ENHANCED HYPERTEXT TRANSFER PROTOCOL DISTRIBUTED DENIAL OF SERVICE DETECTION SCHEME WITH GET HEADERS ADOPTION

ABDUL GHAFAR JAAFAR

UNIVERSITI TEKNOLOGI MALAYSIA

ENHANCED HYPERTEXT TRANSFER PROTOCOL DISTRIBUTED DENIAL OF SERVICE DETECTION SCHEME WITH GET HEADERS ADOPTION

ABDUL GHAFAR JAAFAR

A thesis submitted in fulfilment of the
requirements for the award of the degree of
Doctor of Philosophy

Razak Faculty of Technology and Informatics
Universiti Teknologi Malaysia

OCTOBER 2020

DEDICATION

This thesis is dedicated to my father, who taught me that the best kind of knowledge to have is that which is learned for its own sake. It is also dedicated to my mother, who taught me that even the largest task can be accomplished if it is done one step at a time.

ACKNOWLEDGEMENT

First of all, I would like to express my profound gratitude to my main supervisor, Associate Professor Dr. Mohd Shahidan Abdullah, for encouragement, guidance, critics and friendship. I am also very thankful to my co-supervisor Dr Saiful Adli Ismail for their guidance, advices and motivation. Without their continued support and interest, this thesis would not have been the same as presented here.

I especially express all the gratitude to my beloved mother, Azizah Othman and grateful to my adored wife, Hasliana for her patience, understanding, support and many sacrifices to keep me on track and helped me survive during this adventure. My aunty Sariyah Ismail and mother in law Mejar Aini which provided me funding throughout my PhD journey. I dedicate my work to my little hero son, Amsyar Iskandar for giving me smiles, hope and endless joyfulness. I wish as your curiosity grows, so will your passion for science and the majestic life.

Finally, greatly appreciated the opportunity from UTM to purse my PhD, I thank very much to every UTM community, including each person at (faculties, library, staff, employees, students and laborer's).

ABSTRACT

A transaction between a user and a web server involves several layers that are known as Open Systems Interconnection (OSI) layers. The application layer is the highest layer which is vulnerable to manipulation by attackers that execute Hypertext Transfer Protocol Distributed Denial of Service (HTTP DDoS) attacks. In the event of attack traffic, this manipulation causes the HTTP DDoS traffic to appear legitimate and is therefore complex to be recognized due to the forged request headers. The attack produces various patterns which lead to the inability of the current detection to recognize HTTP DDoS attacks. The approaches that have been adopted by prior studies thus far were unable to accurately detect malicious GET request resulting the attack traffic to be predicted as genuine. Besides, the current approach is unable to detect HTTP DDoS attacks through proxies and forged request query. The purpose of this research is to enhance the detection schemes in detecting HTTP DDoS attacks. To achieve this purpose, a three-phased research methodology has been structured. The first phase is literature and problem analysis, the second phase is design and implementation while the third phase is verification and validation. The research objectives outlined in this research are parallel with the research methodology. The first objective is to improve the HTTP DDoS detection framework by adopting new components and appending new attributes in the existing component. The next objective is to enhance and develop detection algorithms by the adoption of GET headers and web browser attributes. The third objective is to improve the True Positive rate and True Negative rate and to decrease the False Positive rate and False Negative rate to detect HTTP DDoS attack. The enhanced detection scheme comprised a detection framework made up of several components to indicate the detection flow and algorithms involved in recognizing HTTP DDoS attacks. The detection framework was constructed to have a sequential inspection to detect different attack patterns produced by HTTP DDoS attacks. The source inspection algorithm was developed to improve the identification of the source initiator by adopting web browser attributes. The request headers inspection was devised to improve the detection of the authenticity of HTTP request traffic by checking the existence of GET headers. The request query inspection was designed to detect any forged query using the query attached during GET requests transaction and comparing the query with the detection rules and database. The proxy inspection was fabricated to detect HTTP DDoS attacks executed through a proxy by utilizing the proxy GET headers that were involved during GET requests. Experimental results show that an improvement of 19.72% for True Positive rate and 1.00% for True Negative rate with a reduction of 19.72% for False Negative rate and 1.00% for False Positive rate have been recorded using the detection algorithms. As a conclusion, this research has made an enhancement with regards to the proposed detection framework and has introduced three new detection algorithms as well as has modified one detection algorithm that contributes to the body of knowledge in network and security in detecting DDoS attacks executed at the application layer.

ABSTRAK

Transaksi antara pengguna dan pelayan web melibatkan beberapa lapisan yang dikenali sebagai lapisan Sambungan Sistem Terbuka (SST). Lapisan aplikasi adalah lapisan tertinggi yang terdedah terhadap manipulasi oleh penyerang yang melaksanakan serangan HTTP *Distributed Denial of Service* (DDoS). Sekiranya berlaku trafik serangan, manipulasi ini menyebabkan trafik HTTP DDoS kelihatan sebagai sah dan rumit untuk dikenali kerana adanya permintaan tajuk dalam. Serangan tersebut menghasilkan pelbagai corak yang menyebabkan ketidakupayaan pengesanan semasa mengenali serangan HTTP DDoS. Pendekatan yang telah digunakan oleh kajian sebelum ini tidak dapat mengesan permintaan GET yang mencurigakan sehingga lalu lintas serangan dapat dianggap sebagai asli. Selain itu, pendekatan semasa tidak dapat mengesan serangan HTTP DDoS melalui proksi dan permintaan palsu. Tujuan penyelidikan ini adalah untuk meningkatkan skema pengesanan dalam mengesan serangan HTTP DDoS. Untuk mencapai tujuan ini, tiga fasa metodologi penyelidikan telah dirangka. Fasa pertama adalah literatur dan analisis masalah, fasa kedua adalah reka bentuk dan pelaksanaan sementara fasa ketiga adalah penentusahan dan pengesahan. Objektif kajian yang digariskan dalam penyelidikan ini adalah selari dengan metodologi kajian. Objektif pertama adalah untuk memperbaiki kerangka pengesanan HTTP DDoS dengan menggunakan komponen baru dan menambahkan atribut baru dalam komponen yang ada. Objektif seterusnya adalah untuk meningkatkan dan membangunkan algoritma pengesanan dengan penggunaan tajuk GET dan atribut penyemak imbas web. Objektif ketiga adalah untuk meningkatkan kadar Positif Benar dan kadar Negatif Benar dan menurunkan kadar Positif Palsu dan kadar Negatif Palsu untuk mengesan serangan HTTP DDoS. Skema pengesanan yang dipertingkatkan merangkumi kerangka pengesanan yang terdiri dari beberapa komponen untuk menunjukkan aliran pengesanan dan algoritma yang terlibat dalam mengesan serangan HTTP DDoS. Rangka kerja pengesanan direka bentuk untuk melakukan pemeriksaan berturutan untuk mengesan corak serangan yang berbeza yang dihasilkan oleh serangan HTTP DDoS. Algoritma pemeriksaan sumber dibentuk untuk meningkatkan pengenalpastian pemula sumber dengan menggunakan atribut penyemak imbas web. Pemeriksaan tajuk permintaan dibuat untuk meningkatkan pengesanan keaslian lalu lintas permintaan HTTP dengan memeriksa keberadaan tajuk GET. Pemeriksaan permintaan dibuat untuk mengesan permintaan palsu menggunakan pertanyaan yang dilampirkan semasa transaksi permintaan GET dan membandingkan permintaan dengan peraturan dan pangkalan data pengesanan. Pemeriksaan proksi diwujudkan untuk mengesan serangan HTTP DDoS yang dilakukan melalui proksi dengan menggunakan tajuk GET proksi yang terlibat semasa permintaan GET. Hasil penyelidikan menunjukkan bahawa peningkatan 19.72% untuk kadar Positif Benar dan 1.00% untuk kadar Negatif Benar dengan penurunan 19.72% untuk kadar Negatif Salah dan 1.00% untuk kadar Positif Palsu telah dicatat menggunakan algoritma pengesanan. Sebagai kesimpulan, penyelidikan ini telah membuat peningkatan berkaitan dengan kerangka pengesanan yang dicadangkan dan memperkenalkan tiga algoritma pengesanan baru serta mengubahsuai satu algoritma pengesanan yang menyumbang kepada pengetahuan tentang rangkaian dan keselamatan ketika mengesan serangan DDoS yang dilakukan di lapisan aplikasi.

TABLE OF CONTENTS

	TITLE	PAGE
	DECLARATION	**iii**
	DEDICATION	**iv**
	ACKNOWLEDGEMENT	**v**
	ABSTRACT	**vi**
	ABSTRAK	**vii**
	TABLE OF CONTENTS	**viii**
	LIST OF TABLES	**xi**
	LIST OF FIGURES	**xii**
	LIST OF ABBREVIATIONS	**xiv**
	LIST OF APPENDICES	**xv**
CHAPTER 1	**INTRODUCTION**	**1**
1.1	Overview	1
1.2	Background of the Problem	2
1.3	Problem Statement	15
1.4	Research Questions	15
1.5	Research Aim	16
1.6	Research Objectives	16
1.7	Research Scope and Assumption	16
1.8	Significant of the Research	17
1.9	Thesis Outline	18
CHAPTER 2	**LITERATURE REVIEW**	**21**
2.1	Introduction	21
2.2	An Overview of Cyber Attacks	22
2.3	HTTP DDoS Taxonomy	24
2.4	HTTP DDoS Attack Categories and Strategies	26
2.5	Defense Life Cycles and Strategies	30
2.6	HTTP Protocol	34
2.7	HTTP Headers Security	37

2.8 Web Application Firewall 37
2.9 HTTP DDoS GET Headers 40
2.10 Existing Approach based on Signature Based Detection 41
2.11 Existing Approach based on Anomaly Detection 49
2.12 Summary of Existing Approaches 56
2.13 Overview of General Analysis 60
2.13.1 Research Dataset 60
2.13.2 Evaluation Method 64
2.13.3 DDoS Defense Strategy 65
2.13.4 Results 66
2.14 Overview of Critical Analysis 68
2.14.1 Detection Strategy 69
2.15 Analysis Outcome and Research Direction 73
2.16 Summary 77

CHAPTER 3 RESEARCH METHODOLOGY 79
3.1 Introduction 79
3.2 Operational Framework 79
3.2.1 Implementation Theories 81
3.2.2 Attack Characteristic 81
3.2.3 Problem Analysis 82
3.2.4 Design and Implementation 87
3.2.5 Verification and Validation 92
3.3 Testbed 96
3.3.1 Software and Hardware Configuration 99
3.4 Summary 99

CHAPTER 4 DESIGN AND IMPLEMENTATION 101
4.1 Introduction 101
4.2 Enhanced Process for Detection HTTP DDoS 101
4.3 Feature Extraction of GET Headers Attributes 105
4.4 Source Inspection Algorithm 106
4.5 GET Headers Inspection Algorithm 109
4.6 Request Query Inspection Algorithm 112
4.7 Proxy Inspection Algorithm 115

4.8 Quadrable Inspection 118
4.9 Summary 120

CHAPTER 5 RESULTS AND DISCUSSION 121
5.1 Introduction 121
5.2 Overview of Experimental Environment 121
5.3 Detection Rule 125
5.4 Overview of Comparison 126
5.5 Comparison with Signaling 128
5.6 Comparison with Logistic Regression 129
5.7 Comparison with HADM 132
5.7.1 Test Case 1 133
5.7.2 Test Case 2 136
5.7.3 Test Case 3 139
5.8 Comparison with Information Entropy 142
5.9 General Discussion 144
5.10 Comparison Discussion with Signaling Technique 149
5.11 Comparison Discussion with Logistic Regression 150
5.12 Comparison Discussion with HADM 151
5.12.1 Source Inspection 153
5.12.2 GET Headers Inspection 155
5.12.3 Request Query Inspection 156
5.13 Comparison Discussion with Information Entropy 158
5.14 Summary 160

CHAPTER 6 CONCLUSION 161
6.1 Achievements and Contributions 161
6.1.1 Enhance Detection Framework 161
6.1.2 Enhance and Develop Detection Algorithms 163
6.1.3 Improve Detection Performance 164
6.1.4 Additional Contributions 165
6.2 Advantages of Enhanced Detection Scheme 166
6.3 Limitation and Recommendation of Future Works 166

REFERENCES 169

LIST OF TABLES

TABLE NO.	TITLE	PAGE
Table 2.1	List of HTTP Request Headers	36
Table 2.2	Detection Performance	46
Table 2.3	Detection Rates	47
Table 2.4	Detection Rules	52
Table 2.5	Summary of Related Works	56
Table 2.6	Old Datasets	61
Table 2.7	Equipment's Employ by Past Studies	64
Table 2.8	Summary of Detection by Prior Studies	72
Table 3.1	Filtering Commands	86
Table 3.2	Problem Situations and Solutions	88
Table 3.3	Confusion Matrix	94
Table 3.4	Research Software's	96
Table 4.1	Components of Enhanced Detection Scheme	105
Table 5.1	Algorithms to Capture True Positive and True Negative	123
Table 5.2	Detection Rules	125
Table 5.3	Comparison of Test Cases	133
Table 5.4	HTTP 2 GET Headers	148
Table 5.5	Result for Test Case 1	152
Table 5.6	Result for Test Case 2	152
Table 5.7	Result for Test Case 3	153
Table 6.1	Outline of Phase 1 to Achieve Objective 1	161
Table 6.2	Outline of Phase 2 to Achieve Objective 2	163
Table 6.3	Outline of Phase 3 to Achieve Objective 3	164

LIST OF FIGURES

FIGURE NO.	TITLE	PAGE
Figure 1.1	Scenario Leading to The Problem	4
Figure 2.1	Cyber-attack Categories	23
Figure 2.2	Taxonomy of HTTP DDoS	25
Figure 2.3	GET Requests and Response	34
Figure 2.4	Request Headers	35
Figure 2.5	Response Headers	35
Figure 2.6	Genuine and Forged GET Headers	40
Figure 2.7	Forged ASCII Code	40
Figure 2.8	HTTP DDoS GET Headers	41
Figure 2.9	Detection Architecture	43
Figure 2.10	Detector Module	43
Figure 2.11	Mass Value for IP Address	45
Figure 2.12	Detection Framework	48
Figure 2.13	Information Entropy Formula	50
Figure 2.14	Detection Framework for Flooding Attack	51
Figure 2.15	Detection Flow Chart	55
Figure 2.16	Detection Performance by Past Studies	67
Figure 3.1	Operational Framework	80
Figure 3.2	Architecture for Simulation of Real Attack	97
Figure 4.1	Enhanced Framework for HTTP DDoS Detection	104
Figure 4.2	Architecture for Source Inspection	106
Figure 4.3	Traffic Flow for Source Inspection	107
Figure 4.4	Architecture for GET Headers Inspection	110
Figure 4.5	Traffic flow for GET Headers Inspection	111
Figure 4.6	Architecture for Request Query Inspection	113
Figure 4.7	Traffic Flow for Request Query Inspection	113
Figure 4.8	Architecture for Proxy Inspection	116
Figure 4.9	Traffic Flow for Proxy Inspection	116
Figure 4.10	Quadrable Inspection Architecture	119

Figure 5.1	Graphically View to Capture True Positive and True Negative	124
Figure 5.2	Evaluation Architecture	124
Figure 5.3	Comparison Architecture with Prior Studies	127
Figure 5.4	Source Inspection and Signaling Technique	128
Figure 5.5	Source Inspection and Logistic Regression	130
Figure 5.6	GET Headers Inspection and Logistic Regression	131
Figure 5.7	Request Query Inspection and Logistic Regression	132
Figure 5.8	Test Case 1: Source Inspection and HADM	134
Figure 5.9	Test Case 1: GET Headers Inspection and HADM	135
Figure 5.10	Test Case 1: Request Query Inspection and HADM	136
Figure 5.11	Test Case 2: Source Inspection and HADM	137
Figure 5.12	Test Case 2: GET Headers Inspection and HADM	138
Figure 5.13	Test Case 2: Request Query Inspection and HADM	139
Figure 5.14	Test Case 3: Source Inspection and HADM	140
Figure 5.15	Test Case 3: GET Headers Inspection and HADM	141
Figure 5.16	Test Case 3: Request Query Inspection and HADM	142
Figure 5.17	Source Inspection and Information Entropy	143
Figure 5.18	GET Headers Inspection and Information Entropy	143

LIST OF ABBREVIATIONS

CPU	-	Central Processing Unit
CRLF	-	Carriage Return Line Feed
DoS	-	Denial of Services
DDoS	-	Distributed Denial of Services
DNS	-	Domain Name System
FTP	-	File Transfer Protocol
HTTP	-	Hyper Text Transfer Protocol
IP	-	Internet Protocol
ISP	-	Internet Services Provider
MIME	-	Multiple Purpose Internet Mail Extensions
OSI	-	Open Systems Interconnection
RFC	-	Request for Comments
SVD-RM	-	Sparse Vector Decomposition and Rhythm Matching
TCP/IP	-	Transmission Control Protocol/Internet Protocol
URL	-	Uniform Resource Locator

LIST OF APPENDICES

APPENDIX	TITLE	PAGE
Appendix A	List of Publication	181
Appendix B	Configuration and Connectivity Test	183 - 192
Appendix C	Investigation	193 - 231
Appendix D	Preliminary Test	233 - 254
Appendix E	Source Inspection	255 - 258
Appendix F	GET Headers Inspection	259 - 261
Appendix G	Request Query Inspection	263 - 267
Appendix H	Public Proxy Inspection	269 - 272

CHAPTER 1

INTRODUCTION

1.1 Overview

Web servers use Hypertext Transfer Protocol (HTTP) to allow users to browse content. The structure of the protocol is fragile since most GET headers in GET requests transactions need not be compulsorily attached. Hence, the protocol is unable to force GET headers to appear in request transactions. This drawback that exists in GET requests allows for Hypertext Transfer Protocol Distributed Denial of Services (HTTP DDoS) capable of emulating legitimate access. Furthermore, the absence of security headers for HTTP DDoS to examine GET headers whether genuine or malicious has further increased the frequency of attacks. The second quarter security report produced by Kaspersky (https://securelist.com/ddos-attacks-in-q2-2017/79241) revealed that HTTP DDoS attacks have increased from 8.43% in the 1st quarter of 2017 to 9.38% in the 2nd quarter of the same year. Singh et al. (2018a) stated that in 2016 DDoS attacks generated about 50,000 GET requests against websites of jewelry shops. The current detection schemes to recognize HTTP DDoS attacks have various limitations. These attacks are varied and forged GET headers often look genuine. This is caused by weak HTTP protocol structure that allows GET headers to be emulated to appear as legitimate and making it more difficult to detect the attacks.

1.2 Background of the Problem

HTTP DDoS needs to establish valid Transmission Control Protocol (TCP) connection and this attack is unrecognizable in network layers (Singh et al., 2018a; Singh and De, 2017a). Yi and Shun-Zheng (2009) stated that when DDoS attacks fails at network layers, attackers switched to a more sophisticated approach known as application layer attack. There are various defense strategies in use to protect networks at layer three and layer four against DDoS attackers as they tend to target the application layer located at layer seven of the networks (Rahman et al., 2017).

DDoS attacks can be categorized into two: Denial of Service (DoS) and Distributed Denial of Service (DDoS). DoS attacks often come from one location while DDoS attacks are more sophisticated originating from multiple resources. McGregory (2013) stated that DDoS attacks occur when vast number of systems located in different geographical locations works together on similar objectives. Iyengar and Ganapathy (2015) mentioned that DDoS attacks are initiated by large groups of attackers that are globally distributed causing high volumes of invalid traffic to grab resources such as networks and memory. Masdari and Jalali (2016) highlighted that DDoS are formed from DOS applied to multiple hosts and networks. The execution of DDoS attacks requires a lot of botnets forming massive Internet traffic causing services on web servers to become unavailable (Zeb et al., 2015). Ramanauskaite et al. (2015) stated that botnets are used to cause DDoS attacks to generate plenty of traffic to consume victims' resources. Meanwhile, DDoS attacks can paralyze targets quicker compared to DoS (Singh and De, 2017b).

DDoS attacks are massive and time-driven where huge simultaneous connections are required to force network bandwidth and victims' resources to their limits. Hence, the use of botnets to generate huge traffic since botnet traffic is difficult to distinguish and looks similar to authentic traffic. Attackers usually launch attacks at the application layers by manipulating HTTP protocol to send enormous GET requests to drain servers' resources such as request queues, servers' memory and processors. This results in the servers reaching their limits and hinders servers from handling legitimate requests (Singh et al., 2015).

HTTP DDoS is difficult to detect since request packets may appear similar to normal request packets (Choi et al., 2010). Differentiating normal and attack traffic is very complex if it is not properly handled (Sree and Bhanu, 2016). Other studies also noted the difficulties of detecting the attack (Idhammad et al., 2018; Yuan et al., 2017; Subramanian et al., 2015; Beitollahi and Deconinck, 2012). Singh et al. (2017b) found that HTTP DDoS are related to application functions and configurations. Meng et al. (2017) stated that HTTP DDoS occurred due to difficulties in recognizing attack patterns. Rahman et al. (2017) noted that poor coding structure in the application layer protocol causes HTTP DDoS. Meanwhile, Jin et al. (2015) opined that HTTP DDoS attacks occur when massive numbers of legitimate GET requests are received by servers.

Choi et al. (2014) stated that security devices are having difficulties to distinguish between genuine HTTP traffic and fake HTTP traffic. The traffic must be handled correctly as it can introduce higher number of false positives. Furthermore, the existence of DDoS as services such as booters makes the execution of the attacks simpler (Hameed and Ali, 2018). Rahman et al. (2017) explained that HTTP DDoS attacks require only minimal botnet army and a single machine with the help of efficient attack scripts can be sufficient to devastate targets (Rahman et al., 2017; Beitollahi and Deconinck, 2013). Figure 1.1 shows the various scenarios in graphical form and the problems addressed in this research.

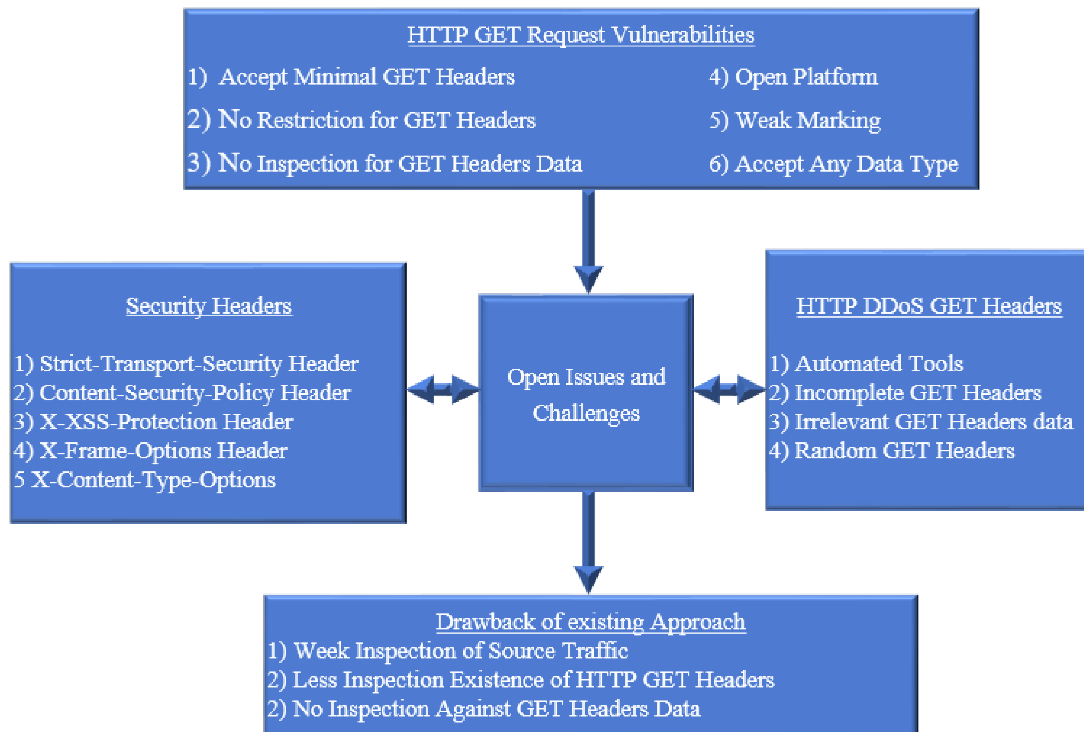

Figure 1.1 Scenario Leading to The Problem

Web servers use HTTP and HTTPS protocols to process requests from users. These protocols are widely used by companies to manage their online presence such as banks, credit card payment gateways, government web servers, online shopping server, social media servers and broadcasting servers. DDoS attacks against web servers may result in monetary loss and loss of public trust (Beitollahi and Deconinck, 2012). Najafabadi et al. (2017) stated that HTTP protocol was designed to process requests and responses to allow communication between clients and web servers. Two methods known as GET and POST are used to accommodate the interactions between clients and web servers. The protocol works when clients start browsing through the content of web servers and this operation is known as GET requests. Once the web servers received GET requests from clients, the servers reply to the requests and this is known as HTTP response. HTTP DDoS attacks happen when attackers exploit these weaknesses.

The weaknesses at the application layer is found at GET requests as it accept minimal GET headers, no restriction for GET headers, no GET headers data inspection, open platforms, weak markings and tendency to accept any data type. Forged request headers make it difficult to detect HTTP DDoS since they can be similar to legitimate GET requests. HTTP DDoS can be generated without complete GET headers. However, the attacks can be simply tagged as genuine users trying to access web browsers to obtain web server contents with complete GET headers. Thus, in order to mask their malicious activities, attackers use GET headers as dummy to forge GET requests making them look authentic. The large number of GET requests may overload web servers. GET requests as part of HTTP operation accept minimal GET headers and these requests are still processed by web servers even if many GET headers are absent such as user-agent, accept, accept-language, accept charset. GET requests accept any headers attached during GET request operations and these requests are not rejected by web servers. Besides accepting any header, the values associated to the headers are not scrutinized to verify the accuracy of the values attached to the headers.

Genuine users usually use web browsers to browse web server contents, however generating GET requests through web browsers is not the only approach that can be used as HTTP is an open platform that allows GET requests to be generated from other platforms such as automated tools. Furthermore, GET headers suffer from connection identity as every GET request made is not identifiable in terms of the source whether through proxies, address translations and direct connections. Singh and Kumar (2016) noted that HTTP protocol version 1.1 is an unconnected protocol, accepting any data type and HTTP is a stateless protocol meaning clients and the servers are in touch with each other only during current requests. Most of the GET headers need not be attached to GET requests (Fielding and Reschke, 2014a; Fielding and Reschke, 2014b; Petersson and Nilsson, 2014; Reid, 2004). Hence, GET requests can be easily emulated, making them look as authentic requests.

Hoque et al. (2015) stated that protocol weaknesses that exist at the application layers provide opportunities for cyber intruders to exploit vulnerabilities and to initiate malicious activities at the application layers such as HTTP, FTP and telnet. Parziale et al. (2006) noted that the use of cookies in HTTP also allows information retrieval. Calzarossa and Massari (2014) analyzed 315,000 GET requests and found that some genuine GET requests do not contain any headers. However, other GET requests are found to have as many as fourteen headers. The studies also mentioned that GET headers and its value can be easily forged. Attackers can take advantage of existing vulnerabilities to disguise attack traffic to appear as genuine. Idhammad et al. (2018) explained that HTTP vulnerabilities have a direct impact on user services. HTTP DDoS exploits HTTP protocol employing it in a legitimate way (Nithyanandam and Dhanapal, 2019). The use of web robots like web crawlers to generate GET requests usually delivers three headers known as Host, Connection and User-Agent compared to GET requests generated by web browsers that tend to include as many GET headers as possible (Calzarossa and Massari, 2014). To overcome the vulnerability of HTTP 1.1, security headers are used to secure clients' access to web server contents.

The are many security headers such as Strict-Transport-Security Header, Content-Security-Policy Header, X-XSS-Protection Header, X-Frame-Options Header and X-Content-Type-Options discussed by several past studies (Lavrenovs and Melon, 2018; Dolnak and Litvik, 2017). However, the use of security headers is limited to securing users from cross-site scripting, click jacking attacks and Multiple Purpose Internet Mail Extensions (MIME) sniffing. Furthermore, the headers above are designed for HTTP POST protocol, thus the need to make them secure to detect forged GET headers produced by HTTP DDoS.

GET headers need not necessarily be attached to GET requests as communication between clients and web servers can still be established without headers. Furthermore, the use of web browsers is not the only way to create GET requests as it can be generated using software known as automated tools. Security headers also have limited functionalities. Due to these shortfalls, attackers execute HTTP DDoS and exploit these vulnerabilities by generating substantial GET requests via automated tools to overload web servers resulting in resource depletion. The

vulnerabilities at the application layers lead to formation of HTTP DDoS (Idhammad et al., 2018; Bawany et al., 2017; Rahman et al., 2017; Osanaiye et al., 2016; Hoque et al., 2015; Choi et al., 2014) capable of making forged genuine GET headers look authentic.

During GET requests, connection headers indicate the status of TCP connections established by GET headers as either Keep-Alive or close. Parziale et al. (2006) stated that once servers reply to GET requests with HTTP response the TCP connections are marked as closed. However, in circumstance of a web server is under HTTP DDoS attack the status of the connection header mark as close with repeatable GET requests received by the web server. Attackers need to send continuously large numbers of GET requests to incapacitate web servers. To do this, attackers use automated tools needing only a small number of machines and short duration of time to generate substantial GET requests. The automated tools give tremendous advantage to attackers in executing HTTP DDoS attacks as only a few machines are needed to create substantial GET requests (Rahman et al., 2017; Beitollahi and Deconinck, 2013).

