standard output (stdout), který obsahuje text, který se za normálního běhu programu vypisuje do uživatelské konzole nebo se může jednat standard error (stderr), který vypisuje veškeré chybové hlášky při běhu programu (v ideálním případě bude tento výpis prázdný).

V případě, že některý program implementovaný do generátoru vyvíjeného v této práci generuje užitečná data sám o sobě (slowHTTPtest), je žádoucí tato data získat. Pro tento případ se dostáváme k problematice definovaných objektů v konfiguračním souboru JSON, které jsou zmíněné v kapitole 6.3.1. Jeden z pokročilejších objektů konfiguračního souboru je post_process, znázorněn na výpise 6.9, který umožňuje pomocí protokolu master-slave stáhnout soubory, které jsou zmíněny pod klíčem v download na řádku č. 4 a č. 5. Tyto soubory se na daném zařízení musí nacházet a v tomto případě jsou vytvořeny pomocí aplikace slowHTTPtest a uloženy opět do zmíněné složky scripts.

Výpis 6.9: Objekt post_process.

```
1 "post_process": {
2         "cmd": "",
3         "download": [
4             "SlowHttpTest.csv",
5             "SlowHttpTest.html"]
6         },
```

Závěr

Práce se zabývala především problematikou pomalých útoků na odepření služeb, avšak hlavním cílem práce bylo vytvoření programové nadstavby, nad již existující útoky vyvinuté buď na Vysokém učení technickém v Brně nebo na útoky, které jsou součástí volně dostupných generátorů pomalých útoků. Tato programová nadstavba měla poskytnout hlavně grafické rozhraní, rozšíření o další funkce jako DDoS, generování definovaných statistik a další funkce, které ulehčí testovací podmínky a vyplní tak největší nedostatky, které se v oblasti existujících generátoru pomalých útoků vyskytují.

Cílem diplomové práce bylo nejdříve v teoretické části nastudovat protokol HTTP, prozkoumat různé typy útoků na odepření služeb a jejich podobu v distribuované verzi. Byla také nastudována problematika pomalých útoků na odepření služeb. Útoky byly následně rozděleny do podskupin, kdy v každé byl dopodrobna prozkoumán minimálně jeden útok, který do této podskupiny spadal. V neposlední řadě byly představeny známé generátory pro pomalé útoky na odepření služeb a byl zdůrazněn jejich nedostatek jednotnosti a nutnost vytvoření generátoru, který bude uživatelsky přívětivý, aktuální a jednotný.

V praktické části práce byla nejdříve vymyšlena a vyvinuta logika, která umožnila automatické vytvoření grafického rozhraní a to nezávisle na dodaném skriptu. Jedním z úkolů této práce bylo implementovat alespoň šest různých útoků, které bude uživatel schopen ovládat pomocí již zmíněného grafického rozhraní. Tento cíl práce byl splněn a generátor tak disponuje sedmi různými typy pomalých útoků a vzhledem k vyvinuté logice může uživatel velmi jednoduše přidávat další útoky bez nutnosti jakkoliv zasahovat do zdrojového kódu vyvíjeného generátoru. V další části praktické práce byl navržen vlastní protokol, díky kterému je možno částečně simulovat distribuovaný útok na odepření služeb, což byl jeden z dalších stěžejních cílů této práce. Simulace je docílena jen částečně z důvodu zachování jiných výhod generátoru, jako je například modularita. Během vývoje se muselo přistupovat k určitým kompromisům, které zajistily, aby generátor splňoval zadání práce v co možná největším rozsahu a zároveň zachoval maximální praktickou funkcionalitu, v neposlední řadě i další škálovatelnost pro budoucí vývoj. Dalším bodem zadání bylo vyhodnocování co možná největšího počtu statistik z útoků a zobrazení těchto statistik v rámci grafického rozhraní. Vyhodnocování bylo úspěšně implementováno jak do grafického rozhraní, tak do souborů, se kterými je možno pracovat po skončení dané simulace. Během vývoje aplikace se z důvodu vysoké časové náročnosti přesahující možnosti této diplomové práce nepodařil splnit bod zadání související s úpravou dodaných skriptů tak, aby pracovali i na IPv6 sítích.

Navržený generátor se dá považovat za položený základní kámen jeho vývoje. Na jeho návrhu lze dále stavět a využívat navržené funkce či protokol pro vývoj dalších podobných generátorů. Pro budoucí vývoj by pak bylo vhodné navrhnout jakýsi vzorový příklad skriptu, podle kterého by se dále vyvíjely nové Slow DoS útoky. Toto vylepšení by do budoucna značně ulehčilo přidávání dalších funkcí nad celým generátorem, jelikož by programátor nemusel myslet na možné scénáře chování nevzorových skriptů. Další vývoj by se pak mohl zaměřit na přidání statistik formou grafu v reálném čase.