Minimal GET headers with continuous high GET requests received by web servers are signs of HTTP DDoS. GET requests generated by genuine users contain more GET headers compared to the ones generated through web browsers (Gou et al., 2017) and their request patterns are not excessive like HTTP DDoS as in many online portals, and authentic users provide ratings and reviews of other users and items (Khan and Lee, 2019). Furthermore, GET headers created by HTTP DDoS appear as authentic. However, in certain circumstances GET headers like user-agent generated by the attacks contain irrelevant info, a clear indicator that these GET requests are from malicious sources. Meanwhile, genuine user-agent usually contains information about web browsers name, version and operating system name. Zhang et al. (2015) stated that network administrators and security analysts facing difficulties in identifying malicious user-agent.

Request queries in GET requests show that users need to search for web server contents. However, in case of HTTP DDoS, attackers utilize American Standard Code for Information Interchange (ASCII) to generate massive request queries to mimic genuine search requests. These queries appear longer compared to genuine queries searching for content. Fielding and Reschke (2014b) stated that the limit for GET requests to process headers and its value is not set thus the length of GET requests can be hard to determine. Aside from request queries, GET requests also contain headers known as referrer to indicate web server contents previously access from URLs (Reid, 2004). Hence, HTTP DDoS can take advantage of this by creating headers that contains valid URLs to mask malicious access as genuine access by users. Although the URLs attached to the referrer headers are valid the URLs are irrelevant since the current contents refer to previously access source like URLs of the search engines. Furthermore, the URLs connected to the referrers are randomly generated making these requests look as if they originated from various users.

HTTP DDoS are difficult to detect when public proxies are used together with attackers and genuine users. Public proxies allow attackers to adopt services for free to execute HTTP DDoS. It is difficult to distinguish GET requests generated by user and HTTP DDoS through proxies. This occurs since proxy providers used various names for proxy headers to indicate GET requests originated from proxies. Furthermore, the appearance of the proxy headers is also inconsistent makes it look originated from direct connection. Genuine GET requests through proxies have different names as the headers are not updated by proxy providers and the appearances of the headers are optional displays (Petersson and Nilsson, 2014).

Yadav and Selvakumar (2015) stated that HTTP DDoS deliver random URLs and contain headers known as user-agent and referrer. Sree and Bhanu (2016) noted that HTTP DDoS have equal syntax and are sent in different HTTP formats through multiple GET requests. Sreeram and Vuppala (2017) remarked that excessive login and search requests are among the patterns produced by HTTP DDoS. Dhanapal and Nithyanandam (2019) observed that GET headers created by HTTP DDoS have genuine user-agent, inaccurate URLs and repeated requests toward the same URLs.

GET requests and security headers vulnerabilities have led to various solutions being proposed to detect forged headers from HTTP DDoS. Saleh and Abdul Manaf (2015) proposed a signature-based detection framework to detect HTTP DDoS. The framework is split into three layers that contain components with specific operations. The second layer components used GET headers known as a user-agent, accept, and host as detection attributes. It uses an algorithm to check the existence of GET headers in GET requests to ascertain whether the requestors are using web browsers or automated tools. Yadav and Selvakumar (2015) applied anomaly-based detection with Block Schematic Diagram that adopted two components to detect HTTP DDoS known as training and testing. The training component uses training algorithms to recognize HTTP DDoS while the testing algorithm uses testing algorithms to evaluate the authenticity of HTTP requests. Both studies used similar GET headers detection attributes namely user-agent, Uniform Resource Locator (URLs) and referrer. Liao et al. (2015) designed a detection algorithm using attributes called request interval sequence and request frequency sequence to detect HTTP DDoS. The algorithm measures the time spent by users on specific web pages and observes the sequence of request frequency. The study stated that genuine users spend longer time on specific pages implying longer time intervals compared to shorter time intervals recorded for HTTP DDoS.

The number of GET headers used in both studies (Saleh and Abdul Manaf, 2015; Yadav and Selvakumar, 2015) makes it difficult to detect HTTP DDoS. Both studies used minimal GET headers as detection attributes to check GET requests transactions leading to inaccurate classifications and increased false positive and false negative rates. This is caused by vulnerabilities at the application layers that allow GET requests to be initiated from various platforms such as automated tools allowing attackers to use such tools to execute HTTP DDoS and attached genuine GET headers in GET requests to mask attack traffic to look genuine. Many prior studies have cited the use of automated tool to execute HTTP DDoS (Dhanapal and Nithyanandam, 2019; Sharma and Bhasin, 2018; Wang et al., 2017; Sree and Bhanu, 2016; Yadav and Selvakumar, 2015).

Meanwhile, Yadav and Selvakumar (2015) use of the referrer header is impractical in determining the authenticity of GET requests since inconsistent of the header to be available in each GET requests besides privacy concerns as it has potential to reveal users browsing history (Fielding and Reschke, 2014a). Besides using GET headers, Saleh and Abdul Manaf (2015) utilized JavaScript as the detection attribute to ensure that the requests are from browsers. However, the adoption of JavaScript can be misused and contribute to HTTP DDoS (Kamikubo and Saito, 2017). Saleh and Abdul Manaf (2015) and Yadav and Selvakumar (2015) solutions only assessed GET headers to detect HTTP DDoS and the data associated with the headers are not inspected. Although GET headers data delivered by attacks are real, it is not relevant to processed by web servers. It is important to scrutinize GET headers data for GET requests transactions. HTTP DDoS can emulate GET headers and its value (Nithyanandam and Dhanapal, 2019; Suroto, 2017).

Furthermore, Gulisano et al. (2015) noted that in terms of framework structure the defense frameworks for DDoS show how the source traffic are treated, whether to forward or reject the traffic after inspections. The framework designed by Saleh and Abdul Manaf (2015) adopted single network interface to allow GET requests to reach web servers which also remarked by Bhatia et al. (2014). The framework adopted the concept used by Zargar et al. (2013) where each component inside the framework collaborates with subsequent components to minimize false positive events. However, the framework component designed by Saleh and Abdul Manaf (2015) used less GET headers as the detection attributes in identifying HTTP DDoS through source inspection. Furthermore, the source inspection is located at the second layer near to web servers. The last layer is the last line of defense to protect web servers which exposes web servers to higher attack traffic. Attacks close to web servers are more difficult to deal with since these attacks can more easily overwhelm upstream routers (Gulisano et al., 2015).

Sree and Bhanu (2016) introduced a block diagram using signature-based detection comprising of two components called preprocessor and detector. The attributes used in detection are retrieved from the servers' logs extracted by the preprocessor. The attributes are GET requests per Second (GRPS), Packet Size (PS),

and Response Time (RT). Meanwhile, the detector authenticates GET requests traffic whether they are legitimate or HTTP DDoS. It uses an algorithm called Analytical Hierarchical Process (AHP) to calculate mass values from IP addresses that constitute GET requests, the capacity of GET requests and responses handled by a web server, highest GET request can be received and the maximum time for web server to perform GET request and response. The study also utilized MapReduce to speedup processing since server logs contains a lot of information.

Sree and Bhanu (2016) proposed design has certain drawbacks as the detection needs to extract detection attributes from web server logs. During HTTP DDoS, web servers can be overloaded by large numbers of GET requests continuously generated by HTTP DDoS (Singh et al., 2017b). Continuous attack traffic directed at web servers causes higher false negative rate making web servers unresponsive. Hence, the detector is unable to calculate the mass value to identify the authenticity of GET requests. Meanwhile, the AHP algorithm calculates the mass value for IP addresses that generate GET requests however the calculations can be inaccurate when GET requests originate from proxies. HTTP DDoS can be generated through proxies (Singh et al., 2017b) leading to false positives since the same proxies are utilized by both genuine users and attackers. Furthermore, locating detection inside web servers allow attack traffic to reach web servers although they have been filtered. This minimizes web servers' performance in meeting genuine clients' requests since the servers are handling both genuine and attack traffic.

Wang et al. (2017) introduced a detection framework using anomaly detection. It contains two components called user identification and session identification using user request frequency as detection attributes. The study's detection algorithm measures user request frequency based on information entropy. Sreeram and Vuppala (2017) used machine learning metrics and adopted anomaly detection to differentiate legitimate GET requests and HTTP DDoS based on absolute time interval when users access web servers' contents. The matrix works together with absolute time interval and bat algorithm to achieve fast and early detection. Absolute time interval is the core attribute used to detect HTTP DDoS.

Proposed detection approaches by prior studies (Sreeram and Vuppala, 2017; Wang et al., 2017; Liao et al., 2015) overlooked aggressive user access patterns when accessing web server content. GET requests from aggressive users are tagged as HTTP DDoS having shorter request rates compared to regular users who spend more time moving between pages. Furthermore, user request frequency is also not segregated based on user categories. The absence of user categories may lead to incorrect classifications and contribute to false positive and false negative rates. Legitimate users usually browse web server contents based on interest hence the access patterns are multifarious (Liao et al., 2015). Users behavior in accessing web pages is dynamic due to their habits, needs and their past access behavior may influence their future access behavior (Xiao et al., 2018). Apart from recognizing access patterns for HTTP DDoS Wang et al. (2017) designed a detection framework to detect attacks. However, the framework requires fifteen minutes to recognize HTTP DDoS while Gulisano et al. (2015) opined that defense frameworks need to be able to detect threats immediately.

Bravo and Mauricio (2018) introduced signature-based detection with an interaction detector, a component used to detect is mouse and keyboard attributes in GET requests transactions. The detector contains a classifier algorithm to evaluate the activity of the attributes. The use of the mouse as a detection attribute creates massive web logs (Liao et al., 2015). The study's algorithm is also unable to detect HTTP DDoS executed via online services as attacks through this approach are equal to pressing f5 button on the keyboard repeatability. Furthermore, the classifier algorithm only inspects mouse and keyboard characteristics. It excludes modern devices such as touch screen devices as replacements for the physical mouse and keyboards. This may lead to incorrect classifications with higher false positive and false negative rates. HTTP DDoS attacks can be generated via web browsers by constantly refreshing web pages (Aborujilah and Musa, 2017). Hameed and Ali (2018) stated that DDoS attacks are simple to execute due to its existence as online services to execute the attack.

HTTP DDoS contain forged GET headers, thus detection of these headers leads to detection of HTTP DDoS. HTTP DDoS can occur at communication protocols (Wang et al., 2017). Currently the proposed solutions from past studies unable to addressed forged GET requests due to weak inspection of GET headers. Saleh and Abdul Manaf (2015) dealing with GET headers to differentiate the source requestor originated from the web browser of automated tools. Meanwhile, Yadav and Selvakumar (2015) opined that the use of GET headers can improve detection of HTTP DDoS and adopted GET headers to detect the existence during the GET requests transaction. Although both studies inspected the GET headers, they were unable to distinguish forged GET headers used by attackers from genuine headers. Betarte et al. (2018) noted that information contained in GET headers are specific to GET requests and cannot be identified as application behavior. Web application attacks can be detected by constructing features from HTTP logs with queries and headers (Moradi Vartouni et al., 2019) while Sree and Bhanu (2019) remarked that the logs of GET requests can be adopted to determine HTTP DDoS.

Meanwhile, various other proposed detection designs by prior studies (Bravo and Mauricio, 2018; Sreeram and Vuppala, 2017; Wang et al., 2017; Sree and Bhanu, 2016; Liao et al., 2015) were also unable to detect forged GET headers originated from HTTP DDoS using user access patterns in trying to differentiate GET requests sent to web server. Even though scrutinizing user access pattern able to detect HTTP DDoS attack the proposed solution provided by prior studies has specific limitations. Earlier studies focused on user access pattern to detect HTTP DDoS due to complexity to differentiate characteristics of communication for GET requests created by HTTP DDoS and legitimate user (Aburada et al., 2019).

The main challenge faced by many prior studies is to differentiate forged GET headers attached in GET requests from authentic ones. This makes HTTP DDoS detection complicated since genuine GET requests are almost similar to HTTP DDoS. HTTP DDoS tends to supply false GET headers and data. Thus, the previous research gaps that need to be resolved are forged GET headers and checking for data supplied during GET request transactions. Aceto and Pescape (2015) proposed a solution to detect HTTP manipulation using HTTP response code as detection attributes however

unable to deal with HTTP DDoS. Niakanlahiji et al. (2018) studied the use of headers at the application layers however their study focused on detecting phishing web sites by using response headers. Niu et al. (2019) adopted GET headers such as URI, Host, User-Agent, Request-Method, Request-Version, Accept, Accept-Encoding, Connection, Content-type, Cache-Control, and Content-length to detect malware. Currently no attempt has been made to enhance the use of GET headers to deal with HTTP DDoS.

It is important to ensure the source of GET requests originate from web browsers to web servers as HTTP DDoS often utilize automated tools to generate substantial GET requests. Here, it is crucial to identify source requests to avoid HTTP DDoS. Many prior studies have proposed source request inspections (Wang et al., 2012; Yang et al., 2012; Subbulakshmi et al., 2011; Darapureddi et al., 2010; N et al., 2008) however these studies unable to identify source requestor information whether originating from web browsers or automated tools. There are a number of studies (Lin et al., 2019; Murugesan et al., 2018; Suresh and Sankar Ram, 2018; Singh et al., 2016) proposing a similar approach known as IP traceback to trace source IP addresses that initiated the attacks. However, it can only be used to detect DDoS at network layers. IP traceback is unable to detect HTTP DDoS as the attacks rely on the genuine two way TCP connections (Singh et al., 2016). Saleh and Abdul Manaf (2015) tried to address the issue of source requestors; however, their study utilized minimal GET headers leading to incorrect classifications of GET headers in malicious GET requests. Currently, most solutions are focused on using user access patterns to detect HTTP DDoS instead of scrutinizing GET headers authenticity and source of GET requests.

1.3 Problem Statement

Vulnerabilities that exist at the application layer allow GET headers to be manipulated and made to appear genuine. Furthermore, security headers to secure client communication are also not able to detect HTTP DDoS. Diverse attack patterns making HTTP DDoS attacks look similar to original GET requests are currently being missed by current solutions. Recent studies have proposed various solutions to detect attacks however these solutions have several drawbacks. One such weakness is poor inspection of source traffic whether generated by web browsers or automated tools. Meanwhile, minimal inspection of GET headers during GET requests makes it difficult to differentiate genuine and forged GET headers produced by HTTP DDoS. Furthermore, web server resources are also being overworked trying to process legitimate and illegitimate requests due to absence of request query and data inspection attached to GET headers.

1.4 Research Questions

The following are the research questions that guide the study:

i) How to improve the structure of detection framework to append new components with adoption of new attributes in dealing with forged GET headers and automated tools from HTTP DDoS?

ii) How the increase of GET headers use and adoption of web browsers' attributes can improve inspection of the authenticity of GET requests to detect HTTP DDoS?

iii) How identification of forged GET headers, source requestors and inspection of relevant GET headers data can improve accurate identification and reduce misclassifications in handling genuine and forged GET requests from HTTP DDoS?

1.5 Research Aim

The aim of this research is to detect HTTP DDoS by adopting web browser and GET headers as the detection attributes in enhancing existing detection schemes.

1.6 Research Objectives

The research objectives of this study are:

i) To improve HTTP DDoS detection framework by adoption of new components and append new attributes in the existing component.

ii) To enhance and develop detection algorithms by adoption of GET headers and web browser attributes.

iii) To improve true positive rate and true negative rate and decrease false positive rate and false negative rate.

1.7 Research Scope and Assumption

In order to achieve the objectives, the scope of the study is restrictive and limited to the following conditions:

i) This research focused on detection of DDoS attacks at the application layer that are categorized as request flooding and session flooding and will be referred to as HTTP DDoS throughout this thesis.

ii) The source of the attacks comes from computers and proxies and not from the compromised machines such as botnets or Internet of Things devices (IoT). Recent studies used machines such as computers and laptops to simulate HTTP

DDoS rather than using IoT. Furthermore, the available attack scripts to mimic the attacks are designed to be executed from machines, public proxies and spoofed IP addresses. Since this factor, IoT devices have been excluded from this research.

iii) This research adopted defense strategy known as the victim border.

iv) This research used actual attack scripts to self-generate datasets for investigation of GET headers employed by HTTP DDoS. The actual attack scripts are also employed during evaluation.

v) HTTP 1.1 is used as the protocol in this research as it is widely used by web servers to surf client requests compared to HTTP 2.0 that has not fully implemented worldwide. Furthermore, attackers are currently using HTTP 1.1 to execute HTTP DDoS.

1.8 Significant of the Research

HTTP DDoS attacks are difficult to recognize since it has the ability to generate fake traffic to mimic authentic traffic. Thus, an enhanced detection scheme is needed to recognize the attacks in order to protect web servers. The detection framework and the algorithms introduced in this research can be used by any entities as reference to improve HTTP DDoS detection in light of various attack strategies employed by attackers. The investigation of HTTP DDoS and GET headers vulnerabilities in this research can be used by future researchers to further understand attack patterns and ways to manipulate GET headers to make GET requests look authentic. The results here can be used by others for analysis to reveal the vulnerabilities of current detection schemes. The impact of DDoS attacks would inevitably lead to financial and non-financial loss. This research is expected to provide the way forward in detecting HTTP DDoS.

The research findings are expected to deliver better detection performance to recognize HTTP DDoS compared to existing detection schemes. This research provides a better method to construct an effective detection scheme. It is hoped that the results of the research would benefit both researchers and practitioners working in this area.

1.9 Thesis Outline

This thesis comprises of six chapters as follows:

Chapter 1: Highlights important sections of the research such as background of the problem, problem statement, the research questions and research objectives, research aim, significance of the research, scope and assumptions. The remaining of the thesis is structured as follows.

Chapter 2: Discusses the literature review on HTTP DDoS attacks. It includes a general analysis and critical analysis, and the outcomes help this research to plan the direction of this research in enhancing detection of HTTP DDoS attacks.

Chapter 3: Explain the operations used in this research. The operational framework comprises of three phases and each phase is designed to be in line with the research objectives of this research. Each phase and stages involved in this research are discussed to achieve the research objectives.

Chapter 4: The design of the enhanced detection scheme is explained in this chapter. This involves three new detection algorithms and one modified detection algorithm. The detection algorithms are comprised of several elements such as detection architecture, flowcharts, and programming codes.

Chapter 5: The enhanced detection scheme is evaluated using the confusion matrix. The detection performance for each algorithm is assessed individually by using test

case scenarios. The test case scenarios are designed based on the investigation results in Appendix C. The evaluation of the algorithms is conducted through the Internet while comparisons with prior studies are done in the LAN environment. The results are compared with past studies using similar test case scenarios and attack scripts used by past research.

Chapter 6: This chapter discusses the achievements of the research, contributions, and conclusions. These are followed by the limitations and suggestions for future research.

REFERENCES

Aborujilah, A., & Musa, S. (2017). Cloud-Based DDoS HTTP Attack Detection Using Covariance Matrix Approach. *Journal of Computer Networks and Communications, 2017*, 1-8.

Aburada, K., Arikawa, Y., Usuzaki, S., Yamaba, H., Katayama, T., Park, M., & Okazaki, N. (2019). Use of Access Characteristics To Distinguish Legitimate User Traffic from DDoS Attack Traffic. *Artificial Life and Robotics, 24*(3), 318-323.

Aceto, G., & Pescape, A. (2015). Internet Censorship Detection: A Survey. *Computer Networks, 83*, 381-421.

Adi, E., Baig, Z., & Hingston, P. (2017). Stealthy Denial of Service (DoS) Attack Modelling and Detection for HTTP/2 services. *Journal of Network and Computer Applications, 91*, 1-13.

Almutairi, S., Mahfoudh, S., Almutairi, S., & Alowibdi, J. S. (2020). Hybrid Botnet Detection Based on Host and Network Analysis. *Journal of Computer Networks and Communications, 2020*, 1-16.

Appelt, D., Nguyen, C. D., Panichella, A., & Briand, L. C. (2018). A Machine-Learning-Driven Evolutionary Approach for Testing Web Application Firewalls. *IEEE Transactions on Reliability, 67*(3), 733-757.

Bawany, N. Z., Shamsi, J. A., & Salah, K. (2017). DDoS Attack Detection and Mitigation Using SDN: Methods, Practices, and Solutions. *Arabian Journal for Science and Engineering, 42*(2), 425-441.

Behal, S., & Kumar, K. (2016). Trends in Validation of DDoS Research. International Conference on Computational Modeling and Security (CMS 2016).

Behal, S., & Kumar, K. (2017a). Detection of DDoS Attacks and Flash Events Using Information Theory Metrics–An Empirical Investigation. *Computer Communications, 103*, 18-28.

Behal, S., & Kumar, K. (2017b). Detection of DDoS Attacks and Flash Events Using Novel Information Theory Metrics. *Computer Networks, 116*, 96-110.

Behal, S., Kumar, K., & Sachdeva, M. (2017). Characterizing DDoS Attacks and Flash Events: Review, Research Gaps and Future Directions. *Computer Science Review, 25*, 101-114.

Beitollahi, H., & Deconinck, G. (2012). Analyzing Well-Known Countermeasures Against Distributed Denial of Service Attacks. *Computer Communications, 35*(11), 1312-1332.

Beitollahi, H., & Deconinck, G. (2013). ConnectionScore: A Statistical Technique To Resist Application-Layer DDoS Attacks. *Journal of Ambient Intelligence and Humanized Computing, 5*(3), 425-442.

Betarte, G., Pardo, A., & Martinez, R. (2018). Web Application Attacks Detection Using Machine Learning Techniques. 17th IEEE International Conference on Machine Learning and Applications.

Bhardwaj, A., Subrahmanyam, G., Avasthi, V., Sastry, H., & Goundar, S. (2016). DDoS Attacks, New DDoS Taxonomy and Mitigation Solutions–A Survey. International conference on Signal Processing, Communication, Power and Embedded System (SCOPES).

Bhatia, S., Schmidt, D., Mohay, G., & Tickle, A. (2014). A Framework for Generating Realistic Traffic for Distributed Denial-of-Service Attacks and Flash Events. *Computers & Security, 40*, 95-107.

Brar, H. S., & Kumar, G. (2018). Cybercrimes: A Proposed Taxonomy and Challenges. *Journal of Computer Networks and Communications, 2018*, 1-11.

Bravo, S., & Mauricio, D. (2018). DDoS Attack Detection Mechanism in the Application Layer Using User Features. International Conference on Information and Computer Technologies (ICICT).

Calzarossa, M. C., & Massari, L. (2014). Analysis of Header Usage Patterns of HTTP Request Messages. IEEE International Conference on High Performance Computing and Communications (HPCC), IEEE 6th International Symposium on Cyberspace Safety and Security (CSS) and IEEE 11th International Conference on Embedded Software and Systems (ICESS).

Cheng, J., Li, M., Tang, X., Sheng, V. S., Liu, Y., & Guo, W. (2018a). Flow Correlation Degree Optimization Driven Random Forest for Detecting DDoS Attacks in Cloud Computing. *Security and Communication Networks*, 1-14.

Cheng, J., Zhang, C., Tang, X., Sheng, V. S., Dong, Z., & Li, J. (2018b). Adaptive DDoS Attack Detection Method Based on Multiple-Kernel Learning. *Security and Communication Networks*, 1-19.

Choi, J., Choi, C., Ko, B., & Kim, P. (2014). A Method of DDoS Attack Detection Using HTTP Packet Pattern and Rule Engine in Cloud Computing Environment. *Soft Computing, 18*(9), 1697-1703.

Choi, Y. S., Oh, J. T., Jang, J. S., & Ryou, J. C. (2010). Integrated DDoS Attack Defense Infrastructure for Effective Attack Prevention. 2010 2nd International Conference on Information Technology Convergence and Services.

Darapureddi, A., Mohandas, R., & Pais, A. R. (2010). Throttling DDoS Attacks Using Discrete Logarithm Problem. International Conference on Security and Cryptography (SECRYPT).

Dhanapal, A., & Nithyanandam, P. (2017). An Effective Mechanism to Regenerate HTTP Flooding DDoS attack Using Real Time Data Set. International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICICT).

Dhanapal, A., & Nithyanandam, P. (2019). An OpenStack Based Cloud Testbed Framework for Evaluating HTTP Flooding Attacks. *Wireless Networks*.

Dick, U., & Scheffer, T. (2016). Learning to Control a Structured Prediction Decoder for Detection of HTTP-Layer DDoS Attackers. *Machine Learning, 104*(2-3), 385-410.

Dolnak, I. (2017). Implementation of Referrer Policy in Order To Control HTTP Referer Header Privacy. International Conference on Emerging eLearning Technologies and Applications (ICETA).

Dolnak, I., & Litvik, J. (2017). Introduction to HTTP Security Headers And Implementation of HTTP Strict Transport Security (HSTS) Header for HTTPS Enforcing. 15th International Conference on Emerging eLearning Technologies and Applications (ICETA).

Dong, Y., Zhang, Y., Ma, H., Wu, Q., Liu, Q., Wang, K., & Wang, W. (2018). An Adaptive System for Detecting Malicious Queries in Web Attacks. *Science China Information Sciences, 61*(3).

Eid, M. S. A., & Aida, H. (2017). Trustworthy DDoS Defense: Design, Proof of Concept Implementation and Testing. *IEICE Transactions on Information and Systems, E100.D*(8), 1738-1750.

Fielding, R., & Reschke, J. (2014a). *Hypertext transfer protocol (HTTP/1.1): Semantics and content* (2070-1721). Retrieved from https://tools.ietf.org/html/rfc7231

Fielding, R. T., & Reschke, J. F. (2014b). *Message Syntax and Routing*. IETF RFC 7230, June.

Gou, G., Bai, Q., Xiong, G., & Li, Z. (2017). Discovering Abnormal Behaviors Via HTTP Header Fields Measurement. *Concurrency and Computation: Practice and Experience, 29*(20), e3926.

Gu, Y., Wang, Y., Yang, Z., Xiong, F., & Gao, Y. (2017). Multiple-Features-Based Semisupervised Clustering DDoS Detection Method. *Mathematical Problems in Engineering, 2017*, 1-10.

Gulisano, V., Callau-Zori, M., Fu, Z., Jiménez-Peris, R., Papatriantafilou, M., & Patiño-Martínez, M. (2015). STONE: A Streaming DDoS Defense Framework. *Expert Systems with Applications, 42*(24), 9620-9633.

Hameed, S., & Ali, U. (2018). HADEC: Hadoop-Based Live DDoS Detection Framework. *EURASIP Journal on Information Security, 2018*(1), 2-19.

Hoque, N., Bhattacharyya, D. K., & Kalita, J. K. (2015). Botnet in DDoS Attacks: Trends and Challenges. *IEEE Communications Surveys & Tutorials, 17*(4), 2242-2270.

Hosseini, S., & Azizi, M. (2019). The Hybrid Technique for DDoS Detection with Supervised Learning Algorithms. *Computer Networks, 158*, 35-45.

Idhammad, M., Afdel, K., & Belouch, M. (2018). Detection System of HTTP DDoS Attacks in a Cloud Environment Based on Information Theoretic Entropy and Random Forest. *Security and Communication Networks*, 1-13.

Idris, I., Fabian, O. B., M. Abdulhamid, S. i., Olalere, M., & Meshach, B. (2017). Distributed Denial of Service Detection using Multi Layered Feed Forward Artificial Neural Network. *International Journal of Computer Network and Information Security, 9*(12), 29-35.

Iyengar, N. C. S. N., & Ganapathy, G. (2015). An Effective Layered Load Balance Defensive Mechanism against DDoS Attacks in Cloud Computing Environment. *International Journal of Security and Its Applications, 9*(7), 17-36.

Jazi, H. H., Gonzalez, H., Stakhanova, N., & Ghorbani, A. A. (2017). Detecting HTTP-Based Application Layer DoS Attacks on Web Servers in the Presence of Sampling. *Computer Networks, 121*, 25-36.

Jia, B., Huang, X., Liu, R., & Ma, Y. (2017). A DDoS Attack Detection Method Based on Hybrid Heterogeneous Multiclassifier Ensemble Learning. *Journal of Electrical and Computer Engineering*, 1-9.

Jin, W., Min, Z., Xiaolong, Y., Keping, L., & Jie, X. (2015). HTTP-sCAN: Detecting HTTP-Flooding Attack by Modeling Multi-Features of Web Browsing Behavior from Noisy Web-Logs. *China Communications, 12*(2), 118-128.

Johnson Singh, K., Thongam, K., & De, T. (2016). Entropy-Based Application Layer DDoS Attack Detection Using Artificial Neural Networks. *Entropy, 18*(10), 2-17.

Kamikubo, R., & Saito, T. (2017). Browser-Based DDoS Attacks Without Javascript. *International Journal of Advanced Computer Science and Applications, 8*(12), 276-280.

Karoui, K. (2016). Security Novel Risk Assessment Framework Based on Reversible Metrics: A Case Study of DDoS Attacks on an E-Commerce Web Server. *International Journal of Network Management, 26*(6), 553-578.

Khan, J., & Lee, S. (2019). Implicit User Trust Modeling Based on User Attributes and Behavior in Online Social Networks. *IEEE Access, 7*, 142826-142842.

Kumar, V., & Kumar, K. (2016a). Classification of DDoS Attack Tools and Its Handling Techniques and Strategy At Application Layer. International Conference on Advances in Computing, Communication, & Automation (ICACCA) (Fall).

Kumar, V., & Kumar, K. (2016b). Classification of DDoS Attack Tools and Its Handling Techniques and Strategy At Application Layer. 2nd International Conference on Advances in Computing, Communication, & Automation (ICACCA)(Fall) (pp. 1-6). IEEE.

Lavrenovs, A., & Melon, F. J. R. (2018). HTTP Security Headers Analysis of Top One Million Websites. 10th International Conference on Cyber Conflict (CyCon).

Liao, Q., Li, H., Kang, S., & Liu, C. (2015). Application Layer DDoS Attack Detection Using Cluster with Label Based on Sparse Vector Decomposition and Rhythm Matching. *Security and Communication Networks, 8*(17), 3111-3120.

Lin, H.-C., Wang, P., & Lin, W.-H. (2019). Implementation of a PSO-Based Security Defense Mechanism for Tracing the Sources of DDoS Attacks. *Computers, 8*(4), 88.

Liu, C., Yang, J., & Wu, J. (2020). Web Intrusion Detection System Combined with Feature Analysis and SVM Optimization. *EURASIP Journal on Wireless Communications and Networking*(1).

Ludin, S., & Garza, J. (2017). *Learning HTTP/2: A Practical Guide for Beginners*: O'Reilly Media, Inc.

Masdari, M., & Jalali, M. (2016). A Survey and Taxonomy of DoS Attacks in Cloud Computing. *Security and Communication Networks, 9*(16), 3724-3751.

McGregory, S. (2013). Preparing for The Next DDoS Attack. *Network Security, 2013*(5), 5-6.

Meng, B., Andi, W., Jian, X., & Fucai, Z. (2017). DDOS Attack Detection System Based on Analysis of Users' Behaviors for Application Layer. International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC).

Mirvaziri, H. (2017). A New Method to Reduce the Effects of HTTP-Get Flood Attack. *Future Computing and Informatics Journal, 2*(2), 87-93.

Moradi Vartouni, A., Teshnehlab, M., & Sedighian Kashi, S. (2019). Leveraging Deep Neural Networks for Anomaly-Based Web Application Firewall. *IET Information Security, 13*(4), 352-361.

Munivara Prasad, K., Rama Mohan Reddy, A., & Venugopal Rao, K. (2017). BIFAD: Bio-Inspired Anomaly Based HTTP-Flood Attack Detection. *Wireless Personal Communications, 97*(1), 281-308.

Murugesan, V., Selvaraj, M. S., & Yang, M.-H. (2018). HPSIPT: A High-Precision Single-Packet IP Traceback Scheme. *Computer Networks, 143*, 275-288.

Myint Oo, M., Kamolphiwong, S., Kamolphiwong, T., & Vasupongayya, S. (2019). Advanced Support Vector Machine- (ASVM-) Based Detection for Distributed Denial of Service (DDoS) Attack on Software Defined Networking (SDN). *Journal of Computer Networks and Communications*, 1-12.

N, V., V, D. C., & D, S. (2008). An Effective Defense Against Distributed Denial of Service in GRID. International Conference on Emerging Trends in Engineering and Technology.

Najafabadi, M. M., Khoshgoftaar, T. M., Calvert, C., & Kemp, C. (2017). User Behavior Anomaly Detection for Application Layer DDoS Attacks. International Conference on Information Reuse and Integration (IRI).

Nam, S. Y., & Djuraev, S. (2014). Defending HTTP Web Servers Against DDoS Attacks Through Busy Period-Based Attack Flow Detection. *KSII Transactions on Internet and Information Systems, 8*(7), 2512-2531.

Ni, T., Gu, X., Wang, H., & Li, Y. (2013). Real-Time Detection of Application-Layer DDoS Attack Using Time Series Analysis. *Journal of Control Science and Engineering*, 1-6.

Niakanlahiji, A., Chu, B.-T., & Al-Shaer, E. (2018). PhishMon: A Machine Learning Framework for Detecting Phishing Webpages. International Conference on Intelligence and Security Informatics (ISI).

Nithyanandam, P., & Dhanapal, A. (2019). The Slow HTTP Distributed Denial of Service Attack Detection in Cloud. *Scalable Computing: Practice and Experience, 20*(2), 285-298.

Niu, W., Li, T., Zhang, X., Hu, T., Jiang, T., & Wu, H. (2019). Using XGBoost to Discover Infected Hosts Based on HTTP Traffic. *Security and Communication Networks*, 1-11.

Osanaiye, O., Choo, K.-K. R., & Dlodlo, M. (2016). Distributed Denial of Service (DDoS) Resilience in Cloud: Review and Conceptual Cloud DDoS Mitigation Framework. *Journal of Network and Computer Applications, 67*, 147-165.

Pandiaraja, P., & Manikandan, J. (2015). Web Proxy Based Detection and Protection Mechanisms Against Client Based HTTP Attacks. International Conference on Circuits, Power and Computing Technologies

Parziale, L., Liu, W., Matthews, C., Rosselot, N., Davis, C., Forrester, J., & Britt, D. T. (2006). *TCP/IP Tutorial and Technical Overview*: IBM Redbooks.

Petersson, A., & Nilsson, M. (2014). Forwarded HTTP Extension.

Prasad, K. M., Reddy, A. R. M., & Rao, K. V. (2017). BARTD: Bio-Inspired Anomaly Based Real Time Detection of Under Rated App-DDoS Attack on Web. *Journal of King Saud University - Computer and Information Sciences, 32(1)*, 73-87.

Procopiou, A., Komninos, N., & Douligeris, C. (2019). ForChaos: Real Time Application DDoS Detection Using Forecasting and Chaos Theory in Smart Home IoT Network. *Wireless Communications and Mobile Computing*, 1-14.

Rahman, R. u., Tomar, D. S., & A.V, J. (2017). Application Layer DDOS Attack Detection Using Hybrid Machine Learning Approach. *International Journal of Security and Its Applications, 11*(4), 85-96.

Rai, A., & Challa, R. K. (2016). Survey on Recent DDoS Mitigation Techniques and Comparative Analysis. Second International Conference on Computational Intelligence & Communication Technology.

Ramanauskaite, S., Goranin, N., Cenys, A., & Juknius, J. (2015). Modelling Influence of Botnet Features on Effectiveness of DDoS Attacks. *Security and Communication Networks, 8*(12), 2090-2101.

Rao, U. H., & Nayak, U. (2014). Understanding Networks and Network Security. In U. H. Rao & U. Nayak (Eds.), *The InfoSec Handbook: An Introduction to Information Security* (pp. 187-204). Berkeley, CA: Apress.

Rashidi, B., Fung, C., & Bertino, E. (2017). A Collaborative DDoS Defence Framework using Network Function Virtualization. *IEEE Transactions on Information Forensics and Security, 12(10)*, 2483-2497.

Reid, F. (2004). 4 - HTTP: Communicating with Web Servers. In F. Reid (Ed.), *Network programming in .NET* (pp. 87-130). Burlington: Digital Press.

Saleh, M. A., & Abdul Manaf, A. (2015). A Novel Protective Framework for Defeating HTTP-Based Denial of Service and Distributed Denial of Service Attacks. *Scientific World Journal*, 2-19.

Shah, M., Khattak, S., Farooq, M., Jan, S., Qureshi, M., Jan, N., & Ahmed, S. (2019). A Secured and Enhanced Mitigation Framework for DDoS Attacks. *Journal of Mechanics of Continua and Mathematical Sciences, 14*(6), 985-1004.

Shameli-Sendi, A., Pourzandi, M., Fekih-Ahmed, M., & Cheriet, M. (2015). Taxonomy of Distributed Denial of Service mitigation approaches for cloud computing. *Journal of Network and Computer Applications, 58*, 165-179.

Sharma, A., & Bhasin, A. (2018). Critical Investigation of Denial of Service and Distributed Denial of Service Models and Tools. International Conference on Advances in Computing, Communication Control and Networking (ICACCCN).

Shen, Y., Yang, W., & Huang, L. (2018). Concealed in Web Surfing: Behavior-Based Covert Channels in HTTP. *Journal of Network and Computer Applications, 101*, 83-95.

Shiaeles, S. N., & Papadaki, M. (2014). FHSD: An Improved IP Spoof Detection Method for Web DDoS Attacks. *The Computer Journal, 58*(4), 892-903.

Singh, B., Kumar, K., & Bhandari, A. (2015). Simulation Study of Application Layer DDoS Attack. International Conference on Green Computing and Internet of Things (ICGCIoT).

Singh, K., Dhindsa, K. S., & Bhushan, B. (2017a). Distributed Defense: An Edge over Centralized Defense against DDos Attacks. *International Journal of Computer Network & Information Security, 9*(3), 36-44.

Singh, K., Singh, P., & Kumar, K. (2016). A Systematic Review of IP Traceback Schemes for Denial of Service Attacks. *Computers & Security, 56*, 111-139.

Singh, K., Singh, P., & Kumar, K. (2017b). Application layer HTTP-GET flood DDoS attacks: Research landscape and challenges. *Computers & Security, 65*, 344-372.

Singh, K., Singh, P., & Kumar, K. (2018a). Fuzzy-Based User Behavior Characterization to Detect HTTP-GET Flood Attacks. *International Journal of Intelligent Systems and Applications, 10*(4), 29-40.

Singh, K., Singh, P., & Kumar, K. (2018b). User Behavior Analytics-Based Classification of Application Layer HTTP-GET Flood Attacks. *Journal of Network and Computer Applications, 112*, 97-114.

Singh, K. J., & De, T. (2017a). Analysis of Application Layer DDoS Attack Detection Parameters Using Statistical Classifiers. *Internetworking Indonesia, 9*(2), 23-31.

Singh, K. J., & De, T. (2017b). MLP-GA Based Algorithm to Detect Application Layer DDoS Attack. *Journal of Information Security and Applications, 36*, 145-153.

Singh, S. R., & Kumar, D. S. (2016). An Overview of World Wide Web Protocol (Hypertext Transfer Protocol and Hypertext Transfer Protocol Secure). *International Journal of Advanced Research in Computer Science and Software Engineering, 6*(5), 396-399.

Sree, T. R., & Bhanu, S. M. S. (2016). HADM: Detection of HTTP GET Flooding Attacks by Using Analytical Hierarchical Process and Dempster-Shafer Theory with MapReduce. *Security and Communication Networks, 9*(17), 4341-4357.

Sree, T. R., & Bhanu, S. M. S. (2019). Detection of HTTP Flooding Attacks in Cloud Using Fuzzy Bat Clustering. *Neural Computing and Applications, 32*(13), 9603-9619.

Sree, T. R., & Saira Bhanu, S. M. (2018). Investigation of Application Layer DDoS Attacks Using Clustering Techniques. *International Journal of Wireless and Microwave Technologies, 8*(3), 1-13.

Sreeram, I., & Vuppala, V. P. K. (2017). HTTP Flood Attack Detection in Application Layer Using Machine Learning Metrics and Bio Inspired Bat Algorithm. *Applied Computing and Informatics*.

Subbulakshmi, T., Guru, I. A., & Shalinie, S. M. (2011). Attack Source Identification at Router Level in Real Time Using Marking Algorithm Deployed in Programmable Routers. International Conference on Recent Trends in Information Technology (ICRTIT).

Subramanian, K., Gunasekaran, P., & Selvaraj, M. (2015). Two Layer Defending Mechanism against DDoS Attacks. *International Arab Journal of Information Technology (IAJIT), 12*(4), 317-324.

Suresh, S., & Sankar Ram, N. (2018). Feasible DDoS Attack Source Traceback Scheme by Deterministic Multiple Packet Marking Mechanism. *The Journal of Supercomputing, 76*(6), 4232-4246.

Suroto, S. (2017). A Review of Defense Against Slow HTTP Attack. *JOIV: International Journal on Informatics Visualization, 1*(4), 127-134.

Tekerek, A., & Bay, O. F. (2019). Design and Implementation of an Artificial Intelligence-Based Web Application Firewall Model. *Neural Network World, 29*(4), 189-206.

Tian, Z., Luo, C., Qiu, J., Du, X., & Guizani, M. (2020). A Distributed Deep Learning System for Web Attack Detection on Edge Devices. *IEEE Transactions on Industrial Informatics, 16*(3), 1963-1971.

Tripathi, N., & Hubballi, N. (2018). Slow Rate Denial of Service Attacks Against HTTP/2 and Detection. *Computers & Security, 72*, 255-272.

Wall, D. (2004). Chapter 3 - HTTP in PHP. In D. Wall (Ed.), *Multi-Tier Application Programming with PHP* (pp. 21-43). San Francisco: Morgan Kaufmann.

Wang, B., Zheng, Y., Lou, W., & Hou, Y. T. (2015). DDoS Attack Protection in The Era of Cloud Computing and Software-Defined Networking. *Computer Networks, 81*, 308-319.

Wang, F., Huang, L., Miao, H., & Tian, M. (2014a). A Novel Distributed Covert Channel in HTTP. *Security and Communication Networks, 7*(6), 1031-1041.

Wang, F., Wang, X., Su, J., & Xiao, B. (2012). VicSifter: A Collaborative DDoS Detection System with Lightweight Victim Identification. International Conference on Trust, Security and Privacy in Computing and Communications.

Wang, J., Yang, X., Zhang, M., Long, K., & Xu, J. (2014b). HTTP-SOLDIER: An HTTP-Flooding Attack Detection Scheme with the Large Deviation Principle. *Science China Information Sciences, 57*(10), 1-15.

Wang, Y., Liu, L., Si, C., & Sun, B. (2017). A Novel Approach for Countering Application Layer DDoS Attacks. IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC).

Wong, C. (2000). *Http Pocket Reference: Hypertext Transfer Protocol*: O'Reilly Media, Inc.

Xiao, Y., Shi, J., Zheng, W., Wang, H., & Hsu, C.-H. (2018). Enhancing Collaborative Filtering by User-User Covariance Matrix. *Mathematical Problems in Engineering, 2018*, 9740402.

Yadav, S., & Selvakumar, S. (2015). Detection of Application Layer DDoS Attack by Modeling User Behavior Using Logistic Regression. 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions).

Yadav, S., & Subramanian, S. (2016). Detection of Application Layer DDoS attack by feature learning using Stacked AutoEncoder. International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT).

Yang, L., Zhang, T., Song, J., Wang, J. S., & Chen, P. (2012). Defense of DDoS Attack for Cloud Computing. International Conference on Computer Science and Automation Engineering (CSAE).

Ye, J., Cheng, X., Zhu, J., Feng, L., & Song, L. (2018). A DDoS Attack Detection Method Based on SVM in Software Defined Network. *Security and Communication Networks*, 1-8.

Yevsieieva, O., & Helalat, S. M. (2017). Analysis of The Impact of the Slow HTTP DOS and DDOS Attacks on The Cloud Environment. 4th International

Scientific-Practical Conference Problems of Infocommunications. Science and Technology (PIC S&T).

Yi, X., & Shun-Zheng, Y. (2009). Monitoring the Application-Layer DDoS Attacks for Popular Websites. *IEEE/ACM Transactions on Networking, 17*(1), 15-25.

Yuan, X., Li, C., & Li, X. (2017). DeepDefense: Identifying DDoS Attack Via Deep Learning. IEEE International Conference on Smart Computing (SMARTCOMP).

Zargar, S. T., Joshi, J., & Tipper, D. (2013). A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks. *IEEE Communications Surveys & Tutorials, 15*(4), 2046-2069.

Zeb, K., Baig, O., & Asif, M. K. (2015). DDoS Attacks and Countermeasures in Cyberspace. 2nd World Symposium on Web Applications and Networking (WSWAN).

Zhang, X., Zhang, Y., Altaf, R., & Feng, X. (2018). A Multi-agent System-based Method of Detecting DDoS Attacks. *International Journal of Computer Network and Information Security, 10*(2), 53-64.

Zhang, Y., Mekky, H., Zhang, Z. L., Torres, R., Lee, S. J., Tongaonkar, A., & Mellia, M. (2015). Detecting Malicious Activities With User-Agent-Based Profiles. *International Journal of Network Management, 25*(5), 306-319.

Zhou, W., Jia, W., Wen, S., Xiang, Y., & Zhou, W. (2014). Detection and Defense of Application-Layer DDoS Attacks in Backbone Web Traffic. *Future Generation Computer Systems, 38*, 36-46.

Ziyad R. Al Ashhab, Mohammed Anbar, Manmeet Mahinderjit Singh, Kamal Alieyan, & Ghazaleh, W. I. A. (2019). Detection of HTTP Flooding DDoS Attack using Hadoop with MapReduce : A Survey. *International Journal of Advanced Trends in Computer Science and Engineering, 8*, 1-7.

APPENDIX A LIST OF PUBLICATIONS

Journal with Impact Factor

1) Jaafar, A. G., Ismail, S. A., Abdullah, M. S., Kama, N., Azmi, A., & Yusop, O. M. (2020). Recent Analysis of Forged Request Headers Constituted by HTTP DDoS. Sensors (Basel), 20(14). doi:10.3390/s20143820.
(Q1, IF: 3.275)

Indexed Journal

1) Ghafar A. Jaafar, Shahidan M. Abdullah, & Adli, S. (2019). Review of Recent Detection Methods for HTTP DDoS Attack. Journal of Computer Networks and Communications, 2019, 1-10. doi:10.1155/2019/1283472.
(Indexed by ISI).

2) Ghafar A. Jaafar, Abdullah, S. M., & Adli, S. (2019). A Review of Technique to Self-Generate DDoS Dataset. International Journal of Advanced Trends in Computer Science and Engineering, 8(4). doi:10.30534/ijatcse/2019/88842019. **(Indexed by SCOPUS)**.

3) Ghafar A. Jaafar, Shahidan M. Abdullah, & Adli, S. (2019). Enhanced Detection Algorithms to Detect HTTP DDoS. International Journal of Advanced Trends in Computer Science and Engineering, 8(4), 1609-1520. doi:10.30534/ijatcse/2019/86842019. **(Indexed by SCOPUS)**.

APPENDIX B CONFIGURATION AND CONNECTIVITY TEST

B.1 Firewall Configuration

Formation of lab experiment requires the firewall to be adopted in order to mimic the real environment of the network and the attack. Thus, the firewall requires a proper configuration to ensure that the device is able to operate as planned. The firewall is configured to have three segments to segregate traffic between client and server. The network segmentation was used in this research to simulate source of malicious traffics which comes from the user segment. Furthermore, in a real environment, execution of attack comes from equal segments within the server that is perceived to be impossible to happen as the segment is restricted with several rules to secure the segment. Aside from that, launching the attack with the same segment will result in the connection to be blocked immediately as the source is easy to trace. Due to this reason, this research focused on attack which comes from different server segments. The IP address utilize in this research comes from class C network with fourteen IP address available of each network segment. Table B.1 shows the present network segment that was configured at the firewall.

Table B.1 Present Network Segment Configured at Firewall

Network Interface Name	Network Segment	Gateway IPs
Trusted_Zone_1	192.168.2.0/28	192.168.2.1
Trusted_Zone_2	192.168.3.0/28	192.168.3.1
DMZ_Zone	192.168.4.0/28	192.168.4.1

The lab environment required HTTP and DNS protocol to allow communication between the client and the server. Besides that, connection between the client and the server requires several tests to ensure that a connection can be established successfully. Hence, ICMP and Telnet protocol are required to be enabled. Table B.2 presents the required protocol.

Table B.2 Network Protocol

Protocol	Details
HTTP	Allow client to browse the content of the web server.
DNS	Allow client to browse the content of the web server by using URL.
ICMP	To allow ping connection among machines in network, specifically used for connection testing.
Telnet	To test network port

B.2 Firewall Routing Table

The network segmentation adopted in this research requires network routing to allow IP addresses from different classes to communicate to one another. This research utilized the static network route which requires the network range to be defined manually. Precise configuration of network routing in experimental lab for this research is highly necessary to prevent communication failure between network segments. To ensure communication between different segments in the internal network can be established successfully, the routing gateway was set to all networks (0.0.0.0). All networks allow all existing network segment to communicate to one another. The routing gateway must be defined with an IP address except for 0.0.0.0 in which the internal network is required to refer to outside resources such as the Internet access. Table B.3 illustrates the routing table required by the firewall.

Table B.3 Firewall Routing Table

Network	Gateway
192.168.2.0/24	0.0.0.0
192.168.3.0/24	0.0.0.0
192.168.4.0/24	0.0.0.0

B.3 Firewall Security Policy

The formation of firewall security policy required vigilant steps as invalid settings will lead to connection failure regardless whether the routing table and DHCP are correctly configured. The existence of the firewall security policy is to provide action permit or denial when the network traffic is traveling to another segment. The firewall security policy was built based on several components such as source and destination interface, source and destination network and network protocol to allow further action such as to allow or to block. Table B.4 indicates the security policy required for the lab experiment.

Table B.4 Firewall Security Policy

No.	Connection Type: Http Request	
1.	**Source Interface**	Trusted_Zone_1
	Source Network	192.168.2.0/28
	Destination Interface Name	DMZ_Zone
	Destination Network	192.168.4.0/28
	Service	HTTP, DNS, ICMP, Telnet
Connection Type: Http Request		
2.	**Source Interface**	Trusted_Zone_2
	Source Network	192.168.3.0/28
	Destination Interface Name	DMZ_Zone
	Destination Network	192.168.4.0/28
	Service	HTTP, DNS, ICMP, Telnet
Connection Type: Web Server		
3.	**Source Interface**	DMZ_Zone
	Source Network	192.168.4.0/28
	Destination Interface Name	Trusted_Zone_1, Trusted_Zone_2
	Destination Network	192.168.2.0/28, 192.168.3.0/28
	Service	HTTP, DNS, ICMP, Telnet

B.4 DHCP Configuration

A distribution of IP address is divided into two types known as static and dynamic. The static IP address requires a network administrator to key in the IP address manually into a machine. In contrast, for dynamic IP address, the IP address will automatically be distributed by DHCP. The configuration of lab environment utilized a firewall which acted as DHCP to ease the IP address distribution based on the scope defined. The DHCP distributes the Internet Protocol Version Four (IPV4) which comes from class C network where the first octet of this IP begins with 192.x.x.x and the size of the subnet used /28 which will allocate fourteen IP addresses.

The DHCP will distribute IP addresses to Trusted_Zone_1 and Trusted_Zone_2 as these are the segments that belonged to the client. The DMZ_Zone also utilized IP address distribute by DHCP this to ease the configuration of static IP address which need to manually be configured. Commonly in the real production network environment the use of IP Address from DHCP supposed to be prevented as it will cause accessibility problems when the DHCP starts renewing the distributed IP address. However, for the purposed of this research usage of DHCP IP address for web server is applicable to speed up the process of web server obtaining IP address. Table B.5 demonstrates the scope of IP addresses utilized by DHCP.

Table B.5 DHCP Scope

Scope	**Gateway**
192.168.2.0/28	192.168.2.1
192.168.3.0/28	192.168.3.1
192.168.4.0/28	192.168.4.1

B.5 Web Server Configuration

A web server running on Windows Server 2016 and operating with Microsoft Internet Information Service (IIS) were used in this research to handle HTTP protocol request and response. A HTML web was developed using HTML programming language to become the target for this experiment. The web server also acts as the Domain Name Resolution (DNS) to allow clients to contact a web server by using Uniform Resource Locator (URL). The use of DNS will facilitate users to remember the URL of the online resources which used HTTP protocol to operate like online shopping, web portal and content management. A user will face difficulties without DNS implementation as a user needs to remember an IP address of the web server to browse the contents of the online resources. Table B.6 illustrates the web server configuration.

Table B.6 Web Server Configuration

Web Services Name	Internet Information Service (IIS)
DNS	Microsoft DNS Server
Operating System	Windows Server
Connection Type	HTTP
Port	80
Programming Language	HTML
URL	http://lab.com.my
Memory	8 Gb
Processor Speed and Cores	3.7 Gbps, 4 Cores
Network Segment	DMZ_Zone

B.6 Client Configuration

This research required four clients to be running under a virtual machine and physical environment that are segregated into normal and attack machines. The machines entail the IP address in order to establish a connection to a web server. In order to ease the configuration of manually setting the IP addresses, Dynamic Host Configuration Protocol (DHCP) is used. DHCP is an automatic IP address distribution that is used to automatically release IP address to a client. A client web browser will install ready-made tools to simulate legitimate web browser which repeatedly sends GET requests against a web server. Table B.7 shows the client configuration for experiment settings used in this research.

Table B.7 Client Configuration for Experiment Settings

No.	Operating System	Network Segment	IP Address	Web Browser
1.	Windows 10	Trusted_Zone_1	DHCP	Google Chrome
2.	Windows 10			
3.	Ubuntu 16.04			
4.	Windows 10	Trusted_Zone_2	DHCP	
5.	Windows 10			

B.7 Connectivity Test

The previous section explains the configuration conducted at client, firewall and server. Therefore, to ensure all earlier settings are performed correctly, connectivity tests need to be implemented. There were several commands involved in executing the connectivity test and will be explained in detail in subsequent sub section. Apart from that, the connectivity test must be executed in sequence to ease troubleshooting activity to identify the root cause of the failure when facing connectivity issue. Each test must complete with a no error as failure to resolve existing error will result in the failure to execute the next connectivity test.

B.8 Connectivity Test – DHCP

Connection test for DHCP was done to verify whether the machines was able to acquire an IP address from the DHCP server. The test requires several commands based on the platform utilized. Table B.8 presents the DHCP test command.

Table B.8 DHCP Test Commands

Commands	Platform	Details
ipconfig /release	Windows	Release the current IP address held by client
sudo dhclient -r eth0	Ubuntu	
ipconfig /renew	Windows	Request IP address to DHCP
sudo dhclient eth0	Ubuntu	
ipconfig /all	Windows	Inspection of whether client receive IP address from DHCP or vice versa
ifconfig	Ubuntu	

The result obtained from the command was as expected and the machine was able to gain the IP address automatically from DHCP. The expected result gathered indicate that DHCP configuration was done correctly. Figure B.1 illustrates the results from the machine that used network Trusted_Zone_1 followed by Figure B.2 which presents the result from the machine which utilized network Trusted_Zone_2 while Figure B.3 shows the result from the web server that adopted network DMZ_Zone.

```
IPv4 Address. . . . . . . . . . . : 192.168.2.10(Preferred)
Subnet Mask . . . . . . . . . . . : 255.255.255.0
Lease Obtained. . . . . . . . . . : Saturday, 17 August, 2019 3:39:55 PM
Lease Expires . . . . . . . . . . : Sunday, 18 August, 2019 3:40:02 PM
Default Gateway . . . . . . . . . : 192.168.2.1
DHCP Server . . . . . . . . . . . : 192.168.2.1
```

Figure B.1 DHCP Test from Network Trusted_Zone_1

```
IPv4 Address. . . . . . . . . . . : 192.168.3.5(Preferred)
Subnet Mask . . . . . . . . . . . : 255.255.255.0
Lease Obtained. . . . . . . . . . : Saturday, 17 August, 2019 2:32:15 PM
Lease Expires . . . . . . . . . . : Sunday, 18 August, 2019 2:32:15 PM
Default Gateway . . . . . . . . . : 192.168.3.1
DHCP Server . . . . . . . . . . . : 192.168.3.1
```

Figure B.2 DHCP Test from Network Trusted_Zone_2

```
IPv4 Address. . . . . . . . . . . : 192.168.4.2(Preferred)
Subnet Mask . . . . . . . . . . . : 255.255.255.0
Lease Obtained. . . . . . . . . . : Saturday, August 17, 2019 2:54:08 PM
Lease Expires . . . . . . . . . . : Sunday, August 18, 2019 2:54:09 PM
Default Gateway . . . . . . . . . : 192.168.4.1
DHCP Server . . . . . . . . . . . : 192.168.4.1
```

Figure B.3 DHCP Test from Network DMZ_Zone

B.9 Connectivity Test - Ping

The connectivity test from client to a web server was performed by using ping command which used the ICMP protocol to operate. The ping command is highly useful to test whether the configuration that has been done at the firewall and client is correct such as security policy and routing table. The connectivity test will not succeed if the configuration was done incorrectly. The results of this connectivity test indicate that the client machine had no issues to establish connection against a web server. The command to evaluate the connectivity is ping lab.com.my. Figure B.4 shows results of the ping from machine that was assigned to adopted Trusted_Zone_1 followed by results obtained from the ping command from machine utilize Trusted_zone_2 in Figure B.5.

```
   IPv4 Address. . . . . . . . . . . : 192.168.2.10
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.2.1

C:\Users\admin>ping lab.com.my

Pinging lab.com.my [192.168.4.2] with 32 bytes of data:
Reply from 192.168.4.2: bytes=32 time=1ms TTL=127
Reply from 192.168.4.2: bytes=32 time=1ms TTL=127
Reply from 192.168.4.2: bytes=32 time=1ms TTL=127
Reply from 192.168.4.2: bytes=32 time=1ms TTL=127
```

Figure B.4 Connection test from network Trusted_Zone_1

```
   IPv4 Address. . . . . . . . . . . : 192.168.3.5
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.3.1

C:\Users\admin>ping lab.com.my

Pinging lab.com.my [192.168.4.2] with 32 bytes of data:
Reply from 192.168.4.2: bytes=32 time=2ms TTL=127
Reply from 192.168.4.2: bytes=32 time=2ms TTL=127
Reply from 192.168.4.2: bytes=32 time=1ms TTL=127
Reply from 192.168.4.2: bytes=32 time=1ms TTL=127
```

Figure B.5 Connection Test from Network Trusted_Zone_2

B.10 Connectivity Test – Telnet

A web server utilizes specific port to surf client request such as HTTP. The connectivity test in this section was conducted to verify the port number configured at the web server and to check whether the firewall is open and ready to accept requests from clients. To perform this test, telnet command was employed. The telnet was issued from two machines which are located at different network segments. Results from telnet provide an output to determine whether the HTTP port is closed at the firewall or if the web server is configured incorrectly to accept GET requests which used port 80 to operate. Results from this test show that all machines from a different subnet were successfully connected to a web server by using port 80 and produce same outcomes. The telnet command was executed by using the telnet lab.com.my 80 formats. Figure B.6 indicates the telnet results.

```
HTTP/1.1 400 Bad Request
Content-Type: text/html; charset=us-ascii
Server: Microsoft-HTTPAPI/2.0
Date: Sat, 17 Aug 2019 07:51:58 GMT
Connection: close
Content-Length: 326

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN""http://www.w3.org/TR
<HTML><HEAD><TITLE>Bad Request</TITLE>
<META HTTP-EQUIV="Content-Type" Content="text/html; charset=us-ascii">
<BODY><h2>Bad Request - Invalid Verb</h2>
<hr><p>HTTP Error 400. The request verb is invalid.</p>
</BODY></HTML>
```

Figure B.6 Telnet Result from Network Trusted_Zone_1 and Trusted_Zone_2

B.11 Connectivity Test - Domain Name System (DNS)

A client will use URL to browse contents of a web server and the operation requires DNS to translate the URL into an IP address. As explained in Section 4.4.2, this research employs the web server which acted as a DNS. To ensure that the configuration was correct, a command nslookup lab.com.my is required to be utilized to verify the translation process. The output indicates that the DNS worked as expected and the client was able to browse online contents through URL. Figure B.7 presents the result from the machine utilized network Trusted_Zone_1 followed by Figure B.8 illustrates the result from the machine adopted network Trusted_Zone_2.

```
   IPv4 Address. . . . . . . . . . . : 192.168.2.10
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.2.1

C:\Users\admin>nslookup lab.com.my
Server:  lab.com.my
Address:  192.168.4.2

Name:    lab.com.my
Address:  192.168.4.2
```

Figure B.7 Nslookup Result from Network Trusted_Zone_1

```
   IPv4 Address. . . . . . . . . . . : 192.168.3.5
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.3.1

C:\Users\admin>nslookup lab.com.my
Server:  lab.com.my
Address:  192.168.4.2

Name:    lab.com.my
Address:  192.168.4.2
```

Figure B.8 Nslookup Result from Network Trusted_Zone_2

APPENDIX C INVESTIGATION

C.1 Research Dataset

GET headers investigation conducted by using eight self-generated datasets created from actual attack scripts. The usage of self-generate dataset due to dataset for HTTP DDoS is not publicly available as previously highlighted in Chapter 2 (Section 2.13.1). Chapter 2 (Section 2.2) explained that the scale of DDoS attack can be executed in the internal and external network. Hence to acquire the attack patterns executed from both networks self-generate dataset in this research were segregated into two categories: internal and external dataset. Both types of datasets help to disclose the attack pattern produced by HTTP DDoS. Wang et al. (2015) explained that in order to launch an attack, a cyber intruder can reside in a private network, public network or both.