Literatura

[1] *Computer Network Models* [online]. 2019 [cit. 2021-12-07]. Dostupné z: <https://www.tutorialspoint.com/data_communication_computer_network/computer_network_models.htm>.

[2] LAWRENCE, Williams. *TCP/IP vs OSI Model: What's the Difference?* [online]. 2021 [cit. 2021-12-07]. Dostupné z: <https://www.guru99.com/difference-tcp-ip-vs-osi-model.html>.

[3] *Transport Layer responsibilities* [online]. 2021 [cit. 2021-12-07]. Dostupné z: <https://www.geeksforgeeks.org/transport-layer-responsibilities/>.

[4] *TCP Connection Establish and Terminate* [online]. [cit. 2021-12-07]. Dostupné z: <https://www.vskills.in/certification/tutorial/tcp-connection-establish-and-terminate/>.

[5] *TCP Window Size Scaling* [online]. 2013-2021 [cit. 2021-12-07]. Dostupné z: <https://networklessons.com/cisco/ccie-routing-switching-written/tcp-window-size-scaling>.

[6] *Evolution of HTTP* [online]. 2021 [cit. 2021-12-07]. Dostupné z: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP>.

[7] *HTTP request methods* [online]. 2021 [cit. 2021-12-07]. Dostupné z: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>.

[8] *How Does HTTPS Work to Improve Website Security?* [online]. [cit. 2021-12-07]. Dostupné z: <https://sectigostore.com/page/how-does-https-work/>.

[9] Matthews, Tim. *The Origins of Web Security and the Birth of Security Socket Layer (SSL) Protocol* [online] 2019. [cit. 2021-12-07]. Dostupné z: <https://www.exabeam.com/information-security/web-security-security-socket-layer-protocol-ssl/>.

[10] Biketi, Bradley. *Comparison between the HTTP/3 and HTTP/2 Protocols* [online] 2021. [cit. 2021-12-07]. Dostupné z: <https://www.section.io/engineering-education/http3-vs-http2/>.

[11] *Usage statistics of HTTP/2 for websites* [online] [cit. 2021-12-07]. Dostupné z: <https://w3techs.com/technologies/details/ce-http2>.

[12] *HTTP/1.1 vs HTTP/2: What's the Difference?* 2019 [online] [cit. 2021-12-07]. Dostupné z: <https://www.digitalocean.com/community/tutorials/http-1-1-vs-http-2-what-s-the-difference>.

[13] *Better HTTP/2 Prioritization for a Faster Web* 2019 [online] [cit. 2021-12-07]. Dostupné z: <https://blog.cloudflare.com/better-http-2-prioritization-for-a-faster-web/>.

[14] Sandeep, Verma. *HTTP/1 to HTTP/2 to HTTP/3* 2019 [online] [cit. 2021-12-07]. Dostupné z: <https://medium.com/@sandeep4.verma/http-1-to-http-2-to-http-3-647e73df67a8>.

[15] Bergan, Tom. *HTTP/2 Server Push* 2017 [online] [cit. 2021-12-07]. Dostupné z: <https://go.dev/blog/h2push>.

[16] *Chapter 8. HPACK header compression* [online] [cit. 2021-12-07]. Dostupné z: <https://livebook.manning.com/book/http2-in-action/chapter-8/1>.

[17] Fietkiewicz, Martin. *HTTP/2 vs HTTP/3: A comparison* [online] [cit. 2021-12-07]. Dostupné z: <https://ably.com/topic/http-2-vs-http-3>.

[18] Tellakula, Sreeni. *Comparing HTTP/3 vs. HTTP/2 Performance* 2020 [online] [cit. 2021-12-07]. Dostupné z: <https://blog.cloudflare.com/http-3-vs-http-2/>.

[19] *Types of DDoS Attacks: General Breakdown* 2020 [online] [cit. 2021-12-07]. Dostupné z: <https://www.pentasecurity.com/blog/ddos-attacks-types-explanation/>.

[20] *What is a denial-of-service (DoS) attack?* [online] [cit. 2021-12-07]. Dostupné z: <https://www.cloudflare.com/learning/ddos/glossary/denial-of-service/>.

[21] *DDoS attacks expected to reach 11M by end of 2021* 2021 [online] [cit. 2021-12-07]. Dostupné z: <https://venturebeat.com/2021/10/06/atlas-vpn-ddos-attacks-expected-to-reach-11m-by-end-of-2021/>.

[22] Ferguson, Kevin. *Distributed denial-of-service (DDoS) attack* 2021 [online] [cit. 2021-12-07]. Dostupné z: <https://www.techtarget.com/searchsecurity/definition/distributed-denial-of-service-attack>.