The first type of dataset was generated from a local network and requires several hardware and software to be used to form the architecture. The second type of dataset was executed using the Internet and a web server which was located at the hosting provider. Attack execution for internal network utilizes address http://lab.com.my and use address http://42.1.63.189 for attack launch from outside. The collected datasets were labeled as DTS1, DTS2, DTS3, DTS4, DTS5, DTS6, DTS7 and DTS8 which consist of dataset for HTTP DDoS attack which occurred at both local and external networks.

The self-generate dataset require actual attack scripts to execute real attack. The attack scripts utilize in this research is available on Internet and publicly known as HOIC, Golden Eye, Chihulk, BlackHorizon, Wreckuests, Hibernet and UFONet. All the attack scripts used in this research were labeled as A1.py, A2.py, A3.py, A4.py, A5.py, A6.py, A7.py and A8.py. The usage of the attack script have been mentioned and employed in many previous studies (Dhanapal and Nithyanandam, 2019; Bravo and Mauricio, 2018; Idhammad et al., 2018; Jazi et al., 2017; Rahman et al., 2017; Johnson Singh et al., 2016; Sree and Bhanu, 2016; Shiaeles and Papadaki, 2014).

The HTTP DDoS has generated plenty of GET requests. Due to this, the attack duration to generate the dataset at the local network was set to ten minutes for attack scripts A1.py, A2.py and A3.py. Any execution longer than ten minutes will result in plenty of logs and filtering of GET headers components during an investigation process would require more time. Attack scripts labelled as A4.py, A5.py, A6.py, A7.py and A8.py that generated external dataset required a shorter duration and were set to five minutes. On the other hand, attack scripts A6.py, A7.py and A8.py required an Internet connection and a public proxy as mediator to generate enormous HTTP traffics. Due to this, a minimal duration is more appropriate to ensure that the source IP address is not blocked by the hosting provider as an actual attack script was utilized to attack the web server. Launching the attack for more than five minutes increases the possibility of the IP address to be blacklisted or the connection to be blocked which would jeopardize the investigation process. Thus, to avoid any circumstance which could possibly lead to inaccurate results and failure to meet the objective of this research, a five-minute duration was seen to be the most appropriate to be used to self-generate the dataset.

The consideration to utilize eight attack scripts in this research was due to several factors. Firstly, the use of the minimal attack script to generate dataset will make certain attack patterns to be overlooked. In contrast, using multiple attack scripts to generate dataset will require more time to be analyzed. Hence, usage of eight attack scripts was adequate to investigate the attack patterns. Besides that, a critical analysis presented in Chapter 2 (Section 2.13.4) highlighted that the use of more attack scripts is recommended to acquire more attack patterns. The self-generate dataset included attack patterns from public proxy as well as internal and external networks which are not available to be obtained from outside as previously highlighted. Table C.1 presents the details regarding the property of the self-generate dataset.

Table C.1 Table Self Generate Dataset Property

No.	Script Name	Attack Scale	Attack Pattern	Attack Duration	Target URL	Dataset Name
1.	A1.py	Internal Attack	High rate direct attack	10 Minutes	http://lab.com.my	DTS1
2.	A2.py					DTS2
3.	A3.py					DTS3
4.	A4.py	External Attack	High rate direct attack	5 Minutes	http://42.1.63.189	DTS4
5.	A5.py					DTS5
6.	A6.py	External Attack	High rate through proxy	5 Minutes	http://42.1.63.189	DTS6
7.	A7.py					DTS7
8.	A8.py					DTS8

C.2 Genuine GET Headers without Proxy

This section explicates the genuine GET headers to observe the pattern and consistency during the process of GET requests. This section also provides extension elaboration done in Chapter 2 (Section 2.6) to illustrate the authentic GET requests by using the latest web browsers to observe the current headers utilized along with its structure. Besides that, request headers produced in this section will be a benchmark to be compared with headers generated by HTTP DDoS. Genuine headers are created from the latest web browsers which are commonly used such as Internet Explorer, Google Chrome and Mozilla Firefox to gain the patterns when a client utilizes the browser to access a content of web server. The headers are captured through normal browse of the test page located at the hosting provider by using this address http://42.1.63.189. Results indicate that all web browsers deliver equal headers consistently without missing headers for subsequent GET requests. Besides that, this

research found that the original headers provide different character sizes for connection header like keep-alive was generated from Google Chrome and Mozilla Firefox while Keep-Alive was created by Internet Explorer. Table C.2 illustrates the header generated by different web browsers.

Table C.2 GET Headers from Different Browsers

Internet Explorer Version 11.239	Google Chrome Version 75.0	Mozilla Firefox Version 68.0
1) Accept 2) Accept-Language 3) User-Agent 4) Accept-Encoding 5) Host 6) Connection	1) Host 2) Connection 3) User-Agent 4) Accept 5) Accept-Encoding 6) Accept-Language	1) Host 2) User-Agent 3) Accept 4) Accept-Language 5) Accept-Encoding 6) Connection

C.3 Genuine GET Headers with Proxy

As explained in Chapter 2 (Section 2.6), prior studies only stated one proxy header in the GET requests known as X-Forwarded-For. Therefore, this section will provide supplementary evidence of another proxy header in used in GET requests. Outcome at this section will act as a guideline when conducting dataset investigation for HTTP DDoS launch through external proxy and will contribute to the formation of detection HTTP DDoS launch via proxy. A GET requests employs proxy to establish HTTP connection against a web server which will assign additional headers named as X-Forward and Via. The use of these headers is to inform the connection that utilizes proxy to browse the contents of a web server.

A GET requests generated through proxy must be able to establish a connection to a web server. Due to this, in order to prevent connection failure during transaction of GET requests which involves three parties such as client, proxy and web server, a list of healthy public proxies is required in this section to ensure that the proxy is capable of establishing a connection to a web server. The healthy proxies help to signal that the proxies are still alive and ready to serve a connection received from a client. The location of external proxies is scattered. Thus, the collection of public proxies

requires software named as proxy switcher. This software provides a list of real time available proxies which are active around the world. Figure C.1 presents the list of available public proxies.

Server	State	Response	Country	Uptime
91.237.161.134:8080	(Alive-SSL)	3285ms	POLAND	76%
198.50.168.210:3128	(Alive-SSL)	976ms	CANADA	75%
133.242.231.184:60088	(Alive-SSL)	2000ms	JAPAN	83%
139.99.40.8:3128	(Alive-SSL)	3488ms	SINGAPORE	76%
103.89.56.54:8080	(Alive-SSL)	6582ms	INDIA	95%
79.134.10.16:8080	(Alive-SSL)	6613ms	RUSSIAN FEDERATION	91%
186.65.82.76:7777	(Alive-SSL)	4972ms	ARGENTINA	82%
103.47.94.97:8080	(Alive-SSL)	3218ms	INDIA	100%
91.237.161.19:8080	(Alive-SSL)	3851ms	POLAND	80%
176.193.139.143:53281	(Alive-SSL)	3422ms	RUSSIAN FEDERATION	100%

Figure C.1 List of Public Proxies

Initial findings against public proxies that establish authentic connection against a web server show that all necessary GET headers are provided. However, further investigation showed that not all proxies utilize standard proxy headers such X-Forwarded-For and Via when establishing a connection to a web server. This circumstance introduces some complications in determination of proxy as it is used as a mediator between a client and a web server. External proxies that append headers X-Forward-For and Via in GET requests provide a clear sign that a client utilizes proxy to obtain contents of a web server. Apart from that, the situation becomes more complex when a proxy is used to form HTTP DDoS and when there is no sign in GET headers that indicates a request employed the proxy. To support this explanation, a genuine GET requests was made via public proxy IP address 110.232.86.52 and the results show that no proxy was present for X-Forwarded-For and VIA. Figure C.2 shows the result for proxy with no X-Forwarded-For and Via.

```
Internet Protocol Version 4, Src: 110.232.86.52, Dst: 42.1.63.189
Transmission Control Protocol, Src Port: 44406, Dst Port: 80, Seq: 1,
Hypertext Transfer Protocol
  GET /favicon.ico HTTP/1.1\r\n
  Host: 42.1.63.189\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/
  Accept: image/webp,image/apng,image/*,*/*;q=0.8\r\n
  Referer: http://42.1.63.189/\r\n
  Accept-Encoding: gzip, deflate\r\n
  Accept-Language: en-US,en;q=0.9\r\n
```

Figure C.2 Proxy With No X-Forwarded-For and Via

Findings in this section also indicate that there were other proxy headers that were utilized in the GET requests known as X-Proxy-ID and X-Real-IP. The existence of these headers along with X-Forwarded-For and Via provide a strong evidence that clients utilize proxy to generate GET requests against a web server. This findings support the explanation made by (Petersson and Nilsson, 2014) whereby they lament that the proxy headers is updated by the proxy provider and some of it retained the existing name which causes the headers name to be not up to date. Figure C.3 presents the results for proxy with X-Proxy-ID, X-Forwarded-For and Via.

```
Internet Protocol Version 4, Src: 189.80.50.186, Dst: 42.1.63.189
Transmission Control Protocol, Src Port: 46412, Dst Port: 80, Seq: 1, Ack:
Hypertext Transfer Protocol
  GET /images/search.gif HTTP/1.1\r\n
  Host: 42.1.63.189\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
  Accept: image/webp,image/apng,image/*,*/*;q=0.8\r\n
  Referer: http://42.1.63.189/\r\n
  Accept-Encoding: gzip, deflate\r\n
  Accept-Language: en-US,en;q=0.9\r\n
  X-Proxy-ID: 384404548\r\n
  X-Forwarded-For: 161.139.153.30\r\n
  Via: 1.1 189.80.50.186 (Mikrotik HttpProxy)\r\n
```

Figure C.3 Proxy with X-Proxy-ID, X-Forwarded-For and Via

C.4 Investigation of GET Headers - DTS1

Dataset DTS1 contains 680 GET requests that was constituted through HTTP DDoS to target a web server. DTS1 was generated by utilizing an attack script labelled as A1.py and utilize this command "python A1.py -t 10000 -c 20 http://lab.com.my" to execute the attack script. The GET headers generated by this attack script presented in Figure C.4.

	Request URI	Referer
Windows NT 6.1; en-US; rv:1.9.1.1) …	/?VOXIODC=LUAZQBI	http://www.yandex.com/JMNODVTUN
Windows NT 6.1; en-US; rv:1.9.1.1) …	/?VOXIODC=LUAZQBI	http://www.yandex.com/JMNODVTUN
Windows NT 6.1; en-US; rv:1.9.1.1) …	/?VOXIODC=LUAZQBI	http://www.yandex.com/JMNODVTUN
Windows NT 6.1; en-US; rv:1.9.1.1) …	/?VOXIODC=LUAZQBI	http://www.yandex.com/JMNODVTUN
Windows NT 6.1; en-US; rv:1.9.1.1) …	/?VOXIODC=LUAZQBI	http://www.yandex.com/JMNODVTUN
Windows NT 6.1; en-US; rv:1.9.1.1) …	/?VOXIODC=LUAZQBI	http://www.yandex.com/JMNODVTUN
MSIE 8.0; Windows NT 5.1; Trident/4…	/?VADTUYSBED=MC…	http://www.usatoday.com/search/results?
MSIE 8.0; Windows NT 6.1; WOW64; Tr…	/?UUC=WDVQDCW	http://www.google.com/?q=BEZXMCBHG
MSIE 8.0; Windows NT 6.1; WOW64; Tr…	/?UUC=WDVQDCW	http://www.google.com/?q=BEZXMCBHG
MSIE 8.0; Windows NT 6.1; WOW64; Tr…	/?UUC=WDVQDCW	http://www.google.com/?q=BEZXMCBHG
Windows NT 6.1; en; rv:1.9.1.3) Gec…	/?UKYNQC=CBJZAH…	http://www.bing.com/DWEKV
Windows NT 6.1; en; rv:1.9.1.3) Gec…	/?UEMGF=RLVKDLWJ	http://www.baidu.com/XJQDKLS
MSIE 8.0; Windows NT 6.1; WOW64; Tr…	/?UDWTNTMDOO=ADD	http://www.baidu.com/YXEOCD
MSIE 8.0; Windows NT 5.1; Trident/4…	/?UBR=VYFQR	http://www.google.com/?q=HTCJQ
MSIE 6.1; Windows XP)	/?TWQFS=YYMBE	http://www.bing.com/VBRNGEKMJH
MSIE 6.1; Windows XP)	/?TWQFS=YYMBE	http://www.bing.com/VBRNGEKMJH

Figure C.4 False GET Headers in DTS1

Investigation against DTS1 found that a random user agent leads to the complexity to recognize a genuine GET requests. A value that is attached to a user agent is not able to determine whether the authenticity of GET requests is coming from valid or bogus resources. Further investigation reveals that HTTP DDoS generated a massive same query randomly against the web server to process. A combination of capital letters is used to generate the query which does not bring any meaning and difficult to be understood by a human.

DTS1 also contains irrelevant HTTP referrer. The referrer in GET requests indicates a source of location where a web server is accessed. DDoS attacks at the application layer manipulate the value of HTTP referrer as it contains a valid URL to form bogus GET requests to make the request looks genuine. The URL existed in DTS1 comes from irrelevant search engines and another website that was assigned to be the value of HTTP referrer that were sending in a rotation mode to mimic a human access pattern to prevent detection. Besides that, the connection is marked as close and the web server still receives continues GET requests.

C.5 Investigation of GET Headers - DTS2

DTS2 contains 88769 lines of GET requests against a web server. DTS2 had similarities with DTS1 whereby a fake value of the user agent was employed to generate an invalid GET requests via HTTP DDoS. DTS2 was generated by using an attack script labelled as A2.py and employed this command "python A2.py http://lab.com.my" to operate. The forged user agent produces by this attack script presented in Figure C.5.

```
User-Agent
Mozilla/5.0 (compatible; bingbot/2.0; +http://www.bing.com/bingbot.htm)
Mozilla/5.0 (Windows; U; Windows NT 6.1; ru; rv:1.9.1.3) Gecko/20090824
Opera/8.51 (Windows NT 5.1; U; en)
Mozilla/5.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0; Acoo Bro
Mozilla/5.0 (compatible; Yahoo! Slurp; http://help.yahoo.com/help/us/yse
Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en-US) AppleWebKit/125.4 (KHTML
Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)
Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.1) Gecko/200
Mozilla/5.0 (iPad; U; CPU OS 3_2 like Mac OS X; en-us) AppleW
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.
Mozilla/5.0 (SymbianOS/9.2; U; Series60/3.1 NokiaN95_8GB/31.0
Mozilla/5.0 (PLAYSTATION 3; 2.00)
```

Figure C.5 False User Agents in DTS2

As indicated in Figure C.5 an investigation against DTS2 found that an additional user agent was used to produce a variety of forged user agents which look authentic comes from mobile devices, web crawler, PlayStation and open source operating. The diversity of a used user agent shows that a web server is accessed from different platforms. This investigation also managed to reveal that the use of a web crawler user agent named as bingbot from MSN and Googlebot from Google shows that a website has been indexed by the search engine which makes the GET requests to look genuine. The user agent continuously rotates to mimic an authentic GET requests.

Further investigations reveal that DTS2 contained an invalid GET requests generated by using numbers separated by backslash and query were continuously and repeatedly sent. A thorough inspection against DTS2 found that another GET headers was being manipulated known as the HTTP referrer. The value of this header contains

a false URL and utilizes HTTP and HTTPS protocol to make a request to appear originated from valid resources. DTS2 also contains URL that is binded with search query which makes the URL to look as genuine. The false request queries show in Figure C.6 followed by Figure C.7 indicates the bogus HTTP referrer with query.

```
Request URI Query
\357\277\275\357\277\275\357\277\275{\177\357\277\275\357\277\275\3
\357\277\275\357\277\275{\357\277\275\177=\357\277\275\357\277\275y
{}\357\277\275\357\277\275\357\277\275\357\277\275\357\277\275=\357
y\357\277\275{\357\277\275\357\277\275\357\277\275\357\277\275=\357
\357\277\275|}\357\277\275\357\277\275=\357\277\275\357\277\275\357
\357\277\275\357\277\275|\357\277\275\357\277\275\357\277\275=\357
\357\277\275\357\277\275\357\277\275\357\277\275\357\277\275\357\2
\357\277\275\357\277\275\357\277\275=\357\277\275\357\277\275\357\
```

Figure C.6 False Request Query DTS2

```
Referer
https://www.yandex.com/yandsearch?text=~|\206\213\204\222\237
https://www.yandex.com/yandsearch?text=~|\206\213\204\222\237
https://www.yandex.com/yandsearch?text=~\230\232\231\200x}
https://www.yandex.com/yandsearch?text=~\230\232\231\200x}
https://www.pinterest.com/search/?q=z\217yy\236\213\203\230
https://www.pinterest.com/search/?q=z\217yy\236\213\203\230
https://www.pinterest.com/search/?q=z\217\234\205{\222\220\226
https://www.pinterest.com/search/?q=z\216\216\213\212\202\204\
https://www.npmjs.com/search?q=\224\216}\231\234\232\240\204\240
https://www.npmjs.com/search?q=\224\216}\231\234\232\240\204\240
https://www.npmjs.com/search?q=\224\213\234\222\210\231\217\212
```

Figure C.7 HTTP Referrer With Request Query DTS2

C.6 Investigation of GET Headers - DTS3

DTS3 contains 97,332 GET requests. This dataset contains minimal headers for each GET requests. DTS4 contains different patterns of GET headers as earlier datasets contain false headers which look authentic and the values randomly change while this dataset only contains basic headers and the header is not rotated. A comparison is made between valid and invalid GET headers to acquire information about which GET headers is missing during the transaction of GET requests. The

comparison successfully exposes that genuine GET requests will deliver complete GET headers when a request against a web server is made such as user-agent, accept-language, accept-encoding, connection and referrer are completely presented. Table C.3 indicates the comparison.

Table C.3 Complete and Incomplete Request Headers

Complete GET Headers (Genuine)	Incomplete GET Headers (HTTP DDoS)
Accept: text/html, application/xhtml+xml, image/jxr, */* Accept-Encoding: gzip, deflate Accept-Language: en-US Connection: Keep-Alive Host: test.local.my Referer: http://test.local.my/ User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trid...	Hypertext Transfer Protocol GET / HTTP/1.0\r\n Accept: */*\r\n Accept-Language: en\r\n Host: lab.com.my\r\n \r\n [Full request URI: http://lab.com.my/] [HTTP request 1/1] [Response in frame: 1067951]

C.7 Investigation of GET Headers - DTS4

DTS4 comprised of 21257 GET requests and this dataset was generated from an attack script labelled as A4.py. The attack script requires the command "python A4.py http://42.1.63.189" to operate. An initial investigation found that this dataset contains the fake value of GET headers for user-agent, request query and irrelevant value of HTTP referrer. Detailed investigation against DTS4 found that there is no consistency with regard to presentation of GET headers as some of the headers were missing such as the HTTP referrer. Although the referrer existed for certain GET requests, the value is irrelevant as the value of HTTP referrer was supposed to come from a related site. Further investigation found that DTS4 contains a much longer request query. The query contains a combination of a small letter, capital letter, number and symbol. Figure C.8 illustrate the forged GET headers contain in DTS4.

User-Agent	Request URI	Referer
Mozilla/5.0 (compatible; MSIE 10.0; Linux…	/?mdebDjl=3sVfhMxqbW17JCs&…	http://www.google.com/NpVcl
Mozilla/5.0 (Windows; U; MSIE 7.0b; Windo…	/?pe3S=EwWo4TeyC3&4E7AGV2=…	http://www.baidu.com/GMitC?
Mozilla/5.0 (Macintosh; Intel Mac OS X 10…	/?Mkq=e4cEF5DYDKRnGsU0x4&O…	http://www.baidu.com/XVvdcl
Mozilla/5.0 (Windows NT.6.2; Win64; x64) …	/?x1WX3=D4oIS&dTauJ=EFn&6S…	http://www.google.com/C45Eb
Mozilla/5.0 (Macintosh; Intel Mac OS X 11…	/?FhOjH=vp1JXgqUQODjdpU&v3…	http://42.1.63.189/58LXhnxL
Mozilla/5.0 (Macintosh; Intel Mac OS X 11…	/?p26uYWiI0P=TkGS12dnj42sG…	http://www.bing.com/7HNWcR?
Mozilla/5.0 (Windows; U; MSIE 7.0b; Macin…	/?r14lDO=hSctiM7jpk5oVADl	http://www.bing.com/a4JE0Ad
Mozilla/5.0 (Macintosh; Intel Mac OS X 10…	/?0emGAo=oXmEjb6C&DyT=j7Rj…	http://www.baidu.com/nKaBoG
Mozilla/5.0 (Macintosh; Intel Mac OS X 10…	/?N5f4hjYpWa=PeW4YiFOatWdq…	http://www.baidu.com/wXko7w
Mozilla/5.0 (Windows; U; MSIE 7.0; Linux …	/?7xk=HEbqX5x5rG	
Mozilla/5.0 (Macintosh; Intel Mac OS X 11…	/?xrEPs3O6T=D4jbg7OtYDbuGh…	
Mozilla/5.0 (Linux i386; X11) AppleWebKit…	/?k7xHcmh=RBdMoIFpi	http://www.bing.com/PGJ6Jnx
Mozilla/5.0 (Windows NT.6.2; Win64; x64) …	/?7yGpG=kwsBgn0p4T04&LCfwf…	http://www.yandex.com/AhuWc
Mozilla/5.0 (Linux x86_64; X11) Gecko/200…	/?pEqEfQ7tE=DvmtdNaYjmFjLs…	http://www.bing.com/ij1wYOC
Mozilla/5.0 (Windows; U; MSIE 8.0; Macint…	/?iTUryDq=8HEvAcdTXYfOc&7q…	http://www.yandex.com/BTsp5
Mozilla/5.0 (compatible; MSIE 10.0; Linux…	/?eWk=4xwxnsagBRAXUEbwtp&K…	

Figure C.8 False GET Headers in DTS4

C.8 Investigation of GET Headers - DTS5

DTS5 comprised of 23,974 GET requests received by a web server. DTS5 was generated by using an attack script labelled as A5.py and it utilize an IP address instead of URL as the URL of the target web server is not registered at DNS record hosting provider. The attack script operates by using this command "python A5.py http://42.1.63.189". Initial inquiries against DTS5 found that all necessary headers were supplied completely. However, the dataset contains false request query created from small letter, capital letter, number and symbol. Apart from that, GET headers known as accept-language is missing and the value of HTTP referrer is valid. Despite the HTTP value presented in a correct format, it comes from irrelevant resource. Further inquiries against this dataset found that the dataset also contains irrelevant GET headers as the header named Keep-Alive which was supposed to be employed in HTTP response was incorrectly placed at GET requests. Figure C.9 illustrates the GET headers followed by Table C.4 which shows the existence of Keep-Alive at HTTP response.

```
Hypertext Transfer Protocol
> GET /?b4IOVw8dWJ=DR18m5xoIqJlYSkekWOD&iQVJB=VolfYf2gQQR3Pl
  Host: 42.1.63.189\r\n
  Accept-Encoding: ''\r\n
  Keep-Alive: 272\r\n
  User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 11_7_3)
  Connection: keep-alive\r\n
> Cookie: hkXcBWKSJ=gJpmfAWE&F1n=1Afo&cAda=1RTMm&LSXw=T8YPYy
  Cache-Control: max-age=0\r\n
  Referer: https://www.facebook.com/yMpKooReFD\r\n
  Content-Type: application/x-url-encoded\r\n
```

Figure C.9 DTS5 GET Headers

Table C.4 Valid and Invalid Keep Alive Values

Valid Keep Alive in HTTP Response	Invalid Keep Alive in GET Headers
Cache-Control: max-age=31449600, no-transforr Connection: Keep-Alive Date: Tue, 19 Jun 2018 04:47:38 GMT Expires: Tue, 18 Jun 2019 04:47:38 GMT Keep-Alive: timeout=5, max=99 Server: Apache Vary: Accept-Encoding	Accept-Encoding: '', gzip\r\n Connection: keep-alive\r\n Keep-Alive: 45\r\n Accept-Encoding: '', deflate\r\n Connection: keep-alive\r\n Keep-Alive: 544\r\n

C.9 Investigation of GET Headers - DTS6 (Spoof IP)

DTS6 contains 9397 forged GET requests that was sent through spoof IP address to launch a HTTP DDoS attack. The command requires to execute the attack script was “python3 A6.py -v http://42.1.63.189”. Initial findings against DTS6 disclosed that it contains a requests query that is understood by humans although the query was irrelevant for the web server to process. The GET headers for IP address 62.210.15.199 and 92.222.74.221 appears to be genuine. However, the data included in the HTTP referrer does not come from a relevant URL. As explained in Chapter 2 (Section 2.6) the existence of HTTP referrer in GET requests shows the last page visited by user when browsing the current web page. The existence of HTTP referrer evidently shows that the GET requests is not valid as the actual URL for the web server utilized in this research is not publicly known to be accessible from other resources.

DTS6 comprised of spoof IP addresses. The source of IP address verification was made to determine the owner of the spoof IP address. Firstly, the IP address 62.210.15.199 was selected and the outcome indicates that the IP address belongs to a web server and utilized port 22, 80, 443 and 3306 to operate. Secondly, the IP address 92.222.74.221 was opted, and result shows that the IP address is owned by the web server which adopted port 22 and 80 to be operational. Thirdly, the IP address 103.234.254.10 was utilized, and it was found that it belongs to the router and utilized port number 53 and 1723. Outcome in this section proof that the HTTP DDoS attack can be executed from spoof IP address and the source of attack can be originated from various machines. Figure C.10, Figure C.11 and Figure C.12 illustrates the IP address verification.

```
Completed Parallel DNS resolution of 1 host. at 10:46, 0.35s elapsed
Initiating SYN Stealth Scan at 10:46
Scanning 62-210-15-199.rev.poneytelecom.eu (62.210.15.199) [65535 ports]
Discovered open port 22/tcp on 62.210.15.199
Discovered open port 80/tcp on 62.210.15.199
Discovered open port 443/tcp on 62.210.15.199
Discovered open port 3306/tcp on 62.210.15.199
```

Figure C.10 IP Address Source Verification for 62.210.15.199

```
Completed Parallel DNS resolution of 1 host. at 13:44, 0.70s elapsed
Initiating SYN Stealth Scan at 13:44
Scanning 221.ip-92-222-74.eu (92.222.74.221) [65535 ports]
Discovered open port 80/tcp on 92.222.74.221
Discovered open port 22/tcp on 92.222.74.221
```

Figure C.11 IP Address Source Verification for 92.222.74.221

```
Completed Parallel DNS resolution of 1 host. at 13:24, 0.53s elapsed
Initiating SYN Stealth Scan at 13:24
Scanning 103.234.254.10 [65535 ports]
Discovered open port 53/tcp on 103.234.254.10
Discovered open port 1723/tcp on 103.234.254.10
```

Figure C.12 IP Address Source Verification for 103.234.254.10

C.10 Investigation of GET Headers - DTS7 (Proxy 1)

DTS7 contains 70792 GET requests generated by various public proxies to launch an attack over a web server. Formation of this dataset utilized attack script A7.py and require a list of proxy IP addresses inside the proxy file in order to launch HTTP DDoS through proxy. The command requires by the attack script to operate was "python3 A7.py http://42.1.63.189".

An initial investigation reveals that all necessary headers were supplied in GET requests. However, proxy headers published in GET requests was inconsistent with different proxy IP address. DTS7 contains a variety proxy headers pattern. For instance, some of the proxy employed a single name to acknowledge the receiver of the connection was through proxy by using X-Forwarded-For header. Another form of introduction included in DTS7 to inform the source originates from a proxy by making use of both headers such as X-Forwarded-For and VIA. The last pattern for proxy header is appending X-Proxy-ID with X-Forwarded-For and VIA as identification of the source connection came from a proxy. All these patterns indicate that each proxy has their own approach to display proxy headers to notify whether the proxy is currently used in establishing connection to a web server. Figure C.13 and Figure C.14 presents the first and second pattern of the proxy headers while Figure C.15 displayed the third proxy headers pattern.

```
GET / HTTP/1.1\r\n
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11)
Accept-Language: en-US,en;q=0.5\r\n
Accept-Encoding: gzip, deflate\r\n
Host: 42.1.63.189\r\n
X-Forwarded-For: 197.171.230.7, unknown\r\n
Cache-Control: max-age=259200\r\n
Connection: keep-alive\r\n
```

Figure C.13 X-Forward-For in GET Requests

```
GET / HTTP/1.1\r\n
User-Agent: Opera/9.80 (X11; Linux i686; Ubuntu/14.10)
Accept: text/html,application/xhtml+xml,application/xml
Accept-Language: en-US,en;q=0.5\r\n
Accept-Encoding: gzip, deflate\r\n
Host: 42.1.63.189\r\n
Via: 1.1 amazon-us-proxy01 (squid/3.3.8)\r\n
X-Forwarded-For: 208.246.8.223, 161.139.153.30\r\n
Cache-Control: max-age=259200\r\n
Connection: keep-alive\r\n
```

Figure C.14 X-Forward-For and Via in GET Requests

```
GET / HTTP/1.1\r\n
Host: 42.1.63.189\r\n
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:40.0)
Accept: text/html,application/xhtml+xml,application/xml;
Accept-Language: en-US,en;q=0.5\r\n
Accept-Encoding: gzip, deflate\r\n
X-Forwarded-For: 103.6.113.166, 161.139.153.30\r\n
X-Proxy-ID: 2145478058\r\n
Via: 1.1 103.16.61.134 (Mikrotik HttpProxy)\r\n
```

Figure C.15 X-Forward-For, Via and X-Proxy-ID in GET Requests

Further investigation reveals that some of GET requests were missing. The missing header is named as referrer which is commonly displayed in a GET requests. To confirm that the header was missing due to fake GET requests, the proxy IP address utilized in script A7.py is used to launch a genuine GET requests. This is to ensure that the missing header was not due to the proxy which intentionally hidden the headers. The proxy IP address was 36.66.55.181 and utilized port 8080 for clients to connect. As a result, authentic request against a web server using the proxy supplied all the common headers. Figure C.16 presents the GET headers with absent referrer followed by Figure C.17 which shows the complete headers supplied by the proxy through a valid GET requests.

```
Internet Protocol Version 4, Src: 36.66.55.181, Dst: 42.1.63.189
Transmission Control Protocol, Src Port: 55131, Dst Port: 80, Seq: 1
Hypertext Transfer Protocol
> GET / HTTP/1.1\r\n
  Host: 42.1.63.189\r\n
  User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; AS;
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/
  Accept-Language: en-US,en;q=0.5\r\n
  Accept-Encoding: gzip, deflate\r\n
  X-Forwarded-For: 254.50.33.157, 161.139.153.30\r\n
  X-Proxy-ID: 1991632303\r\n
  Via: 1.1 36.66.55.181 (Mikrotik HttpProxy)\r\n
```

Figure C.16 Missing referrer in GET headers

```
Internet Protocol Version 4, Src: 36.66.55.181, Dst: 42.1.63.189
Transmission Control Protocol, Src Port: 51773, Dst Port: 80, Se
Hypertext Transfer Protocol
▷ GET /style.css HTTP/1.1\r\n
  Host: 42.1.63.189\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWe
  Accept: text/css,*/*;q=0.1\r\n
  Referer: http://42.1.63.189/\r\n
  Accept-Encoding: gzip, deflate\r\n
  Accept-Language: en,en-US;q=0.9\r\n
  X-Proxy-ID: 60846485\r\n
  X-Forwarded-For: 115.164.183.36\r\n
  Via: 1.1 36.66.55.181 (Mikrotik HttpProxy)\r\n
```

Figure C.17 Complete GET headers with referrer

C.11 Investigation of GET Headers - DTS8 (Proxy 2)

DTS8 contains 51 GET requests which utilized a proxy to launch an attack over a web server. The dataset was prepared by using attack script labelled as A8.py which requires command "./ufonet -a http://42.1.63.189 -r 10 -threads 500" to operate. Preliminary investigation of DTS8 found that different names were utilized to provide a sign for a connection to a web server through proxy by using header named as X-Pingback-Forwarded-For. Besides that, there was an absent proxy header named as VIA which is commonly used to display the proxy name used by a client. Detailed investigation revealed that the dataset contains an odd value of user-agent. Figure C.18 presented the user agent contain in DTS8.

User-Agent
LoadImpactPageAnalyzer/1.3.0 (Load Impact; http://loadimpact.com/)
LoadImpactPageAnalyzer/1.3.0 (Load Impact; http://loadimpact.com/)
LoadImpactPageAnalyzer/1.3.0 (Load Impact; http://loadimpact.com/)
LoadImpactPageAnalyzer/1.3.0 (Load Impact; http://loadimpact.com/)
LoadImpactPageAnalyzer/1.3.0 (Load Impact; http://loadimpact.com/)
LoadImpactPageAnalyzer/1.3.0 (Load Impact; http://loadimpact.com/)
LoadImpactPageAnalyzer/1.3.0 (Load Impact; http://loadimpact.com/)
WordPress/4.9.6; http://extensor.no; verifying pingback from 172.21
WordPress/4.9.5; http://vatsim-scandinavia.org; verifying pingback
WordPress/4.7.5; http://www.zoara-fanantenana.org; verifying pingba
WordPress/4.9.6; http://131.220.122.225; verifying pingback from 16
WordPress/4.8.6; http://www.thailin.com.hk; verifying pingback from
WordPress/4.9.5; http://www.ch-orthez.fr; verifying pingback from 1
WordPress/4.9.6; http://www.dall-italia.nl; verifying pingback from

Figure C.18 User Agent in DTS8

Further investigation found that DTS8 also contains query string generated from a number, symbol, small and capital letter. HTTP referrer existed at DTS8 pointed at web server address and appeared inconsistently as the header was missing for certain GET requests. Besides that, referrer that existed at DTS8 was attached together with the query indicate that the query was issued from the previous page and forwarded to another page to be processed. Figure C.19 indicate the forged query and referrer.

Request URI	Referer
/?Favzel0L=ViS8Se	http://42.1.63.189/?Favzel0L=ViS8Se
/?tCYgD3LE=ufWxLz	http://42.1.63.189/?tCYgD3LE=ufWxLz
/?dELLQcEm=PTGEoH	
/?GqHFbMTq=xZe5kY	http://42.1.63.189/?GqHFbMTq=xZe5kY
/?fVZ2XNUo=8pO971	
/?NeZWLB5T=2djhTf	http://42.1.63.189/?NeZWLB5T=2djhTf
/?FE9aHdqZ=isDLW0	http://42.1.63.189/?FE9aHdqZ=isDLW0
/?B8zNpWak=DHEq82	
/?aGPNlvHO=sxvdAq	http://42.1.63.189/?aGPNlvHO=sxvdAq
/?yj0WVTEH=Y9l7CM	http://42.1.63.189/?yj0WVTEH=Y9l7CM
/?wE1wfHkS=OeTe90	http://42.1.63.189/?wE1wfHkS=OeTe90
/?d2PLbuao=i9qSqR	http://42.1.63.189/?d2PLbuao=i9qSqR
/?4n4ZaalS=SIZzek	http://42.1.63.189/?4n4ZaalS=SIZzek

Figure C.19 False Query and Referrer in DTS8

C.12 Attack Category Associated with Real HTTP DDoS

The attack script adopted in this research is well-known and has been utilized by many prior studies as highlighted at Section C.1. However, none of the past studies perform mapping the attack scripts is linked to which attack strategy as explained in Chapter 2 (Section 2.4). Thus, based on the attack patterns executed at Section C.4 until Section C.11, this research suggests that the attack scripts belonged to the category web proxies, constant, server load, main page attack and random attack. Table C.5 provides the details regarding the dataset mapping with the existing attack strategy.

Table C.5 Dataset Mapping with Existing Attack Strategy

No	Attack Strategies	Attack Script Name
1.	Web Proxies	Hibernet.py
		UFONet.py
2.	Spoof	Wreckuests.py
3.	Server Load	Chihulk, HOIC, Golden Eye, BlackHorizon
4.	Main Page Attack	
5.	Random Attack	

C.13 Overview of Dataset Comparison

This section provides a comparison between datasets to obtain similarities and differences between GET headers contained in each dataset which were previously formed by various HTTP DDoS attack scripts executed from the Internet and internal network. This comparison only takes into account the most manipulated GET headers utilized by the attack scripts such as user-agent, query string, referrer, connection, accept-language and proxy headers. Investigation that conducted indicates that headers as mentioned above is highly manipulated as it contains information adopted by users. Although another GET headers existed during the process of GET requests these headers appeared to be the most headers manipulated by an attacker as it sufficient to be looked as legitimate GET requests.

C.14 GET Headers - User-Agent

Dataset DTS1, DTS2, DTS4, DTS5, DTS7 and DTS8 contain a user-agent which delivers information about a client such as the browser type, version, operating system ID, name and platform such as 32 or 64 bits. DTS3 provides minimal headers, and user-agent was not presented in the GET headers. However, assessment against DTS2 indicates that a variety of platforms have been employed to publish information about the source request such as from mobile device, web crawler from Googlebot and video game devices like Play Station. In contrast, DTS8 provides an odd user-agent as the published user-agent which contradicts with an authentic GET requests and is found to be unique compared to other datasets. The user-agent existed at DTS8 contradicted with the genuine user-agent as it contains URL and IP address.

Zhang et al. (2015) illustrated a user-agent from a common web browser such as Internet Explorer, Firefox, Chrome, Safari and Opera and none of these web browsers produce user-agent as seen in DTS8. They further mentioned that malicious user-agent was difficult to be differentiated by network administrators and security analysts.

C.15 GET Headers - Query String

DTS1 contains a query string constituting a capital letter while DTS2 consists of query generated from a number with a few combinations with symbols. Query string is non-existent in DTS4 and DTS7. DTS4 and DTS5 contain a much longer query string and was generated from a combination of small letter, capital letter, number and symbol. The existence of symbols in HTTP header is because HTTP employs either a common syntax separated by white space or delimits character. Besides that, there is no predefined limit for GET requests to process headers and its values. Furthermore, an appropriate length for GET requests is difficult to be defined (Fielding and Reschke, 2014b).

DTS4 and DTS5 contain query string that is much closer to the query generated by a web server to publish items such as a picture. An authentic query string from human will write a string that relates to the contents of a web server to find the answer. DTS6 consists of query understood by humans even it is irrelevant to a web server to process compared to queries existed at DTS1, DTS2, DTS4 and DTS5 in which the query is not readable. DTS8 contains query that is identical to DTS4 and DTS5. However, the query was not longer than those found in DTS3 and DTS5. A web server utilized in this research contains static HTML page which the query processing does not permit. Thus, receiving continuous request query provides evidence that the GET requests is malicious. In terms of query repetition, only DTS1 provides the same query continuously with certain periods of time, the query changed to another structure. Other queries were generated randomly with no replication for subsequent GET requests.

C.16 GET Headers - Referrer

All datasets utilized HTTP referrer in repeatable mode and followed the correct format equal to a genuine GET requests. According to Reid (2004), HTTP referrer refer to previous web page address accessed by a user. Due to this, any web site address is allowed to be a referrer in GET requests as there is no restriction or inspection done at this section to determine whether the source address is appropriate or inappropriate to be attached at the GET headers. This comparison found that all datasets (DTS1 until DTS8) except for DTS4 employs valid HTTP referrer. However, the source of the web address is unsuitable to be a referrer for the current page. This pattern indicates that the value contained in HTTP referrer has been manipulated.

DTS1, DTS2, DTS4, DTS6, DTS7 and DTS8 utilize referrer which comes from HTTP protocol. However, only DTS5 consists of referrer from HTTPS protocol which used Facebook address and presence of this referrer indicates that the current page is accessible from Facebook. Most of the dataset consist of referrer which come from a search engine. Upon further observation, the referrer appears to be genuine and accessible for normal browsing. An attacker’s strategy is to utilize valid referrer and

the existence of referrer will increase the validity of the source connection. In this circumstance, the relevance of the source connection is a limitation for an attacker even when the referrer appears to be authentic. HTTP referrer commonly comes from sources which are related to contents of a web server such as a faculty website which is supposed to have a link to the University website.

HTTP referrer has the potential to disclose browsing history and user information which might be found in this header (Fielding and Reschke, 2014a). Dolnak (2017) provides solution problem highlighted by Fielding and Reschke (2014a) where the study proposed HTTP referrer header policy. The idea behind the solution was to control information shows by referrer to minimize information leakage. However, the idea does not address problem manipulation of HTTP referrer generate by HTTP DDoS.

C.17 GET Headers - Connection and Accept Language

The comparison conducted found that only two datasets (DTS1 and DTS2) indicate a connection status as close in GET headers compared to other datasets connection status which is shown as Keep-Alive. Wall (2004) shows the content of HTTP request and HTTP response headers and indicates that Keep-Alive is the header of HTTP response and value of the connection header in GET requests. However, existence of Keep-Alive as the header in GET requests in DTS1, DTS2, DTS4, DTS5 and DTS8 illustrate that an attacker accidentally assigns the header in GET requests and indicates that HTTP DDoS generate GET headers automatically with the existence of false position of headers. Besides that, only DTS6 and DTS7 does not utilize Keep-Alive as the header. DTS4 consists of minimal GET headers and the status of connection is unknown.

Accept-language in GET headers indicates the web browser language used by users and the existence of this header in GET requests can be used to identify the geolocation of users (Reid, 2004). This comparison found that five datasets are not present accept-language in GET headers during GET requests transaction such as

DTS1, DTS2, DTS4, DTS5 and DTS6. Two datasets contain the header such as DTS4 and DTS7 while DTS8 shows an inconsistent behavior as the header is missing for certain GET requests. Missing accept-language indicates that the source connection utilizes automated tools to generate GET requests as a client will employ a web browser to establish a connection to a web server to send GET requests (Parziale et al., 2006). Besides that, a genuine GET requests regardless of the browser version will bring the header in the GET requests. Gou et al. (2017) explained that the connection and accept-language are few of the general headers located at GET requests.

C.18 GET Headers - Proxy

The GET requests will display proxy headers utilized by a client to establish a connection to a web server via HTTP protocol. Section A.3 shows the investigation of HTTP DDoS launch through proxy found that when a GET requests employs proxy, typically a proxy will provide a proxy header to acknowledge the request utilized proxy to connect to a web server. DTS7 until DTS8 contains an attack traffic generated from plenty of proxies and indicates that proxy headers are not displayed by certain proxy provider. Despite the usage of proxy to execute HTTP DDoS an adoption of spoof IP address also can be utilized to launch the attack. DTS6 contained a number of spoof IP addresses to launch a HTTP DDoS resulted in difficulties to determine whether the source was originally generated by genuine client or whether it has been misused. The spoof IP address that was attached together with the proxy headers and GET headers not only indicate that the request is authentic, it also shows that the request comes from a proxy.

The dataset labeled as DTS7 contains two IP addresses in X-Forwarded-For header where the second IP address belongs to this research equipment and the first IP are believed to come from another proxy as explained by Petersson and Nilsson (2014) if a GET requests from proxy requires an establishment of another proxy to gain the contents of a web server the IP address was present at X-Forwarded-For. Another proxy header named as X-Pingback-Forwarded-For was only found in DTS8 which provides a sign that a different proxy provider has various approaches to utilize a proxy

name to indicate a client employs outside resources to access web server resources. Output at Section C.10 and Section C.11 supported by Petersson and Nilsson (2014) as the existence of distinct proxy headers and the inconsistent appearances were found in datasets due to some of the proxy providers are not updated the proxy headers and the headers are optional to be displayed in GET the request headers.

Gou et al. (2017) explained that the use of a keyword CONNECT in GET requests which indicates that the client established a connection through a proxy to browse the web server's contents. However, there is no keyword CONNECT that existed in DTS7 and DTS8. Although DTS6 contains spoof IP address that utilize a proxy header the key word is not presented. A possible explanation that can justify the differences observed is that the study may utilize another proxy which makes the keyword to exist at their environment during the execution of proxy assessment. Although different results were obtained in this research, it is believed that the proxy headers is supplied by a proxy provider to acknowledge a connection from a client to a web server is established through a proxy and the use of a proxy header's name is dependent on the provider to introduce themselves as a proxy by using an appropriate name. The explanations provided above are supported by results highlighted in Section C.10 and Section C.11 where dissimilar proxy headers names were found from the datasets.

C.19 Overview of Investigation GET Headers Vulnerabilities

The GET requests provides several components during the process of client to obtain services from a web server. Among the component involved in GET requests are accept-language, accept-encoding, accept-charset, connection, content-type and content-length to acknowledge the server's values carried by a client. The forged request done by HTTP DDoS mimics a genuine GET requests component and its values, which will result in difficulties to differentiate the validity of GET requests. Hence, this section investigates the GET headers shortfall that allows attackers to take advantage in generating bogus GET requests when launching HTTP DDoS attack.

C.20 GET Headers Vulnerabilities - User Agent

User agent contains information regarding client information such as the operating system, browser name and browser version. Attack script labelled as A1.py executed and discussed in Section C.4 reveal that false user agent is made identical to the authentic one. A list of user agents must be presented inside attack script to be sent randomly to a web server which makes a GET requests to contain a variety of users-agent and to indicate that various clients are accessing the web server. To increase the trust, a GET requests is set to genuine as the value of a user agent can add as many as possible and the existence of user agents from mobile devices like smart phone and tablet will lead to an understanding that GET requests was generated from those devices. Code modification at script A1.py has implemented to adjust the structure of value user agent, removed the web browse name, client system information and string known as Mozilla/5.0 which includes special characters and contains operating system ID which already absolute. Windows NT 5.1 refer to Windows XP Operating system. Figure C.20 demonstrates the line of codes to create a fake user agent.

```
def useragent_list():
        global headers_useragents
        headers_useragents = []
        headers_useragents.append('Mozilla/5.0 (Windows NT 5.1; WOW64; rv:50.0) Gecko/20100101 Firefox/50.0')
        headers_useragents.append('Mozilla/5.0 (Windows NT 5.1; WOW64; rv:50.0) Gecko/20100101')
        headers_useragents.append('(Windows NT 5.1; WOW64; rv:50.0) Gecko/20100101 Firefox/50.0')
        headers_useragents.append('Gecko/20100101 Firefox/50.0 (Windows NT 5.1; WOW64; rv:50.0)')
        headers_useragents.append('(Windows NT 5.1; WOW64; rv:50.0) Gecko/20100101 Firefox/50.0 `~!@#$%^&*()_')
        return(headers_useragents)
```

Figure C.20 Code to Generate Forged User Agent

Results gathered in this section indicate a clear evidence that user agent in GET requests possess vulnerabilities, which allows modification against values of user agent and a web server accepts the values without performing any inspection against the assigned values. A randomized technique to rotate the usage of the user agent that constitutes a false GET requests made the GET requests to appear genuine. It is difficult to distinguish between a fake and an authentic user agent if there are no queries performed at this header. Figure C.21 illustrates the output when code was executed while Figure C.22 presents a genuine user agent.

```
GET /?INEX=WHQWSVUANX HTTP/1.1\r\n
Accept-Encoding: identity\r\n
Host: 42.1.63.189\r\n
Keep-Alive: 119\r\n
User-Agent: Mozilla/5.0 (Windows NT 5.1; WOW64; rv:50.0) Gecko/20100101\r\n
GET /?JREBSUK=MPGRCET HTTP/1.1\r\n
Accept-Encoding: identity\r\n
Host: 42.1.63.189\r\n
Keep-Alive: 119\r\n
User-Agent: (Windows NT 5.1; WOW64; rv:50.0) Gecko/20100101 Firefox/50.0\r\n
GET /?BUAU=XDKV HTTP/1.1\r\n
Accept-Encoding: identity\r\n
Host: 42.1.63.189\r\n
Keep-Alive: 117\r\n
User-Agent: Gecko/20100101 Firefox/50.0 (Windows NT 5.1; WOW64; rv:50.0)\r\n
GET /?DTJPDL=JAYCXEDE HTTP/1.1\r\n
Accept-Encoding: identity\r\n
Host: 42.1.63.189\r\n
Keep-Alive: 114\r\n
User-Agent: (Windows NT 5.1; WOW64; rv:50.0) Gecko/20100101 Firefox/50.0 `~!@#$%^&*()_\r\n
```

Figure C.21 Result of Code Fake User Agents

MSIE	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 6.1; en-US; .NET CLR 1.1.22315)
	Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; Trident/6.0)
Firefox	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:29.0) Gecko/20120101 Firefox/29.0
	Mozilla/5.0 (X11; U; Linux i686; de-DE; rv:1.7.6) Gecko/20050306 Firefox/1.0.1
Chrome	Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/37.0.2049.0 Safari/537.36
	Mozilla/5.0 (Linux; Android 4.0.3; GT-I9100 Build/IML74K) AppleWebKit/535.19 (KHTML, like Gecko) Chrome/18.0.1025.133 Mobile Safari/535.19
Safari	Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_7; en-us) AppleWebKit/534.16+ (KHTML, like Gecko) Version/5.0.3 Safari/533.19.4
	Mozilla/5.0 (Windows; U; Windows NT 5.1; it) AppleWebKit/522.13.1 (KHTML, like Gecko) Version/3.0.2 Safari/522.13.1

Figure C.22 Genuine User Agent (Zhang et al., 2015)

C.21 GET Headers Vulnerabilities – Referrer

The referrer header in GET requests indicates the web address of which the contents of the web server are accessed from. HTTP DDoS use this header to make the GET requests look legitimate. An attacker manipulates this header by assigning authentic values to referrer header by using accurate URL format which will result in increasing of trust to assume that a GET requests comes from a valid source. In addition, An attacker manipulates this header by using correct URL format to make it complicated to be distinguished with legitimate request and increase trust a GET requests comes from a valid source.

A forged referrer header requires an address in URL format such as http://abc.com to be presented at the GET requests header. In addition, the use of plenty genuine referrer such as those coming from valid search engines will lead to difficulties in evaluating the originality of the GET requests. A code adjustment against an attack script A1.py has been made to include a false URL name which contains special character, capital letter and URL that is non-existent. The attack script rotates the usage of URL and does not permanently utilize one URL to prevent suspicious activity from being detected. Response received from a web server indicates that a web server accepts all values assigned at the referrer header. Besides that, there was no inspection against the presented referrer whether it is regarded as valid, forged or accessible from a relevant source. Figure C.23 presents the code for fake HTTP referrer while Figure C.24 indicates the outcome of the code.

```
def referer_list():
        global headers_referers
        headers_referers = []

        headers_referers.append('http://www.mcdonalds.com.my/')
        headers_referers.append('http://www.false.url/')
        headers_referers.append('http://www.ipta.edu.my/')
        headers_referers.append('http://www.IPTS.edu.my/')
        headers_referers.append('http://www.sushi.com.jpn/')
        headers_referers.append('http://www.!@#.com/')

return(headers_referers)
```

Figure C.23 Code to Generate Forged HTTP Referrer

1)
```
Hypertext Transfer Protocol
 GET /?XVQMXH=CIDVBT HTTP/1.1\r\n
  Accept-Encoding: identity\r\n
  Host: 42.1.63.189\r\n
  Keep-Alive: 115\r\n
  User-Agent: Opera/9.80 (Windows NT 5.2; U; ru) Presto/2.5.22
  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n
  Connection: close\r\n
  Referer: http://www.IPTS.edu.my/\r\n
```

2)
```
Hypertext Transfer Protocol
 GET /?SEVXCSRX=VTXXEAU HTTP/1.1\r\n
  Accept-Encoding: identity\r\n
  Host: 42.1.63.189\r\n
  Keep-Alive: 116\r\n
  User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows
  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n
  Connection: close\r\n
  Referer: http://www.ipta.edu.my/\r\n
```

3)
```
Hypertext Transfer Protocol
 GET /?VNPUXJY=VNWYOD HTTP/1.1\r\n
  Accept-Encoding: identity\r\n
  Host: 42.1.63.189\r\n
  Keep-Alive: 114\r\n
  User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows
  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n
  Connection: close\r\n
  Referer: http://www.!@#.com/\r\n
```

4)
```
Hypertext Transfer Protocol
 GET /?WAUBDRDJYE=SINCOEEX HTTP/1.1\r\n
  Accept-Encoding: identity\r\n
  Host: 42.1.63.189\r\n
  Keep-Alive: 115\r\n
  User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US)
  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n
  Connection: close\r\n
  Referer: http://www.mcdonalds.com.my/\r\n
```

5)
```
Hypertext Transfer Protocol
 GET /?XUDQN=ASODOOXOJF HTTP/1.1\r\n
  Accept-Encoding: identity\r\n
  Host: 42.1.63.189\r\n
  Keep-Alive: 113\r\n
  User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows
  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n
  Connection: close\r\n
  Referer: http://www.false.url/\r\n
```

6)
```
Hypertext Transfer Protocol
 GET /?XYZ=QBSAE HTTP/1.1\r\n
  Accept-Encoding: identity\r\n
  Host: 42.1.63.189\r\n
  Keep-Alive: 118\r\n
  User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows
  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n
  Connection: close\r\n
  Referer: http://www.sushi.com.jpn/\r\n
```

Figure C.24 False URL in HTTP Referrer

C.22 GET Headers Vulnerabilities - Request Query

The request query is one of the attributes presented during the process of GET requests. This header provides an indicator that clients request for specific information from a web server to be published. This section was executed to scrutinize shortfall for request query in transaction of GET requests. The ASCII character was employed to simulate a random request query. Table C.6 presents the symbol to generate a false query followed by Figure C.25 which presents the code modification for the attack script A1.py.

Table C.6 ACSII Format to Generate String

Type	ASCII	Symbol Generate
Lower Case	97 and 122	a-z
Upper Case	65 and 90	A-Z
Numeric	48 and 57	0-9
Special character	33 and 57	!"#$%&`()*+'-./

```
def buildblock(size):
        out_str = ''
        LowerCase =range(97,122)
        UpperCase =range(65,90)
        Numeric =range(48,57)
        SpecialChar =range(33,47)
        CombineString = LowerCase + UpperCase + Numeric + SpecialChar

        for i in range(0, size):
                a = random.choice(CombineString)
                out_str += chr(a)
        return(out_str)
```

Figure C.25 Code to Generate Request Query Using ASCII

The outcome derived from this section indicates that a web server accepts any characters sent from client including the request query with a special character. The results also show that there was no restriction at the web server to accept any irrelevant query, numbers, combination of capital letters and small letters with special character. Besides that, the usage of ASCII code makes the false query of GET headers easier to be generated randomly. Figure C.26 illustrates the results.

```
Request URI Query: 0)Io=YLVj=Vk)Mb=NlDpSM4ie$=DLyHTtSMD=O%G
    Request URI Query Parameter: 0)Io=YLVj=Vk)Mb=NlDpSM4ie$=DLyHTtSMD=O%G

Request URI Query: S%C=ATR$D1OMh=6G&Heo=SUoA3$8g"=2Y%=vF5v4PKW
    Request URI Query Parameter: S%C=ATR$D1OMh=6G
    Request URI Query Parameter: Heo=SUoA3$8g"=2Y%=vF5v4PKW

Request URI Query: 6Xo3(=Sm7S5iPO+w=2Yxj=*&W=0D"m8-=Hqj
    Request URI Query Parameter: 6Xo3(=Sm7S5iPO+w=2Yxj=*
    Request URI Query Parameter: W=0D"m8-=Hqj
```

Figure C.26 Forged Queries Processed by Web Server

C.23 GET Headers Vulnerabilities - Custom Values of GET Requests

This section performs to further investigate vulnerabilities that existed at GET headers. Investigation carried out in this section begins with a custom coding that is generated to simulate GET requests by assigning authentic and false values to user-agent, referrer, connection, language and accept-encoding. The valid value is copied from an original GET requests while invalid values utilized string that does not belong to the GET headers. Two GET headers known as connection and referrer were attached with a bogus value while other GET headers utilize genuine values. The GET headers known as connection is assigned to invalid value "Alive" which contradicts with genuine values. Furthermore, the referrer also carries an incorrect web address as the referrer must be in URL format. Figure C.27 presents the code while Figure C.28 indicates the format of GET requests once the code is executed.

```
Dim request As HttpWebRequest = HttpWebRequest.Create("http://42.1.63.189")

request.Host = request.Host
request.Connection = "Alive"
request.UserAgent = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
request.Accept = "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,
request.Referer = "http://test.local"
request.Headers.Add("Accept-Encoding", "gzip, deflate")
request.Headers.Add("Accept-Languange", "en,en-US;q=0.9")

Dim Response As HttpWebResponse = request.GetResponse
Dim ResponseStream As System.IO.Stream = Response.GetResponseStream

Dim StreamReader As New System.IO.StreamReader(ResponseStream)
Dim Data As String = StreamReader.ReadToEnd
StreamReader.Close()
```

Figure C.27 Code for GET Requests with Genuine and False Value

```
Hypertext Transfer Protocol
 GET / HTTP/1.1\r\n
 Connection: Alive\r\n
 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,
 Referer: http://test\r\n
 Accept-Encoding: gzip, deflate\r\n
 Accept-Languange: en,en-US;q=0.9\r\n
 Host: 42.1.63.189\r\n
```

Figure C.28 GET Headers with Genuine and False Value

This investigation found that legitimate value for GET headers connection is supposed to be Keep-Alive or keep-alive and the capital or small letter at the beginning of the word depends on a web browser as proven at Section C.2. Vigilant steps are required to be taken when GET headers such as connection is used to be part of the detection component as an incorrect inspection against this value will result in false positive as the value depends on the web browsers.