[23] *What is a Botnet?* [online] [cit. 2021-12-07]. Dostupné z: <https://www.cloudflare.com/learning/ddos/what-is-a-ddos-botnet/>.

[24] *DDoS Attacks* [online] [cit. 2021-12-07]. Dostupné z: <https://www.imperva.com/learn/ddos/ddos-attacks/>.

[25] *Denial of Service DDoS attack* 2021 [online] [cit. 2021-12-07]. Dostupné z: <https://www.geeksforgeeks.org/denial-of-service-ddos-attack/>.

[26] *What is a low and slow attack?* [online] [cit. 2021-12-07]. Dostupné z: <https://www.cloudflare.com/learning/ddos/ddos-low-and-slow-attack/>.

[27] CAMBIASO, Enrico, Gianluca PAPALEO, Giovanni CHIOLA a Maurizio AIELLO. *Slow DoS attacks: definition and categorisation. International Journal of Trust Management in Computing and Communications. 2013, roč. 1 (3/4), 20 stran. DOI: 10.1504/IJTMCC.2013.056440. ISSN 2048-8378.*. Dostupné z: <http://www.inderscience.com/link.php?id=56440>.

[28] SIKORA, Marek, Radek FUJDIAK, Karel KUCHAR, Eva HOLASOVA a Jiri MISUREC. *Generator of Slow Denial-of-Service Cyber Attacks. Sensors. 2021, 21(16). ISSN 1424-8220. DOI:10.3390/s21165473.*

[29] *SlowHTTPtest* 2015 [online] [cit. 2021-12-07]. Dostupné z: <https://github.com/shekyan/slowhttptest>.

[30] *Slowloris* 2014 [online] [cit. 2021-12-07]. Dostupné z: <https://github.com/0xc0d/Slow-Loris>.

[31] *PyLoris* 2012 [online] [cit. 2021-12-07]. Dostupné z: <https://sourceforge.net/projects/pyloris/>.

[32] RICHTER, Dominik. *Slow rate DoS útoky nezávislé na protokolu aplikační vrstvy*. Brno, 2020. Bakalářská. VUT. Vedoucí práce Ing. Marek Sikora.

[33] KRIVULČÍK, Andrej. *Generátor Slow DoS útoků*. Brno, 2020. Bakalářská. VUT. Vedoucí práce Ing. Marek Sikora.

[34] MAZÁNEK, Pavel. *Modelování a detekce útoku SlowDrop*. Brno, 2020. Diplomová. VUT. Vedoucí práce Ing. Marek Sikora.

Seznam symbolů a zkratek

DoS	Denial of Service
TCP	Transmission Control Protocol
HTTP	User Datagram Protocol
UDP	User Datagram Protocol
ISO	International Standards Organization
OSI	Open Systems Interconnection
IP	Internet Protocol
SMTP	Simple Mail Transfer Protocol
WWW	World Wide Web
URL	Uniform Resource Locator
FQDN	Fully qualified domain name
RFC	Request For Comment
HTTPS	Hypertext Transfer Protocol Secure
SSL	Secure Sockets Layer
TLS	Transport Layer Security
SPDY	Speedy (networking protocol)
IETF	Internet Engineering Task Force
QUIC	Quick UDP Internet Connections
API	Application Programming Interface
DDoS	Distributed Denial of Service
IoT	Internet of Things
LoRDAS	Low-Rate DoS Attack against Application Servers
IPS	Intrusion Prevention Systems
IDS	Intrusion Detection Systems

ARP	Address Resolution Protocol
NAT	Network address translation
CSV	Comma-separated values
CSS	Cascading Style Sheets
JSON	JavaScript Object Notation
YAML	Ain't Markup Language
MIT	Massachusetts Institute of Technology
PID	Process IDentification
CPU	Central Processing Unit

7 Obsah elektronické přílohy

- / ..... kořenový adresář přiloženého CD
  - __pycache__ ..... python soubory
    - master.cpython-39
    - respTime.cpython-39
    - sockhelper.cpython-39
  - Azure-ttk-theme ..... grafický styl
    - theme
      - dark.ctl .4 light.ctl .3 azure.ctl
  - img ..... použité ikony a možné další ikony
  - scripts ..... skripty a json soubory nutné pro spuštění
    - slowDoSGen.py
    - slowDoSGen.py.json
    - slowDrop.py
    - slowDrop.py.json
    - slowhttptest.cc
    - slowwhttptest.json
    - slowLoris.py
    - slowLoris.py.json
  - agent.py
  - configuration.json
  - graph.py
  - main.py
  - master.py
  - netutil.py
  - README.txt
  - respTime.py
  - sockhelper.py