The investigation under this section continued with further exploration of vulnerabilities that existed at the HTTP protocol by assigning an incorrect value to all GET headers for instance, user-agent, referrer, connection, language, accept-encoding and adding another component in the GET requests. Results obtained from this scenario provide evidence that any values assigned to the GET headers component are accepted. Apart from that assigning new components in the GET headers is also allowed. The result in this section also indicates that the HTTP protocol is vulnerable to manipulation of fake GET requests as a GET header from clients to a web server is

not scrutinized to identify the authenticity of the GET requests. Figure C.29 illustrate the code followed by the result at Figure C.30.

```
Dim request As HttpWebRequest = HttpWebRequest.Create("http://42.1.63.189")

request.Host = "Invalid"
request.Connection = "Invalid"
request.UserAgent = "Invalid"
request.Accept = "Invalid"
request.Referer = "Invalid"
request.Headers.Add("Accept-Encoding", "Invalid")
request.Headers.Add("Accept-Languange", "Invalid")
request.Headers.Add("Additional-Header", "Invalid")

Dim Response As HttpWebResponse = request.GetResponse
Dim ResponseStream As System.IO.Stream = Response.GetResponseStream

Dim StreamReader As New System.IO.StreamReader(ResponseStream)
Dim Data As String = StreamReader.ReadToEnd
StreamReader.Close()
```

Figure C.29 Invalid Values of GET Headers With Additional GET Headers

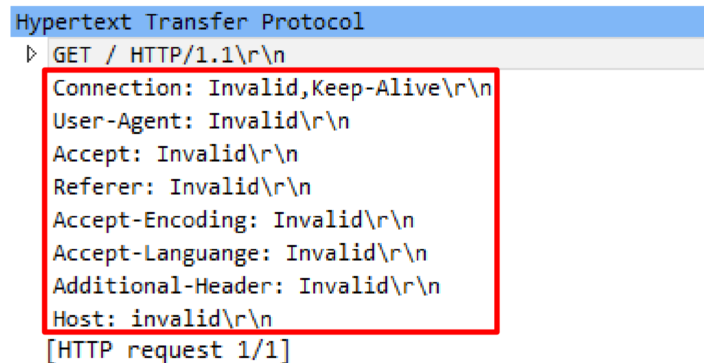

Figure C.30 Invalid GET Requests Value Processed by Server

C.24 Investigation Outcome

Investigation of eight datasets at Section C.4 until Section C.11 successfully found that the GET headers component and its values utilized during the occurrence of HTTP DDoS is almost similar where the traffic contained bogus GET headers and false value to make the traffic look genuine. Dataset comparison was conducted and highlighted in Section C.14 until Section C.18 to obtain information on the attack patterns, its similarities and differences that were produced by various HTTP DDoS attack script which will help in the formation of a correct detection to recognize HTTP DDoS. Outcome in this section also found that HTTP DDoS utilize forged GET header in executing HTTP DDoS in the internal and external network. Investigation of HTTP vulnerabilities discussed in Section C.20 until Section C.23 successfully revealed the vulnerabilities that exist at the application layer.

All datasets in this research provided clear indicators to suggest that HTTP DDoS has the capability of generating thousands of GET requests from a minimal number of machines with the help of an efficient attack script which directly provides further support on the results obtained in previous studies (Rahman et al., 2017; Beitollahi and Deconinck, 2013). The attack becomes more distributed when the group of attackers who launches the attack are in scattered locations (Iyengar and Ganapathy, 2015). According to Hoque et al. (2015), an automatic generation for the application DDoS attack is hard to find. However, it is possible to be executed with the help from a malicious code.

The use of public proxy to launch the attack leads to complexity as authentic users and attackers may utilize the equal proxy to surf the contents of a web server. All datasets investigated indicate that attackers utilized GET headers components in a variety of approaches to make a false GET requests to look genuine through HTTP DDoS. HTTP is a text-based protocol with no limit defined for the header's length and HTTP is an independent protocol which will accept any data type (Singh and Kumar, 2016; Fielding and Reschke, 2014b). Aside from that, most of the headers are not compulsory to be presented during transaction of GET requests as explained in Chapter

2 (Section 2.6). Due to weak design of HTTP protocol it poses security issues and attacker capable of to execute HTTP DDoS and make the attack look genuine.

As previously explained in Section C.1, the attack scripts adopted to conduct the investigation have been utilized and highlighted in a number of past studies. The GET headers produced by the attack scripts contradict with legitimate GET headers which have been thoroughly explained in Section C.2 to Section C.3. The investigation output also extends the elaboration made in past studies which explain about GET headers produced by HTTP DDoS which have previously highlighted in Chapter 2 (Section 2.9). Apart from that, the existence of security headers as described in Chapter 2 (Section 2.7) is designed only for HTTP POST protocol which allows an attacker to be able to execute HTTP DDoS which has a specific pattern.

Exploration of this research regards to HTTP DDoS has extended the attack patterns discussed in prior studies by revealing seven GET headers patterns constitute by the attack. The results obtained from the investigations discussed in Section C.4 until Section C.11 supported by dataset comparison conducted in Section C.14 until Section C.18 with strengthened by several sections involved in Chapter 2 as elaborated above, provide confirmatory findings that HTTP DDoS has a specific pattern to generate a false GET requests. A summary of the patterns is as follows:

i) Automated Tools

An attacker employs automated tools to generate massive GET requests as the tools require minimal duration to overload the web server which made the server become unresponsive and unable to react to the client request.

ii) Minimal GET Headers

HTTP DDoS delivers minimal GET headers to gain web server resources. Furthermore, the attack does not necessarily require a complete GET header to make a request against a web server. Supplying complete components for GET headers is only used to mimic user access pattern and to conceal their malicious activity. The incomplete GET header and higher GET requests received is a sign of HTTP DDoS.

iii) Incorrect Connection Status

Continuous GET requests received with a connection status marked as close indicates that HTTP DDoS currently occurs to overload a web server to process unnecessary request. Genuine GET requests transaction will mark connection status as keep-alive to show that a TCP connection is open and ready to receive connection. Apart from that, it is acknowledged that keep-alive is not a header for GET requests.

iv) Irrelevant GET Headers Attributes

GET headers contains several components which are adopted by clients during transaction of GET requests. Continuous irrelevant component which appears in each GET requests indicates that the source of the request does not come from a genuine client.

v) Irrelevant Request Query and Query Length

A user will utilize human language to search related contents in web servers and the search will be a query delivered through a GET header. Existence of request query in HTTP DDoS is likely to illustrate the query generated by humans. Query generated by humans is readable compared to false query generated by the attacker which contains special character such as !@#$^ to mimic the human access pattern when searching for information. The false query is also much longer than usual.

vi) Irrelevant Value for HTTP Referrer

HTTP referrer shows the previous access page of a client. A referrer must come from a related source such as a university URL will be a referrer for a faculty's web site. HTTP DDoS assigns irrelevant value in HTTP referrer to make it appear as though it comes from a valid source. Although the format and the URL are valid, it is irrelevant to be the source of reference.

vii) False Proxy Headers

Availability of public proxy allows the attacker to misuse the free services to launch a HTTP DDoS. There is no restriction to employ the services which allows the attacker to utilize single machine to command all the proxies to send

plenty of GET requests into a web server. Inconsistent of the appearance proxy header introduce to complexity to in recognizing either the source originated from proxy or directly from client. Aside from that, GET headers delivered by free proxies poses one of the challenges to recognize whether the source request is genuine or if it comes from an attacker due to minimal GET headers generated by proxy which contradicts with a GET headers delivery by non-proxy request.

The findings presented in Section C.9 successfully uncovered one attack strategy known as IP address spoofing capable of launching HTTP DDoS which expands the attack strategy listed by Singh et al. (2017b). The study also elaborated that multiple proxies were utilized to overwhelm a web server. However, they were unable to provide evidence on the use of various proxies to execute HTTP DDoS. To support this statement, results gained from Section C.10 and Section C.11 clearly indicates that usages of variety proxies to launch the attack. The outcome of this chapter also provides a better understanding for HTTP DDoS network architecture, especially when the attack engages a public proxy and spoof IP addresses. Figure C.31 illustrates the attack architecture for HTTP DDoS.

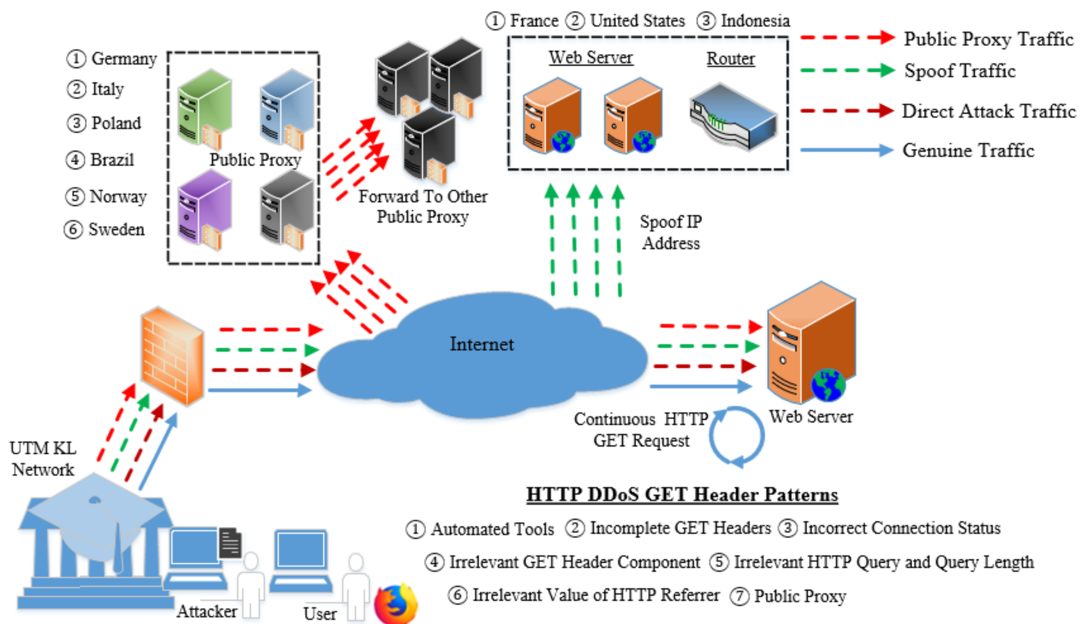

Figure C.31 HTTP DDoS Attack Architecture

The vulnerabilities investigation for HTTP protocol that specifically focused on GET headers has been conducted and reported in Section C.20 until Section C.23 provide further explanation pertaining to vulnerability that existed at the application layer. The outcome for GET requests shortfall reveal in this research successfully highlights five major drawbacks; (1) accept incomplete header, (2) no restriction for GET headers component, (3) no inspection for GET headers data, (4) open platform and (5) weak marking. Past studies (Singh and Kumar, 2016) noted that HTTP is an open platform and that there is no restriction of data.

Besides that, the use of cookie in HTTP allows information be retrieved as stated by Parziale et al. (2006). Apart from that, most of the GET headers involved in making request with the web server are not compulsory to be presented (Fielding and Reschke, 2014a; Fielding and Reschke, 2014b; Petersson and Nilsson, 2014; Reid, 2004). The drawback of the HTTP protocol introduced to HTTP DDoS which capable of making manipulation that results to GET requests to look authentic. This research proposes that the points also contribute to HTTP DDoS as follows.

i) Accept Minimal Headers

The GET requests does not require complete headers as the request will be processed by a web server even when a client sends minimal GET headers.

ii) No Restriction for GET Headers Attribute

Non-standard GET headers are acceptable to be processed by a web server as there was no objection from a web server to reject unrecognized GET headers.

iii) No Inspection for GET Headers Data

The GET headers values from a client to a web server does not go through any inspection to verify whether a value presented is correct or wrong. This is where server utilization will increase as it has to process unnecessary GET requests regardless whether it is valid or otherwise.

iv) Open Platform

HTTP is an open platform as it allows GET requests to execute without a web browser.

v) Weak Marking

No clear marking to show that the GET requests is generated through proxy, Network Address Translate (NAT), and direct connection.

Based on the investigation conducted, forged GET headers is the main factor that causes HTTP DDoS attack to look legitimate. Existence of GET headers in HTTP DDoS is only as a dummy to overload a server through plenty of GET requests. This atmosphere demonstrates the importance of recognizing a false GET headers to detect HTTP DDoS. Besides that, findings for HTTP DDoS pattern and GET headers vulnerabilities are parallel which leads to clearer insights on GET headers vulnerabilities resulted to formation of HTTP DDoS. Table C.7 shows vulnerabilities lead to HTTP DDoS attack.

Table C.7 HTTP DDoS Patterns and GET Headers Vulnerabilities

<table>
<tr><th>No</th><th>HTTP DDoS Patterns</th><th>GET Headers Vulnerabilities</th></tr>
<tr><td>1.</td><td>Automated Tools</td><td>Open Platform</td></tr>
<tr><td>2.</td><td>Incomplete GET headers</td><td>Accept Incomplete Header</td></tr>
<tr><td>3.</td><td>Incorrect Connection Status</td><td rowspan="4">No Inspection for GET headers Data
No Restriction for GET headers Component</td></tr>
<tr><td>4.</td><td>Irrelevant GET headers Component</td></tr>
<tr><td>5.</td><td>Irrelevant HTTP Query and Query Length</td></tr>
<tr><td>6.</td><td>Irrelevant Value of HTTP Referrer</td></tr>
<tr><td>7.</td><td>False Proxy Header</td><td>Weak Marking</td></tr>
</table>

C.25 Expansion of HTTP DDoS Knowledge Area

The findings in this chapter as explained in Section C.24 provide significant impact to the body of knowledge network and security as it delivers knowledge expansion about HTTP DDoS. Nine knowledge areas that are related to HTTP DDoS have been discussed in Chapter 2 which indicates that minimal prior studies delivered excessive experiment and provide detailed explanation with regards to the GET headers pattern during the occurrence of HTTP DDoS. Apart from that, the drawback of the GET headers that led to the formation of HTTP DDoS have not excessively revealed by an HTTP DDoS researcher. The knowledge expansion is illustrated in Figure C.32.

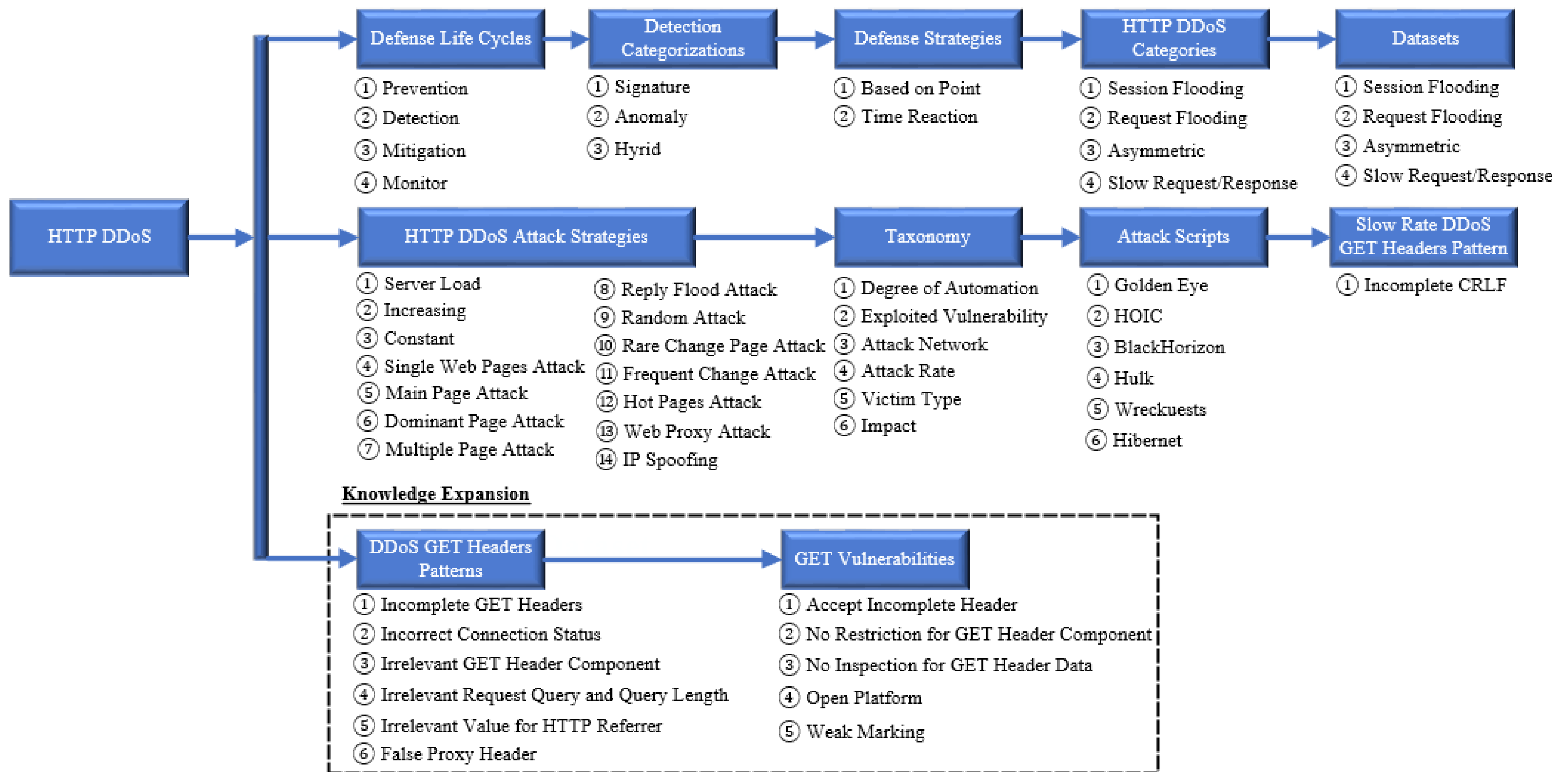

Figure C.32 HTTP DDoS Knowledge Area

APPENDIX D PRELIMINARY TEST

D.1 Test Case 1: GET Requests Frequency and Source Inspection

A high rate HTTP DDoS attack delivers plenty of requests to drain a server's resource and an enhanced version of the detection algorithm is able to detect HTTP DDoS attack when the GET requests passes the predefine threshold limit. Each request against a web server will be counted and has a specific threshold. A GET requests below than this threshold will not be detected as HTTP DDoS and a further inspection will not be executed. However, further inspection will be made if the connection received reaches the threshold.

The algorithm utilizes 20 GET requests of threshold to trigger and detect plenty of requests and inspection of web browser. Hence, to shorten the waiting time of high GET requests to reach the threshold, the threshold was set to 20 GET requests to simulate detection of high frequency of requests coming from a non-web browser while the high request comes from a web browser. This test case was conducted to evaluate whether the GET requests is counted correctly and triggered the inspection of browser check to detect HTTP DDoS once it reaches the threshold limit. It also recognizes the platform used whether it operates through a web browser or automated tools to attack a web server. Table D.1 shows the Test Case 1 attack parameter and Figure D.1 illustrates the Test Case 1 attack architecture.

Table D.1 Test Case 1 Parameter

Attributes	Values
Type of Attack	Direct Attack
Traffic Type	HTTP DDoS Traffic & Clean Traffic
Duration	1 Minute
Inspection Type	Threshold and Browser Inspection
Client 1 (Genuine)	94.242.111.68
Client 2 (Attacker)	115.164.221.151
Client 3 (Attacker)	161.139.153.30
Client 4 (Attacker)	78.30.214.81

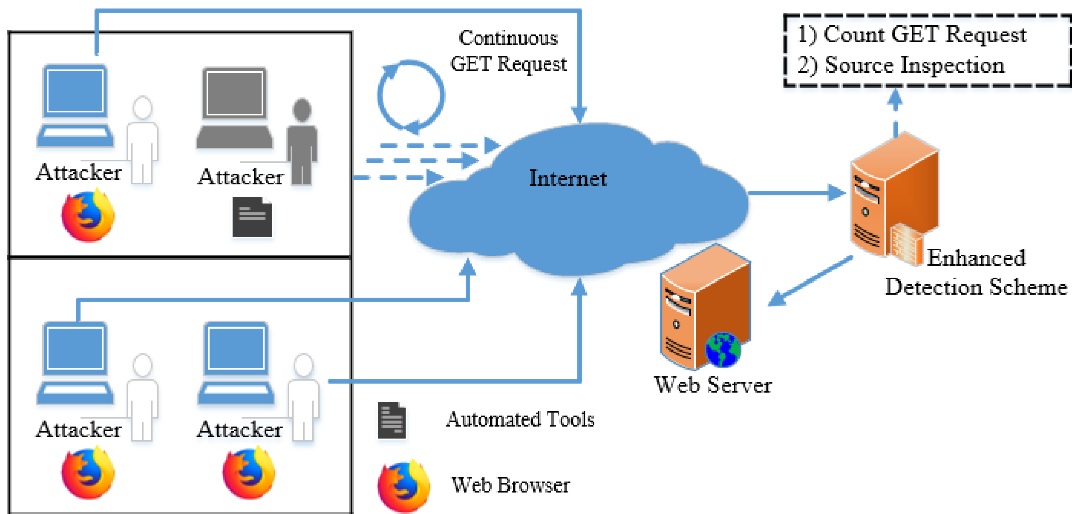

Figure D.1 Test Case 1 Architecture

This test case received 1042 GET requests which came from a genuine client and attacker. The results obtained from this test case indicate a positive outcome where all GET requests are counted individually and that the predefined threshold operates as expected. Results from this test case indicate that Client_1-94.242.111.68 operated below the threshold which only initiated 12 GET requests while clients labeled as Client_2-115.164.221.151, Client_3-161.139.153.30 and Client_4-78.30.214.81 made excessive GET requests against a web server that reached 20 GET requests which leads to the graph showing a significant increase starting from the predefined threshold. This also provides an indicator that the client sent huge number of GET requests against a web server. Besides that, continuous GET requests sent

against a web server causes the connection to increase gradually. Figure D.2 illustrates the detection threshold of GET requests.

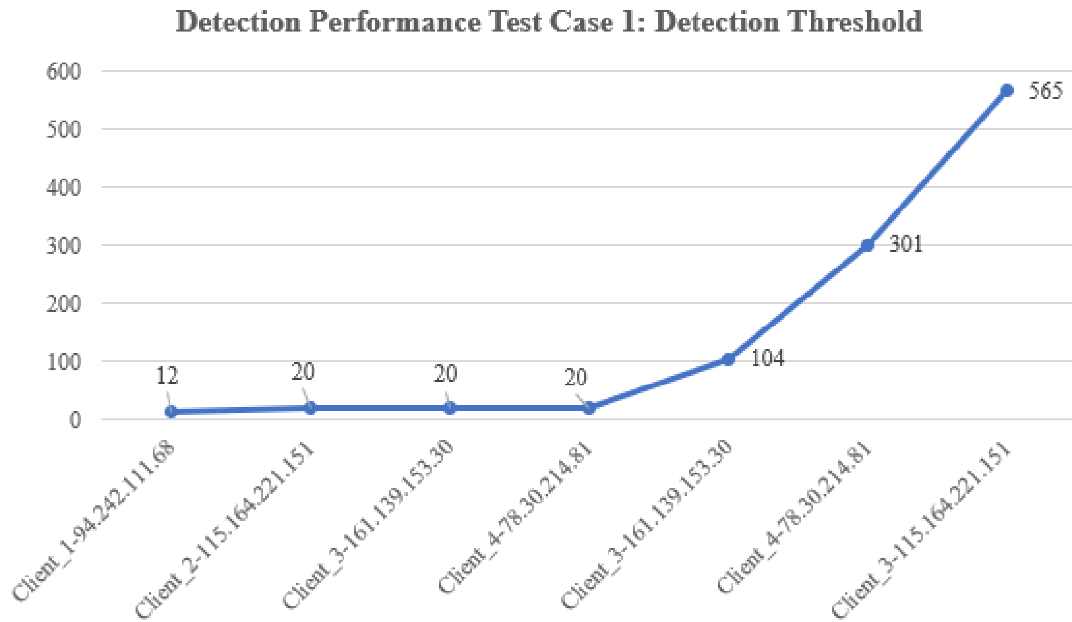

Figure D.2 Detection Threshold of GET Requests

HTTP DDoS constitute multiple platforms such as through automated tools to generate large volumes of HTTP traffics. Results from this test case show that the total traffic detected after a threshold is triggered was 970 and successfully detects 565 GET requests as true positive that utilized automated tools and 405 GET requests were detected as true negative that utilized web browsers to generate GET requests. The calculation of confusion matrix was made to obtain the rates of true positive, true negative, false positive, false negative, precision and accuracy. Results from this calculation indicate that the detection performance for all true positive rate, true negative rate, precision and accuracy were recorded to be 100% and 0% for false positive rate and false negative rate. Table D.2 presents the results for the detection performance for Test Case 1 followed by Figure D.3 which illustrates the performance graph for Test Case 1.

Table D.2 Detection Performance for Test Case 1

Actual Event Normal = 405 Attack = 565	**Normal**	**Attack**			
Normal	405	0			
Attack	0	565			
True Positive	**True Negative**	**False Positive**	**False Negative**	**Precision**	**Accuracy**
100%	100%	0.00%	0.00%	100%	100%

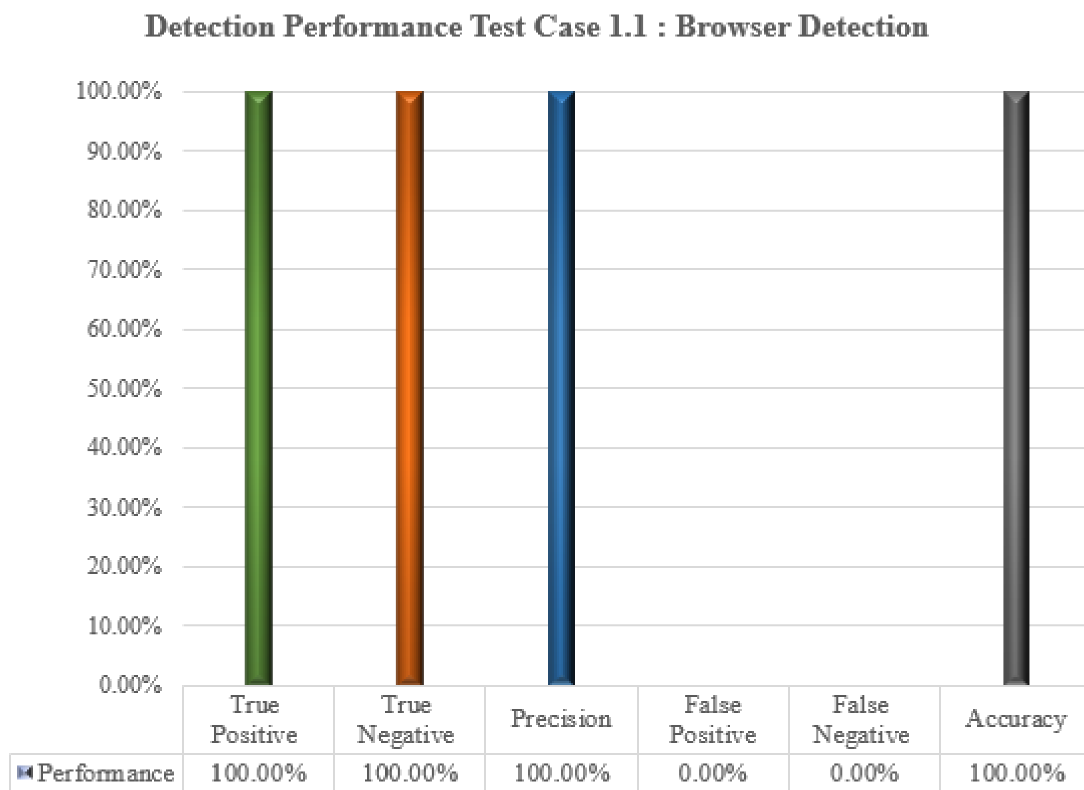

Figure D.3 Performance Graph for Test Case 1

The achievement of 100% detection rate is contributed by several factors such as forming the correct categorization to identify whether the source platform comes from automated tools or web browsers. The selection to adopt browser height and width as the detection components provides a significant impact to the higher rate for the detection results. HTTP DDoS is generated via automated tools and is therefore unable to provide browser height and width. Due to this, untrusted sources with high frequency of GET requests and over the predefined threshold are recognized as HTTP

DDoS. On the other hand, a genuine user will not consistently access the same web page with minimal duration. DDoS constitutes irrelevant request compared to authentic GET requests and the useless packets immediately consume resources to cause the server to become unresponsive (Cheng et al., 2018b).

Test Case 1 performed the detection of GET requests that was sent in a higher rate and initiated browser inspection which directly identifies whether the source connection is normal or HTTP DDoS. This test case has confirmed that the enhanced detection scheme operates as expected in detecting HTTP DDoS. Besides that, no missed calculation and incorrect identification of each GET requests was recorded thus allowing the test case to reach its target.

D.2 Test Case 2: Inspection of Compulsory GET Headers

This test case executed to scrutinize the existence of GET headers in GET requests. The formation of this test case accordance with the investigation result in Appendix C (Section C.4 until Section C.11) revealed that GET headers released by HTTP DDoS were inconsistent and contradict with genuine GET headers which supplied consistently as explained in Appendix C (Section C.2) and in Chapter 2 (Section 2.6). Table D.3 shows the Test Case 2 attack parameter while Figure D.4 illustrates the attack architecture for Test Case 2.

Table D.3 Test Case 2 Parameter

Attributes	Values
Type of Attack	Direct Attack
Traffic Type	HTTP DDoS Traffic & Clean Traffic
Duration	1 Minute
Number of HTTP Traffic Received	13916 GET requests
Inspection Type	GET Headers

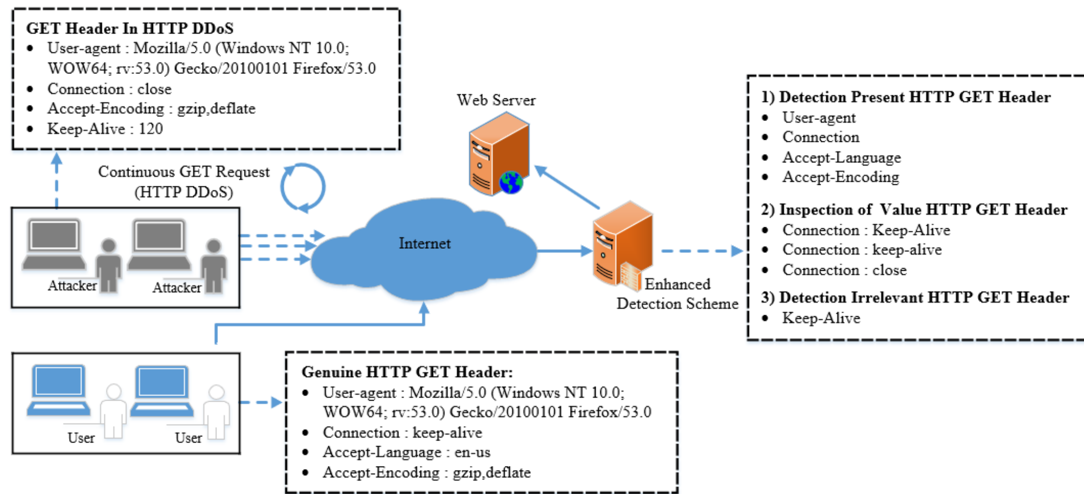

Figure D.4 Attack Architecture for Test Case 2

A total GET requests received from this test is 13916 which contained 59 legitimate traffic and 13857 originated from attacks. The genuine GET requests and attack traffics recognized precisely that led to 100% detection rate for true positive, true negative, precision, accuracy. The achievement of this performance is due to the GET headers supplied by the attack is incomplete for each transaction of GET requests. The insufficient GET headers pattern delivered by HTTP DDoS is fully recognized, which makes a correct classification of whether it is HTTP DDoS or genuine GET requests. The incomplete pattern is previously highlighted in Appendix C (Section C.6).

A genuine GET requests will deliver complete components and will appear in each GET requests. Gou et al. (2017) explained that authentic GET requests will supply sufficient components which commonly comprise of Host, Connection, Accept-Encoding, User-Agent, Accept-Language, Accept and Content-Type. Incomplete GET headers provide a signal that the source request is malicious. Previous studies have found that HTTP DDoS supplies an incomplete GET headers to a web server (Idhammad et al., 2018; Rahman et al., 2017; Liao et al., 2015).

Apart from that, based on the investigation results presented in Appendix C (Section C.2), genuine GET headers will consistently present a set of headers regardless of the browser versions used such as Internet Explorer, Google Chrome and Mozilla Firefox. The strategy to inspect common GET headers attached in GET

requests work perfectly and the performances are indicated in Table D.4 followed by the performance graph for Test Case 2 presented in Figure D.5.

Table D.4 Detection Performance for Test Case 2

Actual Event Normal = 59 Attack = 13857	**Normal**	**Attack**			
Normal	59	0			
Attack	0	13857			
True Positive	**True Negative**	**False Positive**	**False Negative**	**Precision**	**Accuracy**
100%	100%	0.00%	0.00%	100%	100%

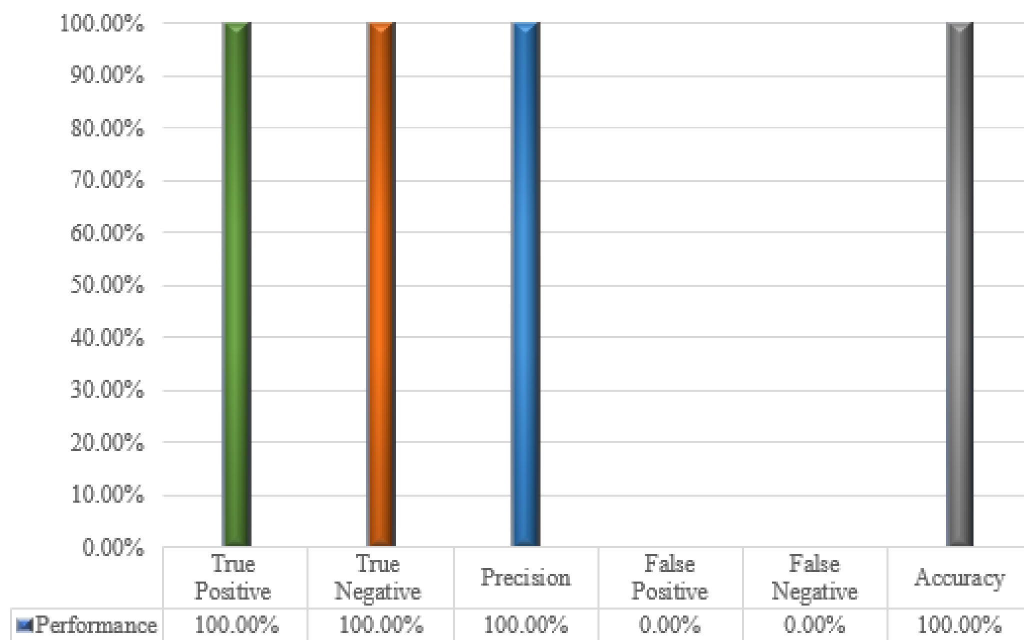

Figure D.5 Performance Graph for Test Case 2

D.3 Test Case 2.1: Dealing with Complete GET Headers

The GET requests created by HTTP DDoS capable of deliver complete GET headers match with authentic. Hence, execution of this test case to evaluate the capabilities of the enhanced detection scheme in handling such situation. Dealing with this circumstance requires inspection against value of the HTTP connection header where the status of the connection must be Keep-Alive or keep-alive to pass the inspection as status other than this will result in detection of HTTP DDoS attack. In this situation, the header connection status plays an important role as incorrect interpretation will result in false result thus allowing DDoS traffic to pass through the detection. This test case was executed in mix types of HTTP traffics whereby DDoS and clean HTTP traffics were executed simultaneously. Table D.5 presents details on Test Case 2.1 parameter.

Table D.5 Test Case 2.1 Parameter

Attributes	**Values**
Type of Attack	Direct Attack
Traffic Type	HTTP DDoS Traffic & Clean Traffic
Duration	1 Minute
Number of HTTP Traffic Received	10995 GET requests
Inspection Type	GET headers Value

Extension of this test case detects 59 GET requests as genuine and 10936 were marked as HTTP DDoS from a total of 10995 GET requests received. The enhanced detection reach reaches the target by marking HTTP connection which was marked with a close status as HTTP DDoS attack. The outcome from this extension indicates a 100% of true positive rate, true negative rate, precision and accuracy. No incorrect result was obtained which leads to 0% rate for false positive and false negative. Table D.6 illustrates the results for the detection performance for Test Case 2.1 followed by Figure D.6 which presents the performance graph for Test Case 2.1.

Table D.6 Detection Performance for Test Case 2.1

Actual Event Normal = 59 Attack = 10936	Normal	Attack			
Normal	59	0			
Attack	0	10936			
True Positive	**True Negative**	**False Positive**	**False Negative**	**Precision**	**Accuracy**
100%	100%	0.00%	0.00%	100%	100%

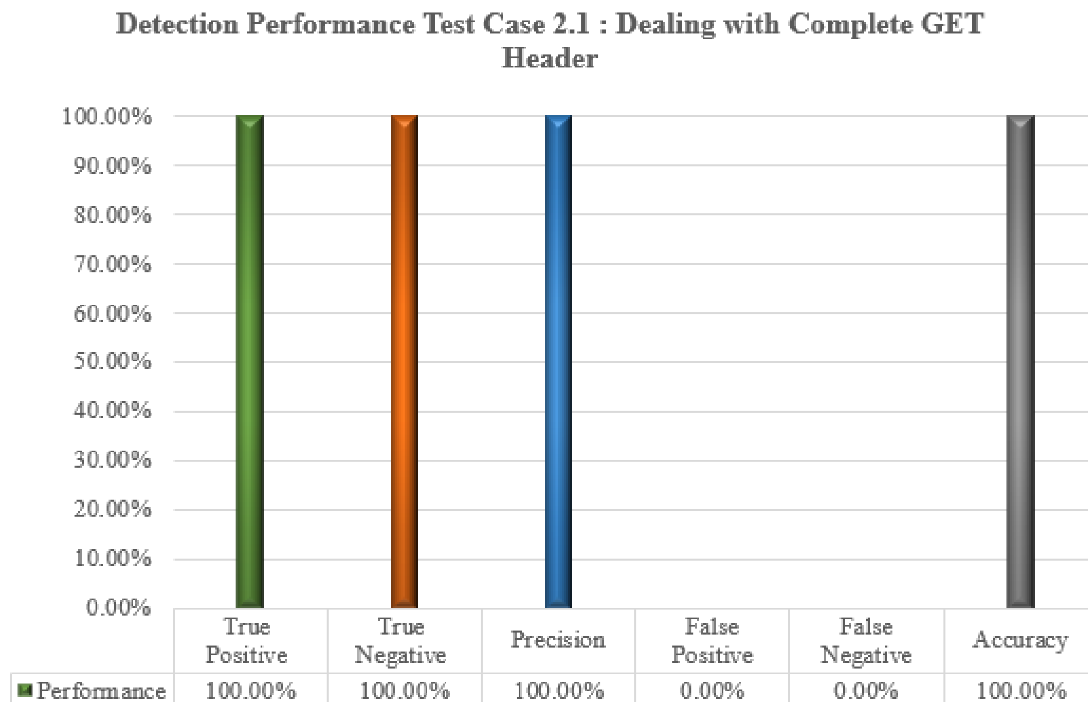

Figure D.6 Performance Graph for Test Case 2.1

This test case provides a clear indication that no unpredictable situation has occurred such as in case of normal HTTP traffic to be marked as close in HTTP connection header. In addition, inspection against the connection header status is designed to be a sub-inspection where a client must provide the required headers before this filter is executed. A genuine GET requests will supply keep-alive connection status and no GET requests will be received after the status has changed to close.

The status of keep alive illustrates that clients still maintain connection to a server and the status will turn to close once client specifies that the connection needs to be closed (Wong, 2000). As soon as the connection is marked as close, request will no longer be received from the client. However, during the occurrence of HTTP DDoS, the connection status is close with continuous GET requests received by a web server as elaborated in Appendix C (Section C.4). HTTP DDoS pattern contradicts with normal transaction of genuine GET requests which makes the malicious pattern to be detected at 100% as HTTP DDoS.

D.4 Test Case 2.2: Dealing with Irrelevant GET Headers

The vulnerabilities at the application layer allow irrelevant GET headers to be included in GET requests. Thus, further expansion from previous test case (Test Case 2 and Test Case 2.1) is conducted by predict that attackers are able to emulate genuine GET headers and successfully manipulated the value of the HTTP connection header. An extension of this test case is in line with the investigation result presented in Appendix C (Section C.8) where HTTP DDoS submits a wrong header in GET requests where keep alive is supposed to be the value of connection headers which were placed incorrectly. Table D.7 presents details on the Test Case 2.2 parameter.

Table D.7 Test Case 2.2 Parameter

Attributes	Values
Type of Attack	Direct Attack
Traffic Type	HTTP DDoS Traffic & Clean Traffic
Duration	1 Minute
Number of HTTP Traffic Received	11277 GET requests
Inspection Type	Irrelevant GET headers

A total GET requests from this expansion is recorded at 11277 which contained 59 genuine traffics and 11218 attacks. Results obtained from this test case show that the enhance detection scheme still maintains its performance. All performance matrices such as true positive rate, true negative rate, precision and accuracy indicated a 100% detection. This means that malicious HTTP traffic which comes from HTTP DDoS attack and genuine traffic which originates from users were able to be distinguished successfully. A percentage zero was also obtained for false positive rate and false negative rate as no misclassification occurred. Table D.8 presents the detection performance for Test Case 2.2 followed by Figure D.7 which displays the performance result for Test Case 2.2 in graphical view.

Table D.8 Detection Performance for Test Case 2.2

Actual Event Normal = 59 Attack = 11218	**Normal**	**Attack**			
Normal	59	0			
Attack	0	11218			
True Positive	**True Negative**	**False Positive**	**False Negative**	**Precision**	**Accuracy**
100%	100%	0.00%	0.00%	100%	100%

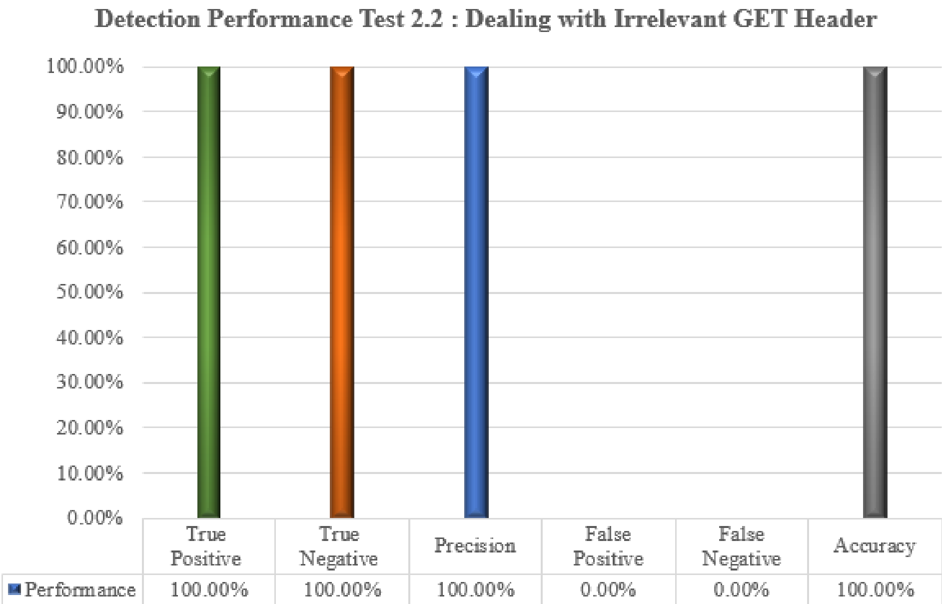

Figure D.7 Performance Graph for Test Case 2.2

This test case illustrates that irrelevant GET headers known as keep-alive was misused and is located at GET headers components. The detection inspects the GET headers utilized by a client to browse contents of a web server. An incorrect supply of GET headers component will result in detection of HTTP DDoS. This test case scenario indicates that attackers accidentally supply incorrect components to be inserted in GET headers components which leads to 100% performance result in recognizing the attack pattern. The Keep-Alive is the value of connection component in GET requests and HTTP Response. The Keep-Alive is also the component for HTTP Response. Figure D.8 provides a graphical view of GET requests headers while Figure D.9 illustrates the HTTP response headers.

```
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macintosh;… )… Firefox/51.0
Accept: text/html,application/xhtml+xml,…,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Content-Type: multipart/form-data; boundary=-12656974
Content-Length: 345
```

Figure D.8 HTTP Request Headers (https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages#)

```
Access-Control-Allow-Origin: *
Connection: Keep-Alive
Content-Encoding: gzip
Content-Type: text/html; charset=utf-8
Date: Wed, 10 Aug 2016 13:17:18 GMT
Etag: "d9b3b803e9a0dc6f22e2f20a3e90f69c41f6b71b"
Keep-Alive: timeout=5, max=999
Last-Modified: Wed, 10 Aug 2016 05:38:31 GMT
Server: Apache
Set-Cookie: csrftoken=……
Transfer-Encoding: chunked
Vary: Cookie, Accept-Encoding
X-Frame-Options: DENY
```

Figure D.9 HTTP Response Headers (https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages#)

As a conclusion for Test Case 2, 2.1 and 2.2, the test cases performed inquiries pertaining to absent GET headers, connection status and value of the connection header. Each GET requests will be inspected and failure to adhere to one of the rules will result in detection of HTTP DDoS. This test case has successfully gathered a positive output where a malicious pattern produced by HTTP DDoS can be fully detected. Results in this test case also show that the strategy to inspect the existence of four headers such as accept-language, accept-encoding, connection and user-agent to detect HTTP DDoS attack has been completed successfully without any false classification. Results received from this test case also provide confirmatory findings that genuine GET requests will supply correct and complete headers and failure to fulfill this rule results in suspicious GET requests and is thereby detected as HTTP DDoS.

D.5 Test Case 3: False Request Query

This test case is conducted to evaluate the detection performance to recognize a delivery of false query that constitutes by HTTP DDoS. The formation of this test case is in line with the investigation results presented in Appendix C (Section C.4 Section C.5, Section C.8, Section C.9 and Section C.11) where the results indicate that bogus request queries were sent against a web server. The vulnerabilities result in Appendix C (Section C.24) prove that any submitted query accepted by a web server allows an attacker to manipulate GET headers to create a false query in GET requests to form HTTP DDoS. In this test case, bad and clean traffics were mixed together to observe the precision the enhanced detection scheme extracts and compares a present string with a string database. A GET requests that is delivered with query must comply with certain rules to avoid detection of HTTP DDoS. Table D.9 indicates details on Test Case 3 parameter while Figure D.10 illustrates the Test Case 3 architecture.

Table D.9 Test Case 3 Parameter

Attributes	Values
Type of Attack	Direct Attack
Traffic Type	Bad traffic & clean traffic
Attack Duration	1 Minutes
Inspection Type	Request query
Accept Query	• Lower case • String exist at string database • Accept String (register subject, student fees, student fees semester 3 semester duration, lecturer name)
Reject Query	• Upper case • Symbol

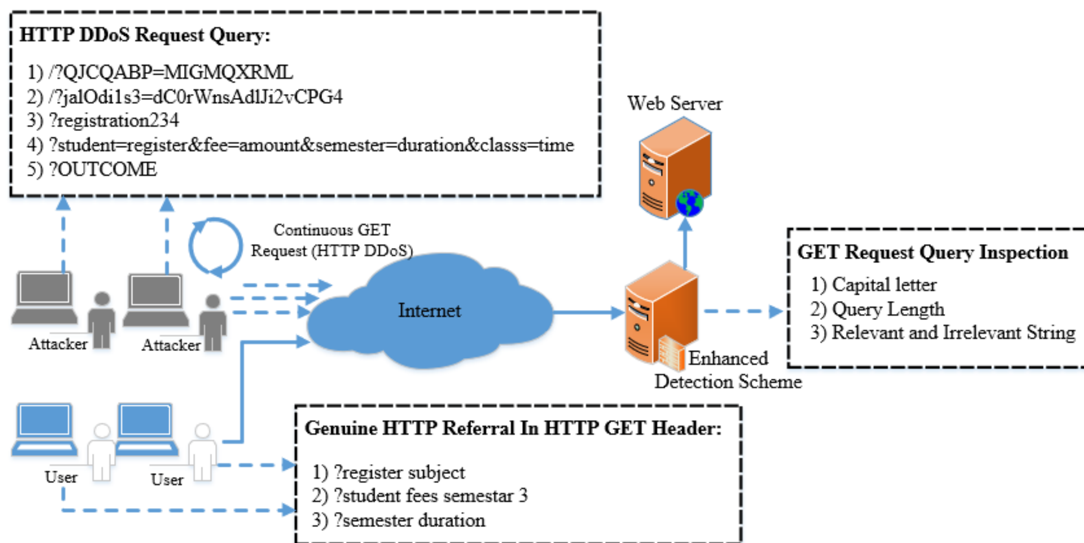

Figure D.10 Architecture for Test Case 3

The total GET requests received was 6971 and out of this value, 271 GET requests have been detected as true negative which indicate that the GET requests contain clean query while 6700 GET requests were recognized as true positive. The GET requests detected as forged due to massive queries received and a web server content contradicted with the queries. HTTP DDoS creates an excessive search request (Prasad et al., 2017). All false queries and authentic queries were recognized precisely which lead to zero percentages of false detections such as false positive and false

negative. The detection rate of true positive, true negative, precision and accuracy indicated 100% rate and 0% rate for false positive and false negative.

Request queries generated by HTTP DDoS through automated tools are much longer and meaningless as they contain a combination of capital letters, small letters, numbers and symbols. Besides that, although HTTP DDoS is capable of generating query understood by humans, the supplied query is irrelevant to be utilized by humans. Request queries made by human are more readable and relate to the contents of a web server. Taking this into account, examining the query with the string database is seen as an appropriate option to be applied to determine the relevance of the query which allows the results in this section to yield a 100% performance rate in recognizing genuine and bogus queries. Irrelevant queries in GET requests and requests in high frequency provides sign HTTP DDoS has occurred.

The expected outcome obtained from this test case indicates that the flow of the detection and algorithm work as coded. Besides that, the results show extraction of queries from GET headers and compare the query with the string database in real time managed to operate smoothly. The string database plays a vital role as it contains a list of key words to determine the relevance of the received query in GET requests. There is no barrier for GET requests to carry any data type which allows HTTP DDoS generates various strings in GET requests to overwhelm a web server. Singh and Kumar (2016) explained that HTTP was an independent protocol and it accepts any data type. Table D.10 presents the detection performance results for Test Case 3 while Figure D.11 illustrates the performance graph for Test Case 3.

Table D.10 Detection Performance for Test Case 3

Actual Event Normal = 271 Attack = 6700	**Normal**	**Attack**			
Normal	271	0			
Attack	0	6700			
True Positive	**True Negative**	**False Positive**	**False Negative**	**Precision**	**Accuracy**
100%	100%	0.00%	0.00%	100%	100%

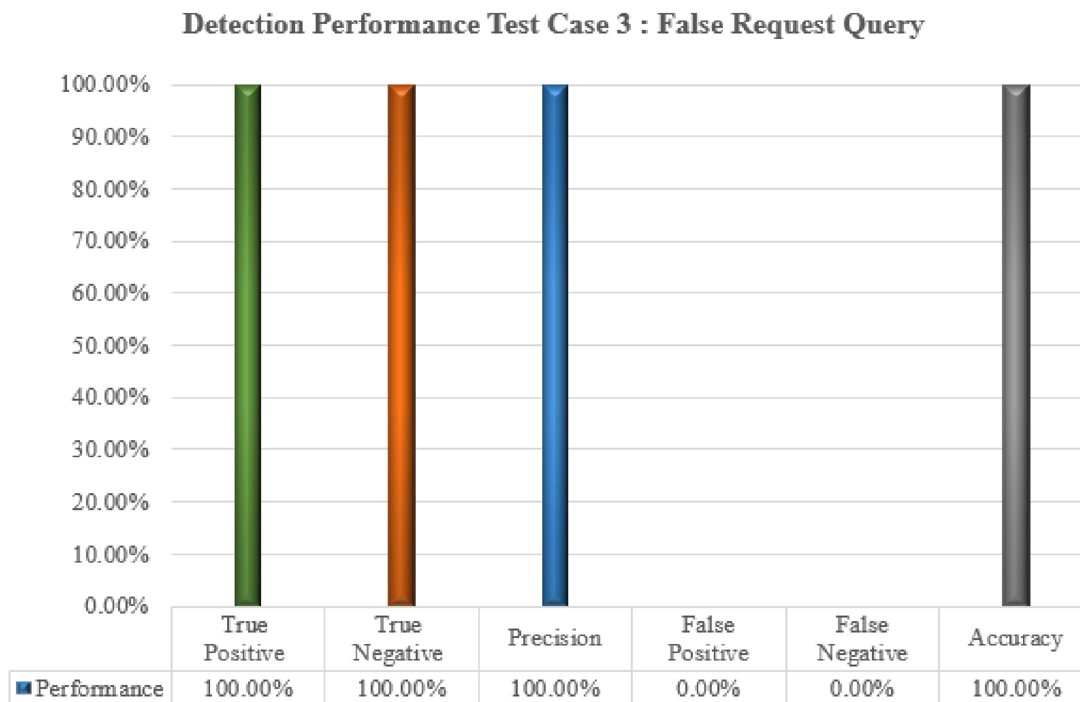

Figure D.11 Performance Graph for Test Case 3

Test Case 3 illustrates that the false GET requests delivered by HTTP DDoS is capable to be detected. Existence of this detection will not allow an attacker to generate bogus and irrelevant queries. Capturing the request query pattern produced by HTTP DDoS is essential to be included in the string database. Obsolete string database will result in false query generated by HTTP DDoS recognized as genuine.

D.6 Test Case 4: Source Inspection Proxy or None Proxy

Proxy utilizations during GET requests present headers known as X-Forwarded-For and Via. Commonly, both headers are paired together during the process of GET requests and the presence of one of these headers is a sign of suspicious GET requests. However, in certain circumstances, proxy headers are absent which lead to difficulties in identifying whether the source connection comes from a proxy or other platforms. Investigation results presented in Appendix C (Section C.9 until Section C.11) thoroughly discussed the public proxy functionality and how attackers manipulate the usage of the public proxy to launch HTTP DDoS attack.

HTTP DDoS detection launched through a proxy has to be detected in a specific sequence. Firstly, recognizing the source platform is essential whether it comes from a proxy or non-proxy as a proxy provides different GET headers. Once recognition of the source connection is complete, inspection against the GET requests traffic will be done to determine whether the traffic is HTTP DDoS or genuine. This test case began with an inspection of source connection to determine the source connection whether it comes from a proxy or direct connection to access a web server. Detection of malicious GET requests will be done in the subsequent test case. Table D.11 shows details on Test Case 4 parameter and Figure D.12 illustrates the architecture for Test Case 4.

Table D.11 Test Case 4 Parameter

No	Attributes	Values
1.	Type of Attack	Attack through proxy
2.	Traffic Type	Bad traffic and clean traffic
3.	Attack Duration	1 Minute
4.	Inspection Type	Proxy

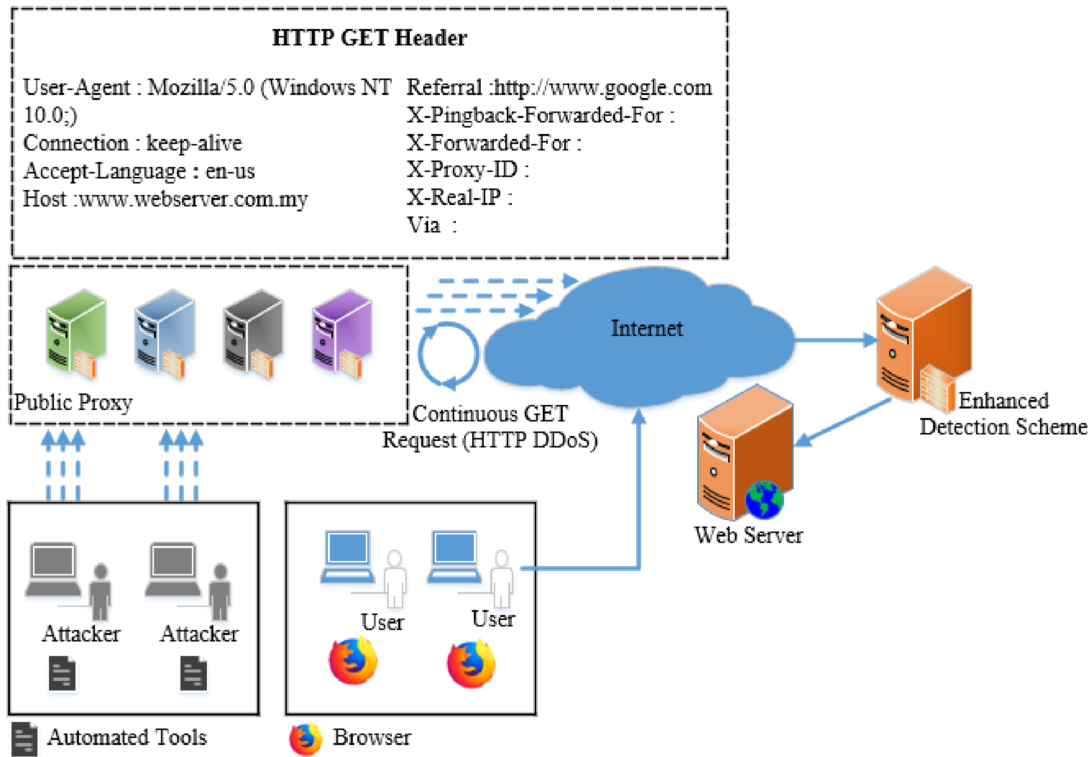

Figure D.12 Architecture for Test Case 4

This test case received 8809 GET requests and successful discovery of GET requests not utilizing proxy 38 resulted in a 100% true negative rate and 0% for false positive rate. Despite the detection of non-proxy fulfill the detection objective, incorrect classification was also recorded as 77 GET requests which employed proxies were recognized as normal connection. Apart from that, the detection of GET requests which adopted proxy were recorded at 8694 and inclusion of incorrect classification leads to a true positive rate of 99.12% and a false negative rate of 0.88%. Precision indicates a 100% rate as there was not any normal connection that was detected as proxy while an accuracy of 99.13% was recorded because it is influenced by incorrect classification. Table D.12 presents the detection performance results for Test Case 4 followed by Figure D.13 which illustrates the performance graph for Test Case 4.

Table D.12 Detection Performance for Test Case 4

Actual Event Normal = 38 Proxy = 8771	Normal	Proxy			
Normal	38	0			
Proxy	77	8694			
True Positive	**True Negative**	**False Positive**	**False Negative**	**Precision**	**Accuracy**
99.12%	100%	0.00%	0.88%	100%	99.13%

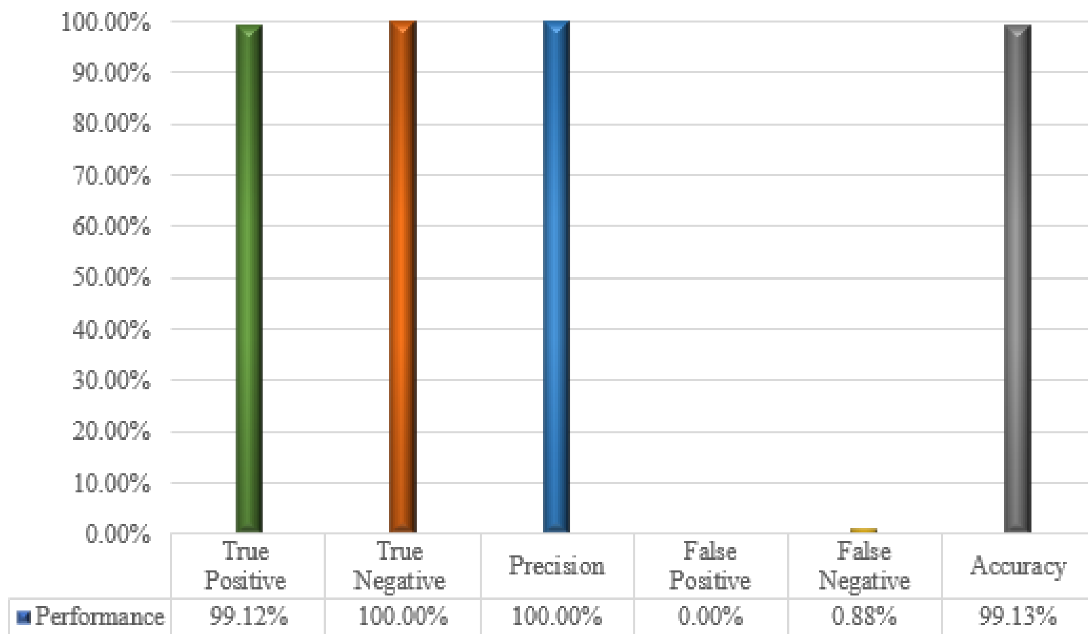

Figure D.13 Performance Graph for Test Case 4

Results show that the performance dropped due to imprecise classification as GET requests utilizing the proxy was detected as non-proxy. Inspection of this result was conducted and indicated that the IP address of the proxy was not included inside the proxy database. As explained earlier in Chapter 4 (Section 4.7), proxy detection requires a dual layer which performs inquiries against proxy headers and a further inspection with a proxy database to confirm whether the source connection comes from a proxy. In this case, the proxy IP address is non-existent in the proxy IP database which leads to an incorrect categorization.

Besides that, the proxy provides absent proxy headers which caused the first layer of the enhanced detection scheme to detect proxy unable to mark the source as a proxy and only relies on the proxy database to determine whether it is a proxy or a normal connection. The proxy database requires frequent updates to prevent incorrect classification of source connection. The existence of new proxy IP which is not included in the proxy database provides a higher possibility in the performance drop. Furthermore, some of the proxies were not showing proxy headers as explained in Appendix C (Section C.9 until Section C.11) which will result in a misclassification in detecting whether the source connection is a proxy or a normal connection.

D.7 Test Case 4.1: Dealing with Attack Through Proxy

As explained earlier, HTTP DDoS through proxy is required to be detected in a particular sequence. The first inspection identifies whether the type of source connection comes from a specific platform which has been executed earlier. The second stage counts the threshold and the type of platform source used to generate a GET requests whether it is constituted from web browsers or automated tools. Results indicate that 892 GET requests have been detected as clean GET requests which recorded a true negative rate of 100%. Accurate detection of authentic GET request leads to zero attack traffic to be recognized as genuine which results in a false positive rate of 0.00%. The amount of inaccurate classification is 350 which indicates that the attack traffics were incorrectly marked as genuine and recorded a 95.58% of true positive rate in line with a false negative rate of 4.42%. Precision shows a 100% rate while accuracy gained a 96.03% rate as imprecise classification provides a significant impact that leads to accuracy to obtain such rate (96.03%).

An inspection of GET request that was launched through proxy was executed when the proxy threshold reaches the limit. Results above indicate that the strategy to utilize proxy headers and proxy IP database with a combination of threshold to constitute the detection operated as expected. However, this test case recorded a slight performance drop unlike the previous test case which indicates an efficient detection performance and operates as coded. The cause of the performance drop due to a higher

resource utilization as noted by prior studies and explained in Chapter 2 (Section 2.3) in which HTTP DDoS consumes resources which results in the server to become unresponsive to process GET requests. On the other hand, proxy IP address that launched the attack was not included in the proxy database which allows a malicious request to reach a web server and therefore results in the detection performance to drop.

HTTP DDoS through proxy is difficult to be detected as the same proxy can be used by attackers and authentic users. Besides that, to differentiate whether both parties make a genuine request or an attack is challenging as they utilize equal resources to access a web server. Pandiaraja and Manikandan (2015) noted that proxy only able to identify the identity of the client system as IP address and the client GET headers are received by the proxy. However, the identity of the initiator whether attacker or genuine users are difficult to discover. The detection of HTTP DDoS attack launched through proxy requires a double inspection to identify whether the source connection comes from a proxy or vice versa. Although complete GET headers presented by proxy to attack a web server, higher GET requests generated from proxy and automated tools utilized by the source connection provide a clear sign that HTTP DDoS has occurred. Table D.13 indicates the result on the detection of HTTP DDoS followed by a graphical view of the results presented in Figure D.14.

Table D.13 Detection Performance for Test Case 4.1

Actual Event Normal = 892 Attack = 7567	Normal	Attack			
Normal	892	0			
Attack	350	7567			
True Positive	**True Negative**	**False Positive**	**False Negative**	**Precision**	**Accuracy**
95.58%	100%	0.00%	4.42%	100%	96.03%

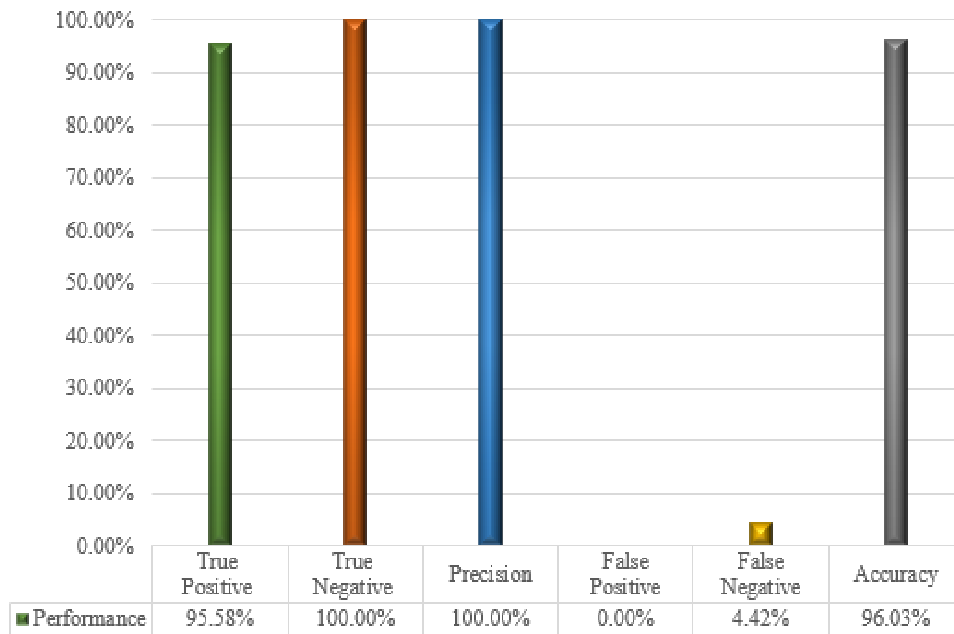

Figure D.14 Performance Graph for Test Case 4.1

Test Case 4.1 focused on the detection of HTTP DDoS that is launched through proxies. This test case is completed with a slight performance drop. Proxy detection requires two stages and these stages operated as expected to recognize proxies that launched the attack. A proxy requires frequent updates as different proxy providers utilized different names to inform a web server the source of the GET requests is originated from proxy. Apart from that, the proxy IP addresses also requires constant updates to facilitate detection of HTTP DDoS through proxy.

APPENDIX E SOURCE INSPECTION

```
Sub InspectionHighRate()
Dim HTTPRequestPrevious = DateTime.Today.AddDays(-
1).ToString("dd:MM:yyyy")
Dim CheckHTTPRequestDate As Integer = 0
For Each line As String In
File.ReadLines("C:\HTTPLogs\HighRequest\CountGETRequest.txt")
If line.Contains(HTTPRequestPrevious) Then
CheckHTTPRequestDate = CheckHTTPRequestDate + 1
End If
Next line

If CheckHTTPRequestDate > 0 Then
File.WriteAllText("C:\HTTPLogs\HighRequest\CountGETRequest.txt",
String.Empty.Trim)
End If

Dim HTTPRequestdate = DateTime.Now.ToString("dd:MM:yyyy")
Dim ClientIP As Object = Request.ServerVariables("REMOTE_ADDR")
Dim StrPath = Request.Path
Dim StrPathReplace As String
StrPathReplace = StrPath.Replace("/", "")
Dim combine = HTTPRequestdate + "-" + ClientIP + "-" + StrPathReplace
Dim path As String = "C:\HTTPLogs\HighRequest\CountGETRequest.txt"
Using sw As StreamWriter = File.AppendText(path)
sw.WriteLine(combine)
End Using

Dim input = File.ReadAllText("C:\HTTPLogs\HighRequest\CountGETRequest.txt")
Dim pattern1 = ClientIP
Dim pattern2 = Request.Path
```

```
Dim pattern3 = HTTPRequestdate
Response.Write(combine)
Dim matches = Regex.Matches(input, combine)
Dim count = matches.Count
Response.Write("<br>" & count)

If count > 20 Then
CheckBrowser()
Else
BelowThreshold.BelowThresholdCount(combine)
End If
End Sub

Imports System.IO
Public Class BelowThreshold

Shared Sub BelowThresholdCount(ByVal BelowThresholdCount As Object)
Dim text As String = "First line" & Environment.NewLine
Dim path As String = "C:\HTTPLogs\HighRequest\BelowThresholdCount.txt"
Using WQAttack As StreamWriter = File.AppendText(path)
WQAttack.WriteLine("Below Threshold: " & BelowThresholdCount)
End Using
End Sub

Shared Sub Proxy_BelowThresholdCount(ByVal Proxy_BelowThresholdCount As
Object)
Dim text As String = "First line" & Environment.NewLine
Dim path As String = "C:\HTTPLogs\Proxy\ProxyIP\Proxy_BelowThreshold.txt"
Using WQAttack As StreamWriter = File.AppendText(path)
WQAttack.WriteLine("Proxy Below Threshold: " & Proxy_BelowThresholdCount)
End Using
End Sub
End Class
Sub CheckBrowser()
```

```
Dim width = hidwidth.Value
Dim heigh = hidheight.Value
Dim input = File.ReadAllText("C:\HTTPLogs\HighRequest\CountGETRequest.txt")
Dim ClientIP As Object = Request.ServerVariables("REMOTE_ADDR")
Dim HTTPRequestdate = DateTime.Now.ToString("dd:MM:yyyy")
Dim combine = HTTPRequestdate + "-" + ClientIP + "-" + Request.Path

If Not IsPostBack Then
Dim script As String = "window.onload = function() { SetHidValue();
};"Page.ClientScript.RegisterClientScriptBlock(Me.GetType(), "setvalues", script,
True)

If hidheight.Value = String.Empty And hidwidth.Value = String.Empty Then
Dim text As String = "First line" & Environment.NewLine
Dim path As String = "C:\HTTPLogs\HighRequest\NoBrowser.txt"
Using WQAttack As StreamWriter = File.AppendText(path)
WQAttack.WriteLine("No Browser Detected:" & ClientIP)
End Using
End If
Else

If Not (hidheight.Value = String.Empty) And Not (hidwidth.Value = String.Empty)
Then
Response.Write("Width: " + hidwidth.Value + " Height: " + hidheight.Value)
Dim UA As String = Request.UserAgent
Dim text As String = "First line" & Environment.NewLine
Dim path As String = "C:\HTTPLogs\HighRequest\Browser.txt"
Using WQAttack As StreamWriter = File.AppendText(path)
WQAttack.WriteLine("Browser Detect :" & UA & "-" & ClientIP)
End Using
End If
End If
GetHeader_Inspection()
End Sub
```

```
<script type="text/javascript">
var width = screen.width;
var height = screen.height;

function SetHidValue()
{
  document.getElementById("hidwidth").value = width;
  document.getElementById("hidheight").value = height;
  document.getElementById("default").submit();
};
</script>
```

APPENDIX F GET HEADERS INSPECTION

```
Sub GetHeader_Inspection()
Dim ClientIP As Object = Request.ServerVariables("REMOTE_ADDR")
CountGETRequest.CountGETRequest("HTTP GET Request", ClientIP)

Dim AcceptLanguange As Object =
Context.Request.ServerVariables.AllKeys.Contains("HTTP_ACCEPT_LANGUAG
E")
Dim UserAgent As Object =
Context.Request.ServerVariables.AllKeys.Contains("HTTP_USER_AGENT")
Dim Connection As Object =
Context.Request.ServerVariables.AllKeys.Contains("HTTP_CONNECTION")
Dim AcceptEncoding As Object =
Context.Request.ServerVariables.AllKeys.Contains("HTTP_ACCEPT_ENCODING
")

Dim UAgent As Object =
Context.Request.ServerVariables("HTTP_USER_AGENT")
Dim ConnectionStatus As Object =
Context.Request.ServerVariables("HTTP_CONNECTION")
Dim HTTPVersion As Object =
Context.Request.ServerVariables("HTTP_VERSION")
Dim header() As String = Context.Request.Headers.AllKeys

If (AcceptLanguange And UserAgent And Connection And AcceptEncoding) Then
Dim countheader As Integer = 0
For Each line As String In
File.ReadLines("C:\HTTPLogs\GetHeader\IrreleventHeader.txt")
If header.Contains(line) Then
countheader = countheader + 1
End If
```

```
Next line
If countheader > 0 Then
'Detect HTTP DDoS
GetHeaderInspection.MaliciousHeaders("Detect Malicious Headers")

ElseIf ConnectionStatus.Equals("Keep-Alive") Or ConnectionStatus.Equals("keep-
alive") Then
'Clean HTTP Request | 'Forward Next Layer / Main Page
GetHeaderInspection.CleanHeaders("Clean HTTP Traffic : Complete HTTP
Headers", ClientIP)
ElseIf ConnectionStatus.Equals("close") Then
'Detect HTTP DDoS
GetHeaderInspection.ConnectionStatus("Connection Close")
End If

Else
'Detect HTTP DDoS
GetHeaderInspection.IncompleteHeaders("Incomplete HTTP Headers", ClientIP)
End If
QStringInspection()
End Sub

Imports Microsoft.VisualBasic
Imports System.IO
Public Class GetHeaderInspection

Shared Sub MaliciousHeaders (ByVal MaliciousHeaders As Object)
Dim text As String = "First line" & Environment.NewLine
Dim path As String = "C:\HTTPLogs\GetHeader\MaliciousHeader.txt"
Using WQAttack As StreamWriter = File.AppendText(path)
WQAttack.WriteLine(MaliciousHeaders)
End Using
End Sub
```

```
Shared Sub CleanHeaders(ByVal CleanHeader As Object, ByVal ClientIP As
Object)
Dim text As String = "First line" & Environment.NewLine
Dim path As String = "C:\HTTPLogs\GetHeader\CleanHeader.txt"
Using WQAttack As StreamWriter = File.AppendText(path)
WQAttack.WriteLine(CleanHeader + "-" + ClientIP)
End Using
End Sub

Shared Sub ConnectionStatus(ByVal ConnectionStatus As Object)
Dim text As String = "First line" & Environment.NewLine
Dim path As String = "C:\HTTPLogs\GetHeader\ConnectionStatus.txt"
Using WQAttack As StreamWriter = File.AppendText(path)
WQAttack.WriteLine(ConnectionStatus)
End Using
End Sub
Shared Sub IncompleteHeaders(ByVal IncompleteHeader As Object, ByVal ClientIP
As Object)
Dim text As String = "First line" & Environment.NewLine
Dim path As String = "C:\HTTPLogs\GetHeader\IncompleteHeader.txt"
Using WQAttack As StreamWriter = File.AppendText(path)
WQAttack.WriteLine(IncompleteHeader + "-" + ClientIP)
End Using
End Sub
End Class
```

APPENDIX G REQUEST QUERY INSPECTION

```
Sub QStringInspection()
Dim ClientIP As Object = Request.ServerVariables("REMOTE_ADDR")
Dim PresentString As String = Request.QueryString.ToString()
Dim Qlength As Int32 = Request.QueryString.Count.ToString
Dim NewString As String
Dim Stringcheck As Boolean
QString.QStringCountGETRequest(PresentString, ClientIP)

If PresentString.Equals("") Then
'No Query | 'Forward Next Layer / Main Page
QString.QStringNoQuery(ClientIP)
Else

If PresentString.Contains("+") Then
NewString = PresentString.Replace("+", " ")

If Stringcheck = Regex.IsMatch(NewString, "[`~!@#\$%\^\*\(\)_\-
\+\{\}\[\]\\\|:;""'<>,\.\/A-Z]")  Then

If Qlength >= 4 Then
'Detect HTTP DDoS
QString.QStringExceedLength(ClientIP, NewString, Qlength)
Else

Dim CheckPresentString As Integer = 0
For Each line As String In
File.ReadLines("C:\HTTPLogs\QueryString\Refer\QStringRelevantCheck.txt")
If line.Equals(NewString) Then
CheckPresentString = CheckPresentString + 1
End If
```

```
Next line
If CheckPresentString > 0 Then
'Clean HTTP Request | 'Forward Next Layer / Main Page
QString.QStringDetectRelevant(ClientIP, NewString)
Else
'Detect HTTP DDoS
QString.QStringDetectNotRelevant(ClientIP, NewString)
End If
End If

Else
'Detect HTTP DDoS
QString.QStringDetectSpecialCharacter(ClientIP, NewString)
End If

ElseIf Stringcheck = Regex.IsMatch(PresentString, "[`~!@#\$%\^\*\(\)_\-
\+\{\}\[\]\\\|:;""'<>,\.\/A-Z]") Then

If Qlength >= 4 Then
'Detect HTTP DDoS
QString.QStringExceedLength(ClientIP, PresentString, Qlength)
Else

Dim CheckPresentString As Integer = 0
For Each line As String In
File.ReadLines("C:\HTTPLogs\QueryString\Refer\QStringRelevantCheck.txt")
If line.Equals(PresentString) Then
CheckPresentString = CheckPresentString + 1
End If
Next line

If CheckPresentString > 0 Then
'Clean HTTP Request | 'Forward Next Layer / Main Page
QString.QStringDetectRelevant(ClientIP, PresentString)
```

```
Else
'Detect HTTP DDoS
QString.QStringDetectNotRelevant(ClientIP, PresentString)
End If
End If
Else
'Detect HTTP DDoS
QString.QStringDetectSpecialCharacter(ClientIP, PresentString)
End If
End If
End Sub
Imports System.IO
Public Class QString

Shared Sub QStringCountGETRequest(ByVal QStringCountGETRequest As Object,
ByVal ClientIP As Object)
Dim text As String = "First line" & Environment.NewLine
Dim path As String = "C:\HTTPLogs\QueryString\QStringCountGETRequest.txt"

Using WQAttack As StreamWriter = File.AppendText(path)
WQAttack.WriteLine("Query String GET Request : " & QStringCountGETRequest
& "-" & ClientIP)
End Using
End Sub

Shared Sub QStringNoQuery(ByVal ClientIP As Object)
Dim text As String = "First line" & Environment.NewLine
Dim path As String = "C:\HTTPLogs\QueryString\QStringNoQuery.txt"
Using WQAttack As StreamWriter = File.AppendText(path)
WQAttack.WriteLine("No Query : " & "-" & ClientIP)
End Using
End Sub
```

```
Shared Sub QStringExceedLength(ByVal ClientIP As Object, ByVal NewString As
Object, ByVal QStringExceedLength As Object)
Dim text As String = "First line" & Environment.NewLine
Dim path As String = "C:\HTTPLogs\QueryString\QStringExceedLength.txt"

Using WQAttack As StreamWriter = File.AppendText(path)
WQAttack.WriteLine("Query String Exceed Length : " & NewString & " Length : "
& QStringExceedLength & "-" & ClientIP)
End Using
End Sub

Shared Sub QStringDetectRelevant(ByVal ClientIP As Object, ByVal
QStringDetectRelevant As Object)
Dim text As String = "First line" & Environment.NewLine
Dim path As String = "C:\HTTPLogs\QueryString\QStringDetectRelevant.txt"
Using WQAttack As StreamWriter = File.AppendText(path)
WQAttack.WriteLine("Detect IP : " & ClientIP & " Query Detect Relevant : " &
QStringDetectRelevant)
End Using
End Sub

Shared Sub QStringDetectNotRelevant(ByVal ClientIP As Object, ByVal
QStringDetectNotRelevant As Object)
Dim text As String = "First line" & Environment.NewLine
Dim path As String = "C:\HTTPLogs\QueryString\QStringDetectNotRelevant.txt"
Using WQAttack As StreamWriter = File.AppendText(path)
WQAttack.WriteLine("Detect IP : " & ClientIP & " Query Detect Not Relevant : " &
QStringDetectNotRelevant)
End Using
End Sub
```

```
Shared Sub QStringDetectSpecialCharacter(ByVal QStringDetectSpecialCharacter
As Object, ByVal ClientIP As Object)
Dim text As String = "First line" & Environment.NewLine
Dim path As String =
"C:\HTTPLogs\QueryString\QStringDetectSpecialCharacter.txt"
Using WQAttack As StreamWriter = File.AppendText(path)
WQAttack.WriteLine("Detect Query String Special Character : " &
QStringDetectSpecialCharacter & "-" & ClientIP)
End Using
End Sub
End Class
```

APPENDIX H PROXY INSPECTION

```
Sub Proxy_Inspection()
Dim XF, XVIA As Object
Dim count As Integer = 0
Dim SourceConnection As Object = Request.ServerVariables("REMOTE_ADDR")
Dim ClientIPProxy As Object = Request.ServerVariables("REMOTE_ADDR")
XF = Context.Request.ServerVariables("HTTP_X_FORWARDED_FOR")
XVIA = Context.Request.ServerVariables("HTTP_VIA")

If
(Context.Request.ServerVariables.AllKeys.Contains("HTTP_X_FORWARDED_FO
R")) Or
(Context.Request.ServerVariables.AllKeys.Contains("HTTP_VIA")) Or
(Context.Request.ServerVariables.AllKeys.Contains("HTTP_X_PROXY_ID")) Or
(Context.Request.ServerVariables.AllKeys.Contains("HTTP_X_REAL_IP")) Or
(Context.Request.ServerVariables.AllKeys.Contains("HTTP_X_PINGBACK_FOR
WARDED_FOR")) Then

Dim Path_ProxyHeader As String =
"C:\HTTPLogs\Proxy\ProxyIP\ProxyWith_Header.txt"
Using sw As StreamWriter = File.AppendText(Path_ProxyHeader)
sw.WriteLine("Detect Proxy With Proxy Headers: " & ClientIPProxy)
End Using
InspectionHighRate_Proxy()
Else

Dim CheckProxy As Integer = 0
For Each line As String In
File.ReadLines("C:\HTTPLogs\Proxy\ProxyIP\ProxyIP_Database.txt")
If line.Equals(ClientIPProxy) Then
CheckProxy = CheckProxy + 1
```

```
End If
Next line
If CheckProxy > 0 Then

Dim Path_ProxyAbsent_Header As String =
"C:\HTTPLogs\Proxy\ProxyIP\ProxyWith_No_Header.txt"
Using sw As StreamWriter = File.AppendText(Path_ProxyAbsent_Header)
sw.WriteLine("Detect Proxy With Absent Proxy Headers: " & ClientIPProxy)
End Using
InspectionHighRate_Proxy()
Else
Dim Path_NoProxy As String = "C:\HTTPLogs\Proxy\ProxyIP\DetectNotProxy.txt"
Using sw As StreamWriter = File.AppendText(Path_NoProxy)
sw.WriteLine("Detect No Proxy: " & ClientIPProxy)
'No Proxy Detected
'Forward to Next Layer
End Using
End If
End If
End Sub

Sub InspectionHighRate_Proxy()
Dim HTTPRequestPrevious = DateTime.Today.AddDays(-
1).ToString("dd:MM:yyyy")
Dim CheckHTTPRequestDate As Integer = 0
For Each line As String In
File.ReadLines("C:\HTTPLogs\Proxy\ProxyIP\CountGetRequest_Proxy.txt")
If line.Contains(HTTPRequestPrevious) Then
CheckHTTPRequestDate = CheckHTTPRequestDate + 1
End If
Next line
If CheckHTTPRequestDate > 0 Then
File.WriteAllText("C:\HTTPLogs\Proxy\ProxyIP\CountGetRequest_Proxy.txt",
String.Empty.Trim)
```

```
End If
Dim HTTPRequestdate = DateTime.Now.ToString("dd:MM:yyyy")
Dim ProxyIP As Object = Request.ServerVariables("REMOTE_ADDR")
Dim StrPath = Request.Path
Dim StrPathReplace As String
StrPathReplace = StrPath.Replace("/", "")
Dim combine = HTTPRequestdate + "-" + ProxyIP + "-" + StrPathReplace

Dim Path_CountProxyRequest As String =
"C:\HTTPLogs\Proxy\ProxyIP\CountGetRequest_Proxy.txt"
Using sw As StreamWriter = File.AppendText(Path_CountProxyRequest)
sw.WriteLine(combine)
End Using

Dim input =
File.ReadAllText("C:\HTTPLogs\Proxy\ProxyIP\CountGetRequest_Proxy.txt")
Dim pattern1 = ProxyIP
Dim pattern2 = Request.Path
Dim pattern3 = HTTPRequestdate
Response.Write(combine)
Dim matches = Regex.Matches(input, combine)
Dim count = matches.Count
Response.Write("<br>" & count)

If count > 25 Then
CheckBrowser_Proxy()
Else
BelowThreshold.Proxy_BelowThresholdCount(combine)
End If
End Sub
Sub CheckBrowser_Proxy()
Dim width = hidwidth.Value
Dim heigh = hidheight.Value
Dim input = File.ReadAllText("C:\HTTPLogs\HighRequest\CountGETRequest.txt")
```

```
Dim ClientIP As Object = Request.ServerVariables("REMOTE_ADDR")
Dim HTTPRequestdate = DateTime.Now.ToString("dd:MM:yyyy")
Dim combine = HTTPRequestdate + "-" + ClientIP + "-" + Request.Path

If Not IsPostBack Then
Dim script As String = "window.onload = function() { SetHidValue(); };"
Page.ClientScript.RegisterClientScriptBlock(Me.GetType(), "setvalues", script,
True)
If hidheight.Value = String.Empty And hidwidth.Value = String.Empty Then
Dim text As String = "First line" & Environment.NewLine
Dim path As String = "C:\HTTPLogs\Proxy\ProxyIP\NoBrowser_Proxy.txt"
Using WQAttack As StreamWriter = File.AppendText(path)
WQAttack.WriteLine("Proxy : No Browser Detected:" & ClientIP)
End Using
End If
Else
If Not (hidheight.Value = String.Empty) And Not (hidwidth.Value = String.Empty)
Then
Response.Write("Width: " + hidwidth.Value + " Height: " + hidheight.Value)
Dim UA As String = Request.UserAgent
Dim text As String = "First line" & Environment.NewLine
Dim path As String = "C:\HTTPLogs\Proxy\ProxyIP\Browser_Proxy.txt"
Using WQAttack As StreamWriter = File.AppendText(path)
WQAttack.WriteLine("Proxy : Browser Detect :" & UA & "-" & ClientIP)
End Using
End If
End If
End Sub
